CS112, CS212: Operating Systems

Spring 2025

Course Material

- Syllabus (sched/)
- Attendance check-in (CS212 only) (https://cs212.stanford.edu/cgi-bin/attendance.cgi)
- Lecture and section notes (notes/)
- Lab 0 (labs/lab0.html)
- Git setup guide (labs/git.html)
- Programming Projects (labs/project.html)
- Reference Materials (reference/)
- FAQ: CS112 vs. CS212 (which.html)
- Exam Archive (exams/index.html)

Meetings

Lecture: Mondays and Wednesdays, 1:30pm-2:50pm, Skilling Auditorium
Section: some Fridays (as indicated on syllabus (sched/)), 1:30pm-2:20pm, NVIDIA Auditorium
Office hours: First half of office hours will be round-robin public questions. Priority for second half will go to students who attended first half. Sign up on the OH Queue:
Office Hours Queue Link (https://queue.cs.stanford.edu/queues/2v6bft0TxQHbljcrnU1chp39VbD)

Staff

Staff list:
cs212-staff@scs.stanford.edu

Instructor: David Mazières (https://www.scs.stanford.edu/~dm/)
Office hours: Monday 3pm-4pm
Address: CoDa 328W (zoom (https://stanford.zoom.us/j/92180252368?pwd=9qUTwiOu0pXQ9M3xT1UPJwWj6SGkVd.1))

CA: Hari Vallabhaneni
Office hours: Monday, Wednesday 4pm-6pm
Address: CoDa B48 (zoom (https://stanford.zoom.us/j/98556886169?pwd=XUGAstUpzfeLy9Ta1KeTSflzfLG92q.1))

CA: Sam Do
Office hours: Tuesday, Thursday 1:30pm-3:30pm
Address: CoDa B42 (zoom (https://stanford.zoom.us/j/98074587887?pwd=Y9VO2rx6LEUL4b9msfBGEaNlKWTJxo.1))

CA: Poojan Pandya
Office hours: Wednesday, Friday 9:30am-11:30am
Address: Huang Basement (zoom (https://stanford.zoom.us/j/95487821623?pwd=8zmzpSixytwYwyNtE4onhgoMUW4zeF.1))

CA: Alice Liu
Office hours: Tuesday, Thursday 3:30pm-5:30pm
Address: CoDa B50 (zoom (https://stanford.zoom.us/j/97850789676?pwd=5q3cGblzXVr3vGbsyCU4euYwQyqrzP.1))

CA: June Lee
Office hours: Tuesday, Thursday 12:45pm-2:45pm
Address: CoDa B46 (zoom (https://stanford.zoom.us/j/93070536990?pwd=baQMLsahJUT0areXZPv8gExchJo0Yz.1))

Table of Contents

1. Introduction

CS212 – Operating Systems

Instructor: David Mazières

CAs: Sam Do, Poojan Pandya, and Hari Vallabhaneni

Stanford University

Outline

CS212 vs. CS112

- **CS212 (previously CS140) is a standalone OS class**
  - Lectures introduce OS topics, similar to CS111
  - Exams test you on material from lecture
  - Programming projects make ideas concrete in an instructional OS
- **CS112 is just the projects from CS212**
  - Only makes sense if you've previously taken CS111
  - Idea: projects in separate quarter from lectures allows more time
  - Feel free to attend any lectures if you want to review a topic (but most will be similar to CS111)
  - A few recommended lectures/sections marked in syllabus
- **In case there are still bugs in program sheets**
  - CS111 or CS212 should fulfill any OS breadth requirement
  - CS112 or CS212 should satisfy significant implementation
  - Ask for exception if something doesn't make sense

Lecture attendance

- **In-person lecture attendance expected for CS212 students**
  - Use phone or laptop logged into Stanford to check in, or jot down attendance code and check-in right after lecture
  - Please check-in over Stanford WiFi, not mobile network if possible
  - Exception: SCPD students (welcome to attend but not required), or Instructor gave you permission to be treated as an SCPD student
- **Don't just watch the videos if you are a non-SCPD student**
  - Grade is partly based on attendance
  - Saving videos until night before exam proven bad idea
- **Lectures will be available by zoom and recorded**
  - When practical, SCPD encouraged to join synchronously via zoom
  - Otherwise, videos will be on panopto

Administrivia

- **Class web page:** `http://cs212.scs.stanford.edu/`
  - All assignments, handouts, lecture notes on-line
- **Textbook: *Operating System Concepts, 8th Edition*, by Silberschatz, Galvin, and Gagne**
  - Out of print and highly optional (weening class from textbook)
- **Goal is to make lecture slides the primary reference**
  - Almost everything I talk about will be on slides
  - PDF slides contain links to further reading about topics
  - Please download slides from class web page
  - Will try to post before lecture for taking notes (but avoid calling out answers if you read them from slides)

Administrivia 2

- **Edstem is the main discussion forum**
- **Staff mailing list:** `cs212-staff@scs.stanford.edu`
- **CA split office hours, first round-robin, then individual/group**
  - Please ask non-private questions in RR portion
  - Priority for individual group will go to people who attended RR
- **Key dates:**
  - Lectures: MW 1:30pm–2:50pm
  - 6 sections starting this Friday (time, location TBD) 6th section (final review) is in Wednesday lecture slot
  - Midterm: Monday, May 5, in class (1:30pm-2:50pm)
  - Final: Monday, June 9, 3:30pm-6:30pm
  - No alternate exam arrangements (except OAE, SCPD), In-person attendance required for both midterm and final
  - SCPD can use exam monitor, return within 24 hours of exam start
- **Exams open note, but not open book (bring slide print-outs)**

Course topics

- **Threads & Processes**
- **Concurrency & Synchronization**
- **Scheduling**
- **Virtual Memory**
- **I/O**
- **Disks, File systems**
- **Protection & Security**
- **Virtual machines**
- **Note: Lectures will often take Unix as an example**
  - Most current and future OSes heavily influenced by Unix
  - Won't talk much about Windows

Course goals

- **Introduce you to operating system concepts**
  - Hard to use a computer without interacting with OS
  - Understanding the OS makes you a more effective programmer
- **Cover important systems concepts in general**
  - Caching, concurrency, memory management, I/O, protection
- **Teach you to deal with larger software systems**
  - Programming assignments much larger than many courses
  - Warning: Many people will consider course very hard
  - In past, majority of people report $\geq$15 hours/week
  - We hope it's more manageable with CS111 background and no lectures or exams
- **Prepare you to take graduate OS classes (CS240, 240[a-z])**

Programming Assignments

- **Implement parts of Pintos operating system**
  - Built for x86 hardware, you will use hardware emulators
- **One setup homework (lab 0) due this Friday**
- **Four two-week implementation projects:**
  - Threads
  - User processes
  - Virtual memory
  - File system
- **Lab 1 on web site, officially distributed Wednesday**
  - Attend section this Friday for project 1 overview
- **Implement projects in groups of up to 3 people**
  - CS112/CS212 mixed groups allowed
  - Disclose to partners if you are taking class pass/fail
  - Use "Forming Teams" category on edstem to meet people

Grading

- **No incompletes (talk to me ASAP if you have problems)**
- **50% of CS212 grade based on exams using this quantity:**
  resurrection $\leftarrow$ (**midterm** $>$ 0 && **missed** $\leq$ 7 **lectures**)

```
// final review section doesn't count as lecture
```

  $\max\left(\frac{1}{2}\left(\textbf{midterm} + \textbf{final}\right), \textbf{resurrection}\ ?\ \textbf{final} : 0\right)$
- **50% of CS212 grade, 100% of CS112 grade from projects**
  - For each project, 50% of score based on passing test cases
  - Remaining 50% based on design and style
- **Most people's projects pass most test cases**
  - Please, please, please turn in working code, or **no credit** here
- **Means design and style matter a lot**
  - Large software systems not just about producing working code
  - Need to produce code other people can understand
  - That's why we have group projects

Style

- **Must turn in a design document along with code**
  - We supply you with templates for each project's design doc
- **CAs will manually inspect code for correctness**
  - E.g., must actually implement the design
  - Must handle corner cases (e.g., handle `malloc` failure)
- **Will deduct points for error-prone code w/o errors**
  - Don't use global variables if automatic ones suffice
  - Don't use deceptive names for variables
- **Code must be easy to read**
  - Indent code, keep lines and (when possible) functions short
  - Use a uniform coding style (try to match existing code)
  - Put comments on structure members, globals, functions
  - Don't leave in reams of commented-out garbage code

Assignment requirements

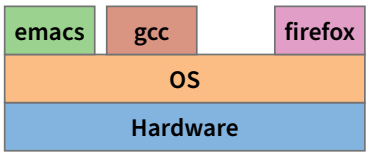
- **Do not look at other people's solutions to projects**
  - We reserve the right to run MOSS on present and past submissions
  - Do not publish your own solutions in violation of the honor code
  - That means using (public) github can get you in big trouble
- **You may read but not copy other OSes**
  - E.g., Linux, OpenBSD/FreeBSD, etc.
- **Cite any code that inspired your code**
  - As long as you cite what you used, it's not cheating
  - In worst case, we deduct points if it undermines the assignment
- **Projects due at start of class on due date**
  - Free extension to 5pm if you attend lecture or if all group members are in CS112
- **Ask `cs212-staff` for extension if you run into trouble**
  - Be sure to tell us: How much have you done? How much is left? When can you finish by?

Outline

1. Administrivia
2. Substance

What is an operating system?

- **Layer between applications and hardware**



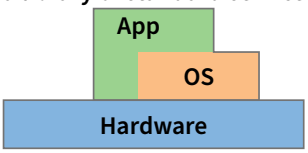
- **Makes hardware useful to the programmer**
- **[Usually] Provides abstractions for applications**
  - Manages and hides details of hardware
  - Accesses hardware through low/level interfaces unavailable to applications
- **[Often] Provides protection**
  - Prevents one process/user from clobbering another

Why study operating systems?

- **Operating systems are a mature field**
  - Most people use a handful of mature OSes
  - Hard to get people to switch operating systems
  - Hard to have impact with a new OS
- **Still open questions in operating systems**
  - Security – Hard to achieve security without a solid foundation
  - Scalability – How to adapt concepts when hardware scales 10× (fast networks, low service times, high core counts, big data…)
- **High-performance servers are an OS issue**
  - Face many of the same issues as OSes, sometimes bypass OS
- **Resource consumption is an OS issue**
  - Battery life, radio spectrum, etc.
- **New "smart" devices need new OSes**

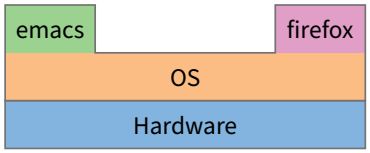
Primitive Operating Systems

- **Just a library of standard services [no protection]**



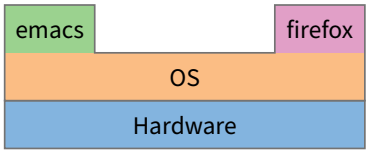
  - Standard interface above hardware-specific drivers, etc.
- **Simplifying assumptions**
  - System runs one program at a time
  - No bad users or programs (often bad assumption)
- **Problem: Poor utilization**
  - …of hardware (e.g., CPU idle while waiting for disk)
  - …of human user (must wait for each program to finish)

Multitasking



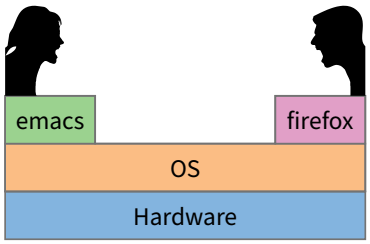
- **Idea: More than one process can be running at once**
  - When one process blocks (waiting for disk, network, user input, etc.) run another process
- **Problem: What can ill-behaved process do?**

Multitasking



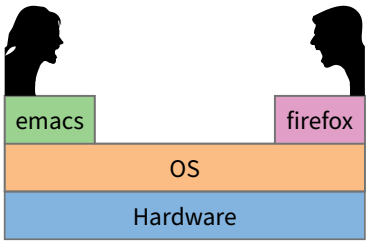
- **Idea: More than one process can be running at once**
  - When one process blocks (waiting for disk, network, user input, etc.) run another process
- **Problem: What can ill-behaved process do?**
  - Go into infinite loop and never relinquish CPU
  - Scribble over other processes' memory to make them fail
- **OS provides mechanisms to address these problems**
  - *Preemption* – take CPU away from looping process
  - *Memory protection* – protect processes' memory from one another

Multi-user OSes



- **Many OSes use *protection* to serve distrustful users/apps**
- **Idea: With *N* users, system not *N* times slower**
  - Users' demands for CPU, memory, etc. are bursty
  - Win by giving resources to users who actually need them
- **What can go wrong?**

Multi-user OSes

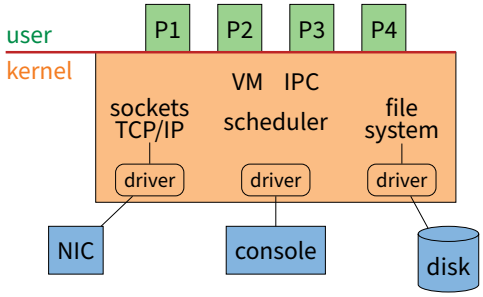


- **Many OSes use *protection* to serve distrustful users/apps**
- **Idea: With *N* users, system not *N* times slower**
  - Users' demands for CPU, memory, etc. are bursty
  - Win by giving resources to users who actually need them
- **What can go wrong?**
  - Users are gluttons, use too much CPU, etc. (need policies)
  - Total memory usage greater than machine's RAM (must virtualize)
  - Super-linear slowdown with increasing demand (thrashing)

Protection

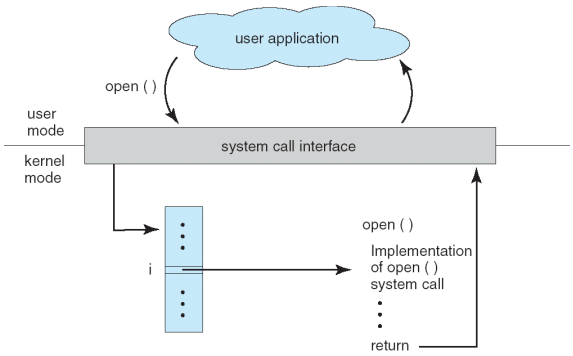
- **Mechanisms that isolate bad programs and people**
- **Pre-emption:**
  - Give application a resource, take it away if needed elsewhere
- **Interposition/mediation:**
  - Place OS between application and "stuff"
  - Track all pieces that application allowed to use (e.g., in table)
  - On every access, look in table to check that access legal
- **Privileged & unprivileged modes in CPUs:**
  - Applications unprivileged (unprivileged *user* mode)
  - OS privileged (privileged supervisor/*kernel* mode)
  - Protection operations can only be done in privileged mode

Typical OS structure



- **Most software runs as user-level processes (P[1-4])**
  - process ≈ instance of a program
- **OS *kernel* runs in *privileged* mode** (orange)
  - Creates/deletes processes
  - Provides access to hardware

System calls

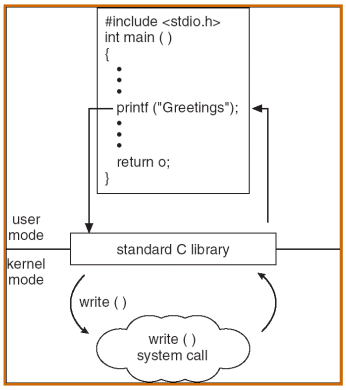


- **Applications can invoke kernel through *system calls***
  - Special instruction transfers control to kernel
  - . . . which dispatches to one of few hundred syscall handlers

System calls (continued)

- **Goal: Do things application can't do in unprivileged mode**
  - Like a library call, but into more privileged kernel code
- **Kernel supplies well-defined *system call* interface**
  - Applications set up syscall arguments and *trap* to kernel
  - Kernel performs operation and returns result
- **Higher-level functions built on syscall interface**
  - `printf`, `scanf`, `fgets`, etc. all user-level code
- **Example: POSIX/UNIX interface**
  - `open`, `close`, `read`, `write`, `...`

System call example



- **Standard library implemented in terms of syscalls**
  - *printf* – in libc, has same privileges as application
  - calls *write* – in kernel, which can send bits out serial port

UNIX file system calls

- **Applications "open" files (or devices) by name**
  - I/O happens through open files
- `int open(char *path, int flags, /*int mode*/...);`
  - `flags`: `O_RDONLY`, `O_WRONLY`, `O_RDWR`
  - `O_CREAT`: create the file if non-existent
  - `O_EXCL`: (w. `O_CREAT`) create if file exists already
  - `O_TRUNC`: Truncate the file
  - `O_APPEND`: Start writing from end of file
  - `mode`: final argument with `O_CREAT`
- **Returns file descriptor—used for all I/O to file**

Error returns

- **What if `open` fails? Returns -1 (invalid fd)**
- **Most system calls return -1 on failure**
  - Specific kind of error in global int `errno`
  - In retrospect, bad design decision for threads/modularity
- `#include <sys/errno.h>` **for possible values**
  - 2 = `ENOENT` "No such file or directory"
  - 13 = `EACCES` "Permission Denied"
- `perror` **function prints human-readable message**
  - `perror ("initfile");`
    → "`initfile: No such file or directory`"

Operations on file descriptors

- `int read (int fd, void *buf, int nbytes);`
  - Returns number of bytes read
  - Returns 0 bytes at end of file, or -1 on error
- `int write (int fd, const void *buf, int nbytes);`
  - Returns number of bytes written, -1 on error
- `off_t lseek (int fd, off_t pos, int whence);`
  - `whence`: 0 – start, 1 – current, 2 – end
    - Returns previous file offset, or -1 on error
- `int close (int fd);`

File descriptor numbers

- **File descriptors are inherited by processes**
  - When one process spawns another, same fds by default
- **Descriptors 0, 1, and 2 have special meaning**
  - 0 – "standard input" (`stdin` in ANSI C)
  - 1 – "standard output" (`stdout`, `printf` in ANSI C)
  - 2 – "standard error" (`stderr`, `perror` in ANSI C)
  - Normally all three attached to terminal
- **Example:** `type.c`
  - Prints the contents of a file to `stdout`

`type.c`

```
void
typefile (char *filename)
{
  int fd, nread;
  char buf[1024];

  fd = open (filename, O_RDONLY);
  if (fd == -1) {
    perror (filename);
    return;
  }

  while ((nread = read (fd, buf, sizeof (buf))) > 0)
    write (1, buf, nread);

  close (fd);
}
```

- **Can see system calls using strace utility (ktrace on BSD)**

Protection example: CPU preemption

- **Protection mechanism to prevent monopolizing CPU**
- **E.g., kernel programs timer to interrupt every 10 ms**
  - Must be in supervisor mode to write appropriate I/O registers
  - User code cannot re-program interval timer
- **Kernel sets interrupt to vector back to kernel**
  - Regains control whenever interval timer fires
  - Gives CPU to another process if someone else needs it
  - Note: must be in supervisor mode to set interrupt entry points
  - No way for user code to hijack interrupt handler
- **Result: Cannot monopolize CPU with infinite loop**
  - At worst get $1/N$ of CPU with N CPU-hungry processes

Protection is not security

- **How *can* you monopolize CPU?**

Protection is not security

- **How *can* you monopolize CPU?**
- **Use multiple processes**
- **For many years, could wedge most OSes with**

```
int main() { while(1) fork(); }
```

  - Keeps creating more processes until system out of proc. slots
- **Other techniques: use all memory (`chill` program)**
- **Typically solved with technical/social combination**
  - Technical solution: Limit processes per user
  - Social: Reboot and yell at annoying users
  - Social: Ban harmful apps from play store

Address translation

- **Protect memory of one program from actions of another**
- **Definitions**
  - *Address space*: all memory locations a program can name
  - *Virtual address*: addresses in process' address space
  - *Physical address*: address of real memory
  - *Translation*: map virtual to physical addresses
- **Translation done on every load, store, and instruction fetch**
  - Modern CPUs do this in hardware for speed
- **Idea: If you can't name it, you can't touch it**
  - Ensure one process's translations don't include any other process's memory

More memory protection

- **CPU allows kernel-only virtual addresses**
  - Kernel typically part of all address spaces, e.g., to handle system call in same address space
  - But must ensure apps can't touch kernel memory
- **CPU lets OS disable (invalidate) particular virtual addresses**
  - Catch and halt buggy program that makes wild accesses
  - Make virtual memory seem bigger than physical (e.g., bring a page in from disk only when accessed)
- **CPU enforced read-only virtual addresses useful**
  - E.g., allows sharing of code pages between processes
  - Plus many other optimizations
- **CPU enforced execute disable of VAs**
  - Makes certain code injection attacks harder

Different system contexts

- **At any point, a CPU (core) is in one of several contexts**
- ***User-level* – CPU in user mode running application**
- **Kernel process context – i.e., running kernel code on behalf of a particular process**
  - E.g., performing system call, handling exception (memory fault, numeric exception, etc.)
  - Or executing a kernel-only process (e.g., network file server)
- **Kernel code not associated with a process**
  - Timer interrupt (hardclock)
  - Device interrupt
  - "Softirqs", "Tasklets" (Linux-specific terms)
- **Context switch code – change which process is running**
  - Requires changing the current address space
- **Idle – nothing to do (bzero pages, put CPU in low-power state)**

Transitions between contexts

- **User → kernel process context: syscall, page fault, . . .**
- **User/process context → interrupt handler: hardware**
- **Process context → user/context switch: return**
- **Process context → context switch: sleep**
- **Context switch → user/process context**

Resource allocation & performance

- **Multitasking permits higher resource utilization**
- **Simple example:**
  - Process downloading large file mostly waits for network
  - You play a game while downloading the file
  - Higher CPU utilization than if just downloading
- **Complexity arises with cost of switching**
- **Example: Say disk 1,000 times slower than memory**
  - 1 GiB memory in machine
  - 2 Processes want to run, each use 1 GiB
  - Can switch processes by swapping them out to disk
  - Faster to run one at a time than keep context switching

Useful properties to exploit

- **Skew**
  - 80% of time taken by 20% of code
  - 10% of memory absorbs 90% of references
  - Basis behind cache: place 10% in fast memory, 90% in slow, usually looks like one big fast memory
- **Past predicts future (a.k.a. temporal locality)**
  - What's the best cache entry to replace?
  - If past ≈ future, then least-recently-used entry
- **Note conflict between fairness & throughput**
  - Higher throughput (fewer cache misses, etc.) to keep running same process
  - But fairness says should periodically preempt CPU and give it to next process

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void
typefile (char *filename)
{
  int fd, nread;
  char buf[1024];

  fd = open (filename, O_RDONLY);
  if (fd == -1) {
    perror (filename);
    return;
  }

  while ((nread = read (fd, buf, sizeof (buf))) > 0)
    write (1, buf, nread);

  close (fd);
}

int
main (int argc, char **argv)
{
  int argno;
  for (argno = 1; argno < argc; argno++)
    typefile (argv[argno]);
  exit (0);
}
```

2. Processes & Threads

Administrivia

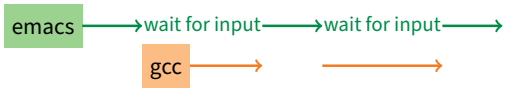
- **Friday 1:30pm section in NVIDIA auditorium (same zoom link)**
  - Please attend first section this Friday to learn about project 1
- **Project 1 due Wednesday, April 16 at 1:30pm (in 2 weeks)**
  - 5pm if you attend/watch lecture
- **Ask `cs212-staff` for extension if you can't finish**
  - Tell us where you are with the project,
  - How much more you need to do, and
  - How much longer you need to finish
- **No credit for late assignments w/o extension**

Processes

- **A *process* is an instance of a program running**
- **Modern OSes run multiple processes simultaneously**
- **Examples (can all run simultaneously):**
  - `gcc file_A.c` – compiler running on file A
  - `gcc file_B.c` – compiler running on file B
  - `emacs` – text editor
  - `firefox` – web browser
- **Non-examples (implemented as one process):**
  - Multiple emacs frames or firefox windows (can be one process)
- **Why processes?**
  - Simplicity of programming
  - Speed: Higher throughput, lower latency

Speed

- **Multiple processes can increase CPU utilization & throughput**
  - Overlap one process's computation with another's wait



- **Multiple processes can reduce latency**
  - Running *A* then *B* requires 100 sec for *B* to complete

  - Running *A* and *B* concurrently makes *B* finish faster

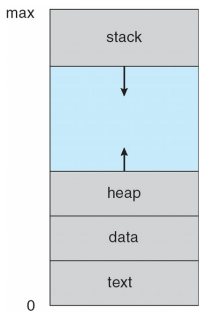
  - *A* is slower than if it had whole machine to itself, but still $<$ 100 sec unless both *A* and *B* completely CPU-bound

Processes in the real world

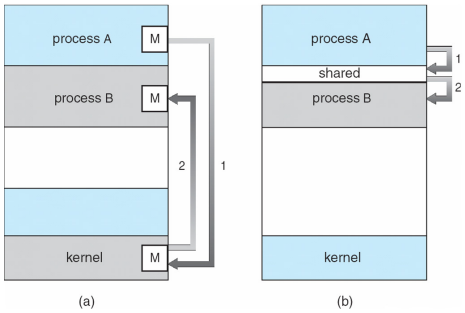
- **Processes and parallelism have been a fact of life much longer than OSes have been around**
  - E.g., say takes 1 worker 10 months to make 1 widget
  - Company may hire 100 workers to make 100 widgets
  - Latency for first widget $\gg 1/10$ month
  - Throughput may be $<$ 10 widgets per month (if can't perfectly parallelize task)
  - Or 100 workers making 10,000 widgets may achieve $>$ 10 widgets/month (e.g., if workers never idly wait for paint to dry)
- **You will see these effects in you Pintos project group**
  - May block waiting for partner to complete task
  - Takes time to coordinate/explain/understand one another's code
  - Labs will take $> 1/3$ time with three people
  - But you will graduate faster than if you took only 1 class at a time

A process's view of the world

- **Each process has own view of machine**
  - Its own address space – `*(char *)0xc000` different in P_1 & P_2
  - Its own open files
  - Its own virtual CPU (through preemptive multitasking)
- **Simplifies programming model**
  - `gcc` does not care that `firefox` is running
- **Sometimes want interaction between processes**
  - Simplest is through files: `emacs` edits file, `gcc` compiles it
  - More complicated: Shell/command, Window manager/app.



Inter-Process Communication



- **How can processes interact in real time?**
  - (a) By passing messages through the kernel
  - (b) By sharing a region of physical memory
  - (c) Through asynchronous signals or alerts

Outline

1. (UNIX-centric) User view of processes
2. Kernel view of processes
3. Threads
4. Thread implementation details

Creating processes

- **Original UNIX paper is a great reference on core system calls**
- `int fork (void);`
  - Create new process that is exact copy of current one
  - Returns *process ID* of new process in "parent"
  - Returns 0 in "child"
- `int waitpid (int pid, int *stat, int opt);`
  - `pid` – process to wait for, or -1 for any
  - `stat` – will contain exit value, or signal
  - `opt` – usually 0 or `WNOHANG`
  - Returns process ID or -1 on error

Deleting processes

- `void exit (int status);`
  - Current process ceases to exist
  - `status` shows up in `waitpid` (shifted)
  - By convention, `status` of 0 is success, non-zero error
- `int kill (int pid, int sig);`
  - Sends signal `sig` to process `pid`
  - `SIGTERM` most common value, kills process by default (but application can catch it for "cleanup")
  - `SIGKILL` stronger, kills process always

Running programs

- `int execve (char *prog, char **argv, char **envp);`
  - `prog` – full pathname of program to run
  - `argv` – argument vector that gets passed to `main` (ending `NULL`)
  - `envp` – environment variables, e.g., `PATH`, `HOME` (ending `NULL`)
  - Replaces current process state with new instance of `prog`
- **Generally called through a wrapper functions**
  - `int execvp (char *prog, char **argv);`
    Search `PATH` for prog, use current environment
  - `int execlp (char *prog, char *arg, ...);`
    List arguments one at a time, finish with `NULL`
- **Example:** `minish.c`
  - Loop that reads a command, then executes it
- **Warning: Pintos `exec` more like combined fork/exec**

`minish.c` (simplified)

```
pid_t pid; char **av;
void doexec () {
  execvp (av[0], av);
  perror (av[0]);
  exit (1);
}

    /* ... main loop: */
    for (;;) {
      parse_next_line_of_input (&av, stdin);
      switch (pid = fork ()) {
      case -1:
        perror ("fork"); break;
      case 0:
        doexec ();
      default:
        waitpid (pid, NULL, 0); break;
      }
    }
```


Manipulating file descriptors

- `int dup2 (int oldfd, int newfd);`
  - Closes `newfd`, if it was a valid descriptor
  - Makes `newfd` an exact copy of `oldfd`
  - Two file descriptors will share same offset (`lseek` on one will affect both)
- `int fcntl (int fd, int cmd, ...)` **– misc fd configuration**
  - `fcntl (fd, F_SETFD, val)` – sets close-on-exec flag
    When `val`$\neq 0$, `fd` not inherited by spawned programs
  - `fcntl (fd, F_GETFL)` – get misc fd flags
  - `fcntl (fd, F_SETFL, val)` – set misc fd flags
- **Example:** `redirsh.c`
  - Loop that reads a command and executes it
  - Recognizes `command < input > output 2> errlog`

redirsh.c

```
void doexec (void) {
  int fd;
  if (infile) {     /* non-NULL for "command < infile" */
    if ((fd = open (infile, O_RDONLY)) < 0) {
      perror (infile);
      exit (1);
    }
    if (fd != 0) {
      dup2 (fd, 0);
      close (fd);
    }
  }

  /* ... do same for outfile→fd 1, errfile→fd 2 ... */

  execvp (av[0], av);
  perror (av[0]);
  exit (1);
}
```


Pipes

- `int pipe (int fds[2]);`
  - Returns two file descriptors in `fds[0]` and `fds[1]`
  - Data written to `fds[1]` will be returned by `read` on `fds[0]`
  - When last copy of `fds[1]` closed, `fds[0]` will return EOF
  - Returns 0 on success, -1 on error
- **Operations on pipes**
  - `read`/`write`/`close` – as with files
  - When `fds[1]` closed, `read(fds[0])` returns 0 bytes
  - When `fds[0]` closed, `write(fds[1])`:
    - Kills process with `SIGPIPE`
    - Or if signal ignored, fails with EPIPE
- **Example:** `pipesh.c`
  - Sets up pipeline `command1 | command2 | command3 ...`

pipesh.c (simplified)

```
void doexec (void) {
  while (outcmd) {
    int pipefds[2]; pipe (pipefds);
    switch (fork ()) {
    case -1:
      perror ("fork"); exit (1);
    case 0:
      dup2 (pipefds[1], 1);
      close (pipefds[0]); close (pipefds[1]);
      outcmd = NULL;
      break;
    default:
      dup2 (pipefds[0], 0);
      close (pipefds[0]); close (pipefds[1]);
      parse_command_line (&av, &outcmd, outcmd);
      break;
    }
  }
  ⋮
```


Multiple file descriptors

- **What if you have multiple pipes to multiple processes?**
- `poll` **system call lets you know which fd you can read/write**[1]

```
typedef struct pollfd {
  int fd;
  short events;  // OR of POLLIN, POLLOUT, POLLERR, ...
  short revents; // ready events returned by kernel
};
int poll(struct pollfd *pfds, int nfds, int timeout);
```

- **Also put pipes/sockets into *non-blocking* mode**

```
if ((n = fcntl (s.fd_, F_GETFL)) == -1
    || fcntl (s.fd_, F_SETFL, n | O_NONBLOCK) == -1)
  perror("O_NONBLOCK");
```

  - Returns errno `EGAIN` instead of waiting for data
  - Does not work for normal files (see aio for that)

[1] In practice, more efficient to use `epoll` on linux or `kqueue` on *BSD

Why fork?

- **Most calls to `fork` followed by `execve`**
- **Could also combine into one *spawn* system call (like Pintos `exec`)**
- **Occasionally useful to fork one process**
  - Unix *dump* utility backs up file system to tape
  - If tape fills up, must restart at some logical point
  - Implemented by forking to revert to old state if tape ends
- **Real win is simplicity of interface**
  - Tons of things you might want to do to child: Manipulate file descriptors, alter namespace, manipulate process limits ...
  - Yet `fork` requires *no* arguments at all

Examples

- `login` **– checks username/password, runs user shell**
  - Runs with administrative privileges
  - Lowers privileges to user before exec'ing shell
  - Note doesn't need `fork` to run shell, just `execve`
- `chroot` **– change root directory**
  - Useful for creating/debugging different OS image in a subdirectory
- **Some more linux-specific examples**
  - `systemd-nspawn` – runs program in container-like environment
  - `ip netns` – runs program with different network namespace
  - `unshare` – decouple namespaces from parent and exec program

Spawning a process without fork

- **Without fork, needs tons of different options for new process**
- **Example: Windows `CreateProcess` system call**
  - Also `CreateProcessAsUser`, `CreateProcessWithLogonW`, `CreateProcessWithTokenW`, ...

```
BOOL WINAPI CreateProcess(
  _In_opt_      LPCTSTR lpApplicationName,
  _Inout_opt_   LPTSTR lpCommandLine,
  _In_opt_      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  _In_opt_      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  _In_          BOOL bInheritHandles,
  _In_          DWORD dwCreationFlags,
  _In_opt_      LPVOID lpEnvironment,
  _In_opt_      LPCTSTR lpCurrentDirectory,
  _In_          LPSTARTUPINFO lpStartupInfo,
  _Out_         LPPROCESS_INFORMATION lpProcessInformation
);
```


Outline

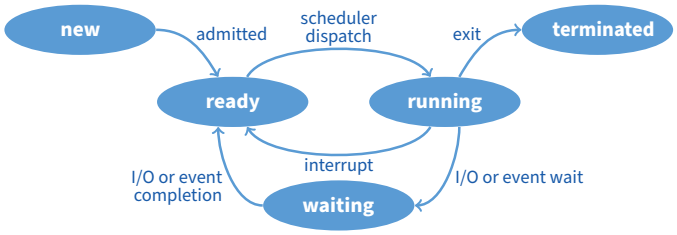
Implementing processes

- **Keep a data structure for each process**
  - Process Control Block (PCB)
  - Called `proc` in Unix, `task_struct` in Linux, and just `struct thread` in Pintos
- **Tracks *state* of the process**
  - Running, ready (runnable), waiting, etc.
- **Includes information necessary to run**
  - Registers, virtual memory mappings, etc.
  - Open files (including memory mapped files)
- **Various other data about the process**
  - Credentials (user/group ID), signal mask, controlling terminal, priority, accounting statistics, whether being debugged, which system call binary emulation in use, ...

Process state
Process ID
User id, etc.
Program counter
Registers
Address space (VM data structs)
Open files

PCB

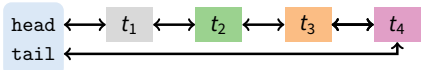
Process states



- **Process can be in one of several states**
  - *new* & *terminated* at beginning & end of life
  - *running* – currently executing (or will execute on kernel return)
  - *ready* – can run, but kernel has chosen different process to run
  - *waiting* – needs async event (e.g., disk operation) to proceed
- **Which process should kernel run?**
  - if 0 runnable, run idle loop (or halt CPU), if 1 runnable, run it
  - if >1 runnable, must make scheduling decision

Scheduling

- **How to pick which process to run**
- **Scan process table for first runnable?**
  - Expensive. Weird priorities (small pids do better)
  - Divide into runnable and blocked processes
- **FIFO?**
  - Put threads on back of list, pull them from front:



  - Pintos does this—see `ready_list` in `thread.c`
- **Priority?**
  - Give some threads a better shot at the CPU

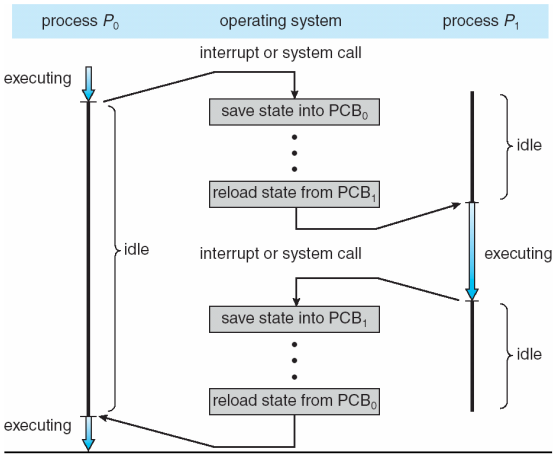
Scheduling policy

- **Want to balance multiple goals**
  - *Fairness* – don't starve processes
  - *Priority* – reflect relative importance of procs
  - *Deadlines* – must do *X* (play audio) by certain time
  - *Throughput* – want good overall performance
  - *Efficiency* – minimize overhead of scheduler itself
- **No universal policy**
  - Many variables, can't optimize for all
  - Conflicting goals (e.g., throughput or priority vs. fairness)
- **We will spend a whole lecture on this topic**

Preemption

- **Can preempt a process when kernel gets control**
- **Running process can vector control to kernel**
  - System call, page fault, illegal instruction, etc.
  - May put current process to sleep—e.g., read from disk
  - May make other process runnable—e.g., fork, write to pipe
- **Periodic timer interrupt**
  - If running process used up quantum, schedule another
- **Device interrupt**
  - Disk request completed, or packet arrived on network
  - Previously waiting process becomes runnable
  - Schedule if higher priority than current running proc.
- **Changing running process is called a *context switch***

Context switch

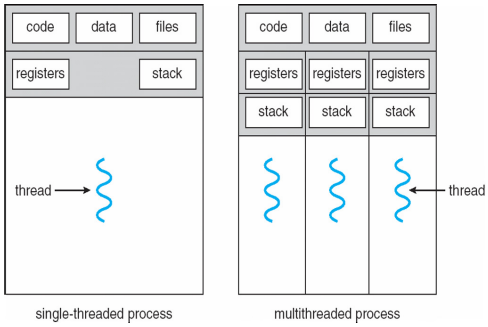


Context switch details

- **Very machine dependent. Typical things include:**
  - Save program counter and integer registers (always)
  - Save floating point or other special registers
  - Save condition codes
  - Change virtual address translations
- **Non-negligible cost**
  - Save/restore floating point registers expensive
    - Optimization: only save if process used floating point
  - May require flushing TLB (memory translation hardware)
    - HW Optimization 1: don't flush kernel's own data from TLB
    - HW Optimization 2: use tag to avoid flushing any data
  - Usually causes more cache misses (switch working sets)

Outline

Threads



- **A thread is a schedulable execution context**
  - Program counter, stack, registers, …
- **Simple programs use one thread per process**
- **But can also have multi-threaded programs**
  - Multiple threads running in same process's address space

Why threads?

- **Most popular abstraction for concurrency**
  - Lighter-weight abstraction than processes
  - All threads in one process share memory, file descriptors, etc.
- **Allows one process to use multiple CPUs or cores**
- **Allows program to overlap I/O and computation**
  - Same benefit as OS running `emacs` & `gcc` simultaneously
  - E.g., threaded web server services clients simultaneously:

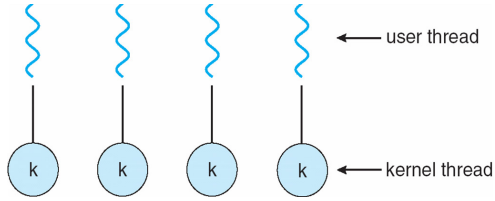
```
for (;;) {
  c = accept_client();
  thread_create(service_client, c);
}
```

- **Most kernels have threads, too**
  - Typically at least one kernel thread for every process
  - Switch kernel threads when preempting process

Thread package API

- `tid thread_create (void (*fn) (void *), void *arg);`
  - Create a new thread, run `fn` with `arg`
- `void thread_exit ();`
  - Destroy current thread
- `void thread_join (tid thread);`
  - Wait for thread `thread` to exit
- **Plus lots of support for synchronization [in 3 weeks]**
- **See [Birell] for good introduction**
- **Can have preemptive or non-preemptive threads**
  - Preemptive causes more race conditions
  - Non-preemptive can't take advantage of multiple CPUs
  - Before prevalence of multicore, most kernels non-preemptive

Kernel threads[2]



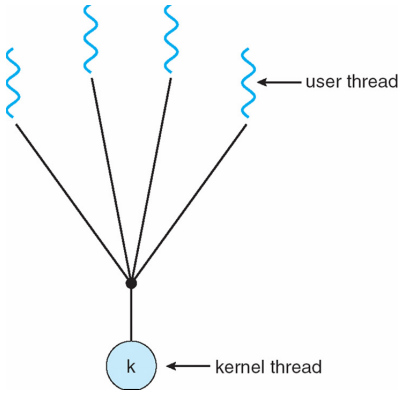
- **Can implement `thread_create` as a system call**
- **To add `thread_create` to an OS that doesn't have it:**
  - Start with process abstraction in kernel
  - `thread_create` like process creation with features stripped out
    - Keep same address space, file table, etc., in new process
    - `rfork`/`clone` syscalls actually allow individual control
- **Faster than a process, but still very heavy weight**

[2] i.e., *native* or non-green threads; "kernel threads" can also mean threads inside the kernel, which typically implement native threads)

Limitations of kernel-level threads

- **Every thread operation must go through kernel**
  - create, exit, join, synchronize, or switch for any reason
  - On my laptop: syscall takes 100 cycles, fn call 5 cycles
  - Result: threads 10x-30x slower when implemented in kernel
- **One-size fits all thread implementation**
  - Kernel threads must please all people
  - Maybe pay for fancy features (priority, etc.) you don't need
- **General heavy-weight memory requirements**
  - E.g., requires a fixed-size stack within kernel
  - Other data structures designed for heavier-weight processes

Alternative: User threads



- **Implement as user-level library (a.k.a. *green* threads)**
  - One kernel thread per process
  - `thread_create`, `thread_exit`, etc., just library functions

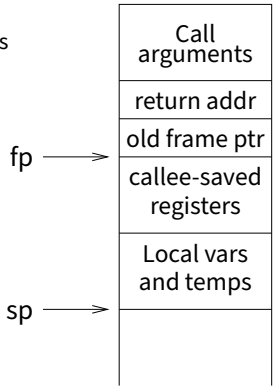
Implementing user-level threads

- **Allocate a new stack for each `thread_create`**
- **Keep a queue of runnable threads**
- **Replace networking system calls (`read`/`write`/etc.)**
  - If operation would block, switch and run different thread
- **Schedule periodic timer signal (`setitimer`)**
  - Switch to another thread on timer signals (preemption)
- **Multi-threaded web server example**
  - Thread calls `read` to get data from remote web browser
  - "Fake" `read` *function* makes `read` *syscall* in non-blocking mode
  - No data? schedule another thread
  - On timer or when idle, check which connections have new data

Outline

Background: calling conventions

- **Registers divided into 2 groups**
  - Functions free to clobber *caller-saved* regs (`%eax` [return val], `%edx`, & `%ecx` on x86)
  - But must restore *callee-saved* ones to original value upon return (on x86, `%ebx`, `%esi`, `%edi`, plus `%ebp` and `%esp`)
- ***sp* register always base of stack**
  - Frame pointer (*fp*) is old *sp*
- **Local variables stored in registers and on stack**
- **Function arguments go in caller-saved regs and on stack**
  - With 32-bit x86, all arguments on stack



Background: procedure calls

Procedure call

save active caller registers
push arguments to stack
call `foo` (pushes pc)

save needed callee registers
…do stuff…
restore callee saved registers
jump back to calling function

restore stack+caller regs.

- **Caller must save some state across function call**
  - Return address, caller-saved registers
- **Other state does not need to be saved**
  - Callee-saved regs, global variables, stack pointer

Pintos thread implementation

- **Pintos implements user processes on top of its own threads**
  - Code for threads in kernel very similar to green threads
- **Per-thread state in thread control block structure**

```
struct thread {
    ...
    uint8_t *stack;  /* Saved stack pointer. */
    ...
};
uint32_t thread_stack_ofs = offsetof(struct thread, stack);
```

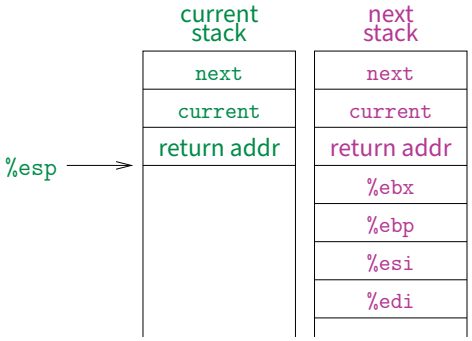
- **C declaration for asm thread-switch function:**
  - `struct thread *switch_threads (struct thread *cur, struct thread *next);`
- **Also thread initialization function to create new stack:**
  - `void thread_create (const char *name, thread_func *function, void *aux);`

i386 `switch_threads`

```
pushl %ebx; pushl %ebp          # Save callee-saved regs
pushl %esi; pushl %edi

mov thread_stack_ofs, %edx      # %edx = offset of stack field
                                #        in thread struct

movl 20(%esp), %eax             # %eax = cur
movl %esp, (%eax,%edx,1)        # cur->stack = %esp

movl 24(%esp), %ecx             # %ecx = next
movl (%ecx,%edx,1), %esp        # %esp = next->stack

popl %edi; popl %esi            # Restore calle-saved regs
popl %ebp; popl %ebx

ret                             # Resume execution
```

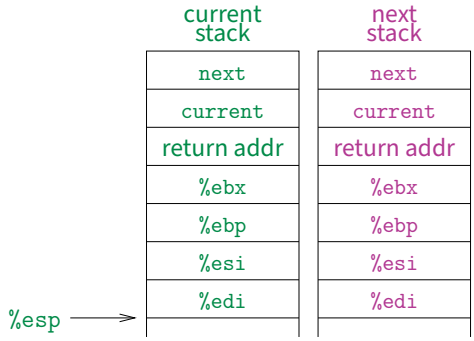
- **This is actual code from Pintos `switch.S` (slightly reformatted)**
  - See Thread Switching in documentation

i386 `switch_threads`



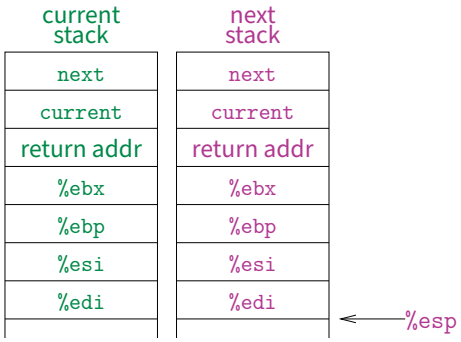
- **This is actual code from Pintos `switch.S` (slightly reformatted)**
  - See Thread Switching in documentation

i386 `switch_threads`



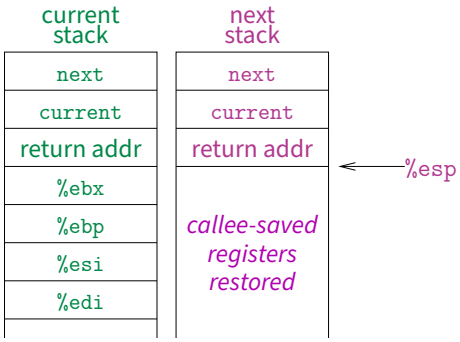
- **This is actual code from Pintos `switch.S` (slightly reformatted)**
  - See Thread Switching in documentation

i386 `switch_threads`



- **This is actual code from Pintos `switch.S` (slightly reformatted)**
  - See Thread Switching in documentation

i386 `switch_threads`

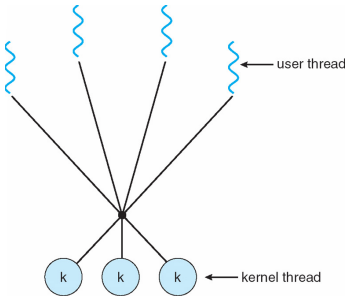


- **This is actual code from Pintos `switch.S` (slightly reformatted)**
  - See Thread Switching in documentation

Limitations of user-level threads

- **A user-level thread library can do the same thing as Pintos**
- **Can't take advantage of multiple CPUs or cores**
- **A blocking system call blocks all threads**
  - Can use `O_NONBLOCK` to avoid blocking on network connections
  - But doesn't work for disk (e.g., even aio doesn't work for metadata)
  - So one uncached disk read/synchronous write blocks all threads
- **A page fault blocks all threads**
- **Possible deadlock if one thread blocks on another**
  - May block entire process and make no progress
  - [More on deadlock in future lectures.]

User threads on kernel threads



- **User threads implemented on kernel threads**
  - Multiple kernel-level threads per process
  - `thread_create`, `thread_exit` still library functions as before
- **Sometimes called $n : m$ threading**
  - Have n user threads per m kernel threads
    (Simple user-level threads are $n : 1$, kernel threads $1 : 1$)

Limitations of $n : m$ threading

- **Many of same problems as $n : 1$ threads**
  - Blocked threads, deadlock, …
- **Hard to keep same # ktrheads as available CPUs**
  - Kernel knows how many CPUs available
  - Kernel knows which kernel-level threads are blocked
  - But tries to hide these things from applications for transparency
  - So user-level thread scheduler might think a thread is running while underlying kernel thread is blocked
- **Kernel doesn't know relative importance of threads**
  - Might preempt kthread in which library holds important lock

Lessons

- **Threads best implemented as a library**
  - But kernel threads not best interface on which to do this
- **Better kernel interfaces have been suggested**
  - See Scheduler Activations [Anderson et al.]
  - Maybe too complex to implement on existing OSes (some have added then removed such features)
- **Standard threads still fine for most purposes**
  - Use kernel threads if I/O concurrency main goal
  - Use $n : m$ threads for highly concurrent (e.g,. scientific applications) with many thread switches
- **But concurrency greatly increases complexity**
  - More on that in concurrency, synchronization lectures…

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

char **av;
int avsize;

void
avreserve (int n)
{
  int oldavsize = avsize;

  if (avsize > n + 1)
    return;

  avsize = 2 * (oldavsize + 1);
  if (avsize <= n)
    avsize = n + 1;
  av = realloc (av, avsize * sizeof (*av));
  while (oldavsize < avsize)
    av[oldavsize++] = NULL;
}

void
parseline (char *line)
{
  char *a;
  int n;

  for (n = 0; n < avsize; n++)
    av[n] = NULL;

  a = strtok (line, " \t\r\n");
  for (n = 0; a; n++) {
    avreserve (n);
    av[n] = a;
    a = strtok (NULL, " \t\r\n");
  }
}

void
doexec (void)
{
  execvp (av[0], av);
  perror (av[0]);
  exit (1);
}

int
main (void)
{
  char buf[512];
  char *line;
  int pid;

  avreserve (10);

  for (;;) {
    write (2, "$ ", 2);
    if (!(line = fgets (buf, sizeof (buf), stdin))) {
      write (2, "EOF\n", 4);
```

```
      exit (0);
    }
    parseline (line);
    if (!av[0])
      continue;

    switch (pid = fork ()) {
    case -1:
      perror ("fork");
      break;
    case 0:
      doexec ();
      break;
    default:
      waitpid (pid, NULL, 0);
      break;
    }
  }
}
```

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

char **av;
char *infile;
char *outfile;
char *errfile;
int avsize;

void
avreserve (int n)
{
  int oldavsize = avsize;

  if (avsize > n + 1)
    return;

  avsize = 2 * (oldavsize + 1);
  if (avsize <= n)
    avsize = n + 1;
  av = realloc (av, avsize * sizeof (*av));
  while (oldavsize < avsize)
    av[oldavsize++] = NULL;
}

void
parseline (char *line)
{
  char *a;
  int n;

  infile = outfile = errfile = NULL;
  for (n = 0; n < avsize; n++)
    av[n] = NULL;

  a = strtok (line, " \t\r\n");
  for (n = 0; a; n++) {
    if (a[0] == '<')
      infile = a[1] ? a + 1 : strtok (NULL, " \t\r\n");
    else if (a[0] == '>')
      outfile = a[1] ? a + 1 : strtok (NULL, " \t\r\n");
    else if (a[0] == '2' && a[1] == '>')
      errfile = a[2] ? a + 2 : strtok (NULL, " \t\r\n");
    else {
      avreserve (n);
      av[n] = a;
    }
    a = strtok (NULL, " \t\r\n");
  }
}

void
doexec (void)
{
  int fd;

  if (infile) {
    if ((fd = open (infile, O_RDONLY)) < 0) {
      perror (infile);
      exit (1);
```

```
    }
    if (fd != 0) {
      dup2 (fd, 0);
      close (fd);
    }
  }

  if (outfile) {
    if ((fd = open (outfile, O_WRONLY|O_CREAT|O_TRUNC, 0666)) < 0) {
      perror (outfile);
      exit (1);
    }
    if (fd != 1) {
      dup2 (fd, 1);
      close (fd);
    }
  }

  if (errfile) {
    if ((fd = open (errfile, O_WRONLY|O_CREAT|O_TRUNC, 0666)) < 0) {
      perror (outfile);
      exit (1);
    }
    if (fd != 2) {
      dup2 (fd, 2);
      close (fd);
    }
  }

  execvp (av[0], av);
  perror (av[0]);
  exit (1);
}

int
main (void)
{
  char buf[512];
  char *line;
  int pid;

  avreserve (10);

  for (;;) {
    write (2, "$ ", 2);
    if (!(line = fgets (buf, sizeof (buf), stdin))) {
      write (2, "EOF\n", 4);
      exit (0);
    }
    parseline (line);
    if (!av[0])
      continue;

    switch (pid = fork ()) {
    case -1:
      perror ("fork");
      break;
    case 0:
      doexec ();
      break;
    default:
      waitpid (pid, NULL, 0);
      break;
    }
  }
```

```
}
```

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

char **av;
char *infile;
char *outfile;
char *errfile;
char *outcmd;
int avsize;

void
avreserve (int n)
{
  int oldavsize = avsize;

  if (avsize > n + 1)
    return;

  avsize = 2 * (oldavsize + 1);
  if (avsize <= n)
    avsize = n + 1;
  av = realloc (av, avsize * sizeof (*av));
  while (oldavsize < avsize)
    av[oldavsize++] = NULL;
}

void
parseline (char *line)
{
  char *a;
  int n;

  outcmd = infile = outfile = errfile = NULL;
  for (n = 0; n < avsize; n++)
    av[n] = NULL;

  a = strtok (line, " \t\r\n");
  for (n = 0; a; n++) {
    if (a[0] == '<')
      infile = a[1] ? a + 1 : strtok (NULL, " \t\r\n");
    else if (a[0] == '>')
      outfile = a[1] ? a + 1 : strtok (NULL, " \t\r\n");
    else if (a[0] == '|') {
      if (!a[1])
        outcmd = strtok (NULL, "");
      else {
        outcmd = a + 1;
        a = strtok (NULL, "");
        while (a > outcmd && !a[-1])
          *--a = ' ';
      }
    }
    else if (a[0] == '2' && a[1] == '>')
      errfile = a[2] ? a + 2 : strtok (NULL, " \t\r\n");
    else {
      avreserve (n);
      av[n] = a;
    }
    a = strtok (NULL, " \t\r\n");
  }
```

```
}

void
doexec (void)
{
  int fd;

  while (outcmd) {
    int pipefds[2];

    if (outfile) {
      fprintf (stderr, "syntax error: > in pipe writer\n");
      exit (1);
    }

    if (pipe (pipefds) < 0) {
      perror ("pipe");
      exit (0);
    }

    switch (fork ()) {
    case -1:
      perror ("fork");
      exit (1);
    case 0:
      if (pipefds[1] != 1) {
        dup2 (pipefds[1], 1);
        close (pipefds[1]);
      }
      close (pipefds[0]);
      outcmd = NULL;
      break;
    default:
      if (pipefds[0] != 0) {
        dup2 (pipefds[0], 0);
        close (pipefds[0]);
      }
      close (pipefds[1]);
      parseline (outcmd);
      if (infile) {
        fprintf (stderr, "syntax error: < in pipe reader\n");
        exit (1);
      }
      break;
    }
  }

  if (infile) {
    if ((fd = open (infile, O_RDONLY)) < 0) {
      perror (infile);
      exit (1);
    }
    if (fd != 0) {
      dup2 (fd, 0);
      close (fd);
    }
  }

  if (outfile) {
    if ((fd = open (outfile, O_WRONLY|O_CREAT|O_TRUNC, 0666)) < 0) {
      perror (outfile);
      exit (1);
    }
    if (fd != 1) {
      dup2 (fd, 1);
```

```
      close (fd);
    }
  }

  if (errfile) {
    if ((fd = open (errfile, O_WRONLY|O_CREAT|O_TRUNC, 0666)) < 0) {
      perror (errfile);
      exit (1);
    }
    if (fd != 2) {
      dup2 (fd, 2);
      close (fd);
    }
  }

  execvp (av[0], av);
  perror (av[0]);
  exit (1);
}

int
main (void)
{
  char buf[512];
  char *line;
  int pid;

  avreserve (10);

  for (;;) {
    write (2, "$ ", 2);
    if (!(line = fgets (buf, sizeof (buf), stdin))) {
      write (2, "EOF\n", 4);
      exit (0);
    }
    parseline (line);
    if (!av[0])
      continue;

    switch (pid = fork ()) {
    case -1:
      perror ("fork");
      break;
    case 0:
      doexec ();
      break;
    default:
      waitpid (pid, NULL, 0);
      break;
    }
  }
}
```

3. Concurrency

Review: Thread package API

- `tid thread_create (void (*fn) (void *), void *arg);`
  - Create a new thread that calls `fn` with `arg`
- `void thread_exit ();`
- `void thread_join (tid thread);`
- **The execution of multiple threads is interleaved**
- **Can have *non-preemptive threads*:**
  - One thread executes exclusively until it makes a blocking call
- **Or *preemptive threads* (what we usually mean in this class):**
  - May switch to another thread between any two instructions.
- **Using multiple CPUs is inherently preemptive**
  - Even if you don't take CPU_0 away from thread T, another thread on CPU_1 can execute "between" any two instructions of T

Program A

```
int flag1 = 0, flag2 = 0;

void p1 (void *ignored) {
  flag1 = 1;
  if (!flag2) { critical_section_1 (); }
}

void p2 (void *ignored) {
  flag2 = 1;
  if (!flag1) { critical_section_2 (); }
}

int main () {
  tid id = thread_create (p1, NULL);
  p2 ();
  thread_join (id);
}
```

Q: Can both critical sections run?

Program B

```
int data = 0;
int ready = 0;

 void p1 (void *ignored) {
   data = 2000;
   ready = 1;
 }

 void p2 (void *ignored) {
   while (!ready)
     ;
   use (data);
 }

 int main () { ... }
```

Q: Can `use` be called with value 0?

Program C

```
int a = 0;
int b = 0;

void p1 (void *ignored) {
  a = 1;
}

void p2 (void *ignored) {
  if (a == 1)
    b = 1;
}

void p3 (void *ignored) {
  if (b == 1)
    use (a);
}
```

Q: If `p1–3` run concurrently, can `use` be called with value 0?

Correct answers

[git push slides to web site now]

Correct answers

- **Program A: I don't know**

Correct answers

- **Program A: I don't know**
- **Program B: I don't know**

Correct answers

- **Program A: I don't know**
- **Program B: I don't know**
- **Program C: I don't know**
- **Why don't we know?**
  - It depends on what machine you use
  - If a system provides *sequential consistency*, then answers all No
  - But not all hardware provides sequential consistency
- **Note: Examples, other content from [Adve & Gharachorloo]**
- **Another great reference: Why Memory Barriers**

Outline

Sequential Consistency

Definition

Sequential consistency: The result of execution is as if all operations were executed in some sequential order, and the operations of each processor occurred in the order specified by the program.
– Lamport

- **Boils down to two requirements on loads and stores:**
  1. Maintaining *program order* of each individual processor
  2. Ensuring *write atomicity*
- **Without SC (Sequential Consistency), multiple CPUs can be "worse"—i.e., less intuitive—than preemptive threads**
  - Result may not correspond to *any* instruction interleaving on 1 CPU
- **Why doesn't all hardware support sequential consistency?**

SC thwarts hardware optimizations

- **Complicates write buffers**
  - E.g., read flag$n$ before flag$(3 - n)$ written through in Program A
- **Can't re-order overlapping write operations**
  - Concurrent writes to different memory modules
  - Coalescing writes to same cache line
- **Complicates non-blocking reads**
  - E.g., speculatively prefetch `data` in Program B
- **Makes cache coherence more expensive**
  - Must delay write completion until invalidation/update (Program B)
  - Can't allow overlapping updates if no globally visible order (Program C)

SC thwarts compiler optimizations

- **Code motion**
- **Caching value in register**
  - Collapse multiple loads/stores of same address into one operation
- **Common subexpression elimination**
  - Could cause memory location to be read fewer times
- **Loop blocking**
  - Re-arrange loops for better cache performance
- **Software pipelining**
  - Move instructions across iterations of a loop to overlap instruction latency with branch cost

x86 consistency [intel 3a, §8.2]

- **x86 supports multiple consistency/caching models**
  - Memory Type Range Registers (MTRR) specify consistency for ranges of physical memory (e.g., frame buffer)
  - Page Attribute Table (PAT) allows control for each 4K page
- **Choices include:**
  - **WB**: Write-back caching (the default)
  - **WT**: Write-through caching (all writes go to memory)
  - **UC**: Uncacheable (for device memory)
  - **WC**: Write-combining – weak consistency & no caching (used for frame buffers, when sending a lot of data to GPU)
- **Some instructions have weaker consistency**
  - String instructions (written cache-lines can be re-ordered)
  - Special "non-temporal" store instructions (`movnt*`) that bypass cache and can be re-ordered with respect to other writes

x86 WB consistency

- **Old x86s (e.g, 486, Pentium 1) had almost SC**
  - Exception: A read could finish before an earlier write to a different location
  - Which of Programs A, B, C might be affected?
- **Reminder:**
  - Program A: `flag1 = 1; if (!flag2) critical_section_1();`
  - Program B: `while (!ready); use(data);`
  - Program C: P2 `if (a == 1) b = 1;` and P3 `if (b == 1) use(a);`

x86 WB consistency

- **Old x86s (e.g, 486, Pentium 1) had almost SC**
  - Exception: A read could finish before an earlier write to a different location
  - Which of Programs A, B, C might be affected? *Just A*
- **Newer x86s also let a CPU read its own writes early**

```
volatile int flag1;
int p1 (void)
{
  register int f, g;
  flag1 = 1;
  f = flag1;
  g = flag2;
  return 2*f + g;
}
```

```
volatile int flag2;
int p2 (void)
{
  register int f, g;
  flag2 = 1;
  f = flag2;
  g = flag1;
  return 2*f + g;
}
```

  - E.g., *both* `p1` and `p2` can return 2:
  - Older CPUs would wait at "`f = ...`" until store complete

x86 atomicity

- `lock` **prefix makes a memory instruction atomic**
  - Historically locked bus for duration of instruction (expensive!)
  - Now requires exclusively caching memory, synchronizing with other memory operations
  - All lock instructions totally ordered
  - Other memory instructions cannot be re-ordered with locked ones
- `xchg` **instruction is always locked (even without prefix)**
- **Special barrier (or "fence") instructions can prevent re-ordering**
  - `lfence` – can't be reordered with reads (or later writes)
  - `sfence` – can't be reordered with writes (e.g., use after non-temporal stores, before setting a *ready* flag)
  - `mfence` – can't be reordered with reads or writes

Outline

Assuming sequential consistency

- **Often we reason about concurrent code assuming SC**
- **But for low-level code, either know your memory model or program for worst-case relaxed consistency (∼DEC alpha)**
  - May need to sprinkle barrier/fence instructions into your source
  - Or may need compiler barriers to restrict optimization
- **For most code, avoid depending on memory model**
  - Idea: If you obey certain rules (discussed later) …system behavior should be indistinguishable from SC
- **Let's for now say we have sequential consistency**
- **Example concurrent code: Producer/Consumer**
  - `buffer` stores `BUFFER_SIZE` items
  - `count` is number of used slots
  - `out` is next empty buffer slot to fill (if any)
  - `in` is oldest filled slot to consume (if any)

```
void producer (void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();
        while (count == BUFFER_SIZE)
            /* do nothing */;
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
    }
}

void consumer (void *ignored) {
    for (;;) {
        while (count == 0)
            /* do nothing */;
        item *nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        consume_item (nextConsumed);
    }
}
```

Q: What can go wrong in above threads (even with SC)?

Data races

- `count` **may have wrong value**
- **Possible implementation of** `count++` **and** `count--`

register←count	register←count
register←register + 1	register←register − 1
count←register	count←register

- **Possible execution (count one less than correct):**

register←count	
register←register + 1	
	register←count
	register←register − 1
count←register	
	count←register

Data races (continued)

- **What about a single-instruction add?**
  - E.g., i386 allows single instruction `addl $1,_count`
  - So implement `count++/--` with one instruction
  - Now are we safe?

Data races (continued)

- **What about a single-instruction add?**
  - E.g., i386 allows single instruction `addl $1,_count`
  - So implement `count++/--` with one instruction
  - Now are we safe? Not on multiprocessors!
- **A single instruction may encode a load and a store operation**
  - S.C. doesn't make such *read-modify-write* instructions atomic
  - So on multiprocessor, suffer same race as 3-instruction version
- **Can make x86 instruction atomic with** `lock` **prefix**
  - But `lock` potentially very expensive
  - Compiler assumes you don't want penalty, doesn't emit it
- **Need solution to *critical section* problem**
  - Place `count++` and `count--` in critical section
  - Protect critical sections from concurrent execution

Desired properties of solution

- ***Mutual Exclusion***
  - Only one thread can be in critical section at a time
- ***Progress***
  - Say no process currently in critical section (C.S.)
  - One of the processes trying to enter will eventually get in
- ***Bounded waiting***
  - Once a thread *T* starts trying to enter the critical section, there is a bound on the number of times other threads get in
- **Note progress vs. bounded waiting**
  - If no thread can enter C.S., don't have progress
  - If thread *A* waiting to enter C.S. while *B* repeatedly leaves and re-enters C.S. *ad infinitum*, don't have bounded waiting

Peterson's solution

- **Still assuming sequential consistency**
- **Assume two threads,** T_0 **and** T_1
- **Variables**
  - `int not_turn;` // not this thread's turn to enter C.S.
  - `bool wants[2];` // `wants[i]` indicates if T_i wants to enter C.S.
- **Code:**

```
for (;;) { /* assume i is thread number (0 or 1) */
  wants[i] = true;
  not_turn = i;
  while (wants[1-i] && not_turn == i)
    /* other thread wants in and not our turn, so loop */;
  Critical_section ();
  wants[i] = false;
  Remainder_section ();
}
```

Does Peterson's solution work?

```
for (;;) { /* code in thread i */
  wants[i] = true;
  not_turn = i;
  while (wants[1-i] && not_turn == i)
    /* other thread wants in and not our turn, so loop */;
  Critical_section ();
  wants[i] = false;
  Remainder_section ();
}
```

- **Mutual exclusion – can't both be in C.S.**
  - Would mean `wants[0] == wants[1] == true`, so `not_turn` would have blocked one thread from C.S.
- **Progress – given demand, one thread can always enter C.S.**
  - If T_{1-i} doesn't want C.S., `wants[1-i] == false`, so T_i won't loop
  - If both threads want in, one thread is not the `not_turn` thread
- **Bounded waiting – similar argument to progress**
  - If T_i wants lock and T_{1-i} tries to re-enter, T_{1-i} will set `not_turn = 1 - i`, allowing T_i in

Outline

Mutexes

- **Peterson expensive, only works for 2 processes**
  - Can generalize to n, but for some fixed n
- **Must adapt to machine memory model if not SC**
  - If you need machine-specific barriers anyway, might as well take advantage of other instructions helpful for synchronization
- **Want to insulate programmer from implementing synchronization primitives**
- **Thread packages typically provide *mutexes*:**

```
void mutex_init (mutex_t *m, ...);
void mutex_lock (mutex_t *m);
int mutex_trylock (mutex_t *m);
void mutex_unlock (mutex_t *m);
```

  - Only one thread acquires `m` at a time, others wait

Thread API contract

- **All global data should be protected by a mutex!**
  - Global = accessed by more than one thread, at least one write
  - Exception is initialization, before exposed to other threads
  - This is the responsibility of the application writer
- **If you use mutexes properly, behavior should be indistinguishable from Sequential Consistency**
  - This is the responsibility of the threads package (& compiler)
  - Mutex is broken if you use properly and don't see SC
- **OS kernels also need synchronization**
  - Some mechanisms look like mutexes
  - But interrupts complicate things (incompatible w. mutexes)

Same concept, many names

- **Most popular application-level thread API: Pthreads**
  - Function names in this lecture all based on Pthreads
  - Just add `pthread_` prefix
  - E.g., `pthread_mutex_t`, `pthread_mutex_lock`, ...
- **C11 uses `mtx_` instead of `mutex_`, C++11 uses methods on `mutex`**
- **Pintos uses `struct lock` for mutexes:**

```
void lock_init (struct lock *);
void lock_acquire (struct lock *);
bool lock_try_acquire (struct lock *);
void lock_release (struct lock *);
```

- **Extra Pintos feature:**
  - Release checks that lock was acquired by same thread
  - `bool lock_held_by_current_thread (struct lock *lock);`

Improved producer

```
mutex_t mutex = MUTEX_INITIALIZER;

void producer (void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();

        mutex_lock (&mutex);
        while (count == BUFFER_SIZE) {
          mutex_unlock (&mutex);
          thread_yield ();
          mutex_lock (&mutex);
        }

        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        mutex_unlock (&mutex);
    }
}
```

Improved consumer

```
void consumer (void *ignored) {
    for (;;) {
        mutex_lock (&mutex);
        while (count == 0) {
          mutex_unlock (&mutex); /* <--- Why? */
          thread_yield ();
          mutex_lock (&mutex);
        }

        item *nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        mutex_unlock (&mutex);

        consume_item (nextConsumed);
    }
}
```


Condition variables

- **Busy-waiting in application is a bad idea**
  - Consumes CPU even when a thread can't make progress
  - Unnecessarily slows other threads/processes or wastes power
- **Better to inform scheduler of which threads can run**
- **Typically done with *condition variables***
- `struct cond_t;` (`pthread_cond_t` **or** `condition` **in Pintos)**
- `void cond_init (cond_t *, ...);`
- `void cond_wait (cond_t *c, mutex_t *m);`
  - Atomically unlock `m` and sleep until `c` signaled
  - Then re-acquire `m` and resume executing
- `void cond_signal (cond_t *c);`
  `void cond_broadcast (cond_t *c);`
  - Wake one/all threads waiting on `c`

Improved producer

```
mutex_t mutex = MUTEX_INITIALIZER;
cond_t nonempty = COND_INITIALIZER;
cond_t nonfull = COND_INITIALIZER;

void producer (void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();

        mutex_lock (&mutex);
        while (count == BUFFER_SIZE)
          cond_wait (&nonfull, &mutex);

        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        cond_signal (&nonempty);
        mutex_unlock (&mutex);
    }
}
```


Improved consumer

```
void consumer (void *ignored) {
    for (;;) {
        mutex_lock (&mutex);
        while (count == 0)
          cond_wait (&nonempty, &mutex);

        item *nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        cond_signal (&nonfull);
        mutex_unlock (&mutex);

        consume_item (nextConsumed);
    }
}
```


Re-check conditions

- **Always re-check condition on wake-up**

```
while (count == 0) /* not if */
  cond_wait (&nonempty, &mutex);
```

- **Otherwise, breaks with spurious wakeup or two consumers**
  - Start where Consumer 1 has mutex but buffer empty, then:

Consumer 1	Consumer 2	Producer
`cond_wait (...);`		`mutex_lock (...);`
		⋮
		`count++;`
		`cond_signal (...);`
	`mutex_lock (...);`	`mutex_unlock (...);`
	`if (count == 0)`	
	⋮	
	use `buffer[out]` ...	
	`count--;`	
	`mutex_unlock (...);`	
use `buffer[out]` ... ⟵ No items in buffer		

Condition variables (continued)

- **Why must `cond_wait` both release mutex & sleep?**
- **Why not separate mutexes and condition variables?**

```
while (count == BUFFER_SIZE) {
  mutex_unlock (&mutex);
  cond_wait (&nonfull);
  mutex_lock (&mutex);
}
```

Condition variables (continued)

- **Why must `cond_wait` both release mutex & sleep?**
- **Why not separate mutexes and condition variables?**

```
while (count == BUFFER_SIZE) {
  mutex_unlock (&mutex);
  cond_wait (&nonfull);
  mutex_lock (&mutex);
}
```

- **Can end up stuck waiting when bad interleaving**

Producer

```
while (count == BUFFER_SIZE)
  mutex_unlock (&mutex);




  cond_wait (&nonfull);
```

Consumer

```
mutex_lock (&mutex);
...
count--;
cond_signal (&nonfull);
```

- **Problem: `cond_wait` & `cond_signal` do not commute**

Other thread package features

- **Alerts – cause exception in a thread**
- **Timedwait – timeout on condition variable**
- **Shared locks – concurrent read accesses to data**
- **Thread priorities – control scheduling policy**
  - Mutex attributes allow various forms of *priority donation* (will be familiar concept after lab 1)
- **Thread-specific global data**
  - Need for things like `errno`
- **Different synchronization primitives (later in lecture)**

Outline

Implementing synchronization

- **Implement mutex as straight-forward data structure?**

```
typedef struct mutex {
  bool is_locked;         /* true if locked */
  thread_id_t owner;      /* thread holding lock, if locked */
  thread_list_t waiters;  /* threads waiting for lock */

} mutex_t;
```


Implementing synchronization

- **Implement mutex as straight-forward data structure?**

```
typedef struct mutex {
  bool is_locked;         /* true if locked */
  thread_id_t owner;      /* thread holding lock, if locked */
  thread_list_t waiters;  /* threads waiting for lock */
  lower_level_lock_t lk;  /* Protect above fields */
} mutex_t;
```

  - Fine, so long as we avoid data races on the mutex itself
- **Need lower-level lock `lk` for mutual exclusion**
  - Internally, `mutex_*` functions bracket code with `lock(&mutex->lk)` … `unlock(&mutex->lk)`
  - Otherwise, data races! (E.g., two threads manipulating `waiters`)
- **How to implement `lower_level_lock_t`?**
  - Could use Peterson's algorithm, but typically a bad idea (too slow and don't know maximum number of threads)

Approach #1: Disable interrupts

- **Only for apps with $n : 1$ threads (1 kthread)**
  - Cannot take advantage of multiprocessors
  - But sometimes most efficient solution for uniprocessors
- **Typical setup: periodic timer signal caught by thread scheduler**
- **Have per-thread "do not interrupt" (DNI) bit**
- **`lock (lk)`: sets thread's DNI bit**
- **If timer interrupt arrives**
  - Check interrupted thread's DNI bit
  - If DNI clear, preempt current thread
  - If DNI set, set "interrupted" (I) bit & resume current thread
- **`unlock (lk)`: clears DNI bit *and* checks I bit**
  - If I bit is set, immediately yields the CPU

Approach #2: Spinlocks

- **Most CPUs support atomic read-[modify-]write**
- **Example:** `int test_and_set (int *lockp);`
  - Atomically sets `*lockp = 1` and returns old value
  - Special instruction – no way to implement in portable C99 (C11 supports with explicit `atomic_flag_tet_and_set` function)
- **Use this instruction to implement *spinlocks*:**

```
#define lock(lockp)  while (test_and_set (lockp))
#define trylock(lockp) (test_and_set (lockp) == 0)
#define unlock(lockp) *lockp = 0
```

- **Spinlocks implement mutex's** `lower_level_lock_t`
- **Can you use spinlocks instead of mutexes?**
  - Wastes CPU, especially if thread holding lock not running
  - Mutex functions have short C.S., less likely to be preempted
  - On multiprocessor, sometimes good to spin for a bit, then yield

Synchronization on x86

- **Test-and-set only one possible atomic instruction**
- **x86** `xchg` **instruction, exchanges reg with mem**
  - Can use to implement test-and-set

```
_test_and_set:
        movl    4(%esp), %edx  # %edx = lockp
        movl    $1, %eax       # %eax = 1
        xchgl   %eax, (%edx)   # swap (%eax, *lockp)
        ret
```

- **CPU locks memory system around read and write**
  - Recall `xchgl` always acts like it has implicit `lock` prefix
  - Prevents other uses of the bus (e.g., DMA)
- **Usually runs at memory bus speed, not CPU speed**
  - Much slower than cached read/buffered write

Synchronization on alpha

- `ldl_l` **– load locked**
  `stl_c` **– store conditional (reg←0 if not atomic w.** `ldl_l`**)**

```
_test_and_set:
    ldq_l   v0, 0(a0)          # v0 = *lockp (LOCKED)
    bne     v0, 1f             # if (v0) return
    addq    zero, 1, v0        # v0 = 1
    stq_c   v0, 0(a0)          # *lockp = v0 (CONDITIONAL)
    beq     v0, _test_and_set # if (failed) try again
    mb
    addq    zero, zero, v0     # return 0
1:
    ret     zero, (ra), 1
```

- **Note: Alpha memory consistency weaker than x86**
  - Want all CPUs to think memory accesses in C.S. happened after acquiring lock, before releasing
  - *Memory barrier* instruction `mb` ensures this (c.f. `mfence` on x86)
  - See Why Memory Barriers for why alpha still worth understanding

Kernel Synchronization

- **Should kernel use locks or disable interrupts?**
- **Old UNIX had 1 CPU, non-preemptive threads, no mutexes**
  - Interface designed for single CPU, so `count++` etc. not data race
  - . . . *Unless* memory shared with an interrupt handler

```
int x = splhigh (); /* Disable interrupts */
/* touch data shared with interrupt handler ... */
splx (x);           /* Restore previous state */
```

  - C.f., `intr_disable` / `intr_set_level` in Pintos, and `preempt_disable` / `preempt_enable` in linux
- **Used arbitrary pointers like condition variables**
  - `int [t]sleep (void *ident, int priority, ...);` put thread to sleep; will wake up at `priority` (∼`cond_wait`)
  - `int wakeup (void *ident);` wake up all threads sleeping on `ident` (∼`cond_broadcast`)

Kernel locks

- **Nowadays, should design for multiprocessors**
  - Even if first version of OS is for uniprocessor
  - Someday may want multiple CPUs and need *preemptive* threads
  - That's why Pintos uses sleeping locks (*sleeping* locks means mutexes, as opposed to *spin*locks)
- **Multiprocessor performance needs fine-grained locks**
  - Want to be able to call into the kernel on multiple CPUs
- **If kernel has locks, should it ever disable interrupts?**

Kernel locks

- **Nowadays, should design for multiprocessors**
  - Even if first version of OS is for uniprocessor
  - Someday may want multiple CPUs and need *preemptive* threads
  - That's why Pintos uses sleeping locks (*sleeping* locks means mutexes, as opposed to *spin*locks)
- **Multiprocessor performance needs fine-grained locks**
  - Want to be able to call into the kernel on multiple CPUs
- **If kernel has locks, should it ever disable interrupts?**
  - Yes! Can't sleep in interrupt handler, so can't wait for lock
  - So even modern OSes have support for disabling interrupts
  - Often uses DNI trick when cheaper than masking interrupts in hardware

Outline

Semaphores [Dijkstra]

- **A *Semaphore* is initialized with an integer *N***
- **Provides two functions:**
  - `sem_wait (S)` (originally called *P*, called `sema_down` in Pintos)
  - `sem_signal (S)` (originally called *V*, called `sema_up` in Pintos)
- **Guarantees `sem_wait` will return only *N* more times than `sem_signal` called**
  - Example: If $N == 1$, then semaphore acts as a mutex with `sem_wait` as lock and `sem_signal` as unlock
- **Semaphores give elegant solutions to some problems**
  - Unlike condition variables, wait & signal commute
- **Linux primarily uses semaphores for sleeping locks**
  - `sema_init`, `down_interruptible`, `up`, ...
  - Also weird reader-writer semaphores, `rw_semaphore` [Love]

Semaphore producer/consumer

- **Initialize `full` to 0 (block consumer when buffer empty)**
- **Initialize `empty` to *N* (block producer when queue full)**

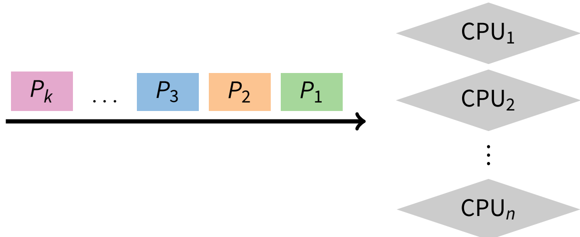
```
void producer (void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();
        sem_wait (&empty);
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        sem_signal (&full);
    }
}
void consumer (void *ignored) {
    for (;;) {
        sem_wait (&full);
        item *nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        sem_signal (&empty);
        consume_item (nextConsumed);
    }
}
```


Various synchronization mechanisms

- **Other more esoteric primitives you might encounter**
  - Plan 9 used a rendezvous mechanism
  - Haskell uses MVars (like channels of depth 1)
- **Many synchronization mechanisms equally expressive**
  - Pintos implements locks, condition vars using semaphores
  - Could have been vice versa
  - Can even implement condition variables in terms of mutexes
- **Why base everything around semaphore implementation?**
  - High-level answer: no particularly good reason
  - If you want only one mechanism, can't be condition variables (interface fundamentally requires mutexes)
  - Because `sem_wait` and `sem_signal` commute, eliminates problem of condition variables w/o mutexes

4. Scheduling

CPU scheduling

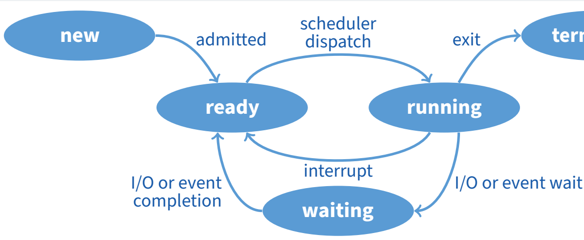


- **The scheduling problem:**
  - Have k jobs ready to run
  - Have $n \geq 1$ CPUs that can run them
- **Which jobs should we assign to which CPU(s)?**

Outline

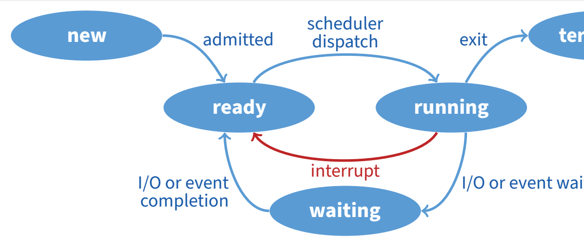
1. Textbook scheduling
2. Priority scheduling
3. Advanced scheduling issues
4. Virtual time case studies

When do we schedule CPU?



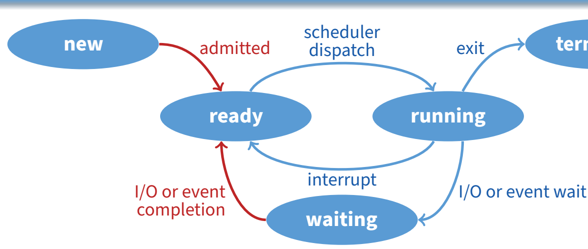
- **Scheduling decisions may take place when a process:**
  1. Switches from running to ready state
  2. Switches from new/waiting to ready
  3. Switches from running to waiting state
  4. Exits
- **Non-preemptive schedulers use 3 & 4 only**
- **Preemptive schedulers run at all four points**

When do we schedule CPU?



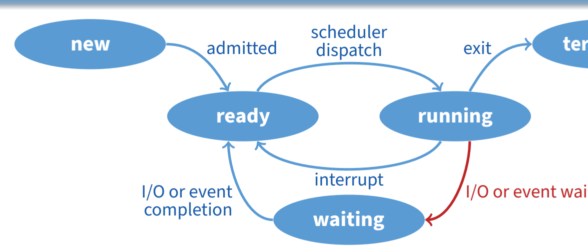
- **Scheduling decisions may take place when a process:**
  1. ⟶ Switches from running to ready state
  2. Switches from new/waiting to ready
  3. Switches from running to waiting state
  4. Exits
- **Non-preemptive schedulers use 3 & 4 only**
- **Preemptive schedulers run at all four points**

When do we schedule CPU?



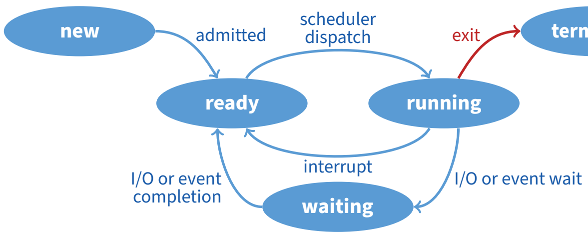
- **Scheduling decisions may take place when a process:**
  1. Switches from running to ready state
  2. ⟶ Switches from new/waiting to ready
  3. Switches from running to waiting state
  4. Exits
- **Non-preemptive schedulers use 3 & 4 only**
- **Preemptive schedulers run at all four points**

When do we schedule CPU?



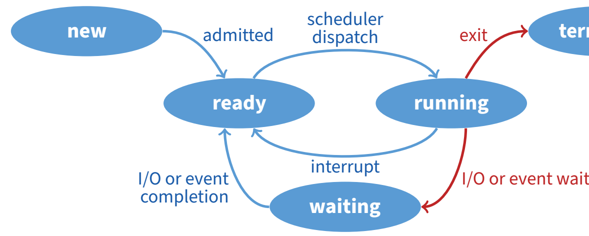
- **Scheduling decisions may take place when a process:**
  1. Switches from running to ready state
  2. Switches from new/waiting to ready
  3. ⟶ Switches from running to waiting state
  4. Exits
- **Non-preemptive schedulers use 3 & 4 only**
- **Preemptive schedulers run at all four points**

When do we schedule CPU?



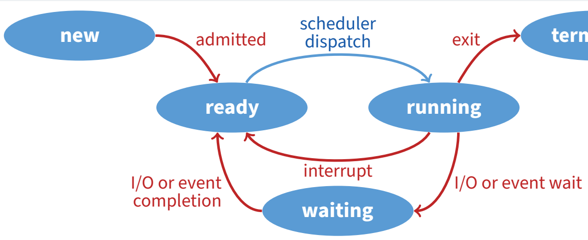
- **Scheduling decisions may take place when a process:**
  1. Switches from running to ready state
  2. Switches from new/waiting to ready
  3. Switches from running to waiting state
  4. Exits
- **Non-preemptive schedulers use 3 & 4 only**
- **Preemptive schedulers run at all four points**

When do we schedule CPU?



- **Scheduling decisions may take place when a process:**
  1. Switches from running to ready state
  2. Switches from new/waiting to ready
  3. Switches from running to waiting state
  4. Exits
- **Non-preemptive schedulers use 3 & 4 only**
- **Preemptive schedulers run at all four points**

When do we schedule CPU?



- **Scheduling decisions may take place when a process:**
  1. Switches from running to ready state
  2. Switches from new/waiting to ready
  3. Switches from running to waiting state
  4. Exits
- **Non-preemptive schedulers use 3 & 4 only**
- **Preemptive schedulers run at all four points**

Scheduling criteria

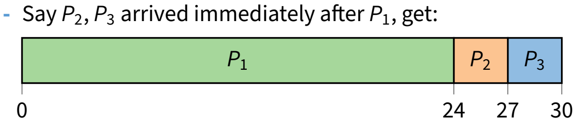
- **Why do we care?**
  - What goals should we have for a scheduling algorithm?

Scheduling criteria

- **Why do we care?**
  - What goals should we have for a scheduling algorithm?
- ***Throughput* – # of processes that complete per unit time**
  - Higher is better
- ***Turnaround time* – time for each process to complete**
  - Lower is better
- ***Response time* – time from request to first response**
  - I.e., time between **waiting**→**ready** transition and **ready**→**running** (e.g., key press to echo, not launch to exit)
  - Lower is better
- **Above criteria are affected by secondary criteria**
  - *CPU utilization* – fraction of time CPU doing productive work
  - *Waiting time* – time each process waits in ready queue

Example: FCFS Scheduling

- **Run jobs in order that they arrive**
  - Called "*First-come first-served*" (FCFS)
  - E.g., Say P_1 needs 24 sec, while P_2 and P_3 need 3.
  - Say P_2, P_3 arrived immediately after P_1, get:



- **Dirt simple to implement—how good is it?**
- **Throughput: 3 jobs / 30 sec = 0.1 jobs/sec**
- **Turnaround Time:** $P_1 : 24$**,** $P_2 : 27$**,** $P_3 : 30$
  - Average TT: $(24 + 27 + 30)/3 = 27$
- **Can we do better?**

FCFS continued

- **Suppose we scheduled P_2, P_3, then P_1**
  - Would get:

- **Throughput: 3 jobs / 30 sec = 0.1 jobs/sec**
- **Turnaround time:** $P_1 : 30$, $P_2 : 3$, $P_3 : 6$
  - Average TT: $(30 + 3 + 6)/3 = 13$ – much less than 27
- **Lesson: scheduling algorithm can reduce TT**
  - Minimizing waiting time can improve RT and TT
- **Can a scheduling algorithm improve throughput?**

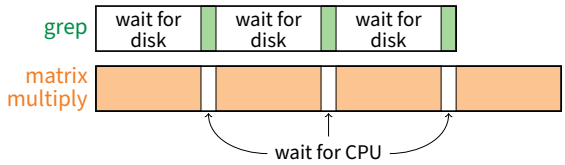
FCFS continued

- **Suppose we scheduled P_2, P_3, then P_1**
  - Would get:

- **Throughput: 3 jobs / 30 sec = 0.1 jobs/sec**
- **Turnaround time:** $P_1 : 30$, $P_2 : 3$, $P_3 : 6$
  - Average TT: $(30 + 3 + 6)/3 = 13$ – much less than 27
- **Lesson: scheduling algorithm can reduce TT**
  - Minimizing waiting time can improve RT and TT
- **Can a scheduling algorithm improve throughput?**
  - Yes, if jobs require both computation and I/O

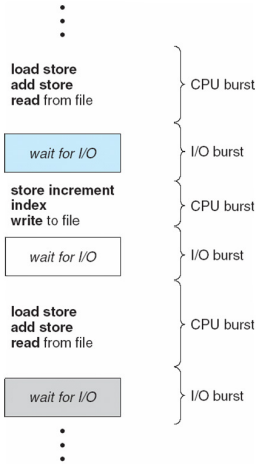
View CPU and I/O devices the same

- **CPU is one of several devices needed by users' jobs**
  - CPU runs compute jobs, Disk drive runs disk jobs, etc.
  - With network, part of job may run on remote CPU
- **Scheduling 1-CPU system with n I/O devices like scheduling asymmetric $(n + 1)$-CPU multiprocessor**
  - Result: all I/O devices + CPU busy $\Longrightarrow$ $(n + 1)$-fold throughput gain!
- **Example: disk-bound grep + CPU-bound matrix multiply**
  - Overlap them just right? throughput will be almost doubled

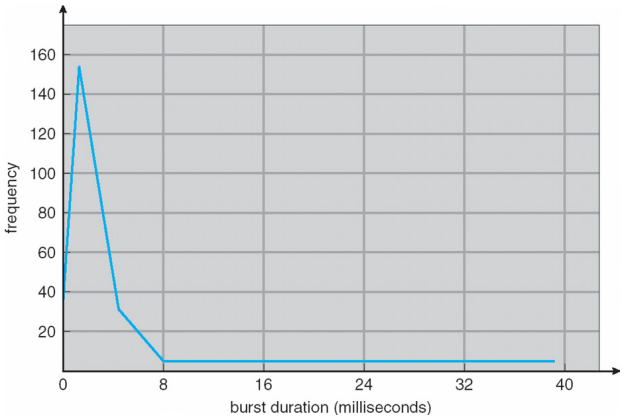


Bursts of computation & I/O

- **Jobs contain I/O and computation**
  - Bursts of computation
  - Then must wait for I/O
- **To maximize throughput, maximize both CPU and I/O device utilization**
- **How to do?**
  - Overlap computation from one job with I/O from other jobs
  - Means *response time* very important for I/O-intensive jobs: I/O device will be idle until job gets small amount of CPU to issue next I/O request



Histogram of CPU-burst times



- **What does this mean for FCFS?**

FCFS Convoy effect

- **CPU-bound jobs will hold CPU until exit or I/O (but I/O rare for CPU-bound thread)**
  - Long periods where no I/O requests issued, and CPU held
  - Result: poor I/O device utilization
- **Example: one CPU-bound job, many I/O bound**
  - CPU-bound job runs (I/O devices idle)
  - Eventually, CPU-bound job blocks
  - I/O-bound jobs run, but each quickly blocks on I/O
  - CPU-bound job unblocks, runs again
  - All I/O requests complete, but CPU-bound job still hogs CPU
  - I/O devices sit idle since I/O-bound jobs can't issue next requests
- **Simple hack: run process whose I/O completed**
  - What is a potential problem?

FCFS Convoy effect

- **CPU-bound jobs will hold CPU until exit or I/O (but I/O rare for CPU-bound thread)**
  - Long periods where no I/O requests issued, and CPU held
  - Result: poor I/O device utilization
- **Example: one CPU-bound job, many I/O bound**
  - CPU-bound job runs (I/O devices idle)
  - Eventually, CPU-bound job blocks
  - I/O-bound jobs run, but each quickly blocks on I/O
  - CPU-bound job unblocks, runs again
  - All I/O requests complete, but CPU-bound job still hogs CPU
  - I/O devices sit idle since I/O-bound jobs can't issue next requests
- **Simple hack: run process whose I/O completed**
  - What is a potential problem?
    I/O-bound jobs can starve CPU-bound one

SJF Scheduling

- ***Shortest-job first* (SJF) attempts to minimize TT**
  - Schedule the job whose next CPU burst is the shortest
  - Misnomer unless "job" = one CPU burst with no I/O
    [term coined for context where there is no I/O, only compute]
- **Two schemes:**
  - *Non-preemptive* – once CPU given to the process it cannot be preempted until completes its CPU burst
  - *Preemptive* – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt
    (Known as the *Shortest-Remaining-Time-First* or SRTF)
- **What does SJF optimize?**

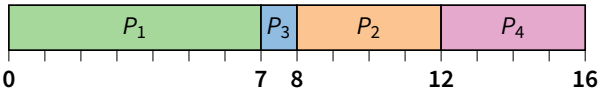
SJF Scheduling

- ***Shortest-job first* (SJF) attempts to minimize TT**
  - Schedule the job whose next CPU burst is the shortest
  - Misnomer unless "job" = one CPU burst with no I/O
    [term coined for context where there is no I/O, only compute]
- **Two schemes:**
  - *Non-preemptive* – once CPU given to the process it cannot be preempted until completes its CPU burst
  - *Preemptive* – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt
    (Known as the *Shortest-Remaining-Time-First* or SRTF)
- **What does SJF optimize?**
  - Gives minimum average *waiting time* for a given set of processes

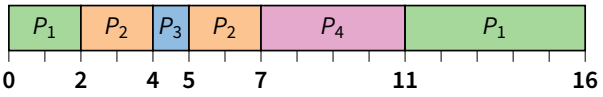
Examples

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- **Non-preemptive**



- **Preemptive**



- **Drawbacks?**

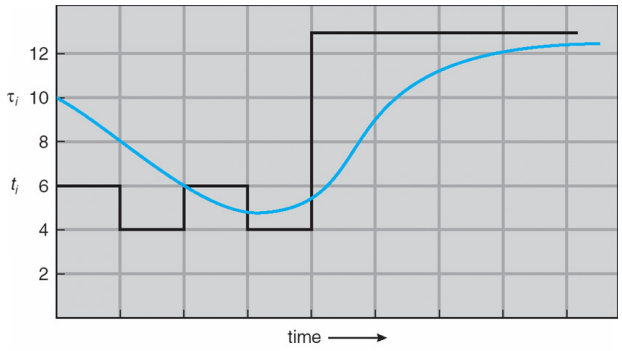
SJF limitations

- **Doesn't always minimize average TT**
  - Only minimizes waiting time
  - Example where turnaround time might be suboptimal?
- **Can lead to unfairness or starvation**
- **In practice, can't actually predict the future**
- **But can estimate CPU burst length based on past**
  - Exponentially weighted average a good idea
  - t_n actual length of process's n^{th} CPU burst
  - τ_{n+1} estimated length of proc's $(n+1)^{\text{st}}$
  - Choose parameter α where $0 < \alpha \leq 1$
  - Let $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

SJF limitations

- **Doesn't always minimize average TT**
  - Only minimizes waiting time
  - Example where turnaround time might be suboptimal?
  - Overall longer job has shorter bursts
- **Can lead to unfairness or starvation**
- **In practice, can't actually predict the future**
- **But can estimate CPU burst length based on past**
  - Exponentially weighted average a good idea
  - t_n actual length of process's n^{th} CPU burst
  - τ_{n+1} estimated length of proc's $(n+1)^{\text{st}}$
  - Choose parameter α where $0 < \alpha \leq 1$
  - Let $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

Exp. weighted average example



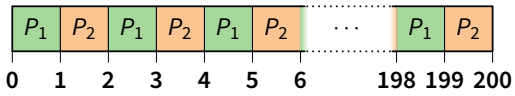
CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Round robin (RR) scheduling

- **Solution to fairness and starvation**
  - Preempt job after some time slice or *quantum*
  - When preempted, move to back of FIFO queue
  - (Many systems do some flavor of this)
- **Advantages:**
  - Fair allocation of CPU across jobs
  - Low average waiting time when job lengths vary
  - Good for responsiveness if small number of jobs
- **Disadvantages?**

RR disadvantages

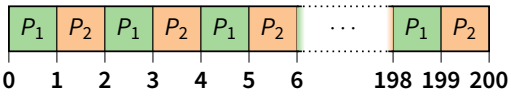
- **Varying sized jobs are good . . . what about same-sized jobs?**
- **Assume 2 jobs of time=100 each:**



- **Even if context switches were free. . .**
  - What would average turnaround time be with RR?
  - How does that compare to FCFS?

RR disadvantages

- **Varying sized jobs are good . . . what about same-sized jobs?**
- **Assume 2 jobs of time=100 each:**



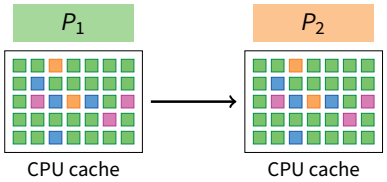
- **Even if context switches were free. . .**
  - What would average turnaround time be with RR? *199.5*
  - How does that compare to FCFS? *150*

Context switch costs

- **What is the cost of a context switch?**

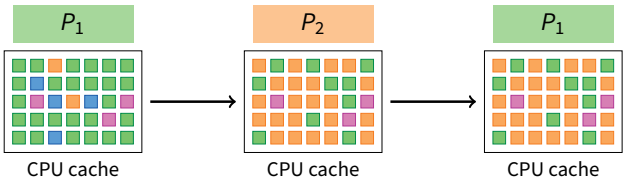
Context switch costs

- **What is the cost of a context switch?**
- **Brute CPU time cost in kernel**
  - Save and restore resisters, etc.
  - Switch address spaces (expensive instructions)
- **Indirect costs: cache, buffer cache, & TLB misses**

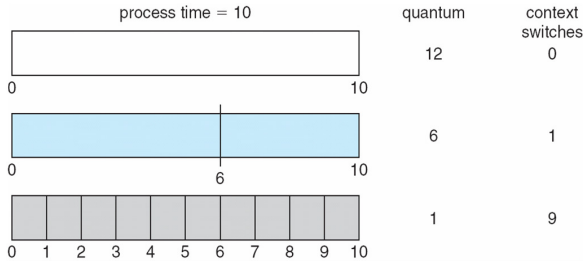


Context switch costs

- **What is the cost of a context switch?**
- **Brute CPU time cost in kernel**
  - Save and restore resisters, etc.
  - Switch address spaces (expensive instructions)
- **Indirect costs: cache, buffer cache, & TLB misses**

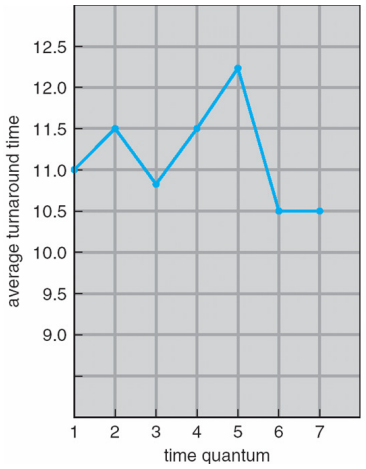


Time quantum



- **How to pick quantum?**
  - Want much larger than context switch cost
  - Majority of bursts should be less than quantum
  - But not so large system reverts to FCFS
- **Typical values: 1–100 msec**

Turnaround time vs. quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Two-level scheduling

- **Under memory constraints, may need to *swap* process to disk**
- **Switching to swapped out process very expensive**
  - Swapped out process has most memory pages on disk
  - Will have to fault them all in while running
  - One disk access costs ~10ms. On 1GHz machine, 10ms = 10 million cycles!
- **Solution: Context-switch-cost aware scheduling**
  - Run in-core subset for "a while"
  - Then swap some between disk and memory
- **How to pick subset? How to define "a while"?**
  - View as scheduling *memory* before scheduling CPU
  - Swapping in process is cost of memory "context switch"
  - So want "memory quantum" much larger than swapping cost

Outline

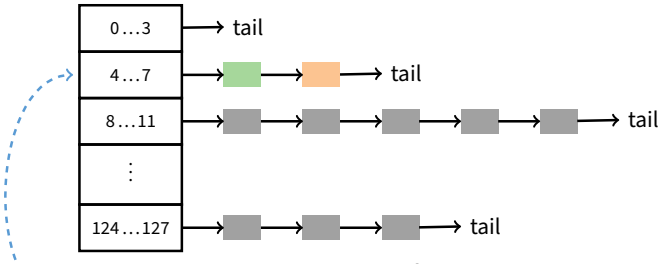
Priority scheduling

- **Associate a numeric priority with each process**
  - E.g., smaller number means higher priority (Unix/BSD)
  - Or smaller number means lower priority (Pintos)
- **Give CPU to the process with highest priority**
  - Can be done preemptively or non-preemptively
- **Note SJF is priority scheduling where priority is the predicted next CPU burst time**
- **Starvation – low priority processes may never execute**
- **Solution?**

Priority scheduling

- **Associate a numeric priority with each process**
  - E.g., smaller number means higher priority (Unix/BSD)
  - Or smaller number means lower priority (Pintos)
- **Give CPU to the process with highest priority**
  - Can be done preemptively or non-preemptively
- **Note SJF is priority scheduling where priority is the predicted next CPU burst time**
- **Starvation – low priority processes may never execute**
- **Solution?**
  - Aging: increase a process's priority as it waits

Multilevel feeedback queues (BSD)



- **Every runnable process on one of 32 run queues**
  - Kernel runs process on highest-priority non-empty queue
  - Round-robins among processes on same queue
- **Process priorities dynamically computed**
  - Processes moved between queues to reflect priority changes
  - If a process gets higher priority than running process, run it
- **Idea: Favor interactive jobs that use less CPU**

Process priority

- `p_nice` **– user-settable weighting factor**
- `p_estcpu` **– per-process estimated CPU usage**
  - Incremented whenever timer interrupt found process running
  - Decayed every second while process runnable

$$\texttt{p\_estcpu} \leftarrow \left(\frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1}\right) \texttt{p\_estcpu} + \texttt{p\_nice}$$

  - Load is sampled average of length of run queue plus short-term sleep queue over last minute
- **Run queue determined by** $\texttt{p\_usrpri}/4$

$$\texttt{p\_usrpri} \leftarrow 50 + \left(\frac{\texttt{p\_estcpu}}{4}\right) + 2 \cdot \texttt{p\_nice}$$

(value clipped if over 127)

Sleeping process increases priority

- `p_estcpu` **not updated while asleep**
  - Instead `p_slptime` keeps count of sleep time
- **When process becomes runnable**

$$\texttt{p\_estcpu} \leftarrow \left(\frac{2 \cdot \textbf{load}}{2 \cdot \textbf{load} + 1}\right)^{\texttt{p\_slptime}} \times \texttt{p\_estcpu}$$

  - Approximates decay ignoring nice and past loads
- **Previous description based on [McKusick][1] (*The Design and Implementation of the 4.4BSD Operating System*)**

[1] See library.stanford.edu for off-campus access

Pintos notes

- **Same basic idea for second half of project 1**
  - But 64 priorities, not 128
  - Higher numbers mean higher priority
  - Okay to have only one run queue if you prefer (less efficient, but we won't deduct points for it)
- **Have to negate priority equation:**

$$\texttt{priority} = 63 - \left(\frac{\texttt{recent\_cpu}}{4}\right) - 2 \cdot \texttt{nice}$$

Thread scheduling

- **With thread library, have two scheduling decisions:**
  - *Local Scheduling* – User-level thread library decides which user (green) thread to put onto an available native (i.e., kernel) thread
  - *Global Scheduling* – Kernel decides which native thread to run next
- **Can expose to the user**
  - E.g., `pthread_attr_setscope` allows two choices
  - `PTHREAD_SCOPE_SYSTEM` – thread scheduled like a process (effectively one native thread bound to user thread – Will return ENOTSUP in user-level pthreads implementation)
  - `PTHREAD_SCOPE_PROCESS` – thread scheduled within the current process (may have multiple user threads multiplexed onto kernel threads)

Thread dependencies

- **Say H at high priority, L at low priority**
  - L acquires lock ℓ.
  - Scenario 1 (ℓ a spinlock): H tries to acquire ℓ, fails, spins. L never gets to run.
  - Scenario 2 (ℓ a mutex): H tries to acquire ℓ, fails, blocks. M enters system at medium priority. L never gets to run.
  - Both scenarios are examples of *priority inversion*
- **Scheduling = deciding who should make progress**
  - A thread's importance should increase with the importance of those that depend on it
  - Naïve priority schemes violate this

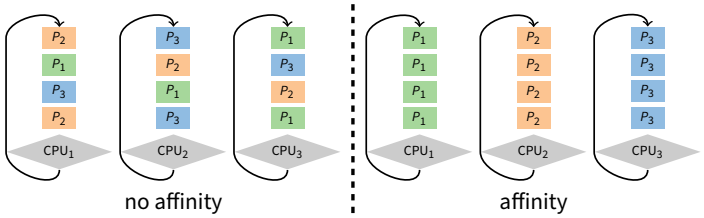
Priority donation

- **Say higher number = higher priority (like Pintos)**
- **Example 1: L (prio 2), M (prio 4), H (prio 8)**
  - L holds lock ℓ
  - M waits on ℓ, L's priority raised to $L_1 = \max(M, L) = 4$
  - Then H waits on ℓ, L's priority raised to $\max(H, L_1) = 8$
- **Example 2: Same L, M, H as above**
  - L holds lock ℓ_1, M holds lock ℓ_2
  - M waits on ℓ_1, L's priority now $L_1 = 4$ (as before)
  - Then H waits on ℓ_2. M's priority goes to $M_1 = \max(H, M) = 8$, *and* L's priority raised to $\max(M_1, L_1) = 8$
- **Example 3: L (prio 2), $M_1, \ldots M_{1000}$ (all prio 4)**
  - L has ℓ, and $M_1, \ldots, M_{1000}$ all block on ℓ. L's priority is $\max(L, M_1, \ldots, M_{1000}) = 4$.

Outline

Multiprocessor scheduling issues

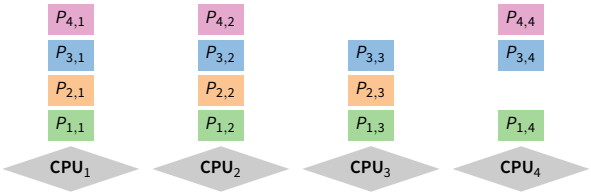
- **Must decide on more than which processes to run**
  - Must decide on which CPU to run which process
- **Moving between CPUs has costs**
  - More cache misses, depending on arch. more TLB misses too
- ***Affinity scheduling*—try to keep process/thread on same CPU**



  - But also prevent load imbalances
  - Do *cost-benefit* analysis when deciding to migrate… affinity can also be harmful, when tail latency is critical

Multiprocessor scheduling (cont)

- **Want related processes/threads scheduled together**
  - Good if threads access same resources (e.g., cached files)
  - Even more important if threads communicate often, otherwise must context switch to communicate
- ***Gang scheduling*—schedule all CPUs synchronously**
  - With synchronized quanta, easier to schedule related processes/threads together



Real-time scheduling

- **Two categories:**
  - *Soft real time*—miss deadline and audio playback will sound funny
  - *Hard real time*—miss deadline and plane will crash
- **System must handle periodic and aperiodic events**
  - E.g., processes A, B, C must be scheduled every 100, 200, 500 msec, require 50, 30, 100 msec respectively
  - *Schedulable* if $\sum \frac{\text{CPU}}{\text{period}} \leq 1$ (not counting switch time)
- **Variety of scheduling strategies**
  - E.g., first deadline first (works if schedulable, otherwise fails spectacularly)

Outline

Scheduling with virtual time

- **Many modern schedulers employ notion of *virtual time***
  - Idea: Equalize virtual CPU time consumed by different processes
  - Higher-priority processes consume virtual time more slowly
- **Forms the basis of the current linux scheduler, CFS**
- **Case study: Borrowed Virtual Time (BVT) [Duda]**
- **BVT runs process with lowest *effective virtual time***
  - A_i – *actual virtual time* consumed by process i
  - *effective virtual time* $E_i = A_i - (\text{warp}_i \; ? \; W_i : 0)$
  - Special warp factor allows borrowing against future CPU time
    …hence name of algorithm

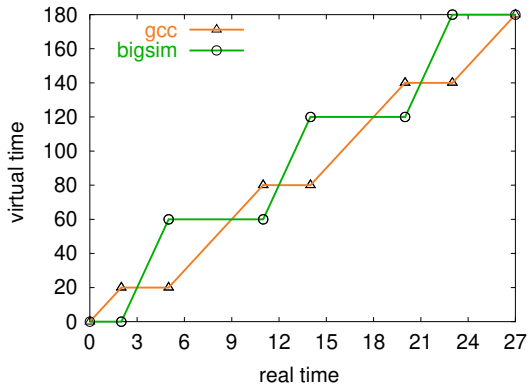
Process weights

- **Each process i's faction of CPU determined by weight w_i**
  - i should get $w_i / \sum_j w_j$ faction of CPU
  - So w_i is real seconds per virtual second that process i has CPU
- **When i consumes t CPU time, track it: A_i += t/w_i**
- **Example: gcc (weight 2), bigsim (weight 1)**
  - Assuming no IO, runs: gcc, gcc, bigsim, gcc, gcc, bigsim, …
  - Lots of context switches, not so good for performance
- **Add in context switch allowance, C**
  - Only switch from i to j if $E_j \leq E_i - C/w_i$
  - C is wall-clock time ($\gg$ context switch cost), so must divide by w_i
  - Ignore C if j just became runable…why?

Process weights

- **Each process i's faction of CPU determined by weight w_i**
  - i should get $w_i / \sum_j w_j$ faction of CPU
  - So w_i is real seconds per virtual second that process i has CPU
- **When i consumes t CPU time, track it: A_i += t/w_i**
- **Example: gcc (weight 2), bigsim (weight 1)**
  - Assuming no IO, runs: gcc, gcc, bigsim, gcc, gcc, bigsim, …
  - Lots of context switches, not so good for performance
- **Add in context switch allowance, C**
  - Only switch from i to j if $E_j \leq E_i - C/w_i$
  - C is wall-clock time ($\gg$ context switch cost), so must divide by w_i
  - Ignore C if j just became runable to avoid affecting response time

BVT example

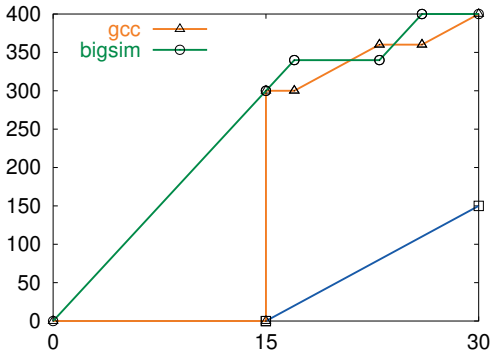


- **gcc has weight 2, bigsim weight 1, $C = 2$, no I/O**
  - bigsim consumes virtual time at twice the rate of gcc
  - Processes run for C time after lines cross before context switch

Sleep/wakeup

- **Must lower priority (increase A_i) after wakeup**
  - Otherwise process with very low A_i would starve everyone
- **Bound lag with Scheduler Virtual Time (SVT)**
  - SVT is minimum A_j for all runnable threads j
  - When waking i from voluntary sleep, set $A_i \leftarrow \max(A_i, SVT)$
- **Note voluntary/involuntary sleep distinction**
  - E.g., Don't reset A_j to SVT after page fault
  - Faulting thread needs a chance to catch up
  - But do set $A_i \leftarrow \max(A_i, SVT)$ after socket read
- **Note: Even with SVT A_i can never decrease**
  - After short sleep, might have $A_i >$ SVT, so $\max(A_i, SVT) = A_i$
  - i never gets more than its fair share of CPU in long run

gcc wakes up after I/O

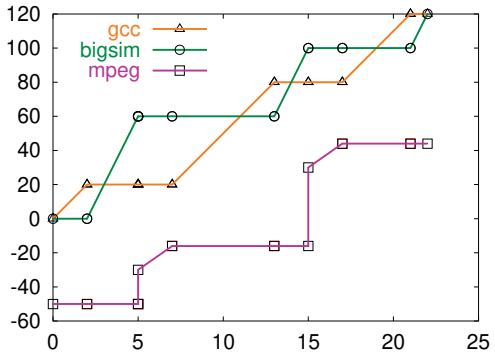


- **gcc's A_i gets reset to SVT on wakeup**
  - Otherwise, would be at lower (blue) line and starve bigsim

Real-time threads

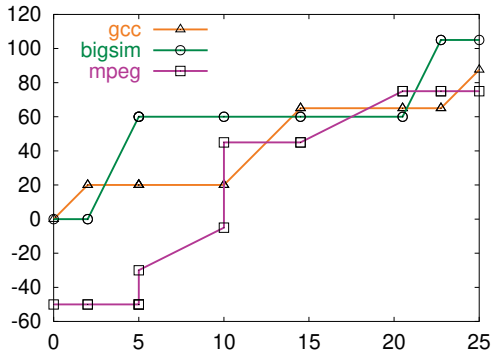
- **Also want to support time-critical tasks**
  - E.g., mpeg player must run every 10 clock ticks
- **Recall $E_i = A_i - (\textbf{warp}_i \; ? \; W_i : 0)$**
  - W_i is *warp factor* – gives thread precedence
  - Just give mpeg player i large W_i factor
  - Will get CPU whenever it is runable
  - But long term CPU share won't exceed $w_i / \sum_j w_j$
- **Note W_i only matters when warp$_i$ is true**
  - Can set warp$_i$ with a syscall, or have it set in signal handler
  - Also gets cleared if i keeps using CPU for L_i time
  - L_i limit gets reset every U_i time
  - $L_i = 0$ means no limit – okay for small W_i value

Running warped



- **mpeg player runs with -50 warp value**
  - Always gets CPU when needed, never misses a frame

Warped thread hogging CPU



- **mpeg goes into tight loop at time 5**
- **Exceeds L_i at time 10, so warp$_i$ $\leftarrow$ false**

BVT example: Search engine

- **Common queries 150 times faster than uncommon**
  - Have 10-thread pool of threads to handle requests
  - Assign W_i value sufficient to process fast query (say 50)
- **Say 1 slow query, small trickle of fast queries**
  - Fast queries come in, warped by 50, execute immediately
  - Slow query runs in background
  - Good for turnaround time
- **Say 1 slow query, but many fast queries**
  - At first, only fast queries run
  - But SVT is bounded by A_i of slow query thread i
  - Recall fast query thread j gets $A_j = max(A_j, SVT) = A_j$; eventually $SVT < A_j$ and a bit later $A_j - W_j > A_i$.
  - At that point thread i will run again, so no starvation

Case study: SMART

- **Key idea: Separate *importance* from *urgency***
  - Figure out which processes are important enough to run
  - Run whichever of these is most urgent
- **Importance = $\langle priority, BVFT \rangle$ value tuple**
  - *priority* – parameter set by user or administrator (higher is better)
    - Takes absolute priority over BVFT
  - *BVFT* – Biased Virtual Finishing Time (lower is better)
    - virtual time consumed + virtual length of next CPU burst
    - I.e., virtual time at which quantum would end if process scheduled now
    - Bias is like negative warp, see paper for details
- **Urgency = next deadline (sooner is more urgent)**

SMART algorithm

- **If most important ready task (ready task with best value tuple) is conventional (not real-time), run it**
- **Consider all real-time tasks with better value tuples than the best ready conventional task**
- **For each such real-time task, starting from the best value-tuple**
  - Can you run it without missing deadlines of more important tasks?
  - If so, add to *schedulable* set
- **Run task with earliest deadline in schedulable set**
- **Send signal to tasks that won't meet their deadlines**

5. Virtual memory HW

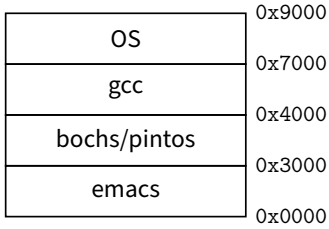
Administrivia

- **Two new CAs: June Lee and Alice Liu**
  - More office hours
- **My office hours moved to 4pm today**
- **Lab 1 due Wednesday 1:30pm (5pm if you attend lecture)**
- **We give will give short extensions to groups that run into trouble. But email us:**
  - How much is done and left?
  - How much longer do you need?
- **Attend section Friday at 1:30pm to learn about lab 2**

Virtual memory

- **Came out of work in late 1960s by Peter Denning (lower right)**
  - Established working set model
  - Led directly to virtual memory

Want processes to co-exist

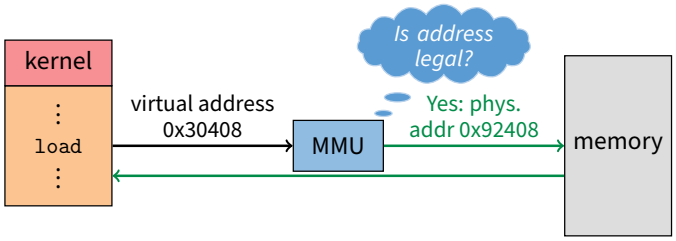


- **Consider multiprogramming on physical memory**
  - What happens if pintos needs to expand?
  - If emacs needs more memory than is on the machine?
  - If pintos has an error and writes to address 0x7100?
  - When does gcc have to know it will run at 0x4000?
  - What if emacs isn't using its memory?

Issues in sharing physical memory

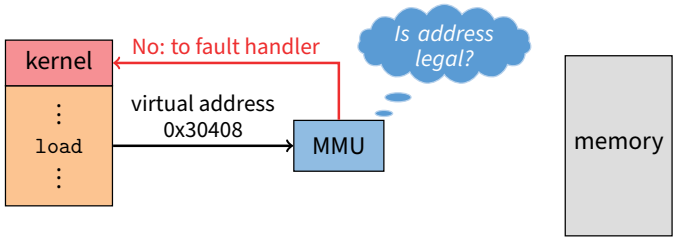
- **Protection**
  - A bug in one process can corrupt memory in another
  - Must somehow prevent process *A* from trashing *B*'s memory
  - Also prevent *A* from even observing *B*'s memory (ssh-agent)
- **Transparency**
  - A process shouldn't require particular physical memory bits
  - Yet processes often require large amounts of contiguous memory (for stack, large data structures, etc.)
- **Resource exhaustion**
  - Programmers typically assume machine has "enough" memory
  - Sum of sizes of all processes often greater than physical memory

Virtual memory goals



- **Give each program its own *virtual* address space**
  - At runtime, *Memory-Management Unit* relocates each load/store
  - Application doesn't see *physical* memory addresses
- **Also enforce protection**
  - Prevent one app from messing with another's memory
- **And allow programs to see more memory than exists**
  - Somehow relocate some memory accesses to disk

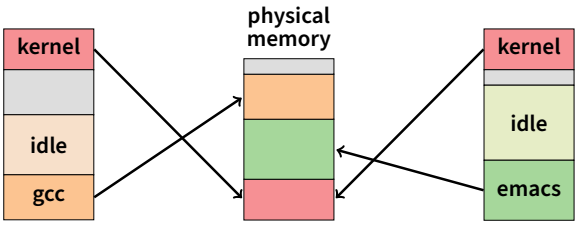
Virtual memory goals



- **Give each program its own *virtual* address space**
  - At runtime, *Memory-Management Unit* relocates each load/store
  - Application doesn't see *physical* memory addresses
- **Also enforce protection**
  - Prevent one app from messing with another's memory
- **And allow programs to see more memory than exists**
  - Somehow relocate some memory accesses to disk

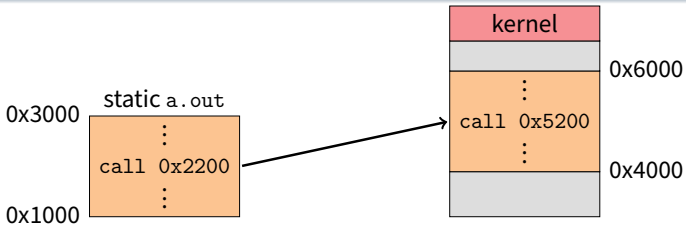
Virtual memory advantages

- **Can re-locate program while running**
  - Run partially in memory, partially on disk
- **Most of a process's memory may be idle (80/20 rule).**



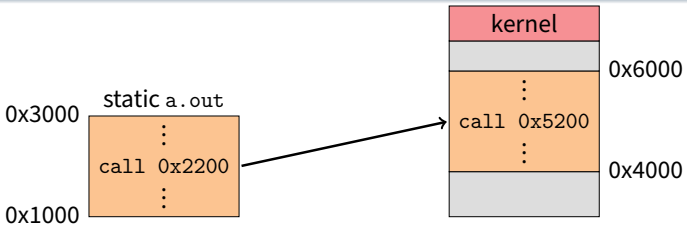
  - Write idle parts to disk until needed
  - Let other processes use memory of idle part
  - Like CPU virtualization: when process not using CPU, switch (Not using a memory region? switch it to another process)
- **Challenge: VM = extra layer, could be slow**

Idea 1: no hardware, load-time linking



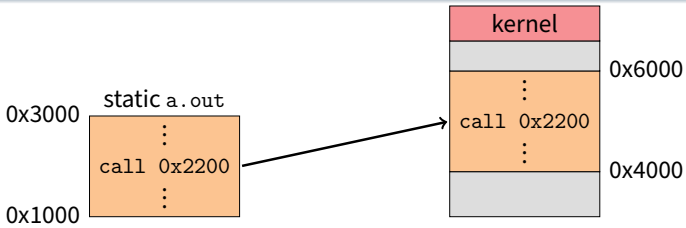
- ***Linker* patches addresses of symbols like `printf`**
- **Idea: link when process executed, not at compile time**
  - Already have PIE (position-independent executable) for security
  - Determine where process will reside in memory at launch
  - Adjust all references within program (using addition)
- **Problems?**

Idea 1: no hardware, load-time linking



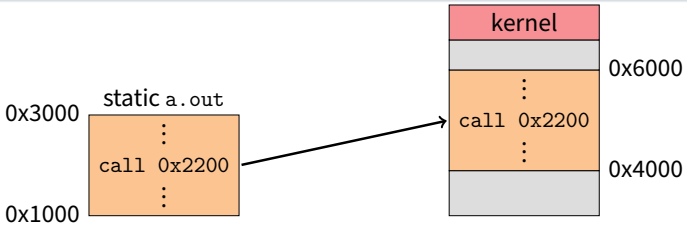
- ***Linker* patches addresses of symbols like `printf`**
- **Idea: link when process executed, not at compile time**
  - Already have PIE (position-independent executable) for security
  - Determine where process will reside in memory at launch
  - Adjust all references within program (using addition)
- **Problems?**
  - How to enforce protection?
  - How to move once already in memory? (consider data pointers)
  - What if no contiguous free region fits program?

Idea 2: base + bound register



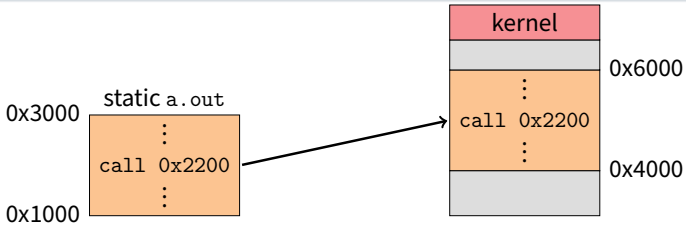
- **Two special privileged registers: base and bound**
- **On each load/store/jump:**
  - Physical address = virtual address + base
  - Check $0 \leq$ virtual address $<$ bound, else trap to kernel
- **How to move process in memory?**
- **What happens on context switch?**

Idea 2: base + bound register



- **Two special privileged registers: base and bound**
- **On each load/store/jump:**
  - Physical address = virtual address + base
  - Check $0 \leq$ virtual address $<$ bound, else trap to kernel
- **How to move process in memory?**
  - Change base register
- **What happens on context switch?**

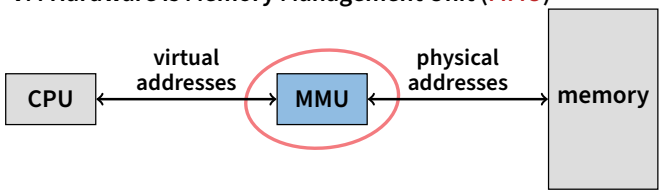
Idea 2: base + bound register



- **Two special privileged registers: base and bound**
- **On each load/store/jump:**
  - Physical address = virtual address + base
  - Check $0 \leq$ virtual address $<$ bound, else trap to kernel
- **How to move process in memory?**
  - Change base register
- **What happens on context switch?**
  - Kernel must re-load base and bound registers

Definitions

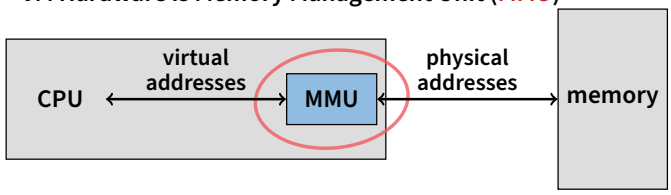
- Programs load/store to virtual addresses
- Actual memory uses physical addresses
- VM Hardware is Memory Management Unit (MMU)



  - Usually part of CPU core (one address space per hyperthread)
  - Configured through privileged instructions (e.g., load bound reg)
  - Translates from virtual to physical addresses
  - Gives per-process view of memory called address space

Definitions

- Programs load/store to virtual addresses
- Actual memory uses physical addresses
- VM Hardware is Memory Management Unit (MMU)



  - Usually part of CPU core (one address space per hyperthread)
  - Configured through privileged instructions (e.g., load bound reg)
  - Translates from virtual to physical addresses
  - Gives per-process view of memory called address space

Base+bound trade-offs

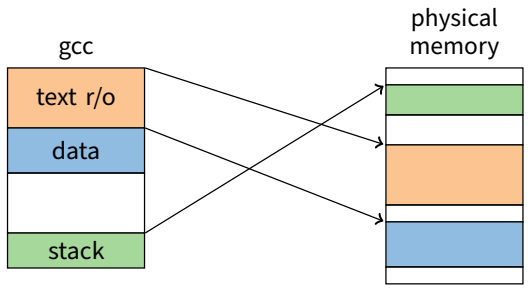
- Advantages
  - Cheap in terms of hardware: only two registers
  - Cheap in terms of cycles: do add and compare in parallel
  - Examples: Cray-1 used this scheme
- Disadvantages

Base+bound trade-offs

- Advantages
  - Cheap in terms of hardware: only two registers
  - Cheap in terms of cycles: do add and compare in parallel
  - Examples: Cray-1 used this scheme

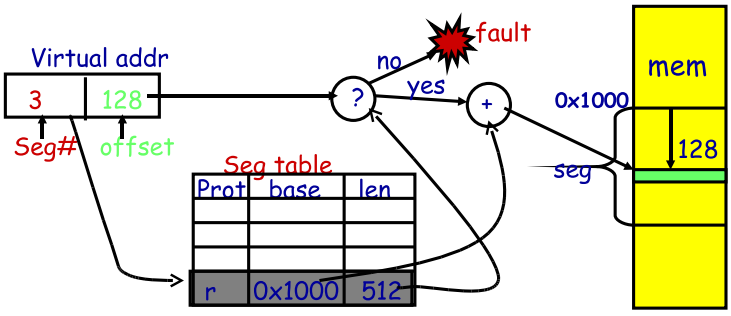
- Disadvantages
  - Growing a process is expensive or impossible
  - No way to share code or data (E.g., two copies of bochs, both running pintos)
- One solution: Multiple segments
  - E.g., separate code, stack, data segments
  - Possibly multiple data segments

Segmentation



- Let processes have many base/bound regs
  - Address space built from many segments
  - Can share/protect memory at segment granularity
- Must specify segment as part of virtual address

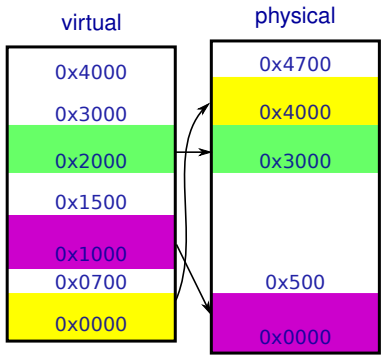
Segmentation mechanics



- Each process has a segment table
- Each VA indicates a segment and offset:
  - Top bits of addr select segment, low bits select offset (PDP-10)
  - Or segment selected by instruction or operand (means you need wider “far” pointers to specify segment)

Segmentation example

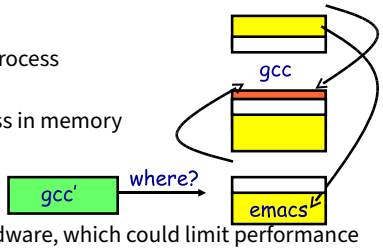
Seg	base	bounds	rw
0	0x4000	0x6ff	10
1	0x0000	0x4ff	11
2	0x3000	0xfff	11
3			00



- **2-bit segment number (1st digit), 12 bit offset (last 3)**
  - Where is 0x0240? 0x1108? 0x265c? 0x3002? 0x1600?

Segmentation trade-offs

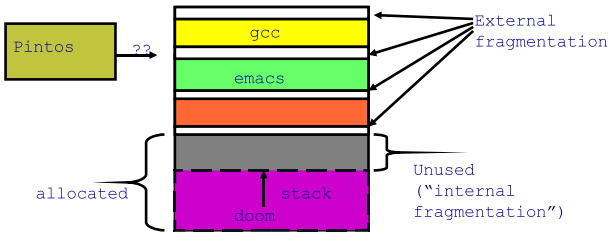
- **Advantages**
  - Multiple segments per process
  - Allows sharing! (how?)
  - Don't need entire process in memory



- **Disadvantages**
  - Requires translation hardware, which could limit performance
  - Segments not completely transparent to program (e.g., default segment faster or uses shorter instruction)
  - *n* byte segment needs *n contiguous* bytes of physical memory
  - Makes *fragmentation* a real problem.

Fragmentation

- **Fragmentation $\Longrightarrow$ Inability to use free memory**
- **Over time:**
  - Variable-sized pieces = many small holes (external fragmentation)
  - Fixed-sized pieces = no external holes, but force internal waste (internal fragmentation)



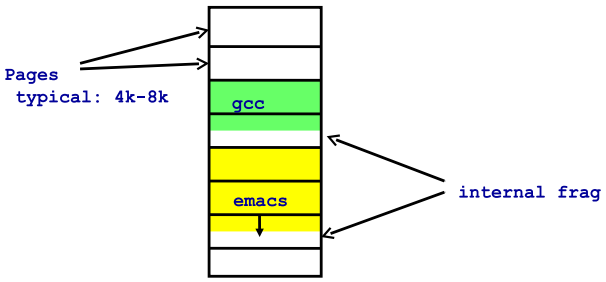
Alternatives to hardware MMU

- **Language-level protection (JavaScript)**
  - Single address space for different modules
  - Language enforces isolation
  - Singularity OS does this with C# [Hunt]
- **Software fault isolation**
  - Instrument compiler output
  - Checks before every store operation prevents modules from trashing each other
  - Google's now deprecated Native Client does this for x86 [Yee]
  - Easier to do for virtual architecture, e.g., Wasm
  - Works really well on ARM64 [Yedidia'24]

Paging

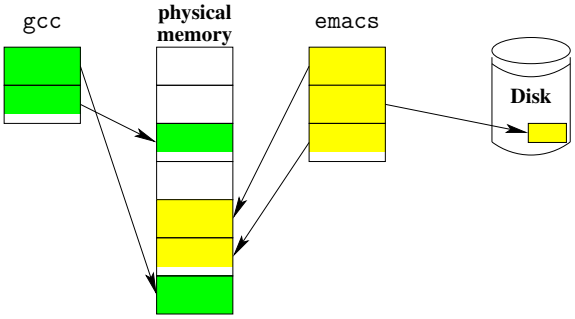
- **Divide memory up into small, equal-size *pages***
- **Map virtual pages to physical pages**
  - Each process has separate mapping
- **Allow OS to gain control on certain operations**
  - Read-only pages trap to OS on write
  - Invalid pages trap to OS on read or write
  - OS can change mapping and resume application
- **Other features sometimes found:**
  - Hardware can set "accessed" and "dirty" bits
  - Control page execute permission separately from read/write
  - Control caching or memory consistency of page

Paging trade-offs



- **Eliminates external fragmentation**
- **Simplifies allocation, free, and backing storage (swap)**
- **Average internal fragmentation of .5 pages per "segment"**

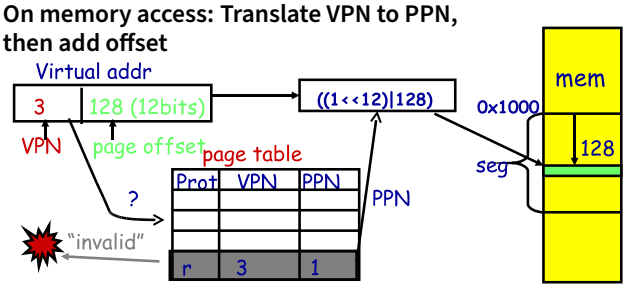
Simplified allocation



- **Allocate any physical page to any process**
- **Can store idle virtual pages on disk**

Paging data structures

- **Pages are fixed size, e.g., 4 KiB**
  - Least significant 12 ($\log_2$ 4 Ki) bits of address are *page offset*
  - Most significant bits are *page number*
- **Each process has a *page table***
  - Maps *virtual page numbers* (VPNs) to *physical page numbers* (PPNs)
  - Also includes bits for protection, validity, etc.
- **On memory access: Translate VPN to PPN, then add offset**



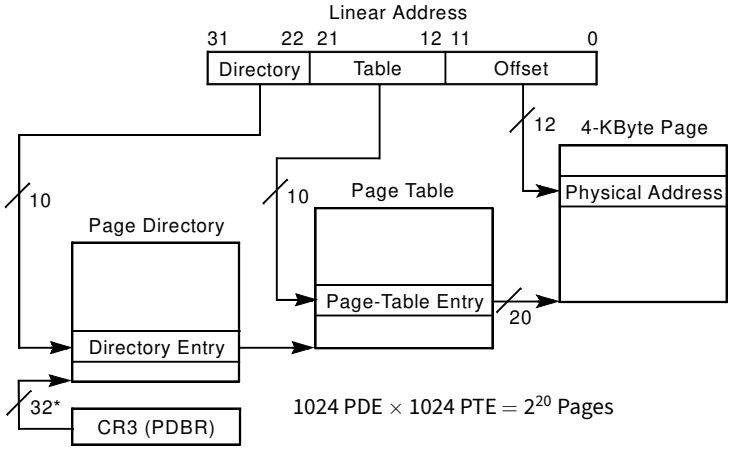
Example: Paging on PDP-11

- **64 KiB virtual memory, 8 KiB pages**
  - Separate address space for instructions & data
  - I.e., can't read your own instructions with a load
- **Entire page table stored in registers**
  - 8 Instruction page translation registers
  - 8 Data page translations
- **Swap 16 machine registers on each context switch**

x86 Paging

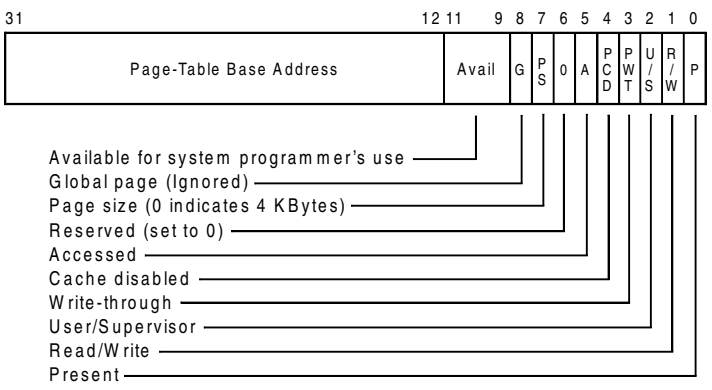
- **Paging enabled by bits in a control register (`%cr0`)**
  - Only privileged OS code can manipulate control registers
- **Normally 4 KiB pages**
- **`%cr3`: points to physical address of 4 KiB page directory**
  - See `pagedir_activate` in Pintos
- **Page directory: 1024 PDEs (page directory entries)**
  - Each contains physical address of a page table
- **Page table: 1024 PTEs (page table entries)**
  - Each contains physical address of virtual 4K page
  - Page table covers 4 MiB of Virtual mem
- **See old intel manual for simplest explanation**
  - Also volume 2 of AMD64 Architecture docs
  - Also volume 3A of latest intel 64 architecture manual

x86 page translation

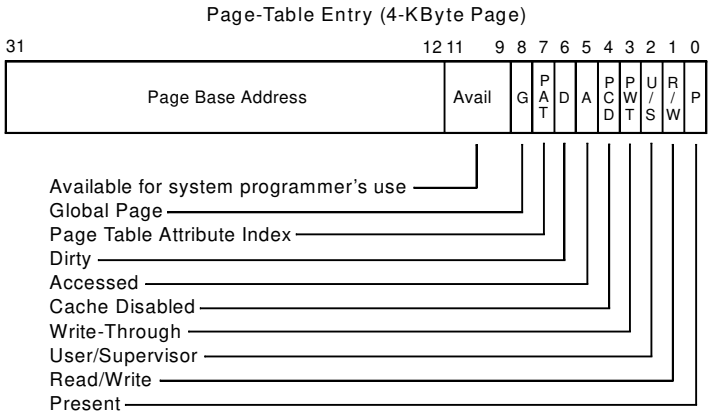


*32 bits aligned onto a 4-KByte boundary

x86 page directory entry



x86 page table entry



x86 hardware segmentation

- **x86 architecture *also* supports segmentation**
  - Segment register base + pointer val = *linear address*
  - Page translation happens on linear addresses
- **Two levels of protection and translation check**
  - Segmentation model has four privilege levels (CPL 0–3)
  - Paging only two, so 0–2 = kernel, 3 = user
- **Why do you want *both* paging and segmentation?**

x86 hardware segmentation

- **x86 architecture *also* supports segmentation**
  - Segment register base + pointer val = *linear address*
  - Page translation happens on linear addresses
- **Two levels of protection and translation check**
  - Segmentation model has four privilege levels (CPL 0–3)
  - Paging only two, so 0–2 = kernel, 3 = user
- **Why do you want *both* paging and segmentation?**
- **Short answer: You don't – just adds overhead**
  - Most OSes use "flat mode" – set base = 0, bounds = 0xffffffff in all segment registers, then forget about it
  - x86-64 architecture removes much segmentation support
- **Long answer: Has some fringe/incidental uses**
  - Keep pointer to thread-local storage w/o wasting normal register
  - 32-bit VMware runs guest OS in CPL 1 to trap stack faults
  - OpenBSD used CS limit for W$\wedge$X when no PTE NX bit

Making paging fast

- **x86 PTs require 3 memory references per load/store**
  - Look up page table address in page directory
  - Look up physical page number (PPN) in page table
  - Actually access physical page corresponding to virtual address
- **For speed, CPU caches recently used translations**
  - Called a *translation lookaside buffer* or TLB
  - Typical: 64-2k entries, 4-way to fully associative, 95% hit rate
  - Modern CPUs add second-level TLB with $\sim$1,024+ entries; often separate instruction and data TLBs
  - Each TLB entry maps a VPN $\rightarrow$ PPN + protection information
- **On each memory reference**
  - Check TLB, if entry present get physical address fast
  - If not, walk page tables, insert in TLB for next time (Must evict some entry)

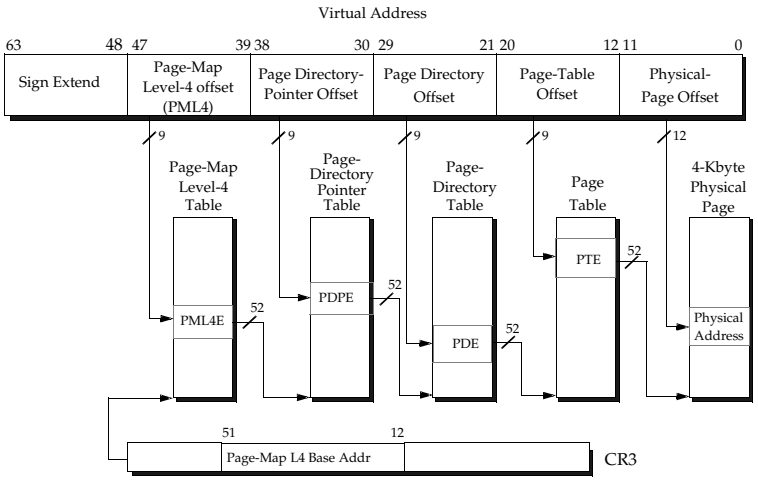
TLB details

- **TLB operates at CPU pipeline speed $\Longrightarrow$ small, fast**
- **Complication: what to do when switching address space?**
  - Flush TLB on context switch (e.g., old x86)
  - Tag each entry with associated process's ID (e.g., MIPS)
- **In general, OS must manually keep TLB valid**
  - Changing page table in memory won't affect cached TLB entry
- **E.g., on x86 must use *invlpg* instruction**
  - Invalidates a page translation in TLB
  - Note: very expensive instruction (100–200 cycles)
  - Must execute after changing a possibly used page table entry
  - Otherwise, hardware will miss page table change
- **More Complex on a multiprocessor (TLB shootdown)**
  - Requires sending an interprocessor interrupt (IPI)
  - Remote processor must execute `invlpg` instruction

x86 Paging Extensions

- **PSE: Page size extensions**
  - Setting bit 7 in PDE makes a 4 MiB translation (no PT)
- **PAE Page address extensions**
  - Newer 64-bit PTE format allows 36+ bits of physical address
  - Page tables, directories have only 512 entries
  - Use 4-entry Page-Directory-Pointer Table to regain 2 lost bits
  - PDE bit 7 allows 2 MiB translation
- **Long mode PAE (x86-64)**
  - In Long mode, pointers are 64-bits
  - Extends PAE to map 48 bits of virtual address (next slide)
  - Why are aren't all 64 bits of VA usable?

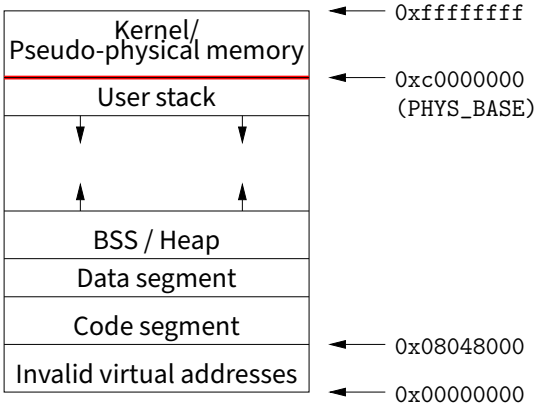
x86 long mode paging



Where does the OS live?

- **In its own address space?**
  - Can't do this on most hardware (e.g., syscall instruction won't switch address spaces)
  - Also would make it harder to parse syscall arguments passed as pointers
- **So in the same address space as process**
  - Use protection bits to prohibit user code from writing kernel
- **Typically all kernel text, most data at same VA in every address space**
  - On x86, must manually set up page tables for this
  - Usually just map kernel in contiguous virtual memory when boot loader puts kernel into contiguous physical memory
  - Some hardware puts physical memory (kernel-only) somewhere in virtual address space
  - Typically kernel goes in high memory; with signed numbers, can mean small negative addresses (small linker relocations)

Pintos memory layout



Very different MMU: MIPS

- **Hardware checks TLB on application load/store**
  - References to addresses not in TLB trap to kernel
- **Each TLB entry has the following fields: Virtual page, Pid, Page frame, NC, D, V, Global**
- **Kernel itself unpaged**
  - All of physical memory contiguously mapped in high VM (hardwired in CPU, not just by convention as with Pintos)
  - Kernel uses these pseudo-physical addresses
- **User TLB fault hander very efficient**
  - Two hardware registers reserved for it
  - utlb miss handler can itself fault—allow paged page tables
- **OS is free to choose page table format!**

DEC Alpha MMU

- **Firmware managed TLB**
  - Like MIPS, TLB misses handled by software
  - Unlike MIPS, TLB miss routines ship with machine in ROM (but copied to main memory on boot—so can be overwritten)
  - Firmware known as "PAL code" (privileged architecture library)
- **Hardware capabilities**
  - 8 KiB, 64 KiB, 512 KiB, 4 MiB pages all available
  - TLB supports 128 instruction/128 data entries of any size
- **Various other events vector directly to PAL code**
  - `call_pal` instruction, TLB miss/fault, FP disabled
- **PAL code runs in special privileged processor mode**
  - Interrupts always disabled
  - Have access to special instructions and registers

PAL code interface details

- **Examples of Digital Unix PALcode entry functions**
  - `callsys`/`retsys` - make, return from system call
  - `swpctx` - change address spaces
  - `wrvptptr` - write virtual page table pointer
  - `tbi` - TLB invalidate
- **Some fields in PALcode page table entries**
  - GH - 2-bit granularity hint $\rightarrow 2^N$ pages have same translation
  - ASM - address space match $\rightarrow$ mapping applies in all processes

Example: Paging to disk

- `gcc` **needs a new page of memory**
- **OS re-claims an idle page from** `emacs`
- **If page is *clean* (i.e., also stored on disk):**
  - E.g., page of text from emacs binary on disk
  - Can always re-read same page from binary
  - So okay to discard contents now & give page to `gcc`
- **If page is *dirty* (meaning memory is only copy)**
  - Must write page to disk first before giving to `gcc`
- **Either way:**
  - Mark page invalid in `emacs`
  - `emacs` will fault on next access to virtual page
  - On fault, OS reads page data back from disk into new page, maps new page into `emacs`, resumes executing

Paging in day-to-day use

- **Demand paging**
- **Growing the stack**
- **BSS page allocation**
- **Shared text**
- **Shared libraries**
- **Shared memory**
- **Copy-on-write (**`fork`**,** `mmap`**, etc.)**
- **Q: Which pages should have global bit set on x86?**

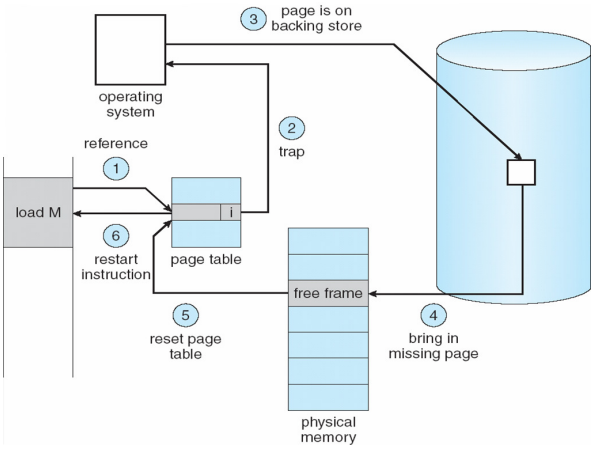
6. Virtual memory OS

Reminder

- **Attend section Friday 1:30pm, NVIDIA auditorium**

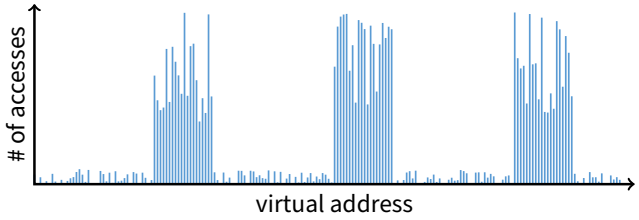
Outline

Paging



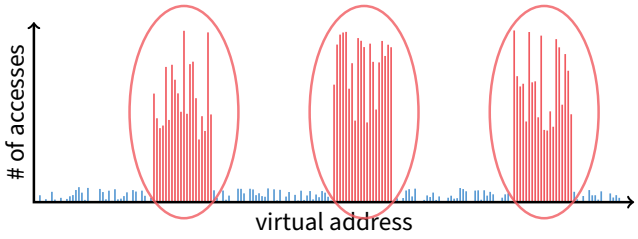
- **Use disk to simulate larger virtual than physical mem**

Working set model



- **Disk much, much slower than memory**
  - Goal: run at memory speed, not disk speed
- **80/20 rule: 20% of memory gets 80% of memory accesses**
  - Keep the hot 20% in memory
  - Keep the cold 80% on disk

Working set model



- **Disk much, much slower than memory**
  - Goal: run at memory speed, not disk speed
- **80/20 rule: 20% of memory gets 80% of memory accesses**

→ Keep the hot 20% in memory
  - Keep the cold 80% on disk

Working set model

- **Disk much, much slower than memory**
  - Goal: run at memory speed, not disk speed
- **80/20 rule: 20% of memory gets 80% of memory accesses**
  - Keep the hot 20% in memory

→ Keep the cold 80% on disk

Paging challenges

- **How to resume a process after a fault?**
  - Need to save state and resume
  - Process may have been in the middle of an instruction!
- **What to fetch from disk?**
  - Just needed page or more?
- **What to eject?**
  - How to allocate physical pages amongst processes?
  - Which of a particular process's pages to keep in memory?

Re-starting instructions

- **Hardware must allow resuming after a fault**
- **Hardware provides kernel with information about page fault**
  - Faulting virtual address (In `%cr2` reg on x86—may see it if you modify Pintos `page_fault` and use `fault_addr`)
  - Address of instruction that caused fault
  - Was the access a read or write? Was it an instruction fetch? Was it caused by user access to kernel-only memory?
- **Observation: Idempotent instructions are easy to restart**
  - E.g., simple load or store instruction can be restarted
  - Just re-execute any instruction that only accesses one address
- **Complex instructions must be re-started, too**
  - E.g., x86 move string instructions
  - Specify src, dst, count in `%esi`, `%edi`, `%ecx` registers
  - On fault, registers adjusted to resume where move left off

What to fetch

- **Bring in page that caused page fault**
- **Pre-fetch surrounding pages?**
  - Reading two disk blocks approximately as fast as reading one
  - As long as no track/head switch, seek time dominates
  - If application exhibits spacial locality, then big win to store and read multiple contiguous pages
- **Also pre-zero unused pages in idle loop**
  - Need 0-filled pages for stack, heap, anonymously mmapped memory
  - Zeroing them only on demand is slower
  - Hence, many OSes zero freed pages while CPU is idle

Selecting physical pages

- **May need to eject some pages**
  - More on eviction policy in two slides
- **May also have a choice of physical pages**
- **Direct-mapped physical caches (older machines)**
  - Physical address A conflicts with $kC + A$ (where k is any integer, C is cache size)
  - Virtual $\rightarrow$ Physical mapping can affect performance
  - Applications can conflict with each other or themselves
  - Scientific applications benefit if consecutive virtual pages do not conflict in the cache
  - Many other applications do better with random mapping
- **Set associative caches (more common)**
  - Multiple (e.g., 2–4) possible slots for each physical address
  - Historically n-way associative cache chooses line by $A \bmod (C/n)$
  - These days: CPUs use more sophisticated mapping [Hund]

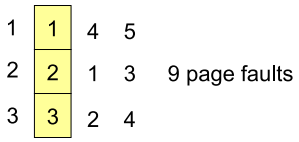
Superpages

- **How should OS make use of "large" mappings**
  - x86 has 2/4MiB pages that might be useful
  - Alpha has even more choices: 8KiB, 64KiB, 512KiB, 4MiB
- **Sometimes more pages in L2 cache than TLB entries**
  - Don't want costly TLB misses going to main memory
  - Try cpuid tool to find CPU's TLB configuration on linux… then compare to cache size reported by lscpu
- **Or have two-level TLBs**
  - Want to maximize hit rate in faster L1 TLB
- **OS can transparently support superpages [Navarro]**
  - "Reserve" appropriate physical pages if possible
  - Promote contiguous pages to superpages
  - Does complicate evicting (esp. dirty pages) – demote

Outline

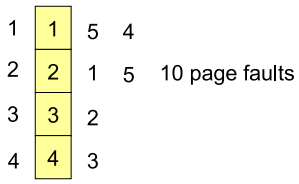
Straw man: FIFO eviction

- Evict oldest fetched page in system
- Example—reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 physical pages: 9 page faults

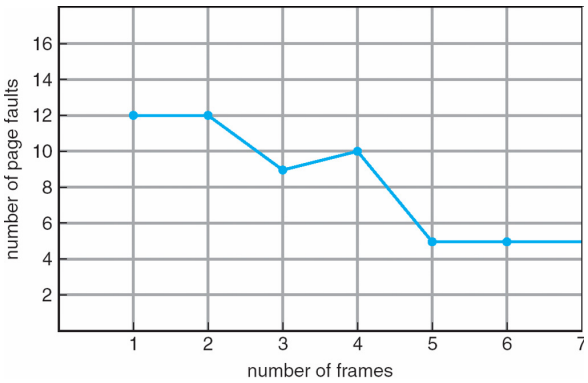


Straw man: FIFO eviction

- Evict oldest fetched page in system
- Example—reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 physical pages: 9 page faults
- **4 physical pages: 10 page faults**



Belady's Anomaly



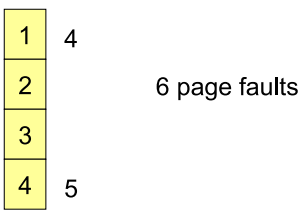
- More physical memory doesn't always mean fewer faults

Optimal page replacement

- What is optimal (if you knew the future)?

Optimal page replacement

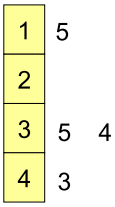
- What is optimal (if you knew the future)?
  - Replace page that will not be used for longest period of time
- Example—reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- With 4 physical pages:



- What do we do when an OS can't predict the future?

LRU page replacement

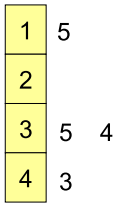
- Approximate optimal with *least recently used*
  - Because past often predicts the future
- Example—reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- With 4 physical pages: 8 page faults



- Problem 1: Can be pessimal – example?
- Problem 2: How to implement?

LRU page replacement

- **Approximate optimal with *least recently used***
  - Because past often predicts the future
- **Example—reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**
- **With 4 physical pages: 8 page faults**



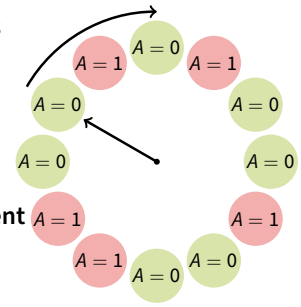
- **Problem 1: Can be pessimal – example?**
  - Looping over memory (then want MRU eviction)
- **Problem 2: How to implement?**

Straw man LRU implementations

- **Stamp PTEs with timer value**
  - E.g., CPU has cycle counter
  - Automatically writes value to PTE on each page access
  - Scan page table to find oldest counter value = LRU page
  - Problem: Would double memory traffic!
- **Keep doubly-linked list of pages**
  - On access remove page, place at tail of list
  - Problem: again, very expensive
- **What to do?**
  - Just approximate LRU, don't try to do it exactly

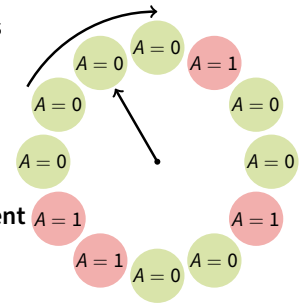
Clock algorithm

- **Use accessed bit supported by most hardware**
  - E.g., x86 will write 1 to A bit in PTE on first access
  - Software managed TLBs like MIPS can do the same
- **Do FIFO but skip accessed pages**
- **Keep pages in circular FIFO list**
- **Scan:**
  - page's A bit = 1, set to 0 & skip
  - else if A = 0, evict
- **A.k.a. second-chance replacement**



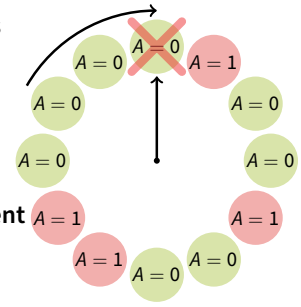
Clock algorithm

- **Use accessed bit supported by most hardware**
  - E.g., x86 will write 1 to A bit in PTE on first access
  - Software managed TLBs like MIPS can do the same
- **Do FIFO but skip accessed pages**
- **Keep pages in circular FIFO list**
- **Scan:**
  - page's A bit = 1, set to 0 & skip
  - else if A = 0, evict
- **A.k.a. second-chance replacement**



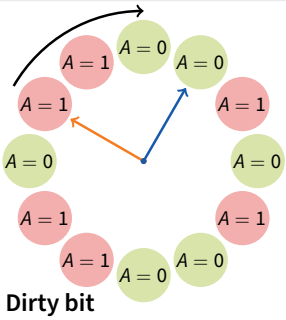
Clock algorithm

- **Use accessed bit supported by most hardware**
  - E.g., x86 will write 1 to A bit in PTE on first access
  - Software managed TLBs like MIPS can do the same
- **Do FIFO but skip accessed pages**
- **Keep pages in circular FIFO list**
- **Scan:**
  - page's A bit = 1, set to 0 & skip
  - else if A = 0, evict
- **A.k.a. second-chance replacement**



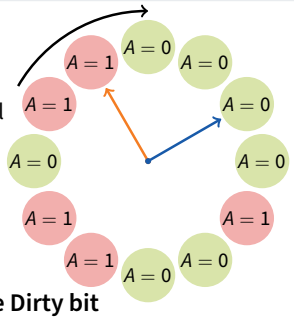
Clock algorithm (continued)

- **Large memory may be a problem**
  - Most pages referenced in long interval
- **Add a second clock hand**
  - Two hands move in lockstep
  - Leading hand clears A bits
  - Trailing hand evicts pages with A=0



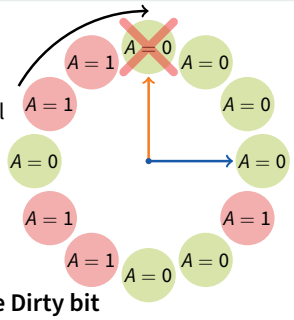
- **Can also take advantage of hardware Dirty bit**
  - Each page can be (Unaccessed, Clean), (Unaccessed, Dirty), (Accessed, Clean), or (Accessed, Dirty)
  - Consider clean pages for eviction before dirty
- **Or use *n*-bit accessed *count* instead just *A* bit**
  - On sweep: $count = (A \ll (n-1)) \mid (count \gg 1)$
  - Evict page with lowest *count*

Clock algorithm (continued)



- **Large memory may be a problem**
  - Most pages referenced in long interval
- **Add a second clock hand**
  - Two hands move in lockstep
  - Leading hand clears A bits
  - Trailing hand evicts pages with A=0
- **Can also take advantage of hardware Dirty bit**
  - Each page can be (Unaccessed, Clean), (Unaccessed, Dirty), (Accessed, Clean), or (Accessed, Dirty)
  - Consider clean pages for eviction before dirty
- **Or use *n*-bit accessed *count* instead just *A* bit**
  - On sweep: $count = (A \ll (n-1)) \mid (count \gg 1)$
  - Evict page with lowest *count*

Clock algorithm (continued)

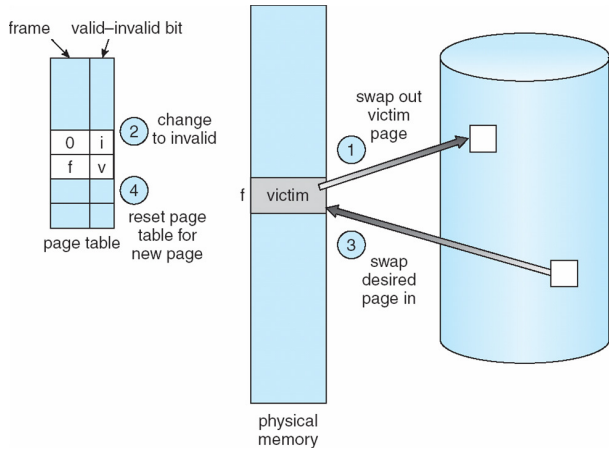


- **Large memory may be a problem**
  - Most pages referenced in long interval
- **Add a second clock hand**
  - Two hands move in lockstep
  - Leading hand clears A bits
  - Trailing hand evicts pages with A=0
- **Can also take advantage of hardware Dirty bit**
  - Each page can be (Unaccessed, Clean), (Unaccessed, Dirty), (Accessed, Clean), or (Accessed, Dirty)
  - Consider clean pages for eviction before dirty
- **Or use *n*-bit accessed *count* instead just *A* bit**
  - On sweep: $count = (A \ll (n-1)) \mid (count \gg 1)$
  - Evict page with lowest *count*

Other replacement algorithms

- **Random eviction**
  - Dirt simple to implement
  - Not overly horrible (avoids Belady & pathological cases)
- ***LFU* (least frequently used) eviction**
  - Instead of just A bit, count # times each page accessed
  - Least frequently accessed must not be very useful (or maybe was just brought in and is about to be used)
  - Decay usage counts over time (for pages that fall out of usage)
- ***MFU* (most frequently used) algorithm**
  - Because page with the smallest count was probably just brought in and has yet to be used
- **Neither LFU nor MFU used very commonly**

Naïve paging



- **Naïve page replacement: 2 disk I/Os per page fault**

Page buffering

- **Idea: reduce # of I/Os on the critical path**
- **Keep pool of free page frames**
  - On fault, still select victim page to evict
  - But read fetched page into already free page
  - Can resume execution while writing out victim page
  - Then add victim page to free pool
- **Can also yank pages back from free pool**
  - Contains only clean pages, but may still have data
  - If page fault on page still in free pool, recycle

Page allocation

- **Allocation can be *global* or *local***
- **Global allocation doesn't consider page ownership**
  - E.g., with LRU, evict least recently used page of any proc
  - Works well if P_1 needs 20% of memory and P_2 needs 70%:

  - Doesn't protect you from memory pigs (imagine P_2 keeps looping through array that is size of mem)
- **Local allocation isolates processes (or users)**
  - Separately determine how much memory each process should have
  - Then use LRU/clock/etc. to determine which pages to evict within each process

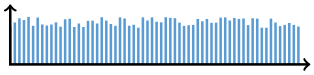
Outline

Thrashing

- **Processes require more memory than system has**
  - Each time one page is brought in, another page, whose contents will soon be referenced, is thrown out
  - Processes will spend all of their time blocked, waiting for pages to be fetched from disk
  - Disk at 100% utilization, but system not getting much useful work done
- **What we wanted: virtual memory the size of disk with access time the speed of physical memory**
- **What we got: memory with access time of disk**

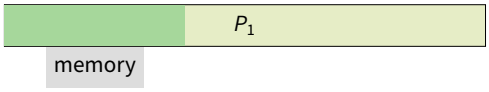
Reasons for thrashing

- **Access pattern has no temporal locality (past $\neq$ future)**

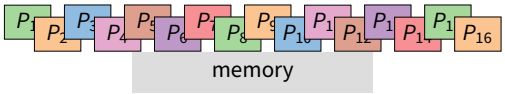


(80/20 rule has broken down)

- **Hot memory does not fit in physical memory**

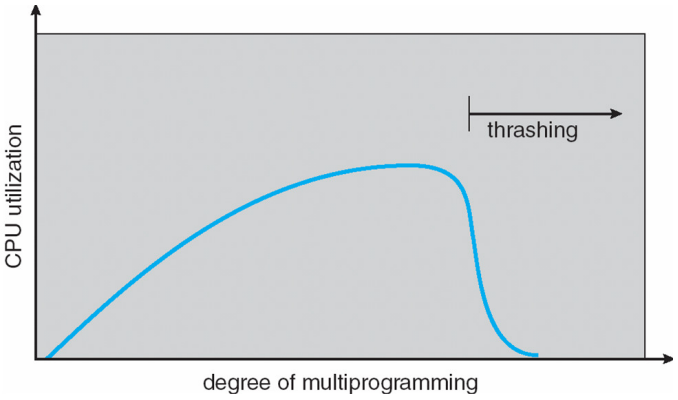


- **Each process fits individually, but too many for system**



  - At least this case is possible to address

Multiprogramming & Thrashing

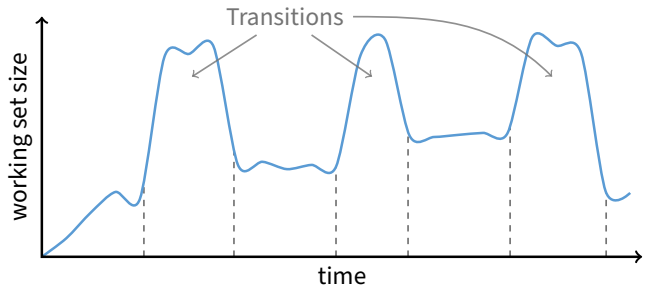


- **Must shed load when thrashing**

Dealing with thrashing

- **Approach 1: working set**
  - Thrashing viewed from a caching perspective: given locality of reference, how big a cache does the process need?
  - Or: how much memory does the process need in order to make reasonable progress (its working set)?
  - Only run processes whose memory requirements can be satisfied
- **Approach 2: page fault frequency**
  - Thrashing viewed as poor ratio of fetch to work
  - PFF = page faults / instructions executed
  - If PFF rises above threshold, process needs more memory. Not enough memory on the system? Swap out.
  - If PFF sinks below threshold, memory can be taken away

Working sets



- **Working set changes across phases**
  - Baloons during phase transitions

Calculating the working set

- **Working set: all pages that process will access in next T time**
  - Can't calculate without predicting future
- **Approximate by assuming past predicts future**
  - So working set $\approx$ pages accessed in last T time
- **Keep idle time for each page**
- **Periodically scan all resident pages in system**
  - **A** bit set? Clear it and clear the page's idle time
  - **A** bit clear? Add CPU consumed since last scan to idle time
  - Working set is pages with idle time $< T$

Two-level scheduler

- **Divide processes into *active* & *inactive***
  - Active – means working set resident in memory
  - Inactive – working set intentionally not loaded
- **Balance set: union of all active working sets**
  - Must keep balance set smaller than physical memory
- **Use long-term scheduler [recall from lecture 4]**
  - Moves procs active $\rightarrow$ inactive until balance set small enough
  - Periodically allows inactive to become active
  - As working set changes, must update balance set
- **Complications**
  - How to chose idle time threshold T?
  - How to pick processes for active set
  - How to count shared memory (e.g., libc.so)

Outline

Some complications of paging

- **What happens to available memory?**
  - Some physical memory tied up by kernel VM structures
- **What happens to user/kernel crossings?**
  - More crossings into kernel
  - Pointers in syscall arguments must be checked
    (can't just kill process if page not present—might need to page in)
- **What happens to IPC?**
  - Must change hardware address space
  - Increases TLB misses
  - Context switch flushes TLB entirely on old x86 machines
    (But not on MIPS…Why?)

Some complications of paging

- **What happens to available memory?**
  - Some physical memory tied up by kernel VM structures
- **What happens to user/kernel crossings?**
  - More crossings into kernel
  - Pointers in syscall arguments must be checked
    (can't just kill process if page not present—might need to page in)
- **What happens to IPC?**
  - Must change hardware address space
  - Increases TLB misses
  - Context switch flushes TLB entirely on old x86 machines
    (But not on MIPS…Why? MIPS tags TLB entries with PID)

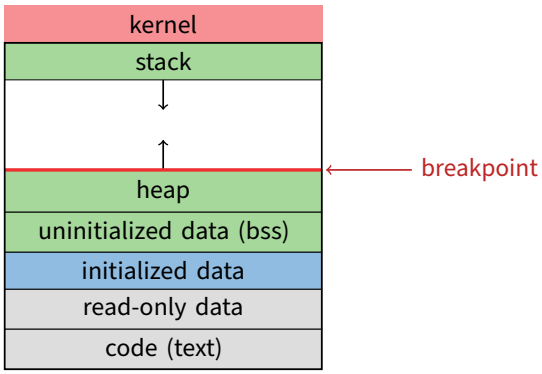
64-bit address spaces

- **Recall x86-64 only has 48-bit virtual address space**
- **What if you want a 64-bit virtual address space?**
  - Straight hierarchical page tables not efficient
  - But software TLBs (like MIPS) allow other possibilities
- **Solution 1: Hashed page tables**
  - Store Virtual $\rightarrow$ Physical translations in hash table
  - Table size proportional to physical memory
  - Clustering makes this more efficient [Talluri]
- **Solution 2: Guarded page tables [Liedtke]**
  - Omit intermediary tables with only one entry
  - Add predicate in high level tables, stating the only virtual address range mapped underneath + # bits to skip

Outline

1. Paging
2. Eviction policies
3. Thrashing
4. Details of paging
5. The user-level perspective
6. Case study: 4.4 BSD

Recall typical virtual address space

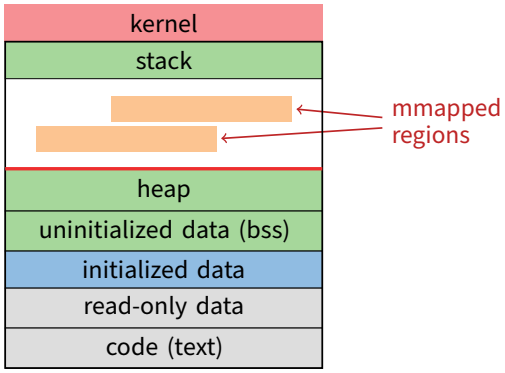


- **Dynamically allocated memory goes in heap**
- **Top of heap called *breakpoint***
  - Addresses between breakpoint and stack all invalid

Early VM system calls

- **OS keeps "Breakpoint" – top of heap**
  - Memory regions between breakpoint & stack fault on access
- `char *brk (const char *addr);`
  - Set and return new value of breakpoint
- `char *sbrk (int incr);`
  - Increment value of the breakpoint & return old value
- **Can implement `malloc` in terms of `sbrk`**
  - But hard to "give back" physical memory to system

Memory mapped files



- **Other memory objects between heap and stack**

`mmap` system call

- `void *mmap (void *addr, size_t len, int prot, int flags, int fd, off_t offset)`
  - Map file specified by `fd` at virtual address `addr`
  - If `addr` is NULL, let kernel choose the address
- `prot` – **protection of region**
  - OR of PROT_EXEC, PROT_READ, PROT_WRITE, PROT_NONE
- `flags`
  - MAP_ANON – anonymous memory (`fd` should be -1)
  - MAP_PRIVATE – modifications are private
  - MAP_SHARED – modifications seen by everyone

More VM system calls

- `int msync(void *addr, size_t len, int flags);`
  - Flush changes of mmapped file to backing store
- `int munmap(void *addr, size_t len)`
  - Removes memory-mapped object
- `int mprotect(void *addr, size_t len, int prot)`
  - Changes protection on pages to bitwise or of some `PROT_...` values
- `int mincore(void *addr, size_t len, char *vec)`
  - Returns in `vec` which pages present

Exposing page faults

```
struct sigaction {
  union {              /* signal handler */
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
  };
  sigset_t sa_mask;    /* signal mask to apply */
  int sa_flags;
};

int sigaction (int sig, const struct sigaction *act,
               struct sigaction *oact)
```

- **Can specify function to run on `SIGSEGV` (Unix signal raised on invalid memory access)**

Example: OpenBSD/i386 siginfo

```
struct sigcontext {
  int sc_gs; int sc_fs; int sc_es; int sc_ds;
  int sc_edi; int sc_esi; int sc_ebp; int sc_ebx;
  int sc_edx; int sc_ecx; int sc_eax;

  int sc_eip; int sc_cs;  /* instruction pointer */
  int sc_eflags;          /* condition codes, etc. */
  int sc_esp; int sc_ss;  /* stack pointer */

  int sc_onstack;         /* sigstack state to restore */
  int sc_mask;            /* signal mask to restore */

  int sc_trapno;
  int sc_err;
};
```

- **Linux uses `ucontext_t` – same idea, just uses nested structures that won't all fit on one slide**

VM tricks at user level

- **Combination of `mprotect`/`sigaction` very powerful**
  - Can use OS VM tricks in user-level programs [Appel]
  - E.g., fault, unprotect page, return from signal handler
- **Technique used in object-oriented databases**
  - Bring in objects on demand
  - Keep track of which objects may be dirty
  - Manage memory as a cache for much larger object DB
- **Other interesting applications**
  - Useful for some garbage collection algorithms
  - Snapshot processes (copy on write)

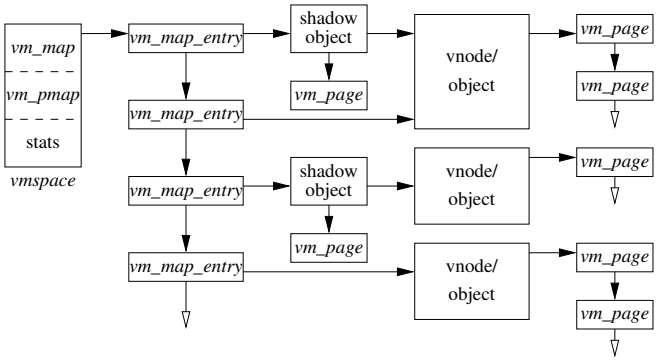
Outline

4.4 BSD VM system [McKusick][1]

- **Each process has a *vmspace* structure containing**
  - *vm_map* – machine-independent virtual address space
  - *vm_pmap* – machine-dependent data structures
  - statistics – e.g., for syscalls like *getrusage ()*
- ***vm_map* is a linked list of *vm_map_entry* structs**
  - *vm_map_entry* covers contiguous virtual memory
  - points to *vm_object* struct
- ***vm_object* is source of data**
  - e.g. vnode object for memory mapped file
  - points to list of *vm_page* structs (one per mapped page)
  - *shadow objects* point to other objects for copy on write

[1] Use link on searchworks page for access

4.4 BSD VM data structures



Pmap (machine-dependent) layer

- **Pmap layer holds architecture-specific VM code**
- **VM layer invokes pmap layer**
  - On page faults to install mappings
  - To protect or unmap pages
  - To ask for dirty/accessed bits
- **Pmap layer is lazy and can discard mappings**
  - No need to notify VM layer
  - Process will fault and VM layer must reinstall mapping
- **Pmap handles restrictions imposed by cache**

Example uses

- ***vm_map_entry* structs for a process**
  - r/o text segment → file object
  - r/w data segment → shadow object → file object
  - r/w stack → anonymous object
- **New *vm_map_entry* objects after a fork:**
  - Share text segment directly (read-only)
  - Share data through two new shadow objects (must share pre-fork but not post-fork changes)
  - Share stack through two new shadow objects
- **Must discard/collapse superfluous shadows**
  - E.g., when child process exits

What happens on a fault?

- **Traverse *vm_map_entry* list to get appropriate entry**
  - No entry? Protection violation? Send process a SIGSEGV
- **Traverse list of [shadow] objects**
- **For each object, traverse *vm_page* structs**
- **Found a *vm_page* for this object?**
  - If first *vm_object* in chain, map page
  - If read fault, install page read only
  - Else if write fault, install copy of page
- **Else get page from object**
  - Page in from file, zero-fill new page, etc.

Paging in day-to-day use

- **Demand paging**
  - Read pages from *vm_object* of executable file
- **Copy-on-write (`fork`, `mmap`, etc.)**
  - Use shadow objects
- **Growing the stack, BSS page allocation**
  - A bit like copy-on-write for `/dev/zero`
  - Can have a single read-only zero page for reading
  - Special-case write handling with pre-zeroed pages
- **Shared text, shared libraries**
  - Share *vm_object* (shadow will be empty where read-only)
- **Shared memory**
  - Two processes `mmap` same file, have same *vm_object* (no shadow)

7. Synchronization 1

Outline

1. Cache coherence – the hardware view
2. Synchronization and memory consistency review
3. C11 Atomics
4. Avoiding locks

Important memory system properties

- **Coherence – concerns accesses to a single memory location**
  - There is a total order on all updates
  - Must obey program order if access from only one CPU
  - There is bounded latency before everyone sees a write
- **Consistency – concerns ordering across memory locations**
  - Even with coherence, different CPUs can see the same write happen at different times
  - Sequential consistency is what matches our intuition (As if operations from all CPUs interleaved on one CPU)
  - Many architectures offer weaker consistency
  - Yet well-defined weaker consistency can still be sufficient to implement thread API contract from concurrency lecture

Multicore cache coherence

- **Performance requires caches**
  - Divided into chuncks of bytes called lines (e.g., 64 bytes)
  - Caches create an opportunity for cores to disagree about memory
- **Bus-based approaches**
  - "Snoopy" protocols, each CPU listens to memory bus
  - Use write-through and invalidate when you see a write bits
  - Bus-based schemes limit scalability
- **Modern CPUs use networks (e.g., AMD infinity fabric, intel UPI, CXL [between CPUs and devices])**
  - CPUs pass each other messages about cache lines

MESI coherence protocol

- **Modified**
  - Exactly one cache has a valid copy
  - That copy is dirty (needs to be written back to memory)
  - Must invalidate all copies in other caches before entering this state
- **Exclusive**
  - Same as Modified except the cache copy is clean
- **Shared**
  - One or more caches and memory have a valid copy
- **Invalid**
  - Doesn't contain any data
- **Owned (for enhanced "MOESI" protocol)**
  - Cached copy may be dirty (like Modified state)
  - But have to broadcast modifications (sort of like Shared state)
  - Can have one owned + multiple shared copies of cache line

Core and Bus Actions

- **Actions performed by CPU core**
  - Read
  - Write
  - Evict (modified/owned? must write back)
- **Transactions on bus (or interconnect)**
  - Read: without intent to modify, data can come from memory or another cache
  - Read-exclusive: with intent to modify, must invalidate all other cache copies
  - Writeback: contents put on bus and memory is updated

cc-NUMA

- **Old machines used *dance hall* architectures**
  - Any CPU can "dance with" any memory equally
- **An alternative: Non-Uniform Memory Access (NUMA)**
  - Each CPU has fast access to some "close" memory
  - Slower to access memory that is farther away
  - Use a directory to keep track of who is caching what
- **Originally for esoteric machines with many CPUs**
  - But AMD and then intel integrated memory controller into CPU
  - Faster to access memory controlled by the local socket (or even local die in a multi-chip module)
- **cc-NUMA = cache-coherent NUMA**
  - Rarely see non-cache-coherent NUMA (BBN Butterfly 1, Cray T3D)

Real World Coherence Costs

- **See [David] for a great reference. Xeon results:**
  - 3 cycle L1, 11 cycle L2, 44 cycle LLC, 355 cycle local RAM
- **If another core in same socket holds line in modified state:**
  - load: 109 cycles (LLC + 65)
  - store: 115 cycles (LLC + 71)
  - atomic CAS: 120 cycles (LLC + 76)
- **If a core in a different socket holds line in modified state:**
  - NUMA load: 289 cycles
  - NUMA store: 320 cycles
  - NUMA atomic CAS: 324 cycles
- **But only a partial picture**
  - Could be faster because of out-of-order execution
  - Could be slower if interconnect contention or multiple hops

NUMA and spinlocks

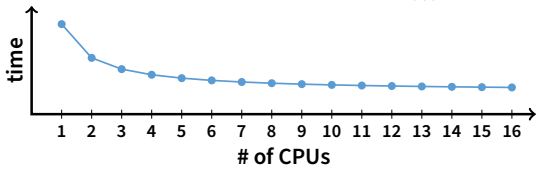
- **Test-and-set spinlock has several advantages**
  - Simple to implement and understand
  - One memory location for arbitrarily many CPUs
- **But also has disadvantages**
  - Lots of traffic over memory interconnect (especially w. > 1 spinner)
  - Not necessarily fair (lacks bounded waiting)
  - Even less fair on a NUMA machine
- **Idea 1: Avoid spinlocks altogether (today)**
- **Idea 2: Reduce interconnect traffic with better spinlocks (next lecture)**
  - Design lock that spins only on local memory
  - Also gives better fairness

Outline

Amdahl's law

$$T(n) = T(1)\left(B + \frac{1}{n}(1 - B)\right)$$

- **Expected speedup limited when only part of a task is sped up**
  - $T(n)$: the time it takes n CPU cores to complete the task
  - B: the fraction of the job that must be serial
- **Even with massive multiprocessors,** $\lim_{n\to\infty} = B \cdot T(1)$



  - Places an ultimate limit on parallel speedup
- **Problem: synchronization increases serial section size**

Locking basics

```
mutex_t m;

lock(&m);
cnt = cnt + 1; /* critical section */
unlock(&m);
```

- **Only one thread can hold a mutex at a time**
  - Makes critical section atomic
- **Recall thread API contract**
  - All access to global data must be protected by a mutex
  - Global = two or more threads touch data and at least one writes
- **Means must map each piece of global data to one mutex**
  - Never touch the data unless you locked that mutex
- **But many ways to map data to mutexes**

Locking granularity

- **Consider two lookup implementations for global hash table:**

```
struct list *hash_tbl[1021];
```

coarse-grained locking

```
mutex_t m;
      ⋮
  mutex_lock(&m);
  struct list_elem *pos = list_begin (hash_tbl[hash(key)]);
  /* ... walk list and find entry ... */
  mutex_unlock(&m);
```

fine-grained locking

```
mutex_t bucket_lock[1021];
      ⋮
  int index = hash(key);
  mutex_lock(&bucket_lock[index]);
  struct list_elem *pos = list_begin (hash_tbl[index]);
  /* ... walk list and find entry ... */
  mutex_unlock(&bucket_lock[index]);
```

- **Which implementation is better?**

Locking granularity (continued)

- **Fine-grained locking admits more parallelism**
  - E.g., imagine network server looking up values in hash table
  - Parallel requests will usually map to different hash buckets
  - So fine-grained locking should allow better speedup
- **When might coarse-grained locking be better?**

Locking granularity (continued)

- **Fine-grained locking admits more parallelism**
  - E.g., imagine network server looking up values in hash table
  - Parallel requests will usually map to different hash buckets
  - So fine-grained locking should allow better speedup
- **When might coarse-grained locking be better?**
  - Suppose you have global data that applies to whole hash table

```
struct hash_table {
  size_t num_elements;     /* num items in hash table */
  size_t num_buckets;      /* size of buckets array */
  struct list *buckets;    /* array of buckets */
};
```

  - Read `num_buckets` each time you insert
  - Check `num_elements` on insert, possibly expand buckets & rehash
  - Single global mutex would protect these fields
- **Can you avoid serializing lookups to growable hash table?**

Readers-writers problem

- **Recall a `mutex` allows access in only one thread**
- **But a data race occurs only if**
  - Multiple threads access the same data, and
  - At least one of the accesses is a write
- **How to allow multiple readers *or* one single writer?**
  - Need lock that can be *shared* amongst concurrent readers
- **Can implement using other primitives (next slides)**
  - Keep integer `i` – # of readers or -1 if held by writer
  - Protect `i` with mutex
  - Sleep on condition variable when can't get lock

Implementing shared locks

```
struct sharedlk {
  int i;    /* # shared lockers, or -1 if exclusively locked */
  mutex_t m;
  cond_t c;
};

void AcquireExclusive (sharedlk *sl) {
  lock (&sl->m);
  while (sl->i) { wait (&sl->m, &sl->c); }
  sl->i = -1;
  unlock (&sl->m);
}

void AcquireShared (sharedlk *sl) {
  lock (&sl->m);
  while (&sl->i < 0) { wait (&sl->m, &sl->c); }
  sl->i++;
  unlock (&sl->m);
}
```


Implementing shared locks (continued)

```
void ReleaseShared (sharedlk *sl) {
  lock (&sl->m);
  if (!--sl->i)
    signal (&sl->c);
  unlock (&sl->m);
}

void ReleaseExclusive (sharedlk *sl) {
  lock (&sl->m);
  sl->i = 0;
  broadcast (&sl->c);
  unlock (&sl->m);
}
```

- **Any issues with this implementation?**

Implementing shared locks (continued)

```
void ReleaseShared (sharedlk *sl) {
  lock (&sl->m);
  if (!--sl->i)
    signal (&sl->c);
  unlock (&sl->m);
}

void ReleaseExclusive (sharedlk *sl) {
  lock (&sl->m);
  sl->i = 0;
  broadcast (&sl->c);
  unlock (&sl->m);
}
```

- **Any issues with this implementation?**
  - Prone to starvation of writer (no bounded waiting)
  - How might you fix?

Review: Test-and-set spinlock

```
struct var {
  int lock;
  int val;
};

void atomic_inc (var *v) {
  while (test_and_set (&v->lock))
    ;
  v->val++;
  v->lock = 0;
}

void atomic_dec (var *v) {
  while (test_and_set (&v->lock))
    ;
  v->val--;
  v->lock = 0;
}
```

- **Is this code correct without sequential consistency?**

Memory reordering danger

- **Suppose no sequential consistency (& don't compensate)**
- **Hardware could violate program order**

Program order on CPU #1	View on CPU #2
`v->lock = 1;`	`v->lock = 1;`
`register = v->val;`	
`v->val = register + 1;`	
`v->lock = 0;`	`v->lock = 0;`
	`/* danger */;`
	`v->val = register + 1;`

- **If `atomic_inc` called at `/* danger */`, bad `val` ensues!**

Ordering requirements

```
void atomic_inc (var *v) {
  while (test_and_set (&v->lock))
    ;
  v->val++;
  /* danger */
  v->lock = 0;
}
```

- **Must ensure all CPUs see the following:**
  1. `v->lock = 1` ran *before* `v->val` was read and written
  2. `v->lock = 0` ran *after* `v->val` was written
- **How does #1 get assured on x86?**
  - Recall `test_and_set` uses `xchgl %eax,(%edx)`
- **How to ensure #2 on x86?**

Ordering requirements

```
void atomic_inc (var *v) {
  while (test_and_set (&v->lock))
    ;
  v->val++;
  /* danger */
  v->lock = 0;
}
```

- **Must ensure all CPUs see the following:**
  1. `v->lock = 1` ran *before* `v->val` was read and written
  2. `v->lock = 0` ran *after* `v->val` was written
- **How does #1 get assured on x86?**
  - Recall `test_and_set` uses `xchgl %eax,(%edx)`
  - `xchgl` instruction always "locked," ensuring barrier
- **How to ensure #2 on x86?**

Ordering requirements

```
void atomic_inc (var *v) {
  while (test_and_set (&v->lock))
    ;
  v->val++;
  asm volatile ("sfence" ::: "memory");
  v->lock = 0;
}
```

- **Must ensure all CPUs see the following:**
  1. `v->lock = 1` ran *before* `v->val` was read and written
  2. `v->lock = 0` ran *after* `v->val` was written
- **How does #1 get assured on x86?**
  - Recall `test_and_set` uses `xchgl %eax,(%edx)`
  - `xchgl` instruction always "locked," ensuring barrier
- **How to ensure #2 on x86?**
  - Might need fence instruction after, e.g., non-temporal stores
  - Definitely need compiler barrier

Gcc extended `asm` syntax [gnu]

`asm volatile (`*template-string* `:` *outputs* `:` *inputs* `:` *clobbers*`);`

- **Puts *template-string* in assembly language compiler output**
  - Expands `%0`, `%1`, … (a bit like printf conversion specifiers)
  - Use "`%%`" for a literal `%` (e.g., "`%%cr3`" to specify `%cr3` register)
- ***inputs*/*outputs* specify parameters as "*constraint*" (*value*)**

```
int outvar, invar = 3;
asm ("movl %1, %0" : "=r" (outvar) : "r" (invar));
/* now outvar == 3 */
```

- ***clobbers* lists other state that get used/overwritten**
  - Special value `"memory"` prevents reordering with loads & stores
  - Serves as *compiler barrier*, as important as hardware barrier
- **`volatile` indicates side effects other than result**
  - Otherwise, gcc might optimize away if you don't use result

Correct spinlock on alpha

- **Recall implementation of `test_and_set` on alpha (with much weaker memory consistency than x86):**

```
_test_and_set:
     ldq_l   v0, 0(a0)          # v0 = *lockp (LOCKED)
     bne     v0, 1f             # if (v0) return
     addq    zero, 1, v0        # v0 = 1
     stq_c   v0, 0(a0)          # *lockp = v0 (CONDITIONAL)
     beq     v0, _test_and_set # if (failed) try again
     mb
     addq    zero, zero, v0     # return 0
1:   ret     zero, (ra), 1
```

- ***Memory barrier* instruction `mb` (like `mfence`)**
  - All processors will see that everything before `mb` in program order happened before everything after `mb` in program order
- **Need barrier before releasing spinlock as well:**

```
asm volatile ("mb" ::: "memory");
v->lock = 0;
```


Memory barriers/fences

- **Fortunately, consistency need not overly complicate code**
  - If you do locking right, only need a few fences within locking code
  - Code will be easily portable to new CPUs
- **Most programmers should stick to mutexes**
- **But advanced techniques may require lower-level code**
  - Later this lecture will see some wait-free algorithms
  - Also important for optimizing special-case locks (E.g., linux kernel `rw_semaphore`, …)
- **Algorithms often explained assuming sequential consistency**
  - Must know how to use memory fences to implement correctly
  - E.g., see [Howells] for how Linux deals with memory consistency
  - And another plug for Why Memory Barriers
- **Next: How C11 allows portable low-level code**

Outline

Atomics and portability

- **Lots of variation in atomic instructions, consistency models, compiler behavior**
  - Changing the compiler or optimization level can invalidate code
- **Different CPUs today: Many laptops (not Apple) are x86, while your cell phone uses ARM**
  - x86: Total Store Order Consistency Model, CISC
  - arm: Relaxed Consistency Model, RISC
- **Could make it impossible to write portable kernels and applications**
- **Fortunately, the C11 standard has builtin support for atomics**
  - If not on by default, use `gcc -std=gnu11` or `-std=gnu17`
- **Also available in C++11, but won't discuss today**

Background: C memory model [C11]

- **Within a thread, many evaluations are *sequenced***
  - E.g., in "`f1(); f2();`", evaluation of `f1` is sequenced before `f2`
- **Across threads, some operations *synchronize with* others**
  - E.g., releasing mutex m synchronizes with a subsequent acquire m
- **Evaluation A *happens before* B, which we'll write $A \rightarrow B$, when:**
  - A is sequenced before B (in the same thread),
  - A synchronizes with B,
  - A is dependency-ordered before B (ignore for now—means A has release semantics and B consume semantics for same value), or
  - There is another operation X such that $A \rightarrow X \rightarrow B$.[1]

[1] Except chain of "$\rightarrow$" cannot end: …, dependency-ordered, sequenced before

C11 Atomics: Big picture

- **C11 says a *data race* produces undefined behavior (UB)**
  - A write *conflicts* with a read or write of same memory location
  - Two conflicting operations *race* if not ordered by happens before
  - Undefined can be anything (e.g., delete all your files, …)
  - Think UB okay in practice? See [Wang], [Lattner]
- **Spinlocks (and hence mutexes that internally use spinlocks) synchronize across threads**
  - Synchronization adds happens before arrows, avoiding data races
- **Yet hardware supports other means of synchronization**
- **C11 atomics provide direct access to synchronized lower-level operations**
  - E.g., can get compiler to issue `lock` prefix in some cases

C11 Atomics: Basics

- **Include new `<stdatomic.h>` header**
- **New `_Atomic` type qualifier: e.g., `_Atomic int foo;`**
  - Convenient aliases: `atomic_bool`, `atomic_int`, `atomic_ulong`, ...
  - Must initialize specially:

```
#include <stdatomic.h>
_Atomic int global_int = ATOMIC_VAR_INIT(140);
        ⋮
    Atomic_(int) *dyn = malloc(sizeof(*dyn));
    atomic_init(dyn, 140);
```

- **Compiler emits read-modify-write instructions for atomics**
  - E.g., `+=`, `-=`, `|=`, `&=`, `^=`, `++`, `--` do what you would hope
  - Act atomically and synchronize with one another
- **Also functions including `atomic_fetch_add`, `atomic_compare_exchange_strong`, ...**

Locking and atomic flags

- **Implementations might use spinlocks internally for most atomics**
  - Could interact badly with interrupt/signal handlers
  - Can check if `ATOMIC_INT_LOCK_FREE`, etc., macros defined
  - Fortunately modern CPUs don't require this
- **`atomic_flag` is a special type guaranteed lock-free**
  - Boolean value without support for loads and stores
  - Initialize with: `atomic_flag mylock = ATOMIC_FLAG_INIT;`
  - Only two kinds of operation possible:
    - `_Bool atomic_flag_test_and_set(volatile atomic_flag *obj);`
    - `void atomic_flag_clear(volatile atomic_flag *obj);`
  - Above functions guarantee sequential consistency (atomic operation serves as memory fence, too)

Exposing weaker consistency

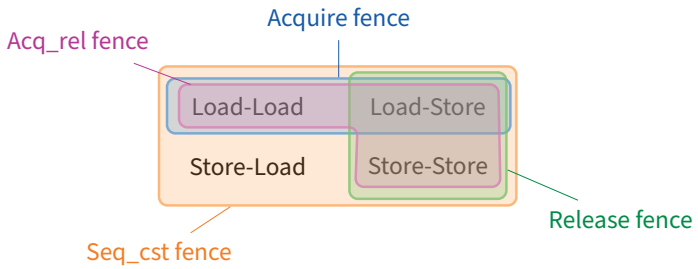
```
enum memory_order { /*...*/ };

_Bool atomic_flag_test_and_set_explicit(
    volatile atomic_flag *obj, memory_order order);
void atomic_flag_clear_explicit(
    volatile atomic_flag *obj, memory_order order);

C atomic_load_explicit(
    const volatile A *obj, memory_order order);
void atomic_store_explicit(
    volatile A *obj, C desired, memory_order order);

bool atomic_compare_exchange_weak_explicit(
    A *obj, C *expected, C desired,
    memory_order succ, memory_order fail);
```

- **Atomic functions have `_explicit` variants**
  - These guarantee coherence but *not* sequential consistency
  - May allow compiler to generate faster code

Memory ordering

- **Six possible `memory_order` values:**
  1. `memory_order_relaxed`: no memory ordering
  2. `memory_order_consume`: super tricky, see [Preshing] for discussion
  3. `memory_order_acquire`: for start of critical section
  4. `memory_order_release`: for end of critical section
  5. `memory_order_acq_rel`: combines previous two
  6. `memory_order_seq_cst`: full sequential consistency
- **Also have fence operation not tied to particular atomic:**
  `void atomic_thread_fence(memory_order order);`
- **Suppose thread 1 releases and thread 2 acquires**
  - Thread 1's preceding accesses can't move past **release** store
  - Thread 2's subsequent accesses can't move before **acquire** load
  - Warning: other threads might see a completely different order

Types of memory fence[2]



- ***X*-*Y* fence = operations of type *X* sequenced before the fence happen before operations of type *Y* sequenced after the fence**

[2] Credit to [Preshing] for explaining it this way

Example: Atomic counters

```
_Atomic(int) packet_count;

void recv_packet(...)
{
  atomic_fetch_add_explicit(&packet_count, 1,
                            memory_order_relaxed);
      ⋮
}
void reset_counter()
{
  atomic_store_explicit(&packet_count, 0,
                        memory_order_relaxed);
}
```

- **Need to count packets accurately**
- **Don't need to order other memory accesses across threads**
- **Relaxed memory order can avoid unnecessary overhead**
  - On x86, `recv_packet` doesn't benefit, but `reset_counter` uses `mov` compared to `xchgl` for `packet_count = 0`

Example: Producer, consumer 1

```
struct message msg_buf;
_Atomic(_Bool) msg_ready;

void send(struct message *m) {
  msg_buf = *m;
  atomic_thread_fence(memory_order_release);
  /* Prior loads+stores happen before subsequent stores */
  atomic_store_explicit(&msg_ready, 1,
                        memory_order_relaxed);
}

struct message *recv(void) {
  _Bool ready = atomic_load_explicit(&msg_ready,
                                     memory_order_relaxed);
  if (!ready)
    return NULL;
  atomic_thread_fence(memory_order_acquire);
  /* Prior loads happen before subsequent loads+stores */
  return &msg_buf;
}
```


Example: Producer, consumer 2

```
struct message msg_buf;
_Atomic(_Bool) msg_ready;

void send(struct message *m) {
  msg_buf = *m;
  atomic_store_explicit(&msg_ready, 1,
                        memory_order_release);
}

struct message *recv(void) {
  _Bool ready = atomic_load_explicit(&msg_ready,
                                     memory_order_acquire);
  if (!ready)
    return NULL;
  return &msg_buf;
}
```

- **This is potentially faster than previous example**
  - E.g., atomic other stores after `send` can be moved before `msg_buf`

Example: Test-and-set spinlock

```
void
spin_lock(atomic_flag *lock)
{
  while(atomic_flag_test_and_set_explicit(lock,
                                  memory_order_acquire))
    ;
}

void
spin_unlock(atomic_flag *lock)
{
  atomic_flag_clear_explicit(lock, memory_order_release);
}
```


Example: Better test-and-set spinlock

```
void spin_lock(atomic_bool *lock)
{
  while(atomic_exchange_explicit(lock, 1,
                                 memory_order_acquire)) {
    while(atomic_load_explicit(lock, memory_order_relaxed)) {
#if __i386 || __x86_64
      __builtin_ia32_pause();
#elif __ARM_ARCH >= 7 && __clang__
      __builtin_arm_yield();
#endif
    }
  }
}

void spin_unlock(atomic_bool *lock)
{
  atomic_store_explicit(lock, 0, memory_order_release);
}
```

- **See [Rigtorp] for a good discussion**

Outline

Recall producer/consumer (lecture 3)

```
/* PRODUCER */
for (;;) {
  item *nextProduced
    = produce_item ();

  mutex_lock (&mutex);
  while (count == BUF_SIZE)
    cond_wait (&nonfull,
               &mutex);

  buffer[in] = nextProduced;
  in = (in + 1) % BUF_SIZE;
  count++;
  cond_signal (&nonempty);
  mutex_unlock (&mutex);
}
```

```
/* CONSUMER */
for (;;) {
  mutex_lock (&mutex);
  while (count == 0)
    cond_wait (&nonempty,
               &mutex);

  nextConsumed = buffer[out];
  out = (out + 1) % BUF_SIZE;
  count--;
  cond_signal (&nonfull);
  mutex_unlock (&mutex);

  consume_item (nextConsumed);
}
```

Eliminating locks

- **One use of locks is to coordinate multiple updates of single piece of state**
- **How to remove locks here?**
  - Factor state so that each variable only has a single writer
- **Producer/consumer example revisited**
  - Assume one producer, one consumer
  - Why do we need `count` variable, written by both?
    To detect buffer full/empty
  - Have producer write `in`, consumer write `out` (both `_Atomic`)
  - Use `in`/`out` to detect buffer state
    (sacrifice one buffer slot to distinguish completely full and empty)
  - But note next example busy-waits, which is less good

Lock-free producer/consumer

```
atomic_int in, out;

void producer (void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();
        while (((in + 1) % BUF_SIZE) == out) thread_yield ();
        buffer[in] = nextProduced;
        in = (in + 1) % BUF_SIZE;
    }
}

void consumer (void *ignored) {
    for (;;) {
        while (in == out) thread_yield ();
        nextConsumed = buffer[out];
        out = (out + 1) % BUF_SIZE;
        consume_item (nextConsumed);
    }
}
```

[Note fences not needed because no relaxed atomics]

Version with relaxed atomics

```
void producer (void *ignored) {
  int slot = atomic_load(&in);
  for (;;) {
    item *nextProduced = produce_item ();
    int next = (slot + 1) % BUF_SIZE;
    while (atomic_load_explicit(&out, memory_order_acquire) ==
           next)        // Could you use relaxed?  ^^^^^^^
      thread_yield();
    buffer[slot] = nextProduced;
    atomic_store_explicit(&in, next, memory_order_release);
    slot = next;
  }
}

void consumer (void *ignored) {
  // Use memory_order_acquire to load in (for latest buffer[myin])
  // Use memory_order_release to store out
}
```

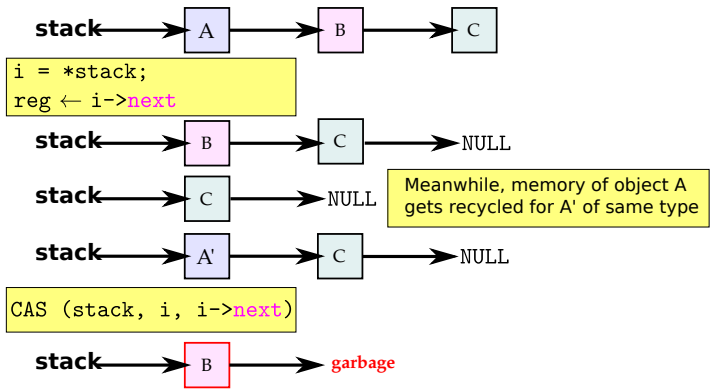

Non-blocking synchronization

- **Design algorithm to *avoid critical sections***
  - Any threads can make progress if other threads are preempted
  - Which wouldn't be the case if preempted thread held a lock
- **Requires that hardware provide the right kind of atomics**
  - Simple test-and-set is insufficient
  - Atomic compare and swap is good: `CAS (mem, old, new)`
    If `*mem == old`, then swap `*mem`⟷`new` and return `true`, else `false`
- **Can implement many common data structures**
  - Stacks, queues, even hash tables
- **Can implement any algorithm on right hardware**
  - Need operation such as atomic compare and swap
    (has property called *consensus number* $= \infty$ [Herlihy])
  - Entire kernels have been written without locks [Greenwald]

Example: non-blocking stack

```
struct item {
  /* data */
  _Atomic (struct item *) next;
};
typedef _Atomic (struct item *) stack_t;

void atomic_push (stack_t *stack, item *i) {
  do {
    i->next = *stack;
  } while (!CAS (stack, i->next, i));
}

item *atomic_pop (stack_t *stack) {
  item *i;
  do {
    i = *stack;
  } while (!CAS (stack, i, i->next));
  return i;
}
```


Wait-free stack issues



- **"ABA" race in pop if other thread pops, re-pushes i**
  - Can be solved by counters or hazard pointers to delay re-use

"Benign" races

- **Could also eliminate locks by having race conditions**
- **Maybe you think you care more about speed than correctness**

```
++hits; /* each time someone accesses web site */
```

- **Maybe you think you can get away with the race (NOT!, really)**

```
if (!initialized) {
  lock (m);
  if (!initialized) {
    initialize ();
    atomic_thread_fence (memory_order_release); /* why? */
    initialized = 1;
  }
  unlock (m);
}
```

- **But don't do this [Vyukov], [Boehm]! Not benign at all**
  - Again, UB really bad! Like use-after free or array overflow bad
  - If needed for efficiency, use relaxed-memory-order atomics

Read-copy update [McKenney]

- **Some data is read way more often than written**
  - Routing tables consulted for each forwarded packet
  - Data maps in system with 100+ disks (updated on disk failure)
- **Optimize for the common case of reading without lock**
  - E.g., global variable: `routing_table *rt;`
  - Call `lookup (rt, route);` with no lock
- **Update by making copy, swapping pointer**

```
routing_table *newrt = copy_routing_table (rt);
update_routing_table (newrt);
atomic_thread_fence (memory_order_release);
rt = newrt;
```

- **Is RCU really safe? Stay tuned next lecture...**

Next class

- **The exciting conclusion of RCU**
  - Spoiler: safe on all architectures except on alpha
- **Building a better spinlock**
- **What interface should kernel provide for sleeping locks?**
- **Deadlock**
- **Scalable interface design**

8. Synchronization 2

Overview of previous and current lectures

- **Locks create serial code**
  - Serial code gets no speedup from multiprocessors
- **Test-and-set spinlock has additional disadvantages**
  - Lots of traffic over memory bus
  - Not fair on NUMA machines
- **Idea 1: Avoid spinlocks**
  - We saw lock-free algorithms last lecture
  - Mentioned RCU last time, dive deeper today
- **Idea 2: Design better spinlocks**
  - Less memory traffic, better fairness
- **Idea 3: Hardware turns coarse- into fine-grained locks!**
  - While also reducing memory traffic for lock in common case

Outline

Read-copy update [McKenney]

- **Some data is read way more often than written**
  - Routing tables consulted for each forwarded packet
  - Data maps in system with 100+ disks (updated on disk failure)
- **Optimize for the common case of reading without lock**
  - Have global variable: `_Atomic(routing_table *) rt;`
  - Use it with no lock

```
#define RELAXED(var) \
  atomic_load_explicit(&(var), memory_order_relaxed)

/* ... */

  route = lookup(RELAXED(rt), destination);
```

- **Update by making copy, swapping pointer**

```
/* update mutex held here, serializing updates */
routing_table *newrt = copy_routing_table(rt);
update_routing_table(newrt);
atomic_store_explicit(&rt, newrt, memory_order_release);
```


Is RCU really safe?

- **Consider the use of global `rt` with no fences:**

```
lookup(RELAXED(rt), route);
```

- **Could a CPU read new pointer but then old contents of `*rt`?**
- **Yes on alpha, No on all other existing architectures**
- **We are saved by *dependency ordering* in hardware**
  - Instruction *B* depends on *A* if *B* uses result of *A*
  - Non-alpha CPUs won't re-order dependent instructions
  - If writer uses release fence, safe to load pointer then just use it
- **This is the point of `memory_order_consume`**
  - Should be equivalent to acquire barrier on alpha
  - But should compile to nothing (be free) on other machines
  - But hard to get semantics right (temporarily deprecated in C++)

Preemptible kernels

- **Recall *kernel process context* from lecture 1**
  - When CPU in kernel mode but executing on behalf of a process (e.g., might be in system call or page fault handler)
  - As opposed to interrupt handlers or context switch code
- **A *preemptible kernel* can preempt process context code**
  - Take a CPU core away from kernel process context code between any two instructions
  - Give the same CPU core to kernel code for a different process
- **Don't confuse with:**
  - Interrupt handlers can always preempt process context code
  - Preemptive threads (always have for multicore)
  - Process context code running concurrently on other CPU cores
- **Sometimes want or need to disable preemption**
  - Code that must not be migrated between CPUs (per-CPU structs)
  - Before acquiring spinlock (could improve performance)

Garbage collection

- **When can you free memory of old routing table?**
  - When you are guaranteed no one is using it—how to determine?
- **Definitions:**
  - *temporary variable* – short-used (e.g., local) variable
  - *permanent variable* – long lived data (e.g., global `rt` pointer)
  - *quiescent state* – when all a thread's temporary variables dead
  - *quiescent period* – time during which every thread has been in quiescent state at least once
- **Free old copy of updated data after quiescent period**
  - How to determine when quiescent period has gone by?
  - E.g., keep count of syscalls/context switches on each CPU
- **Restrictions:**
  - Can't hold a pointer across context switch or user mode (Never copy `rt` into another permanent variable)
  - Must disable preemption while consuming RCU data structure

Outline

1. RCU
2. Improving spinlock performance
3. Kernel interface for sleeping locks
4. Deadlock
5. Transactions
6. Scalable interface design

Useful macros

- **Atomic compare and swap:** `CAS (mem, old, new)`
  - If `*mem == old`, then swap `*mem`↔`new` and return `true`, else `false`
  - On x86, can implement using locked `cmpxchg` instruction
  - In C11, use `atomic_compare_exchange_strong` (note: C atomics version sets `old = *mem` if `*mem != old`)
- **Atomic swap:** `XCHG (mem, new)`
  - Atomically exchanges `*mem`↔`new`
  - Implement w. C11 `atomic_exchange`, or `xchg` on x86
- **Atomic fetch and add:** `FADD (mem, val)`
  - Atomically sets `*mem += val` and returns *old* value of `*mem`
  - Implement w. C11 `atomic_fetch_add`, `lock add` on x86
- **Atomic fetch and subtract:** `FSUB (mem, val)`
- **Note: atomics return previous value (like `x++`, not `++x`)**
- **All behave like sequentially consistent fences**
  - In C11, weaker `_explicit` versions take a `memory_order` argument

MCS lock

- **Idea 2: Build a better spinlock**
- **Lock designed by Mellor-Crummey and Scott**
  - Goal: reduce bus traffic on cc machines, improve fairness
- **Each CPU has a `qnode` structure in local memory**

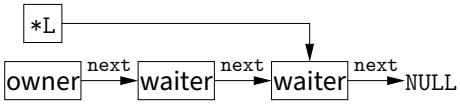
```
typedef struct qnode {
  _Atomic (struct qnode *) next;
  atomic_bool locked;
} qnode;
```

  - Local can mean local memory in NUMA machine
  - Or just its own cache line that gets cached in exclusive mode
- **While waiting, spin on *your local* `locked` flag**
- **A lock is a `qnode` pointer:** `typedef _Atomic (qnode *) lock;`
  - Construct list of CPUs holding or waiting for lock
  - `lock` itself points to tail of list list (or NULL when unlocked)

MCS Acquire

- **If unlocked, `L` is `NULL`**
- **If locked, no waiters, `L` is owner's `qnode`**
- **If waiters, `*L` is tail of waiter list:**

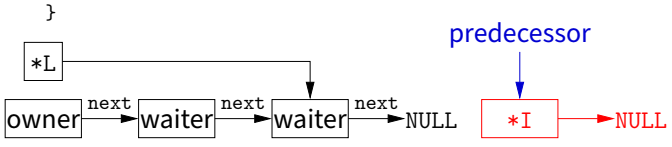
```
acquire (lock *L, qnode *I) {
  I->next = NULL;
  qnode *predecessor = I;
  XCHG (*L, predecessor);
  if (predecessor != NULL) {
    I->locked = true;
    predecessor->next = I;
    while (I->locked)
      ;
  }
}
```



MCS Acquire

- **If unlocked, `L` is `NULL`**
- **If locked, no waiters, `L` is owner's `qnode`**
- **If waiters, `*L` is tail of waiter list:**

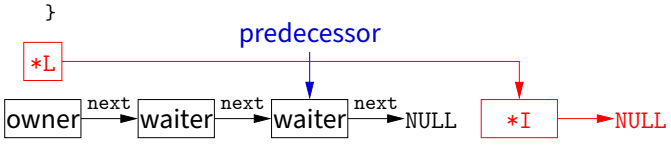
```
acquire (lock *L, qnode *I) {
  I->next = NULL;
  qnode *predecessor = I;
  XCHG (*L, predecessor);
  if (predecessor != NULL) {
    I->locked = true;
    predecessor->next = I;
    while (I->locked)
      ;
  }
}
```



MCS Acquire

- **If unlocked, `L` is `NULL`**
- **If locked, no waiters, `L` is owner's `qnode`**
- **If waiters, `*L` is tail of waiter list:**

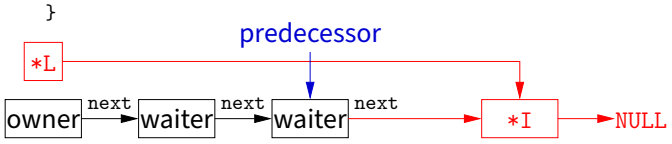
```
acquire (lock *L, qnode *I) {
  I->next = NULL;
  qnode *predecessor = I;
  XCHG (*L, predecessor);
  if (predecessor != NULL) {
    I->locked = true;
    predecessor->next = I;
    while (I->locked)
      ;
  }
}
```



MCS Acquire

- **If unlocked,** `L` **is** `NULL`
- **If locked, no waiters,** `L` **is owner's** `qnode`
- **If waiters,** `*L` **is tail of waiter list:**

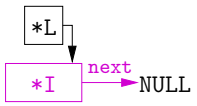
```
acquire (lock *L, qnode *I) {
  I->next = NULL;
  qnode *predecessor = I;
  XCHG (*L, predecessor);
  if (predecessor != NULL) {
    I->locked = true;
    predecessor->next = I;
    while (I->locked)
      ;
  }
}
```



MCS Release with CAS

```
release (lock *L, qnode *I) {
  if (!I->next)
    if (CAS (*L, I, NULL))
      return;
  while (!I->next)
    ;
  I->next->locked = false;
}
```

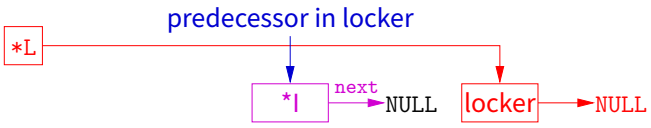
- **If** `I->next` **NULL and** `*L == I`
  - No one else is waiting for lock, OK to set `*L = NULL`



MCS Release with CAS

```
release (lock *L, qnode *I) {
  if (!I->next)
    if (CAS (*L, I, NULL))
      return;
  while (!I->next)
    ;
  I->next->locked = false;
}
```

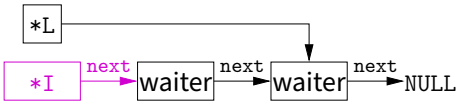
- **If** `I->next` **NULL and** `*L != I`
  - Another thread is in the middle of `acquire`
  - Just wait for `I->next` to be non-NULL



MCS Release with CAS

```
release (lock *L, qnode *I) {
  if (!I->next)
    if (CAS (*L, I, NULL))
      return;
  while (!I->next)
    ;
  I->next->locked = false;
}
```

- **If** `I->next` **is non-NULL**
  - `I->next` oldest waiter, wake up with `I->next->locked = false`



MCS Release w/o CAS

- **What to do if no atomic CAS (consensus number** ∞**), but do have XCHG (consensus number 2)?**
- **Be optimistic—read** `*L` **with two** `XCHG`**s:**
  1. Atomically swap NULL into `*L`
     - If old value of `*L` was `I`, no waiters and we are done
  2. Atomically swap old `*L` value back into `*L`
     - If `*L` unchanged, same effect as `CAS`
- **Otherwise, we have to clean up the mess**
  - Some "userper" attempted to acquire lock between **1** and **2**
  - Because `*L` was `NULL`, the userper succeeded
    (May be followed by zero or more waiters)
  - Graft old list of waiters on to end of new last waiter
    (Sacrifice small amount of fairness, but still safe)

MCS Release w/o C&S code

```
release (lock *L, qnode *I) {
  if (I->next)
    I->next->locked = false;
  else {
    qnode *old_tail = NULL;
    XCHG (*L, old_tail);
    if (old_tail == I)
      return;

    /* old_tail != I? CAS would have failed, so undo XCHG */
    qnode *userper = old_tail;
    XCHG (*L, userper);
    while (I->next == NULL)
      ;
    if (userper)  /* someone changed *L between 2 XCHGs */
      userper->next = I->next;
    else
      I->next->locked = false;
  }
}
```

Outline

1. RCU
2. Improving spinlock performance
3. Kernel interface for sleeping locks
4. Deadlock
5. Transactions
6. Scalable interface design

Kernel support for sleeping locks

- **Sleeping locks must interact with scheduler**
  - For processes or native threads, must go into kernel (expensive)
  - Common case is you can acquire lock—how to optimize?
- **Idea: never enter kernel for uncontested lock**

```
struct lock {
  atomic_flag busy;
  _Atomic (thread *) waiters; /* wait-free stack/queue */
};
void acquire (lock *lk) {
  while (atomic_flag_test_and_set (&lk->busy)) { /* 1 */
    atomic_push (&lk->waiters, self);            /* 2 */
    sleep ();
  }
}
void release (lock *lk) {
  atomic_flag_clear(&lk->busy);
  wakeup (atomic_pop (&lk->waiters));
}
```


Race condition

- **Unfortunately, previous slide not safe**
  - What happens if release called between lines 1 and 2?
  - `wakeup` called on `NULL`, so `acquire` blocks
- ***futex* abstraction solves the problem [Franke]**
  - Ask kernel to sleep only if memory location hasn't changed
- `void futex (int *uaddr, FUTEX_WAIT, int val...);`
  - Go to sleep only if `*uaddr == val`
  - Extra arguments allow timeouts, etc.
- `void futex (int *uaddr, FUTEX_WAKE, int val...);`
  - Wake up at most `val` threads sleeping on `uaddr`
- **`uaddr` is translated down to offset in VM object**
  - So works on memory mapped file at different virtual addresses in different processes

Futex example

```
struct lock {
  atomic_flag busy;
};
void acquire (lock *lk) {
  while (atomic_flag_test_and_set (&lk->busy))
    futex(&lk->busy, FUTEX_WAIT, 1);
}
void release (lock *lk) {
  atomic_flag_clear (&lk->busy);
  futex(&lk->busy, FUTEX_WAKE, 1);
}
```

- **What's suboptimal about this code?**
- **See [Drepper] for these examples and a good discussion**

Futex example

```
struct lock {
  atomic_flag busy;
};
void acquire (lock *lk) {
  while (atomic_flag_test_and_set (&lk->busy))
    futex(&lk->busy, FUTEX_WAIT, 1);
}
void release (lock *lk) {
  atomic_flag_clear (&lk->busy);
  futex(&lk->busy, FUTEX_WAKE, 1);
}
```

- **What's suboptimal about this code?**
- **See [Drepper] for these examples and a good discussion**

Futex example, second attempt

```
static_assert (ATOMIC_INT_LOCK_FREE >= 2);

struct lock {
  atomic_int busy;
};
void acquire (lock *lk) {
  int c;
  while ((c = FADD(&lk->busy, 1)))            /* 1 */
    futex((int*) &lk->busy, FUTEX_WAIT, c+1); /* 2 */
}
void release (lock *lk) {
  if (FSUB(&lk->busy, 1) != 1) {
    lk->busy = 0;
    futex((int*) &lk->busy, FUTEX_WAKE, 1);
  }
}
```

- **Now what's wrong with this code?**

Futex example, second attempt

```
static_assert (ATOMIC_INT_LOCK_FREE >= 2);

struct lock {
  atomic_int busy;
};
void acquire (lock *lk) {
  int c;
  while ((c = FADD(&lk->busy, 1)))            /* 1 */
    futex((int*) &lk->busy, FUTEX_WAIT, c+1); /* 2 */
}
void release (lock *lk) {
  if (FSUB(&lk->busy, 1) != 1) {
    lk->busy = 0;
    futex((int*) &lk->busy, FUTEX_WAKE, 1);
  }
}
```

- **Now what's wrong with this code?**
  - Two threads could interleave lines 1 and 2, never sleep
  - Could even overflow the counter, violate mutual exclusion

Futex example, third attempt

```
struct lock {
  // 0=unlocked, 1=locked no waiters, 2=locked+waiters
  atomic_int state;
};
void acquire (lock *lk) {
  int c = 1;
  if (!CAS (&lk->state, 0, c)) {
    XCHG (&lk->state, c = 2);
    while (c != 0) {
      futex ((int *) &lk->state, FUTEX_WAIT, 2);
      XCHG (&lk->state, c = 2);
    }
  }
}
void release (lock *lk) {
  if (FSUB (&lk->state, 1) != 1) { // FSUB returns old value
    lk->state = 0;
    futex ((int *) &lk->state, FUTEX_WAKE, 1);
  }
}
```


Outline

The deadlock problem

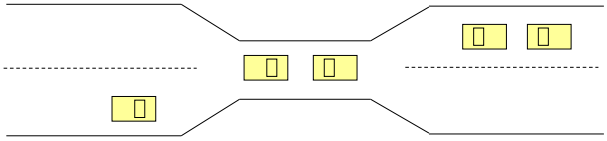
```
mutex_t m1, m2;

void p1 (void *ignored) {
  lock (m1);
  lock (m2);
  /* critical section */
  unlock (m2);
  unlock (m1);
}
void p2 (void *ignored) {
  lock (m2);
  lock (m1);
  /* critical section */
  unlock (m1);
  unlock (m2);
}
```

- **This program can cease to make progress – how?**
- **Can you have deadlock w/o mutexes?**

More deadlocks

- **Same problem with condition variables**
  - Suppose resource 1 managed by c_1, resource 2 by c_2
  - A has 1, waits on $c2$, B has 2, waits on $c1$
- **Or have combined mutex/condition variable deadlock:**
  - `lock (a); lock (b); while (!ready) wait (b, c); unlock (b); unlock (a);`
  - `lock (a); lock (b); ready = true; signal (c); unlock (b); unlock (a);`
- **One lesson: Dangerous to hold locks when crossing abstraction barriers!**
  - I.e., `lock (a)` then call function that uses condition variable

Deadlocks w/o computers



- **Real issue is *resources* & how required**
- **E.g., bridge only allows traffic in one direction**
  - Each section of a bridge can be viewed as a resource.
  - If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
  - Several cars may have to be backed up if a deadlock occurs.
  - Starvation is possible.

Deadlock conditions

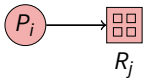
1. **Limited access (mutual exclusion):**
   - Resource can only be shared with finite users
2. **No preemption:**
   - Once resource granted, cannot be taken away
3. **Multiple independent requests (hold and wait):**
   - Don't ask all at once
     (wait for next resource while holding current one)
4. **Circularity in graph of requests**

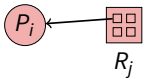
- **All of 1–4 necessary for deadlock to occur**
- **Two approaches to dealing with deadlock:**
  - Pro-active: prevention
  - Reactive: detection + corrective action

Prevent by eliminating one condition

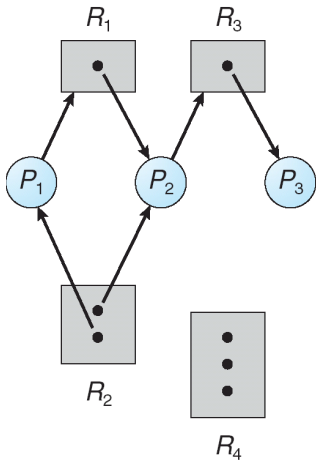
1. **Limited access (mutual exclusion):**
   - Buy more resources, split into pieces, or virtualize to make "infinite" copies
   - Threads: threads have copy of registers = no lock
2. **No preemption:**
   - Physical memory: virtualized with VM, can take physical page away and give to another process!
3. **Multiple independent requests (hold and wait):**
   - Wait on all resources at once (must know in advance)
4. **Circularity in graph of requests**
   - Single lock for entire system: (problems?)
   - Partial ordering of resources (next)

Resource-allocation graph

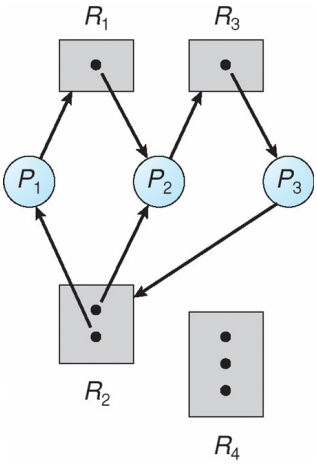
- **View system as graph**
  - Processes and Resources are nodes
  - Resource Requests and Assignments are edges
- **Process:** 
- **Resource with 4 instances:** 
- P_i **requesting** R_j: 

- P_i **holding instance of** R_j: 

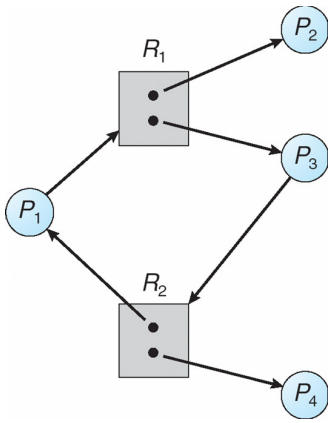
Example resource allocation graph



Graph with deadlock



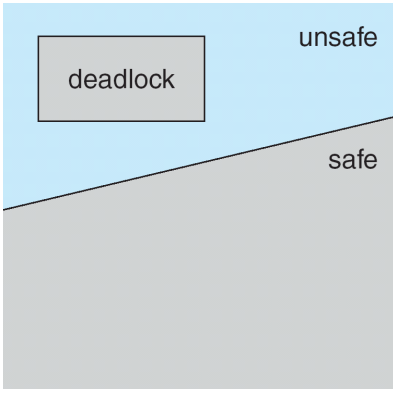
Is this deadlock?



Cycles and deadlock

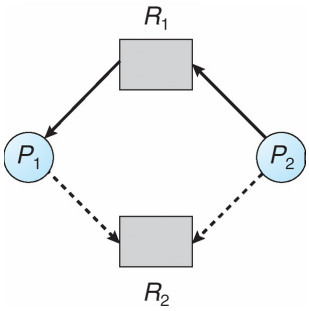
- **If graph has no cycles $\Longrightarrow$ no deadlock**
- **If graph contains a cycle**
  - Definitely deadlock if only one instance per resource
  - Otherwise, maybe deadlock, maybe not
- **Prevent deadlock with partial order on resources**
  - E.g., always acquire mutex m_1 before m_2
  - Usually design locking discipline for application this way

Prevention



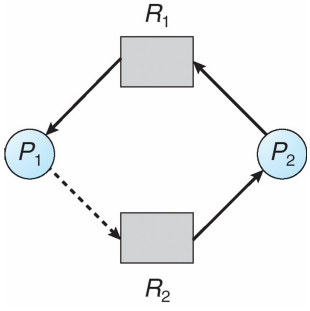
- **Determine safe states based on *possible* resource allocation**
- **Conservatively prohibits non-deadlocked states**

Claim edges



- **Dotted line is *claim edge***
  - Signifies process *may* request resource

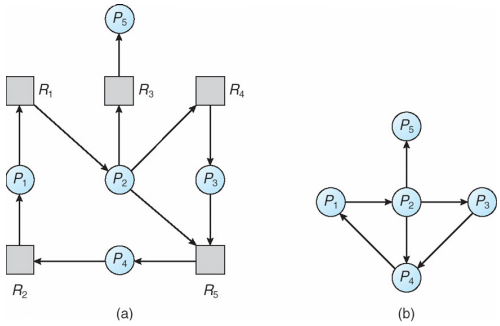
Example: unsafe state



- **Note cycle in graph**
  - P_1 might request R_2 before relinquishing R_1
  - Would cause deadlock

Detecting deadlock

- **Static approaches (hard)**
- **Dynamically, program grinds to a halt**
  - Threads package can diagnose by keeping track of locks held:



Resource-Allocation Graph Corresponding wait-for graph

Fixing & debugging deadlocks

- **Reboot system / restart application**
- **Examine hung process with debugger**
- **Threads package can deduce partial order**
  - For each lock acquired, order with other locks held
  - If cycle occurs, abort with error
  - Detects *potential* deadlocks even if they do not occur
- **Or use *transactions*...**
  - Another paradigm for handling concurrency
  - Often provided by databases, but some OSes use them
  - *Vino* OS used transactions to abort after failures [Seltzer]

Outline

Transactions

- **A *transaction* T is a collection of actions with**
  - *A*tomicity – all or none of actions happen
  - *C*onsistency – T leaves data in valid state
  - *I*solation – T's actions all appear to happen before or after every other transaction
  - *D*urability[1] – T's effects will survive reboots
  - Often hear mnemonic *ACID* to refer to above
- **Transactions typically executed concurrently**
  - But *isolation* means must *appear* not to
  - Must roll-back transactions that use others' state
  - Means you have to record all changes to undo them
- **When deadlock detected just abort a transaction**
  - Breaks the dependency cycle

[1] Not applicable to topics in this lecture

Transactional memory

- **Some modern processors support *transactional memory***
- **Transactional Synchronization Extensions (TSX)** [intel1§16]
  - `xbegin abort_handler` – begins a transaction
  - `xend` – commit a transaction
  - `xabort $code` – abort transaction with 8-bit code
  - Note: nested transactions okay (also `xtest` tests if in transaction)
- **During transaction, processor tracks accessed memory**
  - Keeps read-set and write-set of cache lines
  - Nothing gets written back to memory during transaction
  - Transaction aborts (at `xend` or earlier) if any conflicts
  - Otherwise, all dirty cache lines are "written" atomically (in practice switch to non-transactional M state of MESI)

Using transactional memory

- **Idea 3: Use to get "free" fine-grained locking on a hash table**
  - E.g., concurrent inserts that don't touch same buckets are okay
  - Should *read* spinlock to make sure not taken (but not write) [Kim]
  - Hardware will detect there was no conflict
- **Can also use to poll for one of many asynchronous events**
  - Start transaction
  - Fill cache with values to which you want to see changes
  - Loop until a write causes your transaction to abort
- **Note: Transactions are never guaranteed to commit**
  - Might overflow cache, get false sharing, see weird processor issue
  - Means abort path must always be able to perform transaction (e.g., you do need a lock on your hash table)
- **Sadly, very few CPUs still support this**
  - Buggy implementations disabled through microcode updates

Hardware lock elision (HLE)

- **Idea: make it so spinlocks rarely need to spin**
  - Begin a transaction when you acquire lock
  - Other CPUs won't see lock acquired, can also enter critical section
  - Okay not to have mutual exclusion when no memory conflicts!
  - On conflict, abort and restart without transaction, thereby visibly acquiring lock (and aborting other concurrent transactions)
- **Intel support:**
  - Use `xacquire` prefix before `xchgl` (used for test and set)
  - Use `xrelease` prefix before `movl` that releases lock
  - Prefixes chosen to be noops on older CPUs (binary compatibility)
- **Hash table example:**
  - Use `xacquire xchgl` in table-wide test-and-set spinlock
  - Works correctly on older CPUs (with coarse-grained lock)
  - Allows safe concurrent accesses on newer CPUs!

Outline

Scalable interfaces

- Not all interfaces can scale
- How to tell which can and which can't?
- Scalable Commutativity Rule: *"Whenever interface operations commute, they can be implemented in a way that scales"* [Clements]

Are fork(), execve() broadly commutative?

```
pid_t pid = fork();
if (!pid)
  execlp("bash", "bash", NULL);
```


Are fork(), execve() broadly commutative?

```
pid_t pid = fork();
if (!pid)
  execlp("bash", "bash", NULL);
```

- No, `fork()` doesn't commute with memory writes, many file descriptor operations, and all address space operations
  - E.g., `close(fd); fork();` vs. `fork(); close(fd);`
- `execve()` often follows `fork()` and undoes most of `fork()`'s sub operations
- `posix_spawn()`, which combines `fork()` and `execve()` into a single operation, is broadly commutative
  - But obviously more complex, less flexible
  - Maybe Microsoft will have the last laugh?

Is open() broadly commutative?

```
int fd1 = open("foo", O_RDONLY);
int fd2 = open("bar", O_RDONLY);
```


Is open() broadly commutative?

```
int fd1 = open("foo", O_RDONLY);
int fd2 = open("bar", O_RDONLY);
```

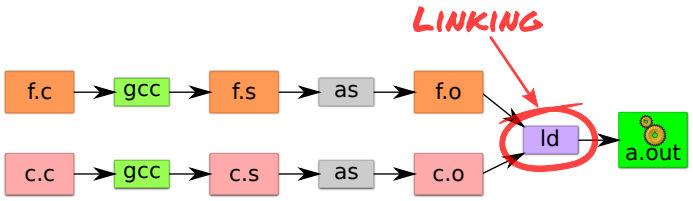
- Actually `open()` does not broadly commute!
- Does not commute with any system call (including itself) that creates a file descriptor
- Why? POSIX requires new descriptors to be assigned the lowest available integer
- If we fixed this, `open()` would commute, as long as it is not creating a file in the same directory as another operation

9. Linking

Administrivia

- **Lab 2 due Wednesday**
- **Midterm review section Friday**
- **Midterm exam in class next Monday May 5**
  - Open note, but no textbook or electronic devices
  - Bring lecture note printouts
  - SCPD must register exam monitor or show up in person (no need to request permission to show up in person)
  - Please remind us if you need OAE arrangements
  - Please send us your exam monitor if you are a non-SCPD with permission to take the exam under SCPD rules. (SCPD won't send the exam to your monitor, so we have to do it directly.)

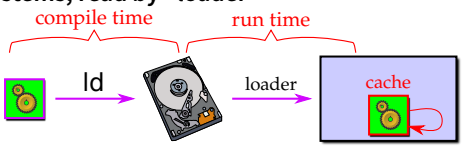
Today's Big Adventure



- **How to name and refer to things that don't exist yet**
- **How to merge separate name spaces into a cohesive whole**
- **More information:**
  - How to write shared libraries
  - Run "`nm`," "`objdump`," and "`readelf`" on a few .o and a.out files.
  - The ELF standard
  - Examine `/usr/include/elf.h`

How is a program executed?

- **On Unix systems, read by "loader"**



  - Reads all code/data segments into buffer cache; Maps code (read only) and initialized data (r/w) into addr space
  - Or…fakes process state to look like paged out
- **Lots of optimizations happen in practice:**
  - Zero-initialized data does not need to be read in.
  - Demand load: wait until code used before get from disk
  - Copies of same program running? Share code
  - Multiple programs use same routines: share code

x86 Assembly syntax

- **Linux uses AT&T assembler syntax – places destination last**
  - Be aware that *intel syntax* (used in manual) places destination first
- **Types of operand available:**
  - Registers start with "%" – `movl %edx,%eax`
  - Immediate values (constants) prefixed by "$" – `movl $0xff,%edx`
  - `(%reg)` is value at address in register *reg* – `movl (%edi),%eax`
  - *n*`(%reg)` is value at address in (register *reg*)+*n* – `movl 8(%ebp),%eax`
  - `*%reg` in an indirection through *reg* – `call *%eax`
  - Everything else is an address – `movl var,%eax; call printf`
- **Some heavily used instructions**
  - `movl` – moves (copies) value from source to destination
  - `pushl`/`popl` – pushes/pops value on stack
  - `call` – pushes next instruction address to stack and jumps to target
  - `ret` – pops address of stack and jumps to it
  - `leave` – equivalent to `movl %ebp,%esp; popl %ebp`

Perspectives on memory contents

- **Programming language view:** `x += 1;` `add $1, %eax`
  - Instructions: Specify operations to perform
  - Variables: Operands that can change over time
  - Constants: Operands that never change
- **Hardware view:**
  - executable: code, usually read-only
  - read only: constants (maybe one copy for all processes)
  - read/write: variables (each process needs own copy)
- **Need *addresses* to use data:**
  - Addresses locate things. Must update them when you move
  - Examples: linkers, garbage collectors, URL
- **Binding time: When is a value determined/computed?**
  - Early to late: Compile time, Link time, Load time, Runtime

Running example: hello program

- **Hello program**
  - Write friendly greeting to terminal
  - Exit cleanly
- **Every programming language addresses this problem**

[demo]

Running example: hello program

- **Hello program**
  - Write friendly greeting to terminal
  - Exit cleanly
- **Every programming language addresses this problem**
- **Concept should be familiar if you took 106B:**

```
int
main()
{
    cout << "Hello, world!" << endl;
}
```

- **Today's lecture: 80 minutes on hello world**

Hello world – CS212-style

```
#include <sys/syscall.h>
int my_errno;
const char greeting[] = "hello world\n";

int my_write(int fd, const void *buf, size_t len)
{
  int ret;
  asm volatile ("int $0x80" : "=a" (ret)
                : "0" (SYS_write),
                  "b" (fd), "c" (buf), "d" (len)
                : "memory");
  if (ret < 0) {
    my_errno = -ret;
    return -1;
  }
  return ret;
}

int main() { my_write (1, greeting, my_strlen(greeting)); }
```


Examining `hello1.s`

- **Grab the source and try it yourself**
  - `tar xzf /afs/ir.stanford.edu/class/cs212/hello.tar.gz`
- `gcc -S hello1.c` **produces assembly output in** `hello1.s`
- **Check the definitions of** `my_errno`, `greeting`, `main`, `my_write`
- `.globl` **symbol makes** *symbol* **global**
- **Sections of** `hello1.s` **are directed to various segments**
  - `.text` says put following contents into text segment
  - `.data`, `.rodata` says to put into data or read-only data
  - `.comm` *symbol,size,align* declares *symbol* and allows multiple definitions (like C but not C++, now requires `-fcommon` flag)
- **See how function calls push arguments to stack, then pop**

```
pushl   $greeting  # Argument to my_strlen is greeting
call    my_strlen  # Make the call (length now in %eax)
addl    $4, %esp   # Must pop greeting back off stack
```


Disassembling `hello1`

```
 my_write (1, greeting, my_strlen(greeting));
8049208:  68 08 a0 04 08     push   $0x804a008
804920d:  e8 93 ff ff ff     call   80491a5 <my_strlen>
8049212:  83 c4 04           add    $0x4,%esp
8049215:  50                 push   %eax
8049216:  68 08 a0 04 08     push   $0x804a008
804921b:  6a 01              push   $0x1
804921d:  e8 aa ff ff ff     call   80491cc <my_write>
8049222:  83 c4 0c           add    $0xc,%esp
```

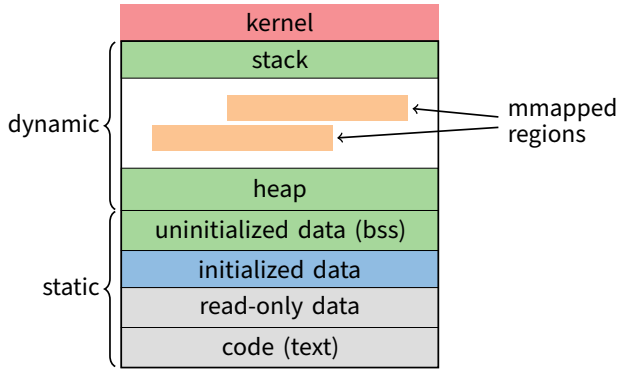
- **Disassemble from shell with** `objdump -Sr hello1`
- **Note** `push` **encodes address of greeting (0x804a008)**
- **Offsets in** `call` **instructions: 0xffffff93 = -109, 0xfffffaa = -86**
  - Binary encoding takes offset relative to next instruction

How is a process specified?

```
$ readelf -h hello1
ELF Header:
  ...
  Entry point address:               0x8049030
  Start of program headers:          52 (bytes into file)
  Start of section headers:          14968 (bytes into file)
  Number of program headers:         8
  Number of section headers:         23
  Section header string table index: 22
```

- **Executable files are the linker/loader interface. Must tell OS:**
  - What is code? What is data? Where should they live?
  - This is part of the purpose of the ELF standard
- **Every ELF file starts with ELF an** *header*
  - Specifies *entry point* virtual address at which to start executing
  - But how should the loader set up memory?

Recall what process memory looks like



- **Address space divided into "segments"**
  - Text, read-only data, data, bss, heap (dynamic data), and stack
  - Recall gcc told assembler in which segments to put what contents

Who builds what?

- **Heap: allocated and laid out at runtime by malloc**
  - Namespace constructed dynamically, managed by *programmer* (names stored in pointers, and organized using data structures)
  - Compiler, linker not involved other than saying where it can start
- **Stack: allocated at runtime (func. calls), layout by compiler**
  - Names are relative off of stack (or frame) pointer
  - Managed by compiler (alloc on procedure entry, free on exit)
  - Linker not involved because namespace entirely local: Compiler has enough information to build it.
- **Global data/code: allocated by compiler, layout by *linker***
  - Compiler emits them and names with symbolic references
  - Linker lays them out and translates references
- **Mmapped regions: Managed by programmer or linker**
  - Some programs directly call `mmap`; dynamic linker uses it, too

ELF program header

```
$ readelf -l hello1
Program Headers:
  Type    Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD    0x001000 0x08049000 0x08049000 0x00304 0x00304 R E 0x1000
  LOAD    0x002000 0x0804a000 0x0804a000 0x00158 0x00158 R   0x1000
  LOAD    0x002ff8 0x0804bff8 0x0804bff8 0x0001c 0x0003c RW  0x1000
  ...
 Section to Segment mapping:
  Segment Sections...
   01     ... .text ...
   02     .rodata ...
   03     ... .data .bss
```

- **For executables, the ELF header points to *program headers***
  - Says what segments of file to map where, with what permissions
- **Segment 03 has shorter file size then memory size**
  - Only 0x1c bytes must be read into memory from file
  - Remaining 0x20 bytes constitute the .bss
- **Who creates the program header? The linker**

Linkers (Linkage editors)

- **Unix: ld**
  - Usually hidden behind compiler
  - Run `gcc -v hello.c` to see ld or invoked (may see collect2)
- **Three functions:**
  - Collect together all pieces of a program
  - Coalesce like segments
  - Fix addresses of code and data so the program can run
- **Result: runnable program stored in new object file**
- **Why can't compiler do this?**
- **Usually linkers don't rearrange segments, but can**
  - E.g., re-order instructions for fewer cache misses; remove routines that are never called from `a.out`

Linkers (Linkage editors)

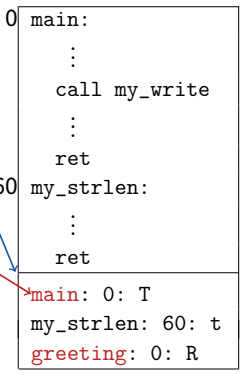
- **Unix: ld**
  - Usually hidden behind compiler
  - Run `gcc -v hello.c` to see ld or invoked (may see collect2)
- **Three functions:**
  - Collect together all pieces of a program
  - Coalesce like segments
  - Fix addresses of code and data so the program can run
- **Result: runnable program stored in new object file**
- **Why can't compiler do this?**
  - Limited world view: sees one file, rather than all files
- **Usually linkers don't rearrange segments, but can**
  - E.g., re-order instructions for fewer cache misses; remove routines that are never called from `a.out`

Simple linker: two passes needed

- **Pass 1:**
  - Coalesce like segments; arrange in non-overlapping memory
  - Read files' symbol tables, construct global symbol table with entry for every symbol used or defined
  - Compute virtual address of each segment (at start+offset)
- **Pass 2:**
  - Patch references using file and global symbol table
  - Emit result
- **Symbol table: information about program kept while linker running**
  - Segments: name, size, old location, new location
  - Symbols: name, input segment, offset within segment

Where to put emitted objects?

- **Assembler:**
  - Doesn't know where data/code should be placed in the process's address space
  - Assumes each segment starts at zero
  - Emits symbol table that holds the name and offset of each created object
  - Routines/variables exported by file are recorded as global definitions
- **Simpler perspective:**
  - Code is in a big byte array
  - Data is in another big byte array
  - Assembler creates (object name, index) tuple for each interesting thing
  - Linker then merges all of these arrays



Object files

```
$ objdump -Sr hello2.o
  ...
  48:   50                      push   %eax
  49:   68 00 00 00 00          push   $0x0
                        4a: R_386_32    greeting
  4e:   6a 01                   push   $0x1
  50:   e8 fc ff ff ff          call   51 <main+0x2a>
                        51: R_386_PC32  my_write
  55:   83 c4 10                add    $0x10,%esp
```

- **Let's create two-file program `hello2` with `my_write` in separate file**
  - Compiler and assembler can't possibly know final addresses
- **Notice `push` uses 0 as address of `greeting`**
- **And `call` uses -4 as address of `my_write`—why?**

Relative relocations

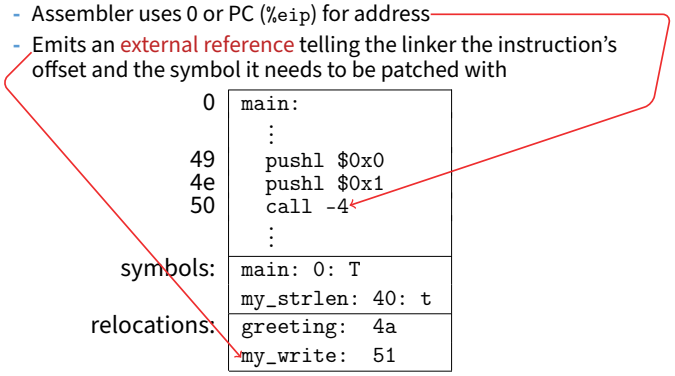
Null call without relocations

```
00000000 <.text>:
   0:   e8 00 00 00 00          call   0x5
   5:   58                      pop    %eax
```

- **Imagine a call to the very next instruction**
  - Doesn't affect control flow, just pushes return address on stack
- **Hardware expects offset 00 00 00 00 embedded in call**
- **Suppose we needed a relocation for the call (not shown)**
  - Linker computes relative offset from relocation to target
  - Target is byte 5, relocation at byte 1, so relative difference is 4
  - Linker will adds 4 to value found in object file
  - Hence, store -4 (0xfffffffc) in file to get linker result 00 00 00 00
- **Must compensate with -4 in binary regardless of the target**
  - Linker is relative to offset, hardware is relative to next instruction

Where is everything?

- **How to call procedures or reference variables?**
  - E.g., call to `my_write` needs a target addr
  - Assembler uses 0 or PC (`%eip`) for address
  - Emits an external reference telling the linker the instruction's offset and the symbol it needs to be patched with



- **At link time the linker patches every reference**

Relocations

```
$ readelf -r hello2.o
   ⋮
 Offset     Info    Type            Sym.Value  Sym. Name
00000039  00000801 R_386_32          00000000   greeting
0000004a  00000801 R_386_32          00000000   greeting
00000051  00000a02 R_386_PC32        00000000   my_write
   ⋮
```

- **Object file stores list of required relocations**
  - `R_386_32` says add symbol value to value already in file (often 0)
  - `R_386_PC32` says add difference between symbol value and patch location to value already in file (often -4 for `call`)
  - Info encodes type (low byte) and symbol index (<<8)
    (`Type` and `Sym. Name` are human-readable translation of `Info`)

ELF sections

```
$ readelf -S hello2.o
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 0000a4 00  AX  0   0  1
  [ 2] .rel.text         REL             00000000 0005f8 000018 08   I 20   1  4
  [ 3] .data             PROGBITS        00000000 0000d8 000000 00  WA  0   0  1
  [ 4] .bss              NOBITS          00000000 0000d8 000000 00  WA  0   0  1
  [ 5] .rodata           PROGBITS        00000000 0000d8 00000d 00   A  0   0  4
   ⋮
  [20] .symtab           SYMTAB          00000000 0004f0 0000d0 10     21   9  4
  [21] .strtab           STRTAB          00000000 0005c0 000038 00      0   0  1
```

- **Memory segments have corresponding PROGBITS file segments**
- **But relocations and symbol tables reside in segments, too**
- **Segments can be arrays of fixed-size data structures**
  - So strings referenced as offsets into special string segments
- **Remember ELF header had section header string table index**
  - That's so you can interpret names in section header

Symbol table

```
$ readelf -s hello2.o
   Num:    Value  Size Type    Bind   Vis      Ndx Name
           ⋮
     3: 00000000    39 FUNC    LOCAL  DEFAULT    1 my_strlen
           ⋮
     9: 00000000    13 OBJECT  GLOBAL DEFAULT    5 greeting
    10: 00000027    62 FUNC    GLOBAL DEFAULT    1 main
    11: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND my_write
           ⋮
```

- **Lists all global, exported symbols**
  - Sometimes local ones, too, for debugging (e.g., `my_strlen`)
- **Each symbol has an offset in a particular section number**
  - On previous slide, 1 = `.text`, 5 = `.rodata`
  - Special undefined section 0 means need symbol from other file

How to lay out emitted objects?

- **At link time, linker first:**
  - Coalesces all like segments (e.g., all `.text`, `.rodata`) from all files
  - Determines the size of each segment and the resulting address to place each object at
  - Stores all global definitions in a global symbol table that maps the definition to its final virtual address
- **Then in a second phase:**
  - Ensure each symbol has exactly 1 definition (except weak symbols, when compiling with `-fcommon`)
  - For each relocation:
    - Look up referenced symbol's virtual address in symbol table
    - Fix reference to reflect address of referenced symbol

What is a library?

- **A static library is just a collection of `.o` files**
- **Bind them together with `ar` program, much like `tar`**
  - E.g., `ar cr libmylib.a obj1.o obj2.o obj3.o`
  - On many OSes, run `ranlib libmylib.a` (to build index)
- **You can also list (`t`) and extract (`x`) files**
  - E.g., try: `ar tv /usr/lib/libc.a`
- **When linking a `.a` (archive) file, linker only pulls in needed files**
  - Ensures resulting executable can be smaller than big library
- **`readelf` will operate on every archive member (unweildy)**
  - But often convenient to disassemble with `objdump -d /usr/lib/libc.a`

Examining programs with nm

```
int uninitialized;
int initialized = 1;
const int constant = 2;
int main ()
{
  return 0;
}
```

```
$ nm a.out
...
0400400 T _start
04005bc R constant
0601008 W data_start
0601020 D initialized
04004b8 T main
0601028 B uninitialized
```

VA, symbol type

- **If don't need full `readelf`, can use `nm` (`nm -D` on shared objects)**
  - Handy `-o` flag prints file, useful with `grep`
- **`R` means read-only data (`.rodata` in elf)**
  - Note `constant` VA on same page as `main`
  - Share pages of read-only data just like text
- **`B` means uninitialized data in "BSS"**
- **Lower-case letters correspond to local symbols (static in C)**

Examining sections with objdump

Note Load mem addr. and File off have same page alignment for easy mmapping

```
$ objdump -h a.out
a.out:     file format elf64-x86-64
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
...
 12 .text         000001a8  0000000000400400  0000000000400400  00000400  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
...
 14 .rodata       00000008  00000000004005b8  00000000004005b8  000005b8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
...
 17 .ctors        00000010  0000000000600e18  0000000000600e18  00000e18  2**3
                  CONTENTS, ALLOC, LOAD, DATA
...
 23 .data         0000001c  0000000000601008  0000000000601008  00001008  2**3
                  CONTENTS, ALLOC, LOAD, DATA
...
 24 .bss          0000000c  0000000000601024  0000000000601024  00001024  2**2
                  ALLOC
...
```

No contents in file

- **Another portable alternative to `readelf`**

Name mangling

```
// C++
int foo (int a)
{
  return 0;
}

int foo (int a, int b)
{
  return 0;
}
```

```
% nm overload.o
0000000 T _Z3fooi
000000e T _Z3fooii
        U __gxx_personality_v0
% nm overload.o | c++filt
0000000 T foo(int)
000000e T foo(int, int)
        U __gxx_personality_v0
```

Mangling not compatible across compiler versions

Demangle names

- **C++ can have many functions with the same name**
- **Compiler therefore *mangles* symbols**
  - Makes a unique name for each function
  - Also used for methods/namespaces (`obj::fn`), template instantiations, & special functions such as `operator new`

Initialization and destruction

```
// C++
int a_foo_exists;
struct foo_t {
  foo_t () {
    a_foo_exists = 1;
  }
};
foo_t foo;
```

- **Initializers run before main**
  - Mechanism is platform-specific
- **Example implementation:**
  - Compiler emits static function in each file running initializers
  - Wrap linker with `collect2` program that generates `___main` function calling all such functions
  - Compiler inserts call to `___main` when compiling real `main`

```
% cc -S -o- ctor.C | c++filt
...
        .text
        .align 2
__static_initialization_and_destruction_0(int, int):
...
        call    foo_t::foo_t()
```

Other information in executables

```
// C++
struct foo_t {
  ~foo_t() {/*...*/}
  except() { throw 0; }
};
void fn ()
{
  foo_t foo;
  foo.except();
  /* ...  */
}
```

- Throwing exceptions destroys automatic variables
- During exception, must find
  - All such variables with non-trivial destructors
  - In all procedures' call frames until exception caught
- Record info in special sections
- Executables can include debug info (compile w. `-g`)
  - What source line does each binary instruction correspond to?

Dynamic (runtime) linking (`hello3.c`)

```
#include <dlfcn.h>
int main(int argc, char **argv, char **envp)
{
  size_t (*my_strlen)(const char *p);
  int (*my_write)(int, const void *, size_t);
  void *handle = dlopen("dest/libmy.so", RTLD_LAZY);
  if (!handle
      || !(my_strlen = dlsym(handle, "my_strlen"))
      || !(my_write = dlsym(handle, "my_write")))
    return 1;
  return my_write (1, greeting, my_strlen(greeting)) < 0;
}
```

- Link time isn't special, can link at runtime too
  - Get code (e.g., plugins) not available when program compiled
- Issues:
  - How can behavior differ compared to static linking?
  - Where to get unresolved symbols (e.g., `my_write`) from?
  - How does `my_write` know its own addresses (e.g., for `my_errno`)?

Dynamic linking (continued)

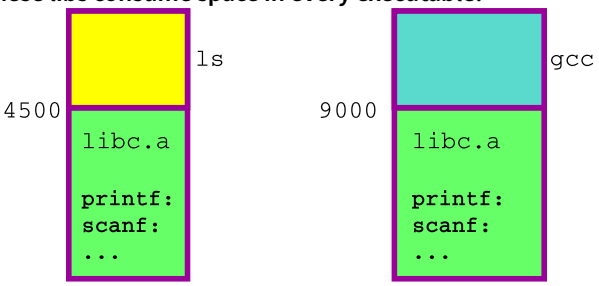
- How can behavior differ compared to static linking?
  - Runtime failure (can't find file, doesn't contain symbols)
  - No type checking of functions, variables
- Where to get unresolved symbols (e.g., `my_write`) from?
  - `dlsym` must parse ELF file to find symbols
- How does `my_write` know its own addresses?

```
$ readelf -r dest/libmy.so

Relocation section '.rel.dyn' at offset 0x20c contains 1 entry:
 Offset     Info    Type            Sym.Value  Sym. Name
00003ffc  00000106 R_386_GLOB_DAT    0000400c   my_errno
```

  - `dlopen`, too, must parse ELF to patch relocations

Static shared libraries

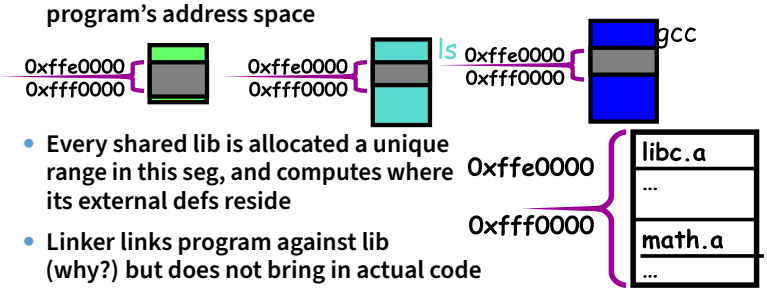
- Observation: everyone links in standard libraries (libc.a.), these libs consume space in every executable.



- Insight: we can have a single copy on disk if we don't actually include libc code in executable

Static shared libraries

- Define a "shared library segment" at same address in every program's address space



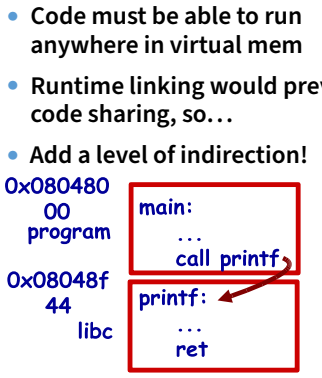
- Every shared lib is allocated a unique range in this seg, and computes where its external defs reside
- Linker links program against lib (why?) but does not bring in actual code
- Loader marks shared lib region as unreadable
- When process calls lib code, seg faults: embedded linker brings in lib code from known place & maps it in.
- Now different running programs can share code!

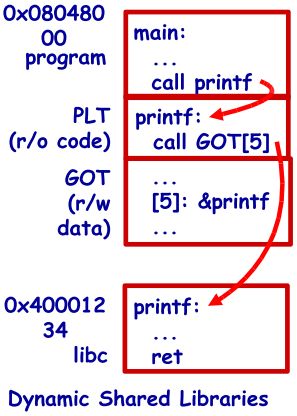
Dynamic shared libraries

- Static shared libraries require system-wide pre-allocation of address space
  - Clumsy, inconvenient
  - What if a library gets too big for its space? (fragmentation)
  - Can't upgrade libraries w/o relinking applications
  - Can space ever be reused?
- Solution: Dynamic shared libraries
  - Combine shared library and dynamic linking ideas
  - Any library can be loaded at any VA, chosen at runtime
- New problem: Linker won't know what names are valid
  - Solution: stub library
- New problem: How to call functions whose position varies?
  - Solution: next page…

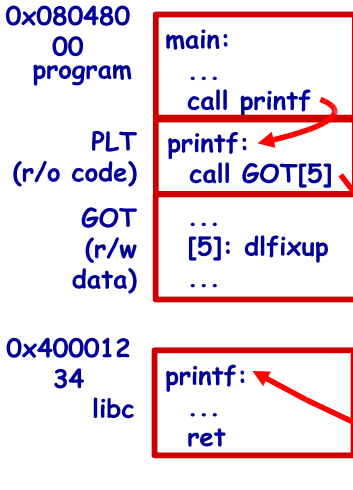
Position-independent code

- Code must be able to run anywhere in virtual mem
- Runtime linking would prevent code sharing, so…
- Add a level of indirection!





Lazy dynamic linking



- Linking all the functions at startup costs time
- Program might only call a few of them
- Only link each function on its first call

Dynamic linking with ELF

- **Every dynamically linked executable needs an *interpreter***
  - Embedded as string in special `.interp` section
  - `readelf -p .interp /bin/ls` → `/lib64/ld-linux-x86-64.so.2`
  - So all the kernel has to do is run `ld-linux`
- `dlfixup` **uses hash table to find symbols when needed**
- **Hash table lookups can be quite expensive** [Drepper]
  - E.g., big programs like OpenOffice very slow to start
  - Solution 1: Use a better hash function
    - linux added `.gnu.hash` section, later removed `.hash` sections
  - Solution 2: Export fewer symbols. Now fashionable to use:
    - `gcc -fvisibility=hidden` (keep symbols local to DSO)
    - `#pragma GCC visibility push(hidden)/visibility pop`
    - `__attribute__(visibility("default"))`, (override for a symbol)

Dynamic shared library example: `hello4`

```
$ objdump -Sr hello4
    ⋮
08049030 <my_write@plt>:
 8049030:       ff 25 0c c0 04 08       jmp    *0x804c00c
 8049036:       68 00 00 00 00          push   $0x0
 804903b:       e9 e0 ff ff ff          jmp    8049020 <.plt>

08049040 <my_strlen@plt>:
 8049040:       ff 25 10 c0 04 08       jmp    *0x804c010
 8049046:       68 08 00 00 00          push   $0x8
 804904b:       e9 d0 ff ff ff          jmp    8049020 <.plt>
    ⋮
 804917a:       68 08 a0 04 08          push   $0x804a008
 804917f:       e8 bc fe ff ff          call   8049040 <my_strlen@plt>
```

- `0x804c00c` **and** `0x804c010` **initially point to next instruction**
  - Calls `dlfixup` with relocation index
  - Note second `jmp` of each entry goes to 0th PLT entry, which jumps to `dlfixup`

`hello4` relocations

```
$ readelf -r hello4
Relocation section '.rel.plt' at offset 0x314 contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804c00c  00000107 R_386_JUMP_SLOT   00000000   my_write
0804c010  00000507 R_386_JUMP_SLOT   00000000   my_strlen
```

- **PLT =** ***procedure linkage table*** **on last slide**
  - Small 16 byte snippets, read-only executable code
- `dlfixup` **Knows how to parse relocations, symbol table**
  - Looks for symbols by name in hash tables of shared libraries
- `my_write` **&** `my_strlen` **are pointers in** ***global offset table*** **(GOT)**
  - GOT non-executable, read-write (so `dlfixup` can fix up)
- **Note** `hello4` **knows address of** `greeting`**, PLT, and GOT**
  - How does a shared object (`libmy.so`) find these?
  - PLT is okay because calls are relative
  - In PIC, compiler reserves one register `%ebx` for GOT address

`hello4` shared object contents

mywrite.c

```
int my_errno;
int my_write(int fd, const void *buf, size_t len) {
  int ret;
  asm volatile (/* ... */);
  if (ret < 0) {
    my_errno = -ret;
    return -1;
  }
  return ret;
}
```

mywrite.s

```
negl  %eax
movl  %eax, my_errno
```

mywrite-pic.s

```
negl  %eax
movl  %eax, %edx
movl  my_errno@GOT(%ebx), %eax
movl  %edx, (%eax)
```

How does %ebx get set?

mywrite-pic.s

```
my_write:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $16, %esp
        call    __x86.get_pc_thunk.bx
        addl    $_GLOBAL_OFFSET_TABLE_, %ebx
         ⋮
__x86.get_pc_thunk.bx:
        movl    (%esp), %ebx
        ret
```

```
$ readelf -r .libs/mywrite.o
 Offset     Info    Type            Sym.Value  Sym. Name
00000008 00000a02 R_386_PC32    00000000   __x86.get_pc_thunk.bx
0000000e 00000b0a R_386_GOTPC   00000000   _GLOBAL_OFFSET_TABLE_
00000036 0000082b R_386_GOT32X  00000000   my_errno
```


Linking and security

```
void fn ()
{
  char buf[80];
  gets (buf);
  /* ... */
}
```

1. **Attacker puts code in buf**
   - Overwrites return address to jump to code
2. **Attacker puts shell command above buf**
   - Overwrites return address so function "returns" to `system` function in libc

- **People try to address problem with linker**
- **W^X: No memory both writable and executable**
  - Prevents 1 but not 2, must be disabled for jits
- **Address space randomization**
  - Makes attack #2 a little harder, not impossible
  - Leads to position-independent executable, compiled `-fpie` and linked `-pie`—like PIC for executables
- **Also address with compiler (stack protector, CFI)**

Linking Summary

- **Compiler/Assembler: 1 object file for each source file**
  - Problem: incomplete world view
  - Where to put variables and code? How to refer to them?
  - Names definitions symbolically ("`printf`"), refers to routines/variable by symbolic name
- **Linker: combines all object files into 1 executable file**
  - Big lever: global view of everything. Decides where everything lives, finds all references and updates them
  - Important interface with OS: what is code, what is data, where is start point?
- **OS loader reads object files into memory:**
  - Allows optimizations across trust boundaries (share code)
  - Provides interface for process to allocate memory (`sbrk`)

Code = data, data = code

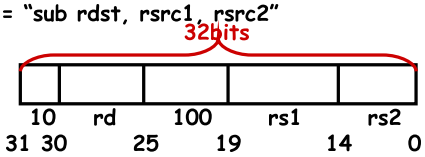
- **No inherent difference between code and data**
  - Code is just something that can be run through a CPU without causing an "illegal instruction fault"
  - Can be written/read at runtime just like data "dynamically generated code"
- **Why? Speed (usually)**
  - Big use: eliminate interpretation overhead. Gives 10-100x performance improvement
  - Example: Just-in-time Javascript compiler, or qemu vs. bochs
  - In general: optimizations thrive on information. More information at runtime.
- **The big tradeoff:**
  - Total runtime = code gen cost + cost of running code

How?

- **Determine binary encoding of desired instructions**

SPARC: sub instruction

symbolic = "sub rdst, rsrc1, rsrc2"



- **Write these integer values into a memory buffer**

```
unsigned code[1024], *cp = &code[0];
/* sub %g5, %g4, %g3 */
*cp++ = (2<<30) | (5<<25) | (4<<19) |(4<<14) | 3;
...
```

- **Use `mprotect` to disable W^X**
- **Jump to the address of the buffer:** `((int (*)())code)();`

```
/* (from glibc sysdeps/unix/sysv/linux/i386/sysdep.h)
   https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/unix/sysv/linux/i386/sysdep
.h

   Linux takes system call arguments in registers:

        syscall number  %eax         call-clobbered
        arg 1           %ebx         call-saved
        arg 2           %ecx         call-clobbered
        arg 3           %edx         call-clobbered
        arg 4           %esi         call-saved
        arg 5           %edi         call-saved
        arg 6           %ebp         call-saved
*/

#include <sys/syscall.h>

typedef unsigned long size_t;

int my_write(int, const void *, size_t);

int my_errno;

size_t
my_strlen(const char *p)
{
  size_t ret;
  for (ret = 0; p[ret]; ++ret)
    ;
  return ret;
}

int
my_write(int fd, const void *buf, size_t len)
{
  int ret;
  asm volatile ("int $0x80" : "=a" (ret)
                : "0" (SYS_write), "b" (fd), "c" (buf), "d" (len) : "memory");
  if (ret < 0) {
    my_errno = -ret;
    return -1;
  }
  return ret;
}

const char greeting[] = "hello world\n";
int
main(int argc, char **argv, char **envp)
{
  my_write (1, greeting, my_strlen(greeting));
}

void
__libc_start_main(int (*mainp)(int, char **, char **),
                  int argc, char **argv)
{
  mainp(argc, argv, argv + argc + 1);
  asm volatile ("int $0x80" :: "a" (SYS_exit), "b" (0));
}
```

```
#include <sys/syscall.h>

typedef unsigned long size_t;

int my_write(int, const void *, size_t);

static size_t
my_strlen(const char *p)
{
  size_t ret;
  for (ret = 0; p[ret]; ++ret)
    ;
  return ret;
}

const char greeting[] = "hello world\n";
int
main(int argc, char **argv, char **envp)
{
  my_write (1, greeting, my_strlen(greeting));
}

void
__libc_start_main(int (*mainp)(int, char **, char **),
                  int argc, char **argv)
{
  mainp(argc, argv, argv + argc + 1);
  asm volatile ("int $0x80" :: "a" (SYS_exit), "b" (0));
}
```

```
#include <dlfcn.h>
#include <sys/syscall.h>

const char greeting[] = "hello world\n";
int
main(int argc, char **argv, char **envp)
{
  size_t (*my_strlen)(const char *p);
  int (*my_write)(int, const void *, size_t);

  void *handle = dlopen("dest/libmy.so", RTLD_LAZY);
  if (!handle
      || !(my_strlen = dlsym(handle, "my_strlen"))
      || !(my_write = dlsym(handle, "my_write")))
    return 1;

  my_write (1, greeting, my_strlen(greeting));
  return 0;
}

void
__libc_start_main(int (*mainp)(int, char **, char **),
                  int argc, char **argv)
{
  mainp(argc, argv, argv + argc + 1);
  asm volatile ("int $0x80" :: "a" (SYS_exit), "b" (0));
}
```

```
#include <sys/syscall.h>

typedef unsigned long size_t;

int my_write(int, const void *, size_t);
size_t my_strlen(const char *p);

const char greeting[] = "hello world\n";
int
main(int argc, char **argv, char **envp)
{
  my_write (1, greeting, my_strlen(greeting));
}

void
__libc_start_main(int (*mainp)(int, char **, char **),
                  int argc, char **argv)
{
  mainp(argc, argv, argv + argc + 1);
  asm volatile ("int $0x80" :: "a" (SYS_exit), "b" (0));
}
```

```
typedef unsigned long size_t;

size_t
my_strlen(const char *p)
{
  size_t ret;
  for (ret = 0; p[ret]; ++ret)
    ;
  return ret;
}
```

```
#include <sys/syscall.h>

typedef unsigned long size_t;

int my_errno;

int
my_write(int fd, const void *buf, size_t len)
{
  int ret;
  asm volatile ("pushl %%ebx\n"      // older gcc before version 5
                "\tmovl %2,%%ebx\n"  // won't allow direct use of
                "\tint $0x80\n"      // %ebx in PIC code
                "\tpopl %%ebx"
                : "=a" (ret)
                : "0" (SYS_write), "g" (fd), "c" (buf), "d" (len) : "memory");
  if (ret < 0) {
    my_errno = -ret;
    return -1;
  }
  return ret;
}
```

```
        .file   "hello1.c"
        .text
        .globl  my_errno
        .bss
        .align 4
        .type   my_errno, @object
        .size   my_errno, 4
my_errno:
        .zero   4
        .text
        .globl  my_strlen
        .type   my_strlen, @function
my_strlen:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $0, -4(%ebp)
        jmp     .L2
.L3:
        addl    $1, -4(%ebp)
.L2:
        movl    8(%ebp), %edx
        movl    -4(%ebp), %eax
        addl    %edx, %eax
        movzbl  (%eax), %eax
        testb   %al, %al
        jne     .L3
        movl    -4(%ebp), %eax
        leave
        ret
        .size   my_strlen, .-my_strlen
        .globl  my_write
        .type   my_write, @function
my_write:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $16, %esp
        movl    $4, %eax
        movl    8(%ebp), %ebx
        movl    12(%ebp), %ecx
        movl    16(%ebp), %edx
#APP
# 36 "hello1.c" 1
        int $0x80
# 0 "" 2
#NO_APP
        movl    %eax, -8(%ebp)
        cmpl    $0, -8(%ebp)
        jns     .L6
        movl    -8(%ebp), %eax
        negl    %eax
        movl    %eax, my_errno
        movl    $-1, %eax
        jmp     .L7
.L6:
        movl    -8(%ebp), %eax
.L7:
        movl    -4(%ebp), %ebx
        leave
        ret
        .size   my_write, .-my_write
        .globl  greeting
        .section        .rodata
        .align 4
```

```
        .type   greeting, @object
        .size   greeting, 13
greeting:
        .string "hello world\n"
        .text
        .globl  main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   $greeting
        call    my_strlen
        addl    $4, %esp
        pushl   %eax
        pushl   $greeting
        pushl   $1
        call    my_write
        addl    $12, %esp
        movl    $0, %eax
        leave
        ret
        .size   main, .-main
        .globl  __libc_start_main
        .type   __libc_start_main, @function
__libc_start_main:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $4, %esp
        movl    12(%ebp), %eax
        addl    $1, %eax
        leal    0(,%eax,4), %edx
        movl    16(%ebp), %eax
        addl    %edx, %eax
        subl    $4, %esp
        pushl   %eax
        pushl   16(%ebp)
        pushl   12(%ebp)
        movl    8(%ebp), %eax
        call    *%eax
        addl    $16, %esp
        movl    $1, %eax
        movl    $0, %edx
        movl    %edx, %ebx
#APP
# 57 "hello1.c" 1
        int $0x80
# 0 "" 2
#NO_APP
        nop
        movl    -4(%ebp), %ebx
        leave
        ret
        .size   __libc_start_main, .-__libc_start_main
        .ident  "GCC: (GNU) 14.2.1 20250207"
        .section        .note.GNU-stack,"",@progbits
```

```
        .file   "hello4.c"
        .text
        .globl  greeting
        .section        .rodata
        .align 4
        .type   greeting, @object
        .size   greeting, 13
greeting:
        .string "hello world\n"
        .text
        .globl  main
        .type   main, @function
main:
        leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $4, %esp
        subl    $12, %esp
        pushl   $greeting
        call    my_strlen
        addl    $16, %esp
        subl    $4, %esp
        pushl   %eax
        pushl   $greeting
        pushl   $1
        call    my_write
        addl    $16, %esp
        movl    $0, %eax
        movl    -4(%ebp), %ecx
        leave
        leal    -4(%ecx), %esp
        ret
        .size   main, .-main
        .globl  __libc_start_main
        .type   __libc_start_main, @function
__libc_start_main:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $4, %esp
        movl    12(%ebp), %eax
        addl    $1, %eax
        leal    0(,%eax,4), %edx
        movl    16(%ebp), %eax
        addl    %edx, %eax
        subl    $4, %esp
        pushl   %eax
        pushl   16(%ebp)
        pushl   12(%ebp)
        movl    8(%ebp), %eax
        call    *%eax
        addl    $16, %esp
        movl    $1, %eax
        movl    $0, %edx
        movl    %edx, %ebx
#APP
# 20 "hello4.c" 1
        int $0x80
# 0 "" 2
#NO_APP
        nop
        movl    -4(%ebp), %ebx
```

```
        leave
        ret
        .size   __libc_start_main, .-__libc_start_main
        .ident  "GCC: (GNU) 14.2.1 20250207"
        .section        .note.GNU-stack,"",@progbits
```

```
        .file   "mywrite.c"
        .text
        .globl  my_errno
        .bss
        .align 4
        .type   my_errno, @object
        .size   my_errno, 4
my_errno:
        .zero   4
        .text
        .globl  my_write
        .type   my_write, @function
my_write:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $4, %eax
        movl    12(%ebp), %ecx
        movl    16(%ebp), %edx
#APP
# 11 "mywrite.c" 1
        pushl %ebx
        movl 8(%ebp),%ebx
        int $0x80
        popl %ebx
# 0 "" 2
#NO_APP
        movl    %eax, -4(%ebp)
        cmpl    $0, -4(%ebp)
        jns     .L2
        movl    -4(%ebp), %eax
        negl    %eax
        movl    %eax, my_errno
        movl    $-1, %eax
        jmp     .L3
.L2:
        movl    -4(%ebp), %eax
.L3:
        leave
        ret
        .size   my_write, .-my_write
        .ident  "GCC: (GNU) 14.2.1 20250207"
        .section        .note.GNU-stack,"",@progbits
```

```
        .file   "mywrite.c"
        .text
        .globl  my_errno
        .bss
        .align 4
        .type   my_errno, @object
        .size   my_errno, 4
my_errno:
        .zero   4
        .text
        .globl  my_write
        .type   my_write, @function
my_write:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $16, %esp
        call    __x86.get_pc_thunk.bx
        addl    $_GLOBAL_OFFSET_TABLE_, %ebx
        movl    $4, %eax
        movl    12(%ebp), %ecx
        movl    16(%ebp), %edx
#APP
# 11 "mywrite.c" 1
        pushl %ebx
        movl 8(%ebp),%ebx
        int $0x80
        popl %ebx
# 0 "" 2
#NO_APP
        movl    %eax, -8(%ebp)
        cmpl    $0, -8(%ebp)
        jns     .L2
        movl    -8(%ebp), %eax
        negl    %eax
        movl    %eax, %edx
        movl    my_errno@GOT(%ebx), %eax
        movl    %edx, (%eax)
        movl    $-1, %eax
        jmp     .L3
.L2:
        movl    -8(%ebp), %eax
.L3:
        movl    -4(%ebp), %ebx
        leave
        ret
        .size   my_write, .-my_write
        .section        .text.__x86.get_pc_thunk.bx,"axG",@progbits,__x86.get_pc_thunk.
bx,comdat
        .globl  __x86.get_pc_thunk.bx
        .hidden __x86.get_pc_thunk.bx
        .type   __x86.get_pc_thunk.bx, @function
__x86.get_pc_thunk.bx:
        movl    (%esp), %ebx
        ret
        .ident  "GCC: (GNU) 14.2.1 20250207"
        .section        .note.GNU-stack,"",@progbits
```

10. Memory allocation

Administrivia

- **Lab 2 due Wednesday**
- **Midterm review section Friday**
- **Midterm exam in class next Monday May 5**
  - Open note, but no textbook or electronic devices
  - Bring lecture note printouts
  - SCPD must register exam monitor or show up in person (no need to request permission to show up in person)
  - Please remind us if you need OAE arrangements
  - Please send us your exam monitor if you are a non-SCPD with permission to take the exam under SCPD rules. (SCPD won't send the exam to your monitor, so we have to do it directly.)
- **My office hours this Friday 3pm, not Monday**
  - Come with questions for midterm
  - I'll also monitor Lectures+Exams tag on edstem

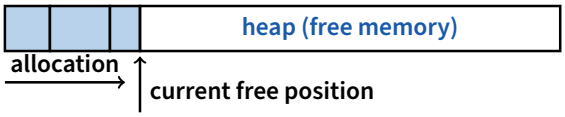
Outline

Dynamic memory allocation

- **Almost every useful program uses it**
  - Gives wonderful functionality benefits
    - Don't have to statically specify complex data structures
    - Can have data grow as a function of input size
    - Allows recursive procedures (stack growth)
  - But, can have a huge impact on performance
- **Today: how to implement it**
  - Lecture based on [Wilson]
- **Some interesting facts:**
  - Two or three line code change can have huge, non-obvious impact on how well allocator works (examples to come)
  - Proven: impossible to construct an "always good" allocator
  - Surprising result: memory management still poorly understood

Why is it hard?

- **Satisfy arbitrary set of allocation and frees.**
- **Easy without free: set a pointer to the beginning of some big chunk of memory ("heap") and increment on each allocation:**



- **Problem: free creates holes ("fragmentation")**
  Result? Lots of free space but cannot satisfy request!

More abstractly

- **What an allocator must do?**

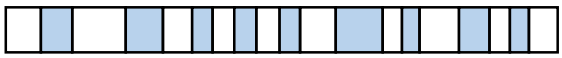
  - Track which parts of memory in use, which parts are free
  - Ideal: no wasted space, no time overhead
- **What the allocator cannot do?**
  - Control order of the number and size of requested blocks
  - Know the number, size, or lifetime of future allocations
  - Move allocated regions (bad placement decisions permanent)

`malloc(20)`?

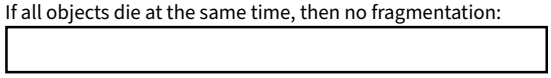
- **The core fight: minimize fragmentation**
  - App frees blocks in any order, creating holes in "heap"
  - Holes too small? cannot satisfy future requests

What is fragmentation really?

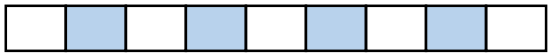
- **Inability to use memory that is free**
- **Two factors required for fragmentation**
  1. Different lifetimes—if adjacent objects die at different times, then fragmentation:



     - If all objects die at the same time, then no fragmentation:

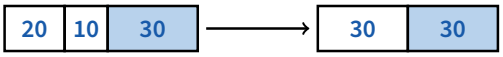


  2. Different sizes: If all requests the same size, then no fragmentation (that's why no external fragmentation with paging):



Important decisions

- **Placement choice: where in free memory to put a requested block?**
  - Freedom: can select any memory in the heap
  - Ideal: put block where it won't cause fragmentation later (impossible in general: requires future knowledge)
- **Split free blocks to satisfy smaller requests?**
  - Fights internal fragmentation
  - Freedom: can choose any larger block to split
  - One way: choose block with smallest remainder (best fit)
- **Coalescing free blocks to yield larger blocks**



  - Freedom: when to coalesce (deferring can save work)
  - Fights external fragmentation

Impossible to "solve" fragmentation

- **If you read allocation papers to find the best allocator**
  - All discussions revolve around tradeoffs
  - The reason? There cannot be a best allocator
- **Theoretical result:**
  - For any possible allocation algorithm, there exist streams of allocation and deallocation requests that defeat the allocator and force it into severe fragmentation.
- **How much fragmentation should we tolerate?**
  - Let M = bytes of live data, n_{min} = smallest allocation, n_{max} = largest – How much gross memory required?
  - Bad allocator: $M \cdot (n_{\text{max}}/n_{\text{min}})$
    - E.g., only ever use a memory location for a single size
    - E.g., make all allocations of size n_{max} regardless of requested size
  - Good allocator: $\sim M \cdot \log(n_{\text{max}}/n_{\text{min}})$

Pathological examples

- **Suppose heap currently has 7 20-byte chunks**

  - What's a bad stream of frees and then allocates?
- **Given a 128-byte limit on malloced space**
  - What's a really bad combination of mallocs & frees?
- **Next: two allocators (best fit, first fit) that, in practice, work pretty well**
  - "pretty well" = ∼20% fragmentation under many workloads

Pathological examples

- **Suppose heap currently has 7 20-byte chunks**

  - What's a bad stream of frees and then allocates?
  - Free every other chunk, then alloc 21 bytes
- **Given a 128-byte limit on malloced space**
  - What's a really bad combination of mallocs & frees?
- **Next: two allocators (best fit, first fit) that, in practice, work pretty well**
  - "pretty well" = ∼20% fragmentation under many workloads

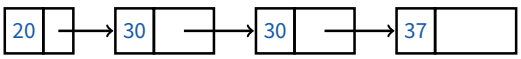
Pathological examples

- **Suppose heap currently has 7 20-byte chunks**

  - What's a bad stream of frees and then allocates?
  - Free every other chunk, then alloc 21 bytes
- **Given a 128-byte limit on malloced space**
  - What's a really bad combination of mallocs & frees?
  - Malloc 128 1-byte chunks, free every other
  - Malloc 32 2-byte chunks, free every other (1- & 2-byte) chunk
  - Malloc 16 4-byte chunks, free every other chunk…
- **Next: two allocators (best fit, first fit) that, in practice, work pretty well**
  - "pretty well" = ∼20% fragmentation under many workloads

Best fit

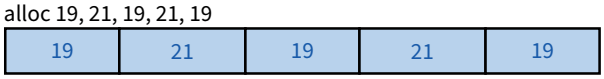
- **Strategy: minimize fragmentation by allocating space from block that leaves smallest fragment**
  - Data structure: heap is a list of free blocks, each has a header holding block size and a pointer to the next block



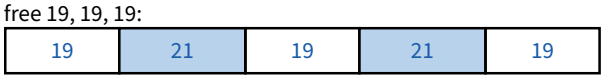
  - Code: Search freelist for block closest in size to the request. (Exact match is ideal)
  - During free (usually) coalesce adjacent blocks
- **Potential problem: Sawdust**
  - Remainder so small that over time left with "sawdust" everywhere
  - Fortunately not a problem in practice

Best fit gone wrong

- **Simple bad case: allocate n, m ($n < m$) in alternating orders, free all the ns, then try to allocate an $n + 1$**
- **Example: start with 99 bytes of memory**
  - alloc 19, 21, 19, 21, 19



  - free 19, 19, 19:



  - alloc 20? Fails! (wasted space = 57 bytes)
- **However, doesn't seem to happen in practice**

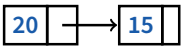
First fit

- **Strategy: pick the first block that fits**
  - Data structure: free list, sorted LIFO, FIFO, or by address
  - Code: scan list, take the first one
- **LIFO: put free object on front of list.**
  - Simple, but causes higher fragmentation
  - Potentially good for cache locality
- **Address sort: order free blocks by address**
  - Makes coalescing easy (just check if next block is free)
  - Also preserves empty/idle space (locality good when paging)
- **FIFO: put free object at end of list**
  - Gives similar fragmentation as address sort, but unclear why

Subtle pathology: LIFO FF

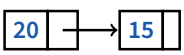
- **Storage management example of subtle impact of simple decisions**
- **LIFO first fit seems good:**
  - Put object on front of list (cheap), hope same size used again (cheap + good locality)
- **But, has big problems for simple allocation patterns:**
  - E.g., repeatedly intermix short-lived $2n$-byte allocations, with long-lived $(n + 1)$-byte allocations
  - Each time large object freed, a small chunk will be quickly taken, leaving useless fragment. Pathological fragmentation

First fit: Nuances

- **First fit sorted by address order, in practice:**
  - Blocks at front preferentially split, ones at back only split when no larger one found before them
  - Result? Seems to roughly sort free list by size
  - So? Makes first fit operationally similar to best fit: a first fit of a sorted list = best fit!
- **Problem: sawdust at beginning of the list**
  - Sorting of list forces a large requests to skip over many small blocks. Need to use a scalable heap organization
- **Suppose memory has free blocks:** 

  - If allocation ops are 10 then 20, best fit wins
  - When is FF better than best fit?

First fit: Nuances

- **First fit sorted by address order, in practice:**
  - Blocks at front preferentially split, ones at back only split when no larger one found before them
  - Result? Seems to roughly sort free list by size
  - So? Makes first fit operationally similar to best fit: a first fit of a sorted list = best fit!
- **Problem: sawdust at beginning of the list**
  - Sorting of list forces a large requests to skip over many small blocks. Need to use a scalable heap organization
- **Suppose memory has free blocks:** 

  - If allocation ops are 10 then 20, best fit wins
  - When is FF better than best fit?
  - Suppose allocation ops are 8, 12, then 12 $\Longrightarrow$ first fit wins

Some worse ideas

- **Worst-fit:**
  - Strategy: fight against sawdust by splitting blocks to maximize leftover size
  - In real life seems to ensure that no large blocks around
- **Next fit:**
  - Strategy: use first fit, but remember where we found the last thing and start searching from there
  - Seems like a good idea, but tends to break down entire list
- **Buddy systems:**
  - Round up allocations to power of 2 to make management faster
  - Result? Heavy internal fragmentation

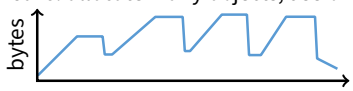
Outline

1. Malloc and fragmentation
2. Exploiting program behavior
3. Allocator designs
4. User-level MMU tricks
5. Garbage collection

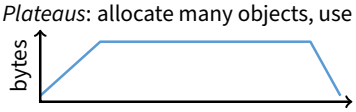
Known patterns of real programs

- **So far we've treated programs as black boxes.**
- **Most real programs exhibit 1 or 2 (or all 3) of the following patterns of alloc/dealloc:**
  - *Ramps*: accumulate data monotonically over time

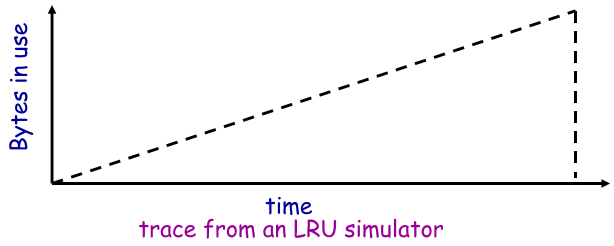
  - *Peaks*: allocate many objects, use briefly, then free all



  - *Plateaus*: allocate many objects, use for a long time

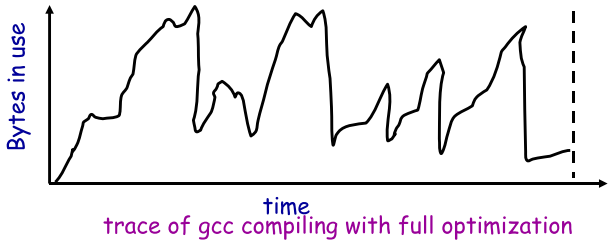


Pattern 1: ramps



- **In a practical sense: ramp = no free!**
  - Implication for fragmentation?
  - What happens if you evaluate allocator with ramp programs only?

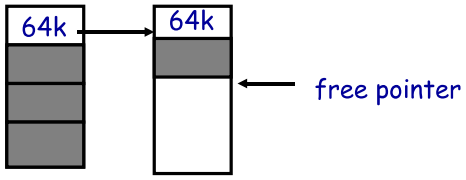
Pattern 2: peaks



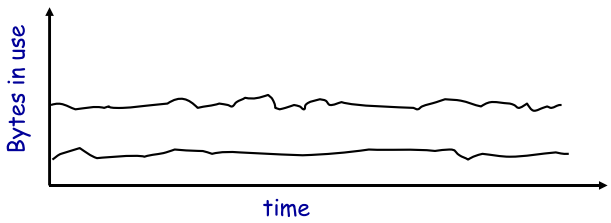
- **Peaks: allocate many objects, use briefly, then free all**
  - Fragmentation a real danger
  - What happens if peak allocated from contiguous memory?
  - Interleave peak & ramp? Interleave two different peaks?

Exploiting peaks

- **Peak phases: allocate a lot, then free everything**
  - Change allocation interface: allocate as before, but only support free of everything all at once
  - Called "arena allocation", "obstack" (object stack), or `alloca`/procedure call (by compiler people)
- **Arena = a linked list of large chunks of memory**
  - Advantages: alloc is a pointer increment, free is "free"
    No wasted space for tags or list pointers



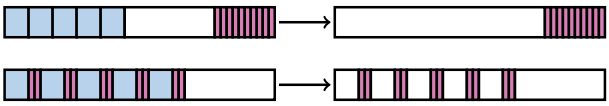
Pattern 3: Plateaus



- **Plateaus: allocate many objects, use for a long time**
  - What happens if overlap with peak or different plateau?

Fighting fragmentation

- **Segregation = reduced fragmentation:**
  - Allocated at same time ∼ freed at same time
  - Different type ∼ freed at different time



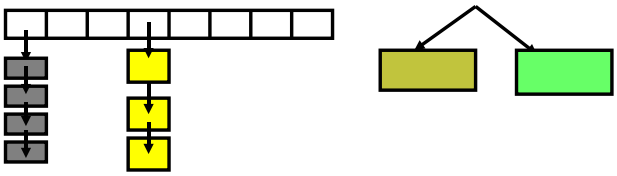
- **Implementation observations:**
  - Programs allocate a small number of different sizes
  - Fragmentation at peak usage more important than at low usage
  - Most allocations small (< 10 words)
  - Work done with allocated memory increases with size
  - Implications?

Outline

Slab allocation [Bonwick]

- **Kernel allocates many instances of same structures**
  - E.g., a 1.7 kB `task_struct` for every process on system
- **Often want contiguous *physical* memory (for DMA)**
- **Slab allocation optimizes for this case:**
  - A slab is multiple pages of contiguous physical memory
  - A cache contains one or more slabs
  - Each cache stores only one kind of object (fixed size)
- **Each slab is full, empty, or partial**
- **E.g., need new `task_struct`?**
  - Look in the `task_struct` cache
  - If there is a partial slab, pick free `task_struct` in that
  - Else, use empty, or may need to allocate new slab for cache
- **Advantages: speed, and no internal fragmentation**

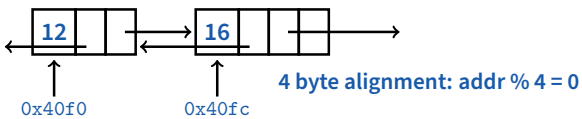
Simple, fast segregated free lists



- **Array of free lists for small sizes, tree for larger**
  - Place blocks of same size on same page
  - Have count of allocated blocks: if goes to zero, can return page
- **Pro: segregate sizes, no size tag, fast small alloc**
- **Con: worst case waste: 1 page per size even w/o free, After pessimal free: waste 1 page per object**
- **TCMalloc [Ghemawat] is a well-documented malloc like this**
  - Also uses "thread caching" to reduce coherence misses

Typical space overheads

- **Free list bookkeeping and alignment determine minimum allocatable size:**
- **If not implicit in page, must store size of block**
- **Must store pointers to next and previous freelist element**

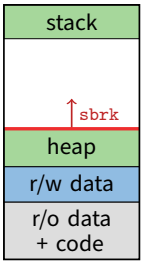


- **Allocator doesn't know types**
  - Must align memory to conservative boundary
- **Minimum allocation unit? Space overhead when allocated?**

[demo mtest]

Getting more space from OS

- **On Unix, can use `sbrk`**
  - E.g., to activate a new zero-filled page:




```
/* add nbytes of valid virtual address space */
void *get_free_space(size_t nbytes) {
  void *p = sbrk(nbytes);
  if (p == (void *) -1)
    error("virtual memory exhausted");
  return p;
}
```

- **For large allocations, `sbrk` a bad idea**
  - May want to give memory back to OS
  - Can't with `sbrk` unless big chunk last thing allocated
  - So allocate large chunk using `mmap`'s `MAP_ANON`

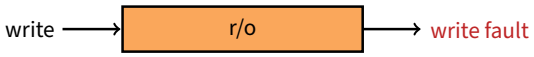
Outline

1. Malloc and fragmentation
2. Exploiting program behavior
3. Allocator designs
4. User-level MMU tricks
5. Garbage collection

Faults + resumption = power

- **Resuming after fault lets us emulate many things**
  - "All problems in CS can be solved by another layer of indirection"
- **Example: sub-page protection**
- **To protect sub-page region in paging system:**

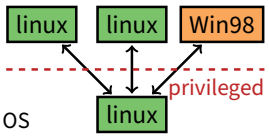
  - Set entire page to most restrictive permission; record in PT



  - Any access that violates permission will cause a fault
  - Fault handler checks if page special, and if so, if access allowed
  - Allowed? Emulate write ("tracing"), otherwise raise error

More fault resumption examples

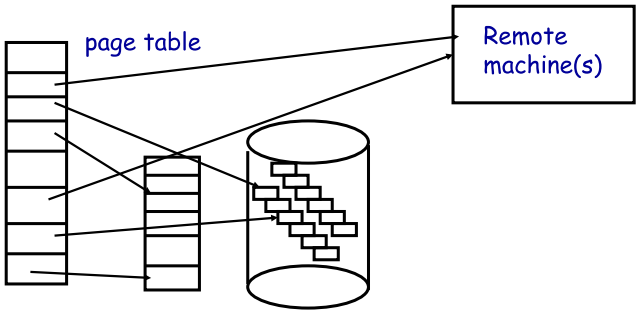
- **Emulate accessed bits:**
  - Set page permissions to "invalid".
  - On any access will get a fault: Mark as accessed
- **Avoid save/restore of floating point registers**
  - Make first FP operation cause fault so as to detect usage
- **Emulate non-existent instructions:**
  - Give inst an illegal opcode; OS fault handler detects and emulates fake instruction
- **Run OS on top of another OS!**
  - Slam OS into normal process
  - When does something "privileged," real OS gets woken up with a fault.
  - If operation is allowed, do it or emulate it; otherwise kill guest
  - IBM's VM/370. Vmware (sort of)



Not just for kernels

- **User-level code can resume after faults, too. Recall:**
  - `mprotect` – protects memory
  - `sigaction` – catches signal after page fault
  - Return from signal handler restarts faulting instruction
- **Many applications detailed by [Appel & Li]**
- **Example: concurrent snapshotting of process**
  - Mark all of process's memory read-only with `mprotect`
  - One thread starts writing all of memory to disk
  - Other thread keeps executing
  - On fault – write that page to disk, make writable, resume

Distributed shared memory



- **Virtual memory allows us to go to memory or disk**
  - But, can use the same idea to go anywhere! Even to another computer. Page across network rather than to disk. Faster, and allows network of workstations (NOW)

Persistent stores

- **Idea: Objects that persist across program invocations**
  - E.g., object-oriented database; useful for CAD/CAM type apps
- **Achieve by memory-mapping a file**
  - Write your own "malloc" for memory in a file
- **But only write changes to file at end if commit**
  - Use dirty bits to detect which pages must be written out
  - Or emulate dirty bits with *mprotect*/*sigaction* (using write faults)
- **On 32-bit machine, store can be larger than memory**
  - But single run of program won't access $>$ 4GB of objects
  - Keep mapping of 32-bit memory pointers $\leftrightarrow$ 64-bit disk offsets
  - Use faults to bring in pages from disk as necessary
  - After reading page, translate pointers—known as *swizzling*

Outline

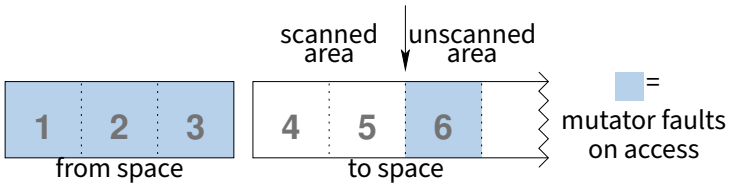
1. Malloc and fragmentation
2. Exploiting program behavior
3. Allocator designs
4. User-level MMU tricks
5. Garbage collection

Garbage collection

- **In safe languages, runtime knows about all pointers**
  - So can move an object if you change all the pointers
- **What memory locations might a program access?**
  - Any globals or objects whose pointers are currently in registers
  - Recursively, any pointers in objects it might access
  - Anything else is *unreachable*, or *garbage*; memory can be re-used
- **Example: stop-and-copy garbage collection**
  - Memory full? Temporarily pause program, allocate new heap
  - Copy all objects pointed to by registers into new heap
    - Mark old copied objects as copied, record new location
  - Start scanning through new heap. For each pointer:
    - Copied already? Adjust pointer to new location
    - Not copied? Then copy it and adjust pointer
  - Free old heap—program will never access it—and continue

Concurrent garbage collection

- **Idea: Stop & copy, but without the stop**
  - *Mutator* thread runs program, *collector* concurrently does GC
- **When collector invoked:**
  - Protect from space & unscanned to space from mutator
  - Copy objects in registers into *to space*, resume mutator
  - All pointers in scanned *to space* point to *to space*
  - If mutator accesses unscanned area, fault, scan page, resume



(See [Appel & Li].)

Heap overflow detection

- **Many GCed languages need fast allocation**
  - E.g., in lisp, constantly allocating cons cells
  - Allocation can be as often as every 50 instructions
- **Fast allocation is just to bump a pointer**

```
char *next_free;
char *heap_limit;

void *alloc (unsigned size) {
  if (next_free + size > heap_limit) /* 1 */
    invoke_garbage_collector ();     /* 2 */
  char *ret = next_free;
  next_free += size;
  return ret;
}
```

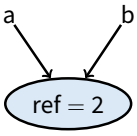
- **But would be even faster to eliminate lines 1 & 2!**

Heap overflow detection 2

- **Mark page at end of heap inaccessible**
  - `mprotect (heap_limit, PAGE_SIZE, PROT_NONE);`
- **Program will allocate memory beyond end of heap**
- **Program will use memory and fault**
  - Note: Depends on specifics of language
  - But many languages will touch allocated memory immediately
- **Invoke garbage collector**
  - Must now put just allocated object into new heap
- **Note: requires more than just resumption**
  - Faulting instruction must be resumed
  - But must resume with different target virtual address
  - Doable on most architectures since GC updates registers

Reference counting

- **Seemingly simpler GC scheme:**
  - Each object has "ref count" of pointers to it
  - Increment when pointer set to it
  - Decremented when pointer killed (C++ destructors handy—c.f. `shared_ptr`)

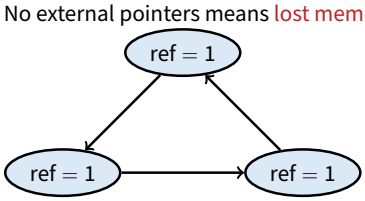



```
void foo(bar c) {
  bar a b;
  a = c;    // c.refcnt++
  b = a;    // a.refcnt++
  a = 0;    // c.refcnt--
  return;   // b.refcnt--
}
```

  - ref count == 0? Free object
- **Works well for hierarchical data structures**
  - E.g., pages of physical memory

Reference counting pros/cons

- **Circular data structures always have ref count > 0**
  - No external pointers means lost memory



- **Can do manually w/o PL support, but error-prone**
- **Potentially more efficient than real GC**
  - No need to halt program to run collector
  - Avoids weird unpredictable latencies
- **Potentially less efficient than real GC**
  - With real GC, copying a pointer is cheap
  - With refcounts, must update count each time & possibly take lock (but C++11 std::move can avoid overhead)

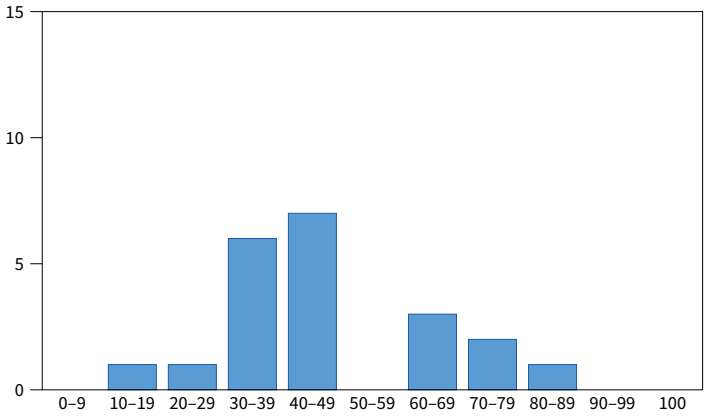
Ownership types

- **Another approach: avoid GC by exploiting type system**
  - Use ownership types, which prohibit copies
- **You can move a value into a new variable (e.g., copy pointer)**
  - But then the original variable is no longer usable
- **You can *borrow* a value by creating a pointer to it**
  - But must prove pointer will not outlive borrowed value
  - And can't use original unless both are read-only (to avoid races)
- **Ownership types available now in Rust language**
  - First serious competitor to C/C++ for OSes, browser engines
- **C++11 does something similar but weaker with unique types**
  - `std::unique_ptr`, `std::unique_lock`,...
  - Can std::move but not copy these

```
#include <stdio.h>
#include <stdlib.h>

int
main()
{
  char *p1 = malloc(1);
  char *p2 = malloc(1);
  printf("%p - %p = %ld\n", p2, p1, p2 - p1);
}
```

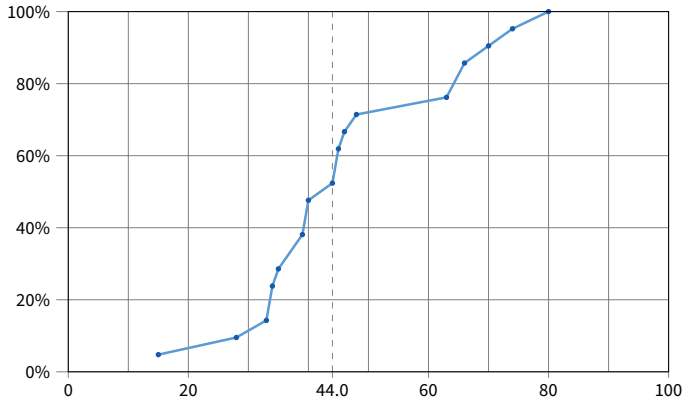
11. I/O and disks

Midterm results



- **Mean: 46.8571, median: 44.0**

Midterm results



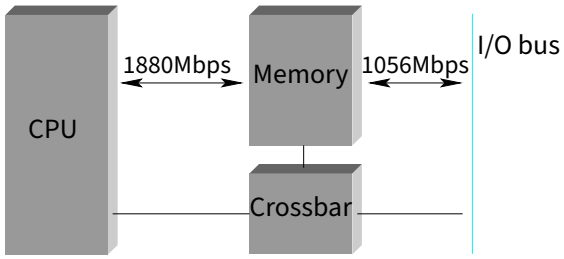
- **Systems students should insist on a CDF!**

Administrivia

- **Recall we will have a resurrection final**
  - Don't panic if you didn't do well on midterm
  - But make sure you understand all the answers
  - There may be questions on same topics on the final
  - Be sure to attend lecture for resurrection final if you are not SCPD
- **Lab 3 section Friday**

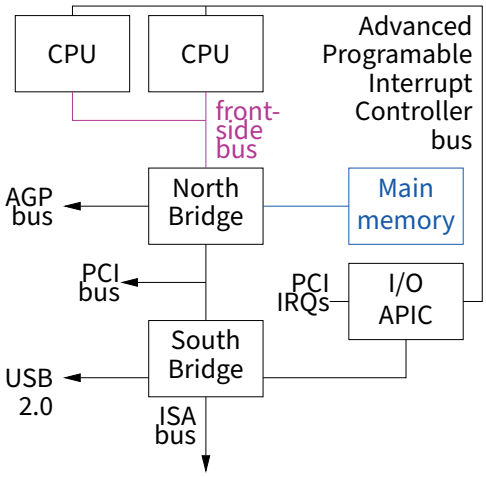
Outline

Old-school memory and I/O buses

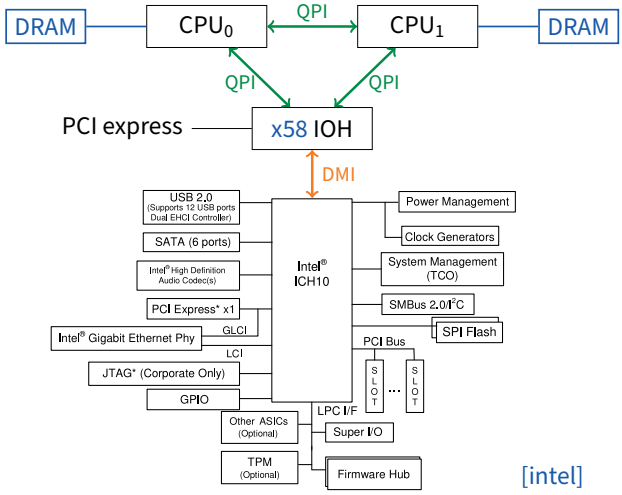


- **CPU accesses physical memory over a bus**
- **Devices access memory over I/O bus with DMA**
- **Devices can appear to be a region of memory**

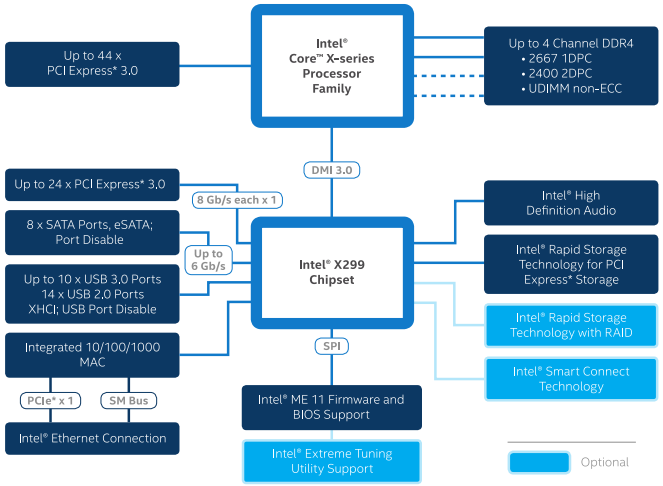
Realistic ~2005 PC architecture



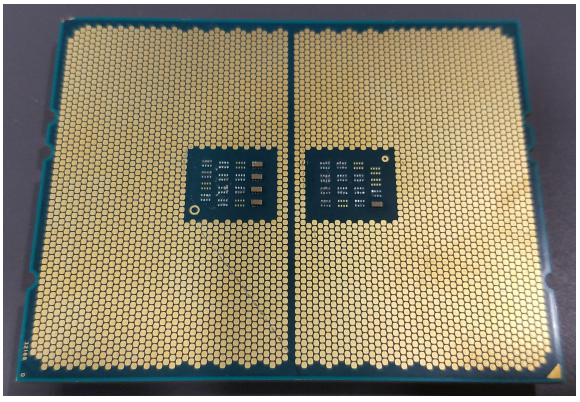
Modern PC architecture (intel)



CPU now entirely subsumes IOH [intel]



AMD EPYC is essentially an SoC

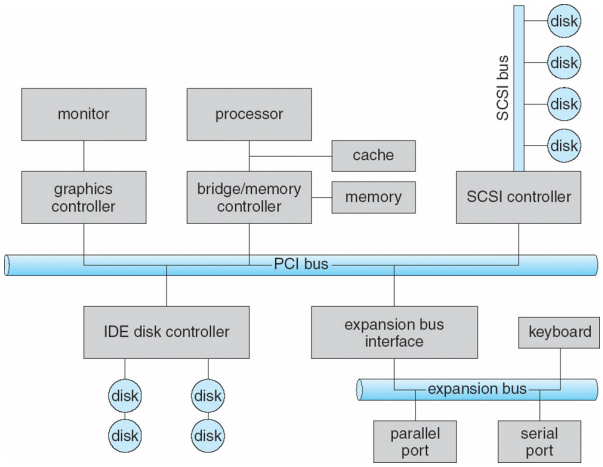


- **4094 pins: both memory controller and 128 lanes PCIe directly on chip!**

What is memory?

- **SRAM – Static RAM**
  - Like two NOT gates circularly wired input-to-output
  - 4–6 transistors per bit, actively holds its value
  - Very fast, used to cache slower memory
- **DRAM – Dynamic RAM**
  - A capacitor + gate, holds charge to indicate bit value
  - 1 transistor per bit – extremely dense storage
  - Charge leaks – need slow comparator to decide if bit 1 or 0
  - Must re-write charge after reading, and periodically refresh
- **VRAM – "Video RAM"**
  - Dual ported DRAM, can write while someone else reads

What is I/O bus? E.g., PCI



Outline

1. PC system architecture
2. Driver architecture
3. Disks
4. Disk scheduling
5. Flash

Communicating with a device

- **Memory-mapped device registers**
  - Certain *physical* addresses correspond to device registers
  - Load/store gets status/sends instructions – not real memory
- **Device memory – device may have memory OS can write to directly on other side of I/O bus**
- **Special I/O instructions**
  - Some CPUs (e.g., x86) have special I/O instructions
  - Like load & store, but asserts special I/O pin on CPU
  - OS can allow user-mode access to I/O ports at byte granularity
- **DMA – place instructions to card in main memory**
  - Typically then need to "poke" card by writing to register
  - Overlaps unrelated computation with moving data over (typically slower than memory) I/O bus

x86 I/O instructions

```
static inline uint8_t
inb (uint16_t port)
{
  uint8_t data;
  asm volatile ("inb %w1, %b0" : "=a" (data) : "Nd" (port));
  return data;
}

static inline void
outb (uint16_t port, uint8_t data)
{
  asm volatile ("outb %b0, %w1" : : "a" (data), "Nd" (port));
}

static inline void
insw (uint16_t port, void *addr, size_t cnt)
{
  asm volatile ("rep insw" : "+D" (addr), "+c" (cnt)
                : "d" (port) : "memory");
}
:
```


Example: parallel port (LPT1)

- **Simple hardware has three control registers:**

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

read/write data register (port 0x378)

BSY	ACK	PAP	OFON	ERR	–	–	–

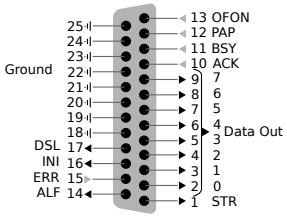
read-only status register (port 0x379)

–	–	–	IRQ	DSL	INI	ALF	STR

read/write control register (port 0x37a) [Messmer]

- **Every bit except IRQ corresponds to a pin on 25-pin connector:**

[image credits: Wikipedia]



Writing a byte to a parallel port [osdev]

```
void
sendbyte(uint8_t byte)
{
  /* Wait until BSY bit is 1. */
  while ((inb (0x379) & 0x80) == 0)
    delay ();

  /* Put the byte we wish to send on pins D7-0. */
  outb (0x378, byte);

  /* Pulse STR (strobe) line to inform the printer
   * that a byte is available */
  uint8_t ctrlval = inb (0x37a);
  outb (0x37a, ctrlval | 0x01);
  delay ();
  outb (0x37a, ctrlval);
}
```


IDE disk driver

```
void IDE_ReadSector(int disk, int off, void *buf)
{
  outb(0x1F6, disk == 0 ? 0xE0 : 0xF0); // Select Drive
  IDEWait();
  outb(0x1F2, 1);           // Read length (1 sector = 512 B)
  outb(0x1F3, off);         // LBA low
  outb(0x1F4, off >> 8);    // LBA mid
  outb(0x1F5, off >> 16);   // LBA high
  outb(0x1F7, 0x20);        // Read command
  insw(0x1F0, buf, 256);    // Read 256 words
}

void IDEWait()
{
  // Discard status 4 times
  inb(0x1F7); inb(0x1F7);
  inb(0x1F7); inb(0x1F7);
  // Wait for status BUSY flag to clear
  while ((inb(0x1F7) & 0x80) != 0)
    ;
}
```

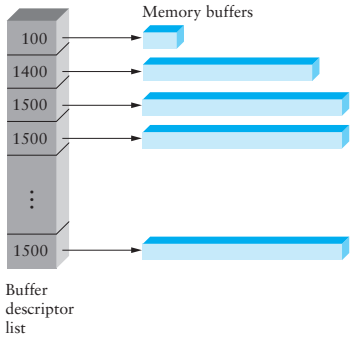

Memory-mapped IO

- `in`/`out` **instructions slow and clunky**
  - Instruction format restricts what registers you can use
  - Only allows 2^{16} different port numbers
  - Per-port access control turns out not to be useful (any port access allows you to disable all interrupts)
- **Devices can achieve same effect with physical addresses, e.g.:**

```
volatile int32_t *device_control
    = (int32_t *) (0xc0100 + PHYS_BASE);
*device_control = 0x80;
int32_t status = *device_control;
```

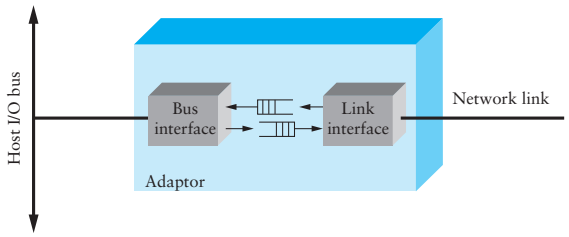
  - OS must map physical to virtual addresses, ensure non-cachable
- **Assign physical addresses at boot to avoid conflicts. PCI:**
  - Slow/clunky way to access configuration registers on device
  - Use that to assign ranges of physical addresses to device

DMA buffers



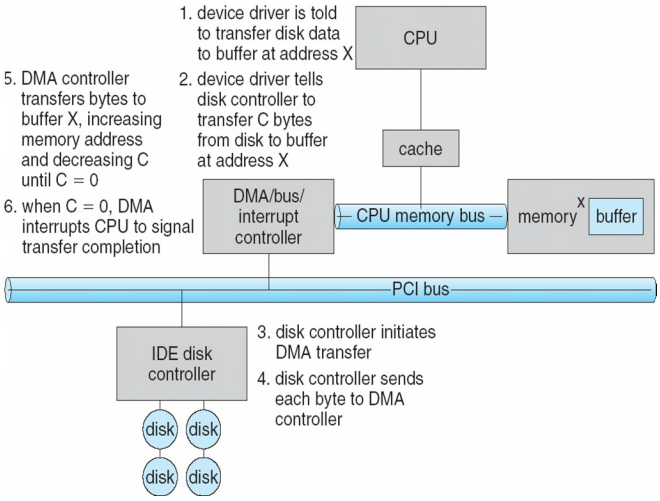
- **Idea: only use CPU to transfer control requests, not data**
- **Include list of buffer locations in main memory**
  - Device reads list and accesses buffers through DMA
  - Descriptions sometimes allow for scatter/gather I/O

Example: Network Interface Card



- **Link interface talks to wire/fiber/antenna**
  - Typically does framing, link-layer CRC
- **FIFOs on card provide small amount of buffering**
- **Bus interface logic uses DMA to move packets to and from buffers in main memory**

Example: IDE disk read w. DMA



Driver architecture

- **Device driver provides several entry points to kernel**
  - Reset, ioctl, output, interrupt, read, write, strategy …
- **How should driver synchronize with card?**
  - E.g., Need to know when transmit buffers free or packets arrive
  - Need to know when disk request complete
- **One approach: *Polling***
  - Sent a packet? Loop asking card when buffer is free
  - Waiting to receive? Keep asking card if it has packet
  - Disk I/O? Keep looping until disk ready bit set
- **Disadvantages of polling?**

Driver architecture

- **Device driver provides several entry points to kernel**
  - Reset, ioctl, output, interrupt, read, write, strategy …
- **How should driver synchronize with card?**
  - E.g., Need to know when transmit buffers free or packets arrive
  - Need to know when disk request complete
- **One approach: *Polling***
  - Sent a packet? Loop asking card when buffer is free
  - Waiting to receive? Keep asking card if it has packet
  - Disk I/O? Keep looping until disk ready bit set
- **Disadvantages of polling?**

  - Can't use CPU for anything else while polling
  - Schedule poll in future? High latency to receive packet or process disk block bad for response time

Interrupt driven devices

- **Instead, ask card to interrupt CPU on events**
  - Interrupt handler runs at high priority
  - Asks card what happened (xmit buffer free, new packet)
  - This is what most general-purpose OSes do
- **Bad under high network packet arrival rate**
  - Packets can arrive faster than OS can process them
  - Interrupts are expensive
  - Interrupt handlers have high priority
  - In worst case, can spend 100% of time in interrupt handler and never make any progress – *receive livelock*
  - Best: Adaptive switching between interrupts and polling
- **Very good for disk requests**
- **Rest of today: Disks (network devices in 3 lectures)**

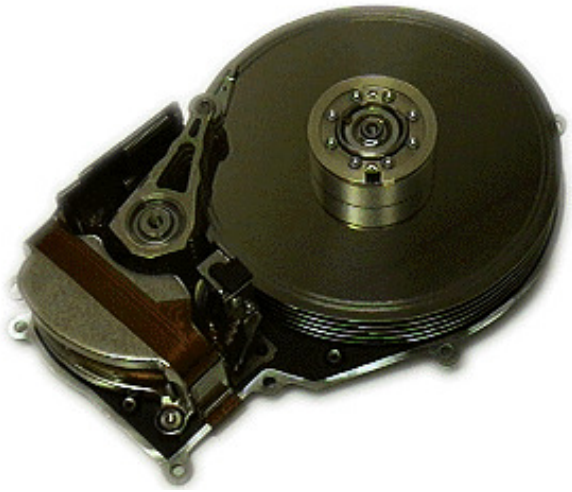
Outline

1. PC system architecture
2. Driver architecture
3. Disks
4. Disk scheduling
5. Flash

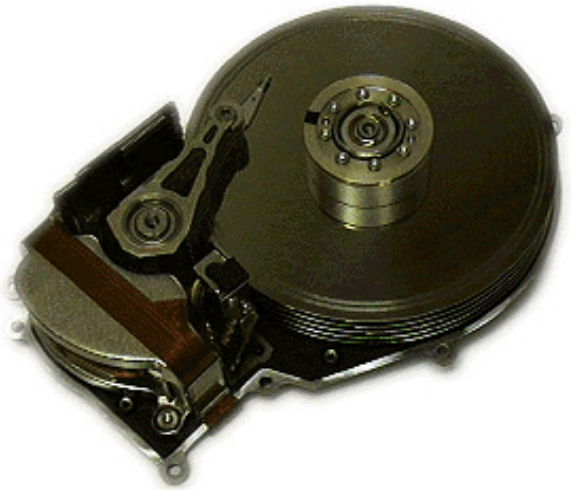
Anatomy of a disk [Ruemmler]

- **Stack of magnetic platters**
  - Rotate together on a central spindle @3,600-15,000 RPM
  - Drive speed drifts slowly over time
  - Can't predict rotational position after 100-200 revolutions
- **Disk arm assembly**
  - Arms rotate around pivot, all move together
  - Pivot offers some resistance to linear shocks
  - One disk head per recording surface (2×platters)
  - Sensitive to motion and vibration [Gregg] (demo on youtube)

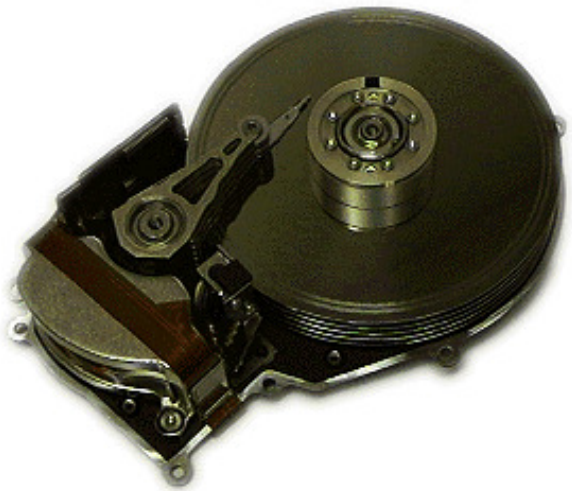
Disk



Disk



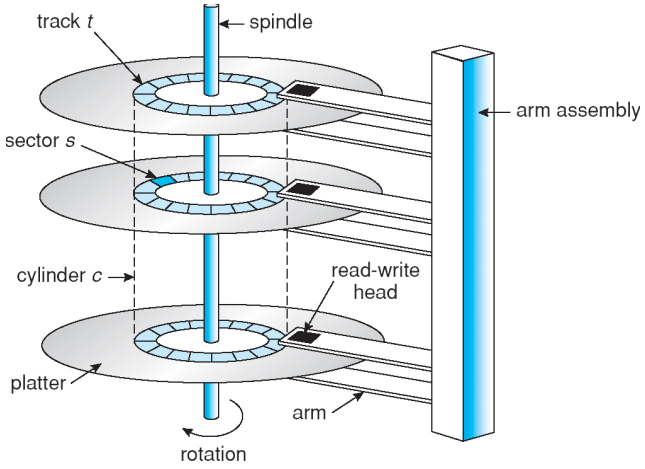
Disk



Storage on a magnetic platter

- **Platters divided into concentric *tracks***
- **A stack of tracks of fixed radius is a *cylinder***
- **Heads record and sense data along cylinders**
  - Significant fractions of encoded stream for error correction
- **Generally only one head active at a time**
  - Disks usually have one set of read-write circuitry
  - Must worry about cross-talk between channels
  - Hard to keep multiple heads exactly aligned

Cylinders, tracks, & sectors



Disk positioning system

- **Move head to specific track and keep it there**
  - Resist physical shocks, imperfect tracks, etc.
- **A *seek* consists of up to four phases:**
  - *speedup*–accelerate arm to max speed or half way point
  - *coast*–at max speed (for long seeks)
  - *slowdown*–stops arm near destination
  - *settle*–adjusts head to actual desired track
- **Very short seeks dominated by settle time (∼1 ms)**
- **Short (200-400 cyl.) seeks dominated by speedup**
  - Accelerations of 40g

Seek details

- **Head switches comparable to short seeks**
  - May also require head adjustment
  - Settles take longer for writes than for reads – Why?
- **Disk keeps table of pivot motor power**
  - Maps seek distance to power and time
  - Disk interpolates over entries in table
  - Table set by periodic "thermal recalibration"
  - But, e.g., ∼500 ms recalibration every ∼25 min bad for AV
- **"Average seek time" quoted can be many things**
  - Time to seek 1/3 disk, 1/3 time to seek whole disk

Seek details

- **Head switches comparable to short seeks**
  - May also require head adjustment
  - Settles take longer for writes than for reads
    If read strays from track, catch error with checksum, retry
    If write strays, you've just clobbered some other track
- **Disk keeps table of pivot motor power**
  - Maps seek distance to power and time
  - Disk interpolates over entries in table
  - Table set by periodic "thermal recalibration"
  - But, e.g., ∼500 ms recalibration every ∼25 min bad for AV
- **"Average seek time" quoted can be many things**
  - Time to seek 1/3 disk, 1/3 time to seek whole disk

Sectors

- **Disk interface presents linear array of *sectors***
  - Historically 512 B, but 4 KiB in "advanced format" disks
  - Written atomically (even if there is a power failure)
- **Disk maps logical sector #s to physical sectors**
  - *Zoning*–puts more sectors on longer tracks
  - *Track skewing*–sector 0 pos. varies by track (why?)
  - *Sparing*–flawed sectors remapped elsewhere
- **OS doesn't know logical to physical sector mapping**
  - Larger logical sector # difference means longer seek time
  - Highly non-linear relationship (*and* depends on zone)
  - OS has no info on rotational positions
  - Can empirically build table to estimate times

Sectors

- **Disk interface presents linear array of *sectors***
  - Historically 512 B, but 4 KiB in "advanced format" disks
  - Written atomically (even if there is a power failure)
- **Disk maps logical sector #s to physical sectors**
  - *Zoning*–puts more sectors on longer tracks
  - *Track skewing*–sector 0 pos. varies by track (sequential access speed
  - *Sparing*–flawed sectors remapped elsewhere
- **OS doesn't know logical to physical sector mapping**
  - Larger logical sector # difference means longer seek time
  - Highly non-linear relationship (*and* depends on zone)
  - OS has no info on rotational positions
  - Can empirically build table to estimate times

Disk interface

- **Controls hardware, mediates access**
- **Computer, disk often connected by bus (e.g., ATA, SCSI, SATA)**
  - Multiple devices may contentd for bus
- **Possible disk/interface features:**
- **Disconnect from bus during requests**
- **Command queuing: Give disk multiple requests**
  - Disk can schedule them using rotational information
- **Disk cache used for read-ahead**
  - Otherwise, sequential reads would incur whole revolution
  - Cross track boundaries? Can't stop a head-switch
- **Some disks support write caching**
  - But data not stable—not suitable for all requests

Disk performance

- **Placement & ordering of requests a huge issue**
  - Sequential I/O much, much faster than random
  - Long seeks much slower than short ones
  - Power might fail any time, leaving inconsistent state
- **Must be careful about order for crashes**
  - More on this in next two lectures
- **Try to achieve contiguous accesses where possible**
  - E.g., make big chunks of individual files contiguous
- **Try to order requests to minimize seek times**
  - OS can only do this if it has multiple requests to order
  - Requires disk I/O concurrency
  - High-performance apps try to maximize I/O concurrency
- **Next: How to schedule concurrent requests**

Outline

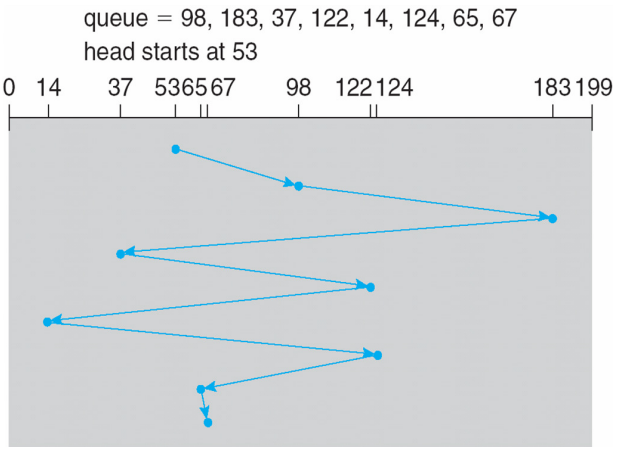
Scheduling: FCFS

- **"First Come First Served"**
  - Process disk requests in the order they are received
- **Advantages**
- **Disadvantages**

Scheduling: FCFS

- **"First Come First Served"**
  - Process disk requests in the order they are received
- **Advantages**
  - Easy to implement
  - Good fairness
- **Disadvantages**
  - Cannot exploit request locality
  - Increases average latency, decreasing throughput

FCFS example



Shortest positioning time first (SPTF)

- **Shortest positioning time first (SPTF)**
  - Always pick request with shortest seek time
- **Also called Shortest Seek Time First (SSTF)**
- **Advantages**
- **Disadvantages**

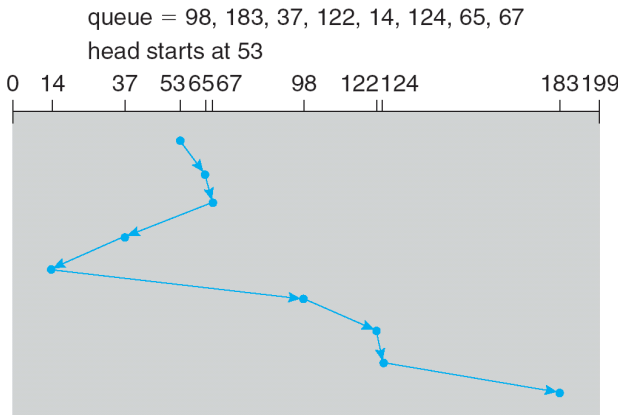
Shortest positioning time first (SPTF)

- **Shortest positioning time first (SPTF)**
  - Always pick request with shortest seek time
- **Also called Shortest Seek Time First (SSTF)**
- **Advantages**
  - Exploits locality of disk requests
  - Higher throughput
- **Disadvantages**
  - Starvation
  - Don't always know what request will be fastest
- **Improvement?**

Shortest positioning time first (SPTF)

- **Shortest positioning time first (SPTF)**
  - Always pick request with shortest seek time
- **Also called Shortest Seek Time First (SSTF)**
- **Advantages**
  - Exploits locality of disk requests
  - Higher throughput
- **Disadvantages**
  - Starvation
  - Don't always know what request will be fastest
- **Improvement: Aged SPTF**
  - Give older requests higher priority
  - Adjust "effective" seek time with weighting factor:
    $T_{\text{eff}} = T_{\text{pos}} - W \cdot T_{\text{wait}}$

SPTF example



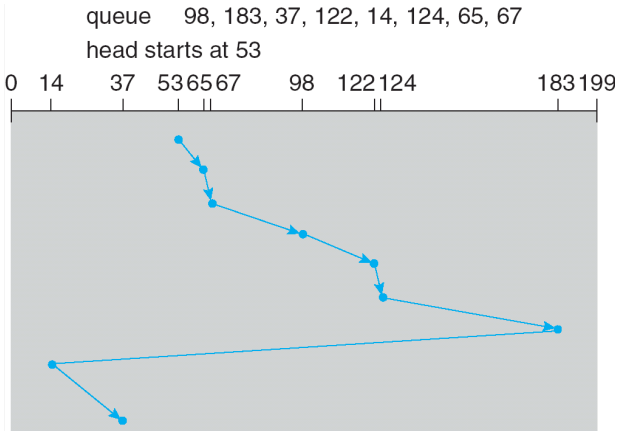
"Elevator" scheduling (SCAN)

- **Sweep across disk, servicing all requests passed**
  - Like SPTF, but next seek must be in same direction
  - Switch directions only if no further requests
- **Advantages**
- **Disadvantages**

"Elevator" scheduling (SCAN)

- **Sweep across disk, servicing all requests passed**
  - Like SPTF, but next seek must be in same direction
  - Switch directions only if no further requests
- **Advantages**
  - Takes advantage of locality
  - Bounded waiting
- **Disadvantages**
  - Cylinders in the middle get better service
  - Might miss locality SPTF could exploit
- **CSCAN: Only sweep in one direction**
  Very commonly used algorithm in Unix
- **Also called LOOK/CLOOK in textbook**
  - (Textbook uses [C]SCAN to mean scan entire disk uselessly)

CSCAN example



VSCAN(r)

- **Continuum between SPTF and SCAN**
  - Like SPTF, but slightly changes "effective" positioning time
    If request in same direction as previous seek: $T_{\text{eff}} = T_{\text{pos}}$
    Otherwise: $T_{\text{eff}} = T_{\text{pos}} + r \cdot T_{\text{max}}$
  - when r = 0, get SPTF, when r = 1, get SCAN
  - E.g., r = 0.2 works well
- **Advantages and disadvantages**
  - Those of SPTF and SCAN, depending on how r is set
- **See [Worthington] for good description and evaluation of various disk scheduling algorithms**

Outline

Flash memory

- **Today, people increasingly using flash memory**
- **Completely solid state (no moving parts)**
  - Remembers data by storing charge
  - Lower power consumption and heat
  - No mechanical seek times to worry about
- **Limited # overwrites possible**
  - Blocks wear out after 10,000 (MLC) – 100,000 (SLC) erases
  - Requires *flash translation layer* (FTL) to provide *wear leveling*, so repeated writes to logical block don't wear out physical block
  - FTL can seriously impact performance
  - In particular, random writes *very* expensive [Birrell]
- **Limited durability**
  - Charge wears out over time
  - Turn off device for a year, you can potentially lose data

Types of flash memory

- **NAND flash (most prevalent for storage)**
  - Higher density (most used for storage)
  - Faster erase and write
  - More errors internally, so need error correction
- **NOR flash**
  - Faster reads in smaller data units
  - Can execute code straight out of NOR flash
  - Significantly slower erases
- **Single-level cell (SLC) vs. Multi-level cell (MLC)**
  - MLC encodes multiple (two) bits in voltage level
  - MLC slower to write than SLC
  - MLC has lower durability (bits decay faster)
- **Nowadays, most flash drives are TLC, QLC, soon PLC**

NAND Flash Overview

- **Flash device has 2112-byte *pages***
  - 2048 bytes of data + 64 bytes metadata & ECC
- ***Blocks* contain 64 (SLC) or 128 (MLC) pages**
- **Blocks segregated into 2–4 *planes***
  - All planes contend for same package pins
  - But can access their blocks in parallel to overlap latencies
- **Can *read* one page at a time**
  - Takes 25 μsec + time to get data off chip
- **Must *erase* whole block before *programing***
  - Historically 256 KiB–4 MiB, now trending higher towards 1 GiB
  - Erase sets all bits to 1—very expensive (2 msec)
  - Programming pre-erased block requires moving data to internal buffer, then 200 (SLC)–800 (MLC) μsec
  - Note SMR magnetic drives starting to behave like this, too!

Flash Characteristics [Caulfield'09]

	Parameter	SLC	MLC
	Density Per Die (GB)	4	8
	Page Size (Bytes)	2048+32	2048+64
	Block Size (Pages)	64	128
	Read Latency (μs)	25	25
	Write Latency (μs)	200	800
	Erase Latency (μs)	2000	2000
40MHz, 16-bit bus	Read b/w (MB/s)	75.8	75.8
	Program b/w (MB/s)	20.1	5.0
133MHz	Read b/w (MB/s)	126.4	126.4
	Program b/w (MB/s)	20.1	5.0

FTL straw man: in-memory map

- **Keep in-memory map of logical → physical page #**
  - On write, pick unused page, mark previous physical page free
  - Repeated writes of a logical page will hit different physical pages
- **Store map in device memory, but must rebuild on power-up**
- **Idea: Put header on each page, scan all headers on power-up:**
  ⟨**logical page #**, $\overline{\textit{Allocated}}$ **bit**, $\overline{\textit{Written}}$ **bit**, $\overline{\textit{Obsolete}}$ **bit**⟩
  - A-W-O = 1-1-1: free page
  - A-W-O = 0-1-1: about to write page
  - A-W-O = 0-0-1: successfully written page
  - A-W-O = 0-0-0: obsolete page (can erase block without copying)
- **Why the 0-1-1 state?**
- **What's wrong still?**

FTL straw man: in-memory map

- **Keep in-memory map of logical → physical page #**
  - On write, pick unused page, mark previous physical page free
  - Repeated writes of a logical page will hit different physical pages
- **Store map in device memory, but must rebuild on power-up**
- **Idea: Put header on each page, scan all headers on power-up:**
  ⟨**logical page #**, $\overline{\textit{Allocated}}$ **bit**, $\overline{\textit{Written}}$ **bit**, $\overline{\textit{Obsolete}}$ **bit**⟩
  - A-W-O = 1-1-1: free page
  - A-W-O = 0-1-1: about to write page
  - A-W-O = 0-0-1: successfully written page
  - A-W-O = 0-0-0: obsolete page (can erase block without copying)
- **Why the 0-1-1 state?** After power failure partly written ≠ free
- **What's wrong still?**

FTL straw man: in-memory map

- **Keep in-memory map of logical → physical page #**
  - On write, pick unused page, mark previous physical page free
  - Repeated writes of a logical page will hit different physical pages
- **Store map in device memory, but must rebuild on power-up**
- **Idea: Put header on each page, scan all headers on power-up:**
  ⟨**logical page #**, $\overline{\textit{Allocated}}$ **bit**, $\overline{\textit{Written}}$ **bit**, $\overline{\textit{Obsolete}}$ **bit**⟩
  - A-W-O = 1-1-1: free page
  - A-W-O = 0-1-1: about to write page
  - A-W-O = 0-0-1: successfully written page
  - A-W-O = 0-0-0: obsolete page (can erase block without copying)
- **Why the 0-1-1 state?** After power failure partly written ≠ free
- **What's wrong still?**
  - FTL requires a lot of RAM on device, plus time to scan all headers
  - Some blocks still get erased more than others (w. long-lived data)
  - Blocks with obsolete pages may also contain live pages

More realistic FTL

- **Store the FTL map in the flash device itself**
  - Add one header bit to distinguish map page from data page
  - Logical read may miss map cache, require 2 flash reads
  - Keep smaller "map-map" in memory, cache some map pages
- **Must garbage-collect blocks with obsolete pages**
  - Copy live pages to a new block, erase old block
  - Always need free blocks, can't use 100% physical storage
- **Problem: write amplification**
  - Small random writes punch holes in many blocks
  - If small writes require garbage-collecting a 90%-full blocks
    …means you are writing 10× more physical than logical data!
- **Must also periodically re-write even blocks w/o holes**
  - *Wear leveling* ensures active blocks don't wear out first

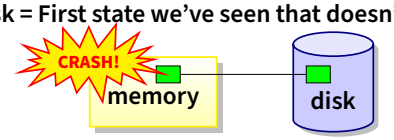
12. File systems

File system fun

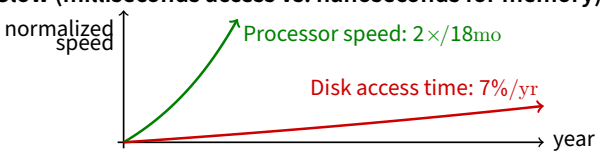
- **File systems: traditionally hardest part of OS**
  - Historically, more papers on FSes than any other single topic
- **Main tasks of file system:**
  - Associate bytes with name (files)
  - Associate names with each other (directories)
  - Don't go away (ever)
  - Can implement file systems on disk, over network, in memory, in non-volatile ram (NVRAM), on tape, w/ paper.
  - We'll focus on disk and generalize later
- **Today: files, directories, and a bit of performance**

Why disks are different

- **Disk = First state we've seen that doesn't go away**



  - So: Where all important state ultimately resides
- **Slow (milliseconds access vs. nanoseconds for memory)**



- **Huge (64–1,000x bigger than memory)**
  - How to organize large collection of ad hoc information?
  - File System: Hierarchical directories, Metadata, Search

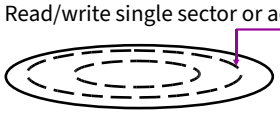
Disk vs. Memory

	Disk	TLC NAND Flash	DRAM
Smallest write	sector	sector	byte
Atomic write	sector	sector	byte/word
Random read	8 ms	3-10 μs	50 ns
Random write	8 ms	9-11 μs*	50 ns
Sequential read	200 MB/s	550–2500 MB/s	$>$ 10 GB/s
Sequential write	200 MB/s	520–1500 MB/s*	$>$ 10 GB/s
Cost	$0.01–0.02/GB	$0.06–0.10/GB	$1.60–2.50/GiB
Persistence	Non-volatile	Non-volatile	Volatile

*Flash write performance degrades over time

Disk review

- **Disk reads/writes in terms of sectors, not bytes**
  - Read/write single sector or adjacent groups



- **How to write a single byte? "Read-modify-write"**
  - Read in sector containing the byte
  - Modify that byte
  - Write entire sector back to disk
  - Key: if cached, don't need to read in

- **Sector = unit of atomicity.**
  - Sector write done completely, even if crash in middle (disk saves up enough momentum to complete)

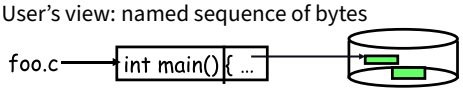
- **Larger atomic units have to be synthesized by OS**

Some useful trends

- **Disk bandwidth and cost/bit improving exponentially**
  - Similar to CPU speed, memory size, etc.
- **Seek time and rotational delay improving *very* slowly**
  - Why? require moving physical object (disk arm)
- **Disk accesses a huge system bottleneck & getting worse**
  - Bandwidth increase lets system (pre-)fetch large chunks for about the same cost as small chunk.
  - Trade bandwidth for latency if you can get lots of related stuff.
- **Desktop memory size increasing faster than typical workloads**
  - More and more of workload fits in file cache
  - Disk traffic changes: mostly writes and new data
- **Memory and CPU resources increasing**
  - Use memory and CPU to make better decisions
  - Complex prefetching to support more IO patterns
  - Delay data placement decisions to reduce random IO

Files: named bytes on disk

- **File abstraction:**
  - User's view: named sequence of bytes



  - FS's view: collection of disk blocks
  - File system's job: translate name & offset to disk blocks:

{file, offset} ⟶ FS ⟶ disk address

- **File operations:**
  - Create a file, delete a file
  - Read from file, write to file
- **Want: operations to have as few disk accesses as possible & have minimal space overhead (group related things)**

What's hard about grouping blocks?

- **Like page tables, file system metadata are simply data structures used to construct mappings**
  - Page table: map virtual page # to physical page #
    23 → Page table → 33
  - File metadata: map byte offset to disk block address
    512 → Unix inode → 8003121
  - Directory: map name to disk address or file #
    foo.c → directory → 44

FS vs. VM

- **In both settings, want location transparency**
  - Application shouldn't care about particular disk blocks or physical memory locations
- **In some ways, FS has easier job than than VM:**
  - CPU time to do FS mappings not a big deal (= no TLB)
  - Page tables deal with sparse address spaces and random access, files often denser $(0 \ldots \text{filesize} - 1)$, ∼sequentially accessed
- **In some ways FS's problem is harder:**
  - Each layer of translation = potential disk access
  - Space a huge premium! (But disk is huge?!?!) Reason? Cache space never enough; amount of data you can get in one fetch never enough
  - Range very extreme: Many files <10 KiB, some files many GiB

Some working intuitions

- **FS performance dominated by # of disk accesses**
  - Say each access costs ∼10 milliseconds
  - Touch the disk 100 extra times = 1 *second*
  - Can do *billions* of ALU ops in same time!
- **Access cost dominated by movement, not transfer:**
  seek time + **rotational delay** + # bytes/disk-bw
  - 1 sector: 5ms + 4ms + 5μs ($\approx 512\ \text{B}/(100\ \text{MB/s})$) ≈ 9ms
  - 50 sectors: 5ms + 4ms + .25ms = 9.25ms
  - Can get 50x the data for only ∼3% more overhead!
- **Observations that might be helpful:**
  - All blocks in file tend to be used together, sequentially
  - All files in a directory tend to be used together
  - All names in a directory tend to be used together

Common addressing patterns

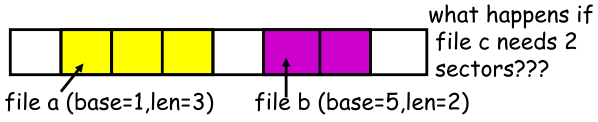
- **Sequential:**
  - File data processed in sequential order
  - By far the most common mode
  - Example: editor writes out new file, compiler reads in file, etc.
- **Random access:**
  - Address any block in file directly without passing through predecessors
  - Examples: data set for demand paging, databases
- **Keyed access**
  - Search for block with particular values
  - Examples: associative data base, index
  - Usually not provided by OS

Problem: how to track file's data

- **Disk management:**
  - Need to keep track of where file contents are on disk
  - Must be able to use this to map byte offset to disk block
  - Structure tracking a file's sectors is called an index node or *inode*
  - Inodes must be stored on disk, too
- **Things to keep in mind while designing file structure:**
  - Most files are small
  - Much of the disk is allocated to large files
  - Many of the I/O operations are made to large files
  - Want good sequential and good random access (what do these require?)

Straw man: contiguous allocation

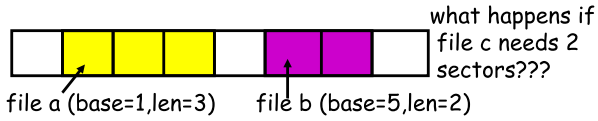
- **"Extent-based": allocate files like segmented memory**
  - When creating a file, make the user pre-specify its length and allocate all space at once
  - Inode contents: location and size



- **Example: IBM OS/360**
- **Pros?**
- **Cons? (Think of corresponding VM scheme)**

Straw man: contiguous allocation

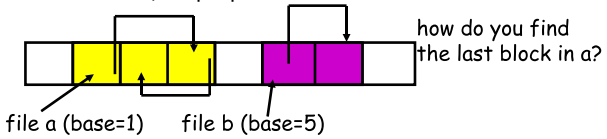
- **"Extent-based": allocate files like segmented memory**
  - When creating a file, make the user pre-specify its length and allocate all space at once
  - Inode contents: location and size



- **Example: IBM OS/360**
- **Pros?**
  - Simple, fast access, both sequential and random
- **Cons? (Think of corresponding VM scheme)**
  - External fragmentation

Straw man #2: Linked files

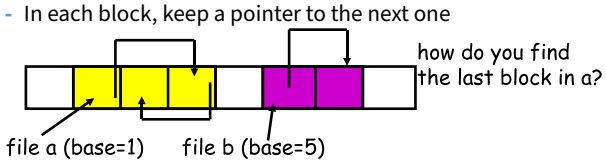
- **Basically a linked list on disk.**
  - Keep a linked list of all free blocks
  - Inode contents: a pointer to file's first block
  - In each block, keep a pointer to the next one



- **Examples (sort-of): Alto, TOPS-10, DOS FAT**
- **Pros?**
- **Cons?**

Straw man #2: Linked files

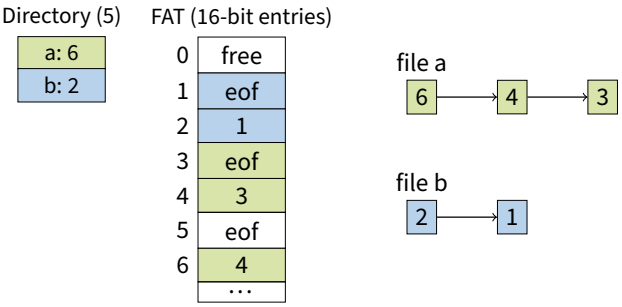
- **Basically a linked list on disk.**
  - Keep a linked list of all free blocks
  - Inode contents: a pointer to file's first block
  - In each block, keep a pointer to the next one



- **Examples (sort-of): Alto, TOPS-10, DOS FAT**
- **Pros?**
  - Easy dynamic growth & sequential access, no fragmentation
- **Cons?**
  - Linked lists on disk a bad idea because of access times
  - Random very slow (e.g., traverse whole file to find last block)
  - Pointers take up room in block, skewing alignment

Example: DOS FS (simplified)

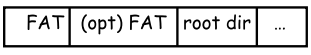
- **Linked files with key optimization: puts links in fixed-size "file allocation table" (FAT) rather than in the blocks.**



- **Still do pointer chasing, but can cache entire FAT so can be cheap compared to disk access**

FAT discussion

- **Entry size = 16 bits**
  - What's the maximum size of the FAT?
  - Given a 512 byte block, what's the maximum size of FS?
  - One solution: go to bigger blocks. Pros? Cons?
- **Space overhead of FAT is trivial:**
  - 2 bytes / 512 byte block = ~ 0.4% (Compare to Unix)
- **Reliability: how to protect against errors?**
  - Create duplicate copies of FAT on disk
  - State duplication a very common theme in reliability
- **Bootstrapping: where is root directory?**
  - Fixed location on disk:

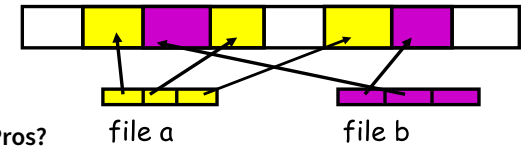


FAT discussion

- **Entry size = 16 bits**
  - What's the maximum size of the FAT? 65,536 entries
  - Given a 512 byte block, what's the maximum size of FS? 32 MiB
  - One solution: go to bigger blocks. Pros? Cons?
- **Space overhead of FAT is trivial:**
  - 2 bytes / 512 byte block = ~ 0.4% (Compare to Unix)
- **Reliability: how to protect against errors?**
  - Create duplicate copies of FAT on disk
  - State duplication a very common theme in reliability
- **Bootstrapping: where is root directory?**
  - Fixed location on disk:

Another approach: Indexed files

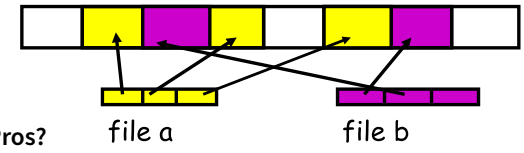
- **Each file has an array holding all of its block pointers**
  - Just like a page table, so will have similar issues
  - Max file size fixed by array's size (static or dynamic?)
  - Allocate array to hold file's block pointers on file creation
  - Allocate actual blocks on demand using free list



- **Pros?**
- **Cons?**

Another approach: Indexed files

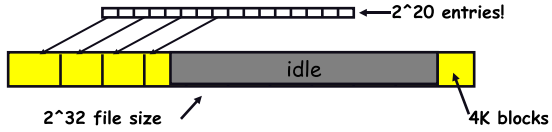
- **Each file has an array holding all of its block pointers**
  - Just like a page table, so will have similar issues
  - Max file size fixed by array's size (static or dynamic?)
  - Allocate array to hold file's block pointers on file creation
  - Allocate actual blocks on demand using free list



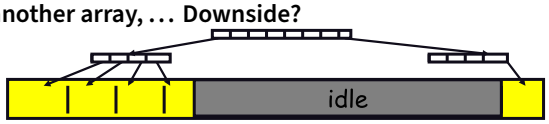
- **Pros?**
  - Both sequential and random access easy
- **Cons?**
  - Mapping table requires large chunk of contiguous space
    …Same problem we were trying to solve initially

Indexed files

- **Issues same as in page tables**

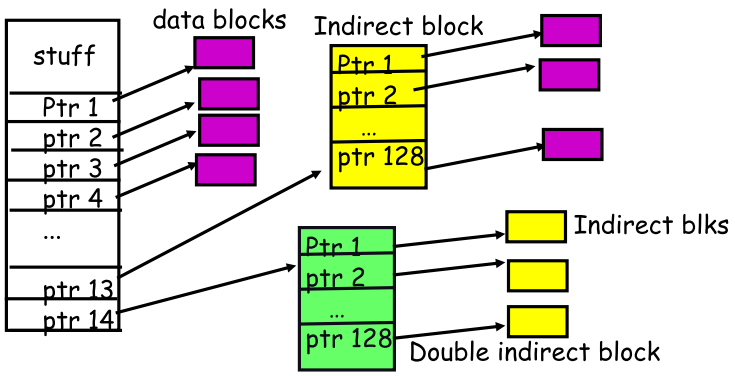


  - Large possible file size = lots of unused entries
  - Large actual size? table needs large contiguous disk chunk
- **Solve identically: small regions with index array, this array with another array, … Downside?**



Multi-level indexed files (old BSD FS)

- **Solve problem of first block access slow**
- **inode = 14 block pointers + "stuff"**



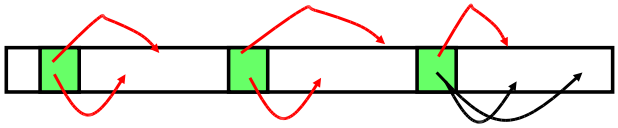
Old BSD FS discussion

- **Pros:**
  - Simple, easy to build, fast access to small files
  - Maximum file length fixed, but large.
- **Cons:**
  - What is the worst case # of accesses?
  - What is the worst-case space overhead? (e.g., 13 block file)
- **An empirical problem:**
  - Because you allocate blocks by taking them off unordered freelist, metadata and data get strewn across disk

More about inodes

- **Inodes are stored in a fixed-size array**
  - Size of array fixed when disk is initialized; can't be changed
  - Lives in known location, originally at one side of disk:

  - Now is smeared across it (why?)



  - The index of an inode in the inode array called an i-number
  - Internally, the OS refers to files by inumber
  - When file is opened, inode brought in memory
  - Written back when modified and file closed or time elapses

Directories

- **Problem:**
  - "Spend all day generating data, come back the next morning, want to use it." – F. Corbató, on why files/dirs invented
- **Approach 0: Users remember where on disk their files are**
  - E.g., like remembering your social security or bank account #
- **Yuck. People want human digestible names**
  - We use directories to map names to file blocks
- **Next: What is in a directory and why?**

A short history of directories

- **Approach 1: Single directory for entire system**
  - Put directory at known location on disk
  - Directory contains $\langle \text{name}, \text{inumber} \rangle$ pairs
  - If one user uses a name, no one else can
  - Many ancient personal computers work this way
- **Approach 2: Single directory for each user**
  - Still clumsy, and `ls` on 10,000 files is a real pain
- **Approach 3: Hierarchical name spaces**
  - Allow directory to map names to files *or other dirs*
  - File system forms a tree (or graph, if links allowed)
  - Large name spaces tend to be hierarchical (ip addresses, domain names, scoping in programming languages, etc.)

Hierarchical Unix

- **Used since CTSS (1960s)**
  - Unix picked up and used really nicely
- **Directories stored on disk just like regular files**
  - Special inode type byte set to directory
  - Users can read just like any other file (historically)
  - Only special syscalls can write (why?)
  - Inodes at fixed disk location
  - File pointed to by the index may be another directory
  - Makes FS into hierarchical tree (what needed to make a DAG?)

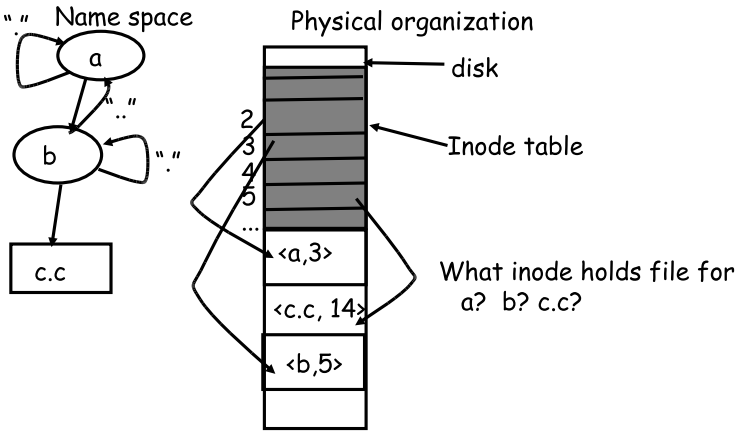
```
<name,inode#>
  <afs,1021>
  <tmp,1020>
  <bin,1022>
<cdrom,4123>
  <dev,1001>
 <sbin,1011>
      ⋮
```

- **Simple, plus speeding up file ops speeds up dir ops!**

Naming magic

- **Bootstrapping: Where do you start looking?**
  - Root directory always inode #2 (0 and 1 historically reserved)
- **Special names:**
  - Root directory: "`/`" (fixed by kernel—e.g., inode 2)
  - Current directory: "`.`" (actual directory entry on disk)
  - Parent directory: "`..`" (actual directory entry on disk)
- **Some special names are provided by shell, not FS:**
  - User's home directory: "~"
  - Globbing: "`foo.*`" expands to all files starting "`foo.`"
- **Using the given names, only need two operations to navigate the entire name space:**
  - `cd` *name*: move into (change context to) directory *name*
  - `ls`: enumerate all names in current directory (context)

Unix example: `/a/b/c.c`

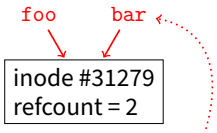


Default context: working directory

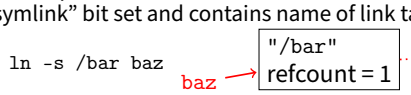
- **Cumbersome to constantly specify full path names**
  - In Unix, each process has a "current working directory" (cwd)
  - File names not beginning with "/" are assumed to be relative to cwd; otherwise translation happens as before
  - Editorial: root, cwd should be regular fds (like stdin, stdout, …) with *openat* syscall instead of *open*
    - In modern linux *open* in libc calls *openat(FD_FDCWD, …)* syscall
- **Shells track a default list of active contexts**
  - A "search path" for programs you run
  - Given a search path $A : B : C$, a shell will check in A, then check in B, then check in C
  - Can escape using explicit paths: "./foo"
- **Example of locality**

Hard and soft links (synonyms)

- **More than one dir entry can refer to a given file**
  - Unix stores count of pointers ("hard links") to inode
  - To make: "`ln foo bar`" creates a synonym (`bar`) for *file* `foo`



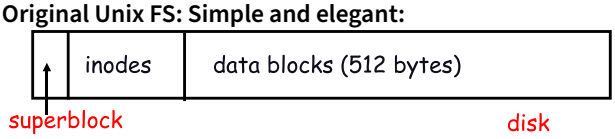
- **Soft/symbolic links = synonyms for *names***
  - Point to a file (or dir) *name*, but object can be deleted from underneath it (or never even exist).
  - Unix implements like directories: inode has special "symlink" bit set and contains name of link target



  - When the file system encounters a symbolic link it automatically translates it (if possible).

Case study: speeding up FS

- **Original Unix FS: Simple and elegant:**



- **Components:**
  - Data blocks
  - Inodes (directories represented as files)
  - Hard links
  - Superblock. (specifies number of blks in FS, counts of max # of files, pointer to head of free list)
- **Problem: slow**
  - Only gets 20Kb/sec (2% of disk maximum) even for sequential disk transfers!

A plethora of performance costs

- **Blocks too small (512 bytes)**
  - File index too large
  - Too many layers of mapping indirection
  - Transfer rate low (get one block at time)
- **Poor clustering of related objects:**
  - Consecutive file blocks not close together
  - Inodes far from data blocks
  - Inodes for files in same directory not close together
  - Poor enumeration performance: e.g., "`ls -l`", "`grep foo *.c`"
- **Usability problems**
  - 14-character file names a pain
  - Can't atomically update file in crash-proof way
- **Next: how FFS fixes these (to a degree)** [McKusic]

Problem: Internal fragmentation

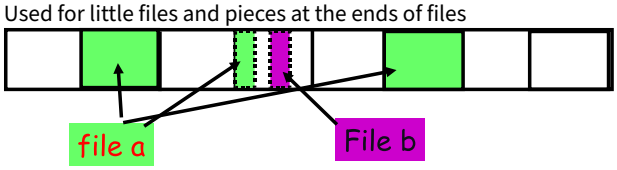
- **Block size was too small in Unix FS**
- **Why not just make block size bigger?**

Block size	space wasted	file bandwidth
512	6.9%	2.6%
1024	11.8%	3.3%
2048	22.4%	6.4%
4096	45.6%	12.0%
1MB	99.0%	97.2%

- **Bigger block increases bandwidth, but how to deal with wastage ("internal fragmentation")?**
  - Use idea from malloc: split unused portion.

Solution: fragments

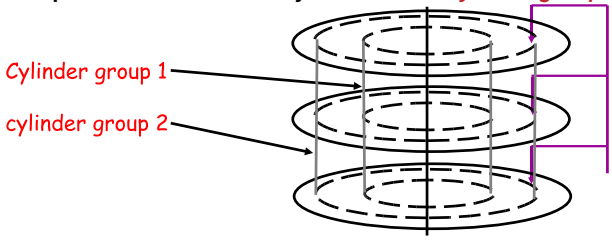
- **BSD FFS:**
  - Has large block size (4096 or 8192)
  - Allow large blocks to be chopped into small ones ("fragments")
  - Used for little files and pieces at the ends of files



- **Best way to eliminate internal fragmentation?**
  - Variable sized splits of course
  - Why does FFS use fixed-sized fragments (1024, 2048)?

Clustering related objects in FFS

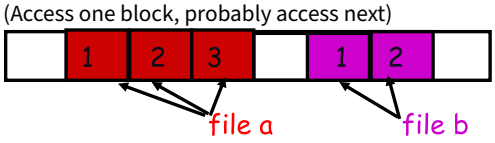
- **Group sets of consecutive cylinders into "*cylinder groups*"**



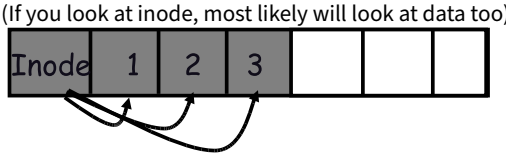
  - Key: can access any block in a cylinder without performing a seek. Next fastest place is adjacent cylinder.
  - Tries to put everything related in same cylinder group
  - Tries to put everything not related in different group

Clustering in FFS

- **Tries to put sequential blocks in adjacent sectors**
  - (Access one block, probably access next)



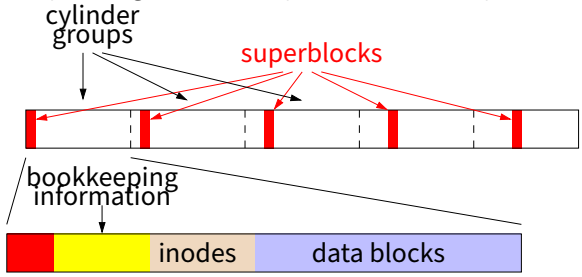
- **Tries to keep inode in same cylinder group as file data:**
  - (If you look at inode, most likely will look at data too)



- **Tries to keep all inodes in a dir in same cylinder group**
  - Access one name, frequently access many, e.g., "`ls -l`"

What does disk layout look like?

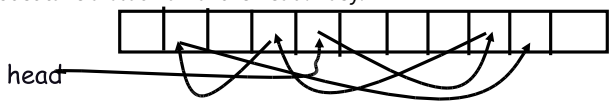
- **Each cylinder group basically a mini-Unix file system:**



- **How how to ensure there's space for related stuff?**
  - Place different directories in different cylinder groups
  - Keep a "free space reserve" so can allocate near existing things
  - When file grows too big (1MB) send its remainder to different cylinder group.

Finding space for related objs

- **Old Unix (& DOS): Linked list of free blocks**
  - Just take a block off of the head. Easy.



  - Bad: free list gets jumbled over time. Finding adjacent blocks hard and slow
- **FFS: switch to bit-map of free blocks**
  - 10101011111110000011111111000101100
  - Easier to find contiguous blocks.
  - Small, so usually keep entire thing in memory
  - Time to find free block increases if fewer free blocks

Using a bitmap

- **Usually keep entire bitmap in memory:**
  - 4G disk / 4K byte blocks. How big is map?
- **Allocate block close to block *x*?**
  - Check for blocks near `bmap[`*x*`/32]`
  - If disk almost empty, will likely find one near
  - As disk becomes full, search becomes more expensive and less effective
- **Trade space for time (search time, file access time)**
- **Keep a reserve (e.g, 10%) of disk always free, ideally scattered across disk**
  - Don't tell users (`df` can get to 110% full)
  - Only root can allocate blocks once FS 100% full
  - With 10% free, can almost always find one of them free

So what did we gain?

- **Performance improvements:**
  - Able to get 20-40% of disk bandwidth for large files
  - 10-20x original Unix file system!
  - Better small file performance (why?)
- **Is this the best we can do? No.**
- **Block based rather than extent based**
  - Could have named contiguous blocks with single pointer and length (Linux ext4fs, XFS)
- **Writes of metadata done synchronously**
  - Really hurts small file performance
  - Make asynchronous with write-ordering ("soft updates") or logging/journaling… more next lecture
  - Play with semantics (/tmp file systems)

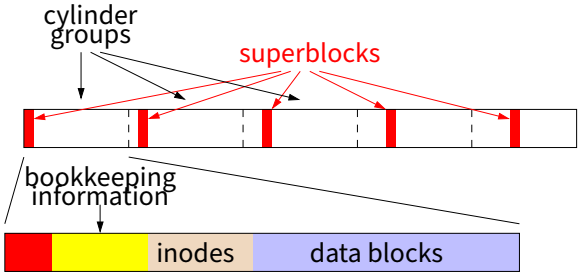
Other hacks

- **Obvious:**
  - Big file cache
- **Fact: no rotation delay if get whole track.**
  - How to use?
- **Fact: transfer cost negligible.**
  - Recall: Can get 50x the data for only ~3% more overhead
  - 1 sector: 5ms + 4ms + 5μs ($\approx 512\ \mathrm{B}/(100\ \mathrm{MB/s})$) $\approx$ 9ms
  - 50 sectors: 5ms + 4ms + .25ms = 9.25ms
  - How to use?
- **Fact: if transfer huge, seek + rotation negligible**
  - LFS: Hoard data, write out MB at a time
- **Next lecture:**
  - FFS in more detail
  - More advanced, modern file systems

13. Advanced file systems

Outline

Review: FFS disk layout



- **Each cylinder group has its own:**
  - Superblock
  - Bookkeeping information
  - Set of inodes
  - Data/directory blocks

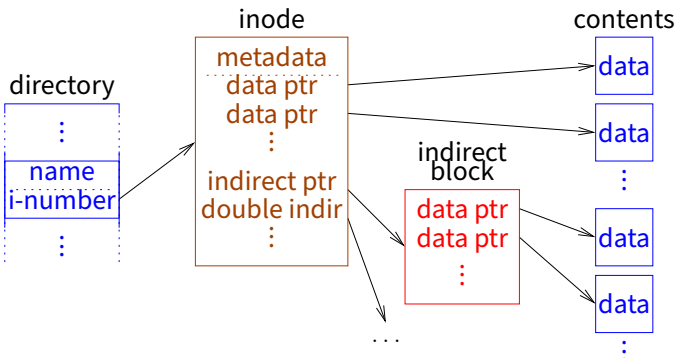
Superblock

- **Contains file system parameters**
  - Disk characteristics, block size, CG info
  - Information necessary to locate inode given i-number
- **Replicated once per cylinder group**
  - At shifting offsets, so as to span multiple platters
  - Contains magic number `0x011954` to find replicas if 1st superblock dies (Kirk McKusick's birthday?)
- **Contains non-replicated "summary information"**
  - # blocks, fragments, inodes, directories in FS
  - Flag stating if FS was cleanly unmounted

Bookkeeping information

- **Block map**
  - Bit map of available fragments
  - Used for allocating new blocks/fragments
- **Summary info within CG**
  - # free inodes, blocks/frags, files, directories
  - Used when picking cylinder group from which to allocate
- **# free blocks by rotational position (8 positions)**
  - Was reasonable in 1980s when disks weren't commonly zoned
  - Back then OS could do stuff to minimize rotational delay

Inodes and data blocks



- **Each CG has fixed # of inodes (default one per 2K data)**
  - Each inode maps offset → disk block for one file
  - Also contains metadata: permissions, mod times, link count, ...

Allocation

- **Place inode of new file in same CG as directory**
  - New directories go in new CG (with above average # free inodes)
- **Allocate blocks to optimize for sequential access**
  - If available, use rotationally close block in same cylinder (obsolete)
  - Otherwise, use block in same CG
  - If CG totally full, find other CG with quadratic hashing i.e., if CG #n is full, try $n + 1^2, n + 2^2, n + 3^2, \ldots \pmod{\#CGs}$
  - Otherwise, search all CGs for some free space
  - Break big files over multiple CGs
- **Fragment allocation could require moving last block a lot**
  - (Partial) soution: new `stat` struct field `st_blksize`
  - stdio library buffers this much data before writing

Directories

- **Directories have normal inodes with different type bits**
- **Contents considered as 512-byte *chunks***
- **Each chunk has `direct` structure(s) with:**
  - 32-bit inumber
  - 16-bit size of directory entry
  - 8-bit file type (added later)
  - 8-bit length of file name
- **Coalesce when deleting**
  - If first `direct` in chunk deleted, set inumber = 0
- **Periodically compact directory chunks**
  - But can never move directory entries across chunks
  - Recall only 512-byte sector writes atomic w. power failure

Outline

Fixing corruption – fsck

- **Must run FS check (fsck) program after crash**
- **Summary info usually bad after crash**
  - Scan to check free block map, block/inode counts
- **System may have corrupt inodes (not simple crash)**
  - Bad block numbers, cross-allocation, etc.
  - Do sanity check, clear inodes containing garbage
- **Fields in inodes may be wrong**
  - Count number of directory entries to verify link count, if no entries but count $\neq 0$, move to `lost+found`
  - Make sure size and used data counts match blocks
- **Directories may be bad**
  - Holes illegal, `.` and `..` must be valid, file names must be unique
  - All directories must be reachable

Crash recovery permeates FS code

- **Have to ensure fsck can recover file system**
- **Strawman: just write all data asynchronously**
  - Any subset of data structures may be updated before a crash
- **Delete/truncate a file, append to other file, crash?**

Crash recovery permeates FS code

- **Have to ensure fsck can recover file system**
- **Strawman: just write all data asynchronously**
  - Any subset of data structures may be updated before a crash
- **Delete/truncate a file, append to other file, crash?**
  - New file may reuse block from old
  - Old inode may not be updated
  - Cross-allocation!
  - Often inode with older mtime wrong, but can't be sure
- **Append to file, allocate indirect block, crash?**

Crash recovery permeates FS code

- **Have to ensure fsck can recover file system**
- **Strawman: just write all data asynchronously**
  - Any subset of data structures may be updated before a crash
- **Delete/truncate a file, append to other file, crash?**
  - New file may reuse block from old
  - Old inode may not be updated
  - Cross-allocation!
  - Often inode with older mtime wrong, but can't be sure
- **Append to file, allocate indirect block, crash?**
  - Inode points to indirect block
  - But indirect block may contain garbage!

Sidenote: kernel-internal disk write routines

- **BSD has three ways of writing a block to disk**

1. `bdwrite` – **delayed write**
   - Marks cached copy of block as dirty, does not write it
   - Will get written back in background within 30 seconds
   - Used if block likely to be modified again soon
2. `bawrite` – **asynchronous write**
   - Start write but return immediately before it completes
   - E.g., use when appending to file and block is full
3. `bwrite` – **synchronous write**
   - Start write, sleep and do not return until safely on disk

Ordering of updates

- **Must be careful about order of updates**
  - Write new inode to disk before directory entry
  - Remove directory name before deallocating inode
  - Write cleared inode to disk before updating CG free map
- **Solution: Many metadata updates synchronous** (`bwrite`)
  - Doing one write at a time ensures ordering
  - Of course, this hurts performance
  - E.g., untar much slower than disk bandwidth
- **Note: Cannot update buffers on the disk queue**
  - E.g., say you make two updates to same directory block
  - But crash recovery requires first to be synchronous
  - Must wait for first write to complete before doing second
  - Makes `bawrite` as slow as `bwrite` for many updates to same block

Performance vs. consistency

- **FFS crash recoverability comes at *huge* cost**
  - Makes tasks such as untar easily 10–20 times slower
  - All because you *might* lose power or reboot at any time
- **Even slowing normal case does not make recovery fast**
  - If fsck takes one minute, then disks get 10× bigger, then 100× …
- **One solution: battery-backed RAM**
  - Expensive (requires specialized hardware)
  - Often don't learn battery has died until too late
  - A pain if computer dies (can't just move disk)
  - If OS bug causes crash, RAM might be garbage
- **Better solution: Advanced file system techniques**
  - Next: two advanced techniques

Outline

First attempt: Ordered updates

- **Want to avoid crashing after "bad" subset of writes**
- **Must follow 3 rules in ordering updates [Ganger]:**
  1. Never write pointer before initializing the structure it points to
  2. Never reuse a resource before nullifying all pointers to it
  3. Never clear last pointer to live resource before setting new one
- **If you do this, file system will be recoverable**
- **Moreover, can recover quickly**
  - Might leak free disk space, but otherwise correct
  - So start running after reboot, scavenge for space in background
- **How to achieve?**
  - Keep a partial order on buffered blocks

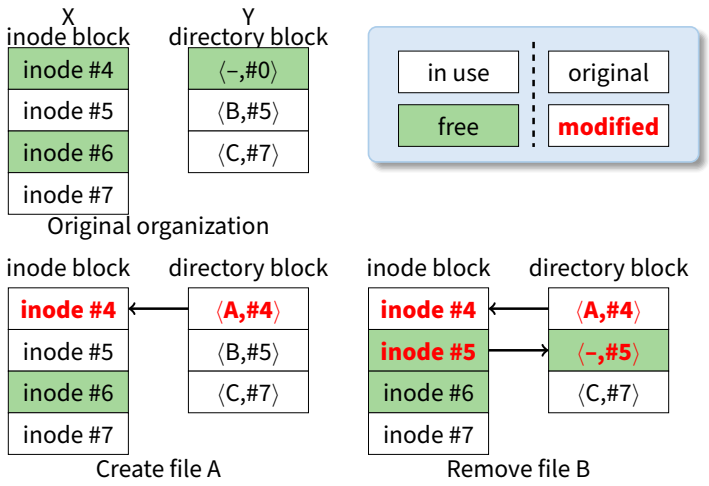
Ordered updates (continued)

- **Example: Create file** *A*
  - Block *X* contains an inode
  - Block *Y* contains a directory block
  - Create file *A* in inode block *X*, dir block *Y*
  - By rule #1, must write *X* before writing *Y*
- **We say** $Y \rightarrow X$**, pronounced "***Y depends on X***"**
  - Means *Y* cannot be written before *X* is written
  - *X* is called the dependee, *Y* the depender
- **Can delay both writes, so long as order preserved**
  - Say you create a second file *B* in blocks *X* and *Y*
  - Only have to write each out once for both creates

Problem: Cyclic dependencies

- **Suppose you create file *A*, unlink file *B*, but delay writes**
  - Both files in same directory block *Y* & inode block *X*
- **Rule #1: Must write *A*'s inode before dir. entry ($Y \rightarrow X$)**
  - Otherwise, after crash directory will point to bogus inode
  - Worse yet, same inode # might be re-allocated
  - So could end up with file name *A* being an unrelated file
- **Rule #2: Must clear *B*'s dir. entry before writing inode ($X \rightarrow Y$)**
  - Otherwise, *B* could end up with too small a link count
  - File could be deleted while links to it still exist
- **Otherwise, fsck has to be slow**
  - Check every directory entry and every inode link count

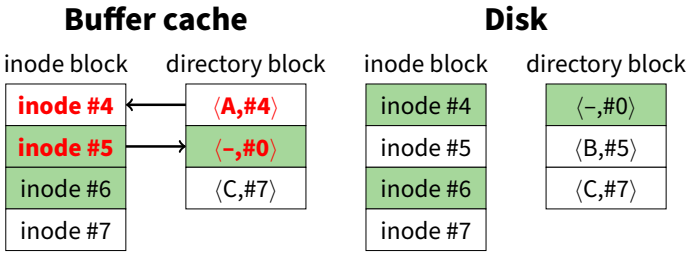
Cyclic dependencies illustrated



More problems

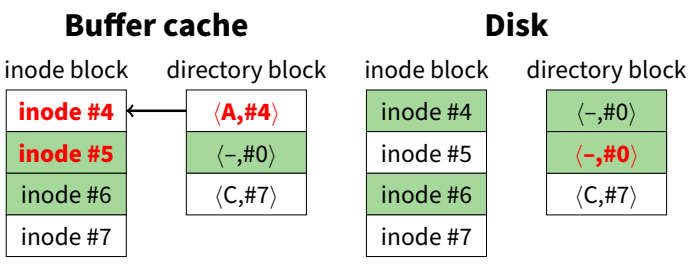
- **Crash might occur between ordered but related writes**
  - E.g., summary information wrong after block freed
- **Block aging**
  - Block that always has dependency will never get written back
- **Solution: *Soft updates* [Ganger]**
  - Write blocks in any order
  - But keep track of dependencies
  - When writing a block, temporarily roll back any changes you can't yet commit to disk
  - I.e., can't write block with any arrows pointing to dependees …but can temporarily undo whatever change requires the arrow

Breaking dependencies with rollback



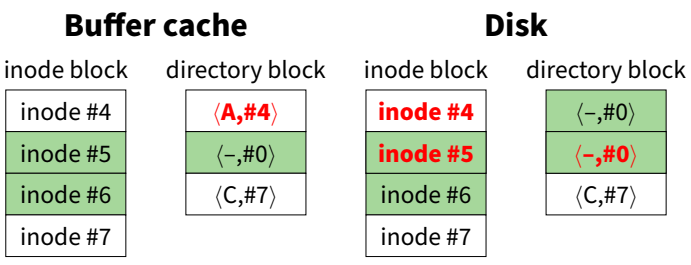
- **Created file A and deleted file B**
- **Now say we decide to write directory block…**
- **Can't write file name *A* to disk—has dependee**

Breaking dependencies with rollback



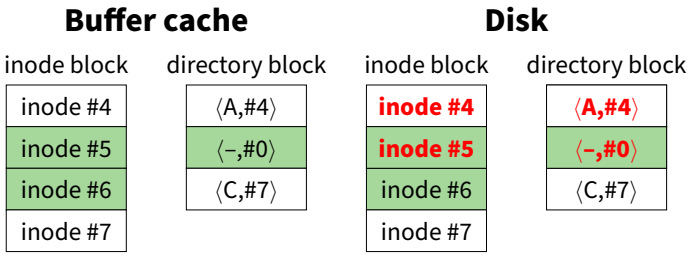
- **Undo file *A* before writing dir block to disk**
  - Even though we just wrote it, directory block still dirty
- **But now inode block has no dependees**
  - Can safely write inode block to disk as-is…

Breaking dependencies with rollback



- **Now inode block clean (same in memory as on disk)**
- **But have to write directory block a second time…**

Breaking dependencies with rollback



- **All data stably on disk**
- **Crash at any point would have been safe**

Soft updates

- **Structure for each updated field or pointer, contains:**
  - old value
  - new value
  - list of updates on which this update depends (*dependees*)
- **Can write blocks in any order**
  - But must temporarily undo updates with pending dependencies
  - Must lock rolled-back version so applications don't see it
  - Choose ordering based on disk arm scheduling
- **Some dependencies better handled by postponing in-memory updates**
  - E.g., when freeing block (e.g., because file truncated), just mark block free in bitmap after block pointer cleared on disk

Simple example

- **Say you create a zero-length file** *A*
- **Depender: Directory entry for** *A*
  - Can't be written untill dependees on disk
- **Dependees:**
  - Inode – must be initialized before dir entry written
  - Bitmap – must mark inode allocated before dir entry written
- **Old value: empty directory entry**
- **New value:** $\langle \text{filename } A, \text{inode } \# \rangle$
- **Can write directory block to disk any time**
  - Must substitute old value until inode & bitmap updated on disk
  - Once dir block on disk contains *A*, file fully created
  - Crash before *A* on disk, worst case might leak the inode

Operations requiring soft updates (1)

1. **Block allocation**
   - Must write: disk block, free map, & pointer (in inode or ind. block)
   - Disk block & free map must be written before pointer
   - Use Undo/redo on pointer (& possibly file size)
2. **Block deallocation**
   - Must write: cleared pointer & free map
   - Just update free map after pointer written to disk
   - Or just immediately update free map if pointer not on disk

- **Say you quickly append block to file then truncate**
  - You will know pointer to block not written because of the allocated dependency structure
  - So both operations together require no disk I/O!

Operations requiring soft updates (2)

3. **Link addition (see simple example)**
   - Must write: directory entry, inode, & free map (if new inode)
   - Inode and free map must be written before dir entry
   - Use undo/redo on i# in dir entry (because i# 0 ignored in dirent)
4. **Link removal**
   - Must write: directory entry, inode & free map (if nlinks==0)
   - Clear directory entry immediately
   - Must decrement nlinks only after pointer cleared
   - Decrement in-memory nlinks after directory written
   - If directory entry was never written, decrement immediately (again will know by presence of dependency structure)

- **Note: Quick create/delete requires no disk I/O**

Soft update issues

- ***fsync* – sycall to flush file changes to disk**
  - Must also flush directory entries, parent directories, etc.
- ***unmount* – flush all changes to disk on shutdown**
  - Some buffers must be flushed multiple times to get clean
- **Deleting large directory trees frighteningly fast**
  - *unlink* syscall returns even if inode/indir block not cached!
  - Dependencies allocated faster than blocks written
  - Cap # dependencies allocated to avoid exhausting memory
- **Useless write-backs**
  - Syncer flushes dirty buffers to disk every 30 seconds
  - Writing all at once means many dependencies unsatisfied
  - Fix syncer to write blocks one at a time
  - Tweak LRU buffer eviction to know about dependencies

Soft updates fsck

- **Split into foreground and background parts**
- **Foreground must be done before remounting FS**
  - Need to make sure per-cylinder summary info makes sense
  - Recompute free block/inode counts from bitmaps – very fast
  - Will leave FS consistent, but might leak disk space or inodes
- **Background does traditional fsck operations**
  - Do after mounting to recuperate free space
  - Can be using the file system while this is happening
  - Must be done in forground after a media failure
- **Difference from traditional FFS fsck:**
  - May have many, many inodes with non-zero link counts
  - Don't stick them all in lost+found (unless media failure)

Outline

An alternative: Journaling

- **Biggest crash-recovery challenge is inconsistency**
  - Have one logical operation (e.g., create or delete file)
  - Requires multiple separate disk writes
  - If only some of them happen, end up with big problems
- **Most of these problematic writes are to metadata**
- **Idea: Use a *write-ahead* log to *journal* metadata**
  - Reserve a portion of disk for a log
  - Write any metadata operation first to log, then to disk
  - After crash/reboot, re-play the log (efficient)
  - May re-do already committed change, but won't miss anything

Journaling (continued)

- **Group multiple operations into one log entry**
  - E.g., clear directory entry, clear inode, update free map—either all three will happen after recovery, or none
- **Performance advantage:**
  - Log is consecutive portion of disk
  - Multiple operations can be logged at disk b/w
  - Safe to consider updates committed when written to log
- **Example: delete directory tree**
  - Record all freed blocks, changed directory entries in log
  - Return control to user
  - Write out changed directories, bitmaps, etc. in background (sort for good disk arm scheduling)

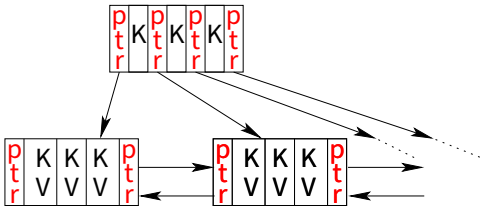
Journaling details

- **Must find oldest relevant log entry**
  - Otherwise, redundant and slow to replay whole log
  - Worse, old directory/indirect blocks reallocated as data could get corrupted by old replay (because only metadata logged)
- **Use checkpoints**
  - Once all records up to log entry *N* have been processed and affected blocks stably committed to disk…
  - Record *N* to disk either in reserved checkpoint location, or in checkpoint log record
  - Never need to go back before most recent checkpointed *N*
- **Must also find end of log**
  - Typically circular buffer; don't play old records out of order
  - Can include begin transaction/end transaction records
  - Also typically have checksum in case some sectors bad

Case study: XFS [Sweeney]

- **Main idea: Think big**
  - Big disks, files, large # of files, 64-bit everything
  - Yet maintain very good performance
- **Break disk up into *Allocation Groups* (AGs)**
  - 0.5 – 4 GiB regions of disk
  - New directories go in new AGs
  - Within directory, inodes of files go in same AG
  - Unlike cylinder groups, AGs too large to minimize seek times
  - Unlike cylinder groups, no fixed # of inodes per AG
- **Advantages of AGs:**
  - Parallelize allocation of blocks/inodes on multiprocessor (independent locking of different free space structures)
  - Can use 32-bit block pointers within AGs (keeps data structures smaller)

B+-trees



- **XFS makes extensive use of B+-trees**
  - Indexed data structure stores ordered Keys & Values
  - Keys must have an ordering defined on them
  - Stored data in blocks for efficient disk access
- **For B+-tree with n items, all operations $O(\log n)$:**
  - Retrieve closest ⟨key, value⟩ to target key k
  - Insert a new ⟨key, value⟩ pair
  - Delete ⟨key, value⟩ pair

B+-trees continued

- **See any algorithms book for details (e.g., [Cormen])**
- **Some operations on B-tree are complex:**
  - E.g., insert item into completely full B+-tree
  - May require "splitting" nodes, adding new level to tree
  - Would be bad to crash & leave B+tree in inconsistent state
- **Journal enables atomic complex operations**
  - First write all changes to the log
  - If crash while writing log, incomplete log record will be discarded, and no change made
  - Otherwise, if crash while updating B+-tree, will replay entire log record and write everything

B+-trees in XFS

- **B+-trees are complex to implement**
  - But once you've done it, might as well use everywhere
- **Use B+-trees for directories (keyed on filename hash)**
  - Makes large directories efficient
- **Make each inode a B+-tree**
  - No more FFS-style fixed block pointers
  - Instead, B+-tree maps: file offset → ⟨start block, # blocks⟩
  - Ideally file is one or small number of contiguous extents
  - Allows small inodes & no indirect blocks even for huge files
- **Use B+-tree to map inumber to location of inode**
  - High bits of inumber specify AG, middle bits are key in per-AG B+-tree, last few bits are position in a block of inodes
  - B+-tree in AG maps: starting i# → ⟨block #, free-map⟩
  - So free inodes tracked right in leaf of B+-tree

More B+-trees in XFS

- **Free extents tracked by *two* B+-trees**
  1. start block # → # free blocks
  2. # free blocks → start block #
- **Use journal to update both atomically & consistently**
- **#1 allows you to coalesce adjacent free regions**
- **#1 allows you to allocate near some target**
  - E.g., when extending file, put next block near previous one
  - When first writing to file, put data near inode
- **#2 allows you to do best fit allocation**
  - Leave large free extents for large files

Contiguous allocation

- **Ideally want each file contiguous on disk**
  - Sequential file I/O should be as fast as sequential disk I/O
  - Also keeps inodes small (fewer extents to index in B+-tree)
- **But how do you know how large a file will be?**
- **Idea: delayed allocation**
  - *write* syscall only affects the buffer cache
  - Allow write into buffers before deciding where to place on disk
  - Assign disk space only when buffers are flushed
- **Other advantages:**
  - Short-lived files never need disk space allocated
  - *mmap*ed files often written in random order in memory, but will be written to disk mostly contiguously
  - Write clustering: find other nearby stuff to write to disk

Journaling vs. soft updates

- **Both much better than FFS alone**
- **Some limitations of soft updates**
  - Very specific to FFS data structures (E.g., couldn't easily add B-trees like XFS—even directory rename not quite right)
  - Metadata updates may proceed out of order (E.g., create *A*, create *B*, crash—maybe only *B* exists after reboot)
  - Still need slow background fsck to reclaim space
- **Some limitations of journaling**
  - Disk write required for every metadata operation (whereas create-then-delete might require no I/O with soft updates)
  - Possible contention for end of log on multi-processor
  - *fsync* must sync other operations' metadata to log, too

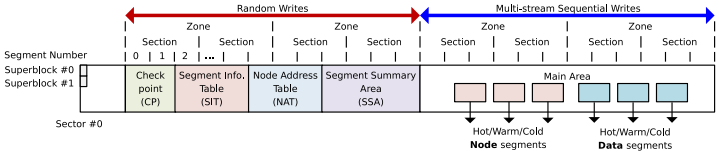
Outline

1. FFS in more detail
2. Crash recovery
3. Soft updates
4. Journaling
5. F2FS

Flash-Friendly File System (F2FS) [Lee]

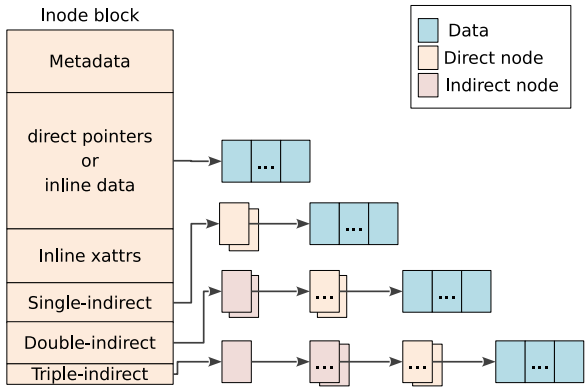
- **File system targeted at flash devices with FTL (e.g., SSDs)**
  - Try to do mostly large sequential writes
  - *Don't* attempt to do wear leveling (since have FTL anyway)
  - See also [Brown]
- **Break disk up into:**
  - Blocks – 4 KiB
  - Segments – 512 blocks, chosen so one block fits segment summary
  - Sections – 2^i segments (default $i = 0$), unit of log cleaning
  - Zones – n sections (default $n = 1$), if device internally comprises "subdevices," send parallel IO to different zones
- **Split device in two parts:**
  - Main area, in which to perform large sequential writes
  - Smaller metadata area has random writes, relies on FTL

F2FS layout



- **CP – Valid SIT/NAT sets, list of orphan (open+deleted) inodes**
  - Place version # in header+footer, use consistent CP with highest #
- **SIT – Per-segment block validity bitmap and count**
  - Two SIT areas and a small journal avoids updating in place
  - CP says which SIT area is active
- **NAT – Translates node numbers to actual block storing node**
  - Updated like SIT
- **SSA – Parent info for each block (e.g., inode+offset)**
  - Just updated in place, CP records active ones to recover

F2FS inode



- **Small files (<3,692 bytes) stored "inline" inside inode**
- **Node pointers use NAT table for level of indirection**
  - Lets F2FS move a node without updating parent pointers

Multi-head logging

Type	Temp.	Objects
Node	Hot	Direct node blocks for directories
	Warm	Direct node blocks for regular files
	Cold	Indirect node blocks
Data	Hot	Directory entry blocks
	Warm	Data blocks made by users
	Cold	Data blocks moved by cleaning; Cold data blocks specified by users; Multimedia file data

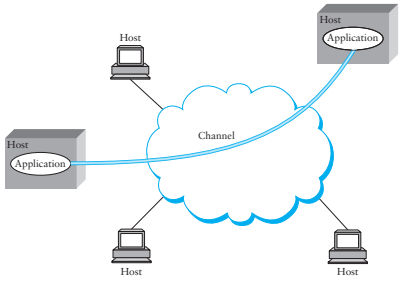
- **Two kinds of cleaning foreground and background**
  - Foreground (only if needed) greedily cleans most free section
  - Background just loads blocks into buffer cache and marks dirty
- **With no disk head, can efficiently maintain multiple logs**
  - Group data by similar expected lifetime (see above)
  - Means can clean empty or mostly empty sections

14. Networking

Outline

1. Networking overview
2. Systems issues
3. Implementing networking in the kernel
4. Network file systems

Computer networking



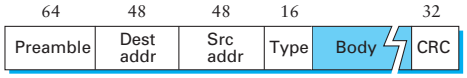
- **Goal: two applications on different computers exchange data**
- **Requires inter-process (not just inter-node) communication**

The 7-Layer and 4-Layer Models

	OSI	TCP/IP
7	Application	Applications (FTP, SMTP, HTTP, etc.)
6	Presentation	
5	Session	
4	Transport	TCP (host-to-host)
3	Network	IP
2	Data link	Network access (usually Ethernet)
1	Physical	

Link Layer: Ethernet

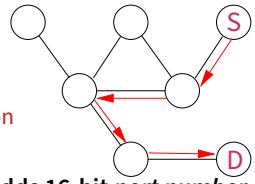
- **Originally designed for shared medium (coax), now generally not shared medium (switched)**
- **Vendors give each device a unique 48-bit *MAC address***
  - Specifies which card should receive a packet
- **Ethernet switches can scale to switch local area networks (thousands of hosts), but not much larger**
- **Packet format:**



  - Preamble helps device recognize start of packet
  - CRC allows receiving card to ignore corrupted packets
  - Body up to 1,500 bytes for same destination
  - All other fields must be set by sender's OS (NIC cards tell the OS what the card's MAC address is, Special addresses used for broadcast/multicast)

Network Layer: Internet Protocol (IP)

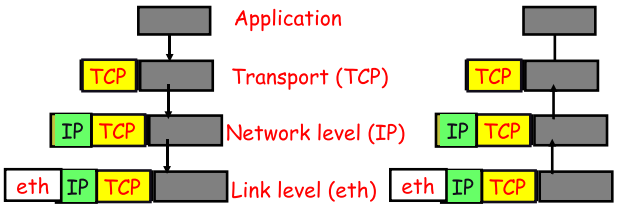
- **IP used to connect multiple networks**
  - Runs over a variety of physical networks—Ethernet, DSL, 5G
- **Every host has a unique 4-byte IP address (16-bytes for IPv6)**
  - (Or at least thinks it has, when there is address shortage)
- **Packets are *routed* based on destination IP address**
  - Address space is structured to make routing practical at global scale
  - E.g., 171.66.*.* goes to Stanford
  - So packets need IP addresses in addition to MAC addresses



- **Inside IP: UDP or TCP transport layer adds 16-bit *port number***
  - UDP – unreliable datagram protocol, exposes lost/reordered/delayed (but typically not corrupted) packets
  - TCP – transmission control protocol ≈ reliable pipe

Principle: Encapsulation

- **Stick packets inside packets**
- **How you realize packet switching and layering in a system**
  - E.g., an Ethernet packet may *encapsulate* an IP packet
  - An IP router *forwards* a packet from one Ethernet to another, creating a new Ethernet packet containing the same IP packet
  - In principle, an inner layer should not depend on outer layers (not always true)



Outline

1. Networking overview
2. Systems issues
3. Implementing networking in the kernel
4. Network file systems

Unreliability of IP

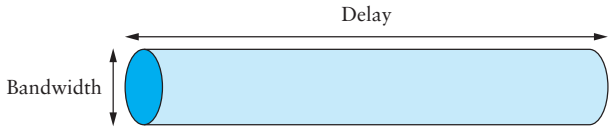
- **Network does not deliver packets reliably**
  - May drop, reorder, delay, corrupt, duplicate packets
- **OS must implement reliable TCP on top of IP**
- **Straw man: Wait for ack for each packet**
  - Send a packet, wait for acknowledgment, send next packet
  - If no ack, timeout and try again
- **Problems?**

Unreliability of IP

- **Network does not deliver packets reliably**
  - May drop, reorder, delay, corrupt, duplicate packets
- **OS must implement reliable TCP on top of IP**
- **Straw man: Wait for ack for each packet**
  - Send a packet, wait for acknowledgment, send next packet
  - If no ack, timeout and try again
- **Problems:**
  - Low performance over high-delay network (bandwidth is one packet per round-trip time)
  - Possible congestive collapse of network (if everyone keeps retransmitting when network overloaded)

Performance: Bandwidth-delay

- **Network *delay* over WAN will never improve much**
- **But *throughput* (bits/sec) is constantly improving**
- **Can view network as a pipe**



  - For full utilization want # bytes in flight $\geq$ bandwidth$\times$delay (But don't want to overload the network, either)
- **What if protocol doesn't involve bulk transfer?**
  - E.g., ping-pong protocol will have poor throughput
- **Another implication: Concurrency & response time critical for good network utilization**

A little bit about TCP

- **Want to save network from congestion collapse**
  - Packet loss usually means congestion, so back off exponentially
- **Want multiple outstanding packets at a time**
  - Get transmit rate up to *n*-packet window per round-trip
- **Must figure out appropriate value of *n* for network**
  - Slowly increase transmission by one packet per acked window
  - When a packet is lost, cut window size in half
- **Connection set up and teardown complicated**
  - Sender never knows when last packet might be lost
  - Must keep state around for a while (2MSL, e.g., 4 min) after close
- **Lots more hacks required for good performance**
  - Initially ramp *n* up faster (but too fast caused collapse in 1986 [Jacobson], so TCP had to be changed)
  - Fast retransmit when single packet lost

Lots of OS issues for TCP

- **Have to track unacknowledged data**
  - Keep a copy around until recipient acknowledges it
  - Keep timer around to retransmit if no ack
  - Receiver must keep out of order segments & reassemble
- **When to wake process receiving data?**
  - E.g., sender calls `write (fd, message, 8000);`
  - First TCP segment arrives, but is only 512 bytes
  - Could wake recipient, but useless w/o full message
  - TCP sets "PUSH" bit at end of 8000 byte write data
- **When to send short segment, vs. wait for more data**
  - Usually send only one unacked short segment
  - But bad for some apps, so provide `NODELAY` option
- **Must ack received segments very quickly**
  - Otherwise, effectively increases RTT, decreasing bandwidth

Outline

1. Networking overview
2. Systems issues
3. Implementing networking in the kernel
4. Network file systems

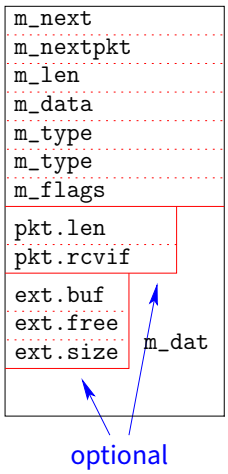
Sockets

- **Sockets ≈ bi-directional pipes**
- **Name endpoints by IP address and 16-bit *port number***
- **A *connection* is thus named by 5 components**
  - Protocol (TCP), local IP, local port, remote IP, remote port
  - Note TCP requires connected sockets, while UDP does not
- **Kernel stores connection state in a *protocol control block* structure (PCB)**
  - Keep all PCB's in a hash table
  - When packet arrives (if destination IP address belongs to host), use 5-tuple to find PCB and determine what to do with packet

Socket implementation

- **Need to implement layering efficiently**
  - Add UDP header to data, Add IP header to UDP packet, …
  - De-encapsulate Ethernet packet so IP code doesn't get confused by Ethernet header
- **Don't store packets in contiguous memory**
  - Moving data to make room for new header would be slow
- **BSD solution: mbufs [Leffler]**
  (Note [Leffler] calls `m_nextpkt` by old name `m_act`)
  - Small, fixed-size (256 byte) structures
  - Makes allocation/deallocation easy (no fragmentation)
- **BSD Mbufs working example for this lecture**
  - Linux uses `sk_buffs`, which are similar idea

mbuf details



- **Packets made up of multiple mbufs**
  - *Chained* together by `m_next`
  - Such linked mbufs called *chains*
- **Chains linked with `m_nextpkt`**
  - Linked chains known as *queues*
  - E.g., device output queue
- **Total mbuf size 256 B ⇒ ∼230 data bytes (depends on size of pointers)**
  - First in chain has `pkt` header
- ***Cluster* mbufs have more data**
  - `ext` header points to data
  - Up to 2 KB not collocated with mbuf
  - `m_dat` not used
- **`m_flags` is bitwise or of various bits**
  - E.g., if cluster, or if `pkt` header used

Adding/deleting data with mbufs

- **`m_data` always points to start of data**
  - Can be `m_dat`, or `ext.buf` for cluster mbuf
  - Or can point into middle of that area
- **To strip off a packet header (e.g., TCP/IP)**
  - Increment `m_data`, decrement `m_len`
- **To strip off end of packet**
  - Decrement `m_len`
- **Can add data to mbuf if buffer not full**
- **Otherwise, add data to chain**
  - Chain new mbuf at head/tail of existing chain

mbuf utility functions

- `mbuf *m_copym(mbuf *m, int off, int len, int wait);`
  - Creates a copy of a subset of an mbuf chain
  - Doesn't copy clusters, just increments reference count
  - `wait` says what to do if no memory (wait or return NULL)
- `void m_adj(struct mbuf *mp, int len);`
  - Trim |`len`| bytes from head or (if negative) tail of chain
- `mbuf *m_pullup(struct mbuf *n, int len);`
  - Put first `len` bytes of chain contiguously into first mbuf
- **Example: Ethernet packet containing IP datagram**
  - Trim Ethernet header using `m_adj`
  - Call `m_pullup (n, sizeof (ip_hdr));`
  - Access IP header as regular C data structure

Socket implementation

- **Each socket fd has associated socket structure with:**
  - Send and receive buffers
  - Queues of incoming connections (on listen socket)
  - A *protocol control block* (PCB)
  - A *protocol handle* (`struct protosw *`)
- **PCB contains protocol-specific info. E.g., for TCP:**
  - 5-tuple of protocol (TCP), source/destination IP address and port
  - Information about received packets & position in stream
  - Information about unacknowledged sent packets
  - Information about timeouts
  - Information about connection state (setup/teardown)

`protosw` structure

- **Goal: abstract away differences between protocols**
  - In C++, might use virtual functions on a generic socket struct
  - Here just put function pointers in `protosw` structure
- **Also includes a few data fields**
  - *domain, type, protocol* – to match `socket` syscall args, so know which `protosw` to select
  - *flags* – to specify important properties of protocol
- **Some protocol flags:**
  - ATOMIC – exchange atomic messages only (like UDP, not TCP)
  - ADDR – address given with messages (like unconnected UDP)
  - CONNREQUIRED – requires connection (like TCP)
  - WANTRCVD – notify socket of consumed data (e.g., so TCP can wake up a sending process blocked by flow control)

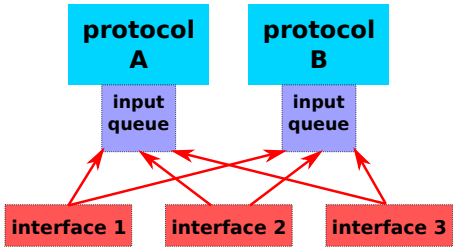
`protosw` functions

- `pr_slowtimo` – **called every 1/2 sec for timeout processing**
- `pr_drain` – **called when system low on space**
- `pr_input` – **returns mbuf chain of data read from socket**
- `pr_output` – **takes mbuf chain of data written to socket**
- `pr_usrreq` – **multi-purpose user-request hook**
  - Used for bind/listen/accept/connect/disconnect operations
  - Used for out-of-band data

Network interface cards

- **Each NIC driver provides an `ifnet` data structure**
  - Like `protosw`, tries to abstract away the details
- **Data fields:**
  - Interface name (e.g., “eth0”)
  - Address list (e.g., Ethernet address, broadcast address, . . .)
  - Maximum packet size
  - Send queue
- **Function pointers**
  - `if_output` – prepend header and enqueue packet
  - `if_start` – start transmitting queued packets
  - Also ioctl, timeout, initialize, reset

Input handling



- **NIC driver figures out protocol of incoming packet**
- **Enqueues packet for appropriate protocol handler**
  - If queue full, drop packet (can create livelock [Mogul])
- **Posts “soft interrupt” for protocol-layer processing**
  - Runs at lower priority than hardware (NIC) interrupt
    . . . but higher priority than process-context kernel code

Routing

- **An OS must route all transmitted packets**
  - Machine may have multiple NICs plus “loopback” interface
  - Which interface should a packet be sent to, and what MAC address should packet have?
- **Routing is based purely on the destination address**
  - Even if host has multiple NICs w. different IP addresses
  - (Linux *rules* let you select among routing tables by source IP)
- **OS maintains routing table**
  - Maps IP address & prefix-length → next hop
- **Use radix tree for efficient lookup**
  - Branch at each node in tree based on single bit of target
  - When you reach leaf, that is your next hop
- **Most OSes provide packet forwarding**
  - Received packets for non-local address routed out another interface

Outline

Network file systems

- **What's a network file system?**
  - Looks like a file system (e.g., FFS) to applications
  - But data potentially stored on another machine
  - Reads and writes must go over the network
  - Also called distributed file systems
- **Advantages of network file systems**
  - Easy to share if files available on multiple machines
  - Often easier to administer servers than clients
  - Access way more data than fits on your local disk
  - Network + remote buffer cache faster than local disk
- **Disadvantages**
  - Network + remote disk slower than local disk
  - Network or server may fail even when client OK
  - Complexity, security issues

NFS version 2 [Sandberg]

- **Background: ND (networked disk)**
  - Creates disk-like device even on diskless workstations
  - Can create a regular (e.g., FFS) file system on it
  - But no sharing—Why?
- **ND idea still used today by Linux NBD**
  - Useful for network booting/diskless machines, not file sharing
- **Some Goals of NFS**
  - Access same FS from multiple machines simultaneously
  - Maintain Unix semantics
  - Crash recovery
  - Competitive performance with ND
- **NFS version 2 protocol specified in [RFC 1094]**

NFS version 2 [Sandberg]

- **Background: ND (networked disk)**
  - Creates disk-like device even on diskless workstations
  - Can create a regular (e.g., FFS) file system on it
  - But no sharing—Why?
  - FFS assumes disk doesn't change under it
- **ND idea still used today by Linux NBD**
  - Useful for network booting/diskless machines, not file sharing
- **Some Goals of NFS**
  - Access same FS from multiple machines simultaneously
  - Maintain Unix semantics
  - Crash recovery
  - Competitive performance with ND
- **NFS version 2 protocol specified in [RFC 1094]**

NFS implementation

- **Virtualized the file system with *vnodes***
  - Ersatz virtual functions/interface/trait (like `protosw`)
- **Vnode structure represents an open (or openable) file**
- **Bunch of generic "vnode operations":**
  - lookup, create, open, close, getattr, setattr, read, write, fsync, remove, link, rename, mkdir, rmdir, symlink, readdir, readlink, …
  - Called through function pointers, so most system calls don't care what type of file system a file resides on
- **NFS vnode operations perform *Remote Procedure Calls* (RPC)**
  - Client sends request to server over network, awaits response
  - Each system call may require a series of RPCs
  - System mostly determined by RPC [RFC 1831] *Protocol*
  - Uses XDR protocol specification language [RFC 1832]

Stateless operation

- **Designed for "stateless operation"**
  - Motivated by need to recover from server crashes
- **Requests are self-contained**
- **Requests are idempotent**
  - Unreliable UDP transport
  - Client retransmits requests until it gets a reply
  - Writes must be stable before server returns
- **Can this really work?**

Stateless operation

- **Designed for "stateless operation"**
  - Motivated by need to recover from server crashes
- **Requests are self-contained**
- **Requests are** *mostly* **idempotent**
  - Unreliable UDP transport
  - Client retransmits requests until it gets a reply
  - Writes must be stable before server returns
- **Can this really work?**
  - Of course, FS not stateless – it stores files
  - E.g., *mkdir* can't be idempotent – second time dir exists
  - But many operations, e.g., *read*, *write* are idempotent
  - Importantly, server doesn't track open files, so reboot doesn't invalidate any file descriptors on clients

NFS version 3

- **Same general architecture as NFS 2**
- **Specified in RFC 1813 (subset of Open Group spec)**
  - XDR defines C structures that can be sent over network; includes tagged unions (to know which union field active)
  - Protocol defined as a set of Remote Procedure Calls (RPCs)
- **New access RPC**
  - Supports clients and servers with different uids/gids
- **Better support for caching**
  - Unstable writes while data still cached at client
  - More information for cache consistency
- **Better support for exclusive file creation**

NFSv3 File handles

```
struct nfs_fh3 {
  /* XDR notation for variable-length array
   * with 0-64 opaque bytes: */
  opaque data<64>;
};
```

- **Server assigns an opaque file handle to each file**
  - Client obtains first file handle out-of-band (mount protocol)
  - File handle hard to guess – security enforced at mount time
  - Subsequent file handles obtained through lookups
- **File handle internally specifies file system & file**
  - Device number, i-number, *generation number*, ...
  - Generation number changes when inode recycled
- **Handle generally *doesn't* contain filename**
  - Clients may keep accessing an open file after it's renamed

File attributes

```
struct fattr3 {
  ftype3 type;
  uint32 mode;
  uint32 nlink;
  uint32 uid;
  uint32 gid;
  uint64 size;
  uint64 used;
  specdata3 rdev;
  uint64 fsid;
  uint64 fileid;
  nfstime3 atime;
  nfstime3 mtime;
  nfstime3 ctime;
};
```

- **Most operations can optionally return** `fattr3`
- **Attributes used for cache-consistency**

Lookup

```
struct diropargs3 {
  nfs_fh3 dir;
  filename3 name;
};

struct lookup3resok {
  nfs_fh3 object;
  post_op_attr obj_attributes;
  post_op_attr dir_attributes;
};

union lookup3res switch (nfsstat3 status) {
case NFS3_OK:
  lookup3resok resok;
default:
  post_op_attr resfail;
};
```

- **Maps** $\langle \text{directory handle}, \text{filename} \rangle \rightarrow \text{handle}$
  - Client walks hierarchy one file at a time
  - No symlinks expanded or file system boundaries crossed
  - Client must expand symlinks

Create

```
struct create3args {
  diropargs3 where;
  createhow3 how;
};

union createhow3 switch (createmode3 mode) {
case UNCHECKED:
case GUARDED:
  sattr3 obj_attributes;
case EXCLUSIVE:
  createverf3 verf;
};
```

- `UNCHECKED` – **succeed if file exists**
- `GUARDED` – **fail if file exists**
- `EXCLUSIVE` – **persistent record of create**

Read

```
struct read3args {
  nfs_fh3 file;
  uint64 offset;
  uint32 count;
};
```

```
struct read3resok {
  post_op_attr file_attributes;
  uint32 count;
  bool eof;
  opaque data<>;
};
```

```
union read3res switch (nfsstat3 status) {
case NFS3_OK:
  read3resok resok;
default:
  post_op_attr resfail;
};
```

- **Offset explicitly specified (not implicit in handle)**
- **Client can cache result**

Data caching

- **Client can cache blocks of data read and written**
- **Consistency based on times in** `fattr3`
  - mtime: Time of last modification to file
  - ctime: Time of last change to inode
    (Changed by explicitly setting mtime, increasing size of file, changing permissions, etc.)
- **Algorithm: If mtime or ctime changed by another client, flush cached file blocks**

Write discussion

- **When is it okay to lose data after a crash?**
  - *Local file system?*

Write discussion

- **When is it okay to lose data after a crash?**
  - *Local file system?*
    If no calls to *fsync*, OK to lose 30 seconds of work after crash
  - *Network file system?*

Write discussion

- **When is it okay to lose data after a crash?**
  - *Local file system?*
    If no calls to *fsync*, OK to lose 30 seconds of work after crash
  - *Network file system?*
    What if server crashes but not client?
    Application not killed, so shouldn't lose previous writes
- **NFSv2 addresses problem by having server write data to disk before replying to a write RPC**
  - Caused performance problems
- **Could NFS2 clients just perform write-behind?**
  - Implementation issues – used blocking kernel threads on write
  - Semantics – how to guarantee consistency after server crash
  - Solution: small # of pending write RPCs, but write through on close; if server crashes, client keeps re-writing until acked

NFSv2 write call

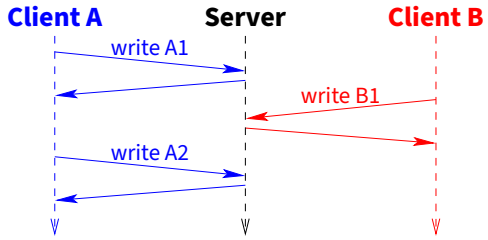
```
struct writeargs {
  fhandle file;
  /* ... */
  unsigned offset;
  /* ... */
  nfsdata data;
};
```

```
union attrstat
    switch (stat status) {
case NFS_OK:
  fattr attributes;
default:
  void;
};
```

```
attrstat NFSPROC_WRITE(writeargs) = 8;
```

- **On successful write, returns new file attributes**
- **Can NFSv2 keep cached copy of file after writing it?**

Write race condition



- **Suppose client overwrites 2-block file**
  - Client A knows attributes of file after writes A1 & A2
  - But client B could overwrite block 1 between the A1 & A2
  - No way for client A to know this hasn't happened
  - Must flush cache before next file read (or at least open)

NFSv3 Write arguments

```
struct write3args {
  nfs_fh3 file;
  uint64 offset;
  uint32 count;
  stable_how stable;
  opaque data<>;
};
```

```
enum stable_how {
  UNSTABLE = 0,
  DATA_SYNC = 1,
  FILE_SYNC = 2
};
```

- **Two goals for NFSv3 write:**
  - Don't force clients to flush cache after writes
  - Don't equate *cache* consistency with *crash* consistency
    I.e., don't wait for disk just so another client can see data

Write results

```
struct write3resok {
  wcc_data file_wcc;
  uint32 count;
  stable_how committed;
  writeverf3 verf;
};

union write3res
    switch (nfsstat3 status) {
case NFS3_OK:
  write3resok resok;
default:
  wcc_data resfail;
};
```

```
struct wcc_attr {
  uint64 size;
  nfstime3 mtime;
  nfstime3 ctime;
};

struct wcc_data {
  wcc_attr *before;
  post_op_attr after;
};
```

- **Several fields added to achieve these goals**

Data caching after a write

- **Write will change mtime/ctime of a file**
  - "`after`" will contain new times
  - With NFSv2, would require cache to be flushed
- **With NFSv3, "`before`" contains previous values**
  - If `before` matches cached values, no other client has changed file
  - Okay to update attributes without flushing data cache

Write stability

- **Server write must be at least as stable as requested**
- **If server returns write UNSTABLE**
  - Means permissions okay, enough free disk space, …
  - But data not on disk and might disappear (after crash)
- **If DATA_SYNC, data on disk, maybe not attributes**
- **If FILE_SYNC, operation complete and stable**

Commit operation

- **Client cannot discard any UNSTABLE write**
  - If server crashes, data will be lost
- **COMMIT RPC commits a range of a file to disk**
  - Invoked by client when client cleaning buffer cache
  - Invoked by client when user closes/flushes a file
- **How does client know if server crashed?**
  - Write and commit return `writeverf3`
  - Value changes after each server crash (can be boot time)
  - Client must resend all writes if verf value changes

Attribute caching

- **Close-to-open consistency**
  - Annoying if writes not visible after a file close (Edit file, compile on another machine, get old version)
  - Nowadays, all NFS opens fetch attributes from server
- **Still, lots of other need for attributes (e.g., `ls -al`)**
- **Attributes cached between 5 and 60 seconds**
  - Files recently changed more likely to change again
  - Do weighted cache expiration based on age of file
- **Drawbacks:**
  - Must pay for round-trip to server on every file open
  - Can get stale info when `stat`ting a file

NFS version 4 [RFC 3530]

- **Much more complicated than version 3**
  - NFS2: 27 page spec, NFS3: 126 pages, NFS4: 275 pages, NFS4.1: 617 pages
- **Designed to run over higher-latency networks**
  - Support for multi-component lookups to save RTTs
  - Support for batching multiple operations in one RPC
  - Support for leases (in two slides) and stateful (open, close) operation
- **Designed to be more generic and less Unix-specific**
  - E.g., support for extended file attributes, etc.
- **Lots of security stuff**
- **NFS 4.1 [RFC5661] has better support for NAS**
  - Store file data and metadata in different places

Callbacks

- **NFSv2 and v3 poll server for cache consistency**
  - Client requests attributes (via ACCESS) when file opened
  - Attributes validate or invalidate cached copy of file
- **Alternative: Server calls back to clients caching file (AFS)**
  - Invalidate immediately, rather than when cache needed
  - Requires server to maintain list of all clients caching info
- **Advantages**
  - Tight consistency, 0 RTT opens of cached files
- **Disadvantages**
  - Server must maintain a lot of state
  - Updates potentially slow
    - Must persistently record who is caching things on server
    - Must wait for *n* clients to acknowledge invalidations
  - When a client goes down, other clients will block

Leases

- **Hybrid mix of polling and callbacks**
  - Server agrees to notify client of changes for a limited period of time – the lease term
  - After the lease expires, client must poll for freshness
- **Avoids paying for a server round trip in many cases**
- **Server doesn't need to keep long-term track of callbacks**
  - E.g., lease time can be shorter than crash-reboot—no need to keep callbacks persistently
- **If client crashes, resume normal operation after lease expiration**

15. Protection

View access control as a matrix

Objects

Subjects	File 1	File 2	File 3	…	File n
User 1	read	write	-	-	read
User 2	write	write	write	-	-
User 3	-	-	-	read	read
…					
User m	read	write	read	write	read

- **Subjects (processes/users) access objects (e.g., files)**
- **Each cell of matrix has allowed permissions**

Two ways to slice the matrix

- **Along columns:**
  - Kernel stores list of who can access object along with object
  - Most systems you've used probably do this
  - Examples: Unix file permissions, Access Control Lists (ACLs)
- **Along rows:**
  - Capability systems do this
  - More on these later…

Outline

Example: Unix protection

- **Each process has a User ID & one or more group IDs**
- **System stores with each file (in the inode):**[1]
  - User who owns the file and group file is in
  - Permissions for user, any one in file group, and other
- **Shown by output of `ls -l` command:**

```
          user group other owner group
          ---- ---- ----- ----- -----
- rw- rw- r-- dm cs212 ... index.html
```

  - Each group of three letters specifies a subset of `read`, `write`, and `execute` permissions
  - User permissions apply to processes with same user ID
  - Else, group permissions apply to processes in same group
  - Else, other permissions apply

[1] Note: AFS mostly ignores these bits in favor of different, per-directory permission bits (per-user/group rlidwka)

Unix continued

- **Directories have permission bits, too**
  - Need write permission on a directory to create or delete a file
  - Execute permission means ability to use pathnames in the directory, separate from read permission which allows listing
- **Special user `root` (UID 0) has all privileges**
  - E.g., Read/write any file, change owners of files
  - Required for administration (backup, creating new users, etc.)
- **Example:**
  - `drwxr-xr-x 56 root wheel 4096 Apr 4 10:08 /etc`
  - Directory writable only by root, readable by everyone
  - Means non-root users cannot directly delete files in `/etc`

Non-file permissions in Unix

- **Many devices show up in file system**
  - E.g., `/dev/tty1` permissions just like for files
- **Other access controls not represented in file system**
- **E.g., must usually be root to do the following:**
  - Bind any TCP or UDP port number less than 1024
  - Change the current process's user or group ID
  - Mount or unmount most file systems
  - Create device nodes (such as `/dev/tty1`) in the file system
  - Change the owner of a file
  - Set the time-of-day clock; halt or reboot machine

Example: Login runs as root

- **List of Unix users with accounts typically stored in files in `/etc`**
  - Files `passwd`, `group`, and often `shadow` or `master.passwd`
- **For each user, files contain:**
  - Textual username (e.g., "`dm`", or "`root`")
  - Numeric user ID, and group ID(s)
  - One-way hash of user's password: {salt, H(salt, passwd)}
  - Should have tunable difficulty d: {d, salt, H_d(salt, passwd)}
  - Other information, such as user's full name, login shell, etc.
- **`/usr/bin/login` runs as root**
  - Reads username & password from terminal
  - Looks up username in `/etc/passwd`, etc.
  - Computes H(salt, typed password) & checks that it matches
  - If matches, sets group ID & user ID corresponding to username
  - Execute user's shell with `execve` system call

Setuid

- **Some legitimate actions require more privs than UID**
  - E.g., how should users change their passwords?
  - Stored in root-owned `/etc/passwd` & `/etc/shadow` files
- **Solution: Setuid/setgid programs**
  - Run with privileges of file's owner or group
  - Each process has *real* and *effective* UID/GID
  - *real* is user who launched setuid program
  - *effective* is owner/group of file, used in access checks
  - Actual rules and interfaces somewhat complicated [Chen]
- **Shown as "s" in file listings**
  - `-rws--x--x 1 root root 52528 Oct 29 08:54 /bin/passwd`
  - Obviously need to own file to set the setuid bit
  - Need to own file and be in group to set setgid bit

Setuid (continued)

- **Examples**
  - `passwd` – changes user's password
  - `su` – acquire new user ID (given correct password)
  - `sudo` – run one command as root
  - `ping` (historically) – uses raw IP sockets to send/receive ICMP
- **Have to be very careful when writing setuid code**
  - Attackers can run setuid programs any time (no need to wait for root to run a vulnerable job)
  - Attacker controls many aspects of program's environment
- **Example attacks when running a setuid program**
  - Change PATH or IFS if setuid prog calls system(3)
  - Set maximum file size to zero (if app rebuilds DB)
  - Close fd 2 before running program—may accidentally send error message into protected file

Linux capabilities

- **Wireshark needs network access, not ability to delete all files**
- **Linux subdivides root's privileges into ∼ 40 capabilities, e.g.:**
  - `cap_net_admin` – configure network interfaces (IP address, etc.)
  - `cap_net_raw` – use raw sockets (bypassing UDP/TCP)
  - `cap_sys_boot` – reboot; `cap_sys_time` – adjust system clock
- **Usually root gets all, but behavior can be modified by "securebits" (see prctl(2))**
- **Capabilities *don't* survive *execve* unless bits are set in *both* thread & inode (exception: ambient capabilities)**
- **"*Effective*" bit in inode acts like setuid for capability**

```
$ ls -al /usr/bin/dumpcap
-rwxr-xr-- 1 root wireshark 116808 Jan 30 06:23 /usr/bin/dumpcap
$ getcap /usr/bin/dumpcap
/usr/bin/dumpcap cap_dac_override,cap_net_admin,cap_net_raw=eip
```

  [Oops, cap_dac_override ≈ root! neeeded for USB capture]
- **See also: getcap(8), setcap(8), capsh(1)**

Other permissions

- **When can process A send a signal to process B with *kill*?**
  - Allow if sender and receiver have same effective UID
  - But need ability to kill processes you launch even if suid
  - So allow if real UIDs match, as well
  - Can also send SIGCONT w/o UID match if in same session
- **Debugger system call *ptrace***
  - Lets one process modify another's memory
  - Setuid gives a program more privilege than invoking user
  - So don't let a process ptrace a more privileged process
  - E.g., Require sender to match real & effective UID of target
  - Also disable/ignore setuid if ptraced target calls *exec*
  - Exception: root can *ptrace* anyone

Outline

A security hole

- **Even without root or setuid, attackers can trick root owned processes into doing things...**
- **Example: Want to clear unused files in** `/tmp`
- **Every night, automatically run this command as root:**

```
find /tmp -atime +3 -exec rm -f -- {} \;
```

- **find identifies files not accessed in 3 days**
  - executes `rm`, replacing `{}` with file name
- `rm -f --` ***path*** **deletes file** ***path***
  - Note "--" prevents *path* from being parsed as option
- **What's wrong here?**

An attack

find/rm	**Attacker**
	mkdir ("/tmp/badetc")
	creat ("/tmp/badetc/passwd")
readdir ("/tmp") → "badetc"	
lstat ("/tmp/badetc") → DIRECTORY	
readdir ("/tmp/badetc") → "passwd"	
unlink ("/tmp/badetc/passwd")	

An attack

find/rm	**Attacker**
	mkdir ("/tmp/badetc")
	creat ("/tmp/badetc/passwd")
readdir ("/tmp") → "badetc"	
lstat ("/tmp/badetc") → DIRECTORY	
readdir ("/tmp/badetc") → "passwd"	
	rename ("/tmp/badetc" → "/tmp/x")
	symlink ("/etc", "/tmp/badetc")
unlink ("/tmp/badetc/passwd")	

- **Time-of-check-to-time-of-use [TOCTTOU] bug**
  - find checks that `/tmp/badetc` is not symlink
  - But meaning of file name changes before it is used

xterm command

- **Provides a terminal window in X-windows**
- **Used to run with setuid root privileges**
  - Requires kernel pseudo-terminal (pty) device
  - Required root privs to change ownership of pty to user
  - Also writes protected utmp/wtmp files to record users
- **Had feature to log terminal session to file**

```
fd = open (logfile, O_CREAT|O_WRONLY|O_TRUNC, 0666);
/* ... */
```

- **What's wrong here?**

xterm command

- **Provides a terminal window in X-windows**
- **Used to run with setuid root privileges**
  - Requires kernel pseudo-terminal (pty) device
  - Required root privs to change ownership of pty to user
  - Also writes protected utmp/wtmp files to record users
- **Had feature to log terminal session to file**

```
if (access (logfile, W_OK) < 0)
  return ERROR;

fd = open (logfile, O_CREAT|O_WRONLY|O_TRUNC, 0666);
/* ... */
```

- `xterm` **is root, but shouldn't log to file user can't write**
- `access` **call avoids dangerous security hole**
  - Does permission check with *real*, not *effective* UID

xterm command

- **Provides a terminal window in X-windows**
- **Used to run with setuid root privileges**
  - Requires kernel pseudo-terminal (pty) device
  - Required root privs to change ownership of pty to user
  - Also writes protected utmp/wtmp files to record users
- **Had feature to log terminal session to file**

```
if (access (logfile, W_OK) < 0)
  return ERROR;

fd = open (logfile, O_CREAT|O_WRONLY|O_TRUNC, 0666);
/* ... */
```

- `xterm` **is root, but shouldn't log to file user can't write**
- `access` **call avoids dangerous security hole**
  - Does permission check with *real*, not *effective* UID
  - **Wrong: Another TOCTTOU bug**

An attack

xterm	Attacker
	creat ("/tmp/log")
access ("/tmp/log") → OK	
	unlink ("/tmp/log")
	symlink ("/tmp/log" → "/etc/passwd")
open ("/tmp/log")	

- **Attacker changes `/tmp/log` between check and use**
  - xterm unwittingly overwrites `/etc/passwd`
  - Another TOCTTOU bug
- **OpenBSD man page: "CAVEATS: access() is a potential security hole and should never be used."**

Preventing TOCCTOU

- **Use new APIs that are relative to an opened directory fd**
  - openat, renameat, unlinkat, symlinkat, faccessat
  - fchown, fchownat, fchmod, fchmodat, fstat, fstatat
  - `O_NOFOLLOW` flag to open avoids symbolic links in last component
  - But can still have TOCTTOU problems with hardlinks
- **Lock resources, though most systems only lock files (and locks are typically advisory)**
- **Wrap groups of operations in OS transactions**
  - Microsoft supports for transactions on Windows Vista and newer CreateTransaction, CommitTransaction, RollbackTransaction
  - A few research projects for POSIX [Valor] [TxOS]

SSH configuration files

- **SSH 1.2.12 client ran as root for several reasons:**
  - Needed to bind TCP port under 1024 (privileged operation)
  - Needed to read client private key (for host authentication)
- **Also needed to read & write files owned by user**
  - Read configuration file `~/.ssh/config`
  - Record server keys in `~/.ssh/known_hosts`
- **Software structured to avoid TOCTTOU bugs:**
  - First bind socket & read root-owned secret key file
  - Second drop *all* privileges—set real, & effective UIDs to user
  - Only then access user files
  - Idea: avoid using any user-controlled arguments/files until you have no more privileges than the user
  - What might still have gone wrong?

Trick question: ptrace bug

- **Actually do have more privileges than user!**
  - Bound privileged port and read host private key
- **Dropping privs allows user to "debug" SSH**
  - Depends on OS, but at the time several had *ptrace* implementations that made SSH vulnerable
- **Once in debugger**
  - Could use privileged port to connect anywhere
  - Could read secret host key from memory
  - Could overwrite local user name to get privs of other user
- **The fix: restructure into 3 processes!**
  - Perhaps overkill, but really wanted to avoid problems
- **Today some linux distros restrict ptrace with Yama**

A Linux security hole

- **Some programs acquire then release privileges**
  - E.g., `su user` is setuid root, becomes `user` if password correct
- **Consider the following:**
  - A and B unprivileged processes owned by attacker
  - A ptraces B (works even with Yama, as B could be child of A)
  - A executes "`su user`" to its own identity
  - With effective UID (EUID) 0, `su` asks for password & waits
  - While A's EUID is 0, B execs `su root`
    (B's exec honors setuid—not disabled—since A's EUID is 0)
  - A types password, gets shell, and is attached to `su root`
  - Can manipulate `su root`'s memory to get root shell

Editorial

- **Previous examples show two limitations of Unix**
- **Many OS security policies *subjective* not *objective***
  - When can you signal/debug process? Re-bind network port?
  - Rules for non-file operations somewhat incoherent
  - Even some file rules weird (creating hard links to files)
  - Lots of complexities when composing these policies
- **Correct code is much harder to write than incorrect**
  - Delete file without traversing symbolic link
  - Read SSH configuration file (requires 3 processes??)
  - Write mailbox owned by user in dir owned by root/mail
- **Don't just blame the application writers**
  - Must also blame the interfaces they program to

Outline

1. Unix protection
2. Unix security holes
3. Capability-based protection
4. Microarchitectural attacks

Another security problem [Hardy]

- **Setting: A multi-user time sharing system**
  - This time it's not Unix
- **Wanted Fortran compiler to keep statistics**
  - Modified compiler `/sysx/fort` to record stats in `/sysx/stat`
  - Gave compiler "home files license"—allows writing to anything in `/sysx` (kind of like Unix setuid)
- **What's wrong here?**

A confused deputy

- **Attacker could overwrite any files in** `/sysx`
  - System billing records kept in `/sysx/bill` got wiped
  - Probably command like `fort -o /sysx/bill file.f`
- **Is this a bug in the compiler** `fort`**?**
  - Original implementors did not anticipate extra rights
  - Can't blame them for unchecked output file
- **Compiler is a "confused deputy"**
  - Inherits privileges from invoking user (e.g., read `file.f`)
  - Also inherits privileges from home files license
  - Which source of authority is it serving on any given system call?
  - OS doesn't know if it just sees `open ("/sysx/bill", ...)`

Recall access control matrix

Objects

Subjects

	File 1	File 2	File 3	…	File n
User 1	read	write	-	-	read
User 2	write	write	write	-	-
User 3	-	-	-	read	read
…					
User m	read	write	read	write	read

Capabilities

- **Slicing matrix along rows yields capabilities**
  - E.g., For each process, store a list of objects it can access
  - Process explicitly invokes particular capabilities
- **Can help avoid confused deputy problem**
  - E.g., Must give compiler an argument that both specifies the output file and conveys the capability to write the file (think about passing a file descriptor, not a file name)
  - So compiler uses no *ambient authority* to write file
- **Three general approaches to capabilities:**
  - Hardware enforced (Tagged architectures like M-machine)
  - Kernel-enforced (Hydra, KeyKOS)
  - Self-authenticating capabilities (like Amoeba)
- **Good history in [Levy]**

Hydra [Wulf]

- **Machine & programing environment built at CMU in '70s**
- **OS enforced object modularity with capabilities**
  - Could only call object methods with a capability
- **Augmentation let methods manipulate objects**
  - A method executes with the capability list of the object, not the caller
- **Template methods take capabilities from caller**
  - So method can access objects specified by caller

KeyKOS [Bomberger]

- **Capability system developed in the early 1980s**
  - Inspired many later systems: EROS, Coyotos
- **Goal: Extreme security, reliability, and availability**
- **Structured as a "nanokernel"**
  - Kernel proper only 20,000 lines of C, 100KB footprint
  - Avoids many problems with traditional kernels
  - Traditional OS interfaces implemented outside the kernel (including binary compatibility with existing OSes)
- **Basic idea: No privileges other than capabilities**
  - Means kernel provides purely *objective* security mechanism
  - As objective as pointers to objects in OO languages
  - In fact, partition system into many processes akin to objects

Unique features of KeyKOS

- **Single-level store**
  - Everything is persistent: memory, processes, …
  - System periodically checkpoints its entire state
  - After power outage, everything comes back up as it was (may just lose the last few characters you typed)
- **"Stateless" kernel design only caches information**
  - All kernel state reconstructible from persistent data
- **Simplifies kernel and makes it more robust**
  - Kernel never runs out of space in memory allocation
  - No message queues, etc. in kernel
  - Run out of memory? Just checkpoint system

KeyKOS capabilities

- **Refered to as "keys" for short**
- **Types of keys:**
  - *devices* – Low-level hardware access
  - *pages* – Persistent page of memory (can be mapped)
  - *nodes* – Container for 16 capabilities
  - *segments* – Pages & segments glued together with nodes
  - *meters* – right to consume CPU time
  - *domains* – a thread context
- **Anyone possessing a key can grant it to others**
  - But creating a key is a privileged operation
  - E.g., requires "prime meter" to divide it into submeters

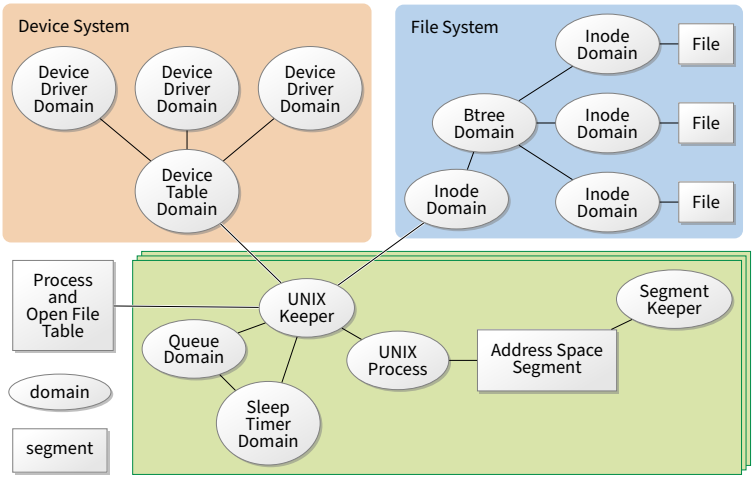
Capability details

- **Each domain has a number of key "slots":**
  - 16 general-purpose key slots
  - *address slot* – contains segment with process VM
  - *meter slot* – contains key for CPU time
  - *keeper slot* – contains key for exceptions
- **Segments also have an associated keeper**
  - Process that gets invoked on invalid reference
- **Meter keeper (allows creative scheduling policies)**
- **Calls generate return key for calling domain**
  - (Not required—other forms of message don't do this)

KeyNIX: UNIX on KeyKOS

- **"One kernel per process" architecture**
  - Hard to crash kernel
  - Even harder to crash system
- **A process's kernel is its keeper**
  - Unmodified Unix binary makes Unix syscall
  - Invalid KeyKOS syscall, transfers control to Unix keeper
- **Of course, kernels need to share state**
  - Use shared segment for process and file tables

KeyNIX overview



Keynix I/O

- **Every file is a different process**
  - Elegant, and fault isolated
  - Small files can live in a node, not a segment
  - Makes the `namei()` function very expensive
- **Pipes require queues**
  - This turned out to be complicated and inefficient
  - Interaction with signals complicated
- **Other OS features perform very well, though**
  - E.g., fork is six times faster than Mach 2.5

Self-authenticating capabilities

- **Every access must be accompanied by a capability**
  - For each object, OS stores random *check* value
  - Capability is: {Object, Rights, MAC(*check*, Rights)}
    (MAC = cryptographic *Message Authentication Code*)
- **OS gives processes capabilities**
  - Process creating resource gets full access rights
  - Can ask OS to generate capability with restricted rights
- **Makes sharing very easy in distributed systems**
- **To revoke rights, must change *check* value**
  - Need some way for everyone else to reacquire capabilities
- **Hard to control propagation**

Amoeba

- **A distributed OS, based on capabilities of form:**
  - server port, object ID, rights, check
- **Any server can listen on any machine**
  - Server port is hash of secret
  - Kernel won't let you listen if you don't know secret
- **Many types of object have capabilities**
  - Files, directories, processes, devices, servers (E.g., X windows)
- **Separate file and directory servers**
  - Can implement your own file server, or store other object types in directories, which is cool
- **Check is like a secret password for the object**
  - Server records check value for capabilities with all rights
  - Restricted capability's check is hash of old check, rights

Limitations of capabilities

- **IPC performance a losing battle with CPU makers**
  - CPUs optimized for "common" code, not context switches
  - Capability systems usually involve many IPCs
- **Capability model never fully took off as kernel API**
  - Requires changes throughout application software
  - Call capabilities "file descriptors" or "Java pointers" and people will use them
  - But discipline of pure capability system challenging so far
  - People sometimes quip that capabilities are an OS concept of the future and always will be
- **But real systems do use capabilities**
  - Firefox security based on language-level object capabilities
  - FreeBSD now ships with Capsicum, making capabilities available

Capsicum [Watson]

- **Capability API in FreeBSD 9**
- `cap_enter` **enters a process into capability mode**
  - Can no longer use absolute pathnames, "..", etc.
- `cap_new` **turns file descriptors into restricted capabilities**
  - ~60 individual permissions can be restricted per capability
  - E.g., disallow `fchmod` (which works on read-only fds)
- **Used by various base system binaries**
- **Supported by a growing number of applications**
- **Patches exist to use Capsicum for Chrome's sandboxing**

Outline

Cache timing attacks

```
const char *table;

int
victim (int secret_byte)
{
    return table[secret_byte*64];
}
```

- **Accessing memory based on secret data can leak the data**
- **Approach 1: Flush/Evict + Reload**
  - Share `table` with victim process (shared lib or deduplication)
  - Flush `table` from cache (`clflush` instruction, or overflow cache)
  - After `victim`, time reads of `table`, fast line tells you `secret_byte`
- **Approach 2: Prime + Probe**
  - No shared memory, but attacker primes cache with its own buffer
  - Victim's `table` access evicts one of attacker's cache lines
  - Slow cache line (+ cache mapping) reveals secret data

Speculative execution key to performance

```
unsigned char *array1, *array2;
int array1_size, array2_size;

int lookup (int input)
{
  if (input < array1_size)
    return array2[array1[input] * 4096];
  return -1;
}
```

- **CPU predicts branches to mask memory latency**
  - E.g., predict `input < array_size` even if `array1_size` not cached
  - Wait to get `array1_size` from memory before retiring instructions
  - *Squash* incorrectly predicted instructions by reverting registers
  - But can't revert cache state, only registers
- **Example: intel Haswell**
  - Specutatively executes up to 192 micro-ops
  - Indexes branch target buffer by bottom 31 bits of branch address

Spectre attack [Kocher]

```
unsigned char *array1, *array2;
int array1_size, array2_size;

int lookup (int input)
{
  if (input < array1_size)
    return array2[array1[input] * 4096];
  return -1;
}
```

- **Say attacker supplies `input`, wants to read `array1[input]`**
  - `input` can exceed bounds, reference any byte in address space
- **Ensure `array1` cached, but `array1_size` and `array2` uncached**
- **Flush+reload attack on `array2` now reveals `array1[input]`**
  - CPU will likely predict branch taken (don't usually overflow)
  - Speculatively load from `array2` before seeing `array1_size`
  - Reloaded cache line reveals `array1[input]`

Many more variants of Spectre

- **Attack on JavaScript JIT**
  - Malicious JavaScript reads secrets outside of JavaScript sandbox
- **eBPF compiles packet filters in kernel (e.g., for tcpdump)**
  - Can generate code to reveal arbitrary kernel memory
- **Can even use victim code that's not supposed to be executed**
  - Mistrain branch predictor on indirect branch
  - Speculatively execute arbitrary "spectre gadget" in victim process
  - Same cache impact even if gadget execution entirely squashed
  - Has been used to leak host memory from inside virtual machine
- **Use other speculation channels**
  - E.g., CPU predicts that previous store does not conflict with a load

Mitigation

- **Replace array bounds checks with index masking (used by Chrome)**
  - `return array2[array1[input&0xffff] * 4096]`
  - Limits distance of bounds violation
- **Place JavaScript sandbox in separate address space**
- **XOR pointers with type-dependent poison values (in JITs)**
  - Branch mispredictions on type checks XOR wrong values
- **Make CPUs a bit better about leaking state through side channels**
- **Insert "gratuitous" memory barriers to prevent speculation on sensitive data**
- **Unfortunately general solution still an open problem**

16. Security

Outline

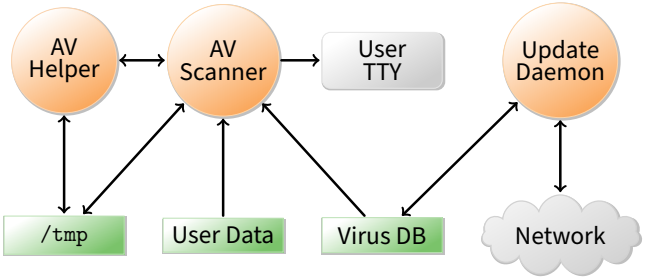
DAC vs. MAC

- **Most people are familiar with *discretionary* access control (DAC)**
  - Unix permission bits are an example
  - E.g., might set file `private` so that only group `friends` can read it:
    `-rw-r--- 1 dm friends 1254 Feb 11 20:22 private`
  - Anyone with access to information can further propagate that information at his/her discretion:
    `$ Mail sigint@enemy.gov < private`
- ***Mandatory* access control (MAC) can restrict propagation**
  - Security administrator may allow you to read but not disclose file
  - Not to be confused with Message Authentication Codes and Medium Access Control, also both "MAC"

MAC motivation

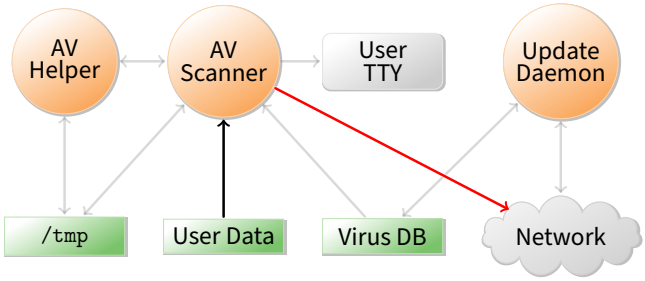
- **Prevent users from disclosing sensitive information (whether accidentally or maliciously)**
  - E.g., classified information requires such protection
- **Prevent software from surreptitiously leaking data**
  - Seemingly innocuous software may steal secrets in the background
  - Such a program is known as a *trojan horse*
- **Case study: Symantec AntiVirus 10**
  - Contained a remote exploit (attacker could run arbitrary code)
  - Inherently required access to all of a user's files to scan them
  - Can an OS protect private file contents under such circumstances?

Example: Anti-virus software



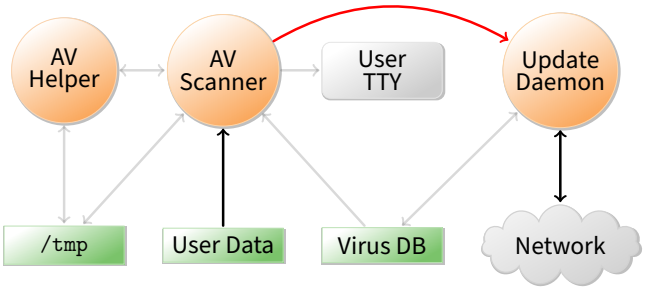
- **Scanner – checks for virus signatures**
- **Update daemon – downloads new virus signatures**
- **How can OS enforce security without trusting AV software?**
  - Must not leak contents of your files to network
  - Must not tamper with contents of your files

Example: Anti-virus software



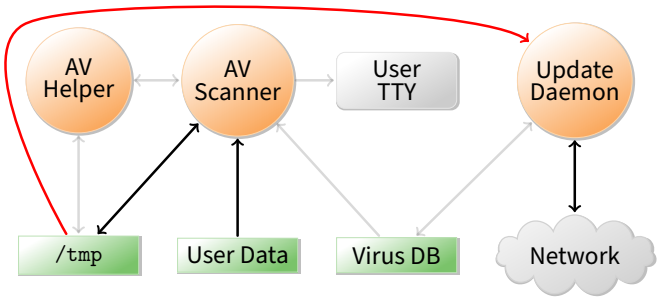
- **Scanner can write your private data to network**
- **Prevent scanner from invoking any system call that might send a network messages?**

Example: Anti-virus software



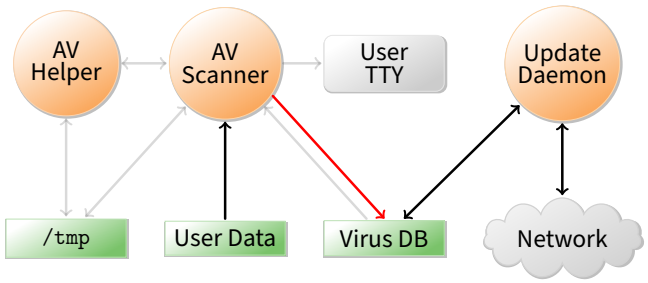
- **Scanner can send private data to update daemon**
- **Update daemon sends data over network**
  - Can cleverly disguise secrets in order/timing of update requests
- **Block IPC & shared memory system calls in scanner?**

Example: Anti-virus software



- **Scanner can write data to world-readable file in** `/tmp`
- **Update daemon later reads and discloses file**
- **Prevent update daemon from using** `/tmp`**?**

Example: Anti-virus software



- **Scanner can acquire read locks on virus database**
  - Encode secret user data by locking various ranges of file
- **Update daemon decodes data by detecting locks**
  - Discloses private data over the network
- **Have trusted software copy virus DB for scanner?**

The list goes on

- **Scanner can call setproctitle with user data**
  - Update daemon extracts data by running ps
- **Scanner can bind particular TCP or UDP port numbers**
  - Sends no network traffic, but detectable by update daemon
- **Scanner can relay data through another process**
  - Call ptrace to take over process, then write to network
  - Use sendmail, httpd, or portmap to reveal data
- **Disclose data by modulating free disk space**
- **Can we ever convince ourselves we've covered all possible communication channels?**
  - Not without a more systematic approach to the problem

Outline

Bell-La Padula model [BL]

- **View the system as subjects accessing objects**
  - Access control: take *requests* as input and output *decisions*
- **Four modes of access are possible:**
  - execute – no observation or alteration
  - read – observation
  - append – alteration
  - write – both observation and modification
- **An *access matrix* M encodes permissible access types**
  - As in last lecture, subjects are rows, objects are columns
- **The *current access set*, b, is** $(\text{subj}, \text{obj}, \text{attr})$ **triples**
  - Encodes accesses in progress (e.g., open files)
  - At a minimum, $(S, O, A) \in b$ requires A permitted by cell $M_{S,O}$

Security levels

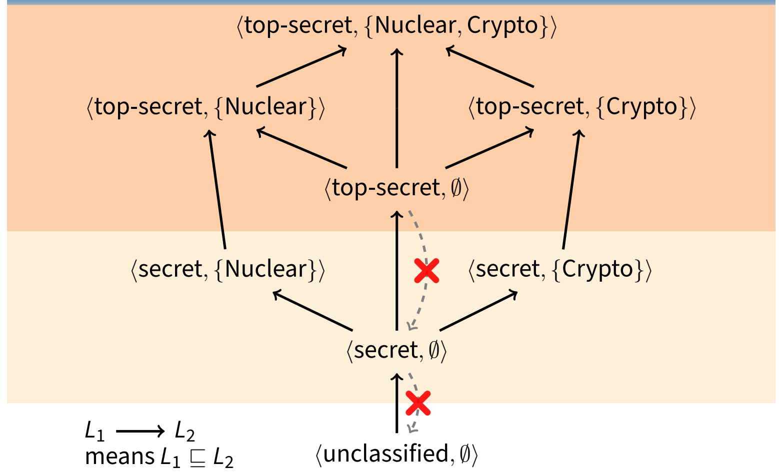
- **A *security level* or *label* is a pair** (c, s) **where:**
  - c = classification – E.g., 1 = unclassified, 2 = secret, 3 = topsecret
  - s = category-set – E.g., Nuclear, Crypto, Russia, …
- (c_1, s_1) ***dominates*** (c_2, s_2) **iff** $c_1 \geq c_2$ **and** $s_1 \supseteq s_2$
  - L_1 *dominates* L_2 is sometimes written $L_1 \succ L_2$ or $L_1 \sqsupseteq L_2$
  - Labels then form a *lattice* (partial order with lub & glb)
- **Inverse of dominates relation is *can flow to*, written** $\sqsubseteq$
  - $L_1 \sqsubseteq L_2$ ("L_1 can flow to L_2") means L_2 dominates L_1
- **Subjects and objects are assigned security levels**
  - level(S), level(O) – security level of subject/object
  - current-level(S) – subject may operate at lower level
  - level(S) bounds current-level(S) (current-level(S) $\sqsubseteq$ level(S))
  - Since level(S) is max, sometimes called S's *clearance*

Security properties

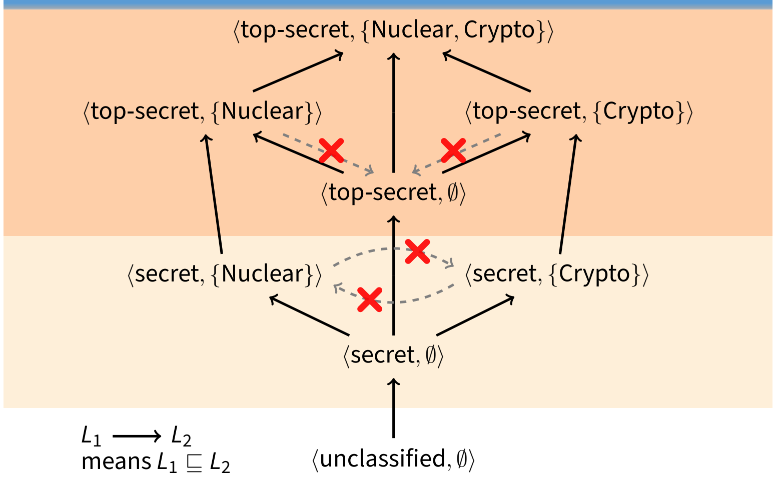
Two access control properties with respect to labels:

- **The simple security or *ss-property* (DAC):**
  - For any $(S, O, A) \in b$, if A includes observation, then level(S) must dominate level(O), i.e., level(O) $\sqsubseteq$ level(S)
  - E.g., an unclassified user cannot read a top-secret document
- **The star security or $\star$-*property* (MAC):**
  - If any subject both observes O_1 and modifies O_2, then level(O_2) dominates level(O_1), i.e., level(O_1) $\sqsubseteq$ level(O_2).
  - E.g., no subject can read a top secret file, then write a secret file
  - More precisely, given $(S, O, A) \in b$:
    if $A = r$ then level(O) $\sqsubseteq$ current-level(S) "no read up"
    if $A = a$ then current-level(S) $\sqsubseteq$ level(O) "no write down"
    if $A = w$ then current-level(S) = level(O)

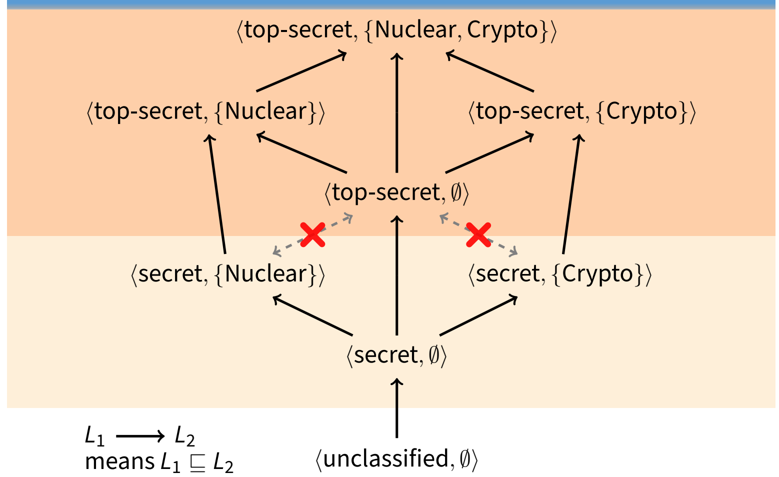
Labels form a lattice [Denning]



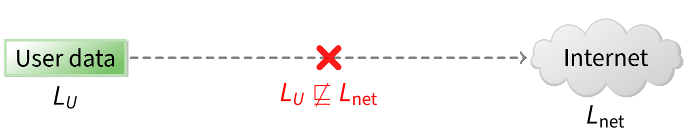
Labels form a lattice [Denning]



Labels form a lattice [Denning]

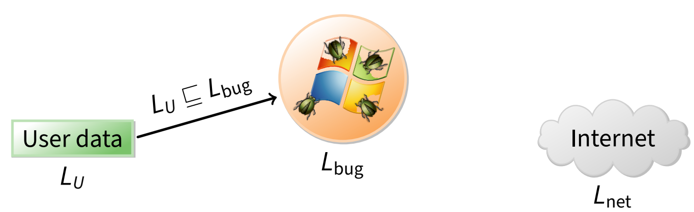


$\sqsubseteq$ is transitive



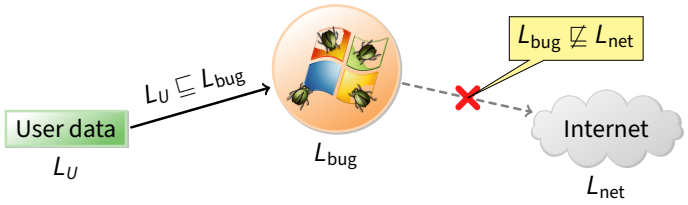
- **Transitivity makes it easier to reason about security**
- **Example: Label user data so it cannot flow to Internet**
  - Policy holds regardless of what other software does

$\sqsubseteq$ is transitive



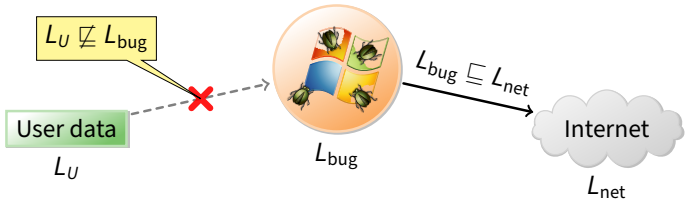
- **Transitivity makes it easier to reason about security**
- **Example: Label user data so it cannot flow to Internet**
  - Policy holds regardless of what other software does
- **Suppose untrustworthy software reads file**
  - Process labeled L_{bug} reads file, so must have $L_U \sqsubseteq L_{bug}$

$\sqsubseteq$ is transitive



- **Transitivity makes it easier to reason about security**
- **Example: Label user data so it cannot flow to Internet**
  - Policy holds regardless of what other software does
- **Suppose untrustworthy software reads file**
  - Process labeled L_{bug} reads file, so must have $L_U \sqsubseteq L_{bug}$
  - If $L_U \sqsubseteq L_{bug}$ and $L_U \not\sqsubseteq L_{net}$, it follows that $L_{bug} \not\sqsubseteq L_{net}$.

$\sqsubseteq$ is transitive



- **Transitivity makes it easier to reason about security**
- **Example: Label user data so it cannot flow to Internet**
  - Policy holds regardless of what other software does
- **Conversely, a process that can write to the network cannot read the file**

Straw man MAC implementation

- **Take an ordinary Unix system**
- **Put labels on all files and directories to track levels**
- **Each user U assigned a security clearance, level(U), on login**
- **Determine current security level dynamically**
  - When U logs in, start with lowest curent-level
  - Increase current-level as higher-level files are observed (sometimes called a *floating label* system)
  - If U's level does not dominate current-level, kill program
  - Kill program that writes to file if current label can't flow to file label
- **Is this secure?**

No: Covert channels

- **System rife with *covert storage channels***
  - Low current-level process executes another program
  - New program reads sensitive file, gets high current-level
  - High program exploits covert channels to pass data to low
- **E.g., high program inherits read-only file descriptor**
  - Can pass 4-bytes of information to low program in file offset
- **Other storage channels:**
  - Exit value, signals, file locks, terminal escape codes, ...
- **If we eliminate storage channels, is system secure?**

No: Timing channels

- **Example: CPU utilization**
  - To send a 0 bit, use 100% of CPU in busy-loop
  - To send a 1 bit, sleep and relinquish CPU
  - Repeat to transfer more bits
- **Example: Resource exhaustion**
  - High program allocates all physical memory if bit is 1
  - If low program slow from paging, knows less memory available
- **More examples: Disk head position, processor cache/TLB polution, ...**

Reducing covert channels

- **Observation: Covert channels come from sharing**
  - If you have no shared resources, no covert channels
  - Extreme example: Just use two computers (common in DoD)
- **Problem: Sharing needed**
  - E.g., read unclassified data when preparing classified
- **In general, can only hope to bound bandwidth of covert channels**
- **One approach: Strict partitioning of resources**
  - Strictly partition and schedule resources between levels
  - Occasionally reapportion resources based on usage [Browne]
  - Do so infrequently to bound leaked information
  - Approach still not so good if many security levels possible

Declassification

- **Sometimes need to prepare unclassified report from classified data**
- **Declassification happens outside of traditional access control model**
  - Present file to security officer for downgrade
- **Job of declassification often not trivial**
  - E.g., Microsoft word saves a lot of undo information
  - This might be all the secret stuff you cut from document
  - Another bad mistake: Redact PDF using black censor bars over or under text, leaving text selectable (e.g., [Cluley1], [Cluley2])

Biba integrity model [Biba]

- **Problem: How to protect integrity**
  - Suppose text editor gets trojaned, subtly modifies files
  - Might mess up attack plans even without leaking anything
- **Observation: Integrity is the converse of secrecy**
  - In secrecy, want to avoid writing to lower-secrecy files
  - In integrity, want to avoid writing higher-integrity files
- **Use integrity hierarchy parallel to secrecy one**
  - Now *security level* is a $\langle c, i, s \rangle$ triple, where i = integrity
  - $\langle c_1, i_1, s_1 \rangle \sqsubseteq \langle c_2, i_2, s_2 \rangle$ iff $c_1 \leq c_2$ and $i_1 \geq i_2$ and $s_1 \subseteq s_2$
  - Only trusted users can operate at higher integrity (which is visually lower in the lattice—opposite of secrecy)
  - If you read less authentic data, your current integrity level gets lowered (putting you up higher in the lattice), and you can no longer write higher-integrity files

Outline

LOMAC [Fraser]

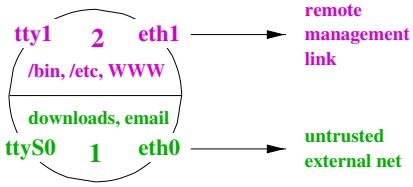
- **MAC not widely accepted outside military**
- **LOMAC's goal: make MAC more palatable**
  - Stands for Low water Mark Access Control
- **Concentrates on Integrity**
  - More important goal for many settings
  - E.g., don't want viruses tampering with all your files
  - Also don't have to worry as much about covert channels
- **Provides reasonable defaults (minimally obtrusive)**
- **Has actually had impact**
  - Originally available for Linux (2.2)
  - Now ships with FreeBSD
  - Windows introduced similar Mandatory Integrity Control (MIC), but not actually mandatory

LOMAC overview

- **Subjects are *jobs* (essentially processes)**
  - Each subject labeled with an integrity number (e.g., 1, 2)
  - Higher numbers mean more integrity (so unfortunately $2 \sqsubseteq 1$ by earlier notation)
  - Subjects can be reclassified on observation of low-integrity data
- **Objects (files, pipes, etc.) also labeled w. integrity level**
  - Object integrity level is fixed and cannot change
- **New objects have level of their creator**
- **Security: Low-integrity subjects cannot write to high integrity objects**

LOMAC defaults

[note: can-flow-to is downward; opposite of earlier diagram]



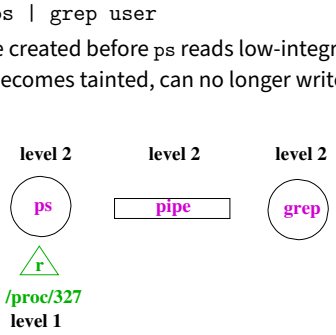
- **Two levels: 1 and 2**
- **Level 2 (high-integrity) contains:**
  - FreeBSD/Linux files intact from distro, static web server config
  - The console, trusted terminals, trusted network
- **Level 1 (low-integrity) contains**
  - NICs connected to Internet, untrusted terminals, etc.
- **Idea: Suppose worm compromises your web server**
  - Worm comes from external network → level 1
  - Won't be able to muck with system files or web server config

The self-revocation problem

- **Want to integrate with Unix unobtrusively**
- **Problem: Application expectations**
  - Kernel access checks usually done at file open time
  - Legacy applications don't pre-declare they will observe low-integrity data
  - An application can "taint" itself unexpectedly, revoking its own permission to access an object it created

Self-revocation example

- **User has high-integrity (level 2) shell**
- **Runs:** `ps | grep user`
  - Pipe created before `ps` reads low-integrity data
  - `ps` becomes tainted, can no longer write to `grep`

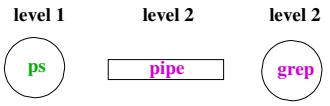


Self-revocation example

- **User has high-integrity (level 2) shell**
- **Runs:** `ps | grep user`
  - Pipe created before `ps` reads low-integrity data
  - `ps` becomes tainted, can no longer write to `grep`

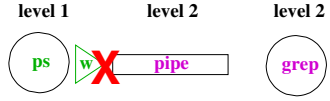
Self-revocation example

- **User has high-integrity (level 2) shell**
- **Runs:** `ps | grep user`
  - Pipe created before `ps` reads low-integrity data
  - `ps` becomes tainted, can no longer write to `grep`



Self-revocation example

- **User has high-integrity (level 2) shell**
- **Runs:** `ps | grep user`
  - Pipe created before `ps` reads low-integrity data
  - `ps` becomes tainted, can no longer write to `grep`



Solution

- **Don't consider pipes to be real objects**
- **Join multiple processes together in a "job"**
  - Pipe ties processes together in job
  - Any processes tied to job when they read or write to pipe
  - So will lower integrity of both ps and grep
- **Similar idea applies to shared memory and IPC**
- **Summary: LOMAC applies MAC to non-military systems**
  - But doesn't allow military-style security policies (i.e., with secrecy, various categories, etc.)

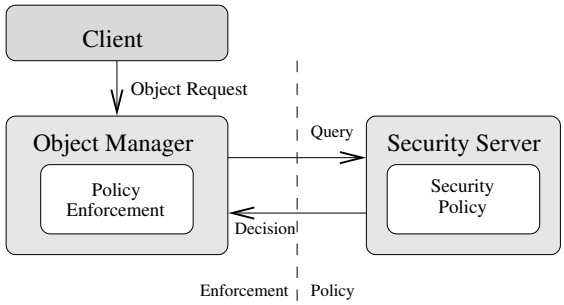
Outline

The flask security architecture

- **Problem: Military needs adequate secure systems**
  - How to create civilian demand for systems military can use?
- **Idea: Separate policy from enforcement mechanism**
  - Most people will plug in simple DAC policies
  - Military can take system off-the-shelf, plug in new policy
- **Requires putting adequate hooks in the system**
  - Each object has manager that guards access to the object
  - Conceptually, manager consults security server on each access
- **Flask security architecture prototyped in fluke**
  - Now part of SElinux

Following figures from [Spencer]

Architecture



- **Kernel mediates access to objects at "interesting" points**
- **Kicks decision up to external (user-level) security server**

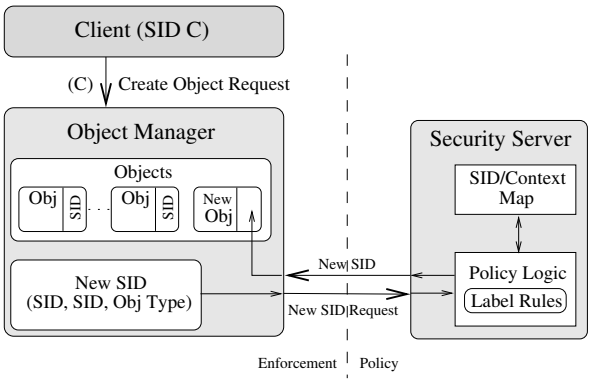
Challenges

- **Performance**
  - Adding hooks on every operation
  - People who don't need security don't want slowdown
- **Using generic enough data structures**
  - Object managers independent of policy still need to associate data structures (e.g., labels) with objects
- **Revocation**
  - May interact in a complicated way with any access caching
  - Once revocation completes, new policy must be in effect
  - Bad guy cannot be allowed to delay revocation completion indefinitely

Basic flask concepts

- **All objects are labeled with a *security context***
  - Security context is an arbitrary string—opaque to object manager in the kernel
- **Security contexts abbreviated with *security IDs* (SIDs)**
  - 32-bit integer, interpretable only by security server
  - Not valid across reboots (can't store in file system)
  - Fixed size makes it easier for object manager to handle
- **Queries to server done in terms of SIDs**
  - Create (client SID, old obj SID, obj type)? $\rightarrow$ SID
  - Allow (client SID, obj SID, perms)? $\rightarrow$ {yes, no}

Creating new object



Security server interface [Loscocco]

```
int security_compute_av(
    security_id_t ssid, security_id_t tsid,
    security_class_t tclass, access_vector_t requested,
    access_vector_t *allowed, access_vector_t *decided,
    __u32 *seqno);
```

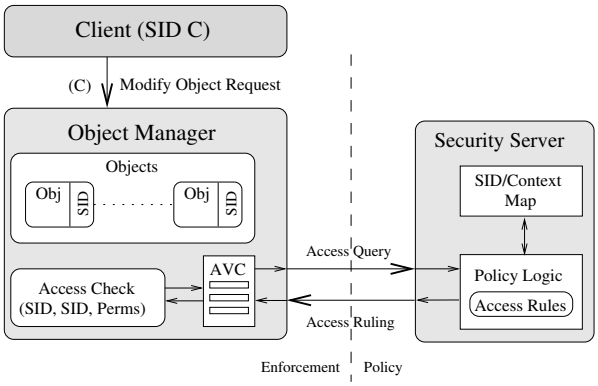
- `ssid`, `tsid` – **source and target SIDs**
- `tclass` – **type of target**
  - E.g., regular file, device, raw IP socket, TCP socket, ...
- **Server can decide more than it is asked for**
  - `access_vector_t` is a bitmask of permissions
  - `decided` can contain more than `requested`
  - Effectively implements decision prefetching
- `seqno` **used for revocation (in a few slides)**

Access vector cache (AVC)

- **Want to minimize calls into security server**
- **AVC caches results of previous decisions**
  - Note: Relies on simple enumerated permissions
- **Decisions therefore cannot depend on parameters:**
  - ✗ Andy can authorize expenses up to $999.99
  - ✗ Bob can run processes at priority 10 or higher
- **Decisions also limited to two SIDs**
  - Complicates file relabeling, which requires 3 checks:

Source	Target	Permission checked
Subject SID	Old file SID	Relabel-From
Subject SID	New file SID	Relabel-To
Old file SID	New file SID	Transition-From

AVC in a query



AVC interface

```
int avc_has_perm_ref(
    security_id_t ssid, security_id_t tsid,
    security_class_t tclass, access_vector_t requested,
    avc_entry_ref_t *aeref);
```

- `avc_entry_ref_t` **points to cached decision**
  - Contains `ssid`, `tsid`, `tclass`, decision vec., & recently used info
- `aeref` **argument is hint**
  - After first call, will be set to relevent AVC entry
  - On subsequent calls speeds up lookup
- **Example: New kernel check when binding a socket:**

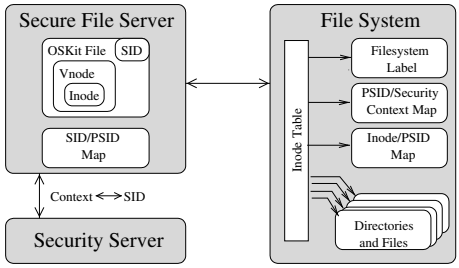
```
ret = avc_has_perm_ref(
    current->sid, sk->sid, sk->sclass,
    SOCKET__BIND, &sk->avcr);
```

  - Now `sk->avcr` is likely to be speed up next socket op

Revocation support

- **Decisions may be cached in AVC entries**
- **Decisions may implicitly be cached in migrated permissions**
  - E.g., Unix checks file write permission on *open*
  - But may want to disallow future writes even on open file
  - Write permission migrated into file descriptor
  - May also migrate into page tables/TLB w. `mmap`
  - Also may migrate into open sockets/pipes, or operations in progress
- **AVC contains hooks for callbacks**
  - After revoking in AVC, AVC makes callbacks to revoke migrated permissions
  - `seqno` can be used to ensure strict ordering of policy changes

Persistence



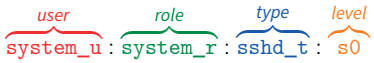
- **Must label persistent objects in file system**
  - Persistently map each file/directory to a security context
  - Security contexts are variable length, so add level of indirection
  - "Persistent SIDs" (PSIDs) – numbers local to each file system

Transitioning SIDs

- **May need to relabel objects**
  - E.g., files in file system
- **Processes may also want to transition their SIDs**
  - Depends on existing permission, but also on program
  - SELinux allows programs to be defined as *entrypoints*
  - Thus, can restrict with which programs users enter a new SID (similar to the way setuid transitions uid on program entry)

SELinux contexts

- **In practice, SELinux contexts have four parts:**



- ***user* is not Unix user ID, e.g.:**

```
$ id
uid=1000(dm) gid=1000(dm) groups=1000(dm) 119(admin)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c255
$ /bin/su
Password:
# id
uid=0(root) gid=0(root) groups=0(root)
context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c255
# newrole -r system_r -t sysadm_t
Password:
# id -Z
unconfined_u:system_r:sysadm_t:s0-s0:c0.c255
```


Users, roles, types

- **SElinux user is assigned on login, based on rules**

```
# semanage login -l
Login Name      SELinux User    MLS/MCS Range
__default__     unconfined_u    s0-s0:c0.c255
root            root_u          s0-s0:c0.c255
```

- **A user is allowed to assume different roles w.** `newrole`
- **But roles are restricted by SElinux (not Unix) users**

```
# semanage user -l
SELinux User    ... SELinux Roles
root                staff_r sysadm_r system_r
unconfined_u        system_r unconfined_r
user_u              user_r
```


Types

- **Each role allows only certain *types***
  - Can check with `seinfo -x --role=`*name*
- **Types allow non-hierarchical security policies**
  - Each subject is assigned a *domain*, each object a *type*
  - Policy stated in terms of what each domain can to do each type
- **Example: Suppose you wish to enforce that each invoice undergoes the following processing:**
  - Receipt of the invoice recorded by a clerk
  - Receipt of of the merchandise verified by purchase officer
  - Payment of invoice approved by supervisor
- **Can encode state of invoice by its type**
  - Set transition rules to enforce all steps of process

Example: Loading kernel modules

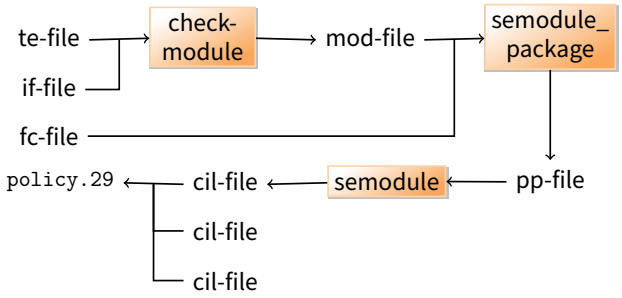
```
(1) allow sysadm_t insmod_exec_t:file x_file_perms;
(2) allow sysadm_t insmod_t:process transition;
(3) allow insmod_t insmod_exec_t:process { entrypoint execute };
(4) allow insmod_t sysadm_t:fd inherit_fd_perms;
(5) allow insmod_t self:capability sys_module;
(6) allow insmod_t sysadm_t:process sigchld;
```

1. Allow sysadm domain to run insmod
2. Allow sysadm domain to transition to insmod
3. Allow insmod program to be entrypoint for insmod domain
4. Let insmod inherit file descriptors from sysadm
5. Let insmod use CAP_SYS_MODULE (load a kernel module)
6. Let insmod signal sysadm with SIGCHLD when done

Policy specification

- **Very complicated sets of rules**
  - E.g., on Fedora, `sesearch --all | wc -l` shows 73K rules
  - Rules based mostly on types
- **Allowed/restricted transitions very important**
  - E.g., init can run initscripts, can run httpd
  - Nowadays `systemd` needs to be able to transition to arbitrary types
  - `httpd` program has special `httpd_exec_t` type, allows process to have `httpd_t` type.
  - Might label `public_html` directories so `httpd` can access them, but not access rest of home directory
- **Can also use levels to enforce MLS**
  - E.g., "`:s0-s0:c0.c255`" means process is at sensitivity `s0` with no categories, but has all categories in clearance.

Policy construction



- **Very low quality tooling around policy construction**
  - Broken build systems, incompatible kernel policy formats, …
- **Hard to check** `/sys/fs/selinux/policy` **matches expectations**
  - No single-pass decompilation, tools seem to hang on real policies
  - Even rebuilding from source is hard (e.g., actual compilation happens during RPM install, using tons of spec macros)