CS212 – Operating Systems

Instructor: David Mazières

CAs: Nirvik Baruah, Nathan Bhak, Dominic DeMarco, Gordon Martinez-Piedra, Suresh Nambi, Jaeyong Park, and Sunny Yu

Stanford University

Outline

1. Administrivia
2. Substance

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 of most CS212 students**
- **Exceptions**
  - You are an SCPD student (welcome to attend but not required)
  - Lecture conflicts with another class for which attendance required
  - Occasional one-of conflicts (travel, COVID, sports competitions)
- **Don't just watch the videos if you are an in-person student**
  - Especially don't save all the videos until the night before the exam
- **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`
  - Please use edstem for any questions others could conceivably have
- **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
  - Section: 6 Fridays, starting this Friday (time, location TBD)
  - Midterm: Monday, February 12, in class (1:30pm-2:50pm)
  - Final: Wednesday, March 20, 3:30pm-6:30pm
  - In-person attendance required for midterm and final (except SCPD)
  - SCPD can use exam monitor, return within 24 hours of exam start
- **Exams open note, but not open book**
  - Bring notes, slides, any printed materials *except* textbook

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 distributed at end of this week**
  - 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 instructor ASAP if you run into real problems
- **Final grades posted March 26**
- **50% of CS212 grade based on exams using this quantity:**

  $\max\left(\textbf{midterm} > 0\ ?\ \textbf{final} : 0, \frac{1}{2}(\textbf{midterm} + \textbf{final})\right)$
- **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 30 minutes before section Fridays**
  - Free extension to 5pm if you attend/watch section
- **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?

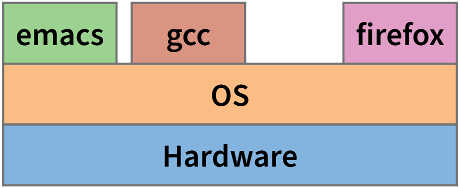
Outline

1. Administrivia

2. Substance

What is an operating system?

- **Layer between applications and hardware**



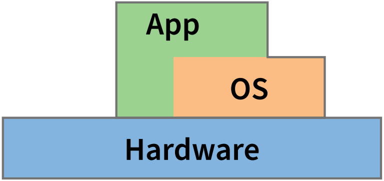
- **Makes hardware useful to the programmer**
- **[Usually] Provides abstractions for applications**
  - Manages and hides details of hardware
  - Accesses hardware through low/level interfaces unavailable to applications
- **[Often] Provides protection**
  - Prevents one process/user from clobbering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

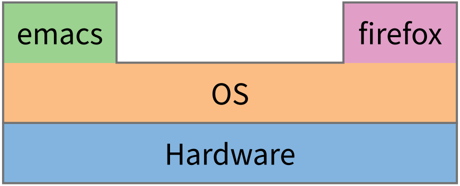
Primitive Operating Systems

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



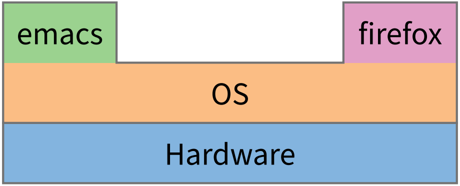
  - Standard interface above hardware-specific drivers, etc.
- **Simplifying assumptions**
  - System runs one program at a time
  - No bad users or programs (often bad assumption)
- **Problem: Poor utilization**
  - …of hardware (e.g., CPU idle while waiting for disk)
  - …of human user (must wait for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

Multitasking



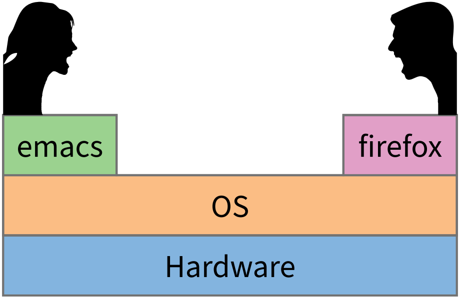
- **Idea: More than one process can be running at once**
  - When one process blocks (waiting for disk, network, user input, etc.) run another process
- **Problem: What can ill-behaved process do?**

Multitasking



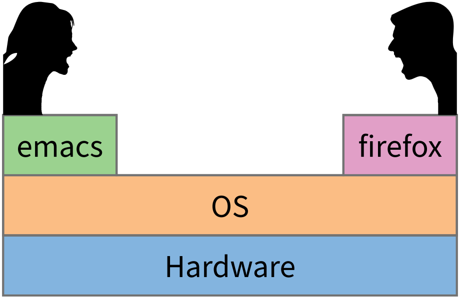
- **Idea: More than one process can be running at once**
  - When one process blocks (waiting for disk, network, user input, etc.) run another process
- **Problem: What can ill-behaved process do?**
  - Go into infinite loop and never relinquish CPU
  - Scribble over other processes' memory to make them 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

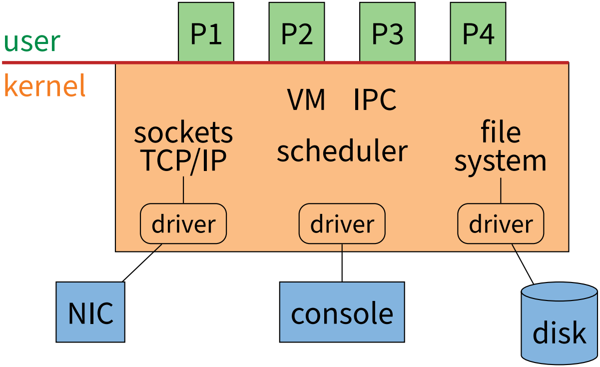


- **Many OSes use *protection* to serve distrustful users/apps**
- **Idea: With *N* users, system not *N* times slower**
  - Users' demands for CPU, memory, etc. are bursty
  - Win by giving resources to users who actually need them
- **What can go wrong?**
  - Users are gluttons, use too much CPU, etc.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

Protection

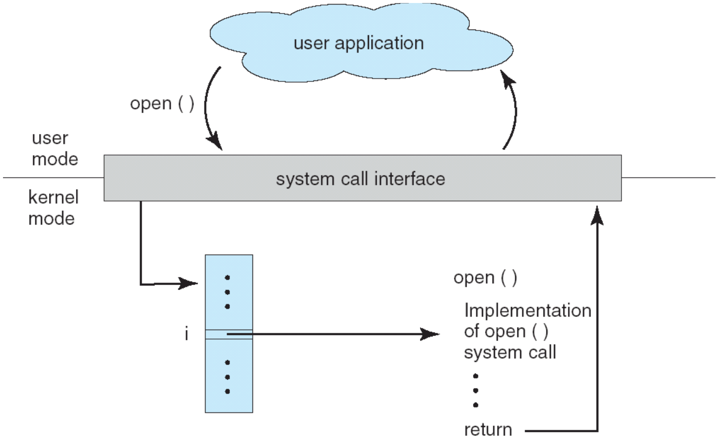
- **Mechanisms that isolate bad programs and people**
- **Pre-emption:**
  - Give application a resource, take it away if needed elsewhere
- **Interposition/mediation:**
  - Place OS between application and “stuff”
  - Track all pieces that application allowed to use (e.g., in table)
  - On every access, 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

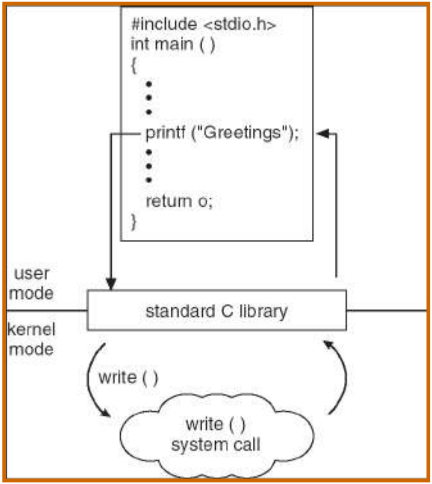


- **Applications can invoke kernel through *system calls***
  - Special instruction transfers control to kernel
  - ...which dispatches to one of few hundred syscall handlers

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");`
    $\rightarrow$ "`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 $\rightarrow$ kernel process context: syscall, page fault, ...
- User/process context $\rightarrow$ interrupt handler: hardware
- Process context $\rightarrow$ user/context switch: return
- Process context $\rightarrow$ context switch: sleep
- Context switch $\rightarrow$ 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 $\approx$ 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

Administrivia

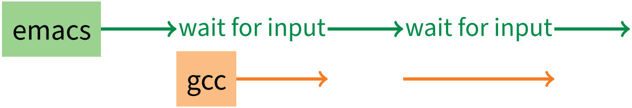
- **Friday 3:20pm section in Skilling (different zoom link, same password)**
  - Please attend first section this Friday to learn about project 1
- **Project 1 due Friday, Jan 26 at 3pm**
  - 5pm if you attend/watch lecture
- **Ask `cs212-staff` for extension if you can't finish**
  - Tell us where you 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**
- **Project groups should ideally be 2–3 people**
  - Solo groups allowed but not recommended

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

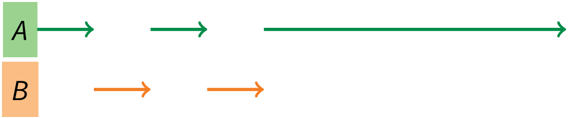
Speed

- **Multiple processes can increase CPU utilization**
  - 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

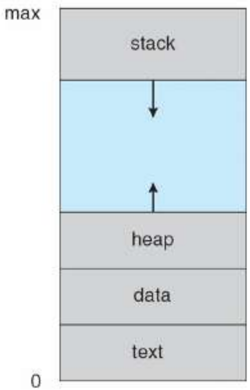


  - *A* is slower than if it had whole machine to itself,
    but still $<$ 100 sec unless both *A* and *B* completely CPU-bound

Processes in the real world

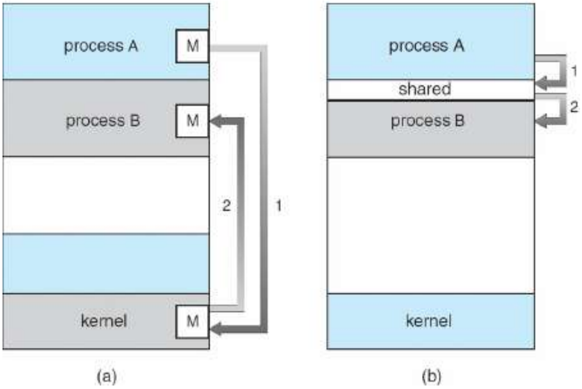
- **Processes and parallelism have been a fact of life much longer than OSes have been around**
  - E.g., say takes 1 worker 10 months to make 1 widget
  - Company may hire 100 workers to make 100 widgets
  - Latency for first widget $\gg 1/10$ month
  - Throughput may be $<$ 10 widgets per month (if 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`
  - `envp` – environment variables, e.g., `PATH`, `HOME`
- **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)

```c
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 ...`

```
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 setting/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

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

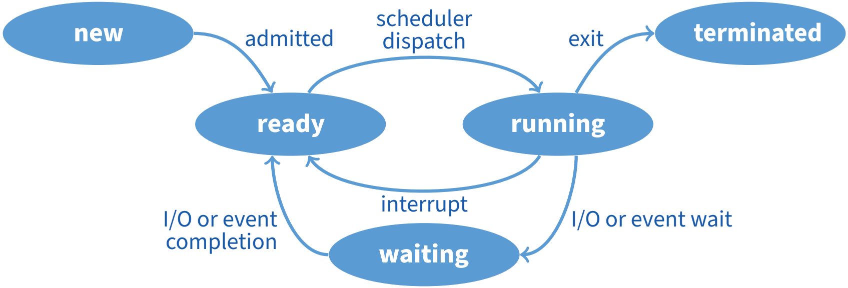
Implementing processes

- **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

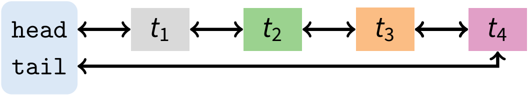
Process states



- **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

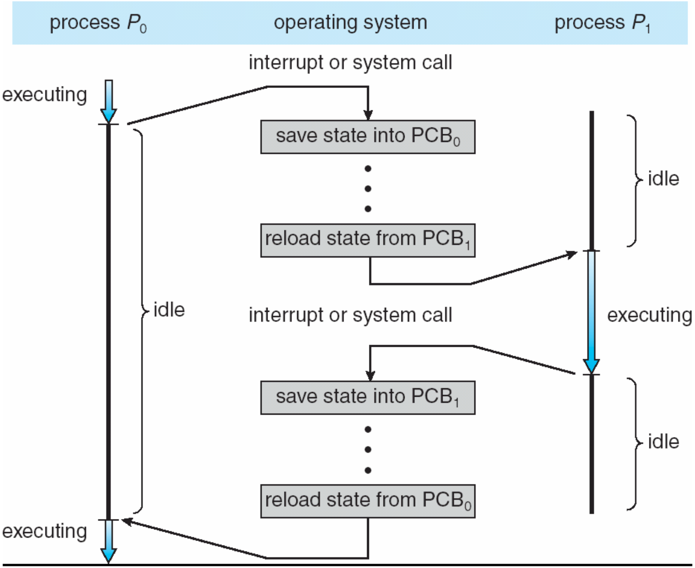
Scheduling policy

- **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



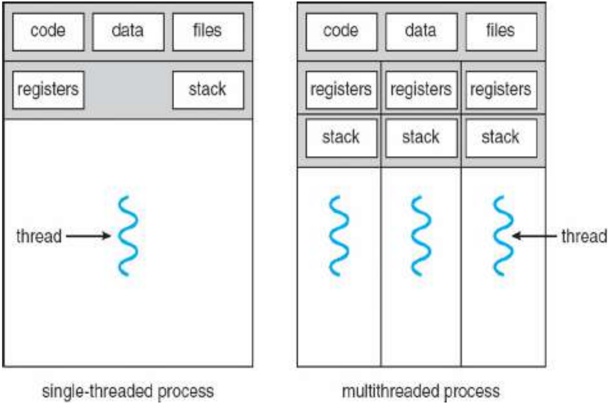
Context switch details

- **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

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

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:

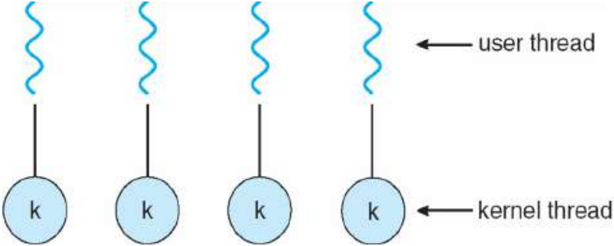
```
for (;;) {
  c = accept_client();
  thread_create(service_client, c);
}
```

- **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]



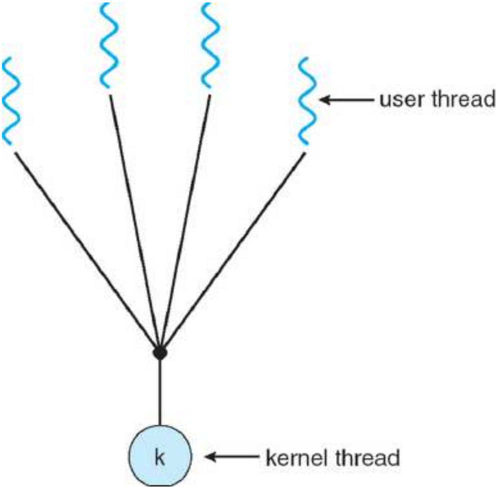
- **Can implement** `thread_create` **as a system call**
- **To add** `thread_create` **to an OS that doesn't have it:**
  - Start with process abstraction in kernel
  - `thread_create` like process creation with features stripped out
    - Keep same address space, file table, etc., in new process
    - `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

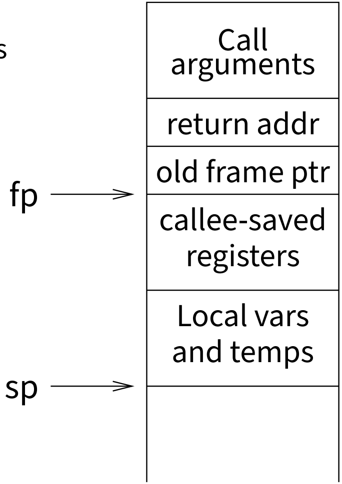
- **Allocate a new stack for each** `thread_create`
- **Keep a queue of runnable threads**
- **Replace networking system calls (**`read`/`write`/**etc.)**
  - If operation would block, switch and run different thread
- **Schedule periodic timer signal (**`setitimer`**)**
  - Switch to another thread 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

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

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**

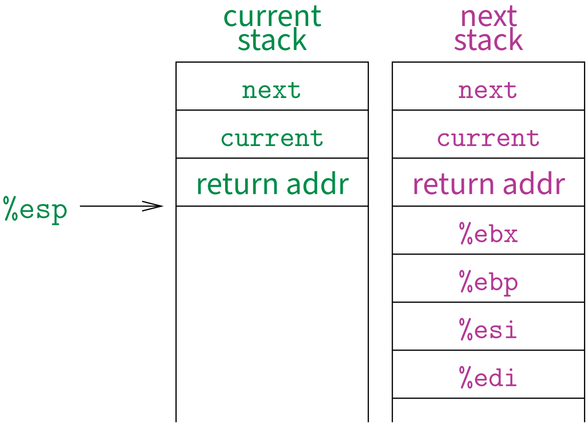
```
struct thread {
    ...
    uint8_t *stack;  /* Saved stack pointer. */
    ...
};
uint32_t thread_stack_ofs = offsetof(struct thread, stack);
```

- **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

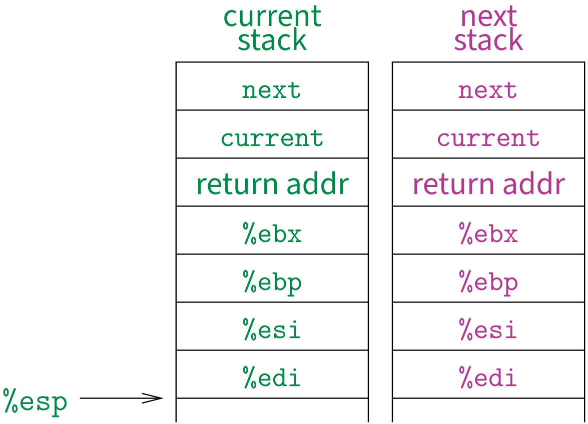
```
pushl %ebx; pushl %ebp          # Save callee-saved regs
pushl %esi; pushl %edi

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



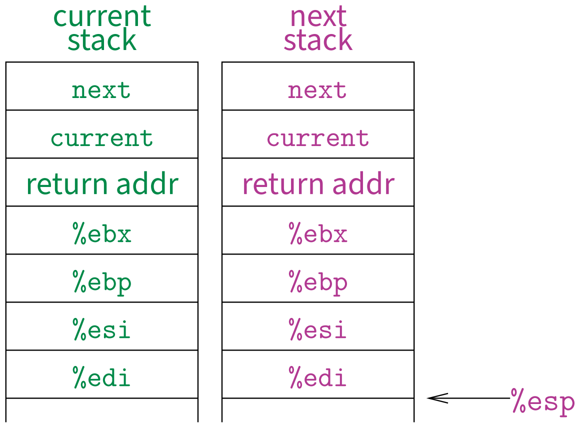
- **This is actual code from Pintos `switch.S` (slightly reformatted)**
  - See Thread Switching in documentation

i386 `switch_threads`



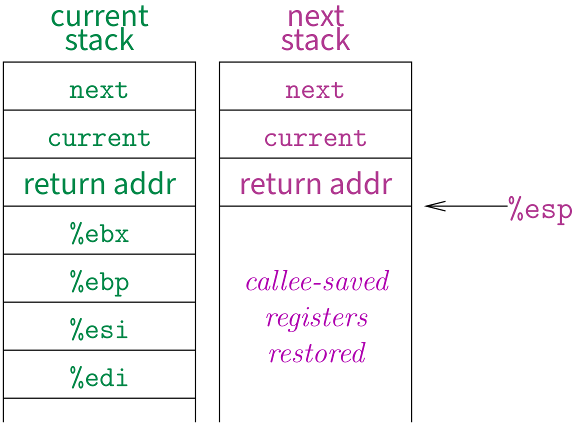
- **This is actual code from Pintos `switch.S` (slightly reformatted)**
  - See Thread Switching in documentation

i386 switch_threads



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

i386 `switch_threads`

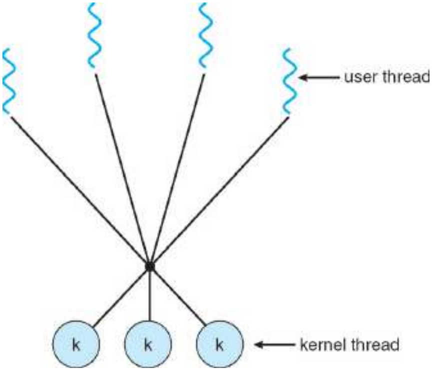


- **This is actual code from Pintos `switch.S` (slightly reformatted)**
  - See Thread Switching in documentation

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…

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

1. Memory consistency
2. The critical section problem
3. Mutexes and condition variables
4. Implementing synchronization
5. Alternate synchronization abstractions

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

1. Memory consistency
2. The critical section problem
3. Mutexes and condition variables
4. Implementing synchronization
5. Alternate synchronization abstractions

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--`

`count++`	`count--`
register $\leftarrow$ count	register $\leftarrow$ count
register $\leftarrow$ register $+ 1$	register $\leftarrow$ register $- 1$
count $\leftarrow$ register	count $\leftarrow$ register

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

register $\leftarrow$ count	
register $\leftarrow$ register $+ 1$	
	register $\leftarrow$ count
	register $\leftarrow$ register $- 1$
count $\leftarrow$ register	
	count $\leftarrow$ 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

1. Memory consistency
2. The critical section problem
3. Mutexes and condition variables
4. Implementing synchronization
5. Alternate synchronization abstractions

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	**Consumer**
`while (count == BUFFER_SIZE)`	
`  mutex_unlock (&mutex);`	
	`mutex_lock (&mutex);`
	`...`
	`count--;`
	`cond_signal (&nonfull);`
`  cond_wait (&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

1. Memory consistency
2. The critical section problem
3. Mutexes and condition variables
4. Implementing synchronization
5. Alternate synchronization abstractions

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

1. Memory consistency
2. The critical section problem
3. Mutexes and condition variables
4. Implementing synchronization
5. Alternate synchronization abstractions

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

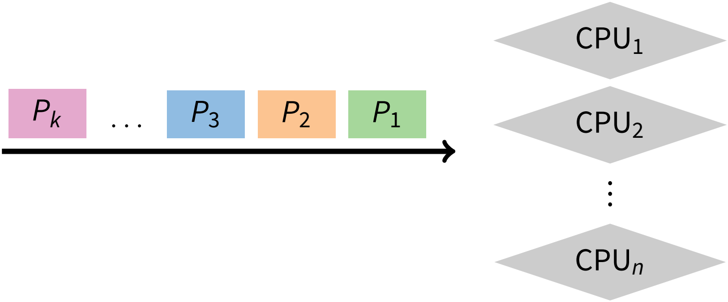
- **Initialize `full` to 0 (block consumer when buffer empty)**
- **Initialize `empty` to *N* (block producer when queue full)**

```
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

CPU scheduling

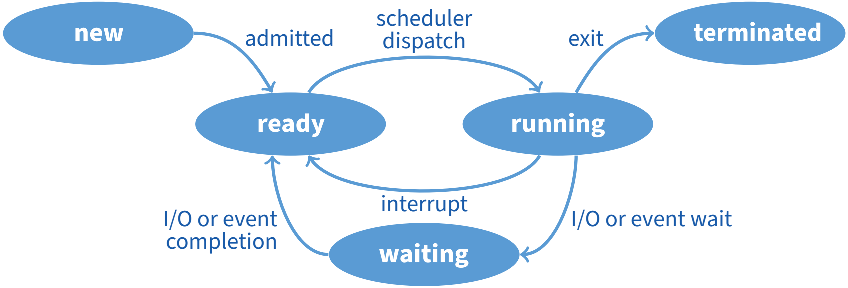


- **The scheduling problem:**
  - Have k jobs ready to run
  - Have $n \geq 1$ CPUs that can run them
- **Which jobs should we assign to which CPU(s)?**

Outline

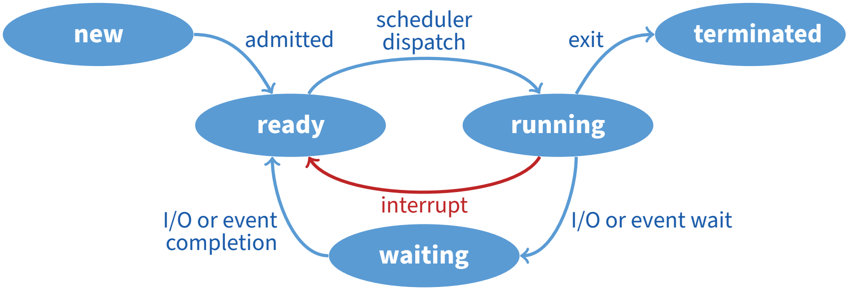
1. Textbook scheduling
2. Priority scheduling
3. Advanced scheduling issues
4. Virtual time case studies

When do we schedule CPU?



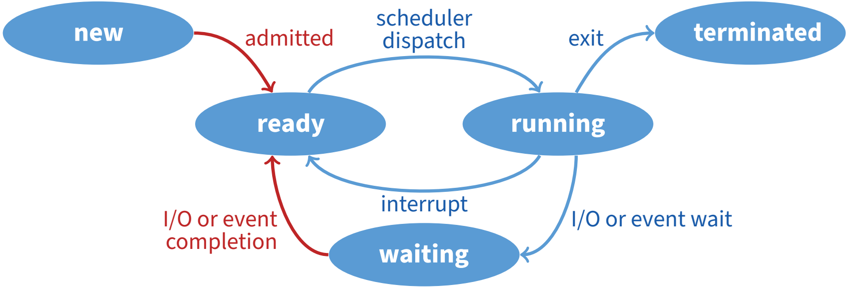
- **Scheduling decisions may take place when a process:**
  1. Switches from running to ready state
  2. Switches from new/waiting to ready
  3. Switches from running to waiting state
  4. Exits
- **Non-preemptive schedules use 3 & 4 only**
- **Preemptive schedulers run at all four points**

When do we schedule CPU?



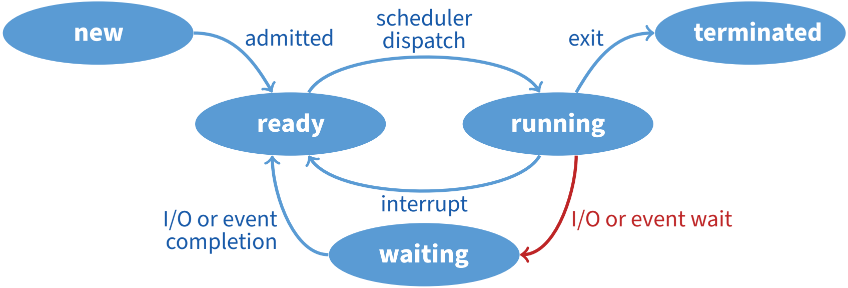
- **Scheduling decisions may take place when a process:**
  1. Switches from running to ready state
  2. Switches from new/waiting to ready
  3. Switches from running to waiting state
  4. Exits
- **Non-preemptive schedules use 3 & 4 only**
- **Preemptive schedulers run at all four points**

When do we schedule CPU?



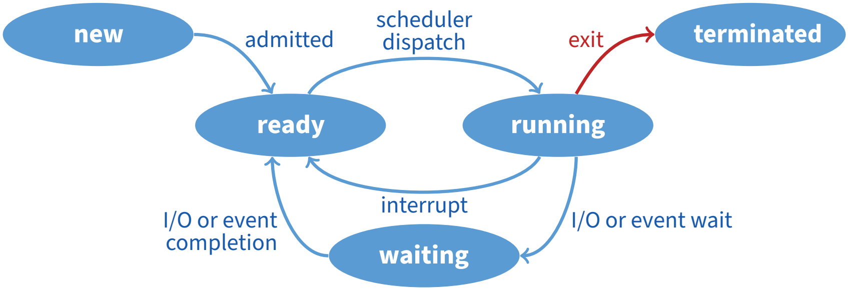
- **Scheduling decisions may take place when a process:**
  1. Switches from running to ready state
  2. Switches from new/waiting to ready
  3. Switches from running to waiting state
  4. Exits
- **Non-preemptive schedules use 3 & 4 only**
- **Preemptive schedulers run at all four points**

When do we schedule CPU?



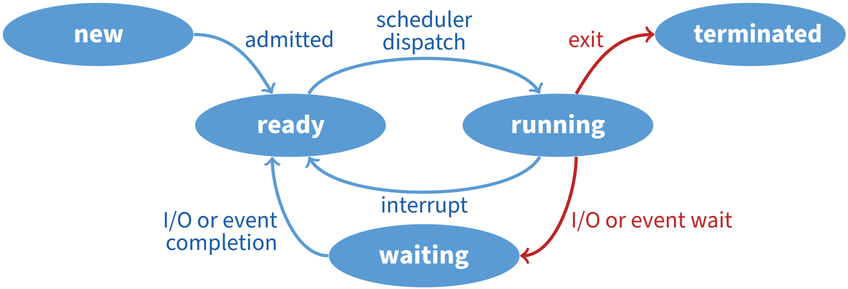
- **Scheduling decisions may take place when a process:**
  1. Switches from running to ready state
  2. Switches from new/waiting to ready
  3. Switches from running to waiting state
  4. Exits
- **Non-preemptive schedules use 3 & 4 only**
- **Preemptive schedulers run at all four points**

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 schedules use 3 & 4 only**
- **Preemptive schedulers run at all four points**

When do we schedule CPU?

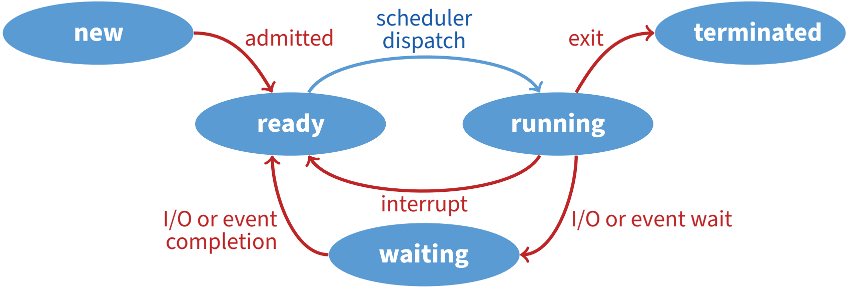


- **Scheduling decisions may take place when a process:**
  1. Switches from running to ready state
  2. Switches from new/waiting to ready
  3. Switches from running to waiting state
  4. Exits

⟶ **Non-preemptive schedules 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 schedules 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?**

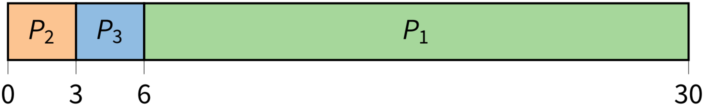
FCFS continued

- **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

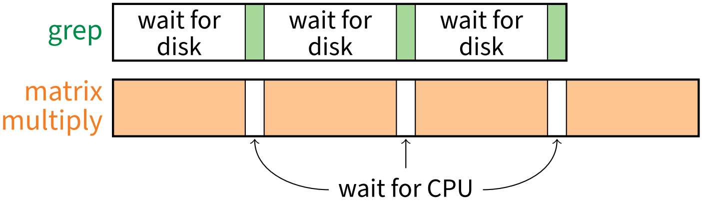
- **Suppose we scheduled P_2, P_3, then P_1**
  - Would get:



- **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

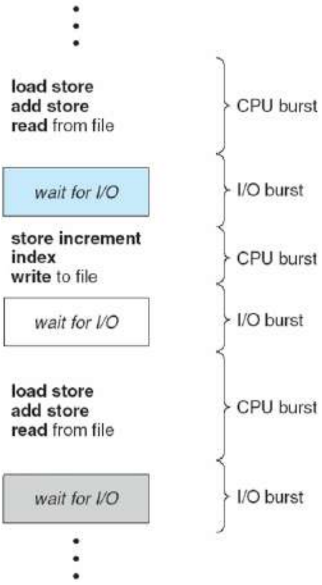
View CPU and I/O devices the same

- **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

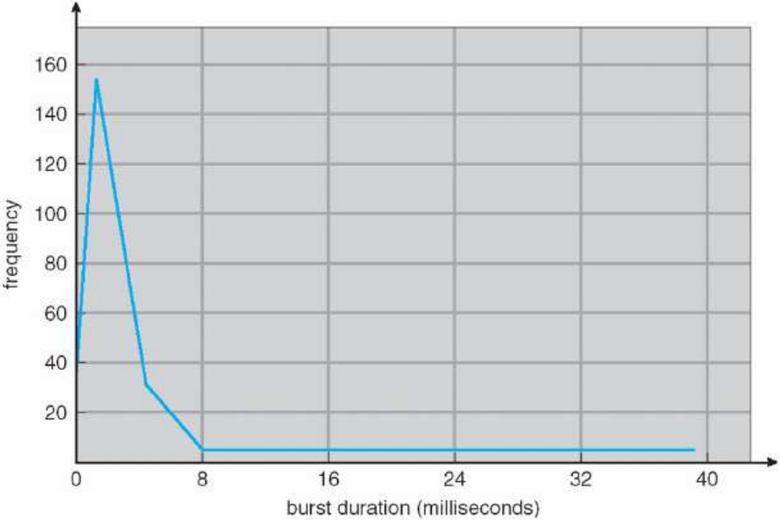


Bursts of computation & I/O

- **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?**

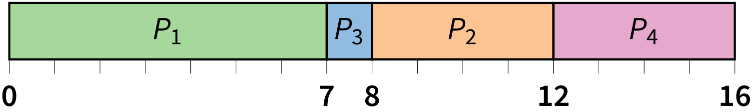
SJF Scheduling

- ***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

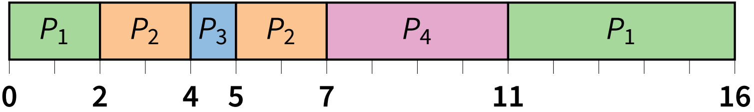
Examples

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?

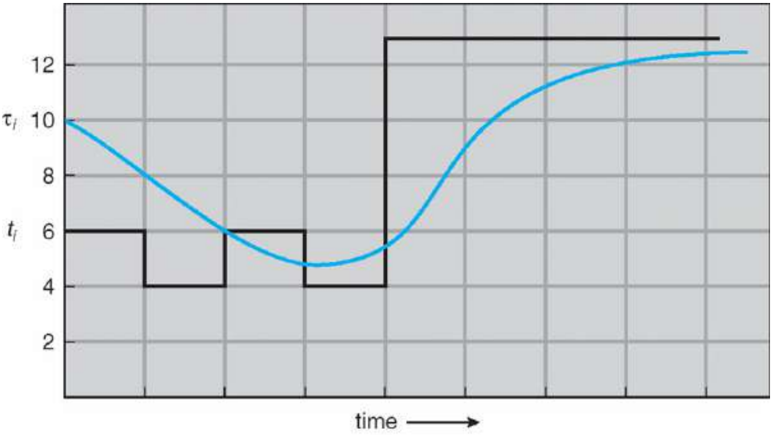
SJF limitations

- **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



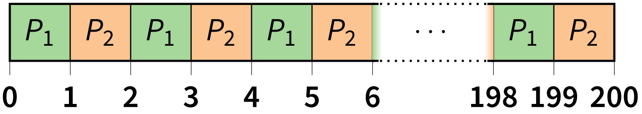
CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

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
  - (Most 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

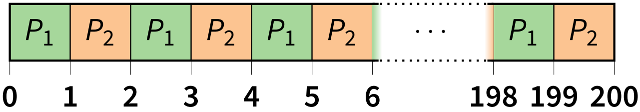
- **Varying sized jobs are good . . . what about same-sized jobs?**
- **Assume 2 jobs of time=100 each:**



- **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:**



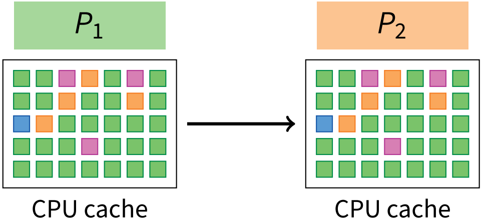
- **Even if context switches were free. . .**
  - What would average turnaround time be with RR? *199.5*
  - How does that compare to FCFS? *150*

Context switch costs

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

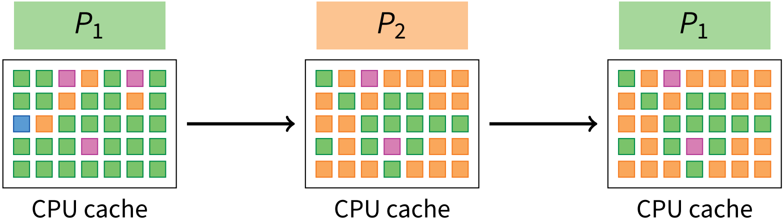
Context switch costs

- **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**

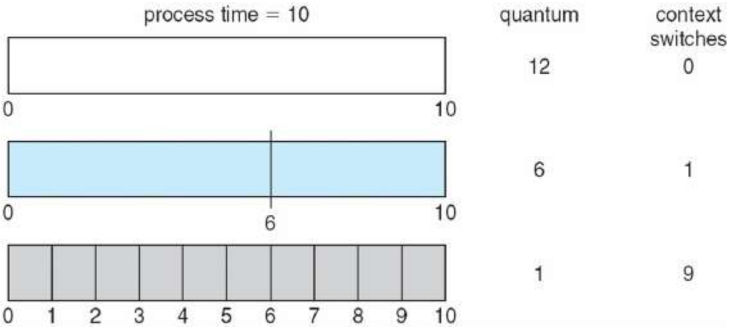


Context switch costs

- **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**

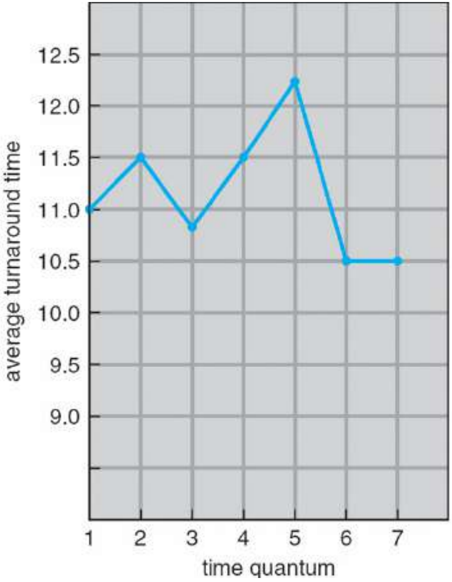


Time quantum



- **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 $\sim$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

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

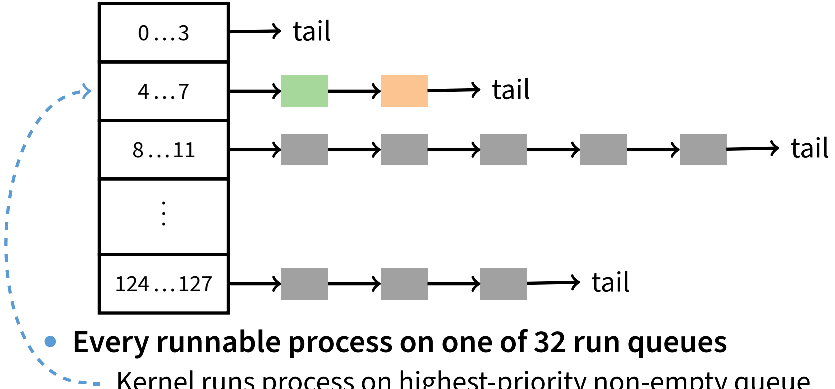
Priority scheduling

- **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**

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

  - 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

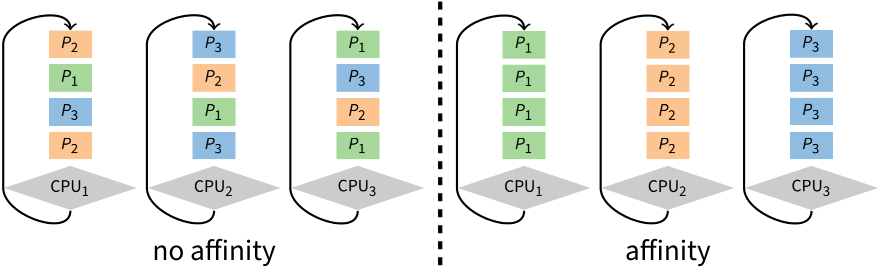
- **Say higher number = higher priority (like Pintos)**
- **Example 1: L (prio 2), M (prio 4), H (prio 8)**
  - L holds lock ℓ
  - M waits on ℓ, L's priority raised to $L_1 = \max(M, L) = 4$
  - Then H waits on ℓ, L's priority raised to $\max(H, L_1) = 8$
- **Example 2: Same L, M, H as above**
  - L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

Outline

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

Multiprocessor scheduling issues

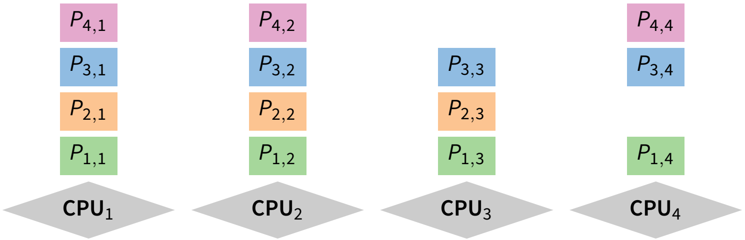
- **Must decide on more than which processes to run**
  - Must decide on which CPU to run which process
- **Moving between CPUs has costs**
  - More cache misses, depending on arch. more TLB misses too
- ***Affinity scheduling*—try to keep process/thread on same CPU**



  - 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

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

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* $E_i = A_i - (\mathrm{warp}_i \; ? \; W_i : 0)$
  - Special warp factor allows borrowing against future CPU time …hence name of algorithm

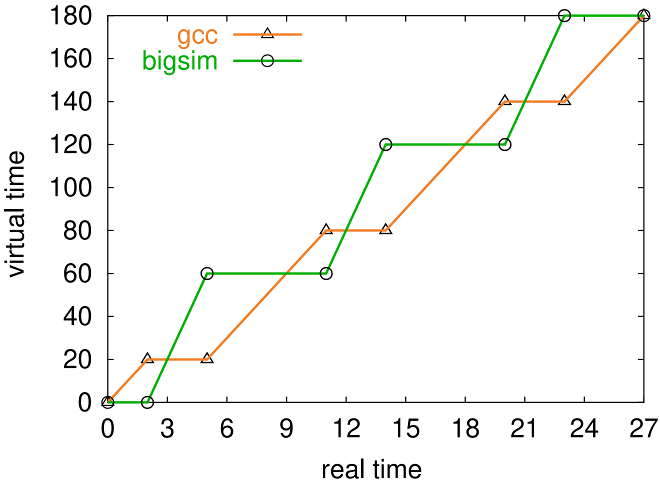
Process weights

- **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

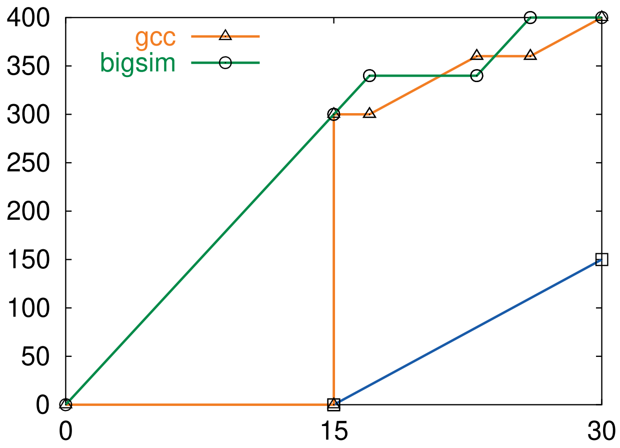


- **gcc has weight 2, bigsim weight 1, $C = 2$, no I/O**
  - bigsim consumes virtual time at twice the rate of gcc
  - Processes run for C time after lines cross before context switch

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

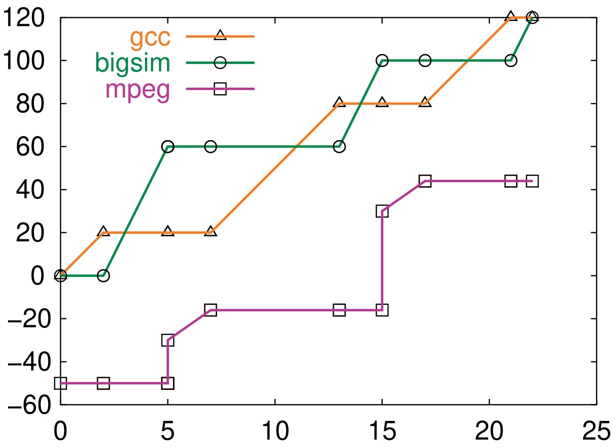


- **gcc's A_i gets reset to SVT on wakeup**
  - Otherwise, would be at lower (blue) line and starve bigsim

Real-time threads

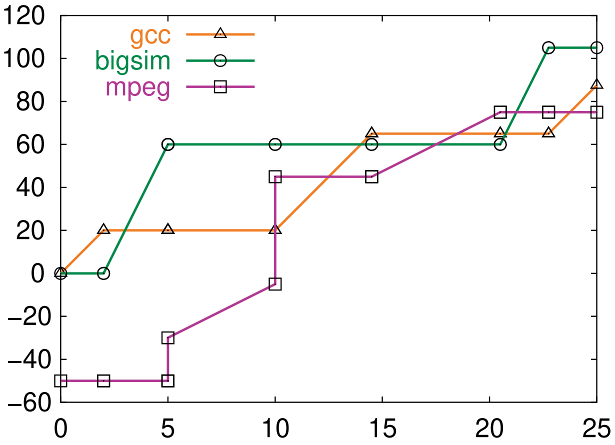
- **Also want to support time-critical tasks**
  - E.g., mpeg player must run every 10 clock ticks
- **Recall** $E_i = A_i - (\textbf{warp}_i \mathbin{?} W_i : 0)$
  - W_i is *warp factor* – gives thread precedence
  - Just give mpeg player i large W_i factor
  - Will get CPU whenever it is runable
  - 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 =** $\langle \mathit{priority}, \mathit{BVFT} \rangle$ **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**

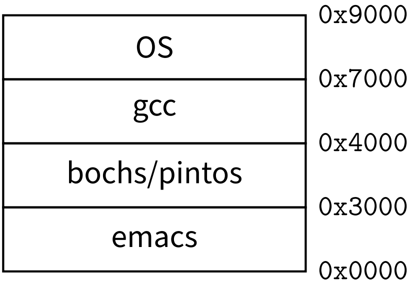
Administrivia

- **Lab 1 due Friday 3pm (5pm if you attend section)**
- **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 3:20pm 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

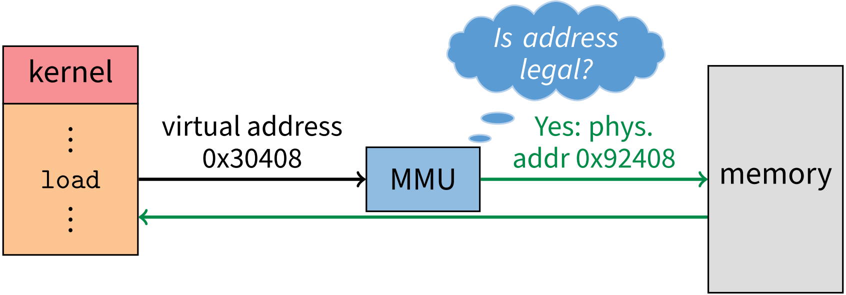


- **Consider multiprogramming on physical memory**
  - What happens if pintos needs to expand?
  - If emacs needs more memory than is on the machine?
  - If pintos has an error and writes to address 0x7100?
  - When does gcc have to know it will run at 0x4000?
  - What if emacs isn't using its memory?

Issues in sharing physical memory

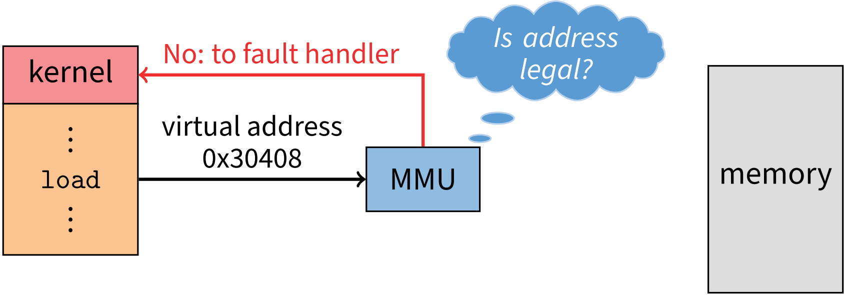
- **Protection**
  - A bug in one process can corrupt memory in another
  - Must somehow prevent process *A* from trashing *B*'s memory
  - Also prevent *A* from even observing *B*'s memory (ssh-agent)
- **Transparency**
  - A process shouldn't require particular physical memory bits
  - Yet processes 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

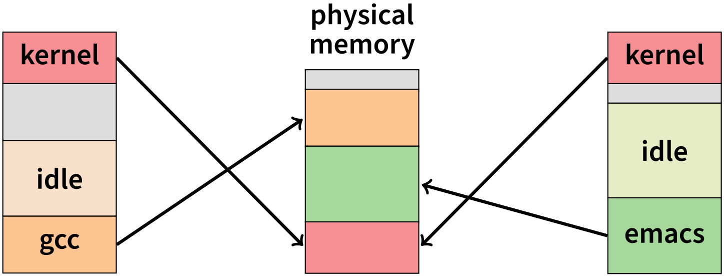
Virtual memory goals



- **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

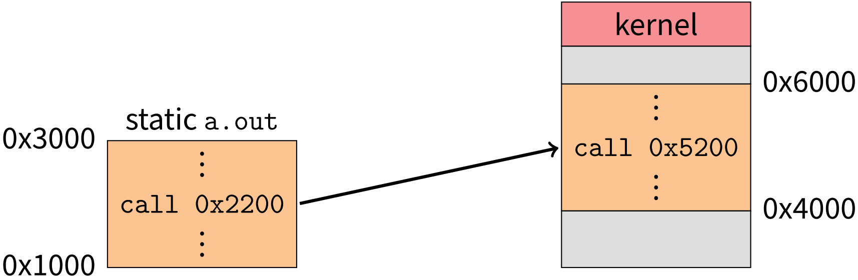
Virtual memory advantages

- **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).**



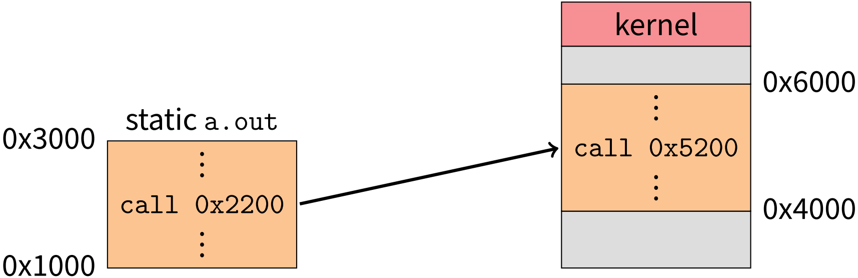
  - Write idle parts to disk until needed
  - Let other processes use memory of idle part
  - Like CPU virtualization: when process not using CPU, switch (Not using a memory region? switch it to another process)
- **Challenge: VM = extra layer, could be slow**

Idea 1: no hardware, load-time linking



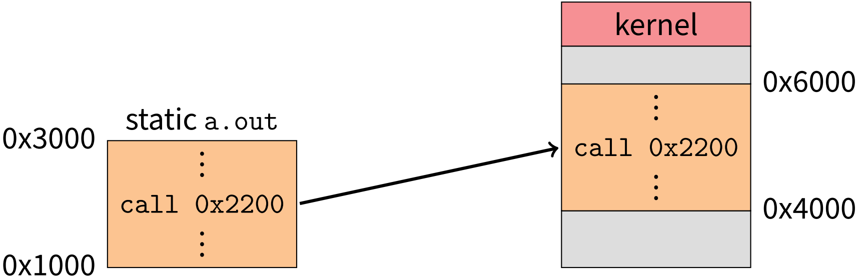
- *Linker* **patches addresses of symbols like** `printf`
- **Idea: link when process executed, not at compile time**
  - Already have PIE (position-independent executable) for security
  - Determine where process will reside in memory at launch
  - Adjust all references within program (using addition)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

Idea 1: no hardware, load-time linking



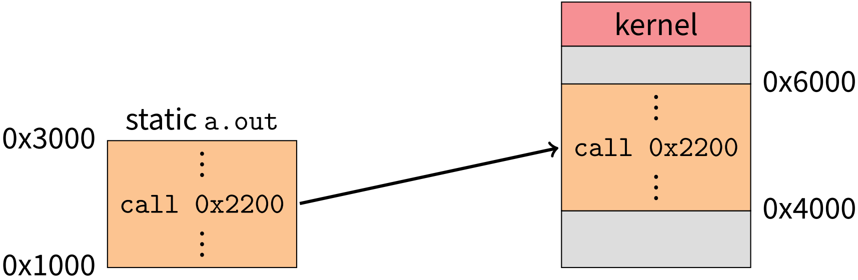
- ***Linker* patches addresses of symbols like `printf`**
- **Idea: link when process executed, not at compile time**
  - Already have PIE (position-independent executable) for security
  - Determine where process will reside in memory at launch
  - Adjust all references within program (using addition)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



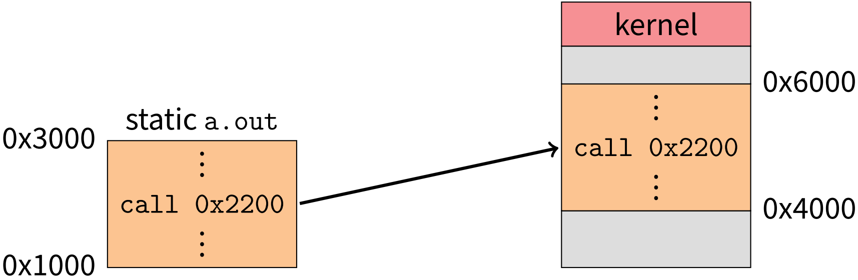
- **Two special privileged registers: base and bound**
- **On each load/store/jump:**
  - Physical address = virtual address + base
  - Check $0 \leq$ virtual address $<$ bound, else trap to kernel
- **How to move process in memory?**
- **What happens on context switch?**

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?**

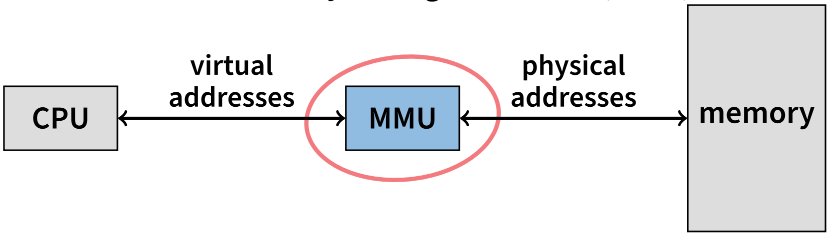
Idea 2: base + bound register



- **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

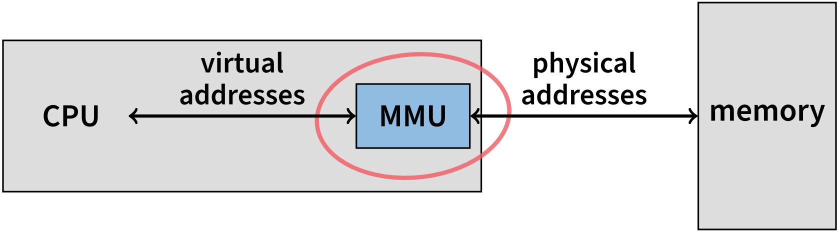
- Programs load/store to virtual addresses
- Actual memory uses physical addresses
- VM Hardware is Memory Management Unit (MMU)



  - 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)



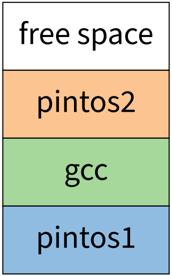
  - Usually part of CPU core (one address space per hyperthread)
  - Configured through privileged instructions (e.g., load bound reg)
  - Translates from virtual to physical addresses
  - Gives per-process view of memory called address space

Base+bound trade-offs

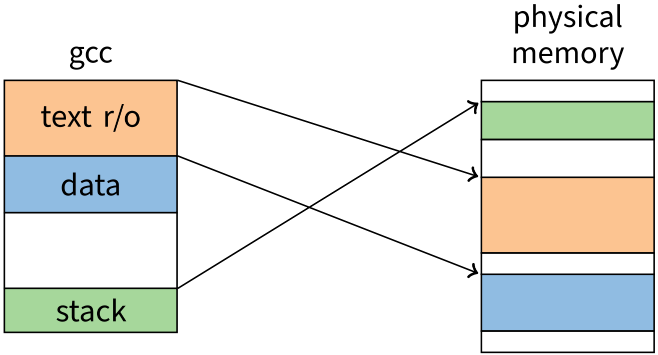
- **Advantages**
  - Cheap in terms of hardware: only two registers
  - Cheap in terms of cycles: do add and compare in parallel
  - Examples: Cray-1 used this scheme
- **Disadvantages**

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

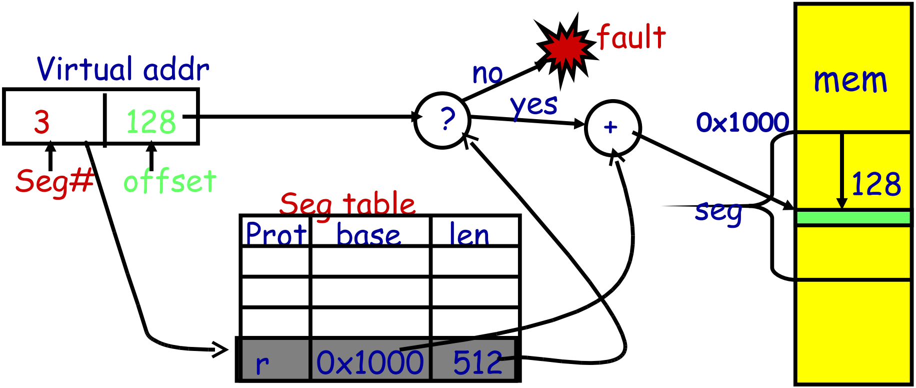


Segmentation



- **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**

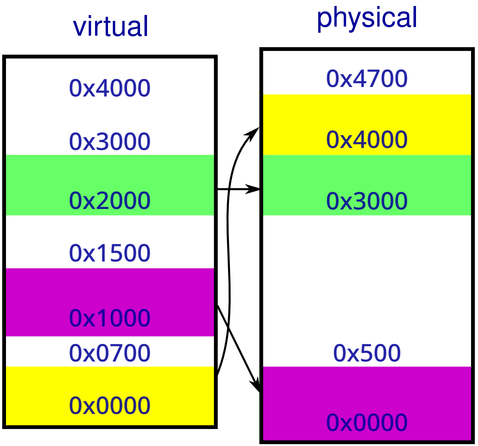
Segmentation mechanics



- **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)

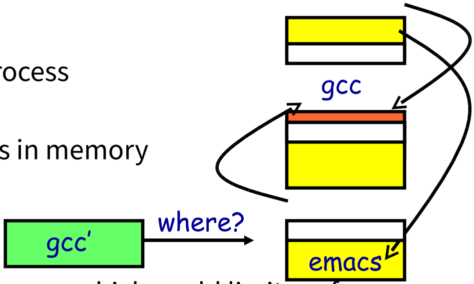
Segmentation example

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?

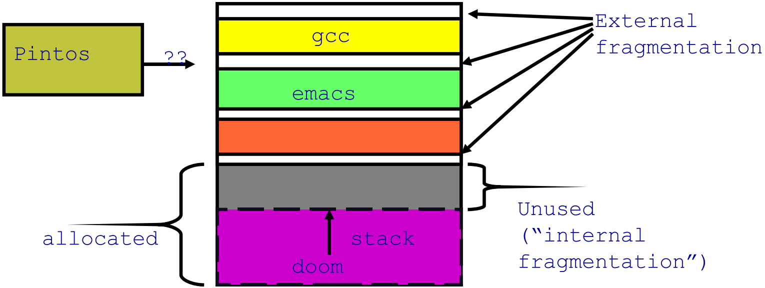
Segmentation trade-offs



- **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)



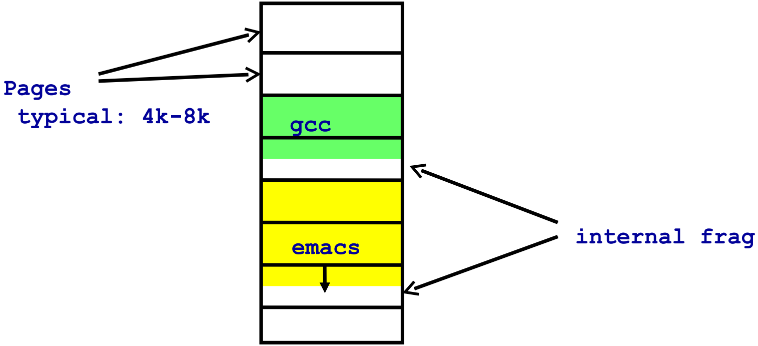
Alternatives to hardware MMU

- **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

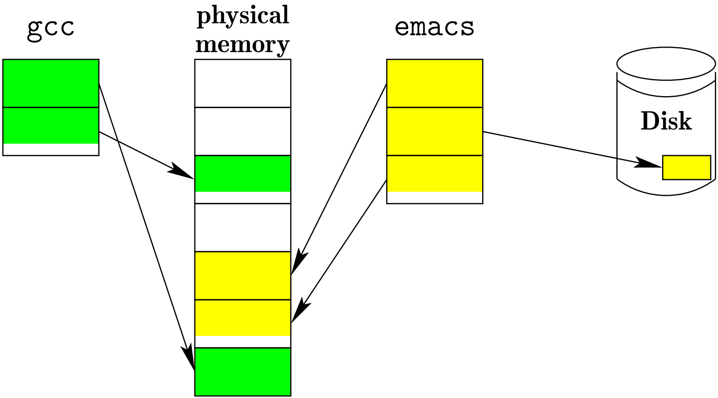
- **Divide memory up into small, equal-size *pages***
- **Map virtual pages to physical pages**
  - Each process has separate mapping
- **Allow OS to gain control on certain operations**
  - Read-only pages trap to OS on write
  - Invalid pages trap to OS on read or write
  - OS can change mapping and 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”

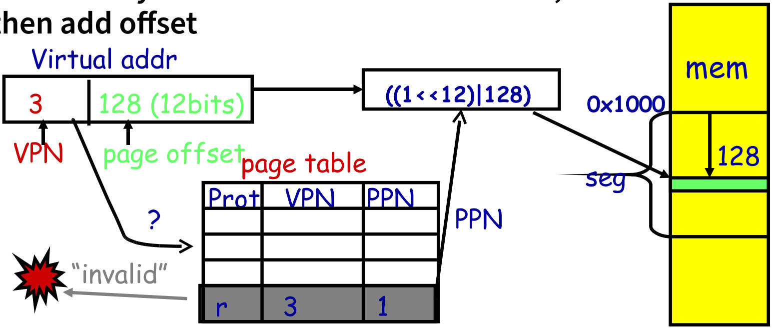
Simplified allocation



- 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**



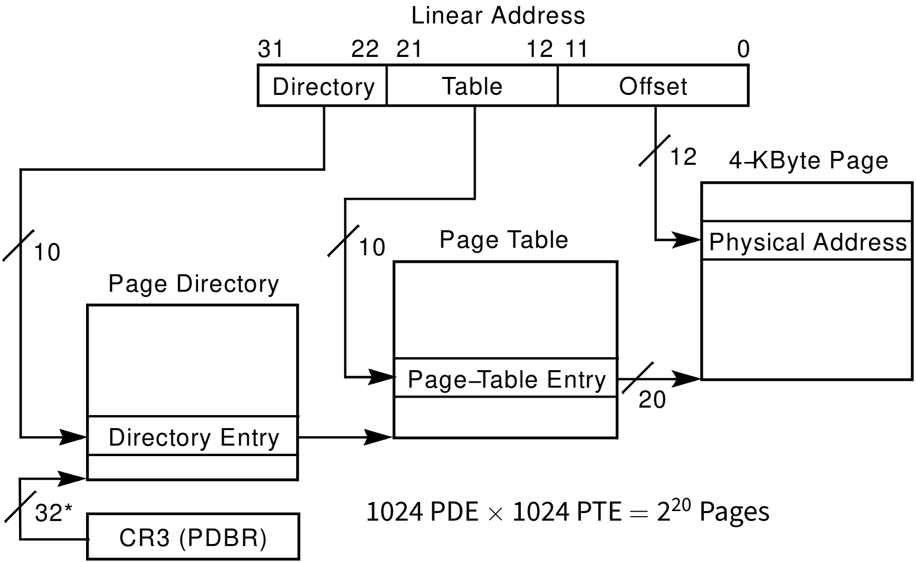
Example: Paging on PDP-11

- **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

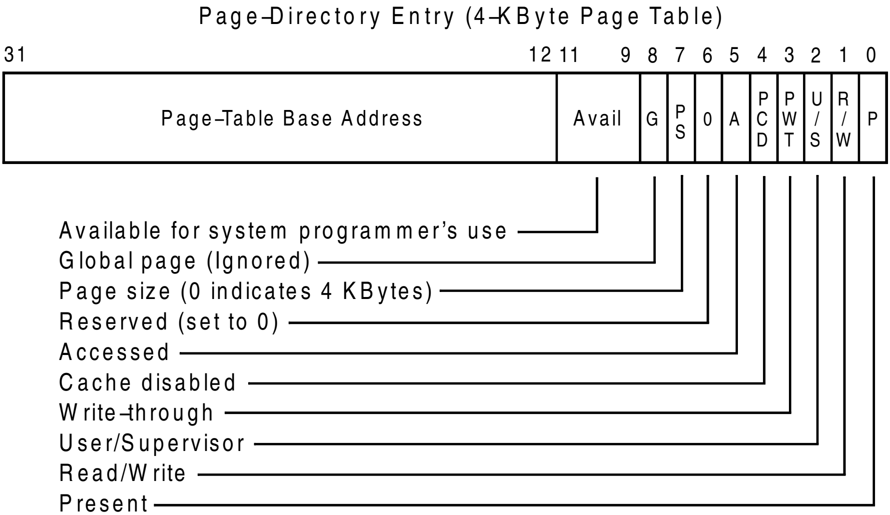
x86 page translation



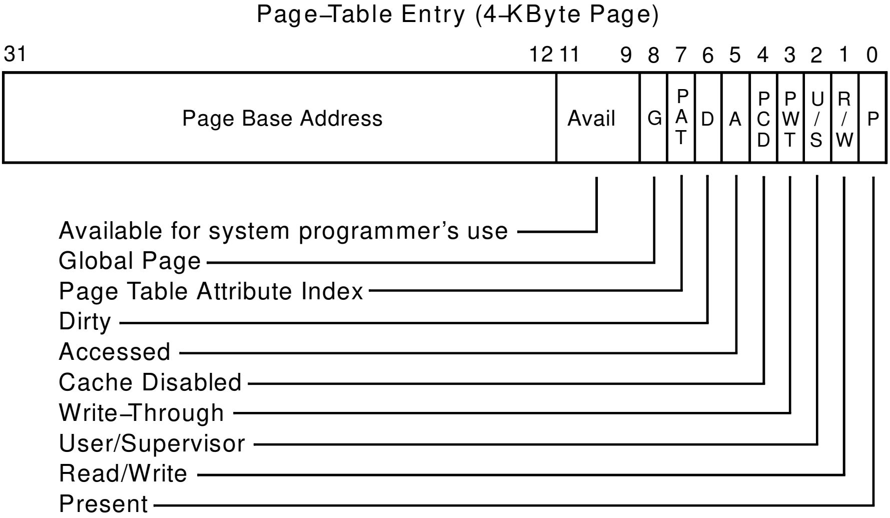
1024 PDE $\times$ 1024 PTE $=$ 2^{20} Pages

*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)

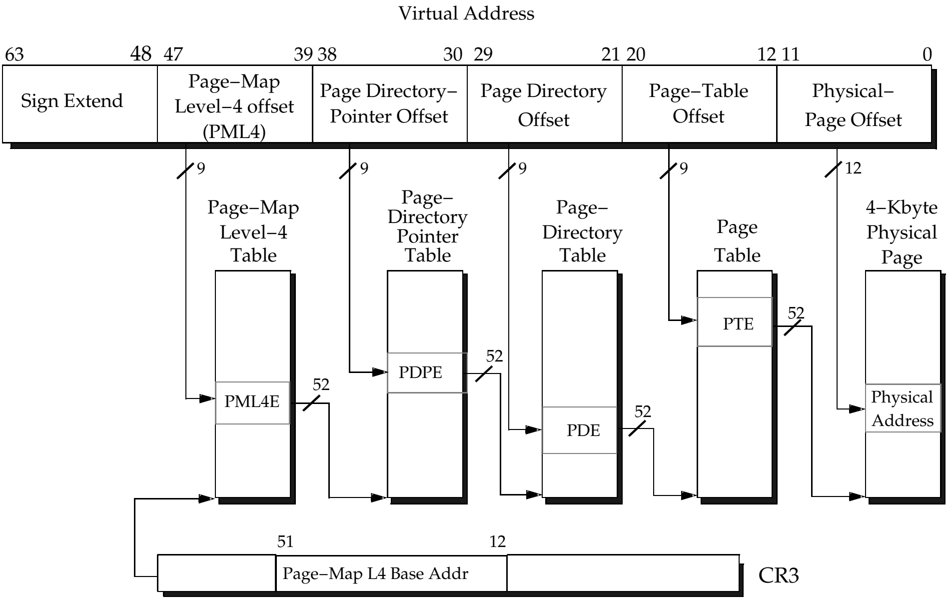
TLB details

- **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?

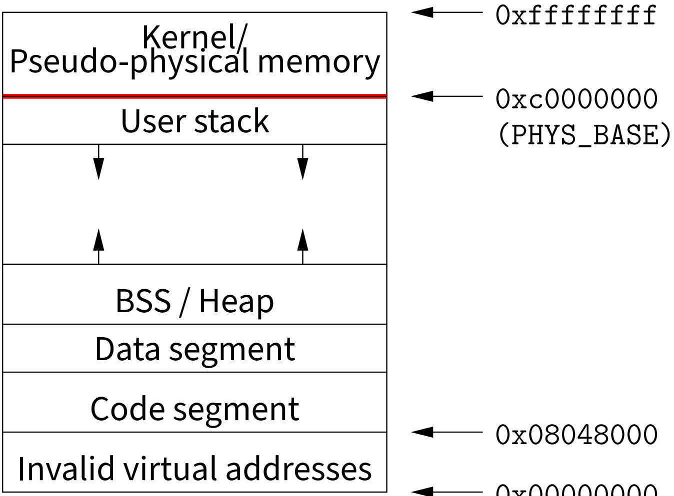
x86 long mode paging



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?