

# CS350: Operating Systems

University of Waterloo

Fall 2023

**Instructors:** Ali Mashtizadeh and Kevin Lanctot

These lecture notes are based on the lecture notes of  
Bernard Wong, Lesley Istead and Ali Mashtizadeh.

# Useful Information About Your Instructor

- Name: Kevin Lanctot
- Office: DC 3106 (in the hallway nearest the skywalk to the MC/M3 buildings).
- Office Hours: Thursdays 1:30-3:30 pm starting next week.
- Email: kevin.lanctot@uwaterloo.ca
- My last name is pronounced *long-k toe*, i.e. “long toe” with the *k* sound after *long*.
- *I'm a talker not a typer*, i.e. it is best to ask me questions about course content in person rather than through email or Piazza.
- I only *check my email a few times a day*.

# Sources of Information about the Course

Important links:

- <http://www.student.cs.uwaterloo.ca/~cs350>

Course personnel, office hours, assignments, tutorials, previous midterms, review problems, course policies etc.

- <https://piazza.com/uwaterloo.ca/fall2023/cs350/home>

Piazza will be used for announcements, extra notes, questions, corrections, etc.

- Please check Piazza regularly.
- Read the pinned posts which are the official announcements for the course.
- Use public or anonymous posts for questions about course material.
- Piazza is good for quick clarification. *For more detailed explanations* (e.g. explain slide 350 to me), [\*go to an office hour\*](#).
- Do not post your code or solution ideas in public piazza posts; use private posts when discussing your assignment solutions.

# Course Material

## Course Notes / Slides

- Available on Learn by noon the day of the lecture.
- *They are not designed to be standalone document* (like a textbook) but merely an outline of what is covered in lecture, i.e. come to class and take notes.
- My version of the slides cover the same material in the same order as the “official” slides on the course website, however I add some additional examples and use different formatting.

## Textbook

- *Operating Systems: Three Easy Pieces* by R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau.
- The textbook is not required, but is highly recommended.
- It is available free (legally) online.
- We have links to all recommended readings from the book on our course website.

# Course Material

## Two Other Textbooks

- The course website gives references to the textbook that used to be used for this course, *Operating System Concepts*, Ninth Edition, by Abraham Silberschatz *et al.* There is now a tenth edition.
- Another textbook which I find particularly helpful is *Modern Operating Systems*, Fourth Edition, by Andrew S. Tanenbaum.
- Eventually, both of these textbooks will be available for 3-hour loan at the Davis Centre Library Circulation Desk. This book is more complete but also more expensive.

# Slide Format

- Slides use red bold letters for any technical terms that you are expected to know as part of this course, such as **threads**.
- Slides use *blue italics for key point (or points) on the slide.*

Additional material that will not be tested on the midterm or final exams are placed in a rounded rectangle.

## Course Deliverables and Assessments

- Two reading assignments. You will have a simple quiz on the material in Learn and likely a question or questions about them in the midterm and final.
- Four programming assignments.
- In-class quizzes using the website <https://www.vevox.com/> or the Vevox App (also called the Vevox Polling App) which you can download and use for free from Google Play or the (Apple) App Store. I will also make these quizzes available for students that are not in my sections through Learn. These are not worth any marks but will help keep you up to date with the material.
- A midterm (Wednesday, November 1, 7:00-08:50 pm) and a final exam scheduled by the Registrar's Office.

# Plagiarism and Academic Offenses

There are two types of assignments.

1. *Research Based:* With this type of assignment you are given a topic and you go out and research the answer and write it up, carefully citing your sources of information. These types of assignments are very common in English Literature and History courses.
2. *Individual Work Based:* With this type of assignment you are given the tools to answer the questions in the course material and you are expected to answer the questions on your own, as a way of understanding the material. These types of assignments are very common in Computer Science, Mathematics and Engineering courses.

CS 350 is the second type of course, you are expected to *do the assignment without researching the answer.*

# Plagiarism and Academic Offenses...

Read and understand UW's policy on academic integrity

<https://uwaterloo.ca/academic-integrity/integrity-students>

And see the following [tip sheet](#).

This course has extra requirements and ignorance is no excuse!

- *Do not copy or view code from friends, web sites, or other sources.*
- Do not search for or look at other code for any reason.
- Avoid blogs that provide instructions.
- We use very good cheat detection software.
- Every term people are caught.
- Typically: 0 on assignment and -5% off final grade.

## Plagiarism and Academic Offenses ...

Other than websites identified in the course, it is *acceptable to use the web* to

- understand the lecture material or how to use the programming tools we use in this course such as Git, make and GDB.

But it is *not acceptable to use the web* to

- get an idea of how to approach the assignment,
- copy or view code that may help you do the assignment.

It is *acceptable to consult with other students* to

- get an idea of how to approach the assignment,
- get an idea of how to overcome a stumbling block or fix a bug.

But it is *not acceptable* to

- view another student's code or have another student view your code.

# Plagiarism and Academic Offenses

If you have taken CS350 before, you may reuse your previous code if...

- you ask your instructor for permission,
- your code was not subject to previous cheating penalties, and
- you understand it will be re-tested using our cheat detection software.

# Course Goal: Introduce You to Systems

- Operating Systems
- Distributed Systems
- Networking
- Internet of Things
- Computer Architecture
- Embedded Systems
- Database Systems
- Systems and Machine Learning
- ...

# Course Goal: Practical Understanding of OSes

- Introduce you to operating systems
  - Every computer, smart phone and smart watch runs an OS.
  - Understanding how an OS works, makes you a more effective programmer.
  - In particular, you should understand how the OS affects your software.
  - We build on the concepts you learned in CS241 and CS251.
- General systems concepts
  - Concurrency, memory management, and I/O
  - Security and protection
  - Tools for software performance.
- Practical skills
  - Learn how to work with large code bases which is useful if you work in industry (and for many research projects).

# Topic 1: Introduction to Operating Systems

What happens when you edit a file on a USB flash drive and forget to save it before trying to remove the USB drive?

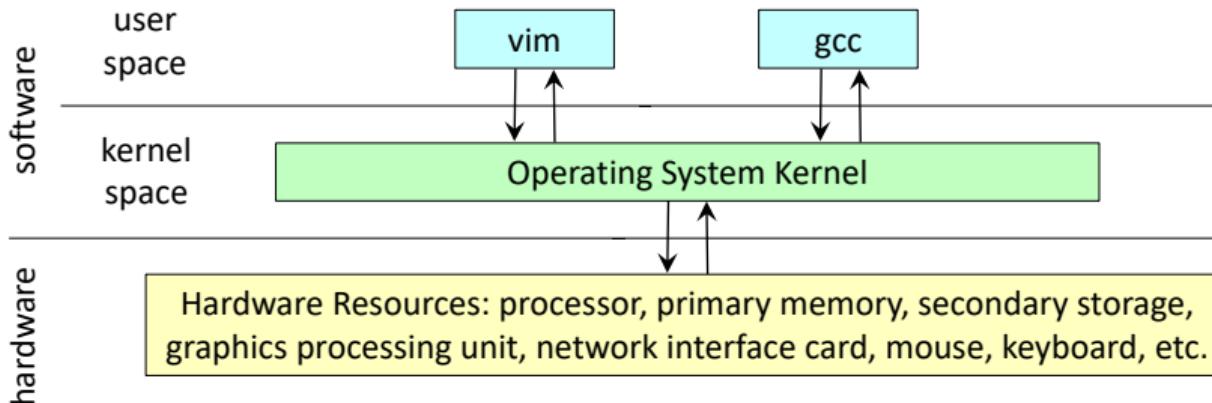
Answer:

What is the function call in C that opens a text file, *blah.txt*, for reading, given that the file is located on a Western Digital 1TB Solid State Drive connected via a NVMe interface and the file system format is NTFS?

Answer:

*Key Point:* An operation system is the layer between the applications and hardware.  
E.g. the OS is part cop, part facilitator.

# What is an operating system?



- Makes hardware useful to the programmer
- Usually: *Provides abstractions* (e.g. interfaces) for applications (e.g. file system for vim),
  - i.e. (1) managing and (2) hiding the details of the hardware.
  - The OS accesses the hardware through low level interfaces unavailable to applications.
- Often: *Provides protection* to prevent one process or user from clobbering another.

# What is an operating system?

- To get a sense how useful the abstractions are, consider the document that describes the SATA interface, which is the interface between the OS and either a hard drive or a solid state drive.
- You would need to understand this interface to access a file on one of these devices.
- [https://sata-io.org/system/files/specifications/SerialATA\\_Revision\\_3\\_1\\_Gold.pdf](https://sata-io.org/system/files/specifications/SerialATA_Revision_3_1_Gold.pdf)
- It is 717 pages long!
- Adding to the challenge, the OS itself also uses resources, which it must share with application programs. This fact makes creating an OS more complex.

To get a sense of all the devices that an OS has to manage on a typical laptop see the Device Manager (in Windows) or About This Mac > System Report (in macOS).

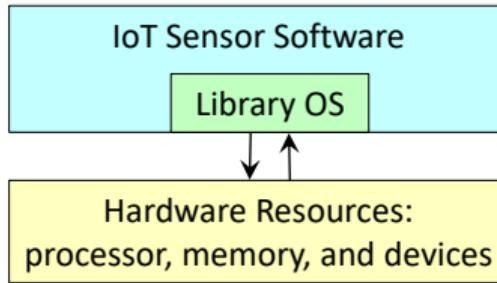
# Why study operating systems?

- The challenge: Operating systems are a maturing field.
  - Most people use select from a handful of mature OSes (e.g. Android vs. iOS).
  - It is hard to get people to switch operating systems.
  - Therefore, it is hard to have impact with a new OS. But ...
- *High-performance servers* are an OS issue.
  - They face many of the same issues as OSes.
- *Resource consumption* is also an OS issue.
  - Battery life (hence the need for smart phone OSes), radio spectrum, etc.
- *Security* is an OS issue.
  - Security requires a solid foundation
- New “smart” devices need new OSes.
- Web browsers, databases, and game engines look like OSes.

# Course topics

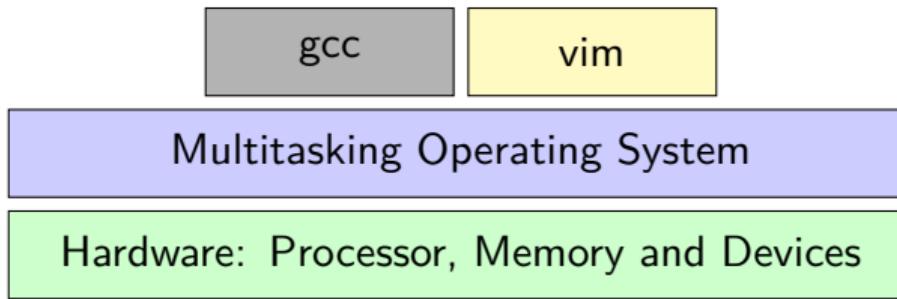
- Threads & Processes
- Concurrency & Synchronization
- Scheduling
- Virtual Memory
- I/O
- Disks, File systems, Network file systems
- Protection & Security
- Virtual machines
- Will often use Unix as the example
  - Most OSes were heavily influenced by Unix: e.g. macOS (is Unix), Linux (inspired by Unix), Android, and OS161.
  - Windows is a notable exception

# Types of Operating Systems: Primitive OS



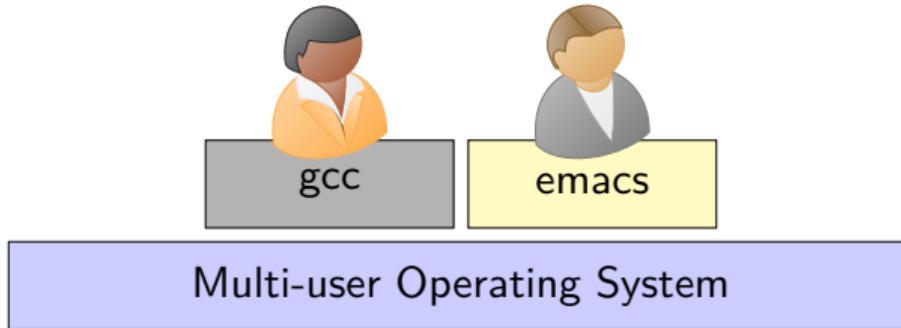
- A **library OS** is a *library of standard services* (without any protection).
  - Standard interface above hardware-specific drivers, etc.
- These make simplifying assumptions, i.e.
  - the system runs one program at a time and
  - there are no bad users or programs (often bad assumption).
- Problem: Poor utilization of ...
  - hardware (e.g., CPU idle while waiting for I/O)
  - of human user (must wait for each program to finish).

# Types of Operating Systems: Multitasking OS



- A **multitasking** OS can *have multiple processes existing at the same time.*
  - When one process blocks (i.e. is waiting for user input, IO, etc.) run another process.
- Problem: What harm can an ill-behaved process do?
  - Go into an infinite loop and never relinquishing the processor.
  - Overwrite another process's memory, making it fail.
- A multitasking OS must provide mechanisms to address these problems, e.g.
  - **Preemption** – take processor away from looping process
  - **Memory protection** – protect processes' memory from one another.

# Types of Operating Systems: Multi-user OS

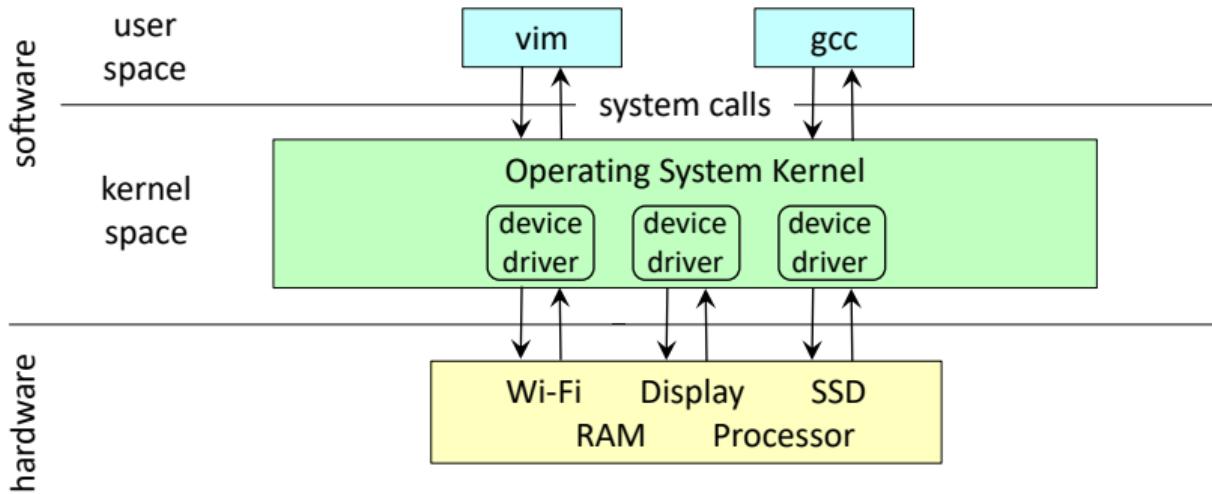


- A **multi-user** OS should provide *protection* to serve distrustful users or buggy apps.
- Idea: With  $n$  users, the system is not  $n$  times slower, because user demand for processor is bursty (e.g. read, type, select a menu item).
- What can go wrong?
  - Users can use too much CPU time, storage etc.  $\Rightarrow$  need policies
  - Total memory usage may be greater than machine has  $\Rightarrow$  must virtualize
  - Super-linear slowdown with increasing demand, called thrashing.

# Important Feature: Protection

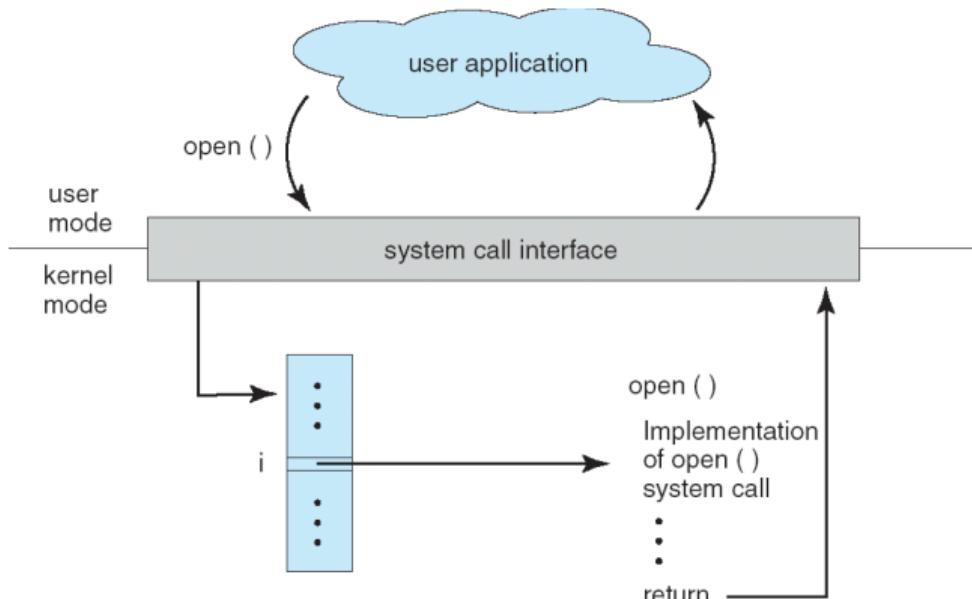
- Mechanisms that isolate bad and buggy programs, bad people and user errors.
- **Pre-emption:** is the ability to give and take away resources such as processor time.
- **Interposition or mediation:**
  - Place OS between application and the resources.
  - Track all the resources that an application is allowed to use (e.g., in table).
  - On each access, look in table to check that the access is legal.
- **Privileged and unprivileged modes** in processors:
  - User applications have unprivileged access (also called **user mode**).
  - The OS has privileged access (also called **kernel mode**)
  - Protection operations can only be done in privileged mode.

# Typical OS structure



- Most software runs as user-level processes, e.g. vim and gcc.
- OS *kernel* runs in *privileged mode*.
  - The kernel manages (i.e. creates and deletes) processes.
  - It provides access to I/O devices using **device drivers**.

# System calls

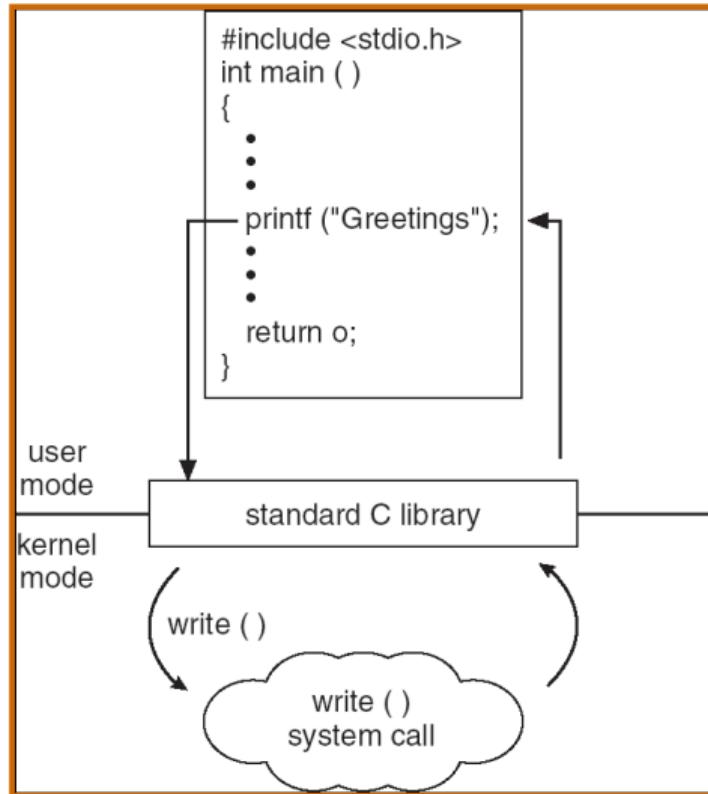


- Applications can invoke the kernel through **system calls**.
- These are special instructions (similar to a function call) which transfers control to kernel which in turn dispatches one of hundreds of system call handlers.

# System calls (continued)

- Goal: Execute functions that the app cannot do in unprivileged mode.
  - Like a library call, but using more privileged kernel code.
- The kernel supplies well-defined **system call** interface
  - Applications set up syscall arguments and **trap** to kernel.
  - The kernel performs operation and returns result.
- Higher-level functions that an application programmer would use are built on this system call interface.
  - `printf`, `scanf`, `gets`, etc. all are user-level code library routines.
- Example: POSIX interface used by Linux and Unix.
  - `open`, `close`, `read`, `write`, ...

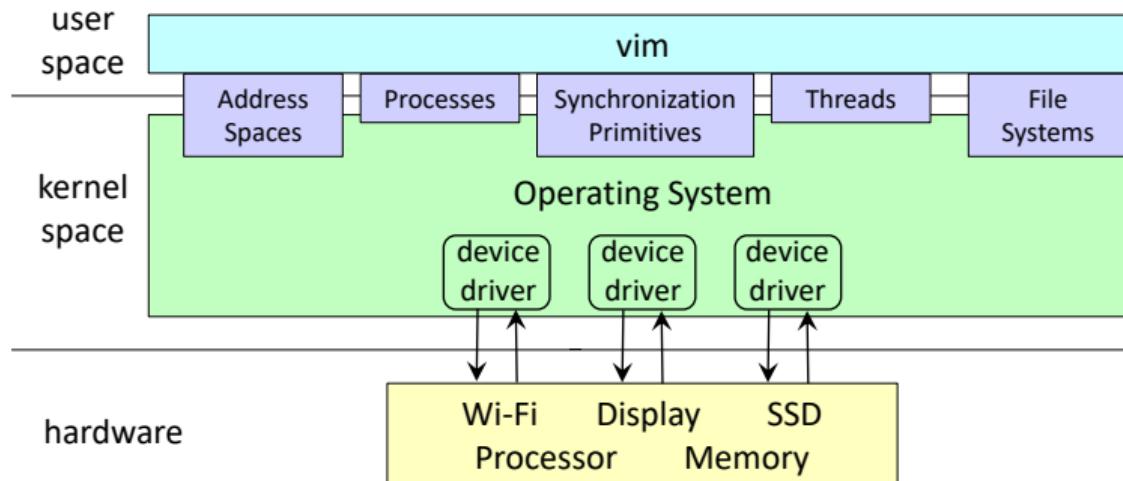
# System call example



- The C standard library is implemented in terms of syscalls.
  - E.g. `printf`, in libc, has the same privileges as the application.
  - It calls `write`, in the kernel, which can send bits out on a serial port.

# Processes

## The Process Abstraction



Processes are one of the abstractions that are commonly part of an operating system, which could also include threads, synchronization, address spaces and file systems.

# What are Processes?

- A **process** is an instance of a running program.
- Examples (can all run simultaneously):
  - `gcc file_A.c` – compiler running on file A
  - `gcc file_B.c` – compiler running on file B
  - `vim` – text editor
- Complications: sometimes what a user thinks is a “running program” does not match what the how it is actually implemented. I.e. it is not always a one-to-one match.
  - Each time notepad (a simple text editor) is launched in Windows 10, it creates a new process.
  - After MS Word is launched in Windows 10, the OS recognizes that Word is running and does not create any additional process each additional time it is launched.

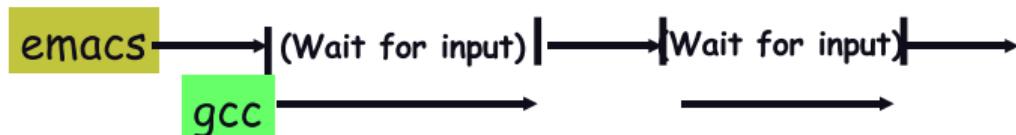
# Processes

Browsers will often have a separate process for each website you go to (in addition to other processes) so that if a website has errors that cause the process to crash, you only lose one tab and the rest of the tabs are unaffected.

- Modern OSes run multiple processes simultaneously. Why?
  - *Simplicity* of programming: a programmer does not have to account for the fact that other processors may be running.
  - As hinted above, with the discussion on browsers, it is done for *robustness*. A program with bugs should not crash other processes or the OS.
  - Higher **throughput**, i.e. better processor utilization (time being used vs. time waiting for I/O etc).
  - Lower **latency**: time between launching a program and getting a response.

# Speed

- Multitasking can *increase processor utilization* by overlapping one process's computation time with another's time waiting for I/O.

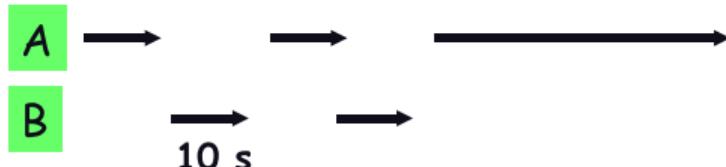


- Multitasking can *reduce latency* compared to sequence execution.

- Running *A* then *B* requires 100 sec for *B* to complete



- Running *A* and *B* concurrently allows short jobs (e.g. *B*) to finish faster.



- *A* is slower than if it had run sequentially, but still < 100 sec unless both *A* and *B* completely CPU-bound.

# I/O and CPU Bound Processes

- A process is **I/O bound** if the time it takes to complete a task is largely determined by the time waiting for input or output.
  - A word processor spends most of its time waiting for the user to input a character.
  - A music player app spends most of its time waiting for the sound card to output the sound.
- A process is **CPU bound** if the time it takes to complete a task is largely determined by the speed of the processor.
  - A lot of scientific computation or processing video is CPU bound.
- If both *A* and *B* are completely CPU-bound, then the system cannot take advantage of running one of them while the other is waiting for I/O.

# Concurrency, Parallelism and Preemptive Multitasking

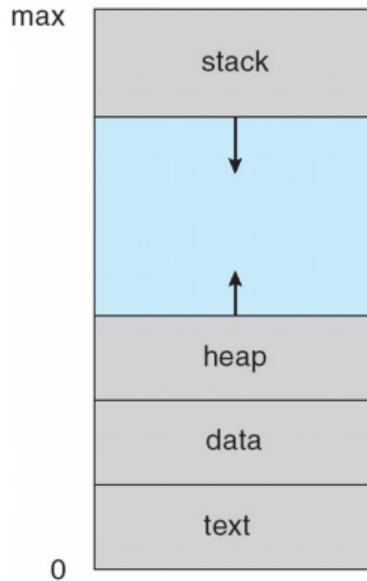
- Recall: multitasking means allowing multiple processes to exist at the same time.
- Concurrency is a way of using multitasking to improve the performance of a computer or smartphone.
- **Concurrency** means multiple programs or sequences of instructions running, or appearing to run, at the same time. It comes in two forms.
  1. **Parallelism** run at the same time because of multiple core processors or multiple processors.
  2. **Preemptive multitasking** appear to run at the same time but are actually rapidly switching between them on the same hardware. Sometimes called timesharing.
- Most modern computers, laptops, and smart phones are multi-core.
- 4 cores = 4 processes can run in parallel = potentially  $4 \times$  throughput
- Most modern OSes support preemptive multitasking.

# Concurrency, Parallelism and Timesharing

- The idea of parallelism has been much longer than OSes.
- E.g. a company could hire 90 workers to make 90 widgets and have them work in parallel.
- If it takes 1 worker 10 months to make 1 widget.
- Latency for first widget  $\gg 1/10$  month.
- I.e. it does not necessarily speed up the time to produce the first widget.
- Throughput may be  $< 10$  widgets per month  
(if the task cannot be perfectly parallelized).
- And 100 workers making 10,000 widgets may achieve  $> 10$  widgets/month.
  - E.g. if for part of the process they have to wait for some subtask to be completed.

# A Process's View of the Computer

- *Each process has own view of the computer* with its own
  - address space,
  - open files.
  - and virtual CPU (through preemptive multitasking).
- An address that each process sees is a virtual address.  
I.e. the same virtual address refers to different physical addresses in  $P_1$  and  $P_2$ .
- Simplifies programming model. E.g. gcc does not care that firefox is running.
- Sometime interaction between processes is desirable.
  - The simplest method is through files. E.g. emacs edits file and gcc compiles it.
  - More complicated: Shell/command, Window manager/app.



# Two Views of Processes

Two views of a process are ...

## 1. The *user view* of processes.

- Introduction to the Unix/Linux system call interface.
- Learn some process management function, e.g.  
how to create, kill, and communicate between processes.
- The running example is how to implement a shell.

## 2. the *kernel view* of processes

- Implementing processes in the kernel

## User View: Creating processes

- `int fork (void);`
  - *Create a new process that is an exact copy of current one.*
  - Returns *process ID* of new process in “parent”
  - Returns 0 in “child”
  - Gets called once and returns twice: once in parent and once in child
- `int waitpid (int pid, int *status, int options);`
  - *Attempt to get exit status of child.*
  - `pid`: process ID to wait for, or -1 for any
  - `status`: will contain exit value, or signal
  - `options`: 0 means wait for the child to terminate (exit or crash) or `WNOHANG` which means do not wait for the child to terminate.
  - Returns process ID or -1 on error

## User View: Deleting Processes

- `void exit (int status);`
  - *Have the current process exit* (like returning from main).
  - `status`: shows up in an encoded format if `waitpid` if called.
  - By convention, `status` of 0 is success, non-zero error.
- `int kill (int pid, int sig);`
  - *Sends signal sig to process pid.*
  - A **signal** is a software interrupt (i.e. interrupts the running of the program).
  - `SIGTERM` most common value, kills process by default  
(but application can catch it for “cleanup”)
  - `SIGKILL` stronger, kills process always.

## User View: Running Programs

- `int execve (char *prog, char **argv, char **envp);`
  - *replaces the current running program (that called execve) with a new one prog.*
  - `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 wrapper functions
  - `int execvp (char *prog, char **argv);`  
Searches `PATH` for `prog`, using current environment
  - `int execlp (char *prog, char *arg, ...);`  
list the arguments one at a time, finish with `NULL`
- Example: A Mini Shell `minish.c`: a loop that reads a command, then executes it.

## minish.c (simplified)

Parent Process (pid = 5)

```
1 pid_t pid; char **av;
2 void doexec() {
3     execvp(av[0], av); // if execvp is successful, program av[0] will run
4     perror(av[0]);    // if unsuccessful, perror will run
5     exit(1);
6 }
7
8 /* ... main loop: */
9 for (;;) {
10     parse_input(&av, stdin);
11     switch (pid = fork()) {
12         case -1:
13             perror("fork"); break;           // if fork is unsuccessful, perror will run
14         case 0:
15             doexec();                     // if successful, child will run this
16         default:
17             waitpid(pid, NULL, 0); break; // if successful, parent will run this
18     }
19 }
```

## minish.c (simplified)

- Line 1: `av` is the argument vector. By convention, `av[0]` is the name of the program.
- Line 4: *perror prints the last error encountered during a call to a system or library function.* If `execvp` is successful, this will not execute. The program specified by `av` will.
- Lines 2–6: attempts to execute the program specified by `av[0]` with the arguments specified in the rest of `av`.
- Line 10: `parse_input` converts the input from a single string (e.g. "ls -l docs") into an NULL terminated array of pointers to strings ( e.g. `av[0] = "ls"`, `av[1] = "-l"`, `av[2] = "docs"` and `av[3] = NULL`).
- Line 11: when `fork()` is called either
  - an error occurs, which is reported on lines 12–13 or
  - a child is created (say `pid=6`), which executes lines 14–15 (since its return value from `fork()` is 0) while the parent (say `pid=5`) executes lines 16–17 (since its return value is an integer  $> 0$ ).

## minish.c (simplified)

Parent Process (pid = 5)

```
1 pid_t pid; char **av;
2 void doexec() {
3     execvp(av[0], av);
4     perror(av[0]);
5     exit(1);
6 }
7
8     /* ... main loop: */
9     for (;;) {
10         parse_input(&av, stdin);
11         switch (pid = fork()) {
12             case -1:
13                 perror("fork"); break;
14             case 0:
15                 doexec();
16             default: // ← After Fork (pid = 6)
17                 waitpid(pid, NULL, 0); break;
18         }
19     }
```

Child Process (pid = 6)

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

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

## minish.c (simplified)

Parent Process (pid = 5)

```
1 pid_t pid; char **av;
2 void doexec() {
3     execvp(av[0], av);
4     perror(av[0]);
5     exit(1);
6 }
7
8     /* ... main loop: */
9     for (;;) {
10         parse_input(&av, stdin);
11         switch (pid = fork()) {
12             case -1:
13                 perror("fork"); break;
14             case 0:
15                 doexec();
16             default: // ← After Fork (pid = 6)
17                 waitpid(pid, NULL, 0); break;
18         }
19     }
```

Child Process (pid = 6)

```
pid_t pid; char **av;
void doexec() {
    execvp(av[0], av); // ← After Fork
    perror(av[0]); // Never executes!
    exit(1);
}

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

# minish.c (simplified)

Parent Process (pid = 5)

```
1 pid_t pid; char **av;
2 void doexec() {
3     execvp(av[0], av);
4     perror(av[0]);
5     exit(1);
6 }
7
8     /* ... main loop: */
9     for (;;) {
10         parse_input(&av, stdin);
11         switch (pid = fork()) {
12             case -1:
13                 perror("fork"); break;
14             case 0:
15                 doexec();
16             default: // ← After Fork (pid = 6)
17                 waitpid(pid, NULL, 0); break;
18         }
19     }
```

Child Process (pid = 6)

- Replaced by the new program (specified by `av[0]`).

```
int
main(int argc, const char *argv[])
{
    // ← Starts executing here!
    ...
    exit(0);
}
```

# minish.c (simplified)

Parent Process (pid = 5)

```
1 pid_t pid; char **av;
2 void doexec() {
3     execvp(av[0], av);
4     perror(av[0]);
5     exit(1);
6 }
7
8     /* ... main loop: */
9     for (;;) {
10         parse_input(&av, stdin);
11         switch (pid = fork()) {
12             case -1:
13                 perror("fork"); break;
14             case 0:
15                 doexec();
16             default:
17                 waitpid(pid, NULL, 0);
18                 break; // ← waitpid returned
19             }
20 }
```

Child Process (pid = 6)

- Replaced by the new program (specified by `av[0]`).

```
int
main(int argc, const char *argv[])
{
    ...
    exit(0); // ← Wakes up waitpid
}
```

# The POSIX File Interface Overview

- Recall that the `stdio` library in C provides functions for working with files such as `fopen`, `fscanf`, `fprintf`, `fseek` and `fclose`.
- Since popular OSes, follow (or mostly follow) a standard that was originally meant for different versions of Unix, called the Portable Operating System Interface (POSIX).
  - These OSes include Linux, iOS and macOS.
  - Packages are available for others: e.g. Android NDK and Windows Subsystem for Linux.
- POSIX functions for working with files include: `open`, `read`, `write`, `lseek`, and `close`.
- `open` *returns a file descriptor* (or identifier or handle).
  - Other file operations (e.g., `read`, `write`, `lseek` and `close`) for that process require this file descriptor as an argument in order to identify that file from all the others.

# The POSIX File Interface Overview

- `close` *invalidates a valid file descriptor* for that process.
  - The kernel tracks which file descriptors are currently valid for each process.
- `read` copies data from a file into the process's address space.
- `write` copies data from process's address space into a file.
- `lseek` enables non-sequential reading and writing.
- *Each open file* (i.e. valid descriptor) *has an associated file position.*
  - This position starts at byte 0 when the file is opened.
- Read and write operations
  - start from the current file position and
  - update the current file position as bytes are read/written.
  - This arrangement makes sequential file I/O easy to use.

# The POSIX File Interface Overview

- `lseek` is used for non-sequential file I/O.
  - `lseek` changes the file position associated with a descriptor.
  - The next read or write from that descriptor will use that new position.
- POSIX also provides some standard file descriptors that for files that are open by default for each process. These include
  - 0 for `stdin`,
  - 1 for `stdout`,
  - 2 for `stderr`.
- Normally these are all directed towards the console but can be redirected to read from or write to files.
- Why use POSIX?
  - POSIX provides a richer set of functions and more options for dealing with files.

# Manipulating file descriptors

- `int dup2 (int oldfd, int newfd);`
  - *Makes newfd an exact copy of oldfd.*
  - Closes `newfd`, if it was a valid descriptor.
  - The two file descriptors will share same offset (`lseek` on one will affect both)
- For the following code snippet

```
fd = open(infile, O_RDONLY);
dup2(fd, 0);
```

the file `infile` is opened and is set to `stdin`, i.e. reading from `stdin` now reads from `infile`.

- This function is useful if you are trying to handle redirecting input e.g.  
`command < input`

# Manipulating file descriptors

- `int fcntl (int fd, F_SETFD, int val)` sets the `close-on-exec` flag. I.e.
  - If `val`=1, a successful call to `execvp` will *close* the file corresponding to `fd`.
  - If `val`=0, a successful call to `execvp` will *leave open* the file corresponding to `fd`.
  - If the call to `execvp` is unsuccessful `fd` will be *left open*.
- An enhanced version our mini shell called `redirsh.c` will use `dup2` to help deal with commands that redirect input, output or error message. I.e. for a command such as:  
`command < input > output 2> errlog`
- The function that parses the command line, `parse_input`, will now recognize that
  - if "< input" is present then the file `input` should become `stdin`,
  - if "> output" is present then the file `output` should become `stdout`,
  - if ">2 errlog" is present then the file `errlog` should become `stderr`.and call them `infile`, `outfile` and `errfile`, respectively.

## redirsh.c

```
1 void doexec (void) {
2     int fd;
3     if (infile) {      /* non-NULL for "command < input" */
4         if ((fd = open(infile, O_RDONLY)) < 0) {
5             perror(infile);
6             exit(1);
7         }
8         if (fd != 0) {
9             dup2(fd, 0);
10            close(fd);
11        }
12    }
13
14    /* ... do same for outfile→fd 1, errfile→fd 2 ... */
15    execvp (av[0], av);
16    perror (av[0]);
17    exit (1);
18 }
```

- Before calling `doexec`, `infile` points to NULL. The function `parse_input` will look for a substring of the form “< input” and if it exists set `infile` to point to it.
- Line 3: If `infile` is not NULL, i.e. the input was redirected,
- Lines 4–7: then try to open `infile` and report an error if it fails (i.e. a negative `fd`).
- Lines 8–11: If `fd` is not `stdin` then set `stdin` to correspond to this file and close the original file.
- Line 14: repeat check for redirected output and error messages.
- Lines 15–17: this is what the old version of `doexec` did in `minish.c`.

# Pipes

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

## pipesh.c (simplified)

```
1 void doexec (void) {
2     while (outcmd) {
3         int pipefds[2]; pipe(pipefds);
4         switch (fork()) {
5             case -1: // error
6                 perror("fork"); exit(1);
7             case 0: // child writes to the pipe
8                 dup2(pipefds[1], 1); //stdout is pipe input
9                 close(pipefds[0]); close(pipefds[1]);
10                outcmd = NULL;
11                break;
12            default: // parent reads from the pipe
13                dup2(pipefds[0], 0); // stdin is pipe output
14                close(pipefds[0]); close(pipefds[1]);
15                parse_input(&av, &outcmd, outcmd);
16                break;
17        }
18    }
```

- Line 2–18: Loop through the different pairs of commands, e.g. `cmd1 | cmd2 | cmd3`
- Line 2: `outcmd` represents the sequence of commands the pipe(s) connect. It will loop until there are no further commands to output to. The child will set `outcmd` to NULL after sending its output the pipe. The parent will not set it to NULL but will continue in the loop to send output to `cmd3`.
- Line 3: Execute the system call `pipe` to set up the two ends of a pipe, say `cmd1 | cmd2`.
- Lines 4–6: Call `fork` and report the error and exit if it fails.
- Lines 7-11: The child will handle `cmd1` and sets the pipe's input to its output, `stdout`.
- Lines 12–14: The parent will handle `cmd2` by setting the pipe's output to its input, `stdin`.
- Lines 15: It will then continue on if there are more commands in the sequence of pipe statements.

# Why fork?

- Most calls to `fork` are followed by `execve`
- Could also combine into one **spawn** system call, which loads and executes a new child process.
- It is occasionally useful to fork one process.
  - Pre-forked Webservers for parallelism
  - Creates one process per core to serve clients
  - Lots of uses: Nginx (a webserver), PostgreSQL (a database), etc.
- The real win is *simplicity of the interface*.
  - There are many things you might want to do to child:  
Manipulate file descriptors, environment, resource limits, etc.
  - Yet `fork` requires *no* arguments at all.

## Spawning process w/o fork

- Without fork, creating a process requires many different options.
- 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
);
```

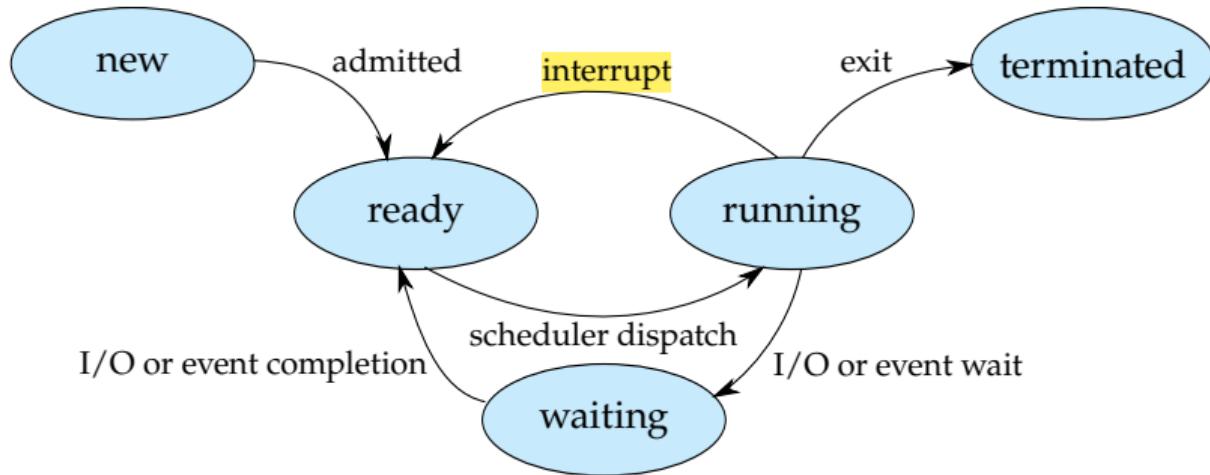
# Implementing processes

- The OS *maintains data about each process.*
  - Typically called a **Process Control Block** (PCB) but also called `proc` in Unix and a `task_struct` in Linux.
- It tracks the **state** of the process. See next slide.
- 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 (which signals are being blocked), controlling terminal, priority, accounting statistics, whether being debugged, etc.

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

PCB

# Process states



- A process can be in one of the following states
  - **new** at creation and **terminated** after exiting or making a fatal error
  - **running** – currently executing (or will execute on kernel return)
  - **ready** or **runnable** – can run, but the kernel has chosen different process to run
  - **waiting** – waiting for an event (e.g., disk operation) to occur before proceeding

# Scheduling

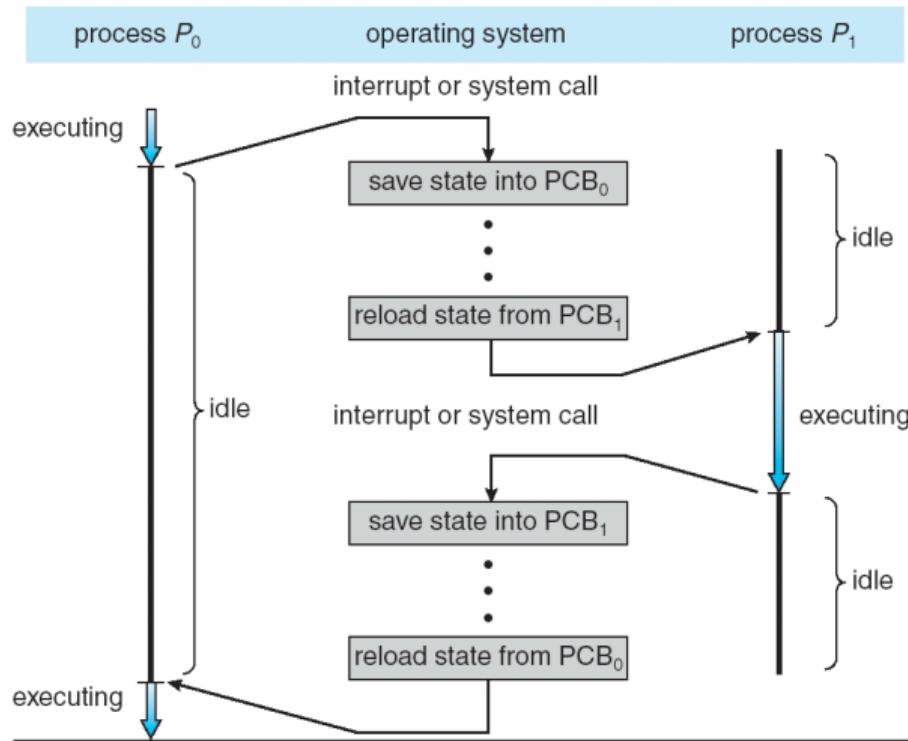
- Which process should kernel run?
  - if 0 are runnable, run idle loop (or halt CPU), if 1 is runnable, run it
  - if  $>1$  are runnable, must make a scheduling decision about which one to run.
- **Scheduling** is the method used to decide which process to run.
- Idea: Scan process table for first runnable?
  - Problems: Expensive. Weird priorities (small pids do better)
- Better Idea: Divide into runnable and waiting processes and use FIFO/Round-Robin on the list of runnable processes.
  - Add to the back of list and dispatch them from the front.
- Priority?
  - Give some a better shot at the being selected.



# When to Preempt

- Can preempt a process when kernel gets control.
  - System call, page fault (data in swap space), illegal instruction, etc.
  - May put current process to sleep, e.g., when reading from a disk.
  - May make other process runnable, e.g., fork, write to pipe.
- Have periodic timer interrupts and if a 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 process.
- Changing the running process is called a **context switch**.

# Context switch

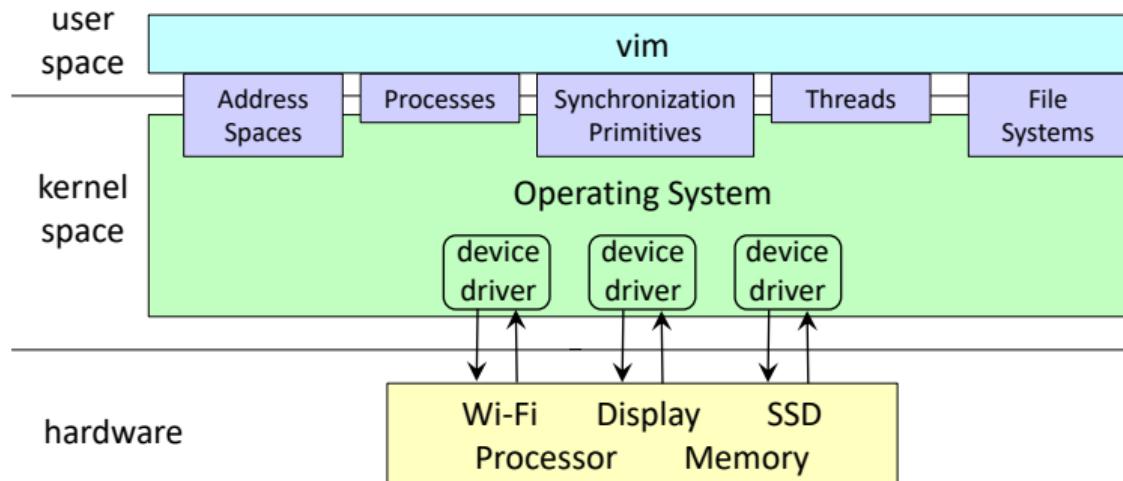


# Context switch details

- 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
- *A context switch has non-negligible cost.*
  - Save/restore floating point registers expensive
    - Optimization: only save if process used floating point
  - May require flushing TLB (address translation hardware)
    - H/W Optimization 1: don't flush kernel's own data from TLB
    - H/W Optimization 2: use tag to avoid flushing any data
  - Usually causes more cache misses (switch working sets)

# Threads

## The Thread Abstraction

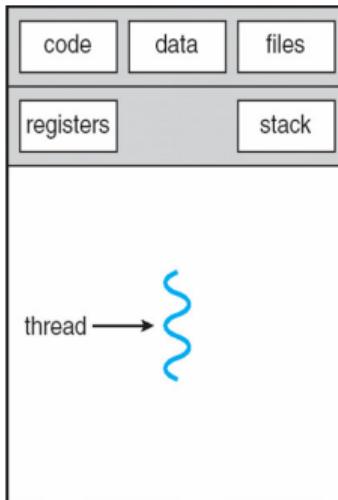


Threads are one of the abstractions that are typically part of an operating system. You can think of a **thread** is *a sequence of instructions*.

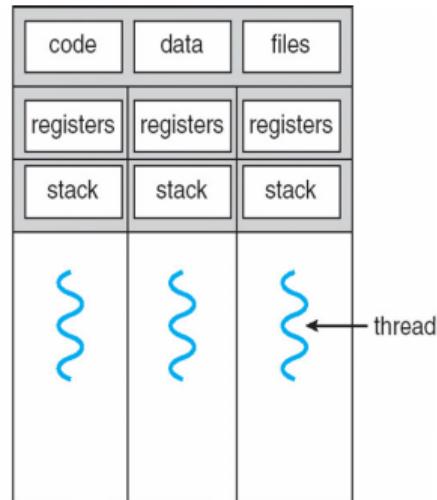
# What are threads?

- A more precise definition is a **thread** is a *schedulable execution context*.
  - I.e. it is something that could execute on a core (the code) but also includes the context, e.g. **global variables, the heap, the stack, file descriptors**.
  - Recall that the stack contains stack frames which **store local variables, return addresses, register values that need to be preserved, and temporary values**.
- A sequential program (as described in CS241) consists of a single thread of execution.
- Somewhat analogous to a DFA (a single current state) versus an NFA (a set of current states) a program can have a single thread of execution or multiple threads of execution.
- What if you wanted to run multiple tasks that a process is responsible for, at the same time on a multiple cores.
  - For example, render some text, some pictures and execute some Javascript on a webpage.
- **Key Question** How would you implement that feature?

# What are threads?



single-threaded process



multithreaded process

- **Key Strategy:** Share what you can. Keep separate versions only if necessary.
- **Each thread has its own stack and register values** (e.g. program counter, instruction register, stack pointer, frame pointer).
- **Threads** in the same process **share**: the code, global variables, heap and file descriptors of the process.

# What Are Threads?

- If one thread was responsible for executing the Javascript code, then *its stack* would contain a lot of stackframes for Javascript functions.
- If another thread was responsible for rendering text, then *its stack* would contain a lot of stack frames for functions for displaying text.
- Each thread would be executing a different set of instructions so they would each have their own values for the *registers*, e.g. program counter, the instruction register, the stack pointer, the frame pointer, return address.

- You can view these multiple sequences of instructions using Windows 10's Task Manager or macOS's Activity Monitor.
- Look at a simple program (e.g. notepad), a more complicated one (e.g. MS Word), a even more complicated one (e.g. Chrome) and the kernel itself.

# Why threads?

- Threads are the most popular abstraction for concurrency.
- They are a **lighter-weight** abstraction than processes. I.e. they *take less system resources* (processor time and memory) to create and manage.
- Allows one process to use multiple processors or cores in parallel.
- Allows a single program to overlap its I/O and computation.
  - Same benefit as OS running emacs & gcc simultaneously
  - E.g., threaded web server services clients simultaneously:

```
for (;;) {
    fd = accept_client ();
    thread_create (service_client, &fd);
}
```
- Most kernels have threads, too.
  - Typically at least one kernel thread for every process.

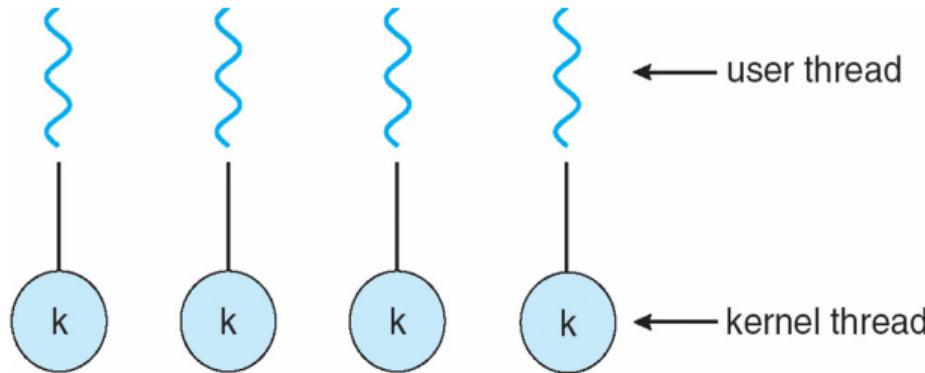
# POSIX thread API

- `int pthread_create (pthread_t *thr, pthread_attr_t *attr,  
void *(*fn)(void *), void *arg);`
  - *Create* a new thread identified by `thr` and it will execute `fn` with arguments `arg`. You can provide optional attributes such as a stack size or use NULL for default values.
- `void pthread_exit(void *return_value);`
  - *Destroy* current thread and return a pointer to its return value.
- `int pthread_join(pthread_t thread, void **return_value);`
  - *Wait for thread* `thread` to exit and receive the return value.
- `void pthread_yield();`
  - Tell the OS scheduler to *run another thread* or process if available.
- Plus lots of support for synchronization (next lecture topic or see [Birell])

# Kernel threads vs. user threads

- There are two types of threads:
  - **User threads** are threads that the user application creates and manages, including which one will run next.
  - **Kernel threads** are threads that the OS creates and manages, including which one will run next.
- Recall that a system call is expensive. It might take 100 or so assembly language instructions for a system call and only a few assembly language instructions for a function call.
- Functions such as `pthread_create` could be implemented as a function call in user space (i.e. inexpensive) or a system call to kernel space (more expensive).

## Approach #1: Kernel threads

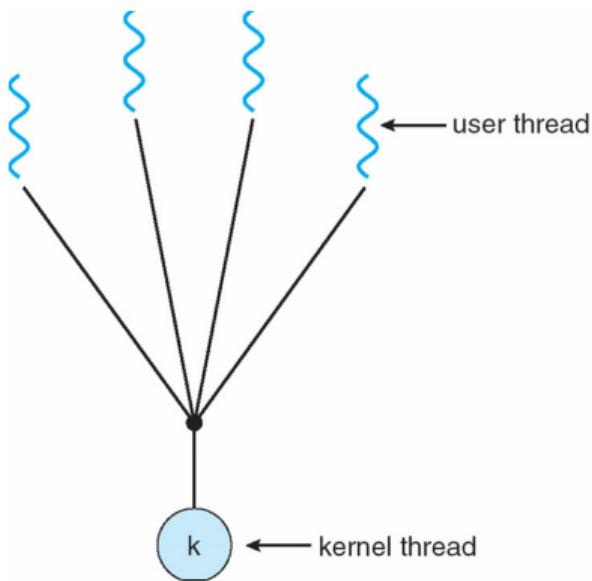


- To add `pthread_create` to an OS as a kernel thread:
  - start with process abstraction in kernel but do not duplicate code, heap, global data, etc.
  - `rfork` (Plan 9) and `clone` (Linux) syscalls allow individual control of what to share.
- This approach *uses less system resources* (processor time and memory) *than creating a new process* (because not as much is copied over into the new thread), but still heavy weight because every thread operation must go through the kernel (using an expensive syscall).

## Approach #1: Limitations of kernel-level threads

- Every thread operation (create, exit, join, yield) must go through the kernel. I.e. there is 1 : 1 correspondence between user and kernel threads.
  - Result: *thread operations are 10×–30× slower when implemented in kernel.*
  - Worse today because of SPECTRE/Meltdown mitigations.
  - These hardware bugs allow unauthorized access to the kernel's address space.
  - System calls are now slower to mitigate this problem.
- *One-size fits all* thread implementation.
  - Kernel threads must *please all sorts of application demands.*
  - User apps may have to *pay for fancy features (priority, etc.) they don't need.*
- General *heavy-weight memory requirements*
  - E.g., requires a fixed-size stack (e.g. 2 MB) within the kernel.
  - Other data structures designed for heavier-weight processes.

## Approach #2: User threads



- An alternative: implement threads in a user-level library.
  - Many user threads but only one kernel thread per process (i.e.  $n : 1$ ).
  - `pthread_create` etc., are library functions accessed via function calls.

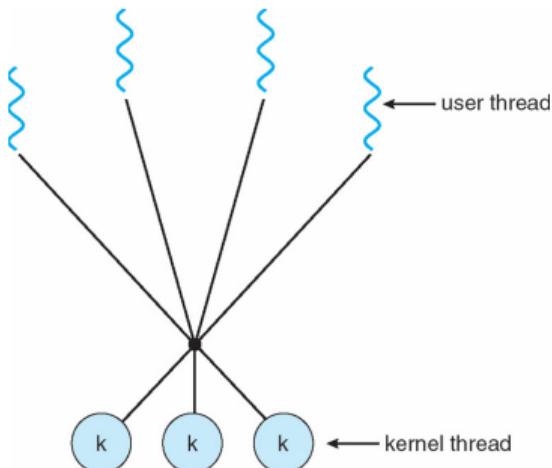
## Approach #2: Implementing user-level threads

- Allocate a new stack for each `pthread_create`.
- Keep a queue of runnable threads.
- **Blocking** describes moving from a running state to a waiting state.
- Replace blocking system calls (`read/write/etc.`) with non-blocking ones.
  - 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 to run.
  - On timer or when idle check which connections have new data.

## Approach #2: Limitations of user-level threads

- This approach *cannot take advantage of multiple processors or cores.*
- A *blocking system call blocks all threads.*
  - Can replace `read` to handle network connections.
  - But usually OSes don't let you do this for disk access.
  - So one uncached disk read blocks all threads.
- A *page fault (page not in RAM but on disk) 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.]

## Approach #3: 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

## Approach #3: Limitations of $n : m$ threading

- Many of the *same problems as  $n : 1$  threads*
  - Blocked threads, deadlock, etc.
- *Hard to* match the number of kernel threads to available cores in order to *maximize parallelism*.
  - Kernel knows how many cores are available.
  - Kernel knows which kernel-level threads are blocked.
  - It tries to hide these details from applications for transparency.
  - User-level thread scheduler might think a thread is running while the underlying kernel thread is blocked
- Kernel doesn't know *relative importance of each thread*.
  - Might preempt kernel thread in which library thread holds an important lock.

# Lessons

- Threads are *best implemented as a library but kernel threads are not the best interface.*
- Better kernel interfaces have been suggested, e.g. Scheduler Activations [Anderson et al.]
- May be too complex to implement on existing OSes.
- Some have added then removed such features. Microsoft is now trying it with Windows.
- Today's limitations shouldn't dissuade you from using threads.
  - Standard user or kernel threads are 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.
- Admittedly, concurrency (and later synchronization) may
  - greatly increase the complexity of a program and
  - lead to all sorts of nasty race conditions (output depends on order the threads ran).

# Case Study: Go Language and Go Routines

- *Go routines* (a.k.a. goroutines) are very *light-weight* user-level threads.
  - Sometimes called a light-weight thread.
  - A 100KiB ( $Ki = 1024$ ) go routines are practical whereas an OS thread typically allocates 2 MiB ( $Mi = 1024^2$ ) just for the stack.
  - For go routines, the stack starts out small (currently 2 KiB) but can grow if needed.
  - Context switch from one go routine to another is a function call rather than a system call.
  - Custom compiler enables stack segmentation, preemption, and garbage collection.
- *Go routines run on top of kernel threads* (*n:m* Model)
  - Multi-core scalability and efficient user-level threads.
  - One pthread (kernel-level thread) per processor core.
  - Supports as many user-level threads as you would like.

# Go Routine Continued

- *Every logical core is owned by a kernel thread when running.*
  - Each kernel-level thread finds and runs a go routine (user-level thread).
- *Convert blocking system calls* (when possible):
  - Converted to non-blocking by yielding the core to another go routine.
  - Cores poll using kernel event API `poll` for files, `epoll` for network connections, or `kqueue` to check whether an event has happened or not.
- Compare this approach with blocking system calls:
  - Release the core to another kernel-level thread before the call (which is expensive).
  - Let the kernel-level thread sleep (move to waiting state).
  - Move to the kernel-level thread to the ready queue when the system call has been completed.

# Go Channels

- Go routine communicate and synchronize through *channels*

```
func worker(done chan bool) {
    // Notify the main routine
    done <- true
}

func main() {
    // Create a channel to notify us
    done := make(chan bool, 1)

    // Create go routine
    go worker(done)

    // Block until we receive a message
    next() // <- start executing here when worker is done done
}
```

# Registers and Processors

## Differences Between CS241 and CS350

- *Processors:* In CS241 we used a MIPS processor. In CS350 we will use an Intel x86-64 or AMD64 processor. x64 is sometimes used to refer to either of them.
- *Register Names:* MIPS register names start with a '\$', e.g. \$30. x64 register names start with '%', e.g. %rsp.
- *Register Names:* In CS241 we named the MIPS registers \$0, \$1, \$2, etc. In CS350 we referred to many of them by their default function, %rsp (for stack pointer).
- *Passing Arguments:* In CS241 passed all arguments on the stack. In CS350 we pass the first 6 args in registers and if there are more than 6 args, pass the rest on the stack.
- *Saving Registers:* In CS241 the callee saved most of the registers (except the Frame Pointer and the Return Address).
  - This strategy has the shortcoming that the callee may be preserving register values that the caller no longer needs.

# Caller and Callee-saved Registers

CS241 tried to keep things simple. Typically, there are two types of registers.

- **caller-save** registers *are not preserved across subroutine calls*, i.e. if the caller has a value in one of these registers that it wants preserved, it should store the value on the stack before calling any subroutines and restore it afterwards otherwise the callee may change its value.
- **callee-save** registers *are preserved across subroutine calls*, i.e. if the subroutine uses one of these registers, it must first store its current value on the stack and restore it before returning.

This strategy tries to *miminize situations where the callee is saving values that the caller is not using*.

- The caller will store temp values in one of the caller-save register if the value will not be needed after any function calls.
- The callee will not preserve (store at the beginning and restore before exiting) these values.

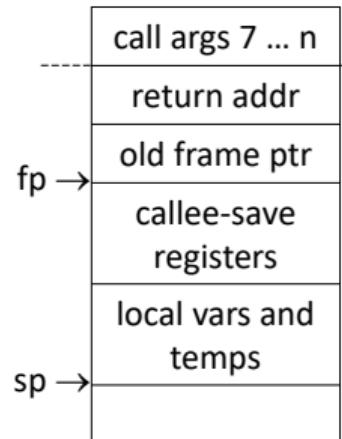
# x64 vs MIPS

Intel and AMD processors (a.k.a x64 processors) use *different terminology compared to MIPS processors* and only have 16 registers. Some of the names are based on conventional usage.

- %rax, accumulator register
- %rbx, base pointer for memory access, e.g. an array
- %rcx, counter register for loops
- %rdx, data register for I/O and arithmetic
- %rdi, destination index (for copying data)
- %rsi, source index (for copying data)
- %rbp, base pointer (a.k.a. frame pointer)
- %rsp, stack pointer
- %rip, instruction pointer (a.k.a. program counter)
- %r8 – %r15.

# Background: AMD64/x86-64 calling conventions

- For subroutine calls, the first 6 args go into %rdi, %rsi, %rdx, %rcx, %r8, %r9 respectively.
- Return values goes in %rax and %rdx.
- The caller-saved registers are: the args and return registers mentioned above, %r10 and %r11.
- The callee-saved registers are: %rbx, %rbp and %r12 – %r15.
- %rsp is the stack pointer.
- %rbp is the base pointer (a.k.a. frame pointer).
- Local variables are stored in registers and on stack.



## Background: procedure calls

save active caller registers

call foo

saves used callee registers

...do stuff...

restores callee registers

jumps back to pc

restore caller regs



- *Some state is saved on stack:*

- Return address, caller-saved registers

- *Some state is not saved:*

- callee-saved registers, global variables, stack pointer (which you can calculate from the callee's frame pointer).

# Threads vs. procedures

- Threads may *resume out of order*. Procedure call and returns are LIFO.
    - Therefore you cannot use a LIFO stack to save state for threads.
    - General solution: each thread has its own stack.
  - Procedures get called more often than threads switch.
    - A thread can make many procedure calls.
    - Don't partition registers (why?): may have many threads.
  - Threads can be *involuntarily interrupted*:
    - Synchronous: with procedure call, compiler saves required state in stack.
    - Asynchronous: when threads switch, all registers are saved.
  - More than one thread can run at a time but only one procedure runs at a time.
    - Procedure call scheduling obvious: Run called procedure.
    - Thread scheduling: What to run next and on which core?
- Multiple threads can execute concurrently, allowing for parallel execution of tasks within the same process.
- Threads run within the context of a process and share the same memory space and resources as other threads within the same process.
- A procedure (function) are executed sequentially as part of the program flow and can only run one at a time.

# Castor OS (COS): Thread Details

- Supports both kernel and user threads.
- It provides basic Pthreads support, see lib/libc posix/pthread.c
- Similar to `pthread_create`, `Thread_KThreadCreate` takes a pointer to a function, `f` and arguments, `arg`, and creates a kernel thread associated with that process.

```
Thread *Thread_KThreadCreate(void (*f)(void *), void *arg)
```

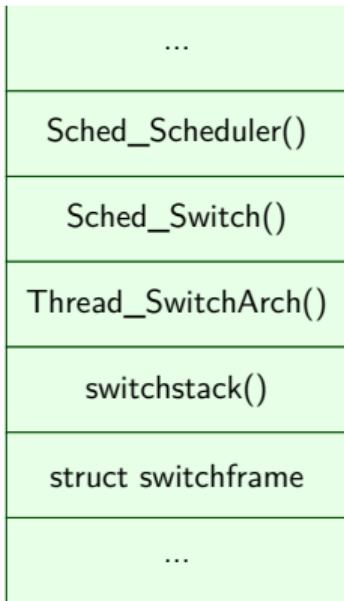
- Castor OS can also create userspace threads, which the app can manage directly. The userspace thread is associated with a kernel thread, `oldThr`, with the address of the function to execute , `rip` along with the arguments, `arg`.

```
Thread *Thread_UThreadCreate(Thread *oldThr, uint64_t rip,  
uint64_t arg)
```

# Castor OS: Switching Threads

There are two ways a thread switch can occur.

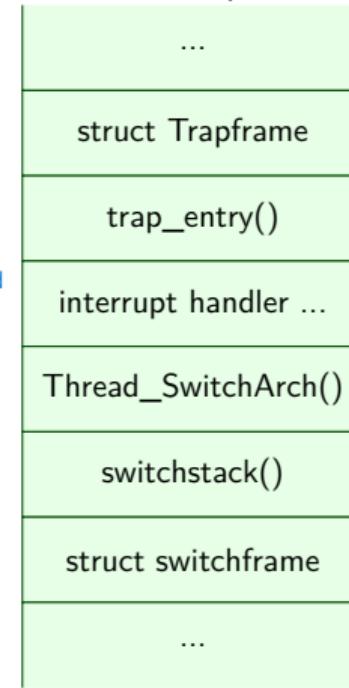
General (from Kernel)



a trapframe is concerned with preserving the state of a user process when handling a trap or exception, while a switchframe is concerned with preserving the state of kernel threads during context switches

hence, trapframe saves all register values and switchframe saves only callee-save registers

Hardware Interrupt (typically Timer)



# Castor OS: Switching Threads

- Thread switches go through `Sched_Scheduler()` to decide which thread to run next.
- It calls `Sched_Switch()` to switch the running thread.
- It calls `Thread_SwitchArch`, which is customized for the particular processor that is used.  
I.e. different processors have a different names and numbers of registers.
- It calls `switchstack`, which switches from one stack to the other while saving the callee-saved registers in a `switchframe`.
- A `struct switchframe` stores all the callee save registers.
- In the case of a hardware interrupt, a `trapframe` is created. A trapframe stores all the registers.
- The kernel must then figure out which device generated the interrupt (`trap_entry()`), identify the appropriate interrupt handler, and then switch to running the code for that device.

## Castor OS: switchstack – switch kernel threads

From COS kern/amd64/switch.S

```
9 # switch(uint64_t *oldsp, uint64_t newsp)
10 # %rdi: oldsp
11 # %rsi: newsp
12 FUNC_BEGIN(switchstack)
13     # Save callee saved registers of old thread
14     pushq    %rbp
15     pushq    %rdi
16     pushq    %rbx
17     pushq    %r12
18     pushq    %r13
19     pushq    %r14
20     pushq    %r15
21
22     # Switch stack from old to new thread
23     movq    %rsp, (%rdi)
24     movq    %rsi, %rsp
```

## Castor OS: switchstack – switch kernel threads (con't)

```
25
26     # Restore callee saved registers of new thread
27     popq    %r15
28     popq    %r14
29     popq    %r13
30     popq    %r12
31     popq    %rbx
32     popq    %rdi
33     popq    %rbp
34     ret
35 FUNC_END(switchstack)
```

## Castor OS: switchstack – switch kernel threads (con't)

- While in MIPS you have to (1) make room on the stack by decrementing the stack pointer and then (2) copy the value into the stack (i.e. two instructions), in x64 both of these steps are done with a single instruction, `pushq`.
- E.g. `pushq %rbp` saves the value stored in register `rbp` on the stack.
- Likewise `popq` removes a value from the stack and stores it in a register in a single instruction.
- The instruction `movq` will copy a value from a source register (the left operand) to a destination register (the right operand).
- `(%rdi)` means don't store it in `rdi` but store it at the address contained in `rdi`.

# Review: Processes and Threads

- A process is an instance of a running program.
  - Process can have one or more threads.
- A thread is an execution context
  - Share address space (code, global data, heap), open files.
  - Have their own CPU registers and stack (local variables).
- POSIX Thread APIs
  - `pthread_create()` – Create a new thread
  - `pthread_exit()` – Destroy the current thread
  - `pthread_join()` – Waits for a thread to exit

# Critical Sections

```
int total = 0;

void add() {
    for (int i=0; i<N; i++) {
        total++;
    }
}

void sub() {
    for (int i=0; i<N; i++) {
        total--;
    }
}
```

Question: If one thread executes `add` and another executes `sub` what is the value of `total` when they have finished for `N` = 100, 1000, 10000?

# Critical Sections

```
int total = 0; /* r8 := &total */

/* Thread 1 */
void add() {
    for (int i=0; i<N; i++) {
        lw r9, 0(r8) /* total++ */
        addi r9, 1
        sw r9, 0(r8)
    }
}

/* Thread 2 */
void sub() {
    for (int i=0; i<N; i++) {
        lw r9, 0(r8) /* total-- */
        subi r9, 1
        sw r9, 0(r8)
    }
}
```

This is one of the few places where you have to consider what goes on at the level of assembly language. *Consider the different ways the sequences of instructions can be interleaved.*

# Critical Sections: Schedule 1

Thread #1

```
-----  
lw r9, 0(r8) /* total++ */  
addi r9, 1  
sw r9, 0(r8)
```

-----

Thread #2

```
-----  
lw r9, 0(r8) /* total-- */  
subi r9, 1  
sw r9, 0(r8)
```

-----

- Schedule 1: Increment completed before decrement is started.
- Result: **total** = 0

## Critical Sections: Schedule 2

Thread #1

```
-----  
lw r9, 0(r8) /* total++ */  
addi r9, 1  
sw r9, 0(r8)  
-----
```

Thread #2

```
-----  
lw r9, 0(r8) /* total-- */  
subi r9, 1  
sw r9, 0(r8)  
-----
```

- Schedule 2: Thread #1 runs a bit *before* Thread #2
- Both load zero, but Thread #2's store overwrites Thread #1's update.
- Result: **total = -1**

## Critical Sections: Schedule 3

Thread #1

```
-----  
lw r9, 0(r8) /* total++ */  
addi r9, 1  
sw r9, 0(r8)  
-----
```

Thread #2

```
-----  
lw r9, 0(r8) /* total-- */  
subi r9, 1  
sw r9, 0(r8)  
-----
```

- Schedule 3: Thread #1 runs a bit *after* Thread #2
- Both load zero, but Thread #1's store overwrites Thread #2's update.
- Result: total = 1

# Need for Synchronization

- Problem: Data races (a.k.a. race conditions) occur without synchronization.
- A **race condition** is when the *program result depends on the order of execution*.
- Options:
  - Atomic Instructions: use processor features so that it appears to other threads that the value was modified instantaneously, i.e. perform it in one uninterruptible step.
  - Locks: prevent concurrent execution of a sequence of instructions by two different threads.
- ... it gets worse when you have multiple cores or multiple processors!

# 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() {
    pthread_t tid;
    pthread_create(tid, NULL, p1, NULL);
    p2(); pthread_join(tid);
}
```

- If one thread runs `p1` and another thread runs `p2`, can both critical sections run?

# Program A

- If thread 1 runs `p1` first ...
  - It will set `flag1` to 1 and since `flag2` is 0, it will execute `critical_section_1()`.
  - Then when thread 2 runs, it will set `flag2` to 1 and since `flag1` is 1, it will *not* execute `critical_section_2()`.
- If thread 2 runs `p2` first ...
  - It will set `flag2` to 1 and since `flag1` is 0, it will execute `critical_section_2()`.
  - Then when thread 1 runs, it will set `flag1` to 1 and since `flag2` is 1, it will *not* execute `critical_section_1()`.
- If both run at the same time ...
  - both will set their `flag`'s to 1 and since the other `flag` is 1, they will *not* execute their critical sections.

## Program A - a write followed by a read

- If `flag2` is read before `flag1` is written to RAM by one thread and `flag1` is read before `flag2` is written to RAM by another thread, then if there is some delay in the processors accessing RAM there will be problems.
- A common hardware optimization is that the write of each thread is buffered (i.e. stored for a few clock cycles while some other core is accessing RAM) and each thread continues to execute without waiting for the write to complete.
- They are allowed to read as long as the read is not an address that is stored in *their own* write buffer.
- This was fine in a system with a single core and a single processor but does not work with multiple cores or multiple processors.
- The result will be that both threads will enter the critical section.
- The system *must maintain program order when a write is followed by a read*.

## Program B

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

void p1(void *ignored) {
    data = 350;
    ready = 1;
}

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

- If one thread runs `p1` and another thread runs `p2`, can `use` be called with value 0?

## Program B - Overlapping Writes

- When `p1` runs, it will set `data` to 350, then set the `ready` flag to 1.
- When `p2` runs, it will busy wait until the `ready` flag is 1 and then it will call `use (data)`.
- if `p1`'s two writes are going to two different memory modules and the write to `data` happens a bit slower than the write to `ready` then `p2` may read the new value of `ready` and the old value of `data`.
- Recall that a **cache line** is the unit of data transfer between the core and Level 1 cache. An optimization may be to reorder writes so that multiple write to the same cache line occur at the same time.
- This was fine a system with a single core and a single processor but does not work with multiple cores or multiple processors.
- The system *must maintain program order between two write operations.*
- I.e., the system cannot reorder write operations.

## Program B - Non-blocking Reads

- The same result can happen with non-blocking reads.
- A blocking read would read the value of `ready` first, waiting for the result, and then proceed to read `data`.
- A non-blocking read would execute the first read and not wait for it to be done before executing the second read.
- If `p2`'s two writes are going to two different memory modules and the read of `data` completes before the read from of `ready` even though it was executed later, then `p2` may read the new value of `ready` and the old value of `data`.
- The system *must maintain program order between two read operations.*
- I.e. *cannot speculatively prefetch `data`.* The system must wait until `ready` is 1 before fetching `data`.

## Program C

```
int a = 0, 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);
}
```

- Three different threads execute the three different functions, could `use()` ever be called with value 0?

# Program C

- `p1` sets `a` to 1.
- If `a` is 1, then `p2` sets `b` to 1.
- Finally if `b` is 1, then `p3` calls `use(a)`.
- **Cache coherence** is making sure that in a multicore or multiprocessor environment, if two or more caches are storing for the same address, then the value stored for that address must be the same. I.e. *changes in one cache must be propagated to other caches*.
- A write should not be considered complete until all cached copies have been updated (or marked as invalid so that they will not be used).
- If one thread executes `p1` and updates the value of `a` and the change may propagate to `p2`'s cache before it propagates to `p3`'s cache. If `p2`'s update of `b` arrives sooner, then `p3` will see the old value of `a` and the new value of `b`.
- The updates to `a` and `b` can overlap from T3's perspective. Each thread has their own view of the variables and executes their instructions in sequential order but none has the global view of the three threads and what they are each executing.

## Correct answers

- Program A: We don't know.
- Program B: We don't know.
- Program C: We don't know.
- Why don't we know?
  - It depends on what machine you use.
  - If a system provides *sequential consistency*, then answers all is “No.”
  - But not all hardware provides sequential consistency,
- Note: Examples and other slide content from [\[Adve & Gharachorloo\]](#)

# Sequential Consistency

**Sequential consistency** means 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]

- This definition boils down to two requirements:
  1. Maintaining *program order* on individual processors.
  2. Ensuring **write atomicity** i.e. that if any processor reads the result of a write, then all subsequent reads by any other processor will see the same result.
- Without sequential consistency, multiple processorss can be “worse” than preemptive threads.
  - May see results that cannot occur with any interleaving of threads on a single processor.
- Why doesn't all hardware support sequential consistency?

## Sequential consistency thwarts hardware optimizations

- Complicates write buffers
  - E.g., read  $\text{flag}[n]$  before  $\text{flag}[3 - n]$  written through in [Program A](#) (for  $n = 1, 2$ ).
- 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**: moving code around to improve performance, e.g. swap the order of the two add instructions to avoid a load-use hazard.

```
lw $1, 0($30)
add $2, $2, $1
add $30, $30, $4
```

- Keeping a value in a register.
  - Collapse multiple loads/stores of same address into one operation
- Common subexpression elimination, e.g.  $(n+1) * (n+1)$ 
  - Calculate value  $n+1$  once and use it twice.
  - Could cause memory location to be read fewer times

# SC thwarts compiler optimizations

- **Loop blocking**

- Re-arrange loops for better cache performance by trying to keep the inner loop entirely in cache
- Rather than a single loop from 1 to 10,000, create a sequence of loops from 1 to 1000, 1001 to 2000, etc.

- **Software pipelining**

- Move instructions across iterations of a loop to overlap instruction latency (keep instructions in cache) with branch cost.

```
for (i = 0; i<N; i++) {  
    f(i);  
    g(i);  
}
```

```
for (i = 0; i<N; i += 2) {  
    f(i);  
    f(i+1);  
    g(i);  
    g(i+1);  
}
```

## x86 consistency [Intel SDM 3A, §8.2]

- x86 supports multiple consistency/caching models.
  - Memory Type Range Registers (MTRR) specify consistency for a few (e.g. 3) ranges of physical memory (e.g., frame buffer) It is an older method.
  - Page Attribute Table (PAT) is a newer method that allows control for each 4K page,
- Choices include:
  - **WB:** **Write-back** caching (the default) - initially, just write to the cache and mark the portion as dirty. Write back to main memory when the item is about to be removed from the cache. Tradeoff: more complex to implement.
  - **WT:** **Write-through** caching - all writes are immediately written back to main memory. Tradeoff: writes take longer.
  - **UC:** **Uncacheable** (for device memory) - i.e. a key gets pressed on the keyboard.
  - **WC:** Write-combining - writes are temporarily stored in a buffer and occur in a burst.

## x86 consistency [Intel SDM 3A, §8.2]

- Write-combining has weak consistency and no caching.
- **weak-consistency**: no guarantee that reads and writes will have sequential consistency.
- Write-combining can be used for frame buffers, when sending a lot of data to graphics processing unit (GPU) for rendering video.
- Some instructions have weaker consistency.
  - Some *string instructions* can be re-ordered. E.g. when storing the string “I love CS360 !!!” at location 0x1000 0000, it does not matter if you store the first 8 bytes at 0x1000 0000 and then the second 8 bytes at 0x1000 0008 or vice versa. Same for video.
  - *Non-temporal instructions*, e.g. `movntq` is a version `movq` move instruction that bypass cache and can be re-ordered with respect to other writes.
  - the `nt` part of the instruction is called an *non-temporal hint* that the data will not be used again for a while so there is no need to cache it.

# x86 WB consistency

- Old x86s (late 80's, early 90s) had almost sequential consistency.
  - 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 processor *read its own writes early (store-to-load forwarding)*

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

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

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. I.e. *g* gets the old value of the flag.
- Older processors would wait at “f = ...” until store complete

# x86 atomicity

- **lock** – prefix *makes the memory instruction that follows it atomic (cannot be interrupted).*
  - Usually locks the memory bus for duration of instruction (expensive!).
  - Can avoid locking if memory already exclusively cached (only in one cache).
  - All lock instructions totally ordered, i.e. will happen in a sequential order.
  - Other memory instructions cannot be re-ordered with locked ones
- **xchg** – exchanges the value stored in a register with the value stored in main memory.  
**xchg** is always locked (even without the lock prefix).
- **cmpxchg** – compares two values and exchanges if they are different.  
It is also locked (even without the lock prefix.)

# x86 atomicity

- Special *fence assembly language instructions can prevent re-ordering* from one side of (i.e. above) the fence to the other side (below the fence).
  - **lfence** (load fence) – *waits until all prior loads (i.e. reads) from memory are complete before executing the next instruction.*
    - No load instructions will be moved from before the **lfence** instruction to after it.
  - **sfence** (store fence) – *waits until all prior stores (i.e. writes) to memory complete before executing the next instruction.*
    - No store instructions will be moved from before the **sfence** instruction to after it.
    - E.g. could use after non-temporal store to **data**, before setting a **ready** flag in Program **B**.
  - **mfence** (memory fence) – waits for all prior reads and writes. It is a combination of **lfence** and **sfence**.

## Assuming sequential consistency

- Often we reason about concurrent code assuming sequential consistency
- But for low-level code, *know your memory model!*
  - May need to sprinkle barriers instructions into your source
- For most code, avoid depending on memory model
  - Idea: If you obey certain rules ([discussed later](#))
    - ... system behavior should be indistinguishable from sequential consistency.
- Let's for now say we have sequential consistency
- Example concurrent code: Producer/Consumer with at least one thread for each.
  - `buffer` stores at most `BUFFER_SIZE` items, one per slots.
  - `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).

# Producer/Consumer

```
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);
    }
}
```

- What can go wrong in above two threads (even with sequential consistency)?

# Data races

- `count` may have the wrong value just like our multithreaded `add()` `sub()` example.
- Possible implementation of incrementing or decrementing `count` and `count-`.

`register←count`

`register←register + 1`

`count←register`

`register←count`

`register←register - 1`

`count←register`

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

`register←count`

`register←register + 1`

`register←count`

`register←register - 1`

`count←register`

`count←register`

A **critical section** is the code that accesses a shared variable (or more generally, a shared resource) and must not be concurrently executed by more than one thread.

## 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 atomic on multiprocessor!
  - It will still have the same race condition.
  - Can potentially make it atomic with the `lock` prefix.
  - But `lock` is very expensive so compilers won't generate it by default. It assumes you don't want penalty.
- Need solution to critical section problem: i.e. *protect critical sections from concurrent execution.*

## Desired properties of solution

- *Mutual Exclusion*
  - Only one thread can be in critical section at a time.
- *Progress*
  - If no thread currently in the critical section and one of the threads is trying to enter, it will eventually get in.
- *Bounded waiting*
  - Once a thread 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 the critical section, we don't have progress.
  - If thread *A* waiting to enter while *B* repeatedly leaves and re-enters *ad infinitum*, we 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`
- `bool wants[2]; // wants[i] indicates if  $T_i$  wants to enter`

- 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 it is not our turn */;
    critical_section();
    wants[i] = false;
    remainder_section();
}
```

## Does Peterson's solution work?

- Mutual exclusion – can't both be in the critical section.
  - Would mean `wants[0] == wants[1] == true`, so `not_turn` would have blocked one thread from entering the critical section.
- Progress – If  $T_{1-i}$  not in critical section, it can't block  $T_i$ 
  - Means `wants[1-i] == false`, so  $T_i$  won't loop.
- 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.

# Mutexes

- *Peterson's solution is expensive:* it uses busy waiting.
  - It only works for 2 processes but it can generalize to  $n$ , but for some fixed  $n$ .
- Must adapt to machine memory model if not sequentially consistent.
  - Ideally you want your code to be able to run everywhere.
- 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 can acquire **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.
  - The exception is initialization, before exposed to other threads.
  - Protection is the responsibility of the application writer
- *Compiler/Runtime Contract* (C, Java, Go, etc.): assuming no data races, the program behaves sequentially consistent.
- If you use mutexes properly, behavior should be indistinguishable from sequential consistency.
  - It is the responsibility of the threads package and compiler.
  - Mutex is broken if you use properly and don't see sequential consistency.
- OS kernels also need synchronization
  - Some mechanisms look like mutexes
  - But interrupts complicate things (incompatible with mutexes).

# PThread Mutex API

- Function names in this lecture all based on pthreads
- `int pthread_mutex_init(pthread_mutex_t *m,  
 pthread_mutexattr_t attr)`
  - Initialize a mutex
- `int pthread_mutex_destroy(pthread_mutex_t *m)`
  - Destroy a mutex
- `int pthread_mutex_lock(pthread_mutex_t *m)`
  - Acquire a mutex
- `int pthread_mutex_unlock(pthread_mutex_t *m)`
  - Release a mutex
- `int pthread_mutex_trylock(pthread_mutex_t *m)`
  - Attempt to acquire a mutex
  - Return 0 if successful, otherwise -1 (`errno == EBUSY`)

# 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); /* <--- Why? */
            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);
            thread_yield();
            mutex_lock(&mutex);
        }

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

        consume_item(nextConsumed);
    }
}
```

# Improved Producer and Consumer

- The global variables `count`, `in`, `out` and `buffer` are accessed by the producer and the consumer, so they should be protected by a mutex.
- If `count` is 0, then release the `mutex` and call `thread_yield` to allow another thread to run.
- Acquire `mutex` again to check the value of `count` and possibly access the buffer.
- While waiting for the `mutex` the thread does not use the processor but instead waits in a wait queue and the kernel wakes it up when the other thread has unlocked the the `mutex`.
- However each thread is continually checking the value of `count`, releasing the `mutex`, yielding, and tryto to lock the `mutex` again, which does use up processor time.

# Condition variables

- *Busy-waiting in an application is inefficient.*
  - Thread consumes processor time even when can't make progress.
  - This approach unnecessarily slows other threads and processes.
- Better to inform scheduler of which threads can run
- Typically done with **condition variables**
- `int pthread_cond_init(pthread_cond_t *, ...);`
  - Initialize with specific attributes
- `int pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);`
  - Atomically unlock `m` and sleep until `c` signaled
  - Then re-acquire `m` and resume executing
- `int pthread_cond_signal(pthread_cond_t *c);`  
`int pthread_cond_broadcast(pthread_cond_t *c);`
  - Wake one/all threads waiting on `c`

# Using Condition Variables

- Condition variables get their name because they *allow threads to wait for arbitrary conditions to become true* inside of a critical section.
- Normally, each condition variable corresponds to a particular condition that is of interest to an application.
- E.g. in the bounded buffer producer/consumer example on the following slides, the two conditions are:
  - `count` > 0 (there are items in the buffer)
  - `count` < BUFFER\_SIZE (there is free space in the buffer)
- When a *condition is not true*, a thread can `cond_wait` on the corresponding condition variable, unlocking the `mutex` until the condition becomes true.
- When another thread detects that a *condition is true*, it uses `cond_signal` or `cond_broadcast` to notify any blocked threads.
- Note that signalling (or broadcasting to) condition variable that has no blocked threads has *no effect*. ⇒ Signals do not accumulate.

# 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 code fails when another consumer,  $C_2$ , grabs item and empties buffer before  $C_1$  wakes up.

$C_1$	$C_2$	$P$
cond_wait (...);		mutex_lock (...);
		⋮
	mutex_lock (...);	count++;
if (count == 0)		cond_signal (...);
⋮		mutex_unlock (...);
	use buffer[out] ...	
	count-;	
	mutex_unlock (...);	
use buffer[out] ...		← No items in buffer

## Condition variables (continued)

- Why must `cond_wait` both release mutex and 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 and 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);
```

# Monitors [BH][Hoar]

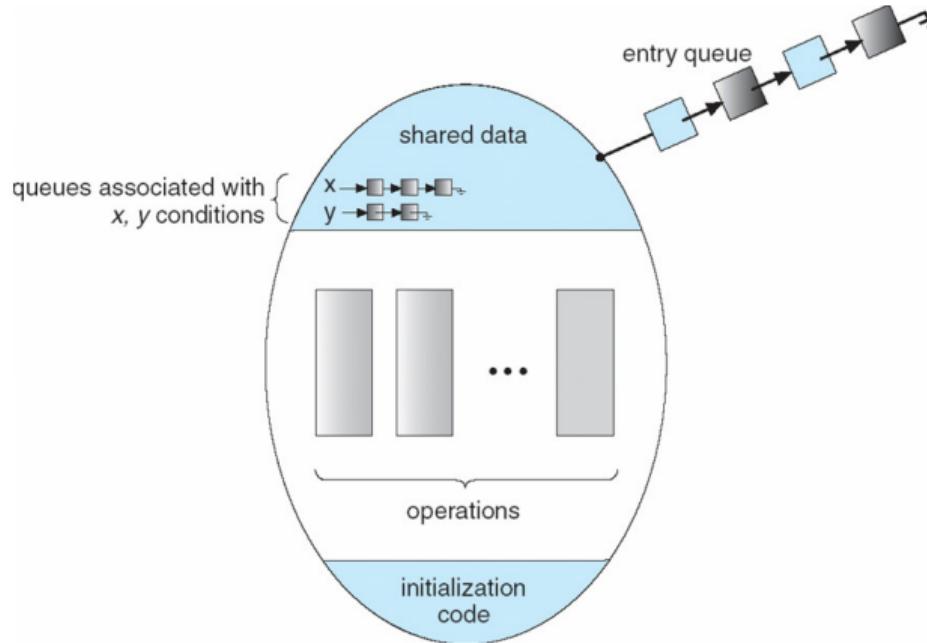
- **Monitors** are a programming language construct
  - *Essentially, it is a class where only one procedure executes at a time.*
  - Possibly less error prone than raw mutexes, but less flexible too.

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { ... }
    ...
    procedure Pn (...) { ... }

    Initialization code (...) { ... }
}
```

- Can implement a mutex with a monitor or vice versa.
  - But monitor alone doesn't give you condition variables.
  - Need some other way to interact with scheduler.
  - Use **conditions**, which are essentially condition variables.

# Monitor implementation



- Queue of threads waiting to get in
  - Might be protected by **spinlock** (an OS primitive used to implement mutexes, etc)
- Queues inside are associated with conditions.

# Semaphores [Dijkstra]

- A **Semaphore** is initialized with an integer  $n$ 
  - `int sem_init(sem_t *s, ... , unsigned int n);`
- Provides two functions:
  - `sem_wait(sem_t *s)` (originally called `P()`)  
*If the semaphore value  $n$  is greater than 0, decrement it.*  
Otherwise, wait until  $n > 0$  and then decrement it.
  - `sem_signal(sem_t *s)` (`sem_post` in pthreads, originally called `V()`)  
*Increment the value of the semaphore by 1.*
- Operation: `sem_wait` will return only  $n$  more times than `sem_signal` called. Instead it will wait until `sem_signal` is called again.

## Semaphores [Dijkstra]

- Example: If `n` is 1, then semaphore is a mutex with `sem_wait` as lock and `sem_signal` as unlock.
- Semaphores give elegant solutions to some problems.
  - In the following slide, semaphores replace `count` in the Producer / Consumer problem.
  - Recall that `count` had to be between 0 and `BUFFER_SIZE`.
  - `empty` counts *how many spots are available* (i.e. how empty it is) and the `producer` has to wait if there are 0 empty spots available.
  - `full` counts *how many spots have items* (i.e. how full it is) and the `consumer` has to wait if there are 0 items available.
  - The code assumes only one `producer` and one `consumer` otherwise you would need to protect the reading and writing of `in` and `out` by mutexes.

# Semaphore producer/consumer

- Initialize `full` semaphore to 0: block consumer when buffer empty.
- Initialize `empty` semaphore to `BUFFER_SIZE`: 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);
    }
}
```

# Approaches to Concurrency - Summary

- Monitors are a class where *only one procedure can be executed at a time*.
- Locks or mutexes *have two states*:
  - `mutex_lock` which allows a thread (executing any function) into a critical section to access a resource.
  - `mutex_unlock` which the thread calls when leaving the critical section, so another thread is allowed access.
- Semaphores *have many states*, i.e. it counts from `0` to `n`
  - `sem_wait` will wait if the semaphore value is `0`.
  - `sem_signal` or `sem_post` increment the semaphore value.
- Condition variables can *wait for an arbitrary condition to be true*, e.g. a complex Boolean expression.
  - `cond_wait` will wait if the condition is not met.
  - `sem_signal` or `sem_post` wake up threads waiting for the condition to be true.

# Benign Race Conditions

- Sometimes *allowing for race conditions provides for greater efficiency.*
- E.g. in cases where there is more concern about speed than accuracy.
  - I.e. you are only looking for an approximate—not an exact—answer.

```
hits++; // each time someone accesses web site
```
- When you know you can get away with race condition.
  - In the code below, it is inexpensive to check if initialization has happened.
  - The result could be inaccurate because the value could change as it is checked.
  - However, if it seems uninitialized, then you pay the price of locking the mutex and if it is still uninitialized, then execute the initialization routine.

```
if (!initialized) {  
    mutex_lock (m);  
    if (!initialized) { initialize(); initialized = 1; }  
    mutex_unlock (m);  
}
```

# Detecting data races

- Static methods, e.g. detected by the compiler (hard).
- Debugging is painful because the race might occur rarely.
- Instrumentation—modify the program to trap (i.e. check) memory accesses.
- **Lockset algorithm** (eraser) is particularly effective:
  - For each global memory location, keep a **lockset**  
e.g. the vector  $(1, 1, 1, 1)$  if there are four locks in all.
  - On each access of a particular global memory location, say **total**, note any locks that are not currently held.  
E.g. if only locks 0 and 2 are being held at that time the vector becomes  $(1, 0, 1, 0)$ .
  - *If the lockset becomes empty  $\Rightarrow$  there is no mutex protecting this memory location.*
  - This approach is simple and it catches potential races even if they haven't yet occurred.

# Synchronization: Motivation

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

Converges to  $T(1)B$  as  $n$  goes to infinity

- **Amdahl's Law** guides the *efficiency gains when using multiple cores.*
  - $T(1)$ : the time one core takes to complete the task
  - $B$ : the fraction of the job that must be serial
  - $n$ : the number of cores
- Suppose  $n$  were infinity!
- Amdahl's law places an ultimate limit on parallel speedup
- Problem: synchronization increases serial section size.
- **Scalable Commutativity Rule:** *"Whenever interface operations commute, they can be implemented in a way that scales"* [Clements]

# Locking Basics

```
pthread_mutex_t m;  
  
pthread_mutex_lock(&m);  
cnt = cnt + 1; /* critical section */  
pthread_mutex_unlock(&m);
```

- Only one thread can hold a lock at a time. I.e. if two threads call `pthread_mutex_lock` the function will return immediately for one of them and the other will block (i.e. wait).
- This function makes the critical section atomic by allowing only one thread at a time to increment `cnt`.
- When do you need a lock?
  - *Anytime two or more threads touch data and at least one writes.*
- Rule: Never touch (global) data unless you hold the right lock.

# Fine-grained Locking

```
struct list_head *hash_tbl[1024];

/* Coarse-grained Locking */
mutex_t m;
mutex_lock(&m); /* a single lock for the whole hash table */
struct list_head *pos = hash_tbl[hash(key)];
/* walk list and find entry */
mutex_unlock(&m);

/* Fine-grained Locking */
mutex_t bucket[1024]; /* a separate lock for each linked list */
int index = hash(key);
mutex_lock(&bucket[index]);
struct list_head *pos = hash_tbl[index];
/* walk list and find entry */
mutex_unlock(&bucket[index]);
```

- Which of these is better?

# Fine-grained Locking

- On the previous slide we have hash table using separate chaining to handle collisions.
  - I.e. if two differ values hash to the same location, we add them to a linked list.
  - There is one linked list per table location.
- With **coarse-grained locking** we create one lock for the whole data structure.
- With **fine-grained locking** we create one lock for each linked list in the data structure.
- Coarse-grained locking makes the most sense if the data structure is not used that often since it uses less space and less complexity as the thread only has to acquire one lock.
- Fine-grained locking makes the most sense if the data structure is being used often and maximizing concurrency and performance is the goal.
- Fine-grained locking could also require a thread to acquire many locks to perform a task, which would slow it down.
- A general approach is to *identify areas where there is a lot of contention* (threads often waiting for a particular resource) and use fine-grained locking in those areas.

# Hardware-Specific Synchronization Instructions

- Goal: implement a mutex function atomically. I.e. given the address of a mutex, between when the `*mutex` is *tested* and *set*, *no other thread can change its value*.
- Called **test-and-set**

```
while (*mutex == true){}; // test if mutex is held  
*mutex = true;           // set it to true
```

- Recall: x64 processors provide an atomic operation used to implement synchronization primitives such as mutexes, call `xchg`.
- Given a source register `src` and an address in memory `addr`  
`xchg src, addr`  
swaps the value stored in register `src` with the value stored at the address `addr` atomically.

## Hardware-Specific Synchronization Instruction: xchg

```
mutex_lock(bool *mutex) {  
    while (xchg(mutex, true) == true) {}; // test and set  
}  
  
mutex_unlock(bool *mutex) {  
    *mutex = false; // give up mutex  
}
```

- Here **true** means the mutex (and the critical section) is locked and **false** means unlocked.
- If **xchg** returns **true**, then the mutex was already set and you have not changed its value  
⇒ continue to busy-wait.
- If **xchg** returns **false**, then the mutex was free and you have changed its value to **true** ⇒  
you have just locked the mutex.
- This construct is known as a spinlock since a thread busy-waits (loops or spins) in  
**mutex\_lock** until it is available.

# Memory reordering danger

- Suppose no sequential consistency guarantee and the programmer did not compensate.
- Multicore hardware could violate the program order between setting the `v->lock` to 1 and incrementing `v->val`.

Program order on CPU #1	View on CPU #2
read/write: <code>v-&gt;lock = 1;</code> read: <code>register = v-&gt;val;</code> <b>write: <code>v-&gt;val = register + 1;</code></b> write: <code>v-&gt;lock = 0;</code>	<code>v-&gt;lock = 1;</code>  <code>v-&gt;lock = 0;</code>  <i>/* danger */</i> <b><code>v-&gt;val = register + 1;</code></b>

- If `atomic_inc` called at */\* danger \*/*, bad `v->val` ensues because CPU #2 had not received the update yet.

# Ordering requirements

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

Ensure that all cores know to:

1. set lock before read and writes (uses xchg)
2. unlock after reads and writes (fence instruction)

- Must ensure all cores see the following:
  1. `v->lock` was set *before* `v->val` was read and written
  2. `v->lock` was cleared *after* `v->val` was written
- How does #1 get assured on x64?
  - Test-and-set uses `xchg` which is always “locked,” ensuring a barrier.
- How to ensure #2 on x64?
  - Might need a `fence` instruction after (the possibly) non-temporal stores.

# Spinlocks in CasterOS

- A **spinlock** is a lock that **spins**, i.e. it repeatedly tests the lock's availability in a loop until the lock is obtained.
- When threads uses a spinlock they **busy-wait** i.e. they *use the processor while they wait* for the lock.
- Spinlocks are efficient for short waiting times (a few instructions).
- They are used by CasterOS.
- The implementation uses `atomic_swap_uint64`, which in turn is using the assembly language instruction `xchgq`, which is a variant of `xchg` that swaps quad words (i.e. 64 bits)  
[castoros/sys/amd64/include/atomic.h](#)
- *Interrupts are disabled on the processor that holds the spinlock* and the thread will only execute a few instructions before the spinlock is released. ⇒ Therefore any other threads busy-waiting for the spinlock will only spin for a short time.

# Spinlocks in CasterOS

```
void Spinlock_Lock(Spinlock *lock)
{
    /* Disable Interrupts */
    Critical_Enter();

    while (atomic_swap_uint64(&lock->lock, 1) == 1)
        /* Spin! */;

    lock->cpu = CPU();
}

void Spinlock_Unlock(Spinlock *lock)
{
    ASSERT(lock->cpu == CPU());

    atomic_set_uint64(&lock->lock, 0);

    /* Re-enable Interrupts (if not spinlocks held) */
    Critical_Exit();
}
```

# Atomics and Portability

- A key *challenge* of multithreaded code is that there are lots of *variation in atomic instructions, consistency models, and compiler behaviour.*
- This variation results in complex code when writing portable kernels and applications.
- It is still a big problem today: Servers and Windows desktops and laptops are typically x64. Smart phones and tablets are ARM. They use two different models.
  - x64 has a complex instruction set (CISC) with Total Store Order Consistency i.e. a thread can read its own writes right away and they becomes *visible to all other threads simultaneously.*
  - ARM has a reduced instruction set (RISC) with Relaxed Consistency i.e. reads and write can occur out of order and possibly in *different orders for different threads.*
- Fortunately, the C11 standard has builtin support for atomics.
  - Enable in GCC with the `-std=c11` flag.
- Also available in C++11, but not discussed today...

# C11 Atomics: Basics

- C11 added the following support for synchronization.
- New atomic type: e.g., `_Atomic(int) foo`.
  - Can wrap most basic types with `_Atomic(...)` and they become atomic.
  - All standard ops (e.g., `+`, `-`, `/`, `*`) become sequentially consistent.
  - Provide C function calls, e.g. `atomic_compare_exchange_strong`, to replace the need to use assembly language instructions, e.g. `cmpxchg`.
- `atomic_flag` is a special type
  - Atomic boolean value without support for loads and stores. It provides `atomic_flag_test_and_set` and `atomic_flag_clear` instead.
  - Must be implemented lock-free, i.e. thread cannot be preempted in the middle of executing this instruction.
  - All other types might require locks, depending on the size and architecture.
- Fences also available to replace hand-coded memory barrier assembly.

# Memory Ordering

- Several choices of memory ordering for fences are available.
  1. `memory_order_relaxed`: no specific memory ordering is required.
  2. `memory_order_consume`: a slightly relaxed version of `memory_order_acquire`
  3. `memory_order_acquire`: reads and writes to memory that *occur after the load cannot be reordered before it*. E.g. all the reads and write that occur after the lock is acquired cannot be moved before the lock is acquired.
  4. `memory_order_release`: reads and writes to memory that *occur before the store cannot move after it*. So all the writes to a `val` must be completed before we release (i.e. store 0) the lock.
  5. `memory_order_acq_rel`: combination of the previous two.
  6. `memory_order_seq_cst`: full sequential consistency
- A `memory_order_consume` can always be replaced by a `memory_order_acquire`.

# Memory Ordering

- The ones we will use are the `memory_order_relaxed`, `memory_order_acquire` to acquire a lock, `memory_order_release` for releasing a lock and `memory_order_seq_cst`.
- Note that these memory ordering choices only apply to the thread that used them.
- What happens if the chosen model is mistakenly too weak? race conditions
- What happens if it is mistakenly too strong? reduced concurrency  $\Rightarrow$  reduced efficiency.
- Suppose thread 1 `releases` and thread 2 `acquires`
  - Thread 1's preceding `writes` can't move past the `release` store.
  - Thread 2's subsequent `reads` can't move before the `acquire` load.
  - Warning: other threads might see a completely different order.

## Example 1: Atomic Counters

```
_Atomic(int) packet_count;  
  
void recv_packet(...) {  
    ...  
    atomic_fetch_add_explicit(&packet_count, 1,  
        memory_order_relaxed);  
    ...  
}
```

- Suppose we want to keep a count of a particular event, say the number of packets of data that arrive at your Wi-Fi card.
- You would use `memory_order_relaxed` when you don't care *when* the increment happens (it can be moved elsewhere in the code) *as long as it does happen*.

## Example 2: Producer, Consumer

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

void send(struct message *m) {
    msg_buf = *m;
    atomic_thread_fence(memory_order_release); // fence
    atomic_store_explicit(&msg_ready, 1,
        memory_order_relaxed); // store
}

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

## Example 2: Producer, Consumer

- The code on the previous slide implements a message passing service from one thread to another (as in the Go language).
- `send` makes a copy of the message.
  - Then `send` asserts `memory_order_release` so the *preceding write* to `msg_buf` *will not be reordered* to occur after the `atomic_store` fence.
  - Once that happens we can be relaxed about when the `msg_read` flag is set because the fence is taking care of the ordering.
- `recv` receives the message.
  - The `atomic_load` function of `recv` synchronizes with the `atomic_store` function of `send` and receives the new value of `msg_buf` if `msg_ready` is 1 or `NULL` if there is no message.
  - The `memory_order_acquire` assertion ensures that the *following read* from `msg_buf` *will not be reordered* to occur before the `memory_order_acquire` fence ensuring the timeliness of the message.

## Example 3: A Spinlock

- **Spinlocks** are similar to Mutexes.
- Kernels use these for small critical regions, e.g. updating the value of a variable.
  - The calling busy-waits for the thread holding the spinlock to release it.
  - There is no waiting (sleeping) and yielding to other threads while holding the spinlock.

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

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

# Cache Coherence: the hardware view

- **Coherence**
  - Concerns accesses to a single memory location (in the caches of different cores).
  - Ensures that stale (out-of-date) copies do not cause problems.
- **Consistency**
  - Concerns apparent ordering between multiple locations (e.g. memory ordering model).

# Multicore Caches

- Performance requires multilevel caches (L1, L2, etc.) since access to RAM is relatively slow.
- Caches create an opportunity for cores to disagree about what value is stored at a particular address.
- Bus-based approaches (older)
  - “Snoopy” protocols, each core listens to memory bus.
  - The core writing the value uses write through caching and the other cores invalidate their values for that address when they see a write to the same address from another core.
  - Problem: *Bus-based schemes limit scalability.*
- *Modern processors use networks*, e.g., hypertransport (older AMD), Infinity Fabric (newer AMD), UPI (Intel).
- Caches are divided into chunks of bytes called **cache lines**.
  - 64-bytes is a typical cache-line size.
  - This is the amount of data that is transferred to different caches or main memory to take advantage of locality of reference.

# 3-state Coherence Protocol: Modified, Shared, Invalid (MSI)

- Each cache line is one of three states:
- **Modified** (sometimes called Exclusive)
  - One cache has a valid copy.
  - That copy is called **dirty** (needs to be written back to main memory).
  - Out of date copies in other caches are called **stale**.
  - Must invalidate all other copies before entering this state.
- **Shared**
  - One or more caches (and memory) have a valid copy.
- **Invalid**
  - Doesn't contain the data.
- Transitions can take 100 (on smaller machines) to 2000 clock cycles on larger ones.

# Core and Bus Actions

- Each core has three possible actions that affect the caches.
- Read (load)
  - *Read without intent to modify*, data can come from memory or another cache
  - Cache line enters shared state
- Write (store)
  - *Read with intent to modify* must invalidate all other cache copies
  - Cache line is shared (some protocols have an exclusive state)
- Evict
  - *Writeback contents to memory if modified.*
  - Discard if in shared state.
- Performance problem:
  - Every transition requires bus communications.
  - Avoid state transitions whenever possible.

# Implications for Multithreaded Design

- Lesson #1: *Avoid false sharing.*
  - Processor shares data in cache line chunks
  - Avoid placing data used by different threads in the same cache line, e.g. make arrays start and stop on a 64B boundary.
- Lesson #2: *Align structures to cache lines.*
  - Place related data you need to access together.
  - Alignment in C11/C++11:
- Lesson #3: *Pad data structures*
  - Arrays of structures lead to false sharing.
  - Add unused fields to ensure alignment.
- Lesson #4: *Avoid contending on cache lines*
  - Reduce costly cache coherence traffic.
  - Advanced algorithms spin on a cache line local to a core (e.g., MCS Locks).

## Real World Coherence Costs

- See [David] for a great reference, summarized here...
  - Access time for an Intel Xeon processor caches and RAM:  
3 cycles for L1 cache, 11 cycles for L2, 44 cycles for last level cache (LLC),  
355 cycles for RAM
- For different cores (on the same processor)
  - load: 109 cycles (LLC) + 65
  - store: 115 cycles (LLC + 71)
  - atomic compare-and-swap (CAS): 120 cycles (LLC + 76)
  - recall that for store you update the value and so values in other caches must be invalidated.

# Real World Coherence Costs

- Servers can have multiple processors each with multiple cores, e.g. our [linux.student.cs](#) servers.
- **Non-uniform memory access (NUMA)** each processor has local memory and can access its memory faster than memory local to another processor or shared between processors.
- For cores accessing data on different processors:
  - NUMA load: 289 cycles
  - NUMA store: 320 cycles
  - NUMA atomic compare-and-swap (CAS): 324 cycles
- But this is only a partial picture
  - Could be faster because of out-of-order execution
  - Could be slower because of bus contention or multiple hops

# Improving Spinlocks

- Test-and-set spinlock has several advantages
  - Simple to implement and understand.
  - Easy to manage: one memory location for arbitrarily many processors and cores.
- But also has disadvantages
  - Lots of traffic over memory bus (especially when > 1 thread spinning).
  - Not necessarily fair (same core acquires lock many times)
  - Even less fair on a NUMA machine.
- Idea 1: Avoid spinlocks altogether
  - Lock free algorithms, read-copy-update (RCU) , transactional memory (e.g. like a database).
- Idea 2: *Reduce bus traffic with better spinlocks.*
  - Design lock that spins only on local memory.
  - Also gives better fairness.

# MCS lock

- Idea 2: Build a better spinlock, e.g. lock designed by Mellor-Crummey and Scott
- *Goal:* reduce bus traffic on cache-coherent machines and improve fairness.
- Each processor has a `qnode` structure in local memory.
- With `next` it can point to another `qnode` that is not in local memory.

```
typedef struct qnode {  
    struct qnode *next;  
    bool locked;  
} qnode;
```

- Local can mean local memory in NUMA machine
- Or just its own cache line that gets cached in exclusive mode.
- *While waiting, spin on your local locked flag.*

# MCS lock

- A lock is a pointer to a `qnode`.

```
typedef qnode *lock;
```

- A `lock` is used to point to the tail of a list of processors holding or waiting to hold the MSC (spin) lock.
- When `L` is `NULL` then the lock is available (i.e. nothing in line holding or waiting).
- Recall that spinlocks are held by processors.
- When a thread tries to acquire the lock, it will add itself to the list.



# MCS Acquire

```
1  acquire(lock *L, qnode *I) {  
2      I->next = NULL;  
3      qnode *predecessor = I;  
4      XCHG(predecessor, *L); /* atomic swap */  
5      if (predecessor != NULL) {  
6          I->locked = true;  
7          predecessor->next = I;  
8          while (I->locked)  
9              /* spin */;  
10     }  
11 }
```

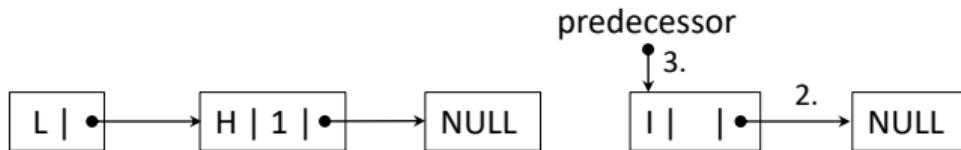
- Lines 2–4: If `I` want to obtain the lock it will add myself to the end of the list by setting my `next` pointer to `NULL` have the `lock` point to me and `predecessor` point to whatever the lock was pointing to.
- Line 5: If the `lock` was pointing to `NULL`, then I have obtained the `lock`.



# MCS Acquire

```
2     I->next = NULL;  
3     qnode *predecessor = I;  
4     XCHG(predecessor, *L); /* atomic swap */
```

- Lines 2–3: If the `L` is not `NULL`, then it points to the `qnode` at the end of the queue, say `H`, that is either holding the lock or the last in line to hold it.



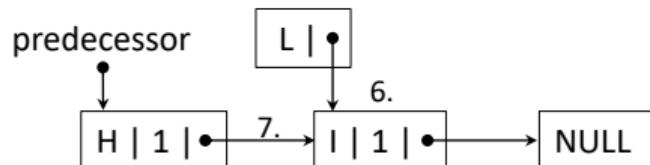
- Line 4: Now `L` points to `H` and `predecessor` points to `I`.



# MCS Acquire

```
5     if (predecessor != NULL) {  
6         I->locked = true;  
7         predecessor->next = I;  
8         while (I->locked)  
9             /* spin */;  
10    }
```

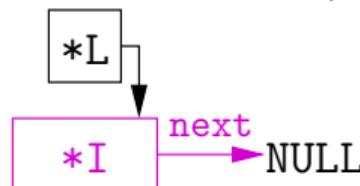
- Lines 6–7: `I` is marked as waiting for its predecessor in the queue, `H`. And `H` now points to `I`, i.e. `]mgtl` is next in line to hold the lock.
- Lines 8–9: `I` now spins until `H` releases the lock.



# MCS Release

```
1.     release(lock *L, qnode *I) {
2.         if (!I->next)
3.             if (CAS(*L, I, NULL)) /* atomic compare-and-swap */
4.                 return;
5.         while (!I->next)
6.             /* spin */;
7.         I->next->locked = false;
8.     }
```

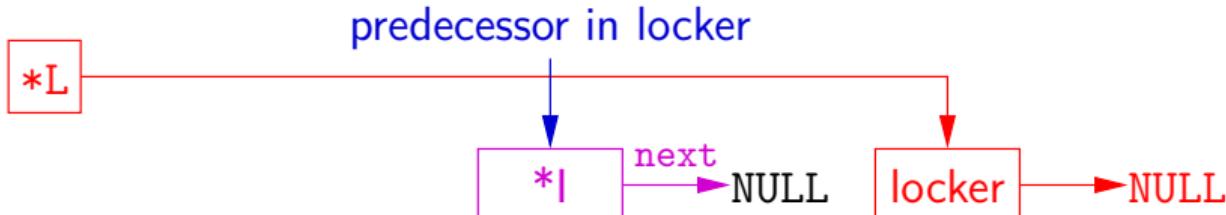
- Line 2: If `I` points to `NULL`.
- `CAS` is compare-and-swap.
- Lines 3–4: If `*L == I` then swap `I` with `NULL` and return.
- I.e. if `*L == I` then `I` is the last `qnode` in the queue so there is nothing waiting for the lock, so swap `I` with `NULL`, i.e. now `*L == NULL`, i.e. the queue is now empty.



# MCS Release

```
1.     release(lock *L, qnode *I) {
2.         if (!I->next)
3.             if (CAS(*L, I, NULL)) /* atomic compare-and-swap */
4.                 return;
5.         while (!I->next)
6.             /* spin */;
7.         I->next->locked = false;
8.     }
```

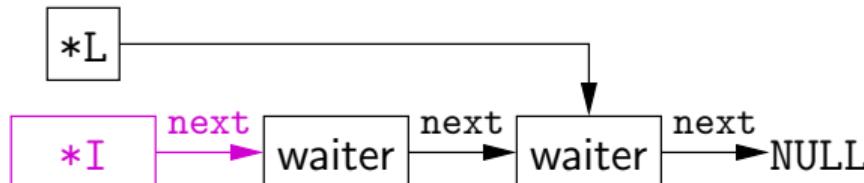
- Line 3–6: If `*L != I` then do not swap and return. If `I->next` is `NULL` then ...
  - Another thread is in the middle of `acquire`
  - Just wait for `I->next` to be non-`NULL`



# MCS Release

```
1.     release(lock *L, qnode *I) {
2.         if (!I->next)
3.             if (CAS(*L, I, NULL)) /* atomic compare-and-swap */
4.                 return;
5.         while (!I->next)
6.             /* spin */;
7.         I->next->locked = false;
8.     }
```

- Line 7: If `I->next` is non-NULL then there are other `qnodes` waiting for the lock.
  - `I->next` is the oldest waiter, so let it proceed by setting `I->next->locked = false`, i.e. it will now stop spinning and proceed to acquire the lock.



# The deadlock problem

```
mutex_t m1, m2;

void f1(void *ignored) {
    lock(m1);
    lock(m2);
    /* critical section */
    unlock(m2);
    unlock (m1);
}

void f2 (void *ignored) {
    lock(m2);
    lock(m1);
    /* critical section */
    unlock(m1);
    unlock(m2);
}
```

- *It is dangerous to acquire locks in different orders:* if T1 executes `f1` exactly when T2 executes `f2` then T1 is wait for T2 to release `m2` and T2 will wait for T1 `m1`. This situation is called **deadlock**, i.e. both threads will wait for ever.

# More deadlocks

- *The same problem can occur with condition variables.*
  - Suppose resource 1 managed by  $c_1$ , resource 2 by  $c_2$
  - T1 has 1, waits on  $c_2$ , T2 has 2, waits on  $c_1$ .
- Or have combined mutex/condition variable deadlock:

```
mutex_t a, b;  
cond_t c;  
lock(a); lock(b); while (!ready) wait(b, c); // releases b but not a  
unlock(b); unlock (a);  
lock(a); lock(b); ready = true; signal(c); // needs a to signal c  
unlock(b); unlock(a);
```

- Lesson: Dangerous to hold locks when crossing abstraction barriers!
  - I.e., top level function holds lock **a** then calls a lower level function that uses condition variables and needs lock **a** and **b** to signal a thread to wake up.

# Deadlock conditions

## 1. *Limited access* (mutual exclusion):

- Resource can only be shared with finite users

## 2. *No preemption of resources*:

- Once resource granted, the resources cannot be taken away.

## 3. Multiple independent requests (*hold and wait*):

- Don't ask all at once  
I.e. wait for next resource while holding current one.

## 4. *Circularity in graph of requests*

- All four are necessary for deadlock to occur,
- Two approaches to dealing with deadlock:
  - Pro-active: prevention
  - Reactive: detection + corrective action

## Prevent by eliminating one condition

### 1. Limited access (mutual exclusion):

- Buy more resources, split into pieces, or virtualize to make "infinite" copies

### 2. No preemption:

- Physical memory: virtualized with VM, can take physical page away and give to another process!

### 3. Multiple independent requests (hold and wait):

- Wait on all resources at once (must know in advance), i.e. do not hold and wait.

### 4. *Circularity in graph of requests*

- Single lock for entire system: (problems?)
- Partial ordering of resources (next)

# Cycles and deadlock

- *View system as graph*
  - Processes, threads, and resources are nodes
  - Resource Requests and Assignments are edges
- *If graph has no cycles → no deadlock*
- If graph contains a cycle
  - Definitely deadlock if only one instance per resource type, e.g. one mutex  $m_1$ .
  - Otherwise, maybe deadlock, maybe not, e.g. multiple pages in RAM.
- Prevent deadlock with *partial order on resources*
  - E.g., always acquire mutexes in a specific order, e.g.  $m_1$  before  $m_2$
  - Statically assert (decide on a) lock ordering for the whole code base (e.g., VMware ESX)
  - Dynamically find potential deadlocks [[Witness](#)]

# Wait Channels

- Mutexes, semaphores, condition variables (but not spinlocks) use wait channels.
  - It *manages a list of sleeping threads*. Each mutex, etc, gets its own wait channel.
  - Works with thread scheduler, i.e. thread waiting for mutex `m1` waits on `m1`'s wait channel.
- `void WaitChannel_Lock(WaitChannel *wc);`
  - *Lock wait channel* for sleep and wake operations.
  - Prevents a race between sleep and wake operations.
- `void WaitChannel_Sleep(WaitChannel *wc);`
  - *Blocks calling thread* on wait channel `wc`, i.e. puts it to sleep.
  - Causes a context switch (e.g., `thread_yield`)
- `void WaitChannel_WakeAll(WaitChannel *wc);`
  - *Unblocks all threads* sleeping on the wait channel
- `void WaitChannel_Wake(WaitChannel *wc);`
  - *Unblocks one thread* sleeping on the wait channel

# Hand-over-Hand Locking

- **Hand-over-hand locking** allows for fine-grained locking, i.e. many locks.
- Useful for concurrent access to a single data structure.
  - Hold at most two locks: the previous lock and the next one: working your way through a sequence of steps.
  - Locks must be ordered in a sequence.
  - You may not go backwards. Why?
- Example: we have locks A, B, C

```
lock(A)
// Operate on A
lock(B)
unlock(A)
// Operate on B
lock(C)
unlock(B)
// Operate on C
unlock(C)
```

# Example Semaphores

- Mutexes, CVs and Semaphores use wait channels and hand-over-hand locking.
- Lock order:
  - first acquire `Spinlock sem_lock`,
  - then `WaitChannel_Lock`,
  - then release `Spinlock sem_lock`.
- While the spinlock `sem_lock` is being held, no other thread can access this struct.
- After `WaitChannel_lock` is called, no other thread can access this wait channel, until the calling thread is put to sleep and the kernel releases the `WaitChannel_lock`.

```
typedef struct Semaphore {  
    int          sem_count;  
    Spinlock    *sem_lock;    // exclusive access to this struct's fields  
    WaitChannel *sem_wchan; // queue where blocked threads wait (i.e. sleep)  
} Semaphore;
```

## Example: Semaphores Implementation

```
Semaphore_Wait(Semaphore *sem) {  
    Spinlock_Lock(&sem->sem_lock);  
    while (sem->sem_count == 0) {  
        /* Locking the wchan prevents a race on sleep */  
        WaitChannel_Lock(sem->sem_wchan);  
        /* Release spinlock before sleeping */  
        Spinlock_Unlock(&sem->sem_lock);  
        /* Wait channel protected by it's own lock */  
        WaitChannel_Sleep(sem->sem_wchan);  
        /* Recheck condition, no locks held */  
        Spinlock_Lock(&sem->sem_lock);  
    }  
    sem->sem_count--;  
    Spinlock_Unlock(&sem->sem_lock);  
}
```

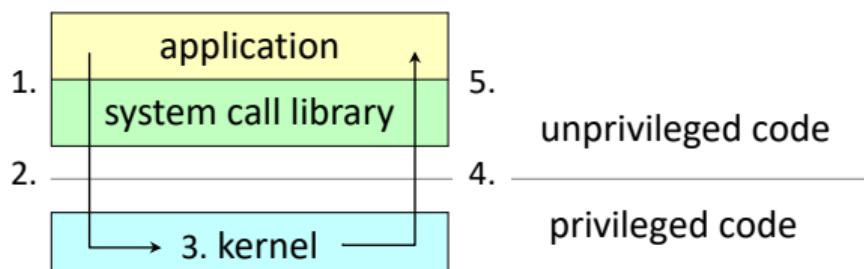
*Decrement the semaphore count xor block if it is 0.*

## Example: Semaphores Implementation

```
Semaphore_Post(Semaphore *sem) {  
    Spinlock_Lock(&sem->sem_lock);  
    sem->sem_count++;  
    WaitChannel_Wake(sem->sem_wchan);  
    Spinlock_Unlock(&sem->sem_lock);  
}
```

- *Increment the semaphore count and wake one thread if it is blocked.*
- The actual spinlock the wait channel uses is abstracted (hidden away) in the calls to `WaitChannel_Lock`, `WaitChannel_Sleep`, `WaitChannel_Wake`, and `WaitChannel_WakeAll`.

# System Call Interface



1. The application calls a library wrapper function,
2. which executes a `syscall` instruction, which traps to the kernel,
3. which completes the task
4. and returns to the wrapper function,
5. which returns to the application program.

# System Call Interface

**System Calls** are the application programmer interface (API) that programmers use to interact with the operating system.

- Applications invoke system calls.
- Examples: `fork()`, `waitpid()`, `open()`, `close()`, ...
- The system call interface can have complex calls
  - `sysctl()` Exposes operating system configuration
  - `ioctl()` Controlling devices
- *Key Requirement:* Need a mechanism to safely enter and exit the kernel.
  - Applications don't call kernel functions directly!
  - Remember: kernels provide protection from bugs in application program.

# Privilege Modes

- Hardware provides multiple (at least two) protection modes:
  - **Kernel Mode** or **Privileged Mode** – for the operating system.
  - **User Mode** for applications.
- Kernel Mode can *access privileged processor features*:
  - can access all restricted processor instructions and registers
  - can enable and disable interrupts, setup interrupt handlers
  - controls the system call interface
  - can modify the TLB (virtual memory ... future lecture)
  - can interact directly with peripherals (e.g. mouse, keyboard, monitor, Wi-Fi card).
- Allows kernel to *protect itself and isolate processes from each other*.
  - Processes cannot read or write kernel memory.
  - Processes cannot directly call kernel functions.

# Mode Transitions

- Kernel Mode can only be entered through well defined entry points.
- Two classes of entry points are provided by the processor:
- **Interrupts**
  - Interrupts are *generated by devices to signal that they need attention.*
  - E.g. Keyboard input is ready, a packet has arrived from the internet.
  - More on this during our IO lecture!
- **Exceptions**
  - Exceptions are *conditions discovered by the processor while executing an instruction.*
  - E.g. Divide by zero, page faults, internal processor errors.
- Interrupts and exceptions cause the processor to *transfer control to the interrupt/exception handler*, which occurs at a fixed entry point (i.e. address) in the kernel.

# Interrupts

- Interrupt are *raised by devices needing attention.*
- The **interrupt handler** is a function in the kernel that services a device request.
- Interrupt Process:
  - The device signals the processor through a physical pin or bus message (e.g. change the value on the line from 0 to 1).
  - The processor interrupts the execution of the current running program.
  - It begins executing the interrupt handler in privileged mode.
- Most interrupts can be disabled, but not all.
  - **Non-maskable interrupts** (NMI) are for urgent system requests, e.g. processor is overheating.

# Exceptions

- Exceptions (or faults) are *conditions encountered during execution of a program.*
- Exceptions are due to multiple reasons:
  - Program Errors: divide-by-zero, illegal instructions, unaligned memory access.
  - Operating System Requests: Page faults (i.e. page not in RAM)
  - Hardware Errors: System check (bad memory or internal CPU failures)
- Processors *handle exceptions similar to interrupts.*
  - The processor stops at the instruction that triggered the exception (usually).
  - Control is transferred to a fixed location where the exception handler is located and starts executing in privileged mode.
- *System calls are a class of exceptions!*

# x86-64 Exception Vectors

- *Interrupts, exceptions and system calls use the same mechanism*, sometimes called a **trap**, i.e. “trap into the kernel.”

```
#define T_DE          0      /* Divide Error Exception */
#define T_DB          1      /* Debug Exception */
#define T_NMI         2      /* NMI Interrupt */
#define T_BP          3      /* Breakpoint Exception */
#define T_OF          4      /* Overflow Exception */
#define T_BR          5      /* BOUND Range Exceeded Exception */
#define T_UD          6      /* Invalid Opcode Exception */
#define T_NM          7      /* Device Not Available Exception */
#define T_DF          8      /* Double Fault Exception */
#define T_TS          10     /* Invalid TSS Exception */
#define T_NP          11     /* Segment Not Present */
#define T_SS          12     /* Stack Fault Exception */
#define T_GP          13     /* General Protection Exception */
#define T_PF          14     /* Page-Fault Exception */
#define T_MF          16     /* x87 FPU Floating-Point Error */
#define T_AC          17     /* Alignment Check Exception */
#define T_MC          18     /* Machine-Check Exception */
...
```

# System Calls

*System calls are performed by using the T\_SYS exception vector.*

1. The application loads the system call arguments into the appropriate registers.
2. In particular, it then loads the system call number into register `rdi` (first arg).
3. Executes the `int 60` instruction, where `int` is the assembly language instruction to generate an interrupt and 60 corresponds to `T_SYS`, the system call exception.
4. The processor looks up the interrupt vector (in this case 60) in a table (associating the interrupt to an address) and
5. jumps to that address (in this case the kernel exception handler).
6. When done, the processor returns to userspace and user mode using the `iret` instruction, return from interrupt instruction.

# System Call Numbering

- System calls numbers defined in kern/include/syscall.h
- The system call number is passed as the first argument into syscall.

```
#define SYSCALL_NULL      0x00
#define SYSCALL_TIME       0x01
#define SYSCALL_GETPID     0x02
#define SYSCALL_EXIT        0x03
#define SYSCALL_SPAWN       0x04
#define SYSCALL_WAIT        0x05
```

```
// Memory
#define SYSCALL_MMAP        0x08
#define SYSCALL_MUNMAP       0x09
#define SYSCALL_MPROTECT     0x0A
```

```
...
```

# Hardware Interrupt Handling in x86–64

The **interrupt descriptor table (IDT)** defines the entry point (address to jump to) for a particular interrupt vector. I.e. for interrupt vector 60, the entry point can be found at **IDT[60]**.

1. The OS initializes the IDT with entry points for up to 256 interrupts (i.e. interrupt vectors 0-255). Each one is called an **Interrupt Gate Descriptor**.

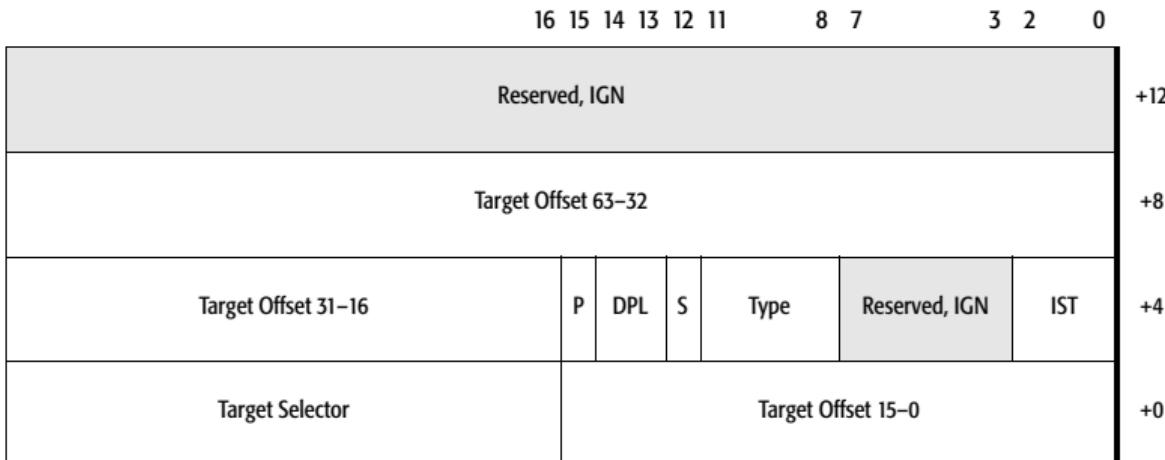


Figure 4-24. Interrupt-Gate and Trap-Gate Descriptors—Long Mode

# Interrupt Gate Descriptor (x86–64)

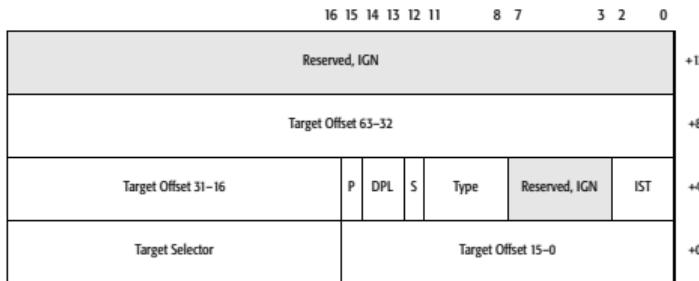
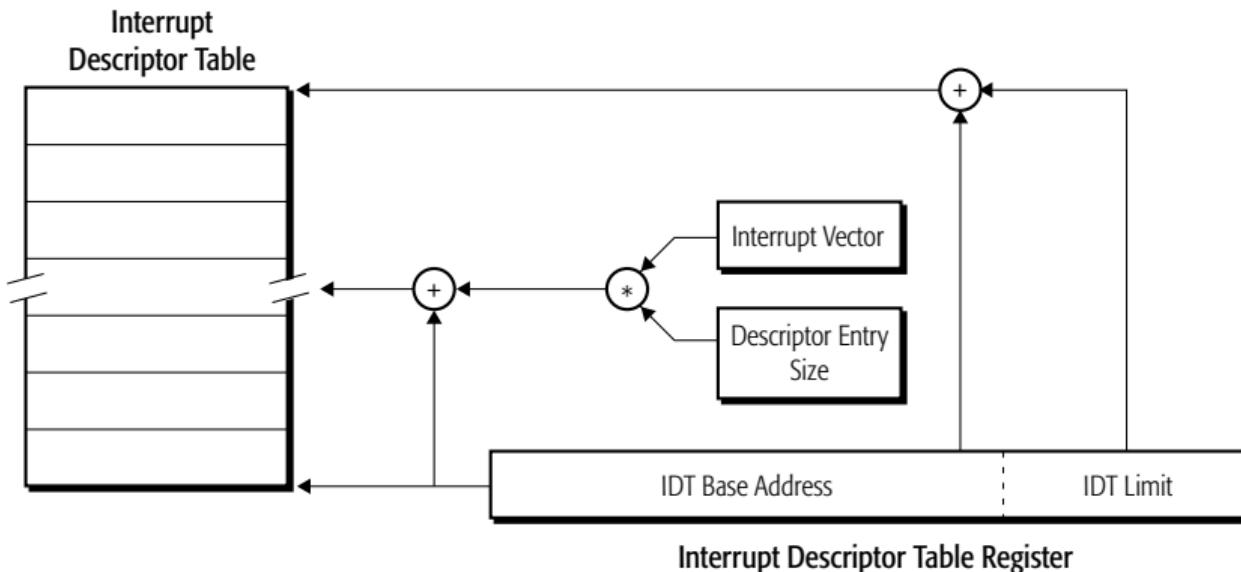


Figure 4-24. Interrupt-Gate and Trap-Gate Descriptors—Long Mode

- **Target Offset:** Is the address of the first instruction (entry point) of the interrupt handler.
- **Target Selector:** sets privilege level (user/kernel mode). More on this later.
- **P:** Present (i.e. is a valid entry in the IDT).
- **Descriptor Privilege Level (DPL):** Minimum privilege level that can trigger this interrupt.
  - Prevents user programs from triggering device interrupts.
- **Type:** Identifies this as a 64-bit IDT entry (as opposed to other types of entries).
- **IST:** Kernel stack to use. There are up to eight stacks available.

# Configuring Interrupt Handling (x86–64)

1. The OS initializes the IDT with entry points for up to 256 interrupts.
2. The OS initializes the IDT base address and length (i.e. limit) of the IDT.
3. The OS executes the `lidt` instruction to load the IDT Register.



# Hardware Interrupt Handling Process (x86–64)

1. *Finds the IDT* (a table) through the IDT Register.
2. *Reads the* IDT descriptor *entry* for that particular interrupt vector (say 60 for `syscall`).
3. The IST field of the entry specifies which stack to use (a number from 0-7).
4. *Looks up the kernel stack location* in the TSS (Task State Segment) which provides information about a particular task (i.e. thread or process).
5. CPU *pushes the interrupt stack frame* onto this stack.

Interrupt-Handler Stack

With Error Code	
	Return SS
+40	
Return RSP	
+32	
Return RFLAGS	
+24	
	Return CS
+16	
Return RIP	
+8	
	Error Code
+8	
← RSP	

With No Error Code	
	Return SS
+32	
Return RSP	
+24	
Return RFLAGS	
+16	
	Return CS
+8	
Return RIP	
+8	
← RSP	

# Hardware Interrupt Handling Process (x86–64)

1. Finds the IDT (a table) through the IDT Register.
2. Reads the IDT descriptor entry for that particular interrupt vector (say 60 for `syscall`).
3. The IST (interrupt stack) field specifies which stack to use (a number from 0-7).
4. Looks up the kernel stack location in the TSS (Task State Segment) which provides information about a particular task (i.e. thread or process).
5. CPU pushes the interrupt stack frame onto this stack.
6. Then the kernel *pushes the trap frame*.
7. Then it *sets up the CPU* to known state to run C code.

Bit Offset	Byte Offset
31	0
16 15	IOPB Base
	+64h
	+60h
	+5Ch
	+58h
	+54h
	+50h
	+4Ch
	+48h
	+44h
	+40h
	+3Ch
	+38h
	+34h
	+30h
	+2Ch
	+28h
	+24h
	+20h
	+1Ch
	+18h
	+14h
	+10h
	+0Ch
	+08h
	+04h
	+00h

Figure 12-8. Long Mode TSS Format

# OS Handler Details

For CastorOS ...

- Interrupt Vectors are defined in `trap_table`.
- IDT is created and IDTR loaded in `Trap_Init`.
- Each interrupt vector has a custom entry point (address of its interrupt handling routine).
- `TRAP_NOEC` (no error code) and `TRAP_EC` (has error code) macro for each
  - Pushes the start of the trap frame including vector number.
  - Most exceptions in x86 push an extra error code on the stack.
  - NOEC (no error code) version pushes an extra 0 to make the stack layout identical.
- `trap_common` pushes the CPU registers
- `trap_entry` is the C handler that dispatches (i.e. starts running) interrupts

# System Call Operation Details

- Application calls into the C library (e.g., calls `write()`).
- Library executes the `syscall` instruction.
- *Kernel exception handler runs.*
  - Switch to kernel stack.
  - Create a *trapframe* which contains the program state.
  - Determine the type of exception (in this case `syscall`).
  - Determine the type of system call (e.g. `SYSCALL_WRITE`).
  - Run the function in the kernel (e.g., `sys_write()`).
  - Restore application state from the trap frame.
  - Return from exception (`iret` instruction).
- Library wrapper function returns to the application.

# How are values passed?

**Application Binary Interface (ABI)** defines the contract between an application's functions and system calls (or more generally between any two machine code modules).

- Operating systems and compilers must obey these rules referred to as the **calling convention**.
- It defines
  - *What is stored* in (i.e. the meaning of) registers during function calls and system calls.
  - *Who is responsible* to save which registers (i.e. caller or callee).
  - Describes *stack alignment* rules (stack pointer address must be divisible 8).
  - Stack alignment was simple in CS241 because everything (ints and addresses) had addresses that were divisible by four.

# x86–64 Calling Conventions

- **Caller-saved registers** are *not preserved across function calls* and must be saved before calling another function if the values need to be preserved.
  - `r10, r11`: Scratch registers
  - `rdi, rsi, rdx, rcx, r8, r9`: Argument registers
  - `rax, rdx`: Return values
- **Callee-saved registers** are *preserved across function calls* and must be saved inside the function if it uses those registers.
  - `rbx, r12–r15`: Saved registers
- *Stack registers*
  - `rsp`: Stack pointer
  - `rbp`: Frame pointer (assuming `-fno-omit-framepointer`)
- Assembly language instructions for functions:
  - `call`: call a function and save the return address on the stack (similar to MIPS `jalr`)
  - `ret`: Return from function (similar to MIPS `jr $31`).

# Functions in x86–64

- *Functions are called with* the `call` instruction, e.g. `foo` calls `bar`.
- `call` pushes the return address to the stack and jumps to the target

```
foo:
```

```
    ...
    push %rbp # Save the frame pointer
    mov %rsp, %rbp # Set the frame pointer to TOS

    # Save caller-save registers (if needed)

    call bar # Call bar

    # Restore registers (if needed)

    pop %rbp
    ret # Return
```

## Functions in x86–64 Continued

- Simple functions may not need to save any registers.
- We save callee-saved registers if needed for performance.
- x64 registers come in 16bit, 32-bit and 64-bit sizes: `%edi` and `%eax` the 32-bit versions of `%rdi` and `%rax`.

```
int bar(int a) {  
    return 41 + a;  
}
```

```
bar:  
    mov %edi, %eax # Move 1st arg to eax (lower 32-bits of rax)  
    add $41, %eax # Add 41 to eax  
    ret
```

# Where are registers saved?

- Registers are saved in memory in the per-thread stack.
- A **stack frame** contains all the saved registers and local variables that must be saved within a single function call.
- The stack is made up of an sequence of stack frames.
- MIPS needs two operations to push (or pop) a stack frame. x64 provides **push** and **pop**.

```
# Push stack element
push %rax
# Equivalent to:
mov %rax, -8(%rsp) # Store into the top of stack
sub $8, %rsp

# Pop stack element
pop %rax
# Equivalent to:
mov 0(%rsp), %rax # Load from the top of stack
add $8, %rsp
```

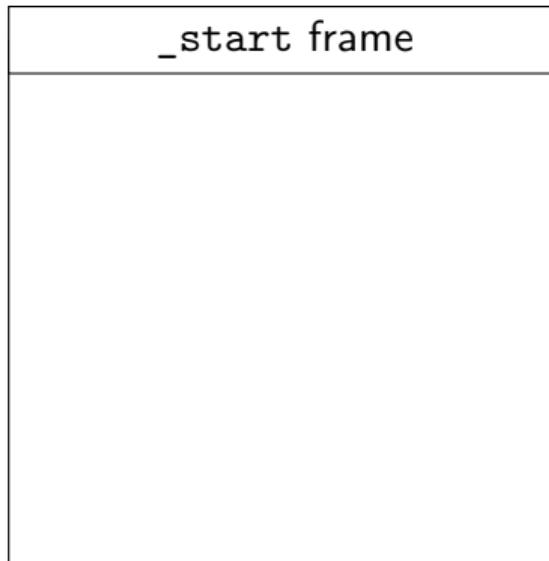
# Execution Contexts

**Execution Context:** The environment where functions execute including their arguments, local variables, memory.

- Context is a unique set of CPU registers including stack pointer.
- There are multiple execution contexts:
  - **Application Context:** Application threads
  - **Kernel Context:** Kernel threads, software interrupts, etc
  - **Interrupt Context:** Interrupt handler
- Kernel and Interrupts usually the same context as the kernel handles the interrupts.
- Context transitions:
  - **Context Switch:** a transition between contexts
  - **Thread Switch:** a transition between threads (usually between kernel contexts).

# Application Stack

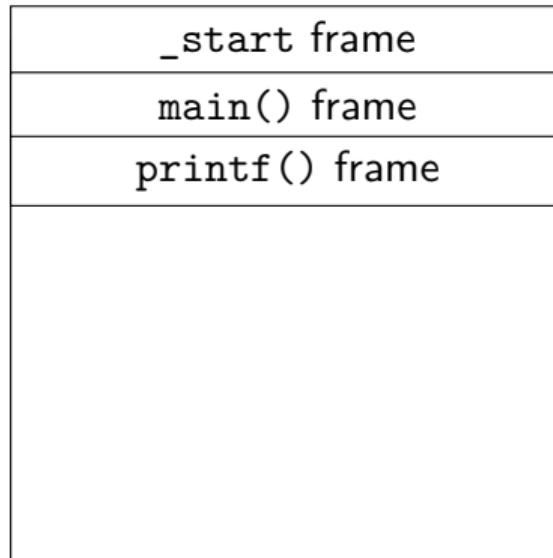
- The stack made of up *frames containing locals, arguments, and spilled* (i.e. saved) *registers*.
- Programs begin execution at `_start`.



User Stack

# Application Stack

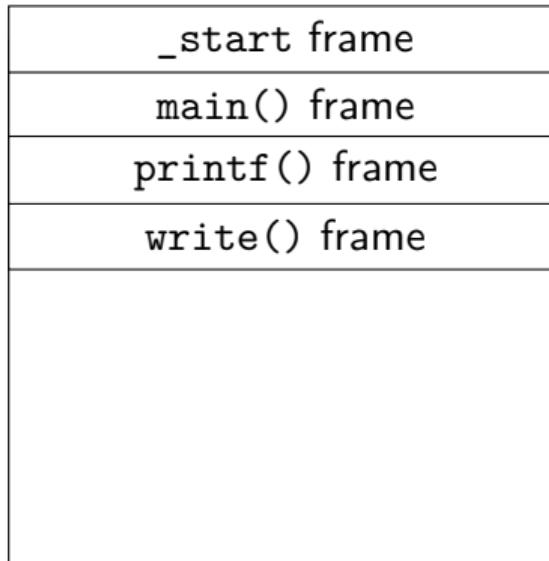
`_start` calls `main`, which in this case, calls the `stdio` library function `printf`.



User Stack

# Application Stack

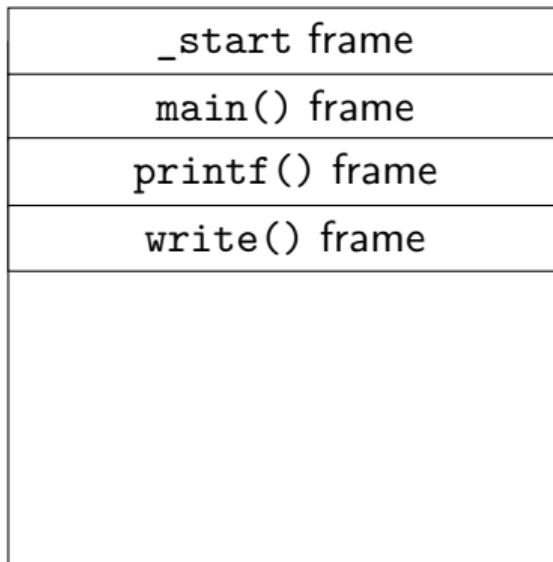
`printf` calls `write`, which is a system call. It puts `write`'s arguments in the appropriate registers and then executes `int 60` which generates an exception.



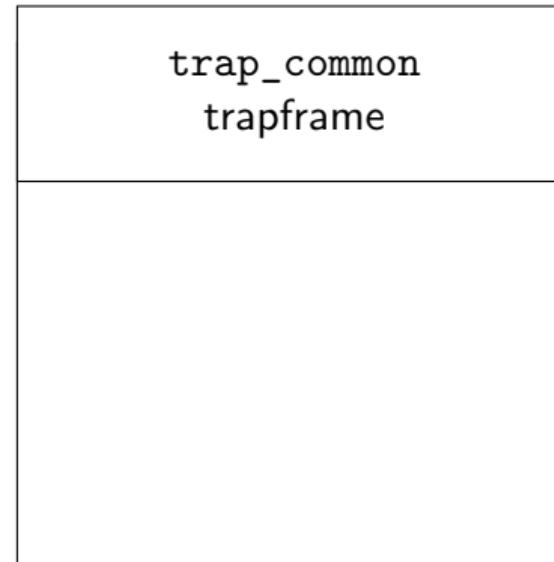
User Stack

# Context Switch: User to Kernel

The kernel routine `trap_common` is called by the processor to handle the exception. It creates a `trapframe`, which stores the application context, so the kernel can now use the core.



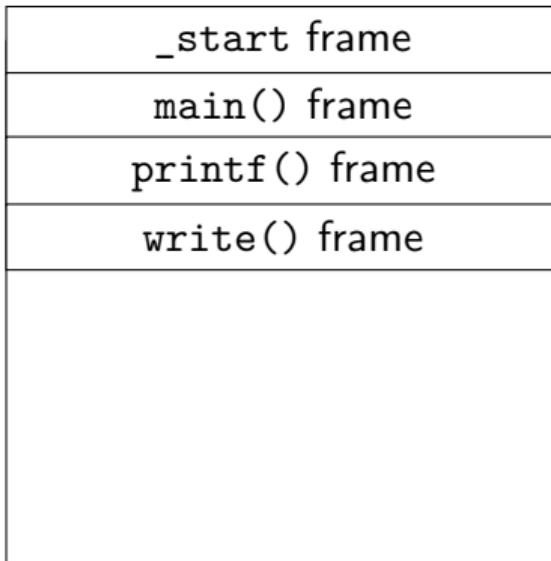
User Stack



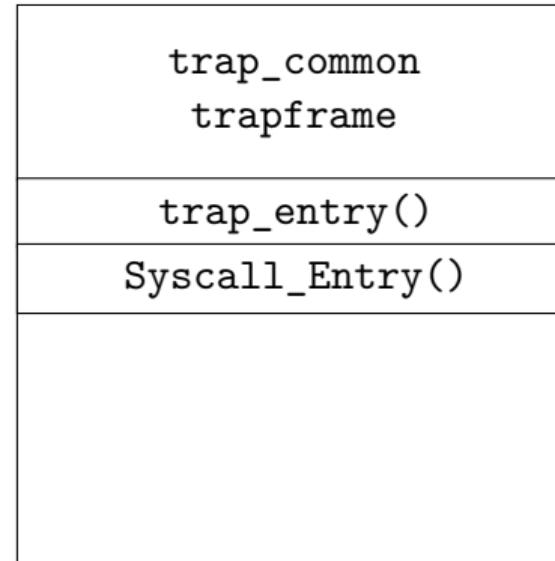
Kernel Stack

# Context Switch: User to Kernel

Once the `trapframe` is saved, `trap_common` calls `trap_entry` to determine the cause of the trap by investigating the `trapframe`. In this case it is a system call (`T_SYSCALL`), so `trap_entry` calls `Syscall_Entry`.



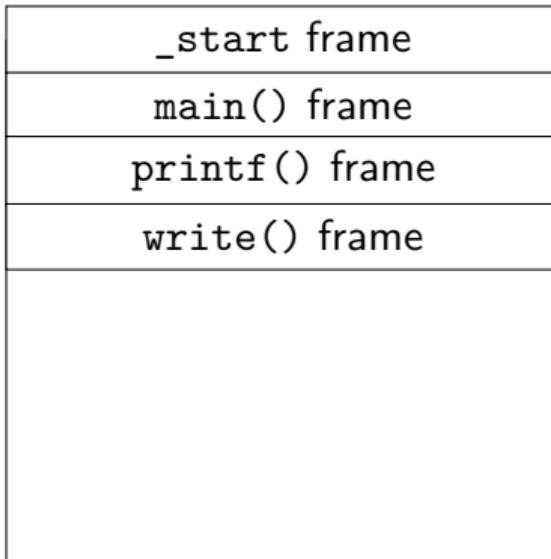
User Stack



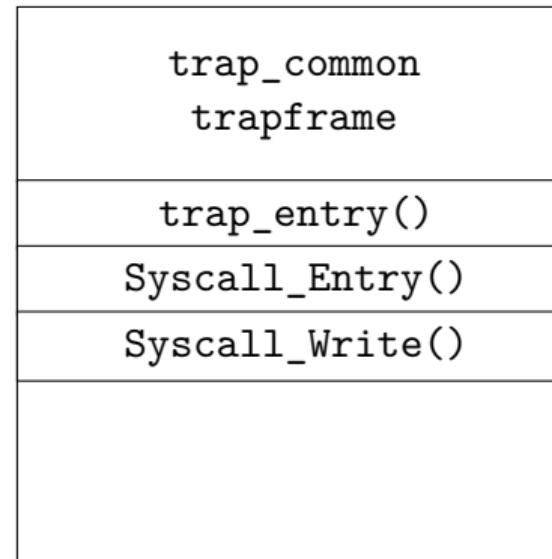
Kernel Stack

# Context Switch: User to Kernel

In a large switch statement `Syscall_Entry` decodes the arguments. Here the value in the `rdi` register was `SYSCALL_WRITE` and so it calls `Syscall_Write`, which in turn calls the console device driver to interact with the monitor.



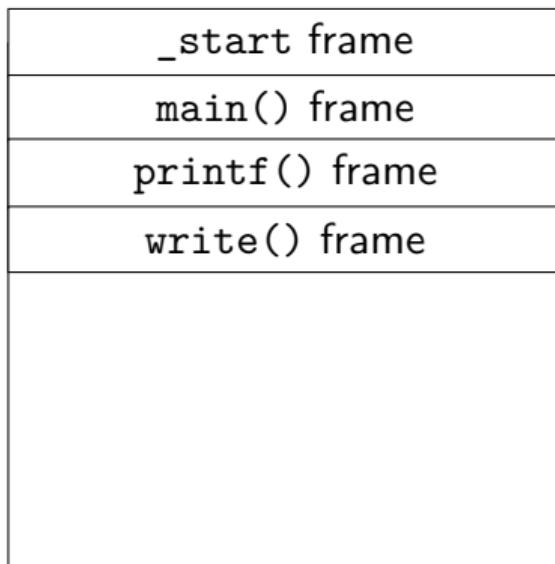
User Stack



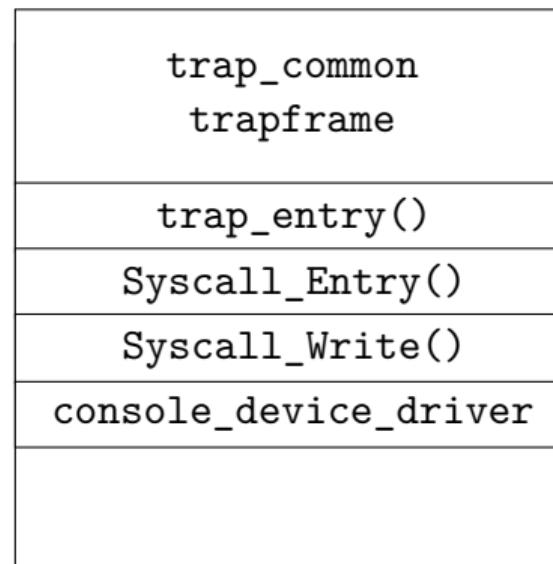
Kernel Stack

# Context Switch: Returning to User Mode

The console device driver writes the text to the screen and then returns, which in turn causes `Syscall_Write` to return, each popping off their respective stack frame.



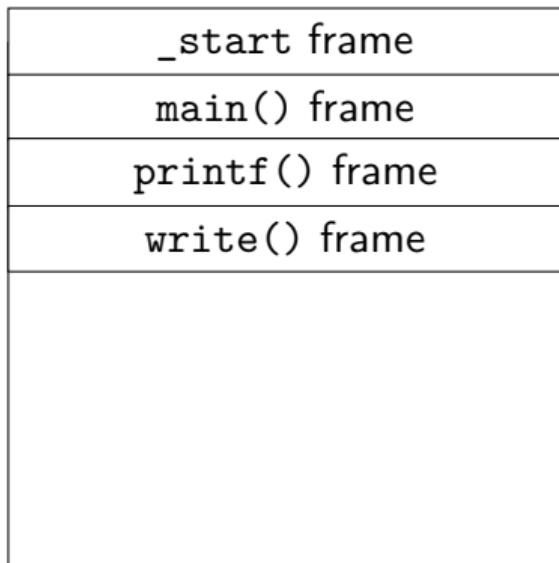
User Stack



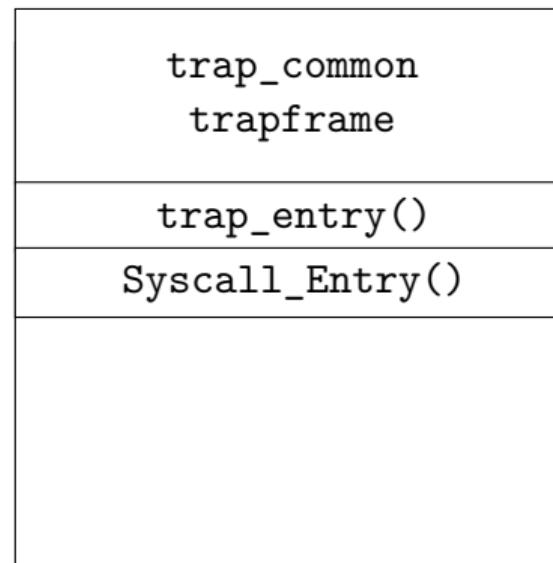
Kernel Stack

# Context Switch: Returning to User Mode

- `Syscall_Entry` returns either a return value or an error code which `trap_entry` stores in trapframe field `rax`.



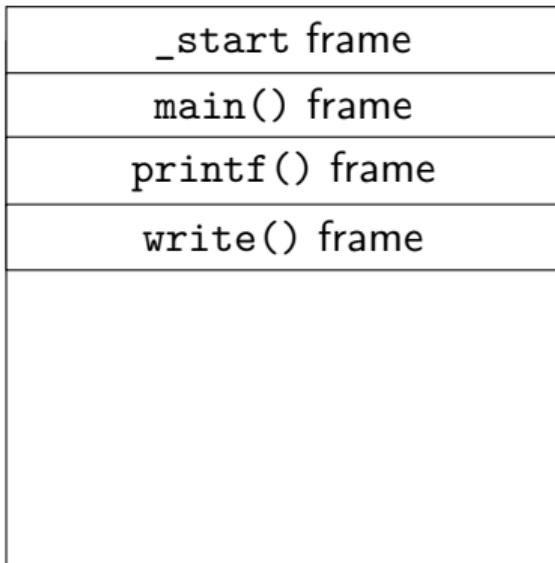
User Stack



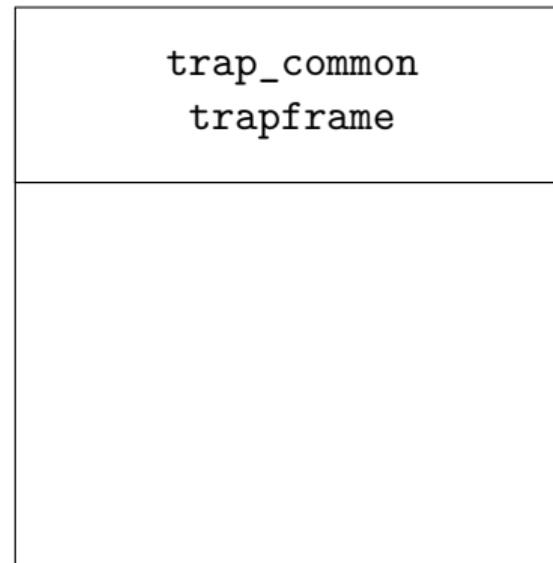
Kernel Stack

# Context Switch: Returning to User Mode

`trap_common` returns to the instruction following `int $60` and now the register `rax` contains return value/error code. It restores the application context by restoring all the values from the trapframe back into their respective registers.



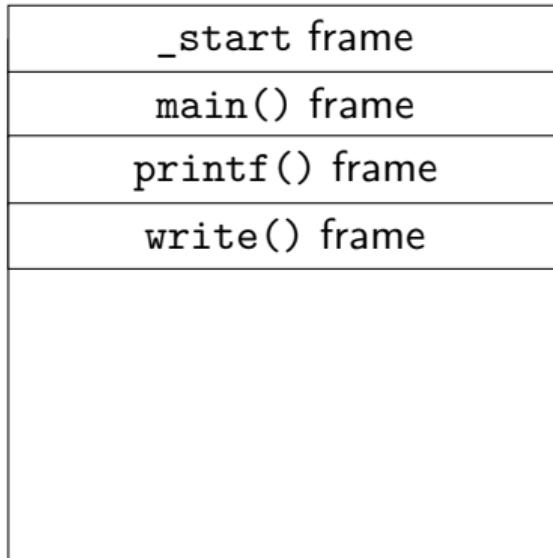
User Stack



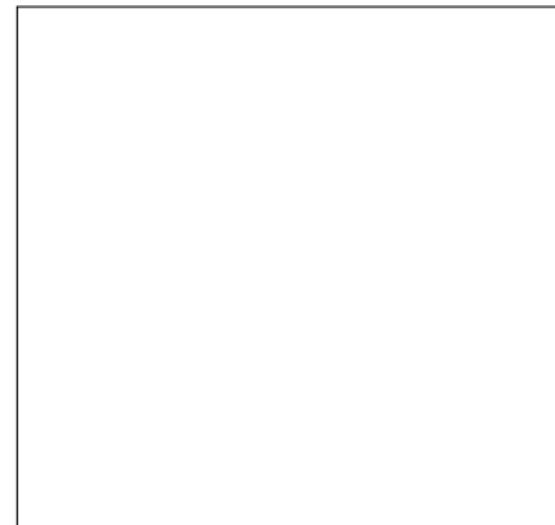
Kernel Stack

# Context Switch: Returning to User Mode

`write` decodes `rax` and updates `errno`, which is where the error codes are stored in POSIX.



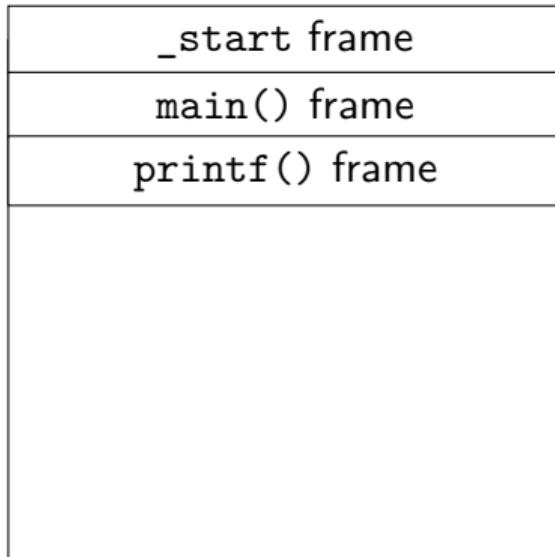
User Stack



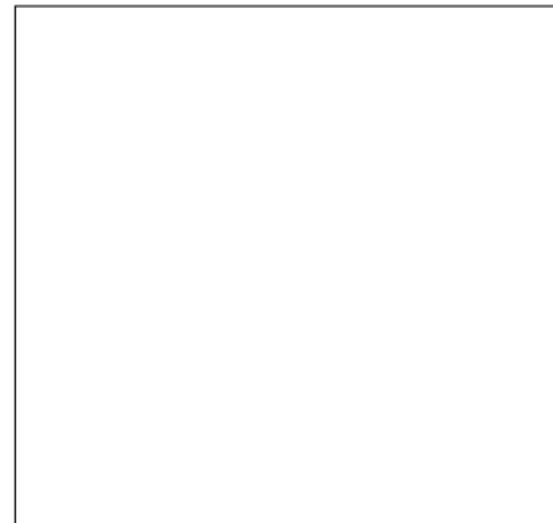
Kernel Stack

# Context Switch: Returning to User Mode

`printf` gets return value (total number of printed chars) or a negative value for an error.



User Stack



Kernel Stack

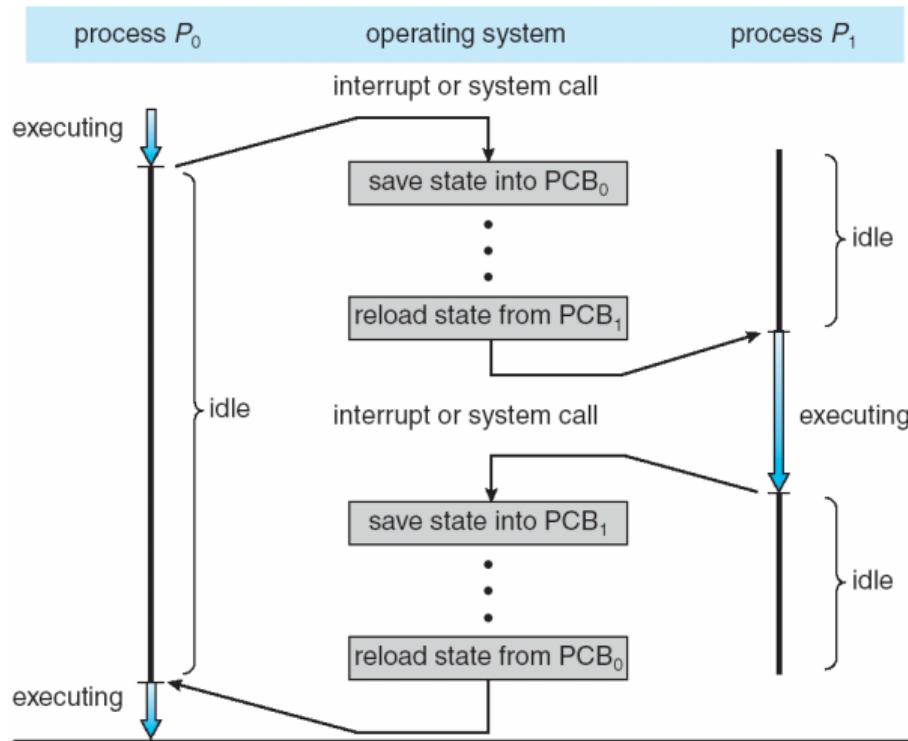
# Scheduling

- *Key Question:* How to pick what to run next?
- Scan process table for first runnable?
  - Expensive. Weird priorities (small pids do better).
  - *Idea:* Divide into runnable and blocked (waiting) threads.
- FIFO/Round-Robin?
  - Put threads on back of list, pull them from front
- What about priority?
  - May want to give some threads a better shot at the processor.

# Preemption

- Option #1: Can *preempt* a process when kernel gets control.
  - Running process can vector (i.e. yields) 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, keyboard key pressed, etc.
  - Previously waiting process becomes runnable.
  - Schedule if higher priority than current running proc.
- Changing running process/thread is called a **context switch**.

# Context switch

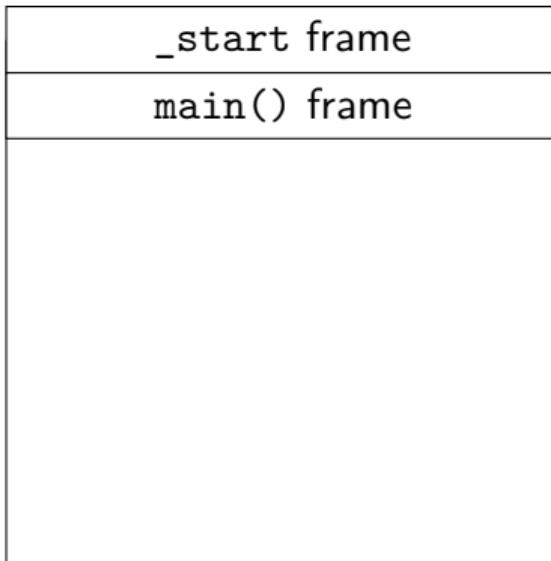


## Context switch details

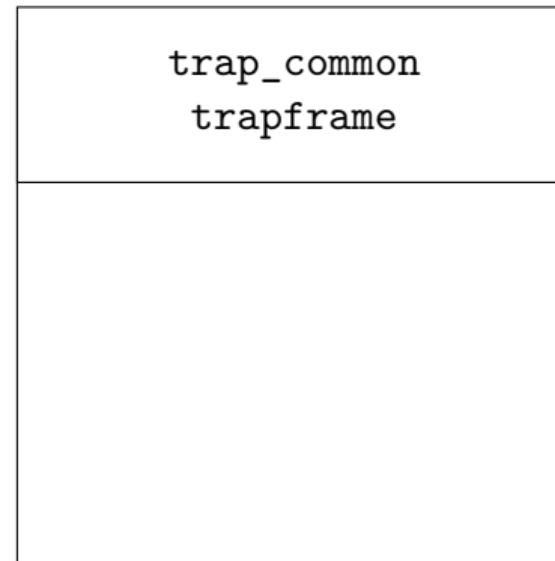
- Very machine dependent. Typical things include:
  - Save program counter and integer registers (always).
  - Save floating point or other special registers
  - Save condition codes
  - Change virtual address translations (e.g. pointers to page table).
- Non-negligible cost
  - Save/restore floating 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)

# Switching Processes: Timer Interrupt

- Starts with a timer interrupt or sleeping in a system call.
- This event *interrupts user process in the middle of the execution.*



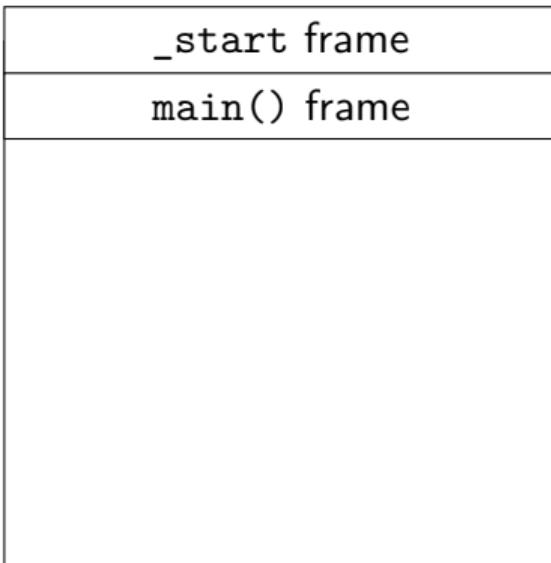
User Stack



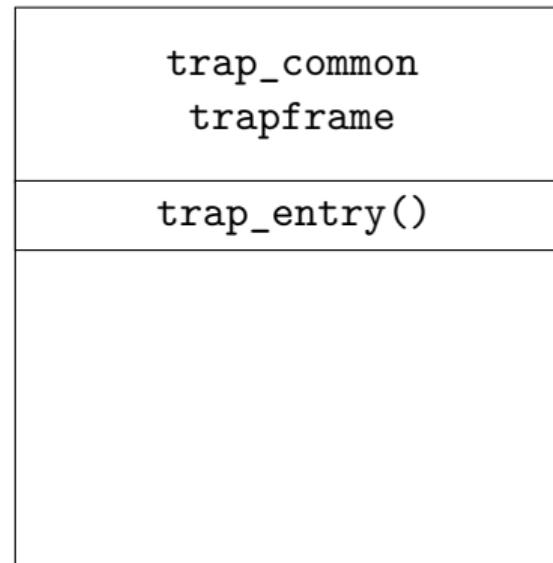
Kernel Stack 1

# Switching Processes: Timer Interrupt

- As before, `trap_common` saves the trapframe
- `trap_entry()` notices a `T_IRQ_TIMER` from the Timer



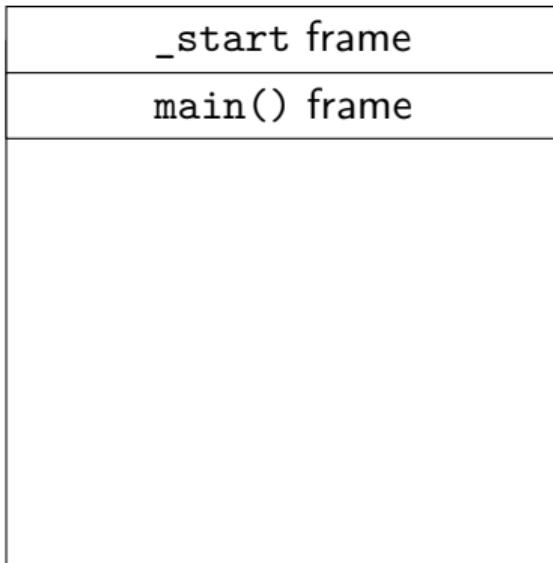
User Stack



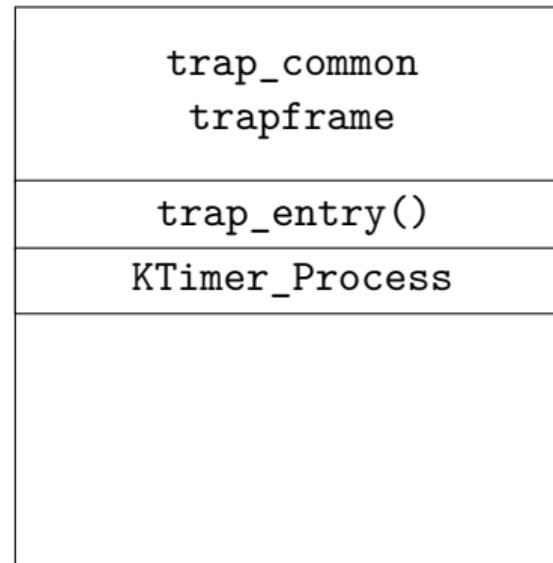
Kernel Stack 1

# Switching Processes: Timer Interrupt

- Calls `KTimer_Process` to process any scheduled timer events.
- Timers trigger processing events in the OS and the CPU scheduler.



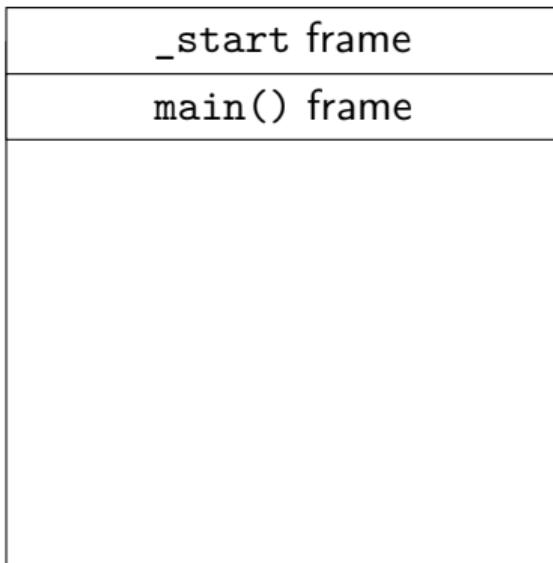
User Stack



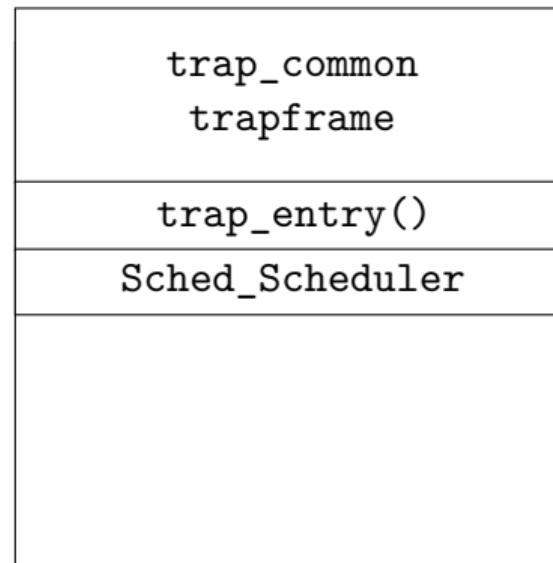
Kernel Stack 1

# Switching Processes: Timer Interrupt

- After `KTimer_Process` returns, `Sched_Scheduler` is called to identify the next runnable thread in the runnable queue
- It calls `Sched_Switch` to perform the switch.



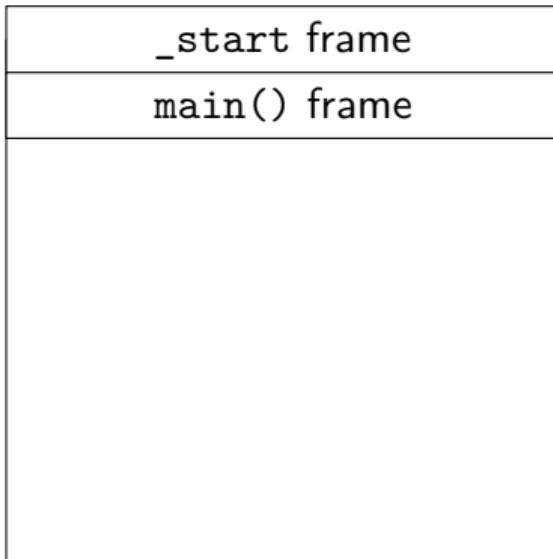
User Stack



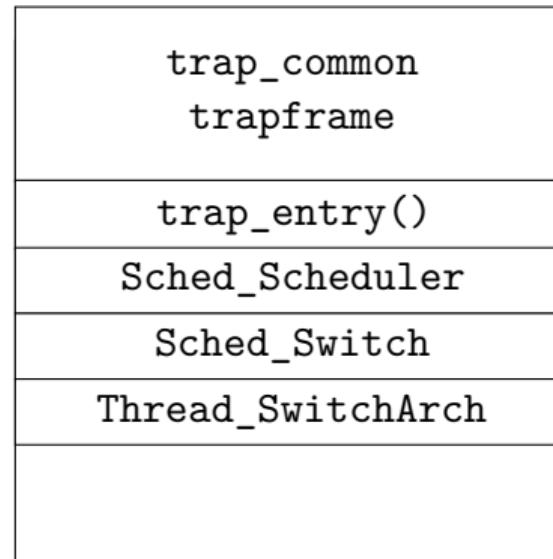
Kernel Stack 1

# Switching Processes: CPU Scheduler

- `Sched_Switch()` loads in the new address space for the new thread and then calls `Thread_SwitchArch` which saves and restore the floating point registers and then calls `switchstack`.



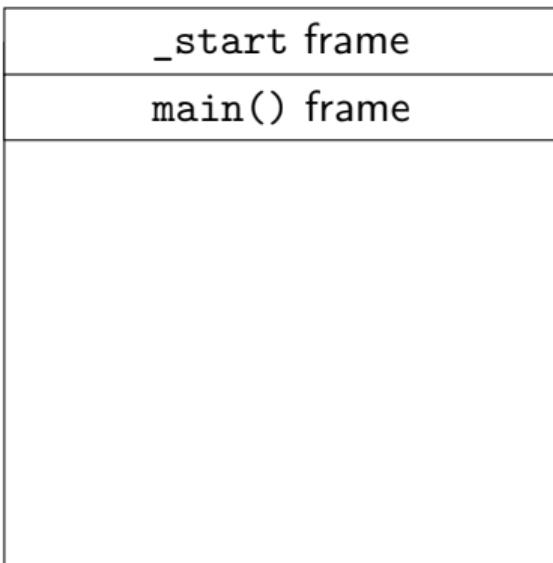
User Stack



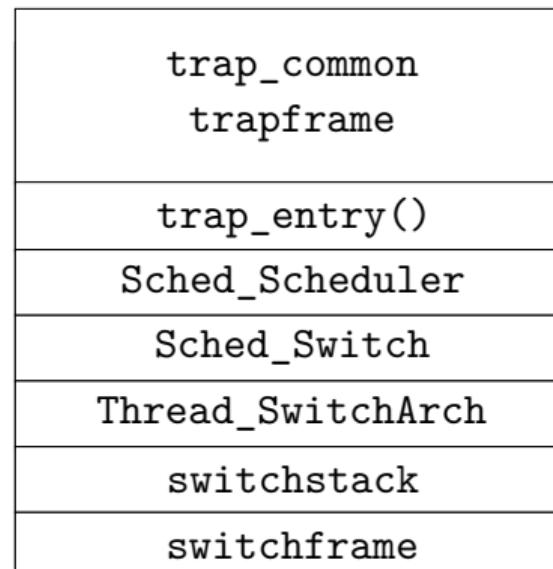
Kernel Stack 1

# Switching Processes: Thread Switch

- `switchstack`: saves and restores kernel thread state in a `switchframe`.
- `switchframe`: contains the kernel's thread context!



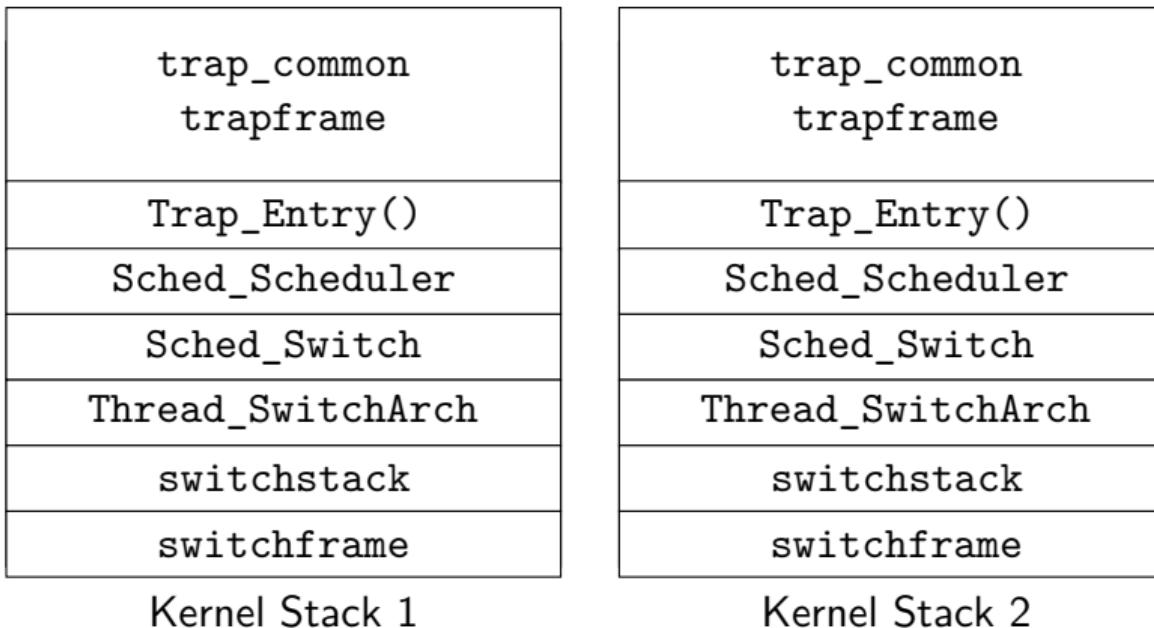
User Stack



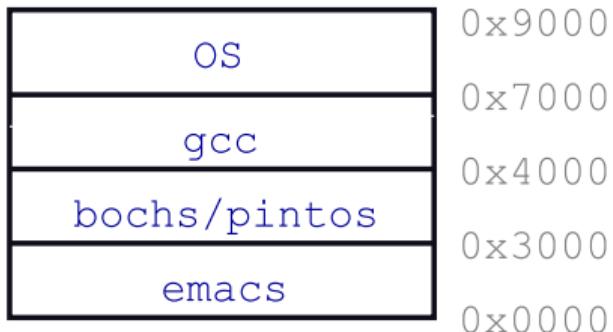
Kernel Stack 1

# Switching Processes: Thread Switch

- `switchstack` saves thread state of thread 1 into Kernel Stack 1 and starts popping off the frames from Kernel Stack 2 and it starts to run again.
- Switching processes is a switch between kernel threads!



# Want processes to co-exist



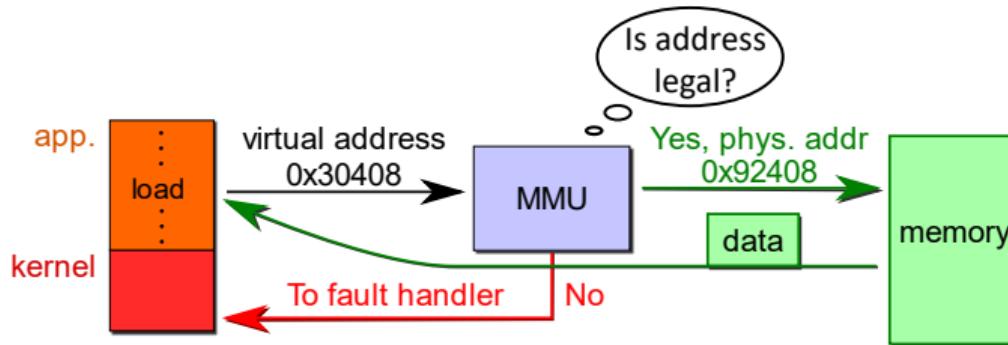
- Consider multiprogramming on physical memory
  - What happens if emacs needs to expand?
  - If emacs needs more memory than is on the machine??
  - If emacs has an error and writes to address 0x7100?
  - When does gcc have to know it will run at 0x4000?
  - What if emacs isn't using its memory?

*Goal:* Manage memory to allow multiple processes to share it.

# 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.

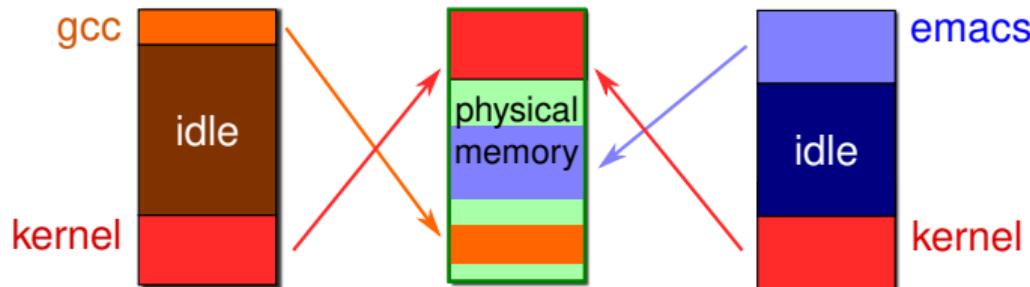
# Virtual memory goals



- *Idea:* Give each program its own “virtual” address space.
  - At run time, the Memory-Management Unit relocates (i.e. translates) each load and store address to an actual memory location. The app does not (and cannot) see physical memory directly.
- The MMU also enforces protection.
  - Prevent one app from messing with (i.e. accessing) another process's memory.
- It also allows programs to see more memory than exists.
  - Somehow relocate some memory accesses to secondary storage (i.e. SSD)

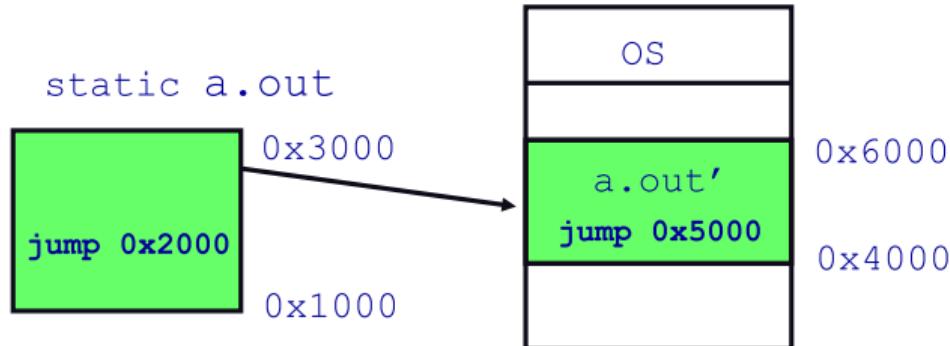
# Virtual memory advantages

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



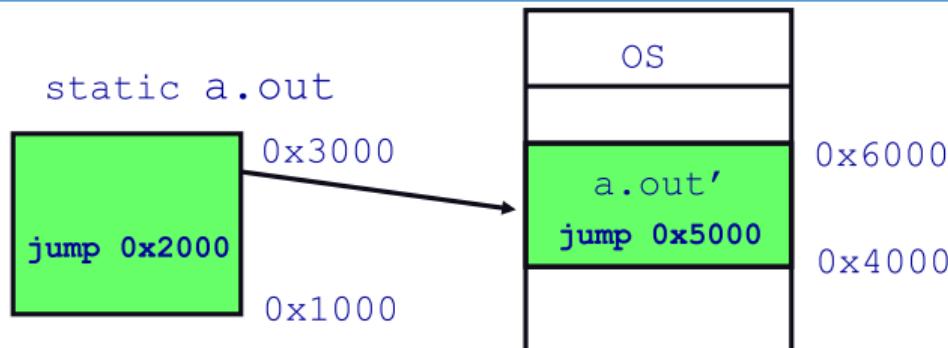
- Write idle parts to disk until needed.
- Let other processes use memory of idle part.
- Like CPU virtualization: when process not using CPU, switch  
(Not using a memory region? switch it to another process)
- *Risk:* VM adds an extra layer  $\Rightarrow$  could be slow.

# Idea 1: load-time linking



- *Linker patches (i.e. adjusts) addresses of symbols such as printf.*
- Idea: link when process executed, not at compile time
  - Determine where process will reside in memory
  - Adjust all references within program (using addition)
- Problems?
  - How to enforce protection?
  - How to move once already in memory (must adjust all pointers)?
  - What if no contiguous free region fits program?

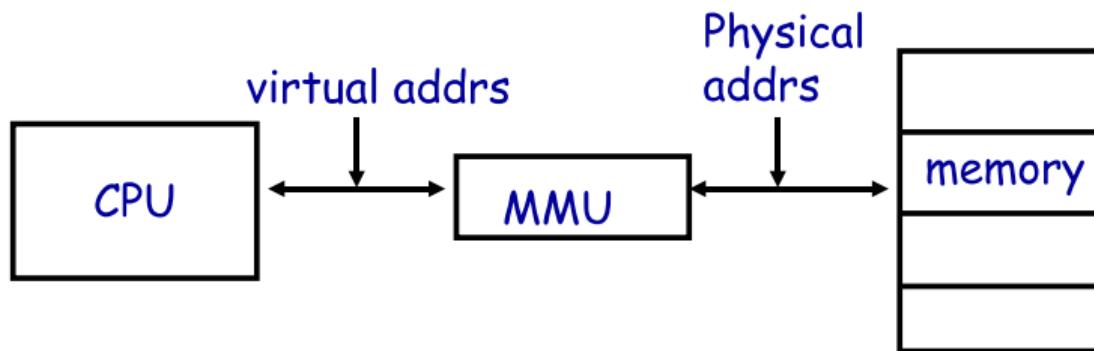
## Idea 2: base + bound register



- Two special privileged registers: **base** and **bound**.
- On each load or store:
  - Calculate the physical address = virtual address + **base**
  - Check  $0 \leq$  virtual address < **bound**, else address ERROR.
- How to move process in memory?
  - Change **base** register.
- What happens on context switch?
  - OS must re-load **base** and **bound** register.

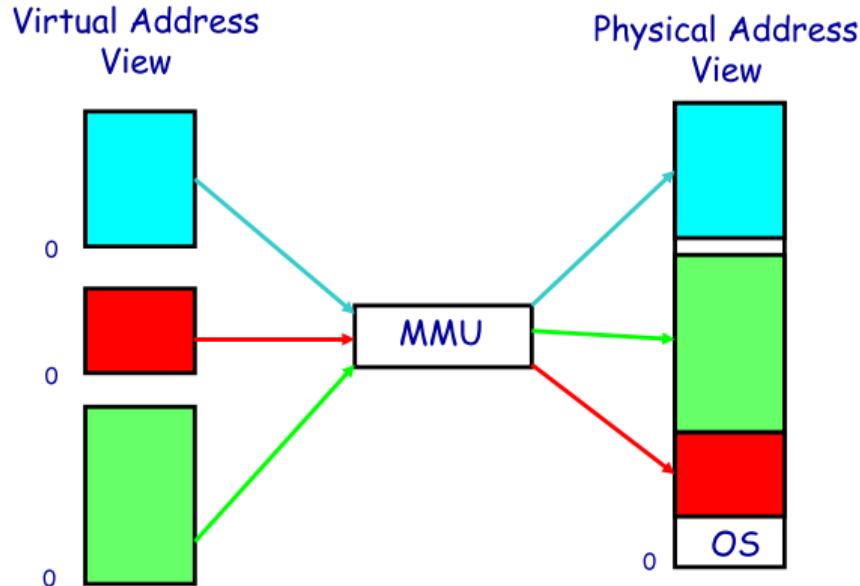
# Definitions

- Programs load and store to **virtual** (or **logical**) **addresses**.
- Actual memory uses **physical** (or **real**) **addresses**.
- The VM hardware is a **Memory Management Unit (MMU)**



- The MMU is usually part of CPU.
- It is accessed with privileged (kernel mode) instructions (e.g., load bound reg).
- It translates from virtual to physical addresses.
- Gives per-process view of memory called an **address space**.

# Address space

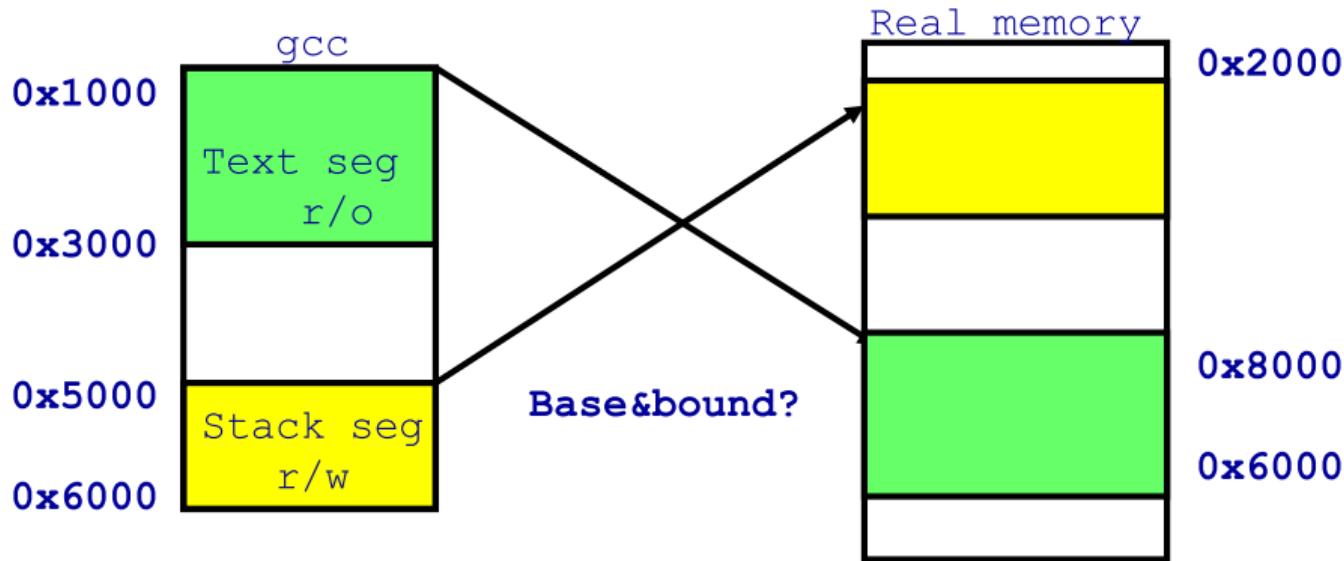


The MMU maps the virtual addresses of each process to their corresponding physical addresses.

# Base+bound trade-offs

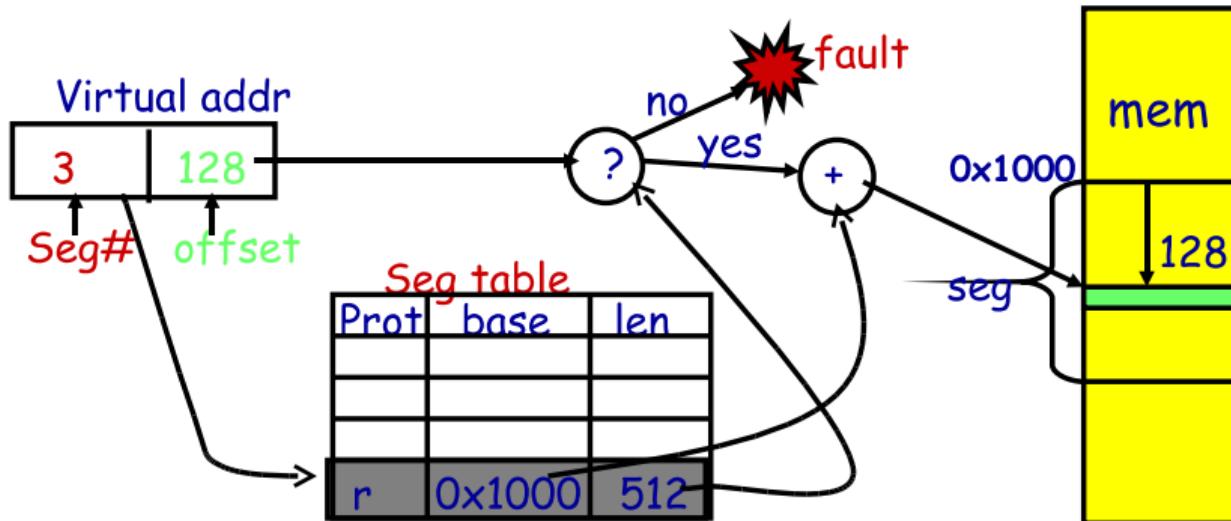
- *Advantages*
  - Inexpensive in terms of hardware: only need two registers.
  - Inexpensive in terms of cycles: add and compare in parallel.
  - Examples: Cray-1 used this scheme.
- *Disadvantages*
  - Growing a process is expensive or impossible.
  - No way to share code or data (E.g., many programs share the stdio library.)
- One solution: Multiple segments (a.k.a. segmentation).
  - E.g., separate code, data, and stack segments.
  - Possibly multiple data segments.

# Idea 3: Segmentation



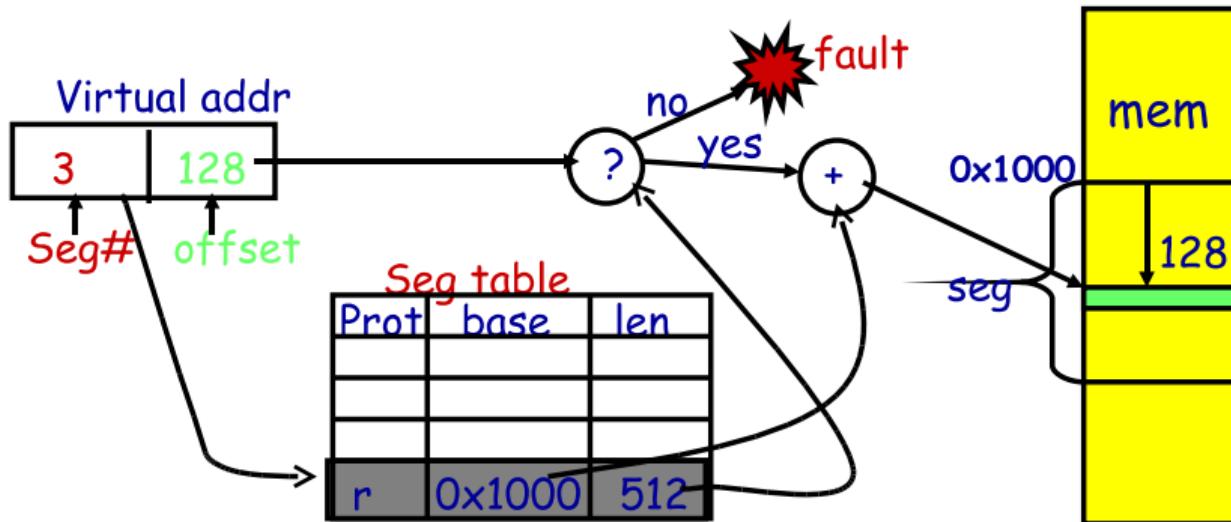
- **Key Idea:** Let processes have many base and bound registers.
  - Address space built from many segments.
  - Can share/protect memory at segment granularity.
  - Segments provide different protection for different segments (e.g. readonly, executable).
- Must specify the segment as part of virtual address.

# Idea 3: Segmentation mechanics



- Each process has its own segment table.
- Each VA is divided into a segment number and an 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)

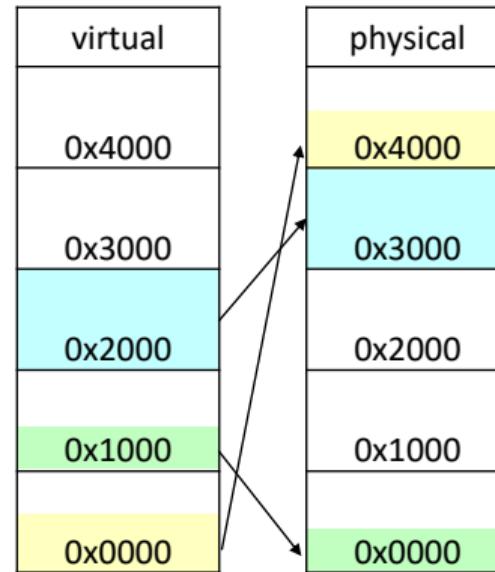
# Idea 3: Segmentation mechanics



1. Is it a valid segment number?
2. If so, check that the offset < segment length (i.e. bound).
3. If so, check that the operation is allow (not writing to readonly memory).
4. If so, physical address = base + offset

## Idea 3: Segmentation example

Seg	base	bound	prot
0	0x4000	0x6ff	10
1	0x0000	0x4ff	11
2	0x3000	0xffff	11
3			00



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

## Idea 3: Segmentation example

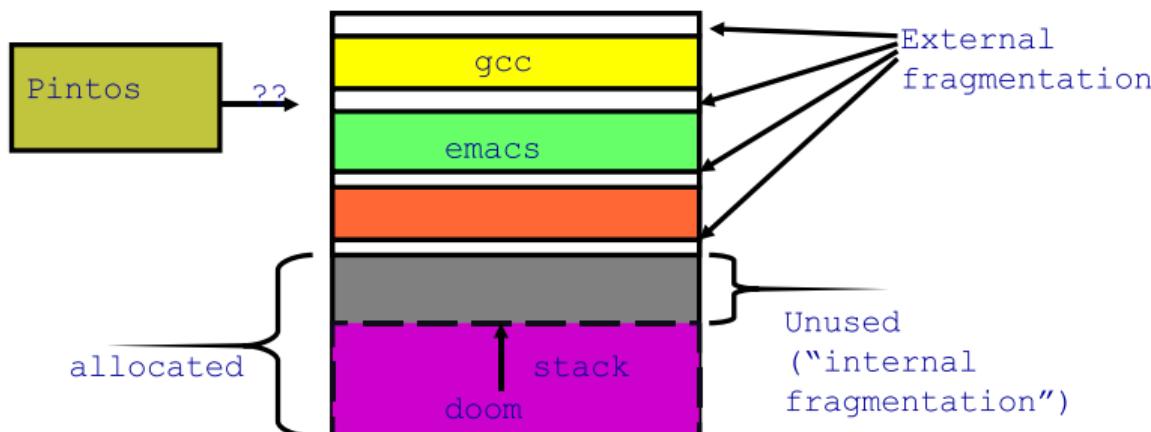
- 2-bit segment number (1st digit), 12 bit offset (last 3)
  - 0x0240 ⇒ segment=0 offset=240  
seg 0 maps to base 4000, so PA is  $4000+240 = 4240$ . It is read only.
  - 0x1108 ⇒ segment=1 offset=108  
seg 1 maps to base 0000, so PA is  $0000+108 = 108$ . It is writeable.
  - 0x265c ⇒ segment=2 offset=65c  
seg 2 maps to base 3000, so PA is  $3000+65c = 365c$ . It is writeable.
  - 0x3002 ⇒ segment=3 offset=002  
seg 3 is not a valid segment number so ERROR: seg fault.
  - 0x1600 ⇒ segment=1 offset=600  
The offset (600) exceeds the bound for segment 1 (0x4ff) so ERROR: seg fault.

## Idea 3: Segmentation trade-offs

- *Advantages*
  - Allows multiple segments per process.
  - Allows sharing! Many segments point to same `stdio` library routines.
  - Don't need entire process in memory.
- *Disadvantages*
  - Requires address translation hardware, which could limit performance.
  - Segments not completely transparent to program (e.g., default segment faster or uses shorter instructions).
  - *n byte segment needs n contiguous bytes of physical memory*
  - Makes **fragmentation** a real problem.

# Idea 3: Fragmentation

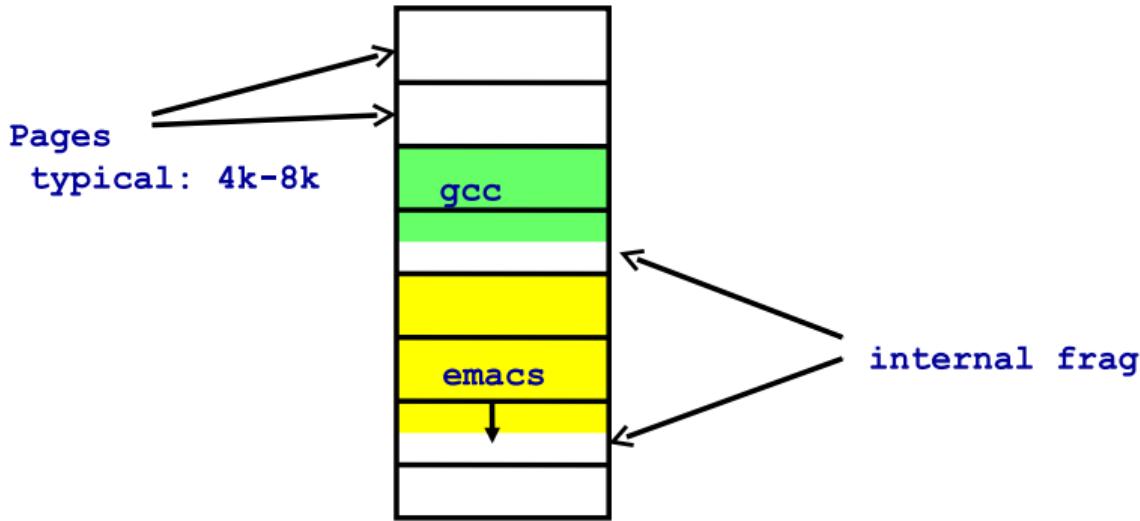
- **Fragmentation:** Inability to use free memory because it is in too many small pieces.
  - Variable-sized pieces: many small holes between pieces as memory gets freed and then partially filled by a request for a smaller pieces. Called **external fragmentation**.
  - Fixed-sized pieces: no external holes, but force waste inside the pieces, e.g. you need 200B but the request size must be a multiple of 4KB. Called **internal fragmentation**.



## Idea 4: Paging

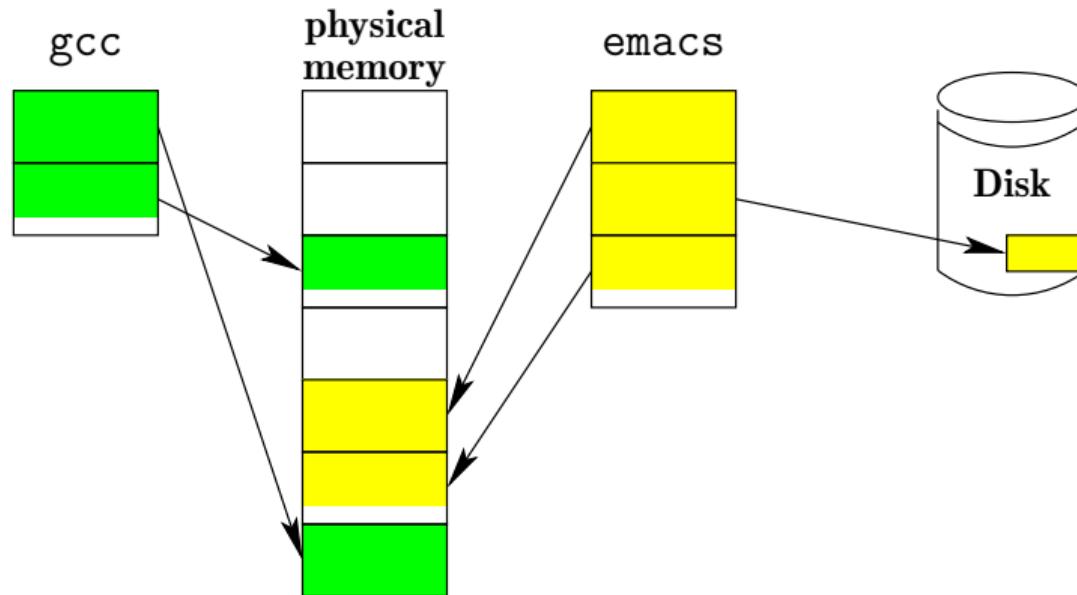
- *Idea* Divide both virtual and physical memory up into small fixed-sized pieces (e.g. 4KB) called **pages**.
- For each process, map its virtual pages to physical pages.
  - Each process has its own separate mapping.
- Allow OS to gain control on (i.e. manage) certain memory operations.
  - Read-only pages trap to OS (i.e. are an error) on write.
  - Invalid pages trap to OS on read or write.
  - OS has the ability to change mapping and resume application.
- Other features sometimes found:
  - Hardware can set “accessed” and “dirty” bits to improve performance.
  - Control page execute permission separately from read/write.
  - Control caching or memory consistency of page.

# Paging trade-offs



- *Eliminates external fragmentation:* every allocation is the same size.
- Simplifies allocation, free, and backing storage (into swap space.)
- Average internal fragmentation of 0.5 pages per “segment.”

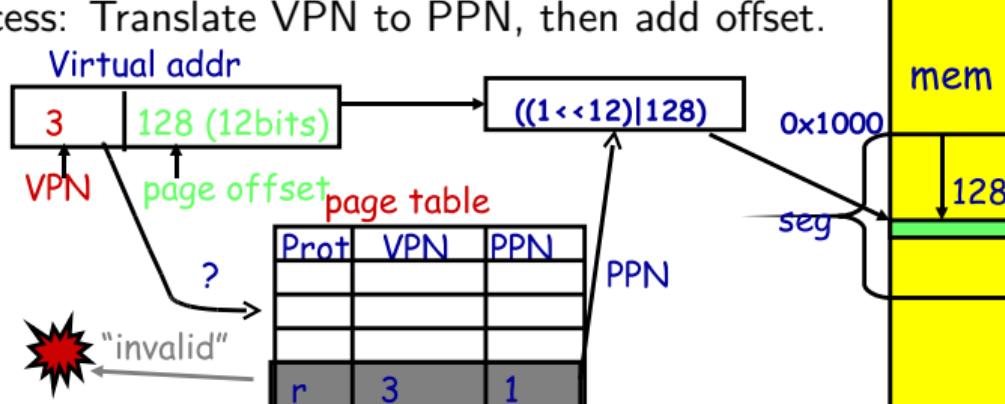
# Simplified allocation



- Allocate any available physical page to any process.
- Can store idle virtual pages on secondary storage (e.g. SSD).

# Paging data structures

- Pages are fixed size, e.g., 4KB
  - Least significant 12 ( $\log_2 4K$ ) bits of address are **page offset**
  - Most significant bits are the **page number**
- Each process has a **page table**
  - The 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

- These were minicomputers, i.e. the step between mainframes and personal computers.
- The PDP-11s were around in the 70s and 80s before personal computers took over.
- 64K virtual memory, 8K pages.
  - Separate address space for instructions (64KB) and data (64KB)
  - I.e., can't read your own instructions with a load instruction, which were for accessing data.
- Entire page table stored in registers
  - 8 Instruction page translation registers
  - 8 Data page translations
- Had to swap 16 machine registers on each context switch.

# MMU Types

- Memory Management Units (MMU) come in two flavours.

## 1 *Software Managed*

- Simpler hardware which asks software to reload pages.
- Requires fast exception handling and optimized software.
- Enables more flexibility in the TLB (cache for address translations), e.g. variable page sizes.
- Examples are mostly old processors: MIPS, Sun SPARC, DEC Alpha, ARM and IBM POWER.

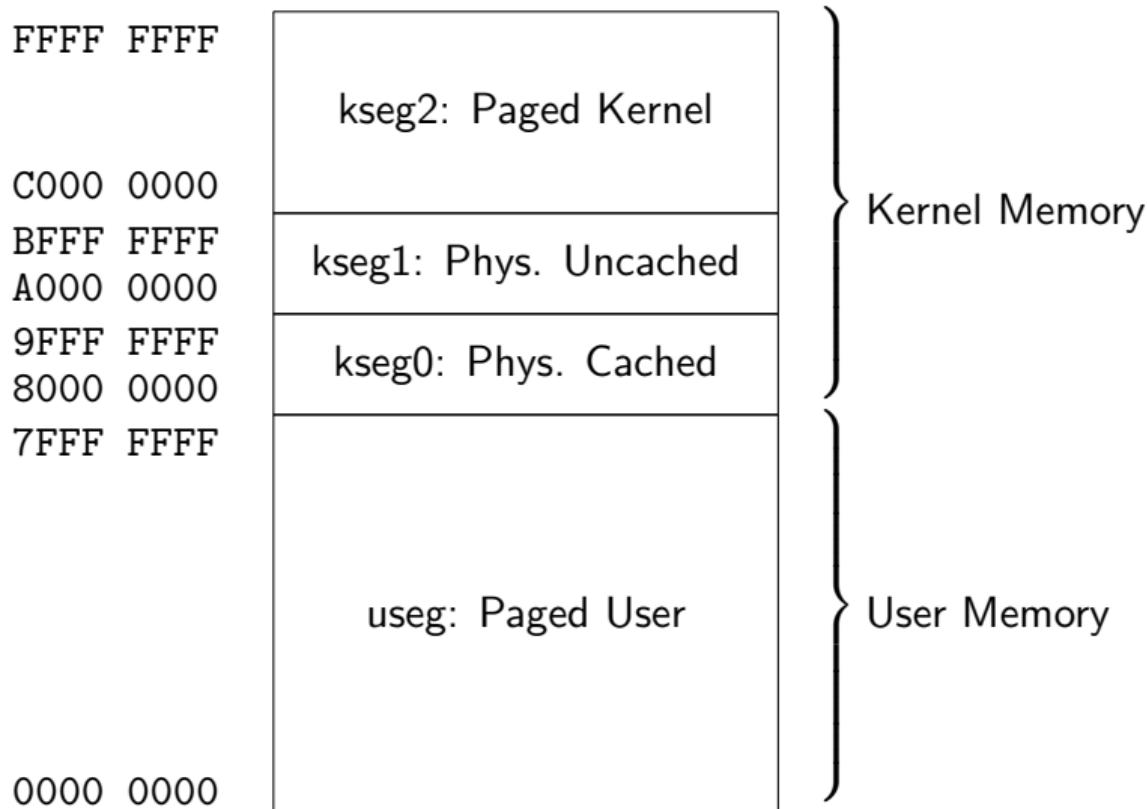
## 2 *Hardware Managed*

- Hardware reloads TLB with pages from a page tables
- Typically hardware page tables are Radix Trees
- Requires complex hardware
- Examples are current processors: x86, ARM64, IBM POWER9+.

# Software managed MMU: MIPS

- Hardware has 64-entry TLB (*cache for page table translations*).
  - References to addresses not in TLB trap to kernel
- Each TLB entry has the following fields:  
Virtual page number, Pid, Page frame (a.k.a. physical page number), plus some flags.
- Much of the *kernel is unpaged*. It uses a different translation method.
  - All of physical memory contiguously mapped in high VM (0x8000 0000 or higher).
  - Kernel has exclusive access to this range of virtual memory addresses.
- User TLB fault handler very efficient
  - Two hardware registers reserved for it
  - The miss handling routine for user addresses can itself have a page fault. This feature allows a page table to be paged and some of those pages possibly swapped out.
- OS is free to choose page table format!

# MIPS Memory Layout



# MIPS Translation Lookaside Buffer

- TLB Entries: 64-bit entries: 0–31 is tlblo and 32–63 is tlbhi
  - PID: Process ID (i.e. it is a tagged TLB)
  - N: No Cache - disables caching for memory mapped I/O
  - D: Writeable - makes the page writeable
  - V: Valid
  - G: Global - ignores the PID during lookups

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Frame Number (VPN)											PID																				
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Physical Page Number (PPN)											N	D	V	G																	

- Page Sizes: Multiples of 4 from 4 kiB–16 MiB
  - 4 kiB, 16 kiB, 64 kiB, 256 kiB, 1 MiB, 4 MiB, 16 MiB

# TLB PID and Global Bit

- Process ID (PID) allows *multiple processes to have translations in the TLB*.
  - Different processes have different translations for the same virtual address.
  - We don't need to flush the TLB on context switch by setting the process ID.
  - I.e. for a match to exist, both the virtual page number and the PID (i.e. all of tlbhi) must match.
  - Only flush TLB entries when reusing a PID.
  - Current PID is stored in `c0_entryhi` (a co-processor register).
- **Global bit**
  - Used for *pages shared across all address spaces* in kseg2 or useg.
  - Ensures the TLB ignores the PID field.
  - Typically in most hardware a TLB flush doesn't flush global pages.

# TLB Instructions

- MIPS co-processor 0 (COP0) provides the TLB functionality
  - COP0 provides most of the privileged functionality for the kernel
- Four *instructions*:
  - `tlbwr`: TLB write a random slot
  - `tlbwi`: TLB write a specific slot
  - `tlbr`: TLB read a specific slot
  - `tlbp`: Probe (look for) the entry that matches `c0_entryhi`.
- For each of these instructions you must load some, or all of, the following *registers*.
  - `c0_entryhi`: high bits of TLB entry
  - `c0_entrylo`: low bits of TLB entry
  - `c0_index`: TLB Index

# Hardware Lookup Exceptions

- TLB Exceptions:
  - UTLB Miss: useg (user segment) page number not found in TLB.
  - TLB Miss: kseg2 (kernel segment) page number not found in TLB.
  - TLB Mod: Generated when writing to read-only page.
- *UTLB handler is separate from general exception handler.*
  - UTLBs are very frequent and require a hand optimized path.
  - 64 entry TLB with 4 kiB pages covers 256 kiB of memory.
  - Modern machines have workloads with far more memory.
  - Require more entries (expensive hardware) or larger pages.

# Hardware Lookup Algorithm

1. If the most significant bit (MSB) is 1 (i.e. virtual address  $\geq 0x8000\ 0000$  so it is a kernel address) and in user mode → address error exception.
2. If no VPN match → TLB/UTLB miss exception. Translation not in TLB.  
if global bit set, implies PID is not required for the translation
3. If PID mismatches and global bit not set → TLB/UTLB miss. Translation not in TLB.
4. If valid bit not set → TLB/UTLB miss exception. The entry is stale.  
the entry is not valid
5. Write to read-only page → TLB mod(ification) exception
6. If N bit is set directly access device memory (disable cache).

Ref: <https://student.cs.uwaterloo.ca/~cs350/common/sys161manual/mips.html>

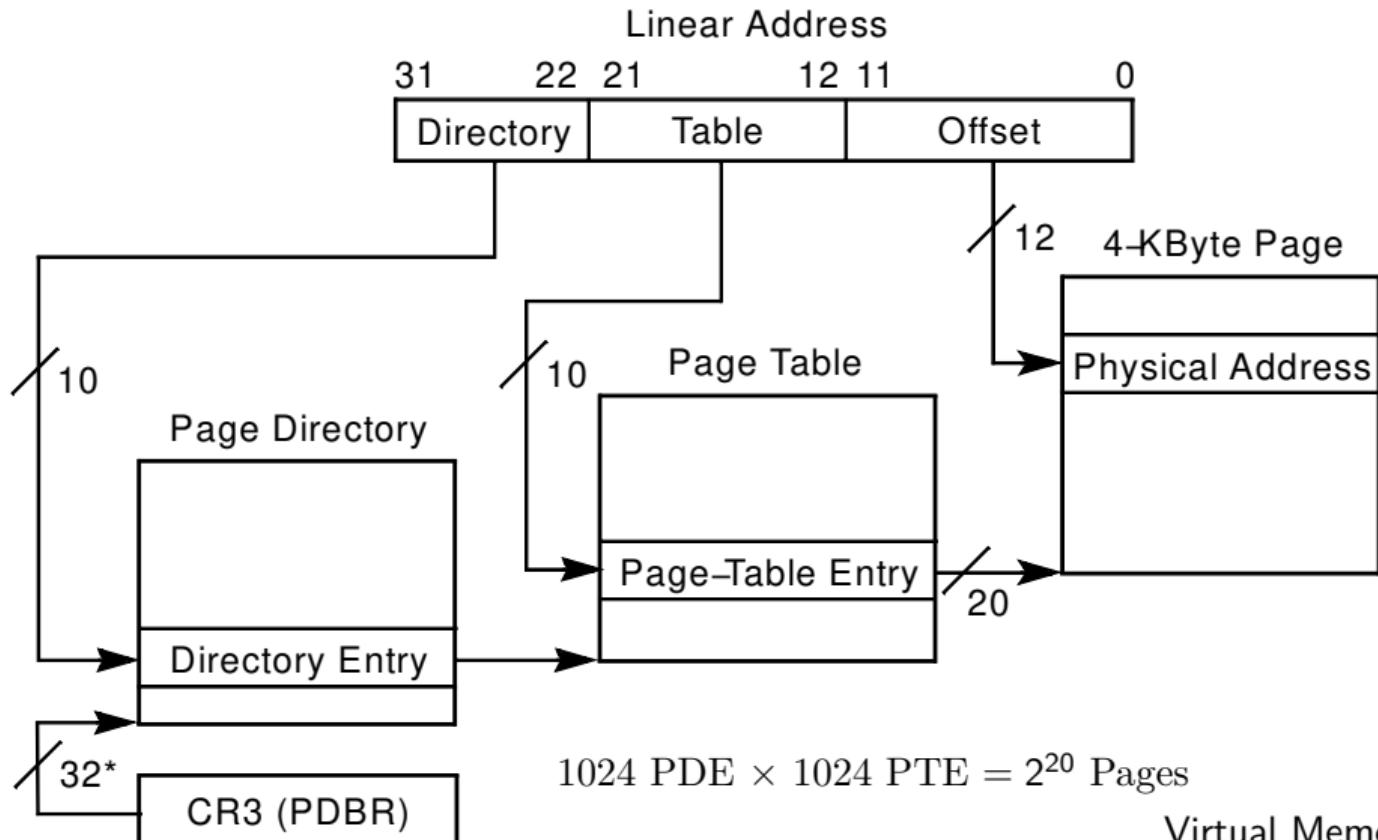
# Hardware Managed MMU: x86

- TLB Managed by Hardware and Microcode (executes instructions using the hardware).
  - It has two levels of TLBs each acting as a cache.
  - Typical: a 1K entry TLB and 512 entry TLB.
  - TLB acts as a cache page table structure.
  - *TLB automatically reloaded from page table by the processor.*
  - If the entry is missing in the page tables, it results in a page fault.
- The OS *builds a Radix tree* (also called a trie) describing memory layout.
  - Control register `%cr3` points to radix-tree root. uses radix tree instead of a page table
- 32-bit mode uses a two-level radix tree
  - 1024 entries per level with page sizes 4 KiB or 4 MiB
- 64-bit mode
  - 512 entries per level with page sizes of 4 KiB, 2 MiB, or 1 GiB.
  - Four levels by default, newer chips support five levels.

# x86 Paging

- Paging enabled by bits in a control register (`%cr0`).
  - Only privileged OS code can manipulate control registers.
- Normally 4KB pages
- `%cr3`: points to 4KB page directory
- **Page directory:** 1024 PDEs (page directory entries)
  - Each contains the *physical addresses of a page tables*.
- **Page table:** 1024 PTEs (page table entries)
  - Each contains the *physical addresses of virtual 4K pages*.
  - Page table covers 4 MB of Virtual memory addresses.
- See intel manual for detailed explanation
  - Volume 2 of [AMD64 Architecture docs](#)
  - Volume 3A of [Intel Pentium Manual](#)

# x86 Page Translation

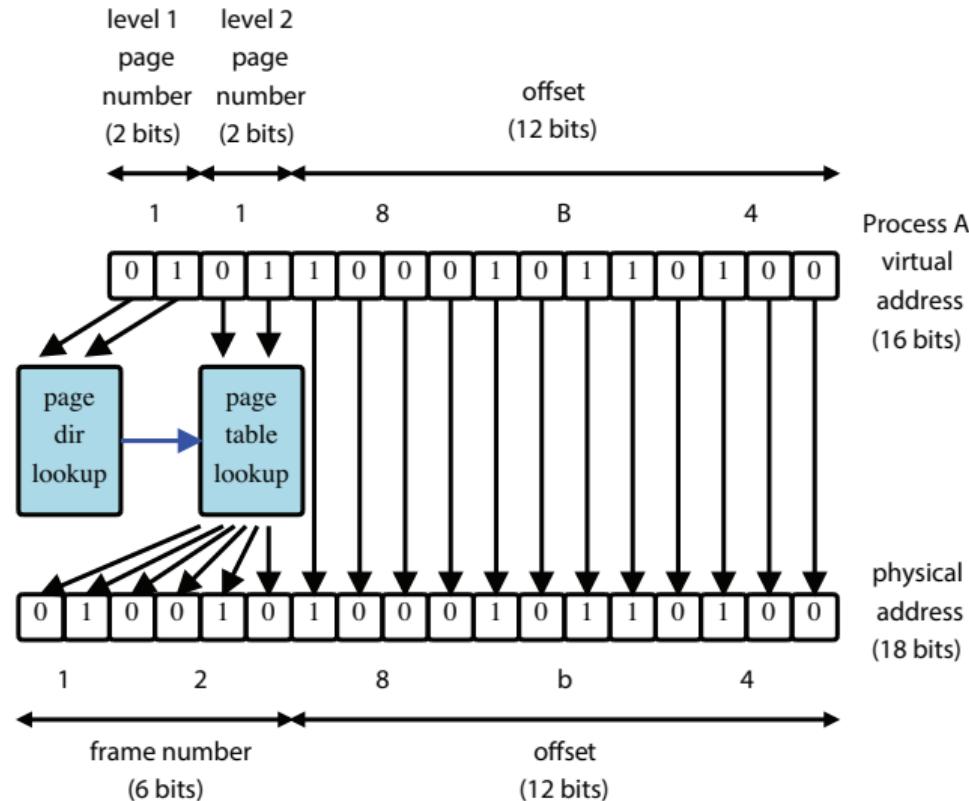


## 4. Simple Two-Level Paging Example

Single-Level Paging		Two-Level Paging	
VPN	PPN	V?	Directory
0x0	0x0F	1	Page
0x1	0x26	1	0x0 Address
0x2	0x27	1	0x1 V?
0x3	0x28	1	0x2 Table 1
0x4	0x11	1	0x3 NULL
0x5	0x12	1	Table 1
0x6	0x13	1	VPN PPN V?
0x7	NULL	0	0x0 0x0F 1
0x8	NULL	0	0x1 0x26 1
...	...	...	0x2 0x27 1
0xE	NULL	0	0x3 0x28 1
0xF	NULL	0	Table 2

- This simple example is using page tables rather than radix trees to keep it simple.
- V? means “Is this entry valid?” If a level 1 entry is not valid, there will be no level 2 table.
- The *address translation is the same* as single-level paging but the *lookup is different*.

## 4. Two-Level Paging: Address Translation



Translating virtual address 0x58B4 to physical address 0x128B4.

Virtual Memory 266

## Two-Level Paging: Address Translation

E.g. translate virtual address 0x58B4.

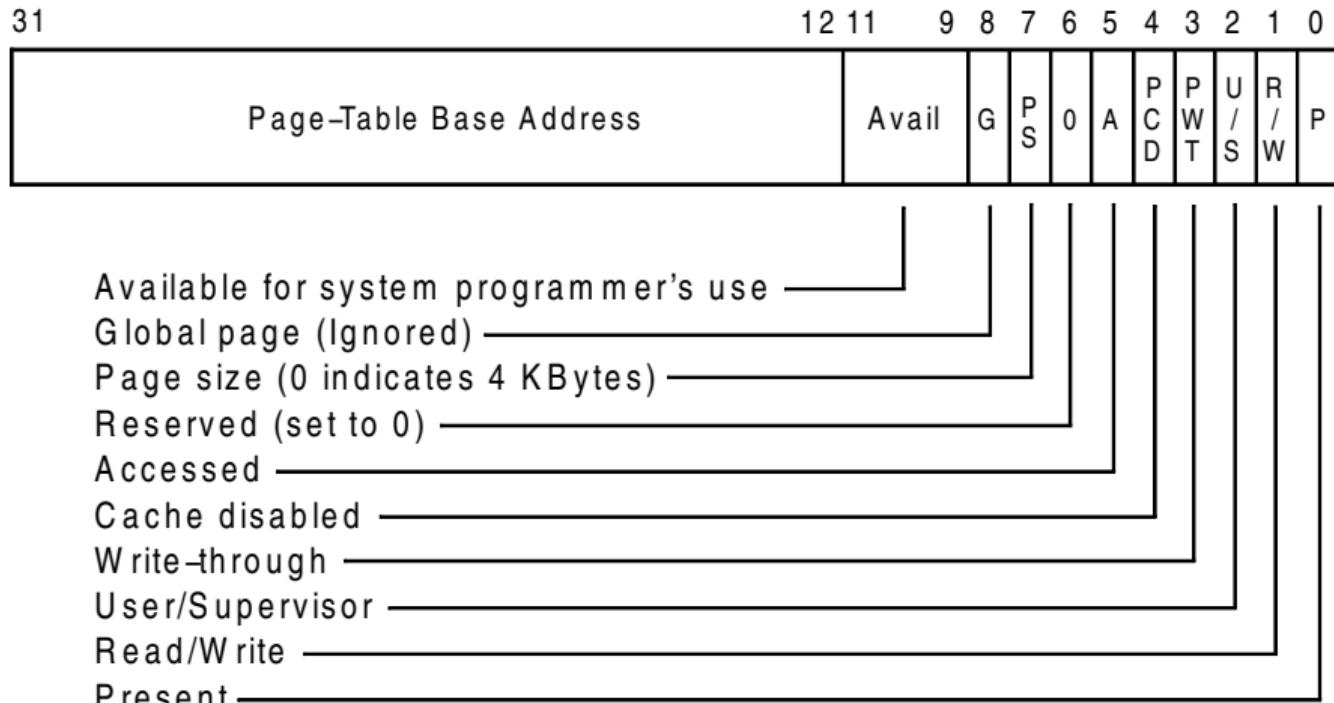
- You need to know the number of bits for each page and the number of levels.
  - In this example, there are 2 levels and each page number uses 2 bits.
  - So the virtual address  $v$  has 3 parts:  $p_1$   $p_2$  offset
  - The first hex digit 0x5 is 0101 in binary.
  - We split up the address into  $p_1 = 01$ ,  $p_2 = 01$ , offset=8B4
    1. Use the first two bits,  $p_1$ , to find the page table.  
⇒ the entry for  $p_1 = 01$  is Table 2.
    2. Use the second two bits,  $p_2$ , to index into Table 2.  
⇒ the entry (i.e. frame number) for  $p_2 = 01$  in Table 2 is 0x12.
  - Combine the frame number 12 with the offset 8B4 to get the physical address 0x128B4.
- This arrangement can be generalized to more than two levels ...

# x86 Page Tables

- The page directory and the page table *contain much of the same information* as MIPS.
  - Global page - all processes can access it
  - Accessed - whether the page has been accessed recently (or not)
  - Cache disabled - do not store in cache
  - *Write-through* - enable write-through caching
  - *User/Supervisor* - privilege needed to access page
  - Read/Write - is the page writeable (i.e. not read only)
  - Present - in RAM rather than in swap space on the SSD.
- Directories ignore the global page bit but specifies *page size* (more flexibility)
- Pages have a dirty bit and can specify *caching behaviour* at the page level (called a Page Table Attribute Index).

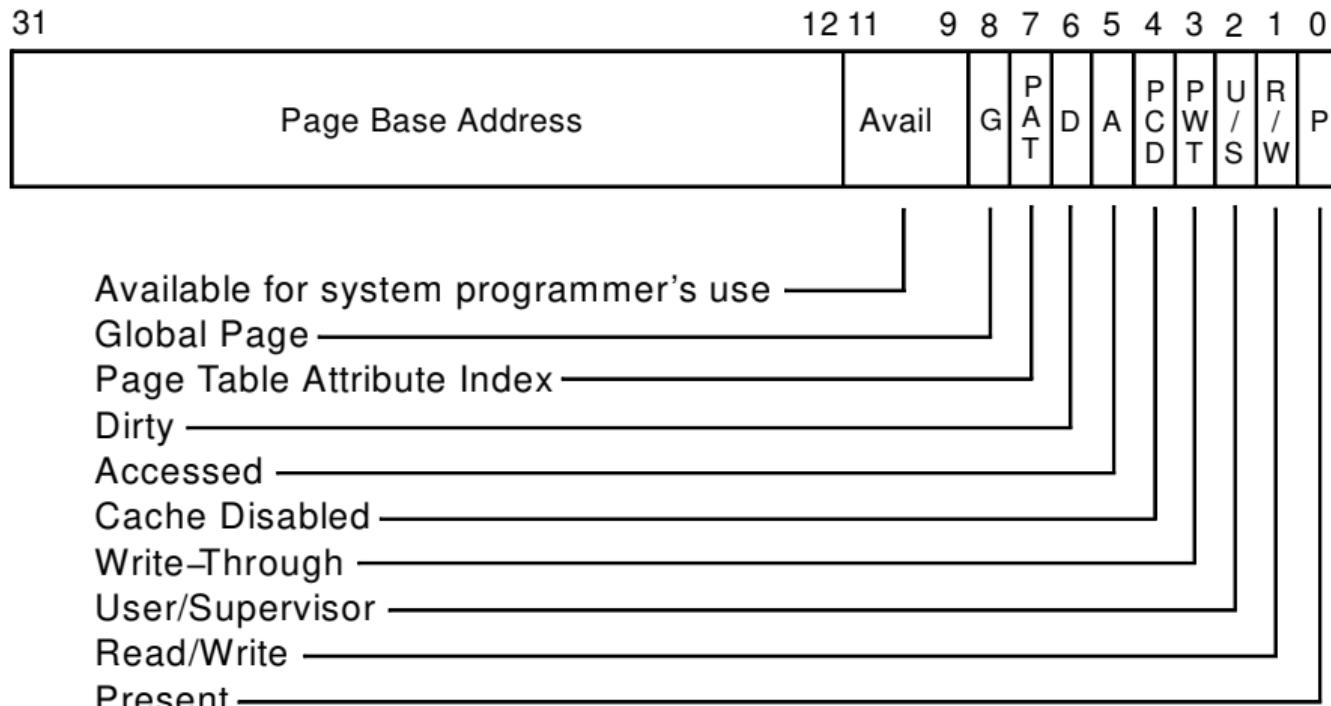
# x86 Page Directory Entry

Page-Directory Entry (4-KByte Page Table)



# x86 Page Table Entry

Page-Table Entry (4-KByte Page)



# x86 Hardware Segmentation

- x86 architecture also supports segmentation, e.g. registers for code, data and stack segments.
  - Also have registers `%fs` and `%gs` for base + offset calculations.
  - x86-64 architecture removes much segmentation support but kept `%fs` and `%gs`.
- *Generally you don't want both paging and segmentation:* too much overhead.
- There are some cases where it is useful: detect stack overflow or **thread-local storage**.
  - Before multi-threaded code, each Linux process has a global variable `errno`, which stored the error number of the latest system call.
  - With multithreaded code, each thread could overwrite this value before another thread could see it.
  - Solution: have this global variable be relative to `%fs` and have each thread have its own base address stored in `%fs`.
  - *Thread-local storage changes a global variable to local for each thread*, e.g. `errno`.

# Making Paging Fast

- x86 address translation requires 3 memory references per load/store
  - Look up the page table address in page directory.
  - Look up the physical page number in page table.
  - Access the physical page.
- *For speed, each core caches recently used translations.*
  - Called a **translation lookaside buffer** or **TLB**
  - Typical: 64–2K entries, 4-way to fully associative with 95% hit rate
  - Each TLB entry maps a virtual page number → physical page number + protection information
- *On each memory reference*
  - Check TLB, if entry present get physical address fast.
  - If not, walk the page tables, and then insert the translation into the TLB for next time  
(Must evict some entry)

# TLB details

- *The TLB operates at processor pipeline speed* → small, fast
- Complication: what to do when switch 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 the TLB entries valid.*
- E.g., x86 `invlpg` instruction
  - Invalidates a page translation in TLB
  - Must execute after changing a possibly used page table entry, e.g. set dirty bit.
  - Otherwise, hardware will miss page table change
- More complex on a multiprocessor: **TLB shootdown**, i.e. invalidate or update TLB entries.

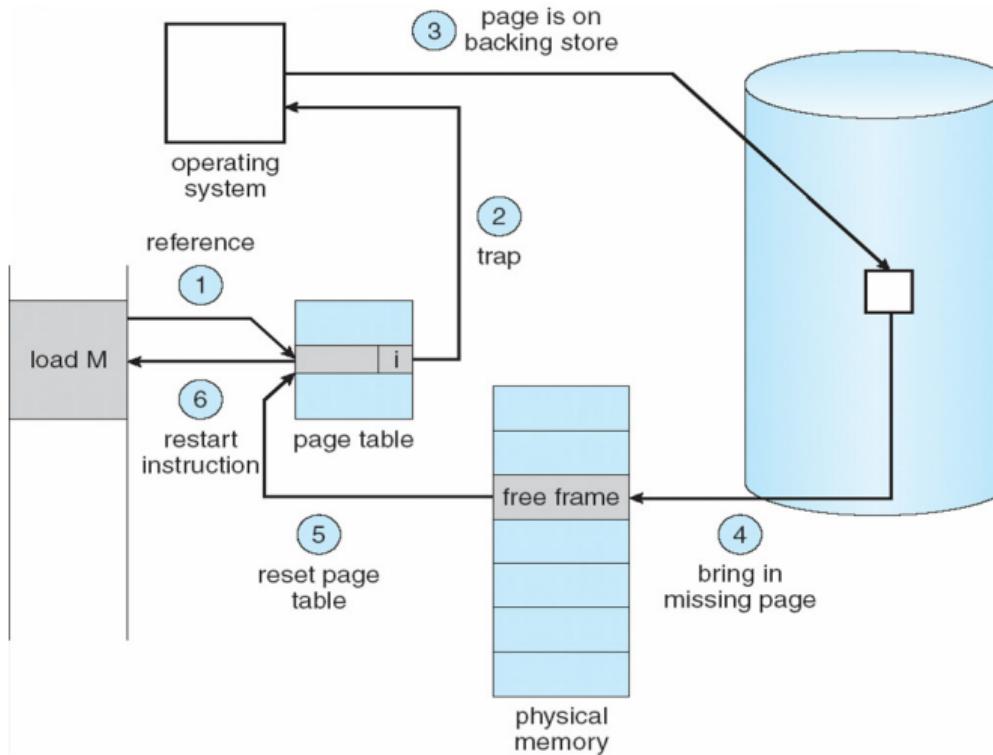
# Where does the OS live?

- 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
- *Kernel addresses are in the same address space as the user process.*
  - Use protection bits to prohibit user code from reading and writing kernel addresses.
- Typically all kernel text and most data *at same virtual address in every address space, upper half of memory.*
  - 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.

# Paging in day-to-day use

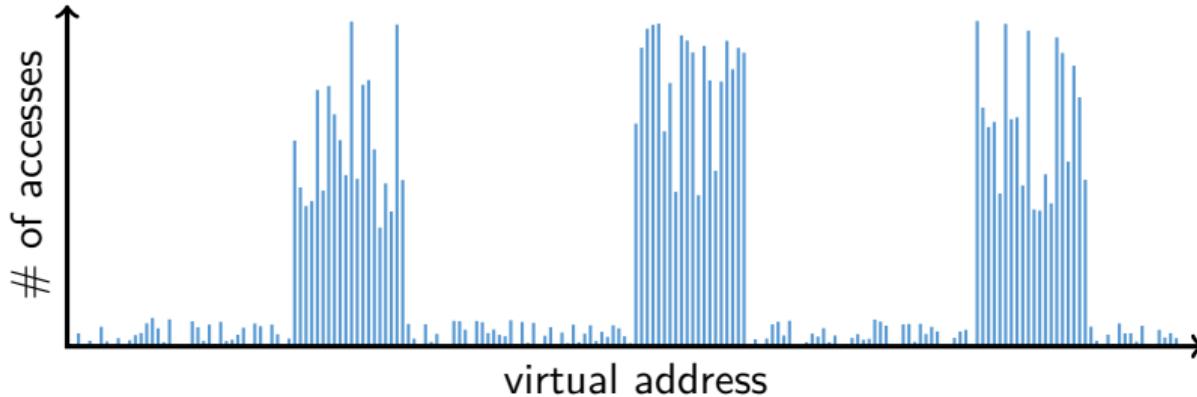
- Paging Examples
  - **Demand paging**: only load pages from the executable file, if they are needed.
  - BSS page allocation - i.e. global variables that have not been initialized, so space has to be allocated at run time. You did this in A1 with the `p_memsz - p_filesz` calculation.
  - **Copy-on-write** (e.g. `fork`, `mmap`, etc.) - share a reference rather than copy, but if a write happens, then make a copy.
  - Shared text
  - Growing the stack.
  - Shared libraries
  - Shared memory

## Subtopic: Paging



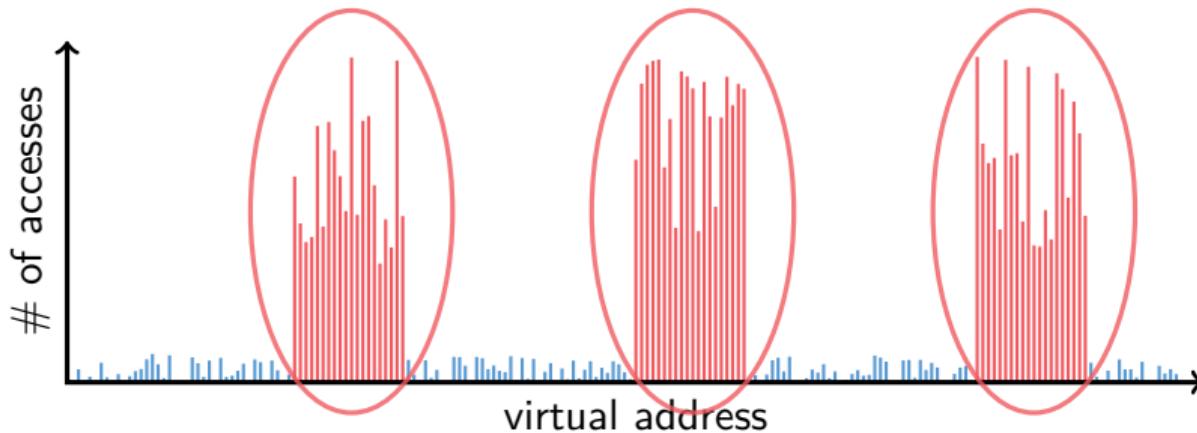
- Key Idea: Use secondary storage to simulate larger virtual than physical memory.

# Working set model



- *Secondary storage much, much slower than memory*: about 1000x for a typical SSD.
  - Goal: run at memory speed, not disk speed.
- 80/20 rule: 20% of memory gets 80% of memory accesses
  - Keep the hot 20% in memory.
  - Keep the cold 80% on disk.

# Working set model



- *Secondary storage much, much slower than memory*: about 1000x for a typical SSD.
  - Goal: run at memory speed, not disk speed.
- 80/20 rule: 20% of memory gets 80% of memory accesses
- Keep the hot 20% in memory.
  - Keep the cold 80% on disk.

# Working set model



- *Secondary storage much, much slower than memory*: about 1000x for a typical SSD.
  - Goal: run at memory speed, not disk speed.
- 80/20 rule: 20% of memory gets 80% of memory accesses
  - Keep the hot 20% in memory.
- Keep the cold 80% on disk.

# Paging challenges

- How to resume a process after a fault?
  - Need to save state and resume.
  - Process might have been in the middle of an instruction!
- Key Question: *What to fetch* from disk?
  - Just needed page or neighbouring pages as well.?
- Key Question: *What to evict* from RAM?
  - How to allocate physical pages amongst processes?
  - Which of a particular process's pages to keep in memory?

## Re-starting instructions

- *The hardware provides the kernel with information about the page fault.*
  - Faulting virtual address (in MIPS: `%c0_vaddr`)
  - Address of instruction that caused fault (in MIPS: `%c0_epc`)
  - Was the access a read or a write? Was it an instruction fetch?
  - Was it caused by an user attempting to access to kernel-only memory?
- The hardware must allow resuming after a fault.
- Idempotent instructions are easy to handle.
  - An **idempotent** instruction is one that can be repeated many times and get the same result.
  - Load word and store word are idempotent. Increment is not.
  - Key Idea: Just *re-execute any instruction that only accesses one address.*

# What to fetch

- At the very least: bring in the page that caused the page fault.
- Pre-fetch surrounding pages?
  - Reading two disk blocks approximately as fast as reading one.
  - If the application exhibits *spacial locality, then big win* to store and read multiple contiguous pages
- Also *pre-zero unused pages in idle loop.*
  - Need 0-filled pages for stack, heap, anonymously mmaped memory (memory for devices).
  - Zeroing them only on demand is slower.
  - Hence, many OSes zero freed pages while CPU is idle.

## Selecting physical pages

- May need to evict some pages (move from RAM to secondary storage).
  - More on eviction policies soon.
- May also have a choice of physical pages to fetch, e.g. avoid interfering with cache.
- Direct-mapped physical caches (an older method):
  - Virtual → Physical mapping can affect performance
  - Physical address  $A$  conflicts with  $kC + A$ , i.e.  $A + kC \bmod C = A$ .  
(where  $k$  is any integer,  $C$  is cache size)
  - Applications can conflict with each other or themselves.
  - Scientific applications benefit if consecutive virtual pages do not conflict in the cache.
  - Many other applications do better with random mapping.
  - Currently use more sophisticated methods e.g.  $n$ -way cache, fully associative cache.

# Superpages

- How should OS make use of “large” mappings.
  - x86 has 2MB or 4MB pages that might be useful.
  - Other processors (e.g. Alpha) provide more choices: 8KB, 64KB, 512KB, 4MB
- Sometimes there are more pages in L2 cache than there are TLB entries.
  - Don’t want costly TLB misses going to main memory.
- Idea: *have two-level TLBs*
  - Want to maximize hit rate in faster L1 TLB.
- OS can transparently support **superpages**. [Navarro]
  - Coalesce contiguous pages that are often accessed together to superpages (one big page).
  - “Reserve” contiguous physical pages if possible.
  - Does complicate evicting (esp. with dirty pages).

# Eviction Policies: FIFO eviction

- **FIFO Eviction**
- Strategy: *Evict oldest fetched page* in system.
- Example reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Result: 9 page faults in total.

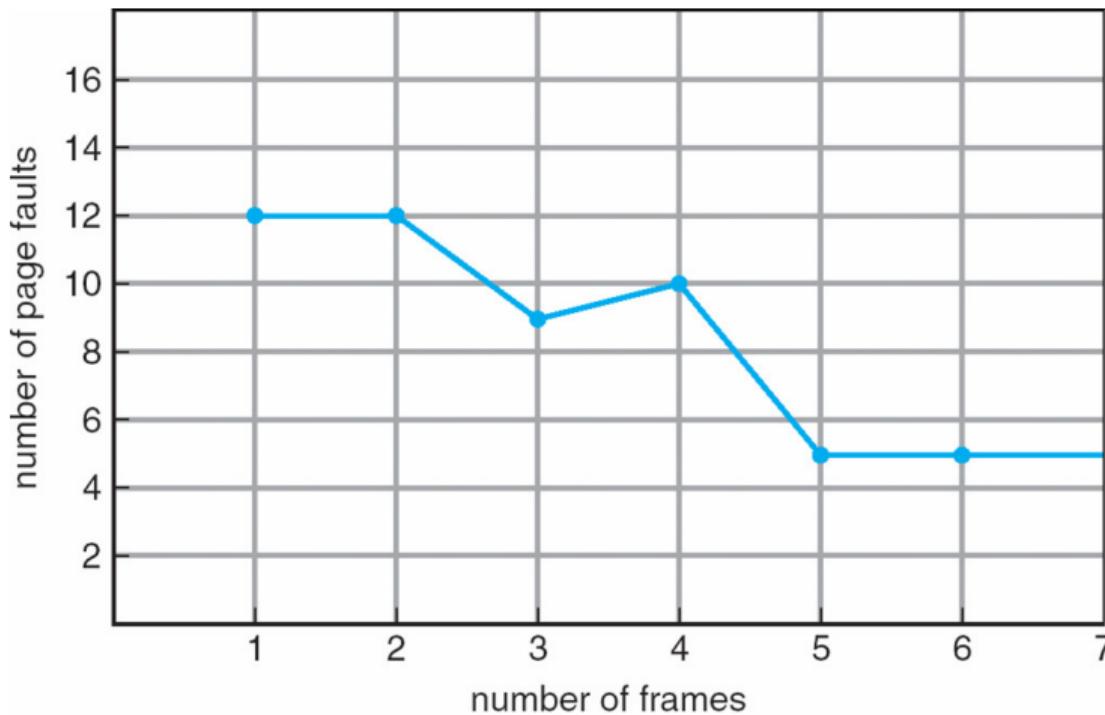
Refs	1	2	3	4	1	2	5	1	2	3	4	5
PPN 0	1	1	1	4	4	4	5	5	5	5	5	5
PPN 1		2	2	2	1	1	1	1	1	3	3	3
PPN 2			3	3	3	2	2	2	2	2	4	4
Fault?	x	x	x	x	x	x	x			x	x	

# Eviction Policies: FIFO eviction

- **FIFO Eviction**
- Strategy: *Evict oldest fetched page* in system.
- Same reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Result: *Adding a physical page → an additional page fault!*

Refs	1	2	3	4	1	2	5	1	2	3	4	5
PPN 0	1	1	1	1	1	1	5	5	5	5	4	4
PPN 1		2	2	2	2	2	2	1	1	1	1	5
PPN 2			3	3	3	3	3	3	2	2	2	2
PPN 3				4	4	4	4	4	4	3	3	3
Fault?	x	x	x	x			x	x	x	x	x	x

## Belady's Anomaly



- Key Observation: *Adding more physical memory doesn't always mean fewer faults.*

# Optimal page replacement

- **Optimal Page Replacement**
- What is optimal (if you knew the future)?
  - Strategy: *Replace page that will not be used for longest period of time in the future.*
- Same reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Result: 6 page faults.

Refs	1	2	3	4	1	2	5	1	2	3	4	5
PPN 0	1	1	1	1	1	1	1	1	1	1	4	4
PPN 1		2	2	2	2	2	2	2	2	2	2	2
PPN 2			3	3	3	3	3	3	3	3	3	3
PPN 3				4	4	4	5	5	5	5	5	5
Fault?	x	x	x	x			x			x		

# LRU page replacement

- **LRU Page Replacement**
- Strategy: *Approximate optimal with least recently used.*
  - Rationale: past behaviour often predicts the future behaviour.
- Same reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Result: 8 page faults.

Refs	1	2	3	4	1	2	5	1	2	3	4	5
PPN 0	1	1	1	1	1	1	1	1	1	1	1	5
PPN 1		2	2	2	2	2	2	2	2	2	2	2
PPN 2			3	3	3	3	5	5	5	5	4	4
PPN 3				4	4	4	4	4	4	3	3	3
Fault?	x	x	x	x			x		x	x	x	

# LRU page replacement

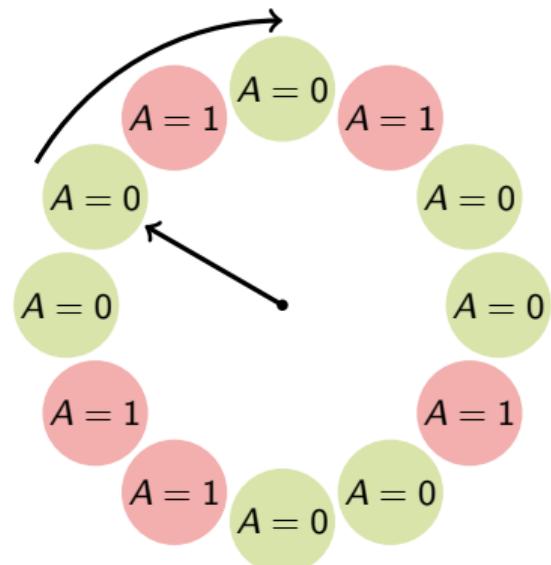
- **LRU Page Replacement**
- Strategy: *Approximate optimal with least recently used.*
  - Rationale: past behaviour often predicts the future behaviour.
- Problem 1: Can give the worst result possible, i.e. page fault every time.
  - E.g. With space for  $n$  physical pages, loop over  $n+1$  virtual pages.
- Problem 2: How to implement LRU efficiently?
  - Constraint: need something efficient.

# Straw man LRU implementations

- Idea # 1: Stamp PTEs with timer value
  - E.g., CPU has cycle counter
  - Automatically writes value to PTE on each page access
  - Scan page table to find oldest counter value = LRU page
  - Problem: Would double memory traffic and take  $n \log n$  time.
- Keep doubly-linked list of pages
  - On access remove page, place at tail of list
  - Problem: again, very expensive
- What to do?
  - Key Idea: *Approximate LRU*. Don't try to do it exactly!

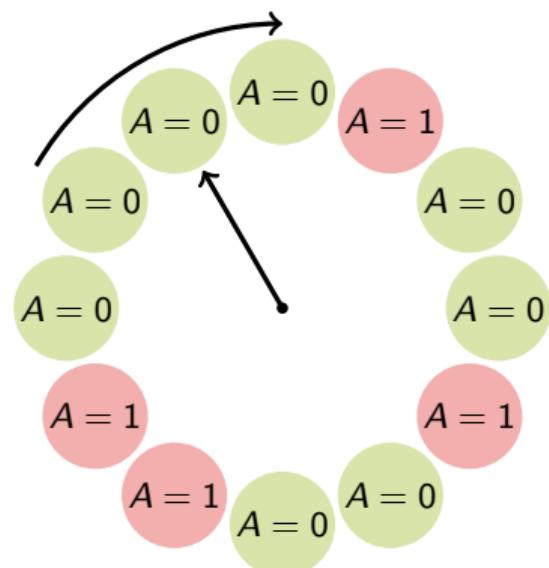
# Clock algorithm

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



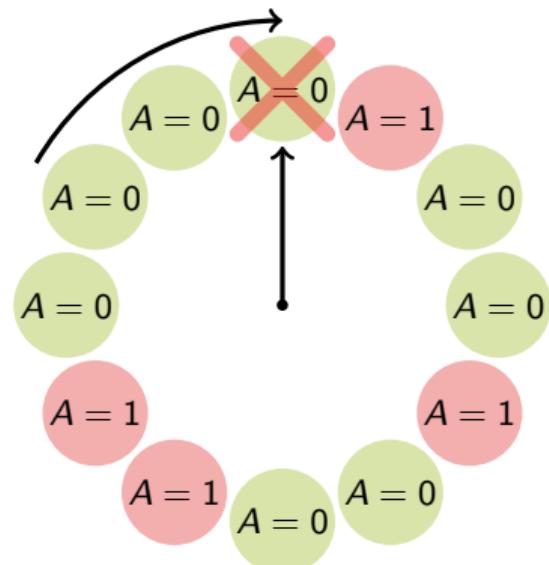
# Clock algorithm

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



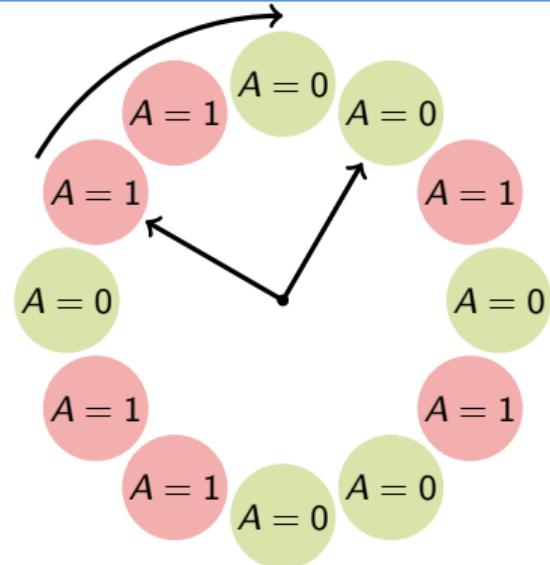
# Clock algorithm

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



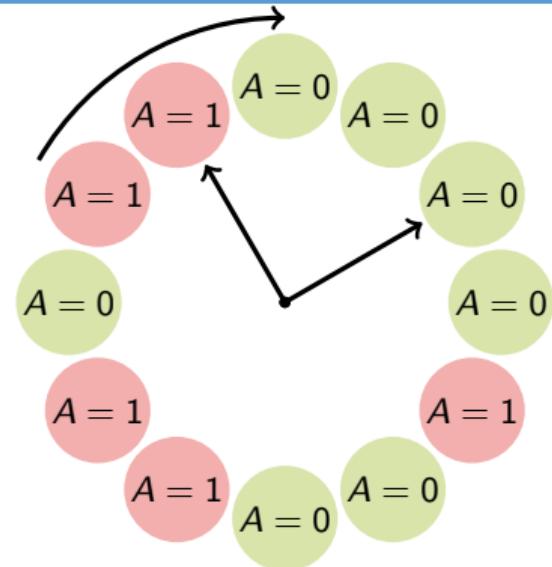
## Clock algorithm (continued)

- Large memory may be a problem.
  - Most pages referenced in long interval.
- Idea: *Add a second clock hand*
  - Two hands move in lockstep.
  - Leading hand clears A bits.
  - *Trailing hand evicts pages with A=0.*
- Can also take advantage of hardware Dirty bit
  - Page can be (accessed xor unaccessed) and (clean xor dirty).
  - Evict *clean pages before dirty ones* because no writeback.
- Or use  $n$ -bit accessed *count* instead just A bit
  - On sweep:  $count = (A \ll (n - 1)) | (count \gg 1)$
  - Evict page with lowest *count*.



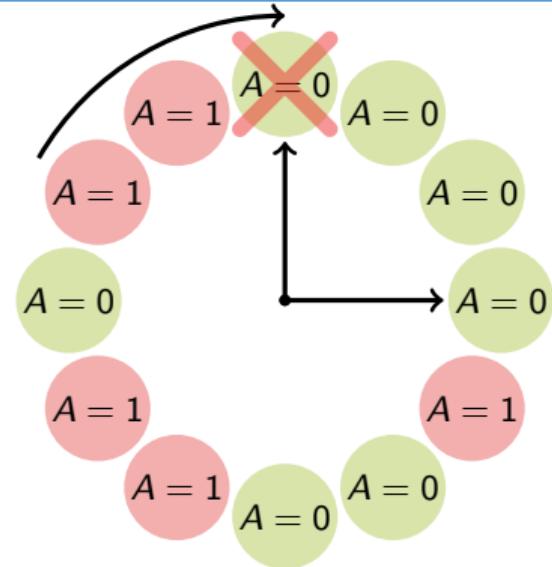
## Clock algorithm (continued)

- Large memory may be a problem.
  - Most pages referenced in long interval.
- Idea: *Add a second clock hand*
  - Two hands move in lockstep.
  - Leading hand clears A bits.
  - *Trailing hand evicts pages with A=0.*
- Can also take advantage of hardware Dirty bit
  - Page can be (accessed xor unaccessed) and (clean xor dirty).
  - Evict *clean pages before dirty ones* because no writeback.
- Or use  $n$ -bit accessed *count* instead just A bit
  - On sweep:  $count = (A \ll (n - 1)) | (count \gg 1)$
  - Evict page with lowest *count*.



## Clock algorithm (continued)

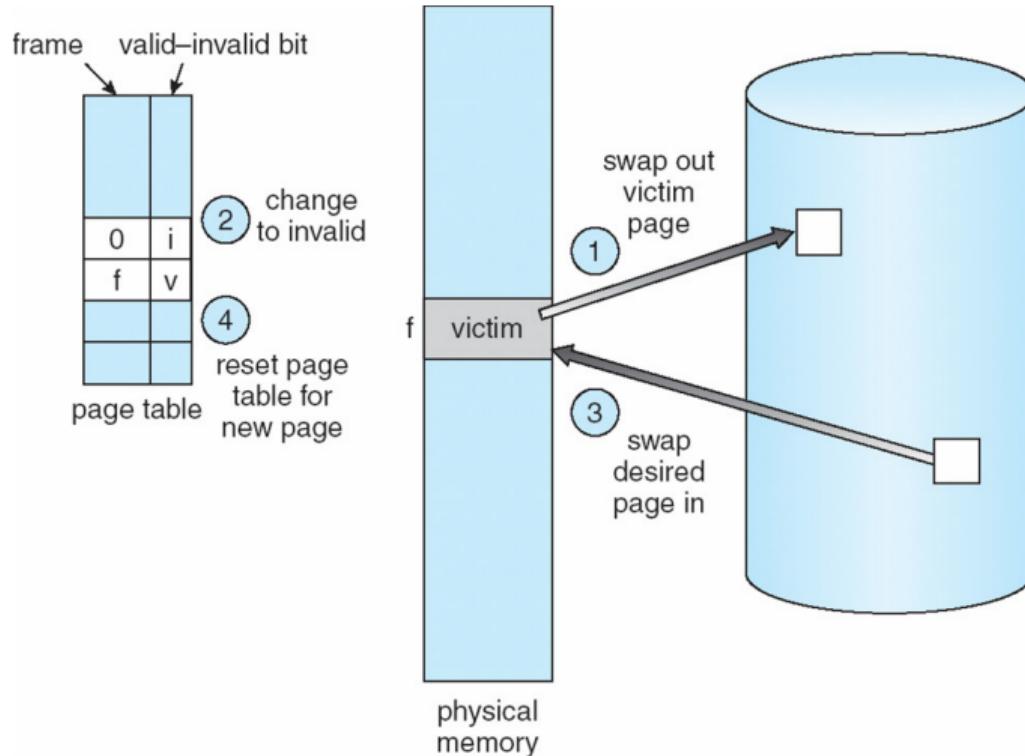
- Large memory may be a problem.
  - Most pages referenced in long interval.
- Idea: *Add a second clock hand*
  - Two hands move in lockstep.
  - Leading hand clears A bits.
  - *Trailing hand evicts pages with A=0.*
- Can also take advantage of hardware Dirty bit
  - Page can be (accessed xor unaccessed) and (clean xor dirty).
  - Evict *clean pages before dirty ones* because no writeback.
- Or use  $n$ -bit accessed *count* instead just A bit
  - On sweep:  $count = (A \ll (n - 1)) | (count \gg 1)$
  - Evict page with lowest *count*.



## Other replacement algorithms

- *Random eviction*
  - Simple to implement.
  - Not overly horrible (avoids Belady & pathological cases).
- *LFU* (least frequently used) eviction
- *MFU* (most frequently used) algorithm
- Neither LFU nor MFU used very commonly.
- Workload specific policies: Databases

# Naïve paging



- *Naïve page replacement use 2 I/Os per page fault:* swap out and swap in.

# Page buffering

- Idea: reduce # of I/Os on the critical path.
- Strategy: *Keep a pool of free page frames.*
  - On fault, still select victim page to evict (same as before).
  - But store page fetched from swap space into already free page from the pool.
  - Can resume execution while writing out victim page.
  - Then add victim page to free pool.
- Can also yank pages back from free pool.
  - Contains only clean pages, but may still have data.
  - If page fault on page still in free pool, reuse it.

# Page allocation

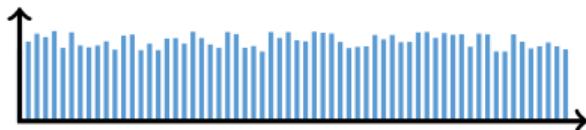
- Allocation can be *global or local*
- Global allocation doesn't consider page ownership.
  - E.g., with LRU, evict least recently used page of any process.
  - Works well if  $P_1$  needs 20% of memory and  $P_2$  needs 70%:
  - Doesn't protect you from memory hogs.  
(imagine  $P_2$  keeps looping through array that is size of memory.)
- Local allocation isolates processes (or users)
  - Separately determine how much memory each process should have
  - Then use LRU/clock/etc. to determine which pages to evict within each process.

# Thrashing

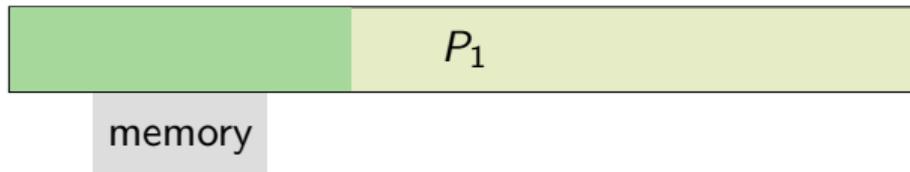
- **Thrashing** is when an application is *constantly swapping pages in and out*, preventing it from making progress at a reasonable rate.
- Cause: Processes require more memory than system has.
  - Each time one page is brought in, another page, whose contents will soon be referenced, is swapped out.
  - Processes will spend much of their time blocked, waiting for pages to be fetched from secondary storage.
  - Secondary storage will be very close to 100% utilization, but the system is not making much progress.
- What we wanted: more virtual memory than installed RAM with access time the speed of primary memory.
- What we got: memory with the access time of secondary storage.

# Reasons for thrashing

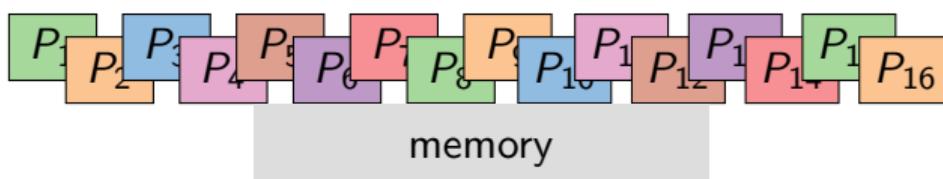
- The memory access pattern does not exhibit temporal locality (past  $\neq$  future). I.e. the 80/20 model does not apply.



- Hot memory does not fit in physical memory.



- Each process fits individually, but there are too many for system.
  - At least it is possible to address this case.



# Dealing with thrashing

## Approach 1: The Working Set Model

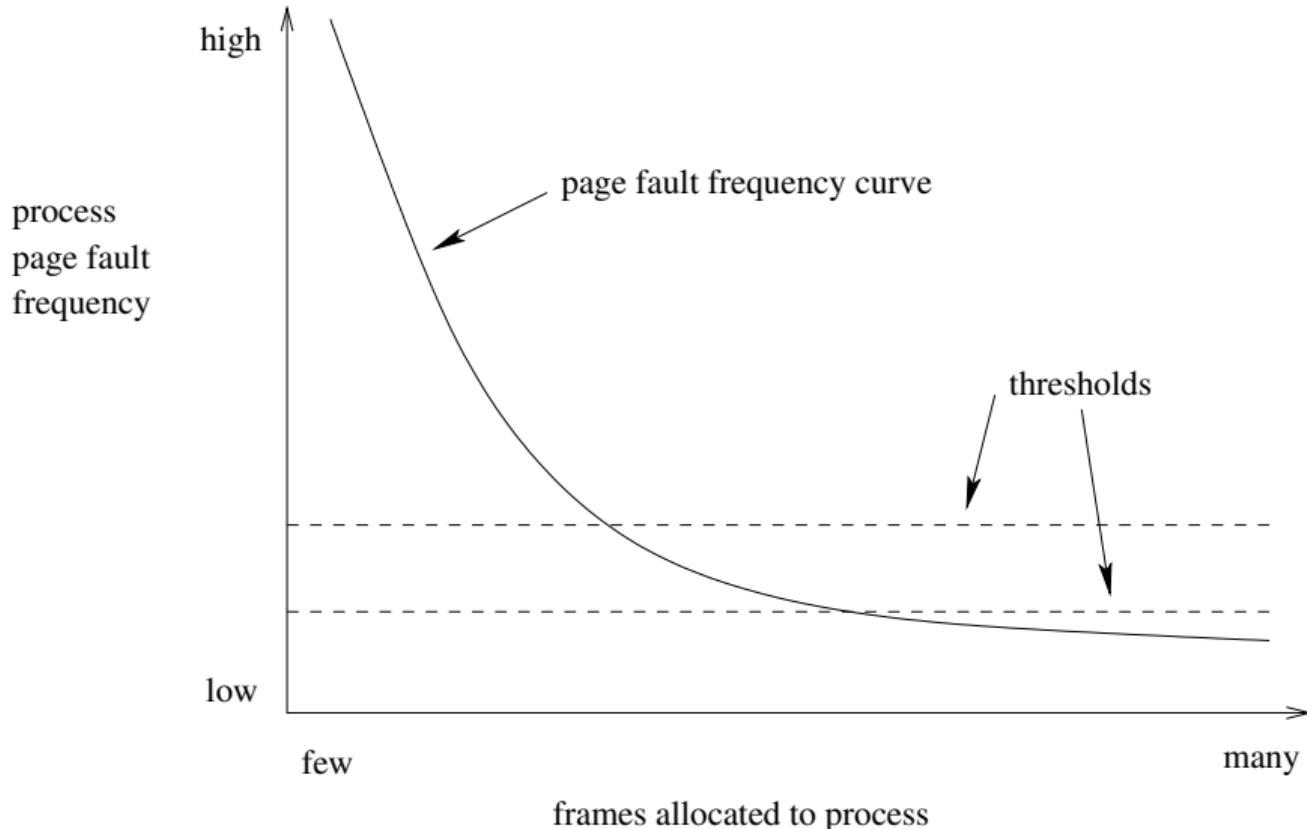
- According to the 80/20 model, *some portion of a programs address space will be heavily used* and the remainder will not. This heavily used portion is called the **working set** of the process.
- The set of pages that are currently in primary memory is called the **resident set**.
- Primary memory (i.e. RAM) can be thought of as a cache for secondary storage, i.e. it stores the recently used code and data.
- Thrashing viewed from a caching perspective: given locality of reference, how big a cache does the process need?
- Or: how much memory does the process need in order to make reasonable progress (i.e. how much memory is needed to store its working set)?
- Only run processes whose memory requirements can be satisfied.

# Dealing with thrashing

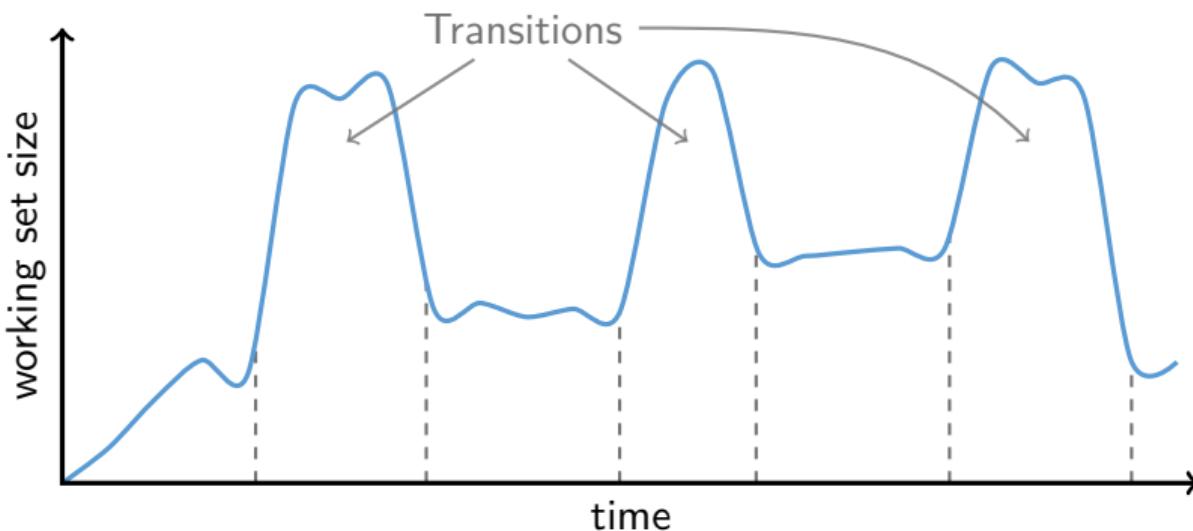
## Approach 2: Page Fault Frequency (PFF)

- Thrashing viewed as poor ratio of page faults to instruction execution.
- $PFF = \text{page faults} / \text{instructions executed}$
- If PFF rises above the High threshold, the process needs more memory.  
Not enough memory on the system? Swap out.
- If PFF sinks below the Low threshold, memory can be taken away.

# A Page Fault Frequency Plot

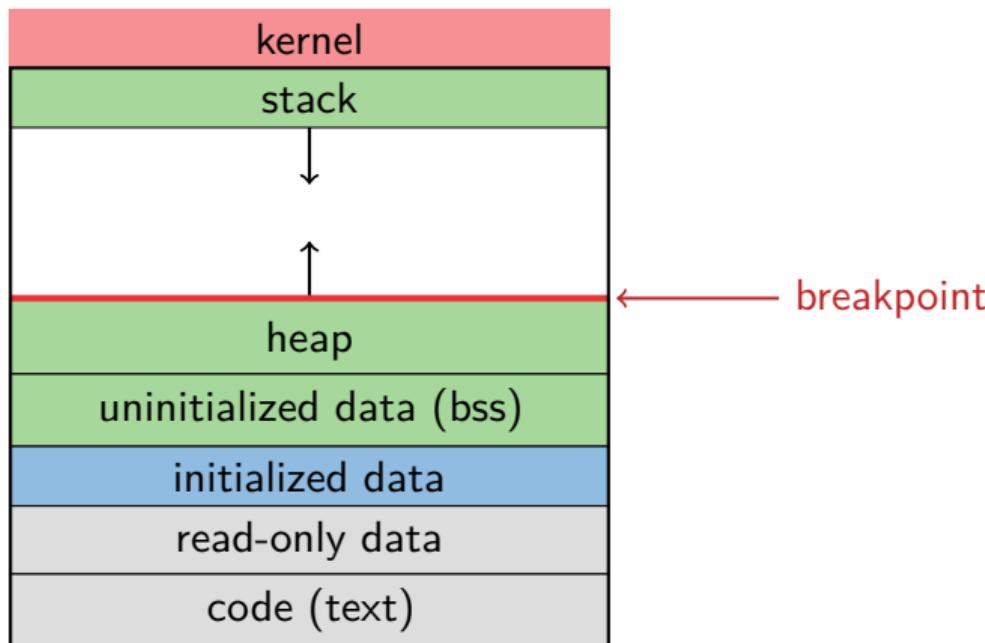


# Working sets



- The *working set changes across phases of using a program.*
- PFF can increase during phase transitions, e.g. launching browser  $\Rightarrow$  going to youtube.com.

## Recall typical virtual address space

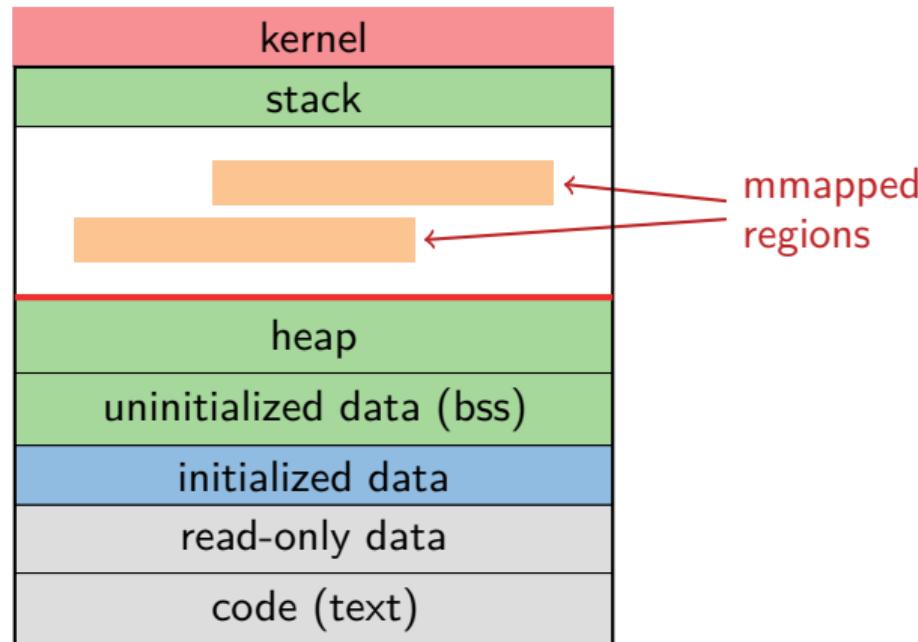


- Recall: Dynamically allocated memory goes in heap.
- The top of the heap is called the **breakpoint**.
- *Addresses between breakpoint and the stack are invalid.*

## Early VM system calls

- The OS tracks the **breakpoint** – i.e. the top of the heap.
- *Attempting to access the memory region between the breakpoint and the stack causes a segmentation violation.* Pages have not been allocation to this region, so they are marked as invalid in the process's page tables.
- Linux provides system calls to allocate more virtual memory in the heap.
  - `char *brk (const char *addr);`  
Set and return new value of breakpoint.
  - `char *sbrk (int incr);`  
Increment value of the breakpoint and return the old value.
- **malloc** can be implemented in terms of **sbrk**.
  - The OS gives **malloc** an initial amount of memory which it gives out in small slices, and **malloc** can request more when that initial allotment is insufficient.
  - But it is hard to “give back” physical memory to system.

# Memory mapped files



- *Other memory objects can exist between the heap and the stack using `mmap`.*

## mmap system call

- `void *mmap (void *addr, size_t len, int prot,  
int flags, int fd, off_t offset)`
  - *Treat a file as if it were memory*, i.e. map the file specified by `fd` to the virtual address `addr` except that changes are backed up to the file. It can be shared with other processes.
  - If `addr` is `NULL`, then let the kernel choose the address.
- `prot` – specifies the protection of region
  - OR of `PROT_EXEC`, `PROT_READ`, `PROT_WRITE`, `PROT_NONE` (i.e. no access).
- `flags`
  - `MAP_SHARED` – modifications seen by everyone
  - `MAP_PRIVATE` – modifications are private
  - `MAP_ANON` – anonymous memory (when `fd` is `-1`), i.e. no file associated with this address range.

## More VM system calls

- `int munmap(void *addr, size_t len)`  
*Remove* memory-mapped object from address space.
- `int mprotect(void *addr, size_t len, int prot)`  
*Change protection* on pages to PROT\_...
- `int msync(void *addr, size_t len, int flags);`  
*Flush changes* of mmapped file to backing store, i.e. the file.
- `int mincore(void *addr, size_t len, char *vec)`  
Return *which pages present* in RAM (i.e. core) vs. swap space in `vec`.
- `int madvise(void *addr, size_t len, int behav)`  
*Advise* the OS on memory use: e.g.  
page access could be `MADV_RANDOM` vs. `MADV_SEQUENTIAL` (so don't prefetch)  
In the near future `MADV_WILLNEED` or `MADV_DONTNEED` these pages (so don't prefetch).

## Exposing page faults

The function `sigaction` is used to specify *what action to take* (i.e. what function to call) for `SIGSEGV` (the signal raised on invalid memory access) or any other signal.

```
struct sigaction {      /* the Linux struct for signal actions */
    union {                  /* system will have one of these signal handlers */
        void (*sa_handler)(int);
        void (*sa_sigaction)(int, siginfo_t *, void *);
    };
    sigset_t sa_mask;      /* signal mask to apply */
    int sa_flags;          /* options, e.g. for dealing with children */
};

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

If `oact` is not `NULL`, then save the old `sigaction` in `oact`.

## Example: OpenBSD/i386 siginfo

```
struct sigcontext {  
    int sc_gs; int sc_fs; int sc_es; int sc_ds;  
    int sc_edi; int sc_esi; int sc_ebp; int sc_ebx;  
    int sc_edx; int sc_ecx; int sc_eax;  
  
    int sc_eip; int sc_cs; /* instruction pointer */  
    int sc_eflags;           /* condition codes, etc. */  
    int sc_esp; int sc_ss; /* stack pointer */  
  
    int sc_onstack;          /* sigstack state to restore */  
    int sc_mask;             /* signal mask to restore */  
  
    int sc_trapno;  
    int sc_err;  
};
```

*Get the context (register values) of a thread.* Linux uses `ucontext_t` but it is too big to fit on the slide.

# VM tricks at user level

- The combination of `mprotect` (set protection) and `sigaction` (set action) are very powerful.
  - *Can use these in user-level functions to perform kernel-like actions.* [Appel]
  - E.g., for a fault, unprotect page (make it unaccessible), then return from signal handler.
- Technique used in object-oriented databases
  - Bring in objects on demand.
  - Keep track of which objects may be dirty.
  - Manage memory as a cache for much larger object DB.
- Other interesting applications
  - Useful for some garbage collection algorithms.
  - Snapshot processes (copy on write).

## Case study: 4.4 BSD Overview

- Windows and most UNIX systems separate the VM system into two parts.
  - **VM PMap**: manages the *hardware interface* (e.g. TLB in MIPS).
  - **VM Map**: *machine independent representation of memory*.
- VM Map consists of one or more **objects** (or **segments**).
  - Each object consists of a contiguous `mmap()`.
  - Objects can be backed by files, shared between processes, or both.
- VM PMap manages the hardware (often caches mappings).
- 4.4 BSD VM is based on [Mach VM].

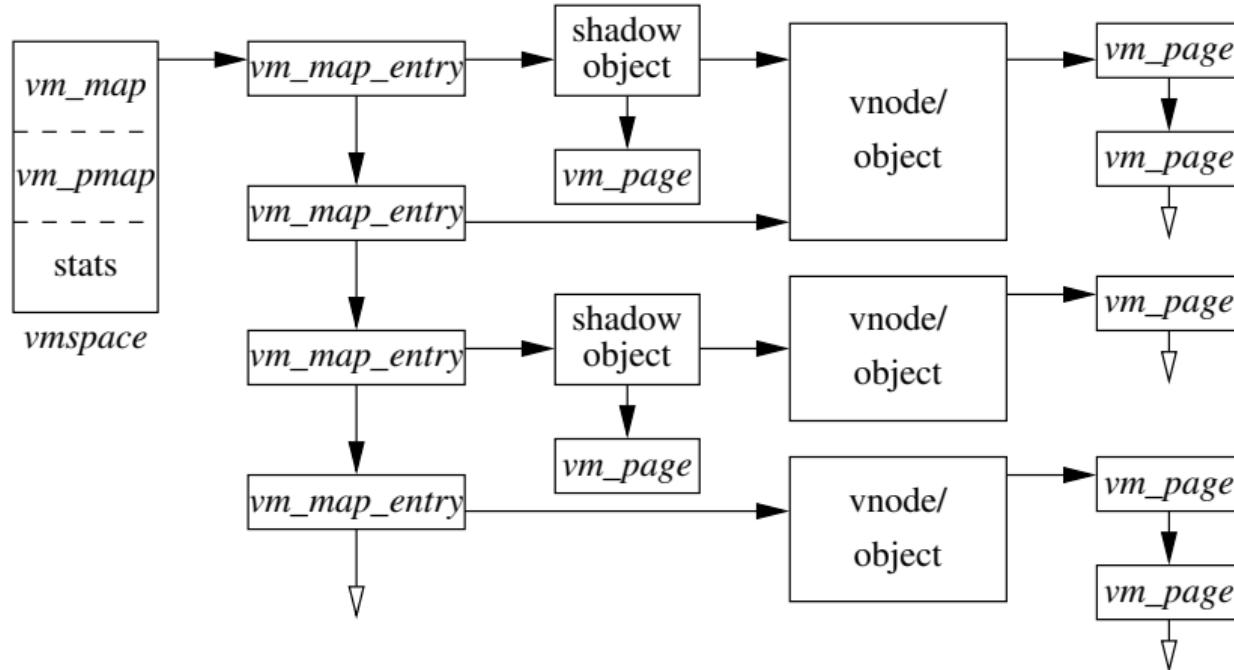
## Case study: 4.4 BSD Operation

- *Calls* to `mmap()`, `munmap()`, `mprotect()`
  - Update the VM Map.
  - VM Map routines call into the VM PMap to invalidate and update the TLB.
- *Page faults*
  - Exception handler calls into the VM PMap to load the TLB.
  - If the page isn't in the PMap, call VM Map code.
- *Low memory options*
  - PMap is a cache and can be discarded during a low memory condition

## Case study: 4.4 BSD VM system [McKusick]

- Each process has a `vm_space` structure containing
  - `vm_map` – *machine-independent virtual address space*
  - `vm_pmap` – *machine-dependent data structures*
  - `statistics` – e.g. for syscall `getrusage()` (get resource usage) such as user time, system time or page faults.
- `vm_map` is a linked list of `vm_map_entry` structs
  - `vm_map_entry` covers contiguous virtual memory
  - points to a `vm_object` struct
- `vm_object` is source of data
  - e.g. `vnode`, which is a file-like interface for kernel-memory objects.
  - points to list of `vm_page` structs (one per mapped page).
- `shadow objects` point to other objects for copy-on-write

## 4.4 BSD VM data structures



## Pmap (machine-dependent) layer

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

## Example uses

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

# What happens on a fault?

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

# Paging in day-to-day use

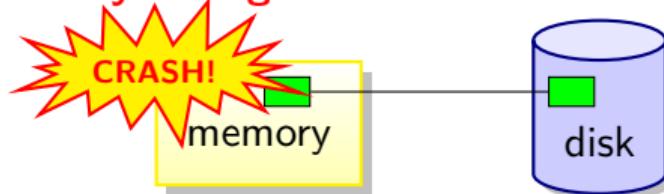
- *Demand paging*
  - Read pages from `vm_object` of executable file
- *Copy-on-write* (`fork`, `mmap`, etc.)
  - Use shadow objects
- *Growing* the stack, BSS *page allocation*
  - A bit like copy-on-write for `/dev/zero`
  - Can have a single read-only zero page for reading
  - Special-case write handling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

# The Challenges of File System

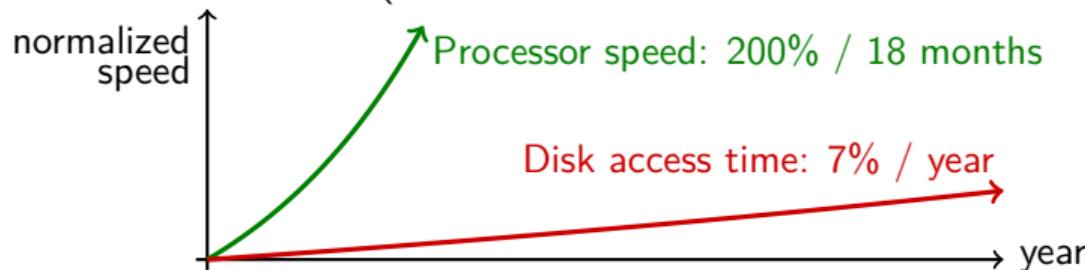
- File systems (FS): traditionally the hardest part of OS
  - More papers on file systems than any other single topic.
- Main tasks of file system:
  - *Don't go away* (ever). Need it to boot up.
  - *Associate bytes with name* (files), e.g. named data objects.
  - Associate names with each other (directories)
- Can implement on a variety of media: solid state drive (SSD), hard disk, optical disk (DVD), over a network (NAS), in memory, in non-volatile RAM (NVRAM), on magnetic tape, ticker tape (paper).
- We'll focus on hard disks and generalize later.
- The first part focuses on files, directories, and a bit of performance.

# Why disks are different

- **Secondary storage** is the first *state info we've seen that doesn't disappear* at shutdown.



- Therefore: this is where all important state ultimately resides. Called persistent storage.
- Downside: it is slow (milliseconds access vs. nanoseconds for memory)



- It has a large capacity (100–1,000x bigger than memory).
  - *Key Challenge:* How to organize a large collection of ad hoc information?
  - Taxonomies! ⇒ file systems are a general way to make them.

# Magnetic Disk vs. Flash Memory

	Disk	MLC NAND Flash	DRAM
Smallest write	sector	sector	byte
Atomic write	sector	sector	byte/word
Random read	8 ms	75 $\mu$ s	50 ns
Random write	8 ms	300 $\mu$ s*	50 ns
Sequential read	100 MB/s	250 MB/s	> 1 GB/s
Sequential write	100 MB/s	170 MB/s*	> 1 GB/s
Cost	\$0.04/GB	\$0.65/GB	\$10/GiB
Persistence	Non-volatile	Non-volatile	Volatile

DRAM is dynamic RAM, commonly referred to as RAM.

\*Flash write performance degrades over time. Different varieties degrade at different speeds:  
SLC (most longevity), MLC, TLC, QLC (least longevity).

# Disk review

- *Secondary storage reads and writes in units of sectors* (a larger size), not bytes.
  - Read or write single sector and possibly adjacent groups.
- How to write a single byte? **read-modify-write**.
  - Read in the entire sector containing the byte. 
  - Modify that byte. 
  - Write entire sector back to secondary storage.
  - Key: if cached, no need to read sector in again.
- Sector = unit of atomicity.
  - Sector write done completely, even if the system crashes in the middle of the operation.  
Hard drives saves up enough momentum and energy to complete it without power.
- Tradeoff: Larger atomic units have to be created by the OS.

## Some useful trends

- Hard disk *bandwidth* (transfer speed) and *cost per bit improving exponentially*.
  - Similar to CPU speed, memory size, etc.
- Seek time and rotational delay improving very slowly. More on this later.
  - Why? require moving physical object (disk arm).
- *Secondary storage accesses are a huge system bottleneck* and getting worse.
  - *Key Idea:* Bandwidth increase lets system (pre-)fetch large chunks for about the same cost as small chunk.
  - Trade bandwidth for latency if you can get lots of related stuff.
  - How to get related stuff? Cluster together.
- *Memory size increasing faster than typical workloads.*
  - *More and more of workload fits in file cache.*
  - Secondary storage traffic changes: mostly writes and new data.
  - Trend does not necessarily apply to big server-side jobs.

# Files: named bytes on disk

- File abstraction:
  - User's view: a *named sequence of bytes* or a named data object.
  - FS's view: a file is a collection of data blocks.
  - File system's job: translate name and offset to data blocks: $\{file, offset\} \rightarrow \text{FS} \rightarrow \text{data block address}$
- File operations:
  - Create a file. Delete a file.
  - Read from a file. Write to a file.
- Goal:** want (1) operations to have as few secondary storage accesses as possible (i.e. group related things) and (2) use minimal space overhead.

# What's hard about grouping blocks?

- Like page tables, *file system metadata are simply data structures used to construct mappings*

- Page table: map virtual page # to physical page #



- File metadata: map byte offset in a file to a data block address



- Directory: map name to data block address or file #



# File Systems vs. Virtual Memory

- In both settings, the goal is **location transparency** i.e. the ability to access data without knowing its location.
- In some ways, FS has easier job than VM:
  - Using processor time for FS mappings is OK (hence no TLB) since FS access is less frequent than memory and access is slow, so *thread blocks waiting for data*.
  - Page tables deal with sparse address spaces and random access, whereas files are often denser (0, ... filesize-1) and *access is often sequential*.
- In some ways FS's problem is harder:
  - Each layer of translation = potential secondary storage access.
  - *Space a huge premium!* (But disk is huge?!?!) Reason?  
Cache space never enough; amount of data you can get in one fetch never enough.
  - *File size range* is very extreme: Many files <10 KB, some files many GB, e.g. videos.

# Some working intuitions

- *FS performance dominated by the number of FS accesses.*
  - Say each access costs  $\sim 10$  milliseconds
  - Touch the disk 100 extra times = 1 second
  - Can do a *billion* ALU ops in same time!
- *Transfer time is only a small part of the total access time:*  
access time = **seek time** + **rotational delay** + transfer time
  - 1 sector:  $5\text{ms} + 4\text{ms} + 5\mu\text{s}$  (i.e.  $\approx (512 \text{ B}) / (100 \text{ MB/s}) \approx 9 \text{ ms}$ .)
  - 50 sectors:  $5\text{ms} + 4\text{ms} + .25\text{ms} = 9.25\text{ms}$
  - Can get *50x more data for only  $\sim 3\%$  more overhead!*
- Observations that might be helpful:
  - All blocks in a file tend to be used together, sequentially.
  - All files in a directory tend to be used together.
  - All names (subdirectories and files) in a directory tend to be used together.

# Common addressing patterns

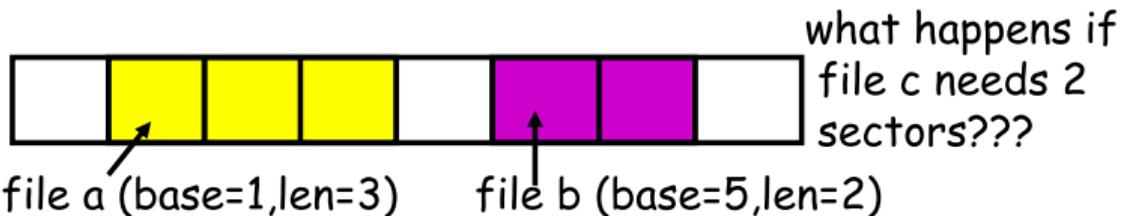
- *Sequential:*
  - File data is typically processed in sequential order.
  - By far the most common mode.
  - Example: editor writes out new file, compiler reads in file, etc
- *Random access:*
  - Address any block in file directly without passing through predecessors
  - Examples: data set for demand paging, databases
- *Keyed access*
  - Search for block with particular values
  - Examples: associative data base, index, text searches
  - Usually not provided by OS (although Windows, macOS and Linux now all provide text searching).

## Problem: how to track file's data

- File system management:
  - Need to keep track of where file contents are in secondary storage.
  - Must be able to use this to map byte offset to data block.
  - *Structure tracking a file's sectors is called an index node* or **inode**.
  - Inodes must be stored within the file system, too.
- Things to keep in mind while designing file structure:
  - Most files are small.
  - Much of the disk is allocated to large files.
  - Many of the I/O operations are made to large files.
  - We want good sequential and good random access.  
(what do these require?)

# FS Attempt 1: contiguous allocation

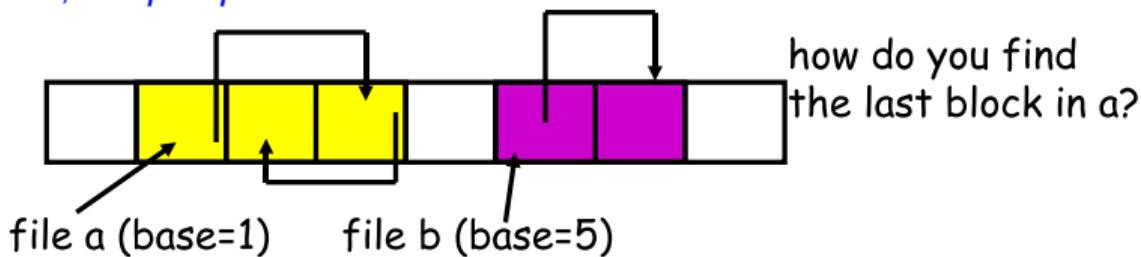
- **Extent-based**, i.e allocate files like segmented memory.
  - When creating a file, make the user *pre-specify its length and allocate all space at once.*
  - Inode contents: location and size



- Example: IBM OS/360
- Pros?
  - Simple, fast access, both sequential and random
- Cons? (Think of corresponding VM scheme)
  - External fragmentation.

## FS Attempt 2: Linked files

- Basically *a linked list of data blocks*.
  - Keep a linked list of all free blocks
  - Inode contents: a pointer to file's first block
  - In each block, keep a pointer to the next block.*



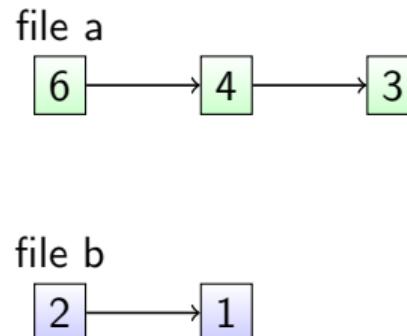
- Examples (sort-of): Alto, TOPS-10, DOS FAT
- Pros?
  - Easy dynamic growth and sequential access, no fragmentation.
- Cons?
  - (1) Non-contiguous blocks cause poor access times. (2) Poor non-sequential access. (3) Pointers take up room in block, skewing alignment.

## FS Attempt 3: FAT (simplified)

- Uses linked files, but *links reside in fixed-sized file allocation table (FAT)* rather than in the blocks.

Directory (5)      FAT (16-bit entries)

a: 6	0 free
b: 2	1 eof
	2 1
	3 eof
	4 3
	5 eof
	6 4
	...



- Still do pointer chasing, but can cache entire FAT so can be cheap compared to disk access.
- Used in DOS but variants (FAT32 and exFAT) are typically used for USB flash drives.

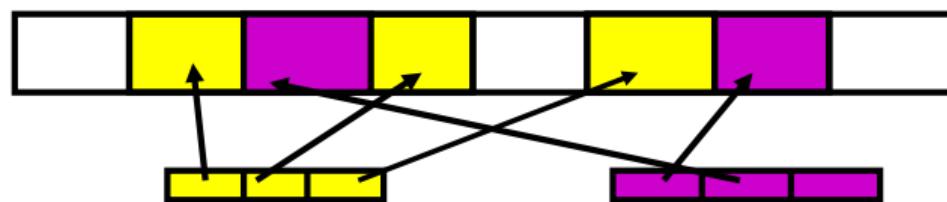
## FS Attempt 3: FAT discussion

- Entry size = 16 bits
  - What's the maximum size of the FAT?  $2^{16} = 65,536$  entries.
  - Given a 512 byte block, what's the maximum size of FS?  $2^{16} \times 2^9 = 2^{25} = 32$  MiB.
  - Pros of bigger blocks? handle larger files Cons? more internal fragmentation
- *Space overhead of FAT is trivial:*
  - 2 bytes / 512 byte block =  $\sim 0.4\%$  (Compare to Unix)
- *Reliability:* i.e. how to protect against errors?
  - Create duplicate copies of FAT on disk
  - State duplication a very common theme in reliability
- *Bootstrapping*, i.e. where is root directory?
  - Fixed location on disk: 

FAT	(opt) FAT	root dir	...
-----	-----------	----------	-----

## FS Attempt 4: Indexed files

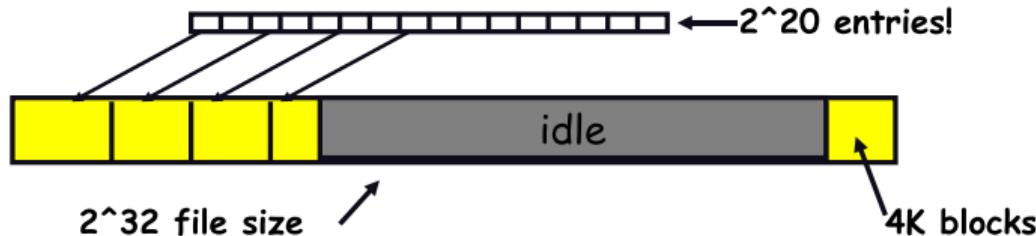
- *Each file has an array holding all of its block pointers.*
  - *It is similar to a page table, so it will have similar issues.*
  - Max file size fixed by array's size (static or dynamic?)
  - Allocate array to hold file's block pointers during file creation
  - Allocate actual blocks on demand using free list.



- Pros?
  - Both sequential and random access are easy.
- Cons?
  - Mapping table requires large chunk of contiguous space. I.e. the same problem we had with Attempt 1 continuous allocation.

## FS Attempt 4: Indexed files

- Issues same as in page tables.

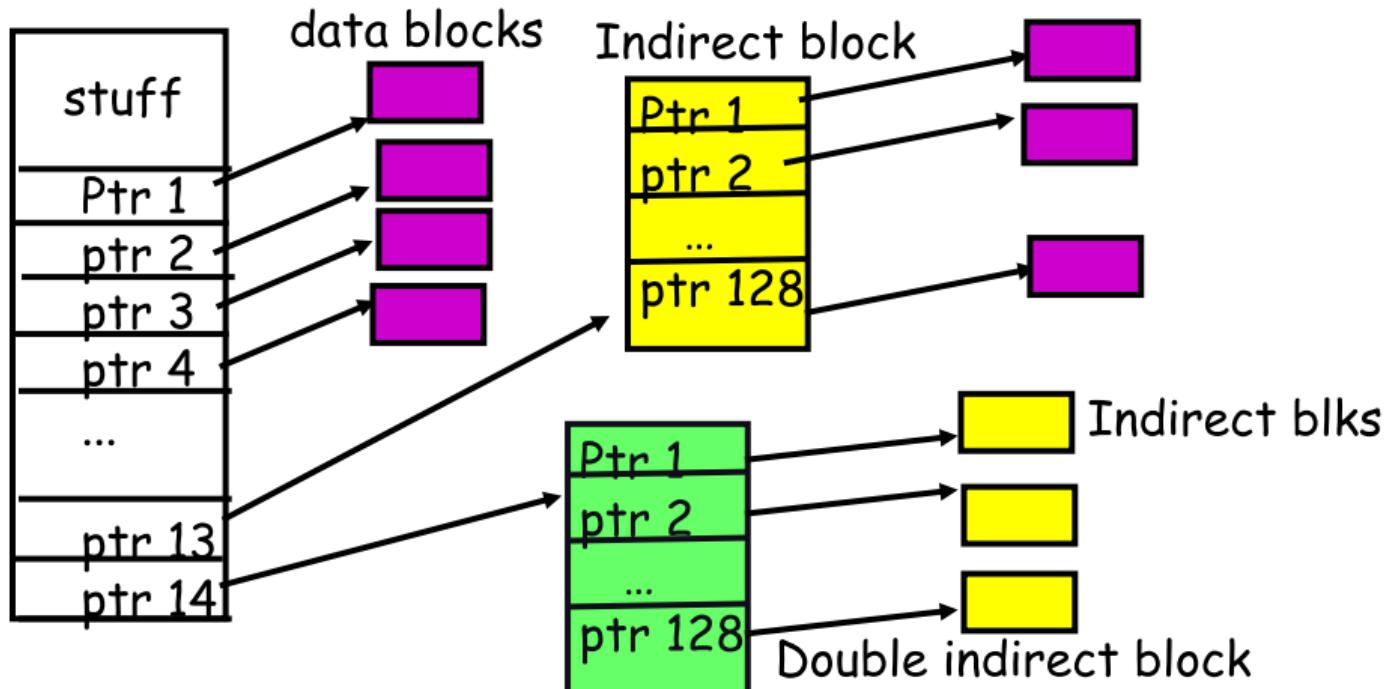


- Since large possible file size needs to be accommodated, there can be lots of unused entries.
- Large actual size  $\Rightarrow$  table needs large contiguous allocation.
- Solve in the same way as multilevel page tables:* small regions with index array, this array with another array, ... Downside? more FS accesses, which are expensive



## FS Attempt 5: Multi-level indexed files (old BSD FS)

- Solve problem of first block access slow, i.e. response time.
- inode = 14 block pointers + “stuff” such as owner, R/W/Execute permissions.

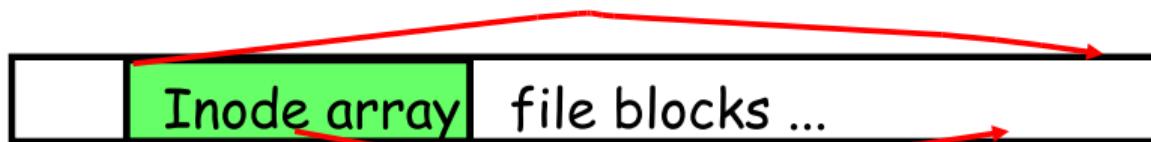


## FS Attempt 5: Multi-level indexed files (old BSD FS)

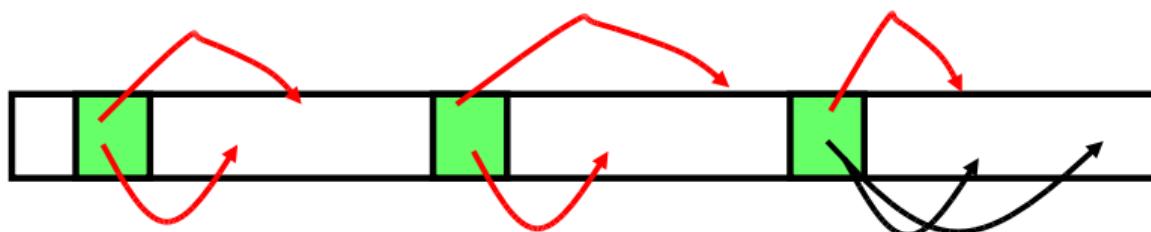
- *Pros:*
  - Simple, easy to build, fast access to small files.
  - Maximum file length fixed, but large.
- *Cons:*
  - What is the worst case # of accesses?
  - What is the worst-case space overhead? (e.g., 13 block file)
- An empirical problem:
  - Because you allocate blocks by taking them off unordered freelist, metadata and data get strewn across disk.

# More about inodes

- *Inodes are stored in a fixed-size array.*
- The size of the array is fixed when the disk is initialized and cannot be changed.
- It lives in known location, originally at one side of disk:



- Now metadata is smeared across it (why?) because of indirect and double indirect blocks.



- *The index of an inode in the inode array* called an **i-number**.
- Internally, the OS refers to files by their i-number.
- When file is opened, inode brought in memory.
- Written back when modified and file closed or time elapses.

# Directories

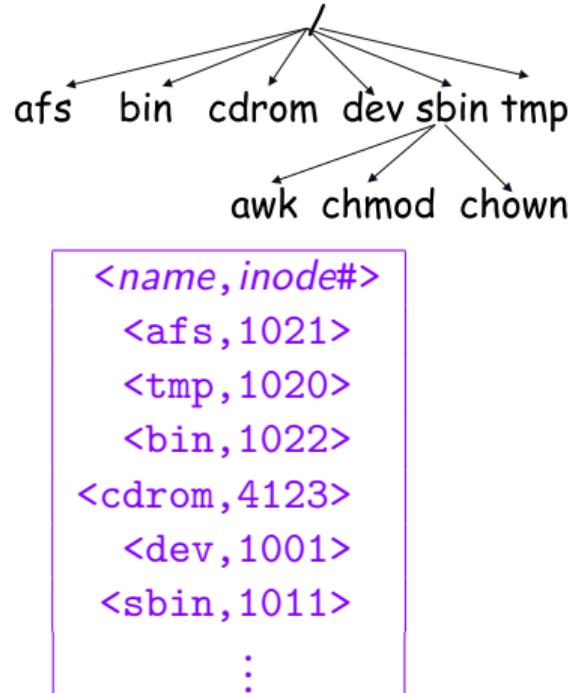
- Problem:
  - “Spend all day generating data, come back the next morning, want to use it.”
    - F. Corbato, on why files and directories were invented.
- Approach 0: Have users remember where on disk their files are
  - (E.g., like remembering your social security or bank account #)
- People want human digestible names.
  - Therefore *use directories to map human friendly names to i-numbers.*
- Next: What is in a directory and why?

# A short history of directories

- Approach 1: *Single directory for entire system.*
  - Put directory at known location on disk.
  - Directory contains  $\langle \text{name}, \text{i-number} \rangle$  pairs.
  - If one user uses a name, no one else can use the same name.
  - Many ancient personal computers (1970s and early 80s) work this way.
- Approach 2: *Single directory for each user.*
  - Still clumsy, and `ls` on 10,000 files is really slow.
- Approach 3: *Hierarchical name spaces*
  - Allow directory to map names to *files or other directories*.
  - File system forms a tree (or graph, if links allowed.) More about this later.
  - Large name spaces tend to be hierarchical (IP addresses, domain names, scoping in programming languages, etc.)

# Hierarchical Unix

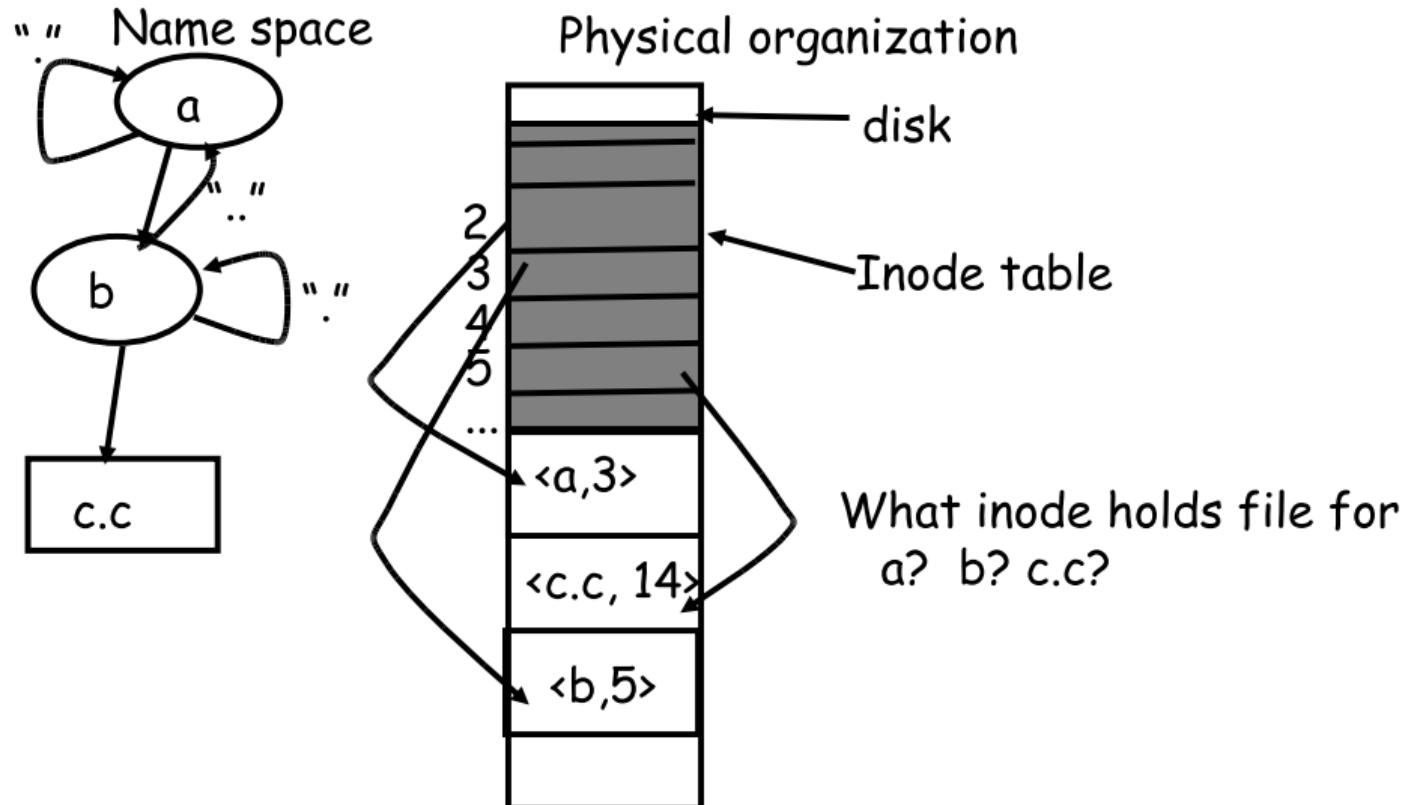
- Hierarchical name spaces used since CTSS (1960s)
  - Unix picked it up and used it.
- *Directories stored just like regular files*
  - Special inode-type byte set for directories.
  - User's can read just like any other file.
  - Only special syscalls can write (why?)
  - Inodes at fixed disk location.
  - File pointed to by the index may be another directory
  - Makes FS into hierarchical tree.
  - What needed to make a DAG? No links up tree.
- Simple, plus speeding up file ops speeds up directory ops!



# Naming magic

- Bootstrapping: Where do you *start looking for a file?*
  - Root directory always inode #2.
  - Inodes 0 and 1 historically reserved for things such as bad blocks.
- *Special names:*
  - Root directory: “/”
  - Current directory: “.”
  - Parent directory: “..”
- Special names not implemented in FS:
  - User’s home directory: “~”
  - Globbing: “foo.\*” expands to all files starting “foo.”
  - **Glob** refers to expanding filenames that have wildcards such as \* in them.
- Using the given names, only need two operations to navigate the entire name space:
  - `cd name`: move into (i.e. change the context to) directory `name`.
  - `ls`: enumerate all names (files, directories, etc.) in the current directory (context).

## Unix example: /a/b/c.c



## Unix example: /a/b/c.c

- By default, the “/” (i.e. root) directory corresponds to inode 2, so start looking there.
- Inode 2 points to a data block (or blocks) that contain the entry  $\langle a, 3 \rangle$ , so the directory **a** has inode 3.
- Looking at inode 3 we can find the data block (or blocks) that contains the contents of the directory **a**.
- One entry in that directory file is  $\langle b, 5 \rangle$ , so the directory **b** has inode 5.
- Looking at inode 5 we can find the data block that contains the contents of the directory **b**.
- One entry in that directory file is  $\langle c.c, 14 \rangle$ , so the file **c.c** has inode 14.
- Looking at inode 14 we can find all the metadata about the file **c.c**.
- Note: Each directory (including **/**) contains entries for **.** and **..**

## Default context: working directory

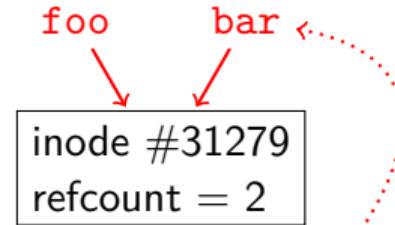
- It is cumbersome to constantly specify full path names.
  - In Linux and Unix, *each process associated with a current working directory* (or cwd) i.e. its file system context.
  - File names that do not begin with “`/`” are interpreted as being relative to the cwd; otherwise pathname translation happens as before.
  - This provides an example of locality, i.e. `./a.out` will refer to different files in different contexts (i.e. the cwd of the process).
- Shells, such as bash, track a *default list of active contexts*.
  - This list is referred to as a **search path** for programs you can run.
  - Given a search path `A:B:C`, a shell will check in directory `A`, then `B`, then `C`.
  - Users can escape the search path by using explicit paths, i.e. using either a full (`/bin/foo`) or relative path (`./foo`).

# Hard and soft links (synonyms)

- More than one directory entry can refer to a given file.

Both Linus and Unix store a count of pointers, called **hard links**, to an inode.

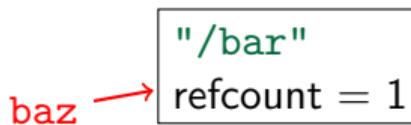
The command “`ln foo bar`” creates a synonym (**bar**) for *file foo*.



- Soft links** (a.k.a. **symbolic links**) are synonyms for *names*.

- It points to a file (or directory) *name*, but the original file can be deleted from underneath it (or never even exist).
- They are implemented like directories: the inode has a **symlink** bit set and its file contents are the name of link target.

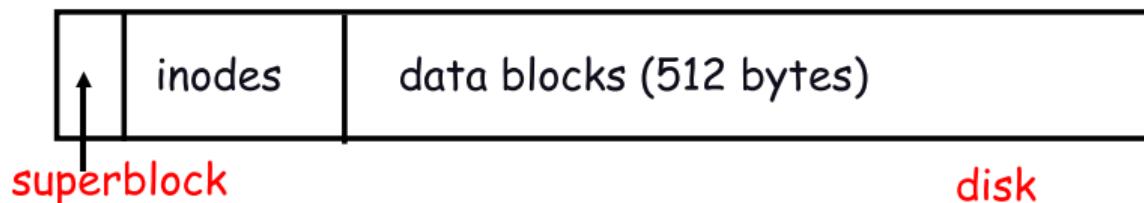
`ln -s /bar baz`



- When the file system encounters a symbolic link it automatically translates it (if possible).

# Case study: speeding up FS

- The original Unix file system FS was simple and elegant.



- Components:
  - Data blocks - contain the file (including directories) data.
  - Inodes - contain the meta information about the file.
  - Hard links
  - Superblock** (specifies information about the file system such as the number of blocks in the FS, counts of max # of files, pointer to head of the list of free data blocks).
- Problem: slow
  - Only gets 20Kb/sec (2% of disk maximum) even for sequential disk transfers!

# A plethora of performance costs

- *Blocks too small* (512 bytes)
  - Too many layers of mapping indirection
  - Transfer rate low (get one block at time)
- *Poor clustering of related objects*:
  - Consecutive file blocks not close together.
  - Inodes far from data blocks.
  - Inodes for directory not close together.
  - Poor enumeration performance: e.g., “`ls`”, “`grep foo *.c`”
- Usability problems (won't discuss this further).
  - 14-character file names is restrictive.
  - Can't atomically update file in crash-proof way.
- Next: how the Fast File System (FFS) fixes these (to a degree). [McKusick]

# Problem: Internal fragmentation

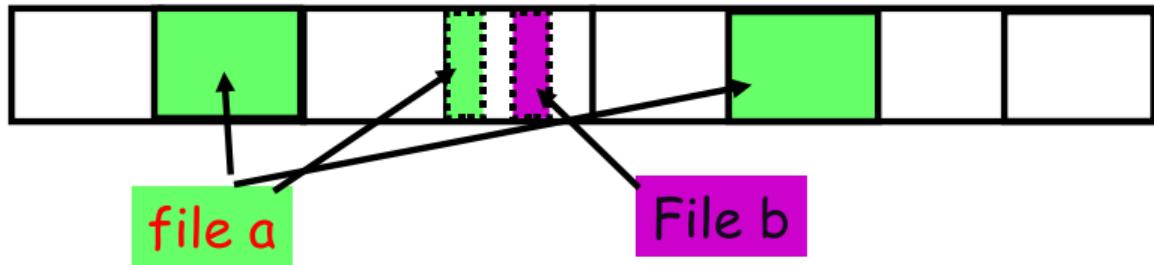
- The block size was too small in the Unix file system.
- Q: Why not just make the block size bigger?

Block size	space wasted	file bandwidth
512	6.9%	2.6%
1024	11.8%	3.3%
2048	22.4%	6.4%
4096	45.6%	12.0%
1MB	99.0%	97.2%

- A: *Bigger block sizes increase bandwidth, but also increase internal fragmentation.*
  - Use an idea from malloc: split unused portion of a block.

# Solution: fragments

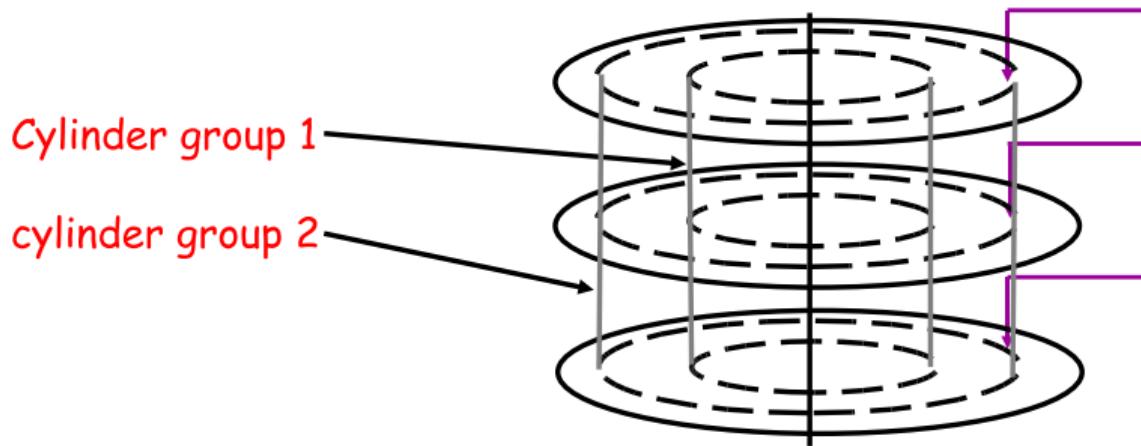
- BSD Fast File System:
  - Has a larger block size (4,096 or 8,192 bytes).
  - It also *allows large blocks to be chopped into smaller ones*, called fragments.
  - Used for little files and pieces at the ends of files.



- What is the best way to eliminate internal fragmentation?
  - Variable sized splits of course.
  - Instead FFS uses fixed-sized fragments (1024, 2048) like malloc, so they *can easily be recombined or reused*.

# Clustering related objects in FFS

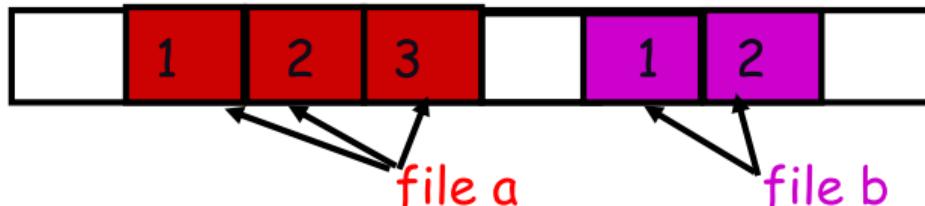
- Group one or more consecutive cylinders into a **cylinder group**.



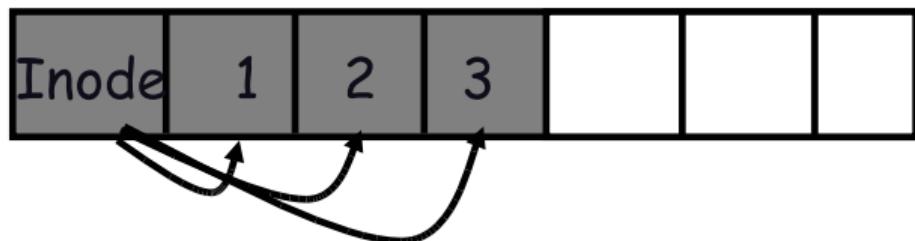
- Key Idea:* The system can access any block in a cylinder without performing a seek. The next fastest place is an adjacent cylinder.
- The file system tries to put everything related in the same cylinder group.
- And tries to put everything not related in different groups.

# Clustering in FFS

- FFS tries to *put sequential blocks in adjacent sectors* because sequential access is common.



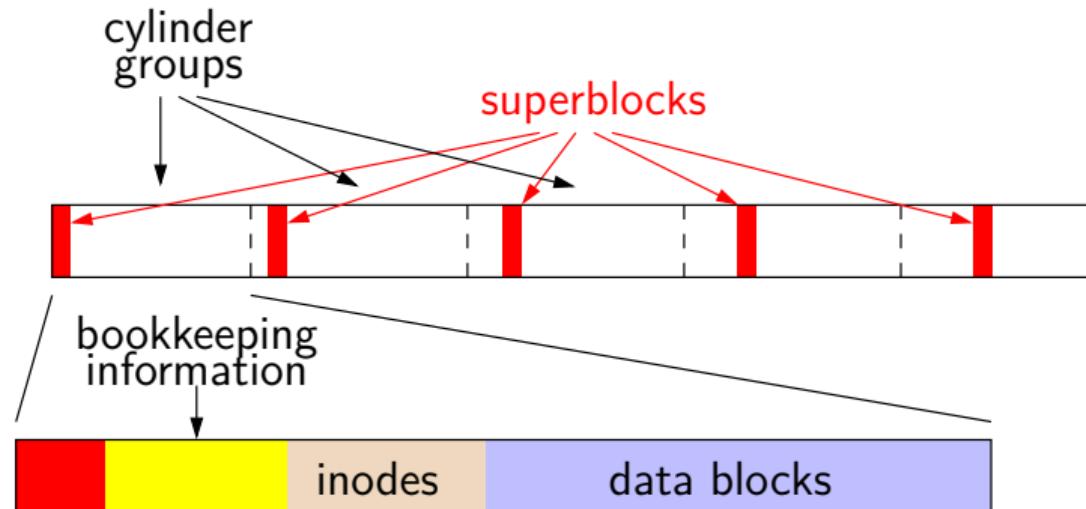
- FFS tries to *keep the inode and file data together* in same cylinder because if you look at the inode, most likely you will look at the data too.



- FFS tries to *keep all inodes in a directory together* in the same cylinder group because if you access one name, you frequently access many, e.g., “`ls -l`” .

# What does disk layout look like?

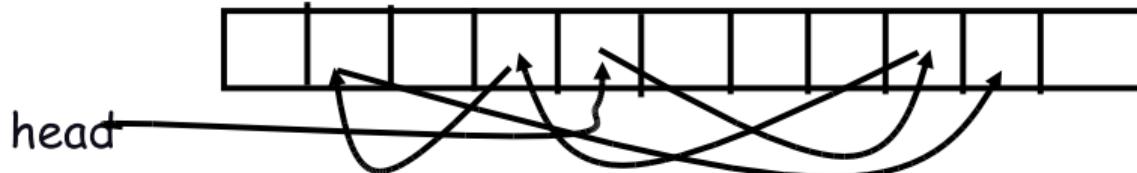
- Key Idea: *Each cylinder group basically a mini-Unix file system:*



- How how to ensure there's space for related stuff?
  - Place different directories in different cylinder groups.
  - Keep a “free space reserve” so can allocate near existing things.
  - When file grows too big (1MB) send its remainder to different cylinder group.

## Finding space for related objs

- Old Unix (and DOS): maintained a linked list of free (i.e. available) data blocks.
  - Need another data block ⇒ remove one from the head of the list. Fast and easy.



- Downside: free list gets jumbled over time. *Finding adjacent blocks hard and slow.*
- FFS: switch to *bit-map of free blocks*.
  - 101010111111000001111111000101100
  - Easier to find contiguous blocks.
  - Small, so usually keep the entire bit-map in memory.
  - Time to find free block increases if there are fewer free blocks.

# Using a bitmap

- Usually *keep entire bitmap in memory*:
  - 4G disk / 4K byte blocks. How big is the bit-map?  $2^{32}/2^{12}/2^3 = 2^{17} = 128$  KB.
- Want to allocate a block close to block  $x$ ?
  - Check for blocks near  $x/32$  (assuming the size of the array elements are 32 bits).
  - If disk almost empty, will likely find one near by.
  - As disk becomes full, search becomes more expensive and less effective.
- *Trade space for time*, specifically search time and file access time.
- Keep a reserve (e.g, 10%) of disk always free, ideally scattered across disk.
  - Don't tell users (`df` can get to 110% full).
  - Only root can allocate blocks once FS 100% full.
  - With 10% free, can almost always find one of them free.

# So what did we gain?

- Performance improvements:
  - Able to get 20-40% of disk bandwidth for large files
  - 10-20x faster than the original Unix file system!
  - Better small file performance (why?) clustering of related items.
- Is this the best we can do? No.
- FFS is block based rather than *extent based*.
  - Could have *named contiguous blocks with single pointer and length* (as done in Linux ext2fs, XFS).
- Writes of metadata done synchronously. *Write asynchronously* (i.e. when convenient).
  - Hurts small file performance where the metadata is larger compared to the file data.
  - Make asynchronous with write-ordering (“soft updates”) or logging/journaling . . . more next lecture.
  - Play with semantics (/tmp file systems).

## Other hacks

- Obvious:
  - *Big file cache* since RAM is getting larger.
- Fact: no rotation delay if getting the whole track.
  - How to use?
- Fact: *transfer cost negligible*.
  - Recall: Can get 50x the data for only  $\sim 3\%$  more overhead.
  - 1 sector:  $5\text{ms} + 4\text{ms} + 5\mu\text{s}$  (i.e.  $\approx (512 \text{ B}) / (100 \text{ MB/s}) \approx 9 \text{ ms}$ ).
  - 50 sectors:  $5\text{ms} + 4\text{ms} + .25\text{ms} = 9.25\text{ms}$
  - How to use?
- Fact: if transfer amount is huge and blocks are clustered, seek + rotation delays are negligible.
  - *LFS*: Idea: Hoard data and write it out a MB at a time

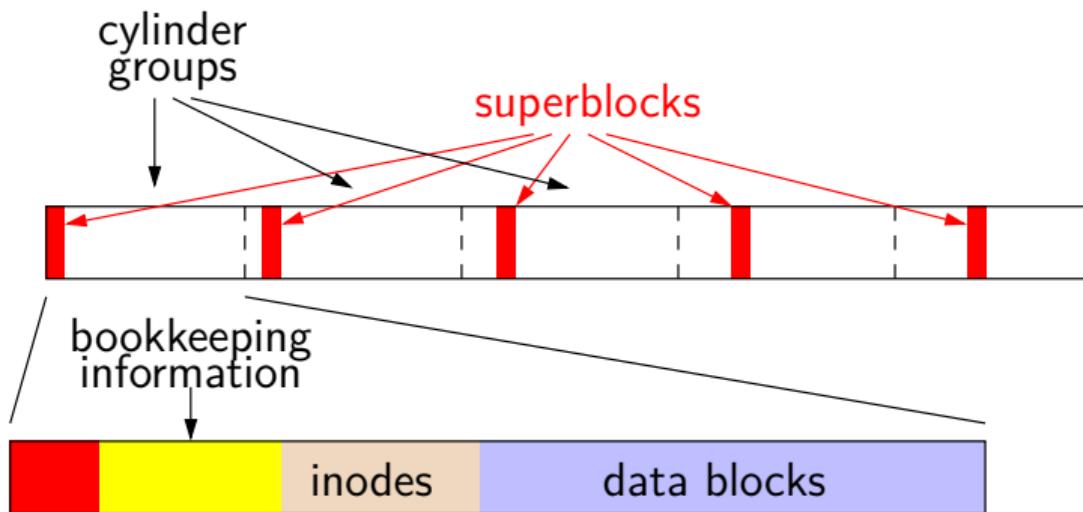
## Summary: FFS background

- 1980s improvement to original Unix FS, which had:
  - 512-byte blocks
  - Free blocks in linked list
  - All inodes at beginning of disk
  - Low throughput: 512 bytes per average seek time
- Unix FS performance problems:
  - Transfers only 512 bytes per disk access
  - Eventually random allocation → 512 bytes / disk seek
  - Inodes far from directory and file data
  - Within directory, inodes far from each other
- Also had some usability problems:
  - 14-character file names a pain
  - Can't atomically update file in crash-proof way

## Summary: FFS [McKusic] basics

- Change block size to at least 4K
  - To avoid wasting space, use “fragments” for ends of files
- Cylinder groups spread inodes around disk
- Bitmaps replace free list
- FS reserves space to improve allocation
  - Tunable parameter, default 10%
  - Only superuser can use space when over 90% full
- Also made usability improvements:
  - File names up to 255 characters
  - Atomic *rename* system call
  - Symbolic links assign one file name to another

# Summary: FFS disk layout



- Each cylinder group has its own:
  - Superblock
  - Bookkeeping information
  - Set of inodes
  - Data/directory blocks

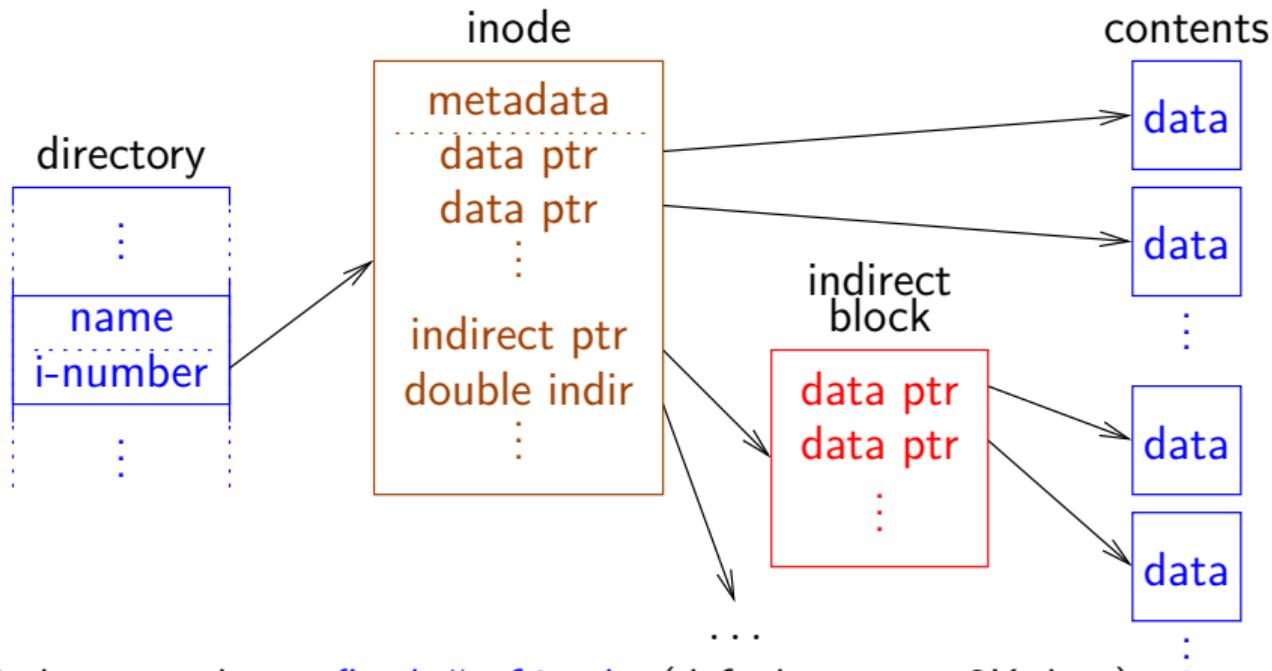
# Superblock

- The **superblock** contains *file system's parameters*.
  - Disk characteristics, block size, cylinder group (CG) info
  - Information necessary to locate an inode given its i-number.
- The superblock was *replicated* once per cylinder group.
  - At shifting offsets, so as to span multiple platters.
  - Contains magic number 0x011954 to find replicas if 1st superblock dies (Kirk McKusick's birthday?)
- Also contains *non-replicated* “summary information”
  - # blocks, fragments, inodes, directories in the entire FS
  - Flag stating if FS was cleanly unmounted.

# Bookkeeping information

- *Block bit-map*
  - Bit map of available fragments
  - Used for allocating new blocks/fragments
- *Summary info* within each cylinder group (CG)
  - # free inodes, blocks, fragments, files, directories.
  - Used when picking cylinder group from which to allocate.
- # free blocks by rotational position (8 positions)
  - Was reasonable in 1980s when disks weren't commonly zoned
  - **Zoned** means more sectors in the outside cylinders where the diameter is larger.
  - Back then OS could do stuff to minimize rotational delay.

# Inodes and data blocks



- Each cylinder group has a *fixed # of inodes* (default one per 2K data).
- The inode maps a file offset → a data block for its file.
- An inode also contains metadata for its file such as: permissions, access/modification/change times, link count.

# Inode allocation

How is a new inode allocated?

- Each file (including directories) created requires a new inode.
- If it is a *new file*, put inode in *same cylinder group* (CG) as the directory, if possible.
- If it is a *new directory*, use a *different CG* from the parent.
  - Consider CGs with greater than average # free inodes.
  - Choose CG with smallest # directories.
- *Within CG*, inodes *allocated randomly* (next free).
  - Would like related inodes as close as possible.
  - OK, because one CG doesn't have that many inodes.
  - All inodes in CG can be read and cached with small # of reads.

# Fragment allocation

When is a new fragment allocated?

- Occurs when system must allocate space when user writes beyond end of file.
- Allow *last block to be a fragment ( $\frac{1}{2}$  or  $\frac{1}{4}$  of a block) to reduce internal fragmentation.*
  - If already a fragment, may contain space for write – done.
  - Else, must deallocate any existing fragment, allocate new.
- *If no appropriate-sized free fragments available, break a full block.*
- Problem: Slow for many small writes.
  - May have to keep moving end of file around.
- (Partial) solution: create a new `stat` struct field `st_blksize`
  - Tells applications file system block size.
  - The stdio library can buffer this much data.

# Block allocation

- Goal: try to *optimize for sequential access*.
  - If available, use rotationally close block in same cylinder (obsolete)
  - Otherwise, *use block in same CG*.
  - If CG totally full, find other CG with quadratic hashing  
i.e., if CG  $n$  is full, try  $n + 1^2, n + 2^2, n + 3^2, \dots \pmod{\#CGs}$
  - Otherwise, search all CGs for some free space.
- Problem: Don't want one file filling up whole CG
  - Otherwise other inodes will not have room for their data, and it will be placed far away.
- Solution: *Break large files over many CGs*.  
But have large extents in each CGs, so sequential access doesn't require many seeks.

**Q:** How big should extents be?

**A:** Extent transfer time should be much greater than seek time

# Directories

- Directories are a type of file except a field in the inode identifies them as directories.
- *Directory contents are broken into 512-byte chunks.*
- Each chunk has **direct** structure(s) with:
  - 32-bit i-number
  - 16-bit size of directory entry
  - 8-bit file type (added later) (e.g. directory)
  - 8-bit length of file name
- Coalesce (add remaining entries to another chunk) when deleting
  - If first **direct** in chunk deleted, set i-number = 0 (i.e. not used)
- *Periodically compact directory chunks.*
  - But can never store directory entry across two different chunks.
  - Recall that only 512-byte sector writes are guaranteed to be atomic during a power failure.

## Updating FFS for the 90s

- No longer wanted to assume rotational delay.
  - With larger disk caches, *want data to be contiguously allocated.*
- Solution: Cluster writes
  - FS delays writing a block back to get more blocks.
    - *Accumulates blocks into 64K clusters, written all at once.*
- Allocation of clusters similar to fragments and blocks.
  - The system maintains summary info.
  - System maintains a cluster bit-map, i.e. has one bit for each 64K if entire cluster is free.
- Also read in 64K chunks when performing read ahead.

## Crash Recovery: Fixing corruption – fsck

- Must run FS check (**fsck**) utility after a crash.
- The file system's *summary info is usually inaccurate right after a crash.*
  - Scan the file system to check free block map, block/inode counts.
- System may have corrupt *inodes*, i.e. it was not a simple crash.
  - Bad block numbers, cross-allocation (block used in more than one place), etc.
  - Do sanity check. Clear inodes with garbage
- Fields in inodes may be wrong.
  - Count number of directory entries to verify link count, if no entries but count  $\neq 0$ , move to **lost+found**
  - Make sure size and used data counts match blocks.
- *Directories* may be bad.
  - Holes are not allowed, i.e. `.` and `..` must be valid, and file names must be unique.
  - All directories must be reachable.

## Crash recovery permeates FS code

- When designing the system *ensure that fsck can recover it.*
- Example: *If all system data is written asynchronously, then any subset of data structures may have been updated before a crash.*
- Scenario 1: delete one file, then append to other file, then *crash*.
  - New file may reuse block from old file.
  - Old inode may not be updated.
  - Therefore cross-allocation: i.e. block marked as used in both files.
  - Often inode with older mtime (last modified time) is wrong, but can't be sure.
- Scenario 2: append to one file, allocate an indirect block, then *crash*.
  - Inode points to indirect block.
  - But indirect block may contain garbage!

# Crash Recovery Idea: Ordering of updates

- *The OS must always update the file system components in a consistent order.*
  - Write new inode to disk *before* directory entry.
  - Remove directory name *before* deallocating inode.
  - Write cleared inode to disk *before* updating CG free map.
- Solution: *Make metadata updates synchronous.*
  - Doing one write at a time ensures consistent ordering.
  - Of course, this hurts performance.
  - E.g., untar much slower than disk bandwidth.
- Note: *The system cannot update buffers on the disk queue.* i.e. blocks that are waiting to be written to secondary storage.
  - E.g., say you want to make two updates to same directory block.
  - But crash recovery requires first to be synchronous.
  - Must wait for first write to complete before doing second write.

# Crash Recovery: Performance vs. consistency

- FFS *crash recoverability comes at huge cost.*
  - Makes some tasks such as untar easily 10-20 times slower.
  - All because you *might lose power* or reboot at any time.
- Even while slowing ordinary usage, recovery is slow and gets much slower with larger disks.
- One solution: battery-backed RAM
  - Expensive (requires specialized hardware).
  - Often don't learn battery has died until it is too late.
  - A pain if computer dies (can't just move disk).
  - If OS bug causes crash, RAM might be garbage.
- Another solution: UPS (uninterruptible power supply): \$100+ depending on capacity.
- Better solution: Advanced file system techniques ...

## Soft Updates: First attempt - Ordered updates

- Goal: *Want to avoid crashing after “bad” subset of writes.*
- Must follow 3 rules in ordering updates [Ganger]:
  1. Never write pointer before initializing the structure it points to.
  2. Never reuse a resource before nullifying all pointers to it.
  3. Never clear last pointer to live resource before setting new one.
- If you follow these steps, the file system will be recoverable.
- Moreover, it can recover quickly.
  - Might leak free disk space, but otherwise correct.
  - So start running after reboot, scavenge for space in background.
- How to achieve?
  - Keep a list of what has happened on buffered blocks.

## Soft Updates: First attempt - Ordered updates (continued)

- Example: Create file **A**
  - Initially: Block **X** contains an inode.
  - Initially: Block **Y** contains a directory block.
  - Task: Create file **A** in inode block **X** and the directory in block **Y**.
- We say **Y → X**, pronounced “ **Y depends on X**.”
  - which means **Y cannot be written before X** is written.
  - **X** is called the **dependee**, **Y** the **depender**.
- *Can delay both writes, so long as the order preserved.*
  - Say you create a second file **B** in blocks **X** and **Y**.
  - You only have to write each out once for both file creations.

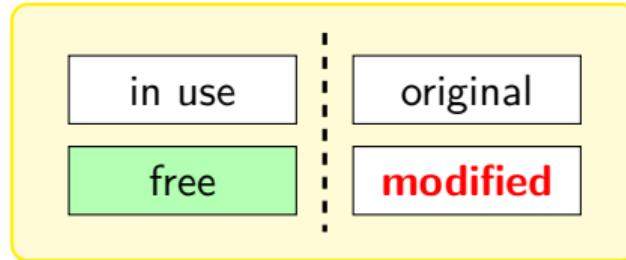
## Problem with First Attempt: Cyclic dependencies

- Suppose you create file **A** and unlink file **B**.
  - Both files are in the same directory block and inode block.
- *Can't write directory until A's inode initialized* by 1, initialize struct then pointer.
  - Otherwise, after crash directory will point to uninitialized inode.
  - Worse yet, same inode # might be re-allocated elsewhere.
  - So could end up with file name **A** being an unrelated file.
- *Can't write inode block until B's directory entry cleared* by 2, nullify pointer before reuse.
  - Otherwise, **B** could end up with too small a link count
  - File could be deleted while links to it still exist.
- Otherwise, **fsck** has to be slow.
  - It must check every directory entry and inode link count.

# Cyclic dependencies illustrated

inode block
inode #4
inode #5
inode #6
inode #7

directory block
$\langle -, \#0 \rangle$
$\langle B, \#5 \rangle$
$\langle C, \#7 \rangle$



Original organization

inode block	directory block
inode #4	$\langle A, \#4 \rangle$
inode #5	$\langle B, \#5 \rangle$
inode #6	$\langle C, \#7 \rangle$
inode #7	

Create file A

inode block	directory block
inode #4	$\langle A, \#4 \rangle$
inode #5	$\langle -, \#5 \rangle$
inode #6	$\langle C, \#7 \rangle$
inode #7	

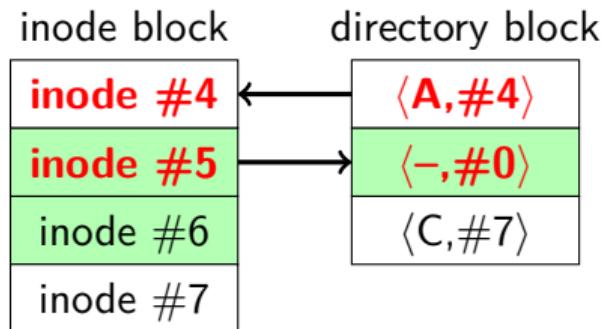
Remove file B

## More problems

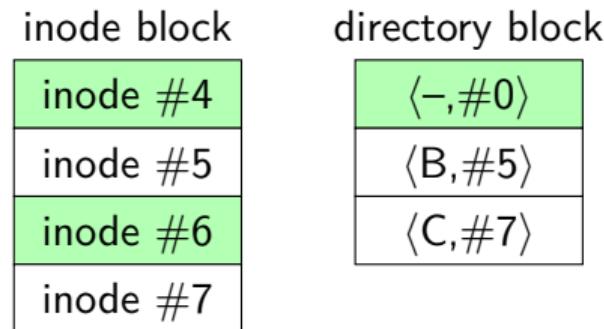
- Crash might occur between *ordered but related writes*.
  - E.g., summary information wrong after block freed.
- *Block aging*
  - Block that always has dependency will never get written back.
- Solution: **Soft updates** [Ganger]
  - Write blocks in any order.
  - But *keep track of dependencies*
  - When writing a block, *temporarily roll back any changes you can't yet commit to disk*.
  - I.e., you cannot write a block with any arrows pointing to dependees
    - ... but you can temporarily undo whatever change requires the arrow.

# Breaking dependencies with rollback

## Buffer cache

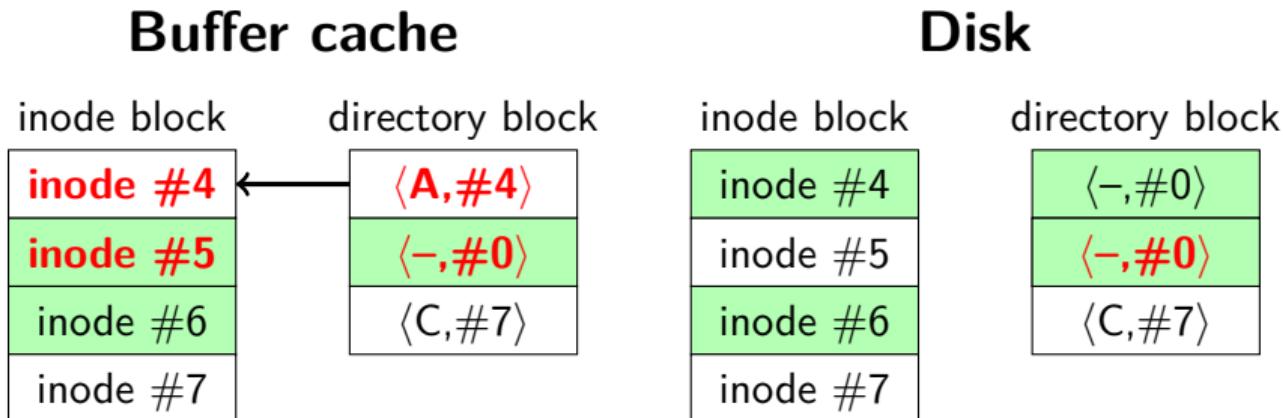


## Disk



- Created file A and deleted file B.
- Now say we decide to write directory block ...
- Can't write file name A to disk—has dependee (inode #4).

# Breaking dependencies with rollback



- Undo file A (not shown in buffer cache) before writing dir block to disk.
  - Even though we just wrote it, directory block still dirty, i.e. another write has to happen.
- But now inode block has no dependees.
  - Can safely write inode block to disk as-is ...

# Breaking dependencies with rollback

## Buffer cache

inode block
inode #4
inode #5
inode #6
inode #7

directory block
$\langle A, \#4 \rangle$
$\langle -, \#0 \rangle$
$\langle C, \#7 \rangle$

## Disk

inode block
inode #4
inode #5
inode #6
inode #7

directory block
$\langle -, \#0 \rangle$
$\langle -, \#0 \rangle$
$\langle C, \#7 \rangle$

- Now inode block clean (i.e. same in buffer cache as on disk).
- But have to write the directory block a second time ...

# Breaking dependencies with rollback

## Buffer cache

inode block
inode #4
inode #5
inode #6
inode #7

directory block
$\langle A, \#4 \rangle$
$\langle -, \#0 \rangle$
$\langle C, \#7 \rangle$

## Disk

inode block
inode #4
inode #5
inode #6
inode #7

directory block
$\langle A, \#4 \rangle$
$\langle -, \#0 \rangle$
$\langle C, \#7 \rangle$

- Now all updates are stably stored on disk.
- Crash at any point would have been safe.

# Soft updates

- The structure for each updated field or pointer contains:
  - old value
  - new value
  - *list of updates on which this update depends* (i.e. the dependees).
- The system can write the blocks in any order.
  - But must temporarily roll-back (i.e. undo) updates with pending dependencies.
  - Must lock rolled-back version so applications don't see it, i.e. applications are using the new version (e.g. that file A now exists) but it may not be recorded on disk yet.
  - The system may choose a convenient ordering, e.g. based on disk arm location.
- Some dependencies *better handled by postponing* (rather than rolling-back) in-memory updates.
  - E.g., when freeing a block (e.g., because file truncated), mark block free in bitmap (reuse resource) after the block pointer cleared on disk (nullify pointer).

# Simple example

Example: creating a zero-length file A. (R1: must initialize before pointing to A).

- *Depender*: Directory entry for A.
- *Dependee*: bitmap – must mark inode allocated before directory entry written
- *Dependee*: Inode – must initialize inode before bitmap written.
  - i.e.: *directory entry → bitmap, inode* (or possibly *directory entry → bitmap → inode*.)
- *Old value*: empty directory entry
- *New value*: ⟨filename A, inode #⟩
- Can write directory block to disk any time.
  - But must substitute old value until inode and bitmap are updated on disk.
  - Once directory block on disk contains A, the file is fully created.
  - Crash before directory on disk, worst case might leak the inode (i.e. it is marked as used in the bit-map but directory does not point to it).

# Operations requiring soft updates (1)

1. Block allocation: *block pointer* → *free block bitmap, data block*
  - Write the data to the data block and update the free block bitmap *before* the block pointer in either the inode or indirect block.
  - Use undo/redo on block pointer (and possibly file size, last modified date, etc).
2. Block deallocation: *free block bitmap* → *block pointer*
  - Must write the cleared block pointer to disk *before* the free block bitmap.
  - Or could just update free block bitmap after pointer written to disk.
  - Or just immediately update free map if old block pointer value not on disk.
  - Say you quickly append block to file then truncate the file.
    - You will know the block pointer is not written because of the allocated dependency structure.
    - So both operations together require no disk I/O!

## Operations requiring soft updates (2)

3. Link addition: (see simple example) *directory entry* → *bitmap, inode*
  - Write inode and free inode map (if new inode) to disk *before* directory entry.
  - Use undo/redo on inode # in directory entry (ignore entries with i-number 0)
4. Link removal: *bitmap, inode* → *directory entry*
  - Must write directory entry, inode and free map (if nlinks==0)
  - Clear directory entry *before* decrementing nlinks.
  - If nlinks becomes 0, nullify i-node's block pointers then update free map.
  - If directory entry was never written, decrement immediately  
(again will know by presence of dependency structure)
- Note: Again, quick create/delete requires no disk I/O.

# Soft update issues

- `fsync` – is syscall to flush (i.e. write) file changes to disk.
  - Must also *flush directory entries, parent directories*, etc. (as in # 4 in previous slide).
- `umount` – flush all changes to disk on shutdown
  - Some buffers must be *flushed multiple times* to become clean because of dependencies.
- Deleting large directory trees frighteningly fast because you nullify directory first.
  - The `unlink` syscall returns even if inode/indirect block is not cached!
  - *Dependencies created faster than blocks written.*
  - Cap the # dependencies allocated to avoid exhausting memory.
- Useless write-backs
  - Syncer flushes dirty buffers to disk every 30 seconds because
  - writing all at once means many dependencies unsatisfied.
  - Fix syncer to write blocks one at a time.
  - Fix LRU buffer eviction to know about dependencies.

## Soft updates fsck

- Split `fsck` into foreground and background parts.
- *Foreground must be done before remounting FS*
  - Need to make sure per-cylinder summary info makes sense.
  - Recompute free block/inode counts from bitmaps – very fast.
  - Will leave FS consistent, but might leak disk space.
- *Background does traditional `fsck`* operations.
  - Do after mounting to recuperate free space.
  - Can be using the file system while this is happening.
  - Must be done in foreground after a media failure.
- Difference from traditional FFS fsck:
  - May have many, many inodes with non-zero link counts.
  - Don't stick them all in `lost+found` (unless media failure).

## An alternative: Journaling

- The biggest crash-recovery challenge is inconsistency.
  - Reason: one logical operation (e.g., creating or deleting a file) requires multiple separate disk writes.
  - If only some of them happen, could end up with big problems.
- *Most of these problematic writes are to metadata.*
- An error in writing file data will ruin one file, but an error in metadata can ruin many files.
- Idea: Use a **write-ahead** log to **journal** metadata.
  - Reserve a portion of disk for a log.
  - Write any metadata operation first to the log, then make the change on the disk.
  - After crash/reboot, re-play the log (efficient).
  - May re-do already committed change, but won't miss anything.

## Journaling (continued)

- *Group multiple operations into one log entry.*
  - E.g., clear directory entry, clear inode, update free map—either all three will happen after recovery, or none will.
- Performance advantage:
  - The log is a consecutive portion of disk.
  - Multiple operations can be logged at disk bandwidth.
  - It is safe to consider updates committed when written to the log.
- Example: delete directory tree
  - Record all freed blocks and changed directory entries in log.
  - Return control to user.
  - Write out changed directories, bitmaps, etc. in background  
(sort for good disk arm scheduling)

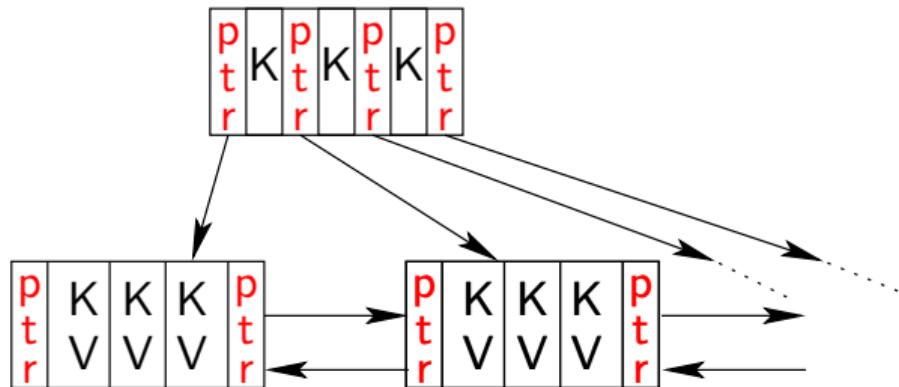
## Journaling details

- Challenge: Must *find oldest relevant log entry*.
  - Otherwise, redundant and slow to replay whole log
- Idea: *Use checkpoints*.
  - Once all records up to log entry  $N$  have been processed and affected blocks stably committed to disk...
  - record  $N$  to disk either in reserved checkpoint location, or in checkpoint log record.
  - Never need to go back before most recent checkpointed  $N$ .
- Must also find the end of the log.
  - It is typically implemented as a circular buffer; don't play old records out of order.
  - Can include begin transaction/end transaction records.
- Also typically have checksum in case some sectors are bad.

# Case study: XFS [Sweeney]

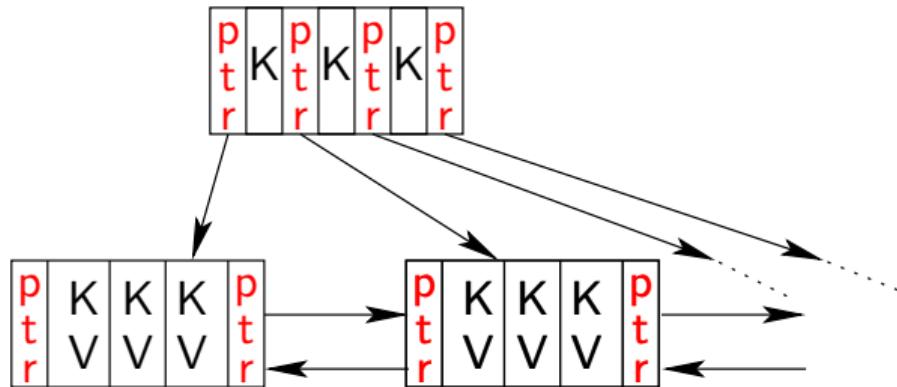
- Main idea: Think big
  - Big disks, big files, large # of files, 64-bit everything, yet maintain very good performance.
- Break disk up into **Allocation Groups** (AGs)
  - 0.5 – 4 GB regions of disk.
  - New directories go in new AGs.
  - Within directory, inodes of files go in same AG.
  - Unlike cylinder groups, AGs *too large to minimize seek times*.
  - Unlike cylinder groups, *no fixed # of inodes per AG*.
- Advantages of AGs:
  - Parallelize allocation of blocks/inodes on multiprocessor (independent locking of different free space structures)
  - Can use 32-bit block pointers within AGs which makes data structures smaller.

# B+ trees



- XFS makes extensive use of B+ trees. They are similar to B trees in the following way.
  - Both are indexed data structures that store ordered  $\langle \text{key, value} \rangle$  pairs.
  - I.e. the keys must have an ordering defined on them.
  - Both stored data in blocks for efficient disk access.
- Unlike B trees, in B+ trees *all the key-value pairs are in the leaves*.
- Therefore more keys can fit in the interior nodes, so B+ trees are shorter and wider.
- B+ trees also allow for duplicate keys.

# B+ trees



- Leaves are also connected as a linked list.
- Therefore, sequential access is easy.
- For a B+ tree with  $n$  items, all operations are  $O(\log n)$ :
  - Retrieve closest  $\langle \text{key}, \text{value} \rangle$  to target key  $k$ .
  - Insert a new  $\langle \text{key}, \text{value} \rangle$  pair.
  - Delete  $\langle \text{key}, \text{value} \rangle$  pair.

## B+ trees continued

- See any algorithms book for details (e.g., [Cormen])
- Some operations on a B+ tree are complex:
  - E.g., insert item into completely full B+ tree
  - May require “splitting” nodes and adding a new level to the tree.
  - It would be bad to *crash and leave B+tree in an inconsistent state.*
- *Journal enables complex operations to be atomic.*
  - First write all changes to the log.
  - If crash while writing log, incomplete log record will be discarded, and no change made.
  - Otherwise, if crash while updating B+ tree, replay entire log record and write everything.

# B+ trees in XFS

- B+ trees are complex to implement.
  - But once you've done it, might as well use it everywhere.
- Use B+ trees for *directories* (keyed on filename hash).
  - Makes large directories efficient.
- Use B+ trees for *inodes to block pointers*.
  - No more FFS-style fixed block pointers.
  - Instead, B+ tree maps: file offset → ⟨start block, # blocks⟩.
  - Ideally file is one or a small number of *contiguous extents*.
  - This feature allows small inodes and no indirect blocks even for huge files.
- Use B+ trees for *i-number to inode*.
  - High bits of i-number specify allocation group (AG).
  - B+ tree in AG maps: starting i-number → ⟨block #, free-map⟩
  - So free inodes tracked right in leaf of the B+ tree.

## More B+ trees in XFS

- *Free extents tracked by two B+ trees.*
  1. start block # → # free blocks
  2. # free blocks → start block #
- Use journal to update both atomically and consistently.
- #1 allows you to *coalesce* adjacent free regions.
- #1 allows you to *allocate near* some target.
  - E.g., when extending file, put next block near previous one.
  - When first writing to file, put data blocks near inode.
- #2 allows you to implement *best fit allocation*.
  - Leave large free extents for large files.

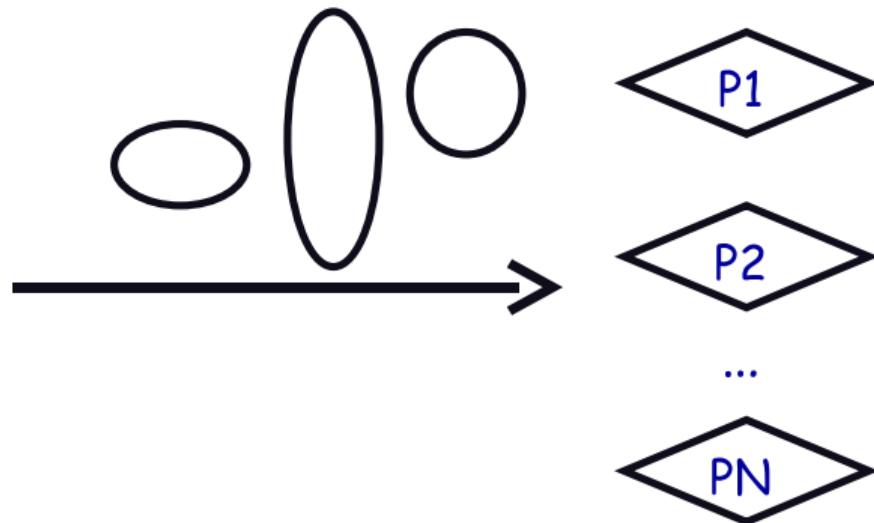
# Contiguous allocation

- Goal: *want each file contiguous on disk.*
  - I.e. Sequential file I/O should be as fast as sequential disk I/O.
- But how do you know how large a file will be?
- Idea: use **delayed allocation**, a.k.a. allocation-on-flush.
  - The `write` syscall only affects the buffer cache.
  - Allow write into buffers before deciding where to place it on disk.
  - *Assign disk space only when the buffers are flushed.*
- Other advantages:
  - Short-lived files never need disk space allocated
  - `mmaped` files are often written in random order in memory, but will be written to disk mostly contiguously.
  - Write clustering: find other nearby stuff to write to disk.

# Journaling vs. soft updates

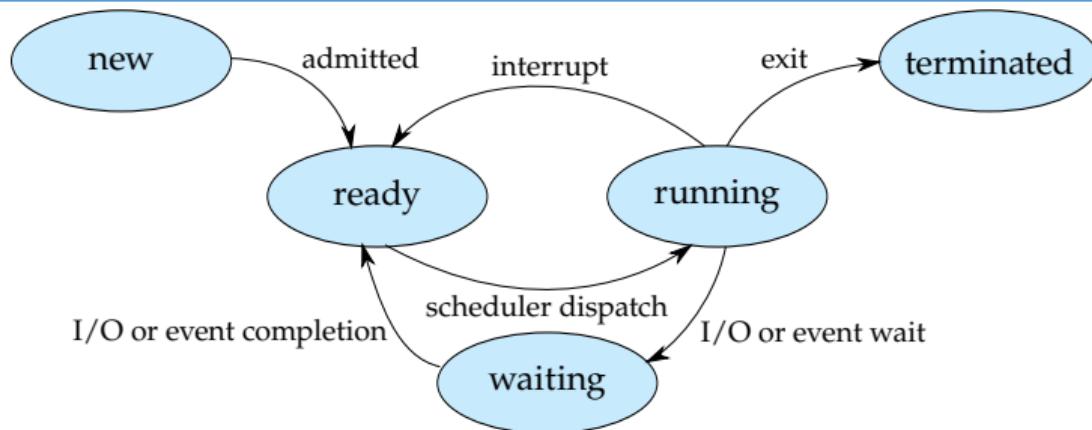
- Both journalling and soft updates are much better than FFS alone.
- Some *limitations of soft updates*:
  - Very specific to FFS data structures:  
E.g., couldn't easily add B+ trees like XFS—even directory rename difficult.
  - Metadata updates may proceed out of order.  
E.g., create A, create B, crash—maybe only B exists after reboot).
  - Still need slow background `fsck` to reclaim space.
- Some *limitations of journalling*:
  - Disk write required for every metadata operation (whereas create-then-delete might require no I/O with soft updates)
  - Possible contention for end of log on multi-processor.
  - `fsync` must sync other operations' metadata to log, too

# CPU Scheduling



- The scheduling problem:
  - Have  $K$  jobs ready to run.
  - Have  $N \geq 1$  cores.
  - Key Task: *Decide which jobs to assign to which core.*
- First: When do we make the decision?

# CPU Scheduling



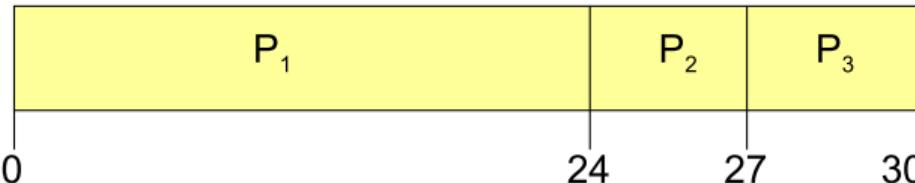
- Scheduling decisions may take place when a process:
  - switches from running to waiting state,
  - switches from running to ready state,
  - switches from new/waiting to ready,
  - exits.
- Non-preemptive schedules use 1 and 4 only.
- Preemptive schedulers run at all four transitions.

# Scheduling criteria

- Why do we care?
  - What goals should we have for a scheduling algorithm?
- **Throughput** – the number 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  
(e.g., key press to character 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 processes *in order that they arrive.*
  - Called **First-come first-served** (FCFS).
  - E.g., Say  $P_1$  needs 24 sec, while  $P_2$  and  $P_3$  each need 3.
  - Say  $P_2$ ,  $P_3$  arrived immediately after  $P_1$ , get:

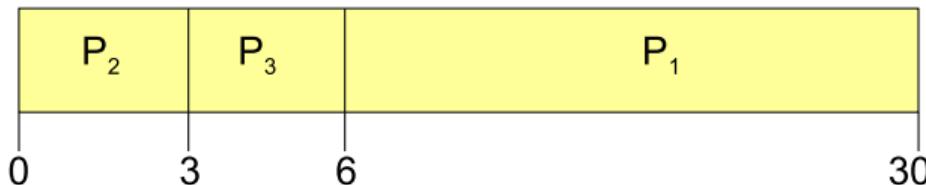


- Easy to implement—but how good is it?
- Throughput:  $3 \text{ jobs} / 30 \text{ sec} = 0.1 \text{ jobs/sec}$
- Turnaround time:  $P_1: 24, P_2: 27, P_3: 30$ 
  - Average turn around time:  $(24+27+30)/3=27$
- Can we do better?

## FCFS continued

- Suppose we scheduled  $P_2$ ,  $P_3$ , then  $P_1$

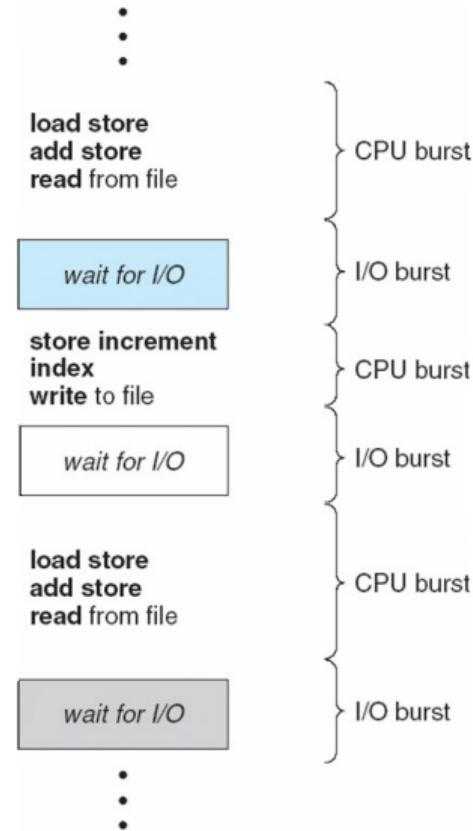
- Would get:



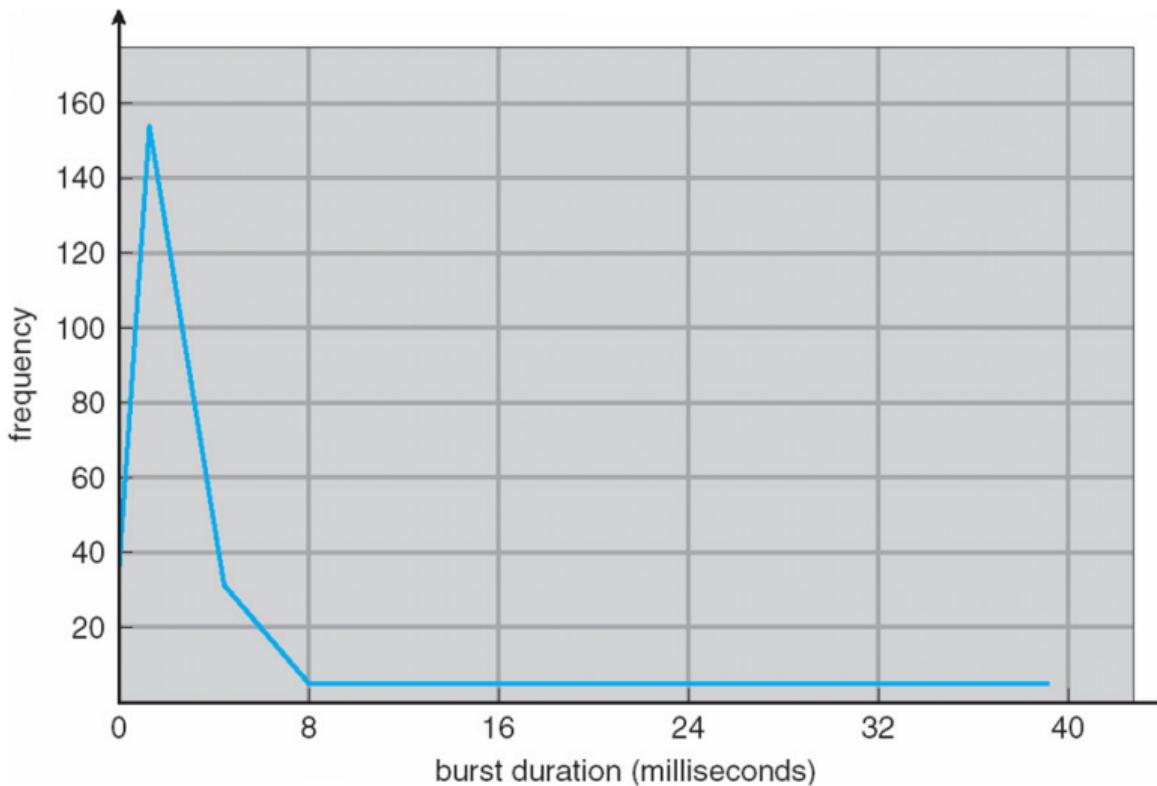
- Throughput:  $3 \text{ jobs} / 30 \text{ sec} = 0.1 \text{ jobs/sec}$
- Turnaround time:  $P_1: 30, P_2: 3, P_3: 6$ 
  - Average turnaround time:  $(30+3+6)/3=13$  – much less than 27.
- Lesson: scheduling algorithm can reduce turnaround time.
  - Minimizing waiting time can improve response time and turnaround time.*
- What about throughput? i.e. how many complete per unit time?

# Bursts of computation and I/O

- Observation: jobs contain I/O and computation: e.g.
  - bursts of computation
  - followed by waiting for I/O.
- To *maximize throughput* the system
  - must *maximize CPU utilization*
  - and *maximize I/O device utilization*.
- How to do it?
  - Idea: overlap I/O and computation from multiple jobs.
  - This means *response time is very important for I/O-intensive jobs, e.g. user interface.*
  - I.e. I/O devices will be idle until job gets small amount of CPU to issue the next I/O request.



## Histogram of CPU-burst times



- *A lot of CPU-burst times are short.* What does this mean for FCFS?

## FCFS Convoy effect

- **FCFS Convoy effect:** a long running job prevents many shorter jobs that are after it from running.
- Without preemption, CPU-bound jobs will hold the CPU until they exit or require I/O, but I/O is rare for CPU-bound threads. The result is *long periods where no I/O requests are issued*, and the CPU is held.
- The result: *poor I/O device utilization* because the I/O bound jobs wait for the CPU.
- Example: one CPU-bound job, many I/O bound
  - CPU-bound job runs for a long time, then blocks. I/O devices idle the whole time.
  - I/O-bound job(s) run and quickly block on I/O.
  - CPU-bound job runs for a long time then blocks again.
  - I/O completes.
  - CPU-bound job continues while I/O devices idle.
- Idea: run process whose I/O completed?
- What is a potential problem?

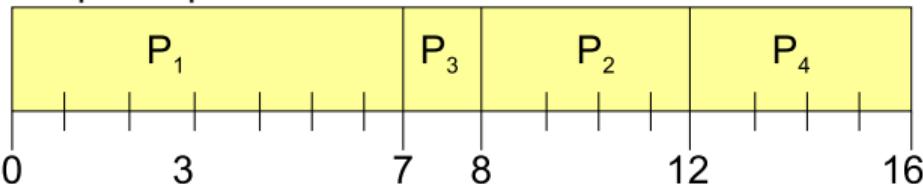
# SJF Scheduling

- **Shortest-job first** (SJF) attempts to minimize turnaround time (i.e. time for each process to finish).
  - Key Idea: *schedule the job whose next CPU burst is the shortest.*
- Two schemes:
  - *Non-preemptive* – once the CPU is given to the process it cannot be preempted until completes its CPU burst.
  - *Preemptive* – **Shortest-Remaining-Time-First** or SRTF : if a new process arrives with a CPU burst length less than remaining time of current executing process, preempt the current one and run the new one.
- What does SJF optimize?
  - It gives the *minimum average waiting time* for a given set of processes.

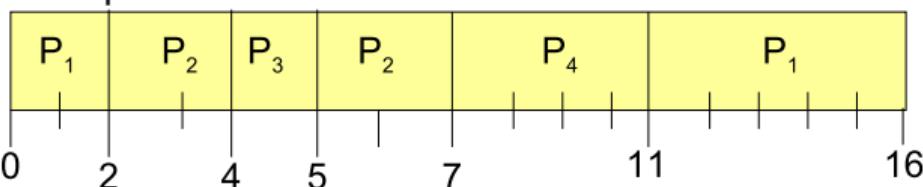
# Examples

Process	Arrival Time	Burst Time
$P_1$	0.0	7
$P_2$	2.0	4
$P_3$	4.0	1
$P_4$	5.0	4

- Non-preemptive



- Preemptive



- Drawbacks?

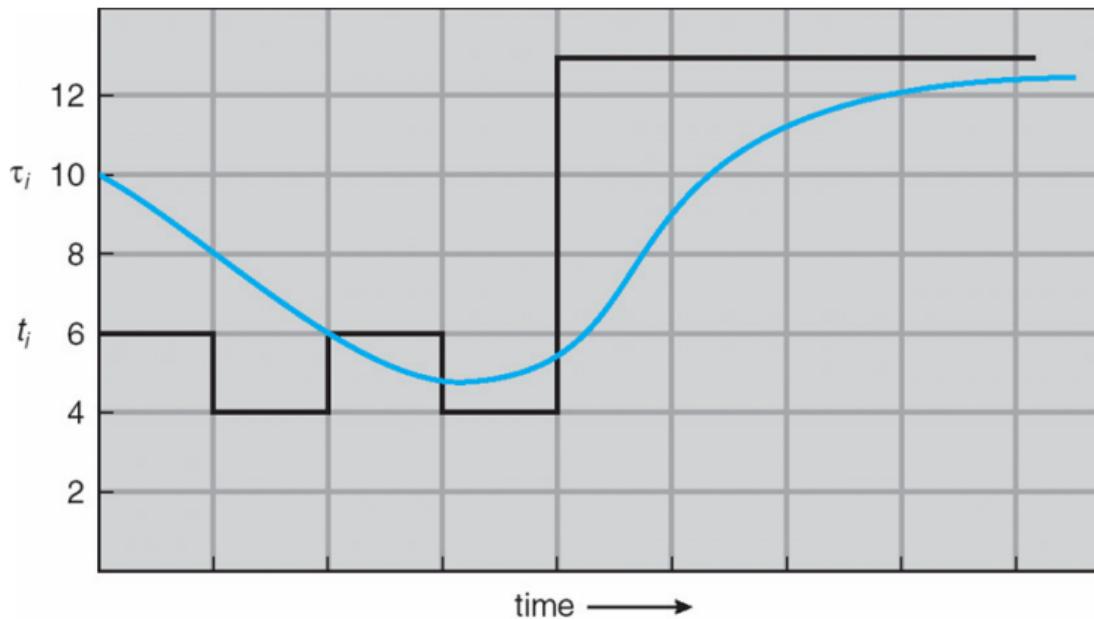
# SJF limitations

- SJF doesn't always minimize the average turnaround time.
  - It only minimizes waiting time, which in turn, minimizes the response time.
  - Example where turnaround time might be suboptimal?
  - A job that is longer overall longer but has shorter bursts.
- SJF can lead to unfairness or **starvation** of *jobs with longer bursts. i.e. they never run.*
- In practice, can't actually predict the future.
- But can estimate CPU burst length based on past bursts.
  - Idea: use exponentially weighted average.
  - $t_n$  actual length of process's  $n^{\text{th}}$  CPU burst.
  - $\tau_{n+1}$  estimated length of proc's  $n + 1^{\text{st}}$ .
  - Choose parameter  $\alpha$  where  $0 < \alpha \leq 1$ .
  - Let  $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$ .

## SJF limitations

- A large  $\alpha$  will put more weight on the actual length of the last burst  $t_n$ .
- A small  $\alpha$  will put more weight on the estimated length of the last burst  $\tau_n$ .
- If  $t_n = 4$  and  $\tau_n = 12$  then ...
  - $\alpha = 0.75$  will give an estimate of  $3 + 3 = 6$  (i.e. close to  $t_n$ ).
  - $\alpha = 0.50$  will give an estimate of  $2 + 6 = 8$  (i.e. in the middle).
  - $\alpha = 0.25$  will give an estimate of  $1 + 9 = 10$  (i.e. close to  $\tau_n$ ).
- Older values are given less weight. The progression is as follows.
  - $\tau_2 = \alpha t_1 + (1 - \alpha) \tau_1$
  - $\tau_3 = \alpha t_2 + (1 - \alpha) \alpha t_1 + (1 - \alpha)^2 \tau_1$
  - $\tau_4 = \alpha t_3 + (1 - \alpha) \alpha t_2 + (1 - \alpha)^2 \alpha t_1 + (1 - \alpha)^3 \tau_1$

## Exp. weighted average example



CPU burst ( $t_i$ )

6

4

6

4

13

13

13

...

"guess" ( $\tau_i$ )

10

8

6

6

5

9

11

12

...

# Round robin (RR) scheduling



- **Round robin** scheduling addresses fairness and starvation.
  - Preempt job after some time slice or **scheduling quantum**.
  - When preempted, *move to back of FIFO queue*.
  - Most systems do some flavor of this approach.
- Advantages:
  - Fair allocation of CPU across all jobs.
  - Low average waiting time when job lengths vary. Short jobs complete quickly.
  - Good for responsiveness if there are a small number of jobs. All jobs get started quickly.
- Disadvantages?

# RR disadvantages

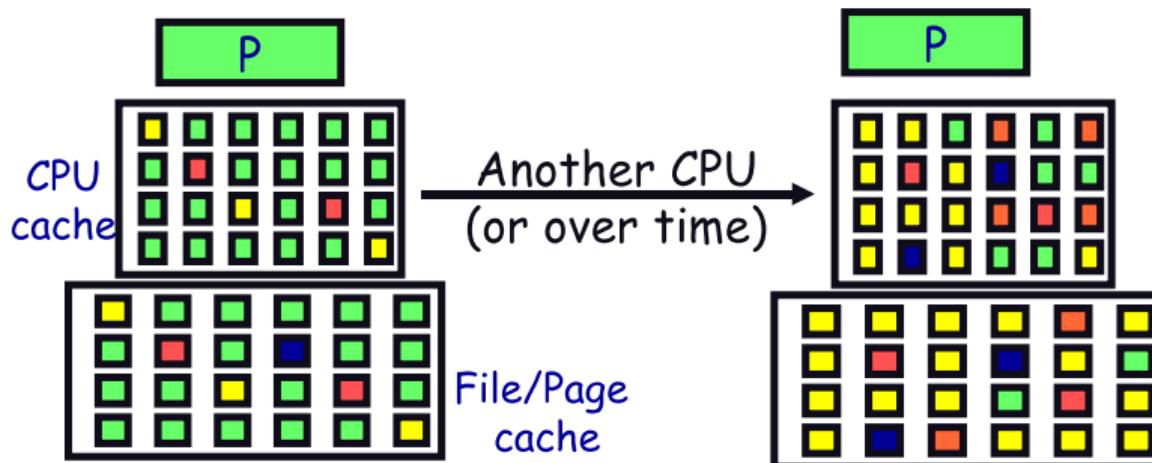
- RR is good for varying sized jobs, but what about same-sized jobs?
- Assume 2 jobs of time=100 each with a scheduling quantum of 1:



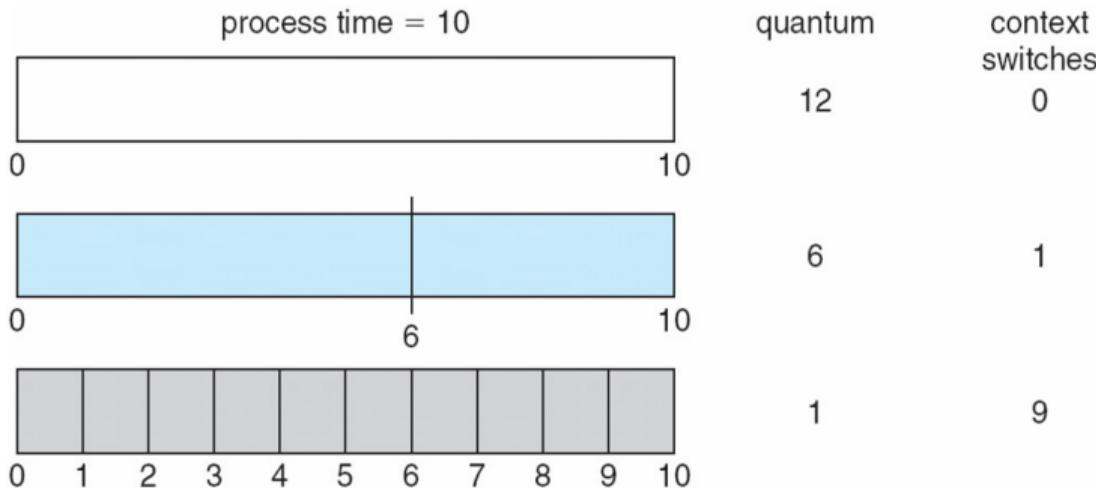
- Even if context switches were free (which is not true) ...
  - What would average completion time be with RR?  
 $(199 + 200) / 2 = 199.5$
  - How does that compare to FCFS?  
 $(100 + 200) / 2 = 150$ , i.e. about 33% longer.
- *For same sized jobs, round robin performs worse than FIFO.*

# Context switch costs

- What is the cost of a context switch?
- Direct cost: *processor time* costs in the kernel.
  - Save and restore registers, etc., for the system call.
  - Switch address spaces (e.g. use different page tables).
- Indirect costs: *more cache misses* in CPU's RAM cache, TLB and system's file buffer cache.



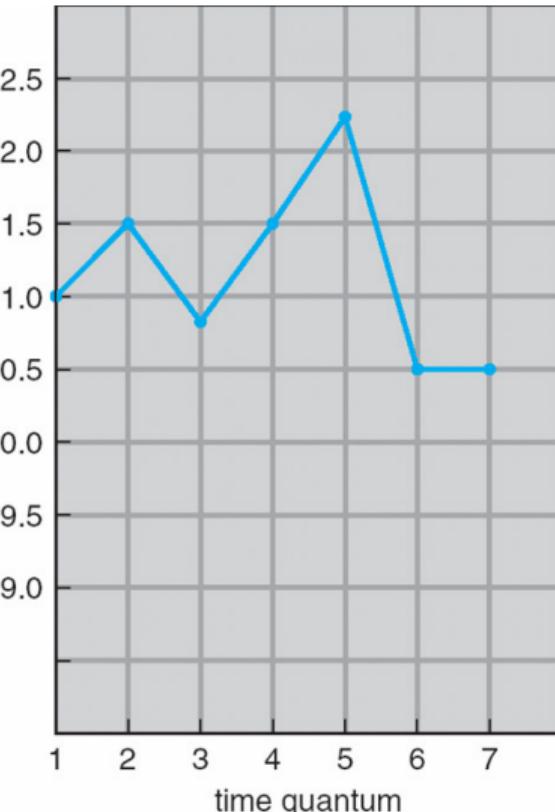
# Time quantum



- How to pick quantum?
  - Want it much larger than the context switching cost (to *keep overhead low*).
  - Want the quantum to be *larger than the majority of bursts*,
  - but not so large system reverts to FCFS
- Typical values: 10–100 msec.

## Turnaround time vs. quantum

average turnaround time



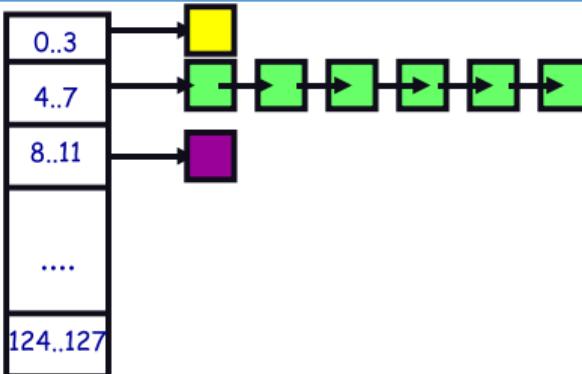
process	time
$P_1$	6
$P_2$	3
$P_3$	1
$P_4$	7

- The average turnaround time has local minima when the time quantum is the size of one of the process times, e.g. 1, 3, 6 and 7.
- Its global minimum is when the quantum size is greater than or equal to most of the process times.

# Priority scheduling

- Associate a numeric priority with each process
  - E.g., smaller number means higher priority (Unix/BSD).
  - Note that in Linux the higher number has higher priority.
- Give the CPU to the process with highest priority.
  - Can be done preemptively or non-preemptively.
- Note SJF is a 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 feedback queues (BSD)



- Every runnable process is on one of 32 run queues.
  - *Kernel runs process on highest-priority non-empty queue.*
  - **Round-robs among processes on same queue.**
  - 0..3 means values (from next slide) from 0 to 3 are mapped to the top-level queue.
- Process priorities are dynamically computed.
  - Processes **moved between queues** to reflect priority changes.
  - If a process gets higher priority than the currently running process, run it.
- Idea: *Favour interactive jobs that use less CPU* i.e. block waiting for I/O.

# Process priority

- `p_nice` – user-settable weighting factor (-20 – 20). Negative numbers give higher priority.
- `p_estcpu` – per-process estimated CPU usage.
  - Incremented whenever timer interrupt found process running, i.e. it got preempted instead of blocked waiting for I/O.
  - Decayed every second while process runnable.

$$p_{\text{estcpu}} \leftarrow \left( \frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1} \right) p_{\text{estcpu}} + p_{\text{nice}}$$

- Where `load` is the sampled average of length of the run queue plus short-term sleep queue over last minute.
- If load is 2, `p_estcpu` becomes 4/5 of its previous value each second.
- If load is 10, `p_estcpu` becomes 20/21 of its previous value each second.
- *The higher the load, the less `p_estcpu` is reduced.*

# Process priority

- From the previous slide, `p_nice` *has a noticeable effect*. For a load of 10 you would reduce it to 20/21 of its previous value but then
  - add 20 to it for the largest possible `p_nice` value or
  - subtract 20 from it for the smallest possible value.
- The run queue is determined by `p_usrpri/4` where

$$p_{\text{usrpri}} \leftarrow 50 + \left( \frac{p_{\text{estcpu}}}{4} \right) + 2 \cdot p_{\text{nice}}$$

- The value is clipped if it is over 127.
- Recall: the range of `p_nice` is from -20 to 20 in BSD Unix, so `p_usrpri/4` runs from  $2.5 + \left( \frac{p_{\text{estcpu}}}{16} \right)$  to  $22.5 + \left( \frac{p_{\text{estcpu}}}{16} \right)$ .
- And `p_estcpu` would have to equal 1,608 to be placed in the lowest queue 123..127.

## Sleeping process increases priority

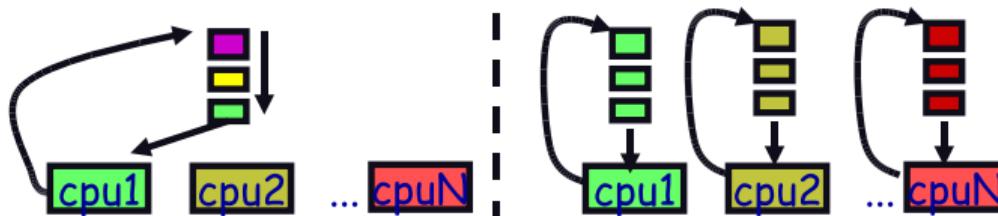
- `p_estcpu` not updated while the thread is asleep.
  - Instead `p_slptime` keeps count of sleep time.
- When the thread becomes runnable again,

$$p_{\text{estcpu}} \leftarrow \left( \frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1} \right)^{p_{\text{slptime}}} \times p_{\text{estcpu}}$$

- This formula *approximates decay ignoring nice and past loads.*
- Previous description based on *The Design and Implementation of the 4.4BSD Operating System* by McKusick

# Multiprocessor scheduling issues

- The scheduler must decide on more than which processes to run.
  - It must decide on which CPU / core runs which process.
- Moving between CPUs / cores has costs.
  - More RAM cache misses and depending on the processor architecture, more TLB misses too.
- **Affinity scheduling**—try to *keep threads on same CPU / core*.



- But also prevent load imbalances in some cases.
- System performs a *cost-benefit* analysis when deciding to migrate.

# Thread dependencies

- Say thread  $H$  has high priority and thread  $L$  has low priority.
  - $L$  acquires lock  $I$ .
  - Scenario 1:  $H$  tries to acquire  $I$ , fails, spins.  $L$  never gets to run.
  - Scenario 2:  $H$  tries to acquire  $I$ , fails, blocks.  $M$  enters system at medium priority.  $L$  never gets to run.
  - Both scenes are examples of **priority inversion**, i.e. lower priority thread blocks higher priority thread from making progress.
- Scheduling = deciding which thread should make progress.
  - Key Idea: *A thread's importance should increase with the importance of those that depend on it.*
  - Naïve priority schemes violate this principle.

# Priority donation

- Example 1 – *direct priority change*:  $L$  low,  $M$  medium,  $H$  high priority threads.
  - $L$  holds lock  $\ell$ .
  - $M$  waits on  $\ell$ , so  $L$ 's *priority is raised to*  $L_1 = \max(M, L) = 4$ .
  - Then  $H$  waits on  $\ell$ , so  $L$ 's priority raised to  $\max(H, L_1) = 8$ .
- Example 2 – *indirect priority change*: same  $L, M, H$  as above.
  - $L$  holds lock  $\ell_1$  and  $M$  holds lock  $\ell_2$ .
  - $M$  waits on  $\ell_1$ ,  $L$ 's priority now  $L_1 = 4$  (as before).
  - Then  $H$  waits on  $\ell_2$ .  $M$ 's priority goes to  $M_1 = \max(H, M) = 8$ , and  $L$ 's priority raised to  $\max(M_1, L_1) = 8$
- Example 3:  $L$  (priority 2),  $M_1, \dots, M_{1000}$  (all priority 4)
  - $L$  has  $\ell$ , and  $M_1, \dots, M_{1000}$  all block on  $\ell$ .  $L$ 's priority is  $\max(L, M_1, \dots, M_{1000}) = 4$ .

# Borrowed Virtual Time Scheduler [Duda]

- Many modern schedulers employ notion of **virtual time**.
  - Idea: *Equalize virtual CPU time consumed by different processes.*
  - Examples: Linux **Completely Fair Scheduler (CFS)**
- Idea: Run process *with the lowest effective virtual time.*
  - $A_i$  – *actual virtual time* consumed by process  $i$
  - *effective virtual time*  $E_i = A_i - (\text{warp}_i ? W_i : 0)$
- Supports real-time applications:
  - Warp factor allows borrowing against future CPU time.
  - *Allows an application to temporarily violate fairness.*

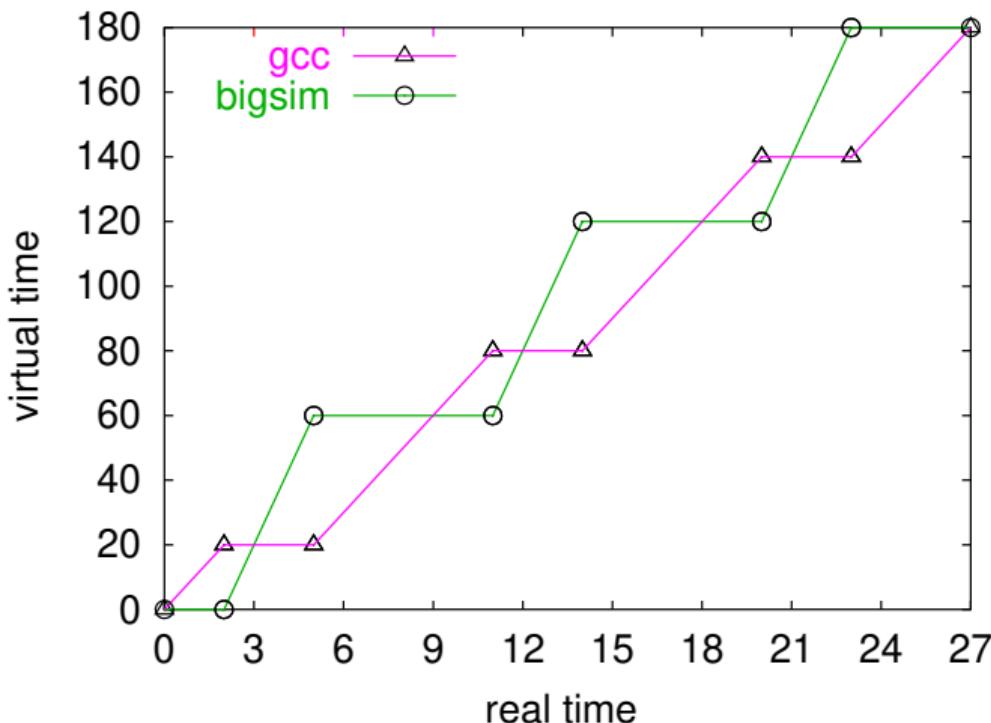
# Process weights

- Key Idea: Each process will be *scheduled based on its effective virtual time*  $E_i$ , which could be different than its *actual virtual time*  $A_i$  due to warping.
- The actual time (i.e. not virtual time) spent running on the processor is  $t$ .
- Each process  $i$ 's share of actual processor time is determined by its weight  $w_i$ .
- Process  $i$  should get  $w_i / (\sum_j w_j)$  fraction of the actual processor time.
- When  $i$  consumes  $t$  processor time, track it as:  $A_i += t/w_i$
- Example: gcc has weight  $w_1 = 2$  and bigsim  $w_2 = 1$ . Therefore  $\sum_j = 1 + 2 = 3$ .
  - gcc will get  $w_1 / (w_1 + w_2) = 2/3$  of the actual processor time and
  - bigsim will get  $w_2 / (w_1 + w_2) = 1/3$  of the time.
  - gcc's weight is twice the size of bigsim's weight, so it gets twice as much processor time.
  - Assuming no IO the schedule would be: gcc, gcc, bigsim, gcc, gcc, bigsim, ...

# Process weights

- Problem: Lots of context switches, which is not good for performance.
- Idea: add in context switch allowance,  $C$ , so each process *runs for a bit of time,  $C/w_i$ , before preemption.*
- Only switch from  $i$  to  $j$  if  $E_j \leq E_i - C/w_i$ .
- I.e.  $E_j$  must exceed  $E_i$  by at least  $C/w_i$  before there is a context switch.
- $C$  is wall-clock time so must divide by  $w_i$  to get virtual time.
- Ignore  $C$  if  $j$  has just became runnable. Why? To avoid affecting its response time.
- On the next slide, ignoring the start up,
  - both gcc and bigsim will run for 60 units of virtual time per time slice,
  - but gcc runs for 6 units of real time and bigsim runs for 3 units.

## BVT example

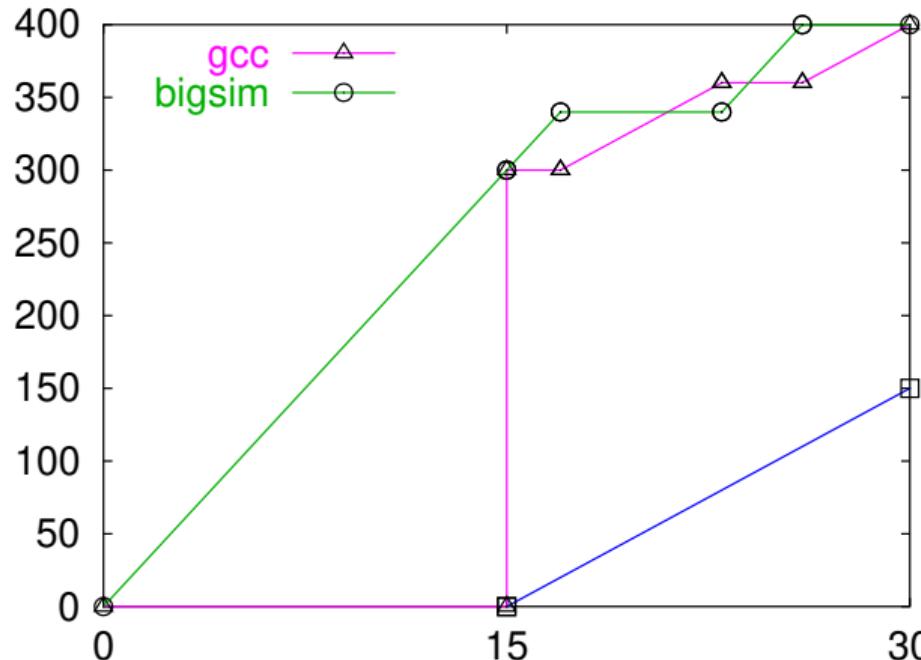


- gcc has weight 2, bigsim weight 1,  $C = 2$ , no I/O

# Sleep/wakeup

- Problem: *Must lower priority after wakeup*, i.e. increase actual virtual time  $A_i$ .
  - Why? Otherwise process with very low  $A_i$  (because it was sleeping) would starve the other processes while its  $A_i$  value catches up.
- Have a lower bound on  $A_i$  with a minimum time, **Scheduler Virtual Time** (SVT).
  - SVT is minimum actual virtual time,  $A$ , for all runnable threads.
  - When waking  $i$  from voluntary sleep, set  $A_i \leftarrow \max(A_i, SVT)$
- Note the 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 a file read.
- Note: Even with SVT,  *$A_i$  can never decrease*
  - After short sleep, might have  $A_i > SVT$ , so  $\max(A_i, SVT) = A_i$
  - This ensures that *process  $i$  never gets more than its fair share of CPU* in the long run.

# gcc wakes up after I/O

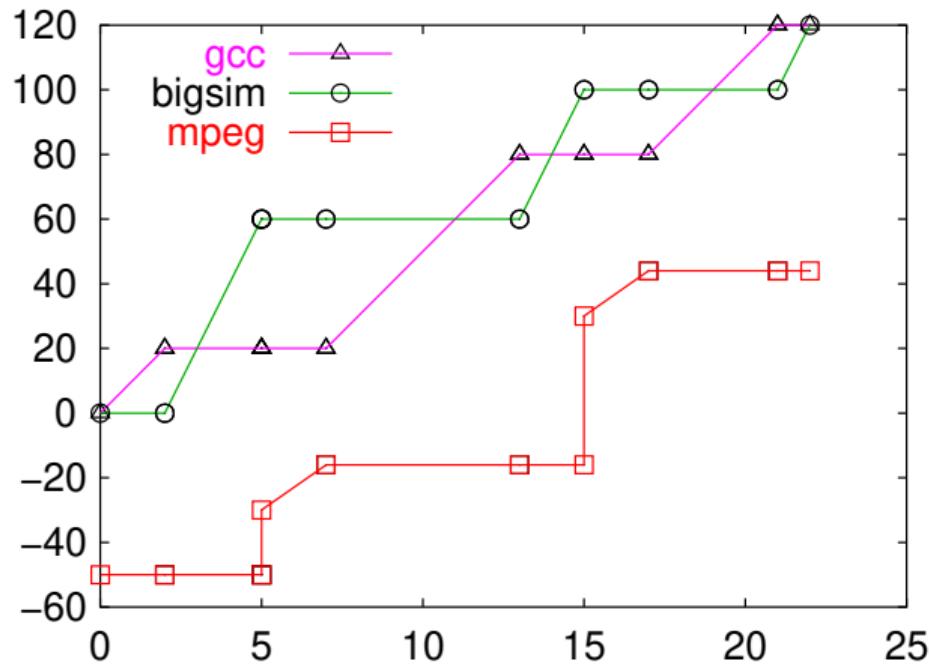


Here gcc's  $A_i$  gets set to SVT on wakeup (top blue line) otherwise, it would starve bigsim (lower blue line) by running for 15 units of time..

# Real-time threads

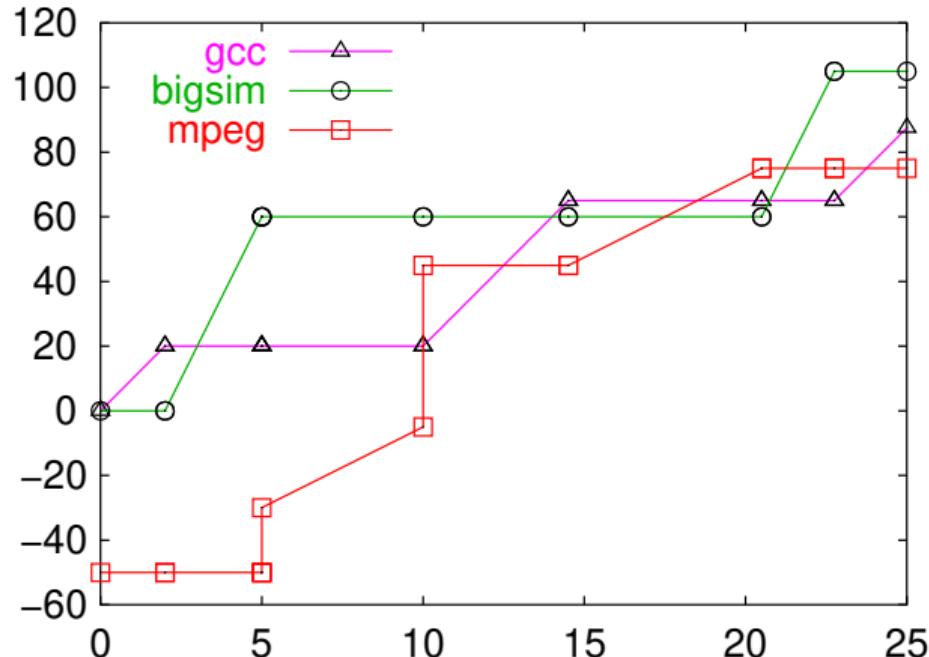
- *Goal: support soft real-time threads* (i.e. threads that need to run within a certain time or performance will degrade).
  - E.g., mpeg player must run every 10 clock ticks or the frame will freeze and skip.
- Recall  $E_i = A_i - (\text{warp}_i ? W_i : 0)$ 
  - This is where we distinguish between  $E_i$  and  $A_i$ .
  - $W_i$  is **warp factor** – i.e. it gives a thread precedence.
  - Give mpeg player  $i$  a large  $W_i$  factor and it will get the CPU whenever it is runnable.
  - But long term CPU share won't exceed  $w_i / \sum_j w_j$ .
- Note  $W_i$  only matters when  $\text{warp}_i$  is true. Need two other parameters,  $L_i$  and  $U_i$ .
  - Can set  $\text{warp}_i$  with a syscall, or have it set by the 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



Here the mpeg player runs with  $-50$  warp value and gets the processor when needed. When it wakes up at 5 and 15,  $A$  gets set to the current SVT, then  $E$  gets set to  $A-50$ , and it runs for 2 units of time. It never missing a frame.

## Warped thread hogging CPU

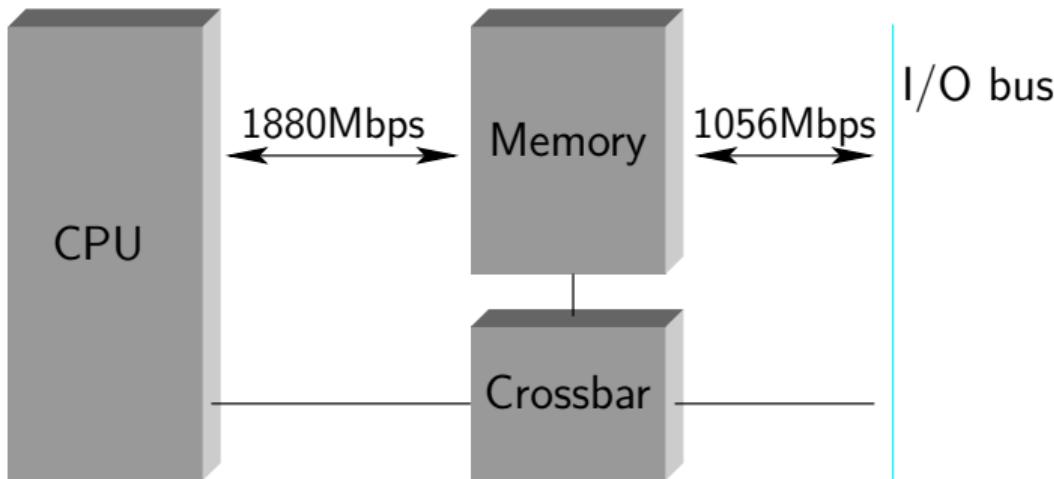


Here mpeg runs at time 5, exceeds its limit  $L_i$  at 10, and so  $\text{warp}_i \leftarrow \text{false}$ . mpeg jumps to its  $E$  without the warp value -50. gcc now has the lowest  $E$  value and so it runs next.

## Lottery Scheduler [Waldspurger]

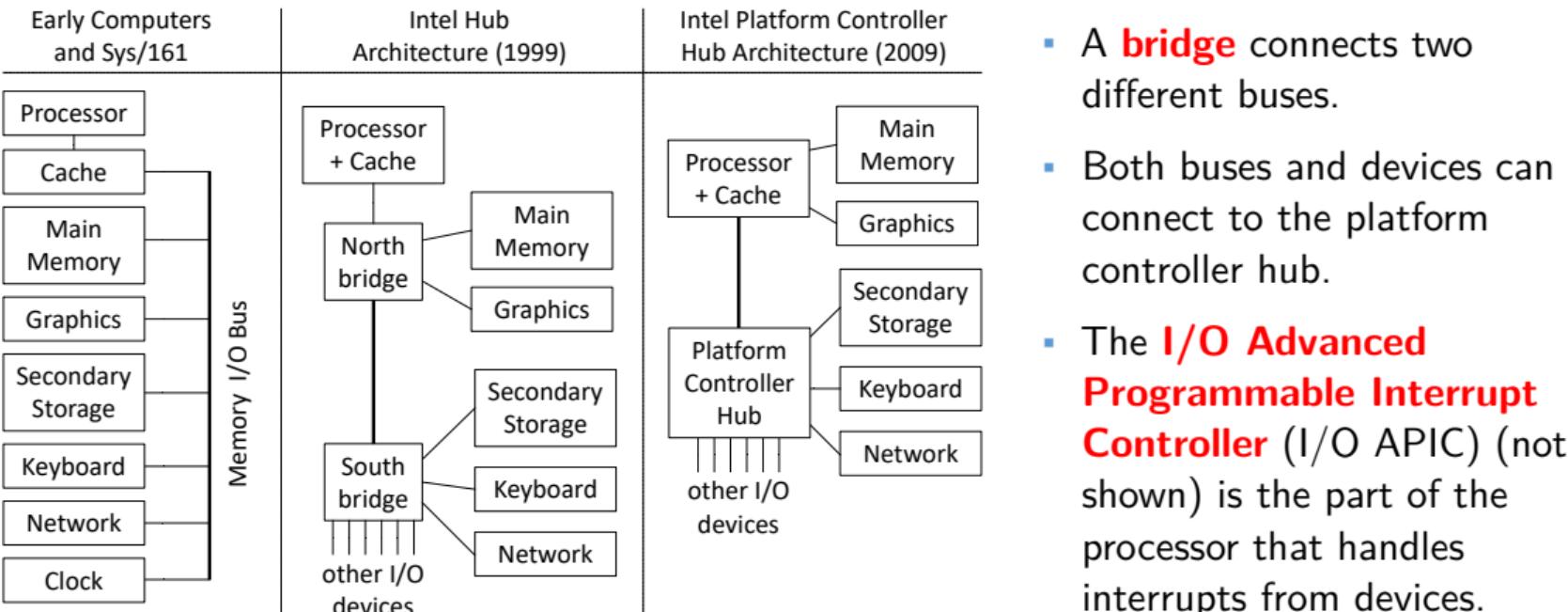
- Reading assignment.
- It is a great paper and simple algorithm.
- Randomly select a process to run!
- Process priorities are determined by a number of tickets the process has.

# Memory and I/O buses



- A CPU (and its cores) accesses **physical memory** over a **bus**, which is a *communication pathway between components* of a computer. A computer will have several buses.
- Devices access memory over I/O bus with DMA (more on DMA soon).
- Devices can appear to be a region of memory (more on memory-mapped IO soon).
- A **crossbar** can connect any input to any output.

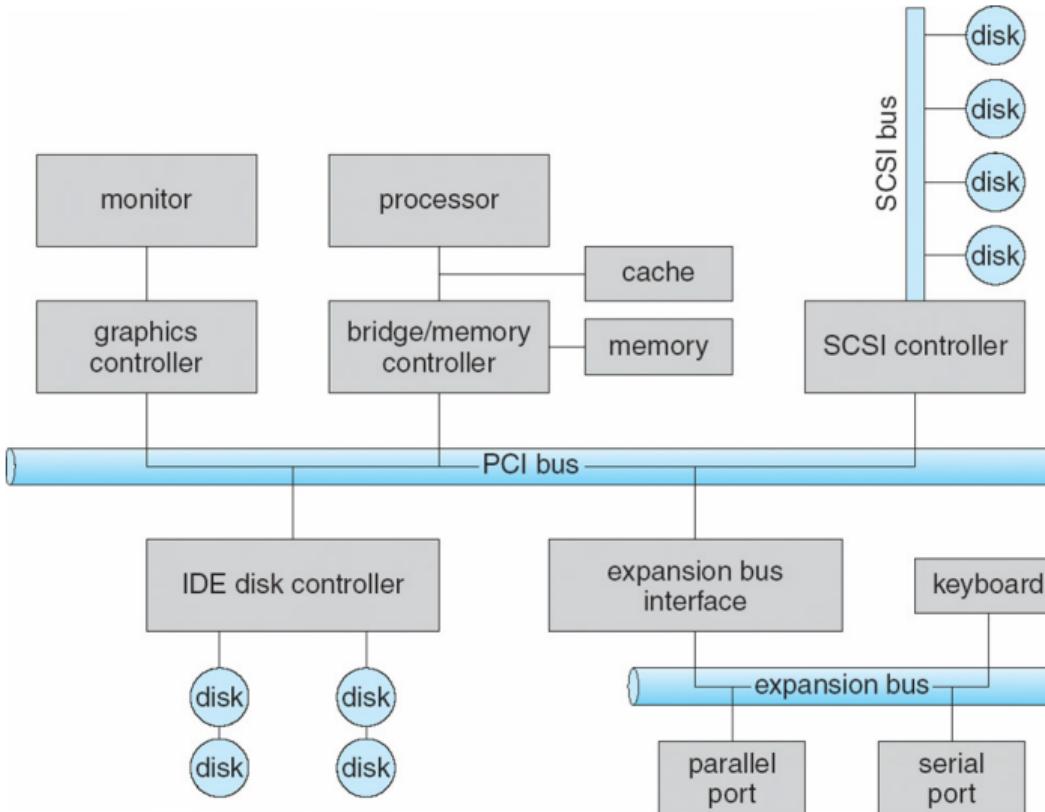
# Evolving PC architecture



# What is memory?

- **SRAM** – Static RAM (review from CS251)
  - Like two NOT gates circularly wired input-to-output (SR Latch, D Latch, D flip-flop)
  - 4–6 transistors per bit, actively holds its value (6T SRAM).
  - Very fast, used to cache slower memory, i.e. DRAM.
- **DRAM** – Dynamic RAM (review from CS251)
  - A capacitor + gate, holds charge to indicate bit value.
  - 1 transistor per bit – extremely dense storage.
  - Charge leaks—need slow comparator to decide if bit 1 or 0.
  - Must re-write charge after reading, and periodically refresh.
  - Reported as RAM with your laptop, e.g. 8GB RAM or 16 GB RAM.
- **VRAM** – “Video RAM” (new)
  - **Dual ported** i.e. can write while another component reads.

# What is I/O bus? E.g., PCI



- High speed devices connected to processor.
- PCI is slowly being replaced by PCI express (PCIe).
- *Some “devices” may be a controller for another bus, e.g. IDE disk controller or expansion bus interface.*
- The signal from a device, e.g. mouse, may travel through several buses, e.g. USB to PCI to a PCIe bus.

# Communicating with a device

- **Memory-mapped device registers.**
  - Some ranges of *physical addresses correspond to device registers*.
  - Load and store instructions gets status or sends instructions to a device, not RAM.
- **Device memory** – device may have memory and the OS can write directly to the device *through the I/O bus* rather than a RAM address. as above.
- Some processors (e.g. x86) provide **special I/O instructions**, e.g. **in** and **out**.
  - Similar to load and store instruction, but go through a separate bus for devices.
  - OS can allow user-mode access to I/O ports with finer granularity than a page.
- **Direct memory access** (DMA) – place instructions to device in main memory.
  - Typically then need to “poke” device by writing to one of its registers.
  - Device then transfers data to RAM over I/O bus and generates an interrupt when it is done.

# Example: parallel port (LPT1)

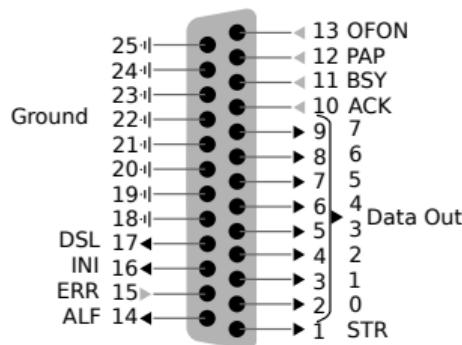
- Simple hardware has three control registers:

$D_7$	$D_6$	$D_5$	$D_4$	$D_3$	$D_2$	$D_1$	$D_0$
<i>read/write data register (port 0x378)</i>							

BSY	ACK	PAP	OFON	ERR	-	-	-
<i>read-only status register (port 0x379)</i>							

-	-	-	IRQ	DSL	INI	ALF	STR
<i>read/write control register (port 0x37a)</i>							

- Every bit except IRQ corresponds to a pin on 25-pin connector:



[Wikipedia][Messmer]

## Example: parallel port (LPT1)

- With this method, communication with devices carried out through device registers.
- There are three primary types of device registers
  - status**: tells you something about the device's current state. Typically, a status register is *read*.
  - command**: issue a command to the device by *writing* a particular value to this register.
  - data**: used to transfer data to and from the device.
- In the example on the next slide
  - 0x80 is 1000 0000 in binary so you are masking the  $\overline{\text{BSY}}$  bit from the input at 0x379.
  - 0x01 is 0000 0001 in binary so you are masking the STR bit from the input at 0x37a

## Writing bit to parallel port [osdev]

```
void
sendbyte(uint8_t byte)
{
    /* Busy wait until BSY bit is 1. */
    while ( (inb(0x379) & 0x80) == 0)
        delay ();

    /* Put the byte we wish to send on pins D7--D0. */
    outb(0x378, byte);

    /* Pulse STR (strobe) line to inform the printer
     * that a byte is available */
    uint8_t ctrlval = inb(0x37a); /* read in control register */
    outb(0x37a, ctrlval | 0x01); /* set STR bit to 1*/
    delay ();
    outb(0x37a, ctrlval);          /* reset STR bit back to 0 */
}
```

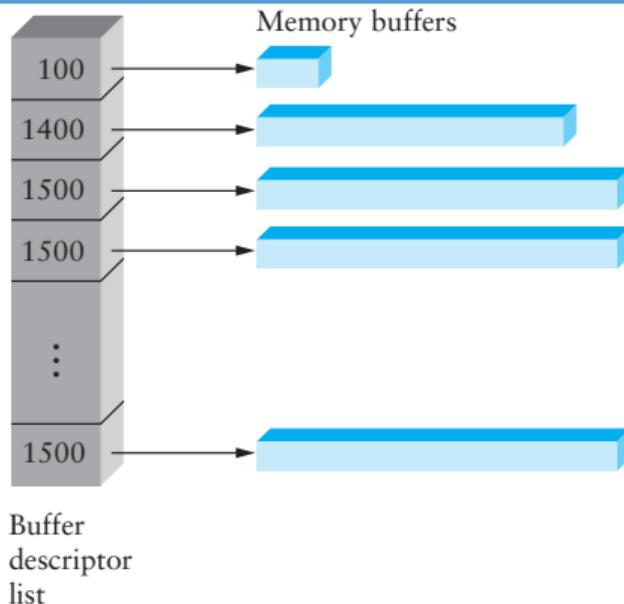
# Special IO instructions vs. Memory-mapped IO

- x86's `in` and `out` instructions are awkward to work with.
  - Instruction format restricts what registers you can use.
  - Only allows  $2^{16}$  different port numbers.
- Devices *can achieve same effect with physical addresses.*
- For this example, assume the device corresponds to address 0xc00c 0100.

```
volatile int32_t *device_control
    = (int32_t *) 0xc00c0100;
*device_control = 0x80;           /* give command */
int32_t status = *device_control; /* get back status */
```

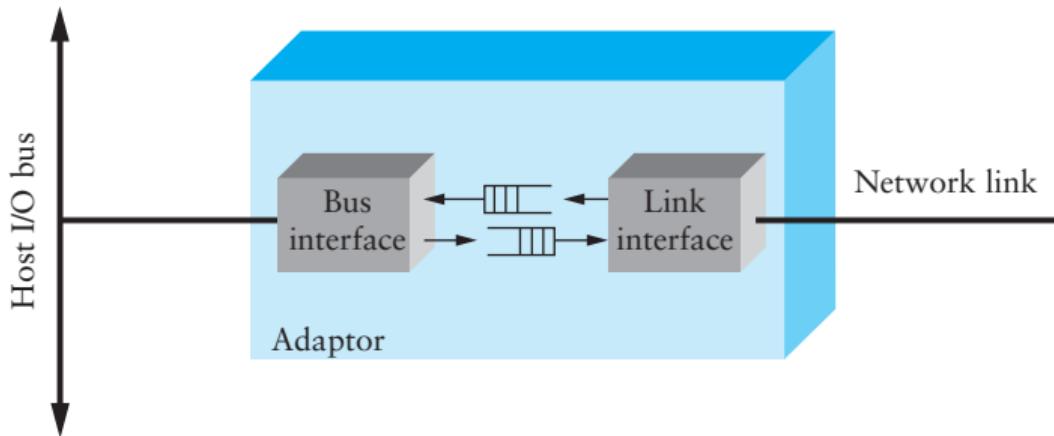
- OS must map device addresses to virtual addresses and ensure they are non-cachable.
- keyword `volatile` tells the compiler not to optimize, i.e. read and write directly to RAM.
- Must assign physical addresses to devices at boot time to avoid conflicts.

# DMA buffers



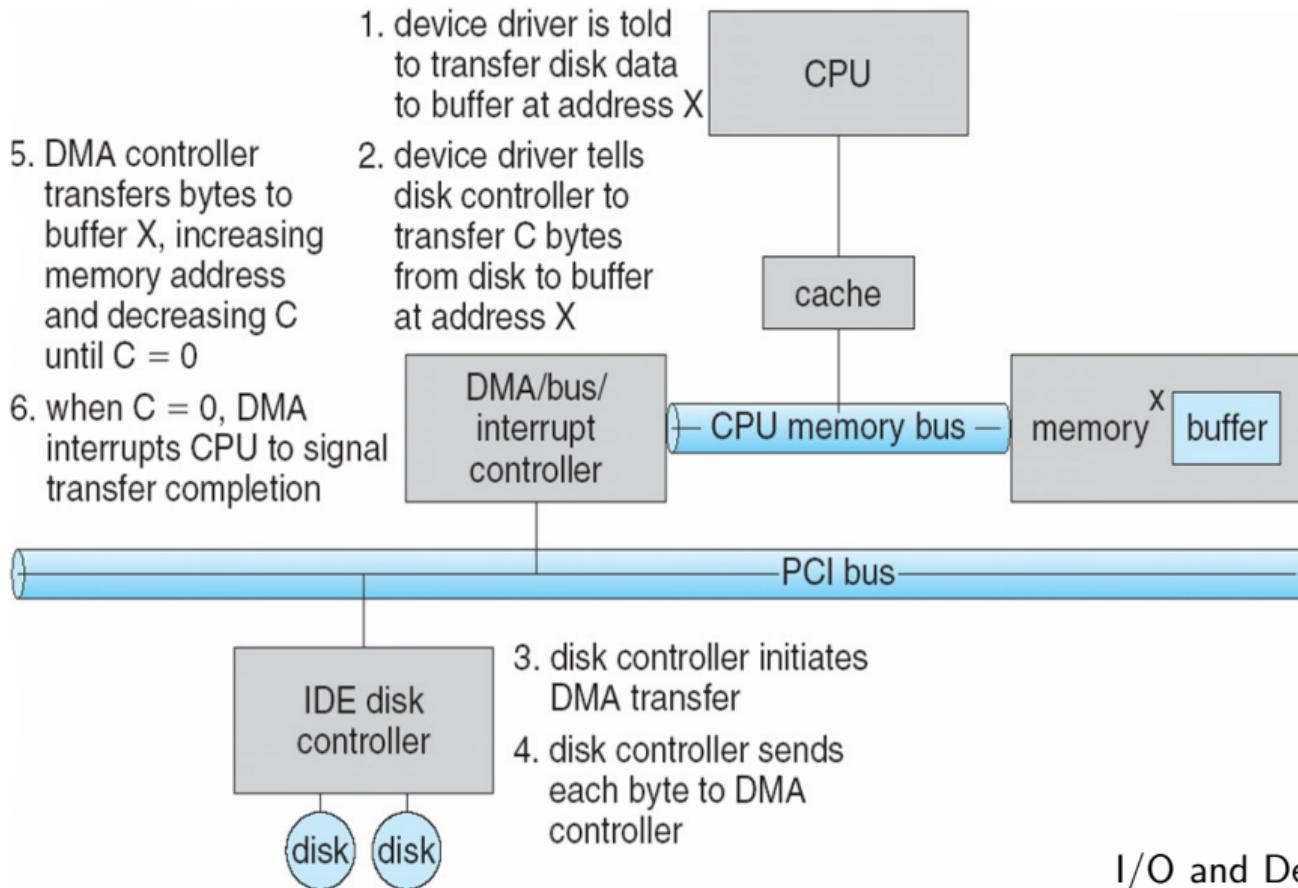
- Idea: *use processor to give commands to transfer data* and let device manage transfer.
- Why: interacting with devices is slow, so this frees up the processor for other tasks.
- Include a list of buffer locations in main memory to transfer (start addr and length).
- The device reads the list and accesses the buffers through DMA.

## Example: Network Interface Card



- Link interface manages data to and from wire, fiber, or antenna.
- FIFOs buffers on card provide small amount of buffering.
- The bus interface logic uses DMA to move packets to and from buffers in main memory.

## Example: IDE disk read w. DMA



# Driver architecture

- Device driver provides several commands (a.k.a. entry points) to the kernel.
  - Reset, ioctl (io control), output, interrupt, read, write, strategy ...
- Key Question: How should the driver synchronize with device?
  - E.g., Need to know when *transmit buffers free or data is available*.
  - Need to know *when the request is complete* or if there was an error.
- One approach: **Polling**
  - Sending a packet? Loop asking card when buffer is free.
  - Waiting to receive? Keep asking card if it has packet.
  - Disk I/O? Keep looping until disk ready bit set.
- Disadvantages of polling?
  - *Cannot use CPU for anything else while polling*.
  - Or schedule poll in future and do something else, but then high latency to receive packet or process disk block.

# Interrupt driven devices

- Instead, ask device to *interrupt processor when an event happens*, i.e. **interrupt driven devices**.
  - The interrupt handler runs at high priority.
  - Handler asks device what happened (e.g. data ready, buffer free, request complete.)
  - This is what most general-purpose OSes do for most devices.
- Bad under high network packet arrival rate.
  - Packets can arrive faster than OS can process them.
  - Interrupts are very expensive because they involve a context switch.
  - Since interrupt handlers have high priority, in worst case, can spend 100% of time in interrupt handler and never make any progress called **receive livelock**.
  - FYI: there are alternatives, e.g. adaptive switching between interrupts and polling.
- Interrupt driven devices are very good for disk requests.

## Anatomy of a disk [Ruemmler]

- Stack of magnetic platters (a.k.a. disks).
  - Rotate together on a central spindle @3,600-15,000 RPM
  - *Drive speed drifts slowly over time.*
  - Can't predict rotational position after 100-200 revolutions
- Disk arm assembly
  - Arms rotate around pivot, all move together.
  - Pivot offers some resistance to linear shocks.
  - Arms contain disk heads—one for each recording surface.
  - Heads read data from and write data to platters.

# Disk



# Disk



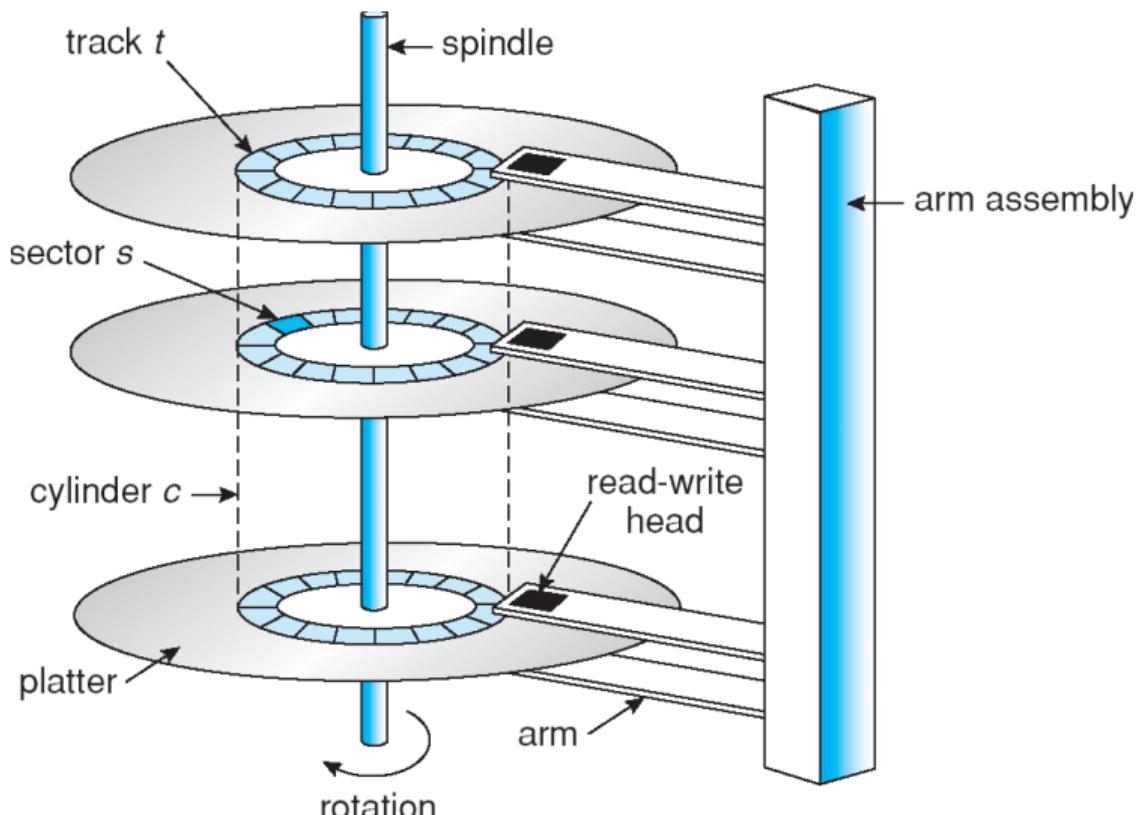
# Disk



# Storage on a magnetic platter

- Platters are organized as concentric circles called **tracks**.
- Both sides (**surfaces**) of the platter are used.
- A stack of tracks all of the same radius for each platter and each surface is called a **cylinder**.
- **Read/write heads** *sense and record data along one cylinder at a time.*
  - Significant fractions of the encoded stream are for error correction.
- Generally only one read/write head active at a time.
  - Disks usually have one set of read-write circuitry.
  - Must worry about cross-talk (electrical interference) between channels.
  - Hard to keep multiple heads exactly aligned so they all move together.

# Cylinders, tracks, & sectors



# Disk positioning system

- The disk positioning system moves the read/write heads to specific cylinder and keeps it there.
  - It resists physical shocks, imperfect tracks, etc.
- The time it takes to move the read/write heads to the correct cylinder is called the **seek** time.
- A seek consists of up to four phases:
  - **speedup**—accelerate arm to max speed or half way point
  - **coast**—at max speed (for long seeks)
  - **slowdown**—stops arm near destination
  - **settle**—adjusts head to actual desired track
- Very short seeks dominated by settle time ( $\sim 1$  ms).
- Short (200-400 cyl.) seeks dominated by speedup
  - Accelerations of  $\approx 40g$

## Seek details

- *Head switch times comparable to short seek times.*
  - May also require head adjustment.
  - Settles take longer for writes than for reads. – Why?
    - If read strays from track, the system can catch the error with checksum and retry.
    - If write strays, you've just overwritten another track.
- The disk controller circuit keeps table of pivot motor power.
  - It *maps the seek distance to the power and time needed* to move that distance.
  - Disk interpolates over entries in table.
  - Table set by periodic “thermal recalibration”
  - But, e.g.,  $\approx 500$  ms recalibration every  $\approx 25$  minutes bad for responsiveness.
- “Average seek time” quoted can be many things
  - E.g. time to seek 1/3 disk, 1/3 time to seek whole disk, etc.

# Sectors

- *The disk interface typically presents the disk as a linear array of blocks* called **logical block addressing (LBA)** rather than the older **cylinder, head, sector (CHS)** method.
  - Generally 512 bytes, written atomically (even if power failure)
- Disk maps logical locations to physical sectors.
  - **Zoning**—puts more sectors on larger (i.e. outside) tracks.
  - **Track skewing**—sector 0 position varies by track to improve sequential access times.
  - **Sparing**—flawed sectors (not properly storing data) are remapped to locations.
- The *OS doesn't know logical to physical mapping*.
  - Larger logical number difference does mean larger seek times.
  - But it is a highly non-linear relationship and it depends on the zone.
  - The OS has no info on rotational positions.
  - But it can empirically build a table to estimate times.

# Cost Model for Disk I/O

The time it takes to move data to/from a disk involves:

1. **seek time:** the time it takes to move the read/write heads to the appropriate track
  - depends on **seek distance**, the distance (in tracks) between previous and current request
  - values range from 0 milliseconds to the max seek time
2. **rotational latency:** the time it takes for the desired sectors spin to the read/write heads
  - depends on rotational speed of disk
  - value range from 0 milliseconds to the time for a single rotation
3. **transfer time:** the time it takes until the desired sectors spin under the read/write heads
  - depends on the rotational speed of the disk and the amount of data accessed

Request Service Time =

$$\text{Seek Time} + \text{Rotational Latency} + \text{Transfer Time}$$

# Request Service Time Example

## Parameters

- Disk Capacity:  $2^{32}$  bytes.
- Number of Tracks:  $2^{20}$  tracks per surface.
- Number of Sectors per Track  $2^8$  sectors per track.
- Rotations per Minute (RPM): 10000 RPM
- Maximum Seek (time): 30 milliseconds

a) How many bytes are in a track?

$$\begin{aligned}\text{BytesPerTrack} &= \text{DiskCapacity}/\text{NumTracks} \\ &= 2^{32}/2^{20} = 2^{12} \text{ bytes per track}\end{aligned}$$

b) How many bytes are in a sector?

$$\begin{aligned}\text{BytesPerSector} &= \text{BytesPerTrack}/\text{NumSectorsPerTrack} \\ &= 2^{12}/2^8 = 2^4 \text{ bytes per sector}\end{aligned}$$

c) What is the maximum rotational latency?

$$\begin{aligned}\text{MaxLatency} &= 60/\text{RPM} \\ &= 60/10000 = 0.006 \text{ or } 6 \text{ milliseconds}\end{aligned}$$

# Request Service Time Example

## Parameters

- Disk Capacity:  $2^{32}$  bytes.
- Number of Tracks:  $2^{20}$  tracks per surface.
- Number of Sectors per Track  $2^8$  sectors per track.
- Rotations per Minute (RPM): 10000 RPM
- Maximum Seek (time): 30 milliseconds

d) What is the average seek time?

$$\begin{aligned}\text{AverageSeek} &= \text{MaxSeek}/3 \\ &= 30/3 = 10 \text{ ms average seek time}\end{aligned}$$

e) What is the average rotational latency assuming a uniform distribution?

$$\begin{aligned}\text{AveRotationalLatency} &= \text{MaxLatency}/2 \\ &= 6/2 = 3 \text{ ms average rotational latency}\end{aligned}$$

# Request Service Time Example

## Parameters

- Disk Capacity:  $2^{32}$  bytes.
- Number of Tracks:  $2^{20}$  tracks per surface.
- Number of Sectors per Track  $2^8$  sectors per track.
- Rotations per Minute (RPM): 10000 RPM
- Maximum Seek (time): 30 milliseconds

f) What is the cost to transfer 1 sector?

$$\begin{aligned}\text{SectorLatency} &= \text{MaxLatency}/\text{NumSectorsPerTrack} \\ &= 6/2^8 = 6/256 = 0.0234 \text{ ms per sector}\end{aligned}$$

g) What is the expected cost to read 10 consecutive sectors?

Expected Request Service Time

$$\begin{aligned}&= \text{AveSeek} + \text{AveRotationalLatency} + \text{TransferTime} \\ &= 10 + 3 + 10 \times 0.0234 = 13.234 \text{ ms}\end{aligned}$$

**Note:** Since we do not know the position of the head, or the platter, we use the average seek and average rotational latency.

# Disk interface

- The disk controller controls the hardware and mediates access.
- The processor and the disk are often connected by bus (e.g., SCSI)
  - Multiple devices may contend for the bus (and so some have to wait).
- Possible disk/interface features:
- Disconnect from bus during requests
- *Command queuing*: Give disk multiple (e.g. up to 32) requests
  - The disk can *make scheduling decisions* using the current position of the read/write head along with rotational information.
- Disk cache used for *read-ahead*.
  - Otherwise, sequential reads would incur whole revolution
  - Cross track boundaries? Can't stop a head-switch
- Some disks support *write caching*.
  - But data not stable—not suitable for all requests

# SCSI overview [Schmidt]

- SCSI *domain* consists of devices and an SDS bus.
  - Devices: *host adapters (system bus to SCSI bus) and SCSI controllers (bus to device)*.
  - *Service Delivery Subsystem* connects the devices—e.g., SCSI bus
- SCSI-2 bus (SDS) connects up to 8 devices
  - Controllers can have > 1 “logical units” (LUNs)
  - Typically, controller built into disk and 1 LUN/target, but “bridge controllers” can manage multiple physical devices.
- Each device can assume role of *initiator* or *target*.
  - Traditionally, the *host adapter was the initiator* and the *controller the target*
  - Now controllers act as initiators (e.g., COPY command)
  - Typical domain has 1 initiator,  $\geq 1$  targets

# SCSI requests

- *A request is a command from initiator to target.*
  - Once transmitted, target has control of bus
  - Target may disconnect from bus and later reconnect  
(very important for multiple targets or even multitasking)
- Commands contain the following:
  - *Task identifier*—initiator ID, target ID, LUN, tag
  - *Command descriptor block*—e.g., read 10 blocks at pos.  $N$
  - *Optional task attribute*—SIMPLE, ORDERED, HEAD OF QUEUE
  - *Optional*: output/input buffer, sense data
  - *Status byte*—GOOD, CHECK CONDITION, INTERMEDIATE, ...

# Executing SCSI commands

- *Each logical unit (LUN) maintains a queue of tasks.*
  - Each task is DORMANT, BLOCKED, ENABLED, or ENDED
  - SIMPLE tasks are dormant until no ordered/head of queue
  - ORDERED tasks dormant until no HoQ/more recent ordered
  - HoQ tasks begin in enabled state
- Task management commands available to initiator
  - Abort/terminate task, Reset target, etc.
- Linked commands
  - *Initiator can link commands, so no intervening tasks.*
  - E.g., could use to implement atomic read-modify-write
  - Intermediate commands return status byte INTERMEDIATE

# SCSI exceptions and errors

- *After error stop executing most SCSI commands.*
  - Target returns with CHECK CONDITION status
  - Initiator will eventually notice error
  - Must read specifics w. REQUEST SENSE
- *Prevents unwanted commands from executing.*
  - E.g., initiator may not want to execute 2nd write if 1st fails
- *Simplifies device implementation.*
  - Don't need to remember more than one error condition
- Same mechanism used to notify of media changes
  - I.e., ejected tape, changed CD-ROM

# Disk performance

- Placement and ordering of requests is a huge issue.
  - Sequential I/O much, much faster than random.
  - Long seeks are much slower than short ones.
  - The power might fail at any time, leaving the drive in an inconsistent state.
- Must be careful about order for crashes (e.g. soft updates).
- Try to achieve contiguous accesses where possible (e.g. XFS)
  - E.g., make big chunks of individual files contiguous
- *Try to order requests to minimize seek times* (e.g. BSD's Fast File System)
  - The OS can only do this if it has multiple requests to order.
  - Requires disk I/O concurrency (handling multiple requests from multiple processes).
  - High-performance apps try to maximize I/O concurrency.
- Key Question: *How to schedule concurrent requests?*

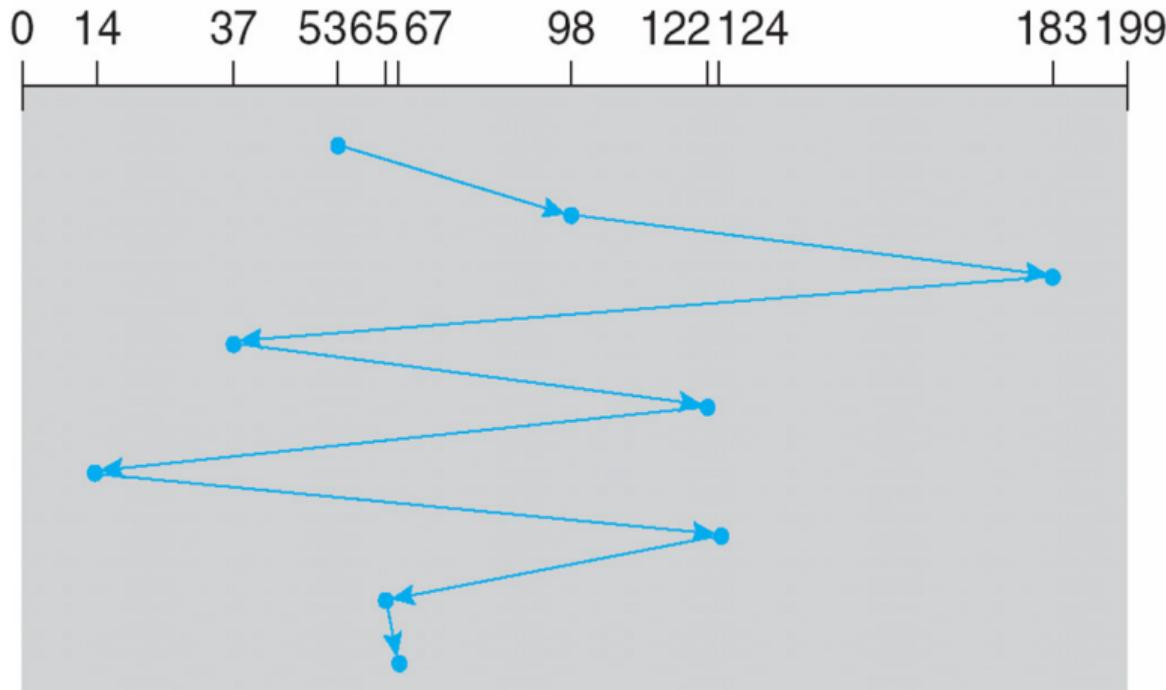
# Scheduling: FCFS

- **First Come First Served (FCFS)**
  - *Process the disk requests in the order they are received.*
- Advantages
  - Easy to implement
  - Good fairness
- Disadvantages
  - Cannot exploit request locality to minimize seek time.
  - *Increases average latency* (delay to start transfer) and *decreasing throughput* (I/O operations completed per second).

## FCFS example

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



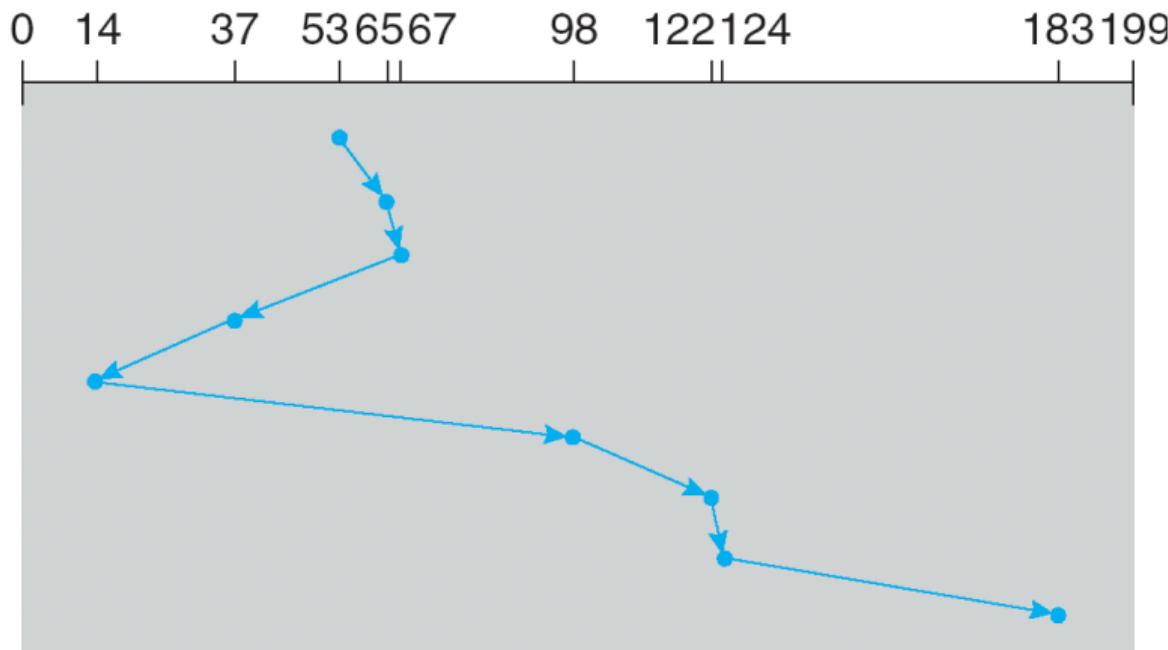
# Shortest positioning time first (SPTF)

- **Shortest positioning time first (SPTF)**
  - *Always pick the request with shortest seek time.*
- Also called Shortest Seek Time First (SSTF)
- Advantages
  - *Exploits locality of disk requests to reduce seek time.*
  - Higher throughput
- Disadvantages
  - *Starvation.*
  - Don't always know what request will be fastest.
- Improvement: *Aged SPTF ?*
  - Give older requests higher priority (similar to multilevel feedback queues).
  - Adjust “effective” seek time with weighting factor:  
 $T_{\text{eff}} = T_{\text{pos}} - W \cdot T_{\text{wait}}$

## SPTF example

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



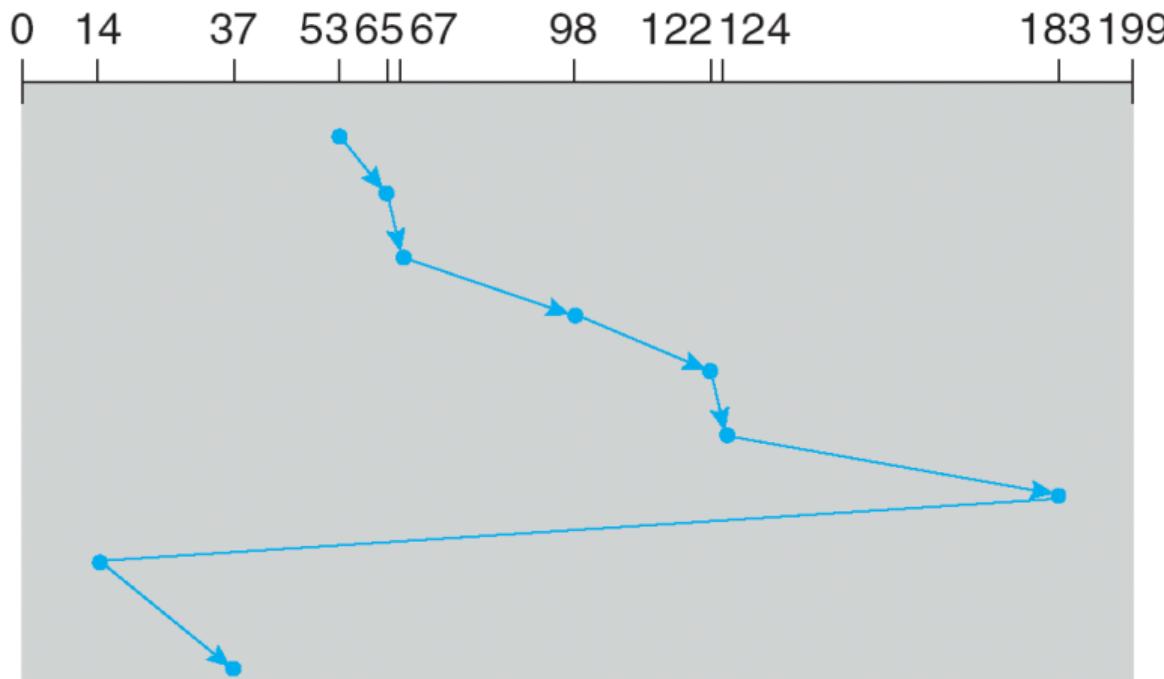
# Elevator scheduling (SCAN)

- **Elevator scheduling (SCAN).**
- *Sweep across the disk and service all requests as you approach that track.*
  - Like SPTF, but next seek must be in same direction.
  - Switch directions only if no further requests in that direction.
- Advantages
  - Takes advantage of locality.
  - Provides bounded waiting.
- Disadvantages
  - *Tracks in the middle get better service because they are serviced in both directions.*
  - Might miss locality SPTF could exploit.
- C-SCAN: Only sweep in one direction and then return to the beginning again.
  - Very commonly used algorithm in Unix.
  - Does not favour middle tracks but when returning the read/write head moves the full radius of the disk without any data transfers.

## C-SCAN example

queue 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



# VSCAN(r)

- **VSCAN(r)** creates a continuum between SPTF and SCAN.
- Like SPTF, but introduces an **effective positioning time**.
- If the *request is in same direction* as the previous seek:  $T_{\text{eff}} = T_{\text{pos}}$   
otherwise (i.e. the *request is in the opposite direction*)  $T_{\text{eff}} = T_{\text{pos}} + r \cdot T_{\text{max}}$   
where  $T_{\text{max}}$  is the maximum seek time.
- I.e.  $r \cdot T_{\text{max}}$  is a *penalty for changing directions*, and if even with the penalty for changing directions  $T_{\text{eff}}$  is lower, then change directions.
  - When  $r = 0$  (i.e. no penalty for changing direction) you get SPTF.
  - When  $r = 1$  (i.e. the largest possible penalty for changing direction) you get SCAN.
  - $r = 0.2$  works well in practice.
- Advantages and disadvantages
  - Those of SPTF and SCAN, depending on how  $r$  is set.
- See [Worthington] for good description and evaluation of various disk scheduling algorithms

# Flash memory

- Today, people are increasingly using flash memory.
- It is completely solid state, i.e. there are no moving parts.
  - It remembers data by storing a charge in a modified MOSFET transistor.
  - It has lower power consumption and heat than an HDD.
  - It also does not have mechanical seek times to worry about.
- *It only has a limited number of overwrites possible.*
  - Blocks wear out after 10,000 (MLC) – 100,000 (SLC) erases.
  - Requires **flash translation layer** (FTL) to provide **wear leveling**, so repeated writes to logical block don't wear out physical block.
  - FTL can seriously impact performance
  - In particular, random writes are *very expensive* [Birrell]
- *It has limited durability, i.e. the charge can wear out over time.*

# Types of flash memory

- **NAND flash** (most prevalent for storage).
  - It allows for higher density and is the most used one for storage.
  - It has faster erase and write times.
  - But there are more errors internally, so it needs error correction.
- **NOR flash**
  - Faster reads in smaller data units.
  - Can execute code straight out of NOR flash.
  - Significantly slower erases.
- Single-level cells (SLC) store 1 bit vs. Multi-level cells (MLC) store 2 bits vs. Triple-level cells (TLC) store 3 bits vs. Quad-level cell (QLC) store 4 bits.
  - The multiple bits are encoded by having more voltage levels (2, 4, 8, or 16) per cell.
  - More levels are slower to write.

# NAND Flash Overview

- Flash device has 2112-byte **pages**.
  - 2048 bytes of data + 64 bytes metadata and error correction code (ECC)
- **Blocks** contain 64 (SLC) or 128 (MLC) pages.
- Blocks divided into 2–4 *planes*
  - All planes contend for same package pins (i.e. input/output).
  - But can access their blocks in parallel to overlap latencies,
- *Can read one page at a time.*
  - Takes 25  $\mu\text{s}$  + time to get data off chip.
- But *must erase a whole block before reusing it.*
  - Erasing sets all bits to 1—very expensive (2 msec)
  - Programming pre-erased block requires moving data to internal buffer, then 200 (SLC)–800 (MLC)  $\mu\text{s}$

## Flash Characteristics [Caulfield'09]

Parameter	SLC	MLC
Density Per Die (GB)	4	8
Page Size (Bytes)	2048+32	2048+64
Block Size (Pages)	64	128
Read Latency ( $\mu$ s)	25	25
Write Latency ( $\mu$ s)	200	800
Erase Latency ( $\mu$ s)	2000	2000
40MHz, 16-bit bus	Read b/w (MB/s)	75.8
	Program b/w (MB/s)	20.1
133MHz	Read b/w (MB/s)	126.4
	Program b/w (MB/s)	20.1
		5.0

Here b/w means bandwidth and program means to write data.

# Preparing for the Final

- I have made a complete copy of my slides available on Learn.
- I have a pinned post in Piazza listing any typos.
- Will have Final Exam [official] post in Piazza by Wednesday.
- We will have extra office hours before the final. See the pinned post in Piazza.
- We will be monitoring and answering questions in Piazza.
- *Good luck on the final!*
- *Good luck with your other courses!*
- *Thank-you for your attention!*

## A Request

If you have not already done so...

- You need a web enabled device.
- Go to <https://perceptions.uwaterloo.ca/>
- Please take a few minutes to complete the evaluation.
- You have until Tuesday, December 5, 11:59 p.m.
- *Your feedback is appreciated. I use it to continually improve the course.*