

# Introduction to CS350

Lesley Istead, Kevin Lanctot

David R. Cheriton School of Computer Science  
University of Waterloo

Winter 2019

# Table of Contents

Introduction	1
Threads and Concurrency	27
Synchronization	85
Processes and the Kernel	161
Virtual Memory	230
Scheduling	324
Devices and I/O	365
File Systems	403

# Welcome to CS350 - Operating Systems!



- Administrative Information
- Introduction to Operating Systems

## Useful Information about the Instructor

- Name: Kevin Lanctot
- Office: DC 2131 (near the skywalk to the MC/M3 buildings)
- Office Hours: Thursdays 12:30-2:30 pm (starting next week)
- Email: [kevin.lanctot@uwaterloo.ca](mailto: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*.

# General Information

## Important links:

- <http://www.student.cs.uwaterloo.ca/~cs350>  
Course personnel, office hours, readings, assignments, tutorials, previous midterms, review problems, etc.
- <https://piazza.com/uwaterloo.ca/winter2019/cs350/home>
  - Piazza will be used for announcements, extra notes, questions, corrections, etc.
  - Please check piazza regularly.
  - *Do not post your code in public piazza posts;* use private posts when appropriate.

# Course Readings

## Course Notes

- Course notes are required.
- *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.
- The course notes are available online from the course website or you may purchase a printed copy, if you desire.

## Textbook

- It is *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.

## My Version of the Slides

- My version of the slides just takes the standard slides and occasionally adds additional material.
- The inserted slides will have “Additional Notes” at the bottom and a slide number with a decimal point, i.e. slide 4.1 is an additional slide that occurs between slides 4 and 5.
- I will use red bold letters for any technical terms that you are expected to know as part of this course, such as **threads**.
- I will use *blue italics for key points on the slide*.
- This version of the slides will be available on Learn, before 10 am the day of the lecture.

## Grading Scheme

$A0, A1, A2a, A2b, A3$ : Assignment marks as a percentage

$M$ : Midterm exam grades as percentages

$F$ : Final exam grade as a percentage

$$\text{Normal} = 0.02 \times A0 + 0.08 \times A1 + 0.07 \times A2a + 0.08 \times A2b + 0.10 \times A3 \\ + 0.20 \times M + 0.45 \times F$$

$$\text{Exams} = (0.20 * M + 0.45 * F) / 0.65$$

```
if (Exams < 50%) {  
    FAIL EXAMS, FAIL THE COURSE  
    Course Grade = min(Normal, Exams)  
} else {  
    Course Grade = Normal  
}
```

*You will fail this course if you fail the weighted exam average,  
regardless of your assignment grades.*

# Assignments

There are 5 assignments.

- *All assignments are to be done individually.*
- You will not be writing your own OS.
- You will be adding/fixing features of an existing OS.
- We use OS/161 (~22,000 lines for kernel), which runs on SYS/161 (MIPS simulator/VM)

Slip days:

- Allows flexibility in assignment deadlines
- Total of 5 slip days
- Can use maximum of 3 slip days per assignment (~~except A3~~)
- You will receive a 0.4% bonus on your final grade for each unused slip day (for a maximum bonus of 2%).

# Assignments

## Level of Difficulty

- the assignments (particularly A2a, A2b and A3) are *significantly more difficult* than what you encountered in CS246
- the bugs will be more difficult to find
- start early
- understand, then design, then implement
- allow lots of time to debug
- use revision control software (Git)
- know how to use the symbolic debugger GDB

# Plagiarism and Academic Offenses

There are two types of assignments.

- *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.
- *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

[https://uwaterloo.ca/academic-integrity/sites/ca.academic-integrity/files/uploads/files/revised\\_undergraduate\\_tip\\_sheet\\_2013\\_final.pdf](https://uwaterloo.ca/academic-integrity/sites/ca.academic-integrity/files/uploads/files/revised_undergraduate_tip_sheet_2013_final.pdf)

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.
- Often: 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.

If you have taken this course before, you may reuse your previous code if:

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

# Topic 1: Introduction to Operating Systems

What happens when I open a file, say *blah.txt*, located in a folder/directory called *temp* and then I try to change the name of the directory?

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 Toshiba Model MQ01ABD100 1TB Hard Disk Drive connected via a SATA interface to an Intel 7 Series Chipset Family SATA AHCI Controller and the file system is NTFS?

Answer:

Conclusion: An operating system is *part cop, part facilitator*.

# Topic 1: Introduction to Operating Systems

Example 1: The System Stack: What is unusual here?

Answer:

Example 2: What is unexpected about the location of the global variable gv and the function main?

Answer:

Example 3: Why isn't the output always 0?

Answer:

We will be answering these questions in more detail in the coming weeks but basically the answer has to do with the role of the OS in computer programming.

# What is an Operating System?

Generally, an OS is a system that:

- *manages resources* (e.g. processor, memory, I/O)
- *creates execution environments* (i.e. interfaces to resources)
- loads programs
- provides common services and utilities

## Operating Systems

- originated 1951, LEO I from J. Lyons and Co.
- started as simple I/O libraries, batch processors

# Three views of an Operating System

Each view addresses a different question.

1. **Application View:** What services does it provide?
2. **System View:** What problems does it solve?
3. **Implementation View:** How is it built?

Let's look a bit more at each of these three views...

View 1: Application View of an Operating System

- (As mentioned earlier) from a programmer's point of view the application view is that...  
An operating system is part cop (i.e. it *provides protection* from program errors), part facilitator (i.e. it *provides an abstract interface* to the underlying system).
- More specifically ...

# View 1: Application View of an Operating System

**Key Question:** What services does an OS provide?

The OS provides an execution environment for running programs. The execution environment:

- provides a program with *resources* (e.g. the processor time, memory space, access to I/O devices like the keyboard and monitor) that it needs to run,
- provides *interfaces* through which a program can use networks, storage, I/O devices, and other system hardware components,
  - (these interfaces provide a simplified, abstract view of hardware to application programs)
- *isolates running programs* from one another and prevents undesirable interactions among them (such as an error in one program causing another program to crash).

## View 2: System View of an Operating System

*Key Question:* What problems does an OS solve?

The OS:

- *manages the hardware resources* of a computer system. Resources include processors, memory, disks and other storage devices, network interfaces, I/O devices such as keyboards, mice and monitors, etc.
- *allocates resources* among running programs.
- *controls the access to* or sharing of resources among programs.

The OS itself also uses resources, which it must share with application programs.

## View 3: Implementation View of an Operating System

*Key Question:* How is it built?

The OS is a **concurrent, real-time** program.

**Concurrent** means multiple programs or sequences of instructions *running, or appearing to run, at the same time.*

A **real-time** program is a program that *must respond to an event* (such as data arriving from the network) *in a specific amount of time* (depending on the event). E.g. when you are watching a video you do not want it to freeze up or skip.

- Concurrency arises naturally in an OS when it supports concurrent applications, and because it must interact directly with the hardware.
- Hardware interactions also impose timing constraints.

How does the OS implement concurrency and hardware interactions?

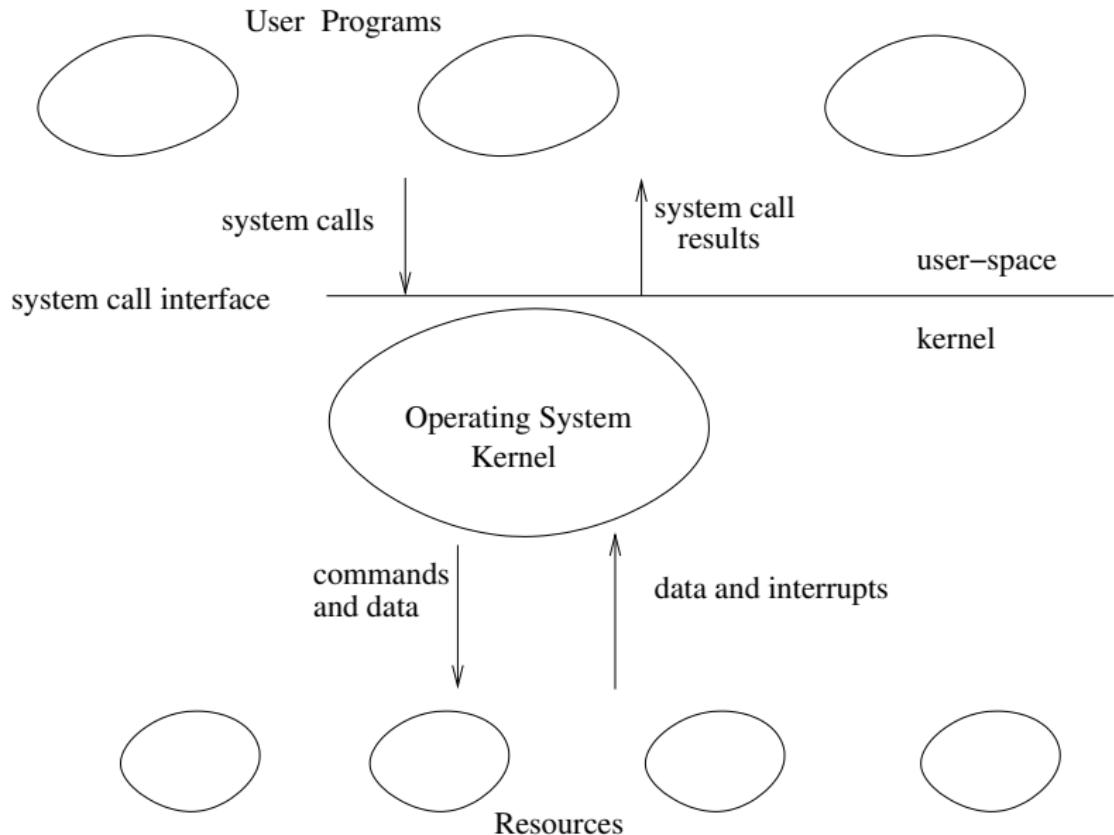
**kernel:** The operating system kernel is the part of the operating system that responds to system calls, interrupts and exceptions.

- You can think of it as the part of the OS that is always running from boot-up to shut down.

**operating system:** The operating system as a whole includes the kernel, and may include other related programs that provide services for applications. This may include things like:

- utility programs (e.g. disk defragmenter, screen saver)
- command interpreters (e.g. cmd.exe in Windows, bash in Linux)
- programming libraries (e.g. POSIX threads in Linux)

# Schematic View of an Operating System



# Schematic View of an Operating System

- The **user programs** are programs that the user interacts with directly, e.g. desktop environments (no choice for Windows or macOS), web browsers, games, editors, compilers, etc.
- The **user space** refers to all programs that run outside the kernel. These programs do not interact with the hardware directly. They run in one region of system memory.
- A **system call** is how a user process interacts with the OS. A programmer uses the C function `fopen`, which is implemented by making a system call to the OS (in Linux it would be `sys_open`).
- **Kernel space** is the region of memory where the kernel runs.
- Resources are the hardware, e.g. processor(s), RAM, graphics card, system clock, keyboard, mouse, monitor, etc.

# Operating System Abstractions

The **execution environment** provided by the OS includes a variety of **abstract entities** that can be manipulated by a running program.

Examples of these abstractions:

**address spaces** → primary memory (RAM)

**files and file systems** → secondary storage (HDD or SSD)

**processes, threads** → program execution

**sockets, pipes** → network or other message channels

This course will cover

- why these abstractions are designed the way they are
- how these abstractions are manipulated by application programs
- how these abstractions are implemented by the OS

# Course Coverage

1. Introduction (Lecture 1)
2. Threads and Concurrency (Lectures 2–3)
3. Synchronization (Lectures 4–6)
4. Processes and the Kernel (Lectures 7–8)
5. Virtual Memory (Lectures 9–15)
6. Scheduling (Lecture 16)  
*Review* (Lecture 17)
7. I/O, Devices, and Device Management (Lectures 18–20)
8. File Systems (Lectures 21–23)
9. Virtual Machines (Lecture 24)

Note that this lecture schedule is only approximate.

# Threads and Concurrency

**key concepts:** threads, concurrent execution, timesharing,  
context switch, interrupts, preemption

Lesley Istead, Kevin Lanctot

David R. Cheriton School of Computer Science  
University of Waterloo

Winter 2019

# What is a thread?

A **thread** is a sequence of instructions.

- A normal **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.

A single program can have

- *different threads responsible for different roles* within the program, e.g. running Javascript, displaying HTML and playing a video in a browser.
- *multiple threads responsible for the same roles* within the program, e.g. a webserver may have one thread for each person that is currently using the web site.

# Why Threads?

- Threads provide a way for programmers to express concurrency in a program.
- Recall: Concurrency is when multiple programs or sequences of instructions make progress, i.e. *run or appearing to run*, at the same time.
- **Parallelism** is when multiple programs or sequences of instructions can *run* at the same time (because there are multiple processors or multiple cores).
- In threaded concurrent programs there are multiple threads of execution, all occurring at the same time.

# Why Threads?

- Parallelism exposed by (i.e. made available by the implementation of) threads enables parallel execution if the underlying hardware supports it (i.e. multicore processor or multiple processors) ⇒ *programs can run faster*
- Parallelism exposed by threads enables better processor utilization ⇒ *when one thread blocks, another may be able to run*
- A thread **blocks**, when it ceases execution for a period of time, or, until some condition has been met.
- When a thread blocks, it is not executing instructions—the CPU is idle.
- Concurrency lets the CPU execute a different thread during this time.

# Why Threads?

## Why use threads?

- *Efficient Use of Resources*: While one thread is waiting (say for a web site to respond) another thread can be running.
- *Parallelism*: Different threads can be running on different cores or different processors to increase throughput.
- *Responsiveness*: For example, one thread could be responsible for the user interface and others for loading a web page.
- *Priority*: You can have some threads (such as user interface code) run at higher priority than others.
- *Modular Code*: For example, separate out the user interface code from the web page loading code.

## Example: Traffic Simulation

- vehicles are trying to pass through an intersection
- each thread simulates a sequence of vehicles attempting to pass through the intersection
- each thread simulates one vehicle at a time
- since there are multiple concurrent threads, there may be multiple vehicles attempting to enter the intersection concurrently
- synchronize the vehicles so that they do not collide in the intersection
- two vehicles may be in the intersection at the same time as long as their paths will not result in a collision, e.g.
  - a car going North-South and a car going South-North *are allowed* in the intersection at the same time
  - a car going North-South and a car going East-West *are not allowed* in the intersection at the same time

## Example: Creating Threads Using `thread_fork()`

An example of using `thread_fork` to create a new thread that is running the function `vehicle_simulation`

From `kern/synchprobs/traffic.c`.

```
for (i = 0; i < NumThreads; i++) {
    error = thread_fork("vehicle_simulation thread", NULL,
        vehicle_simulation, NULL, i);

    if (error) {
        panic("traffic_sim: thread_fork failed: %s",
            strerror(error));
    }
}
```

## Example: Traffic Simulation

- *Simulates the arrival of vehicles to the intersection from one direction*
- Arguments:
  - void \* unusedpointer: currently unused
  - unsigned long origin: vehicle arrival Direction
- Returns: nothing
- Notes: Each simulation thread runs this function

```
static void vehicle_simulation(void * unusedpointer,
                               unsigned long thread_num)
{
    int i;
    Vehicle v;

    /* set up parameters for simulation */ ...
}
```

Key ideas from the example:

- A thread can create new threads using `thread_fork`.
- New threads start execution in a function specified as a parameter to `thread_fork`.
- The original thread (which called `thread_fork`) and the new thread (which is created by the call to `thread_fork`) proceed concurrently, as two simultaneous sequential threads of execution.
- All threads *share* access to the program's *code, global variables and heap*.
- Each thread's *stack*, containing procedure activation records (a.k.a. stack frames) are *private to that thread*.

In the OS a thread is represented as a structure or object.

# OS/161's Thread Interface

- create a new thread:

```
int thread_fork(
    const char *name,           // name of new thread
    struct proc *proc,          // thread's process
    void (*func)                // new thread's function
    (void *, unsigned long),
    void *data1,                // function's first param
    unsigned long data2         // function's second param
);
```

- terminate the calling thread:

```
void thread_exit(void);
```

- voluntarily yield execution:

```
void thread_yield(void);
```

See kern/include/thread.h

# Other Thread Libraries and Functions

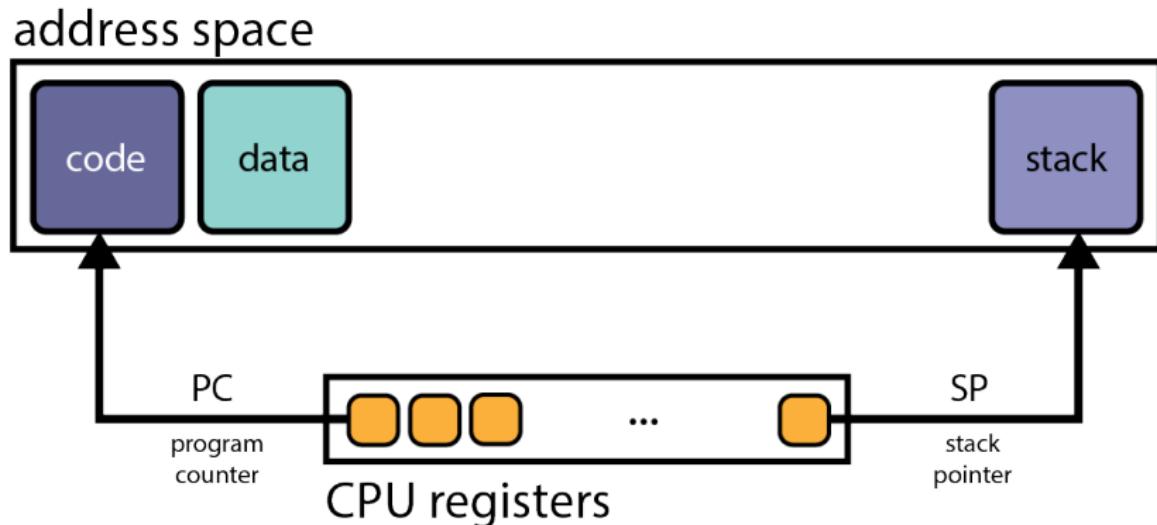
Additional examples (beyond the scope of this course)

- `join` is a common thread function to force one thread to block until another finishes is not offered by OS/161
- `pthreads`: i.e. POSIX threads, is a well-supported, popular, and sophisticated thread API
- OpenMP: is a cross-platform, simple multi-processing and thread API
- GPGPU Programming: general-purpose GPU programming APIs, e.g. nVidia's CUDA, run threads on GPU instead of CPU

## History: Concurrency and Threads

- originated in 1950s to improve CPU utilization during I/O operations
- "modern" timesharing originated in the 1960s

# Review: Sequential Program Execution



## The Fetch/Execute Cycle

- 1) *fetch* the instruction PC points to (from the code segment)
- 2a) decode and *execute* the instruction
- 2b) increment the PC

# Review: MIPS Registers

num	name	use	num	name	use
0	z0	always zero	24-25	t8-t9	temps (caller-save)
1	at	assembler reserved	26-27	k0-k1	kernel temps
2	v0	return val/syscall #	28	gp	global pointer
3	v1	return value	29	sp	stack pointer
4-7	a0-a3	subroutine args	30	s8/fp	frame ptr (callee-save)
8-15	t0-t7	temps (caller-save)	31	ra	return addr (for jal)
16-23	s0-s7	saved (callee-save)			

See `kern/arch/mips/include/kern/regdefs.h`

## Review: MIPS Registers

In CS241 we simplified things a bit.

- *Register Names:* We called the MIPS registers \$0, \$1, \$2, etc. They can also be referred to by their default function, e.g. v0, a0, t0.
- *Passing Arguments:* We passed all arguments on the stack. In OS/161 we pass the first 4 args in registers a0-a3 and if there are more, the rest are passed on the stack.
- *Return Value(s):* Since the return value can sometimes occupy two registers, we reserve v0 and v1 for the return value.
- The CS241 instruction jalr is called jal here.
- *Saving Registers:* In CS241 the callee save most of the registers (except the Frame Pointer and the Return Address).
- This strategy has the short coming that the callee may be preserving register values that the caller is not using.

## Review: MIPS Registers

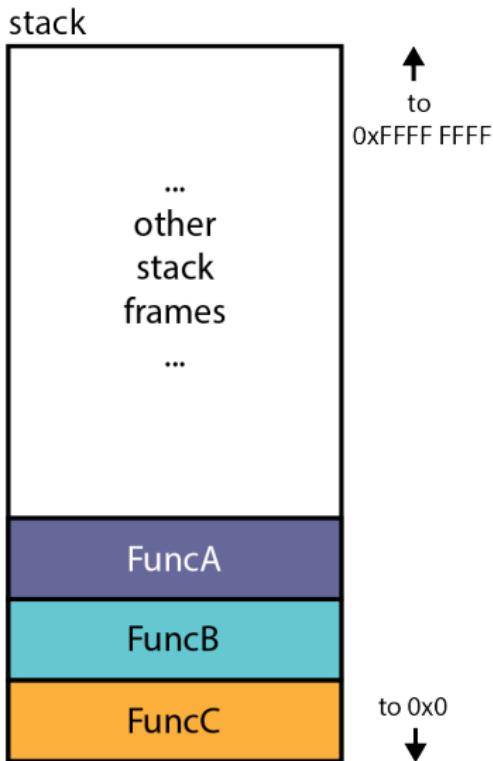
OS/161 uses a different strategy: registers are identified as ...

- **caller-save**, i.e. if the calling subroutine 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.
- **callee-save**, i.e. if the called subroutine uses one of these registers it must first store its current value on the stack and restore it before exiting.

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 (e.g. t0–t9).
- The callee will not preserve (store at the beginning and restore before exiting) these values.

# Review: The Stack



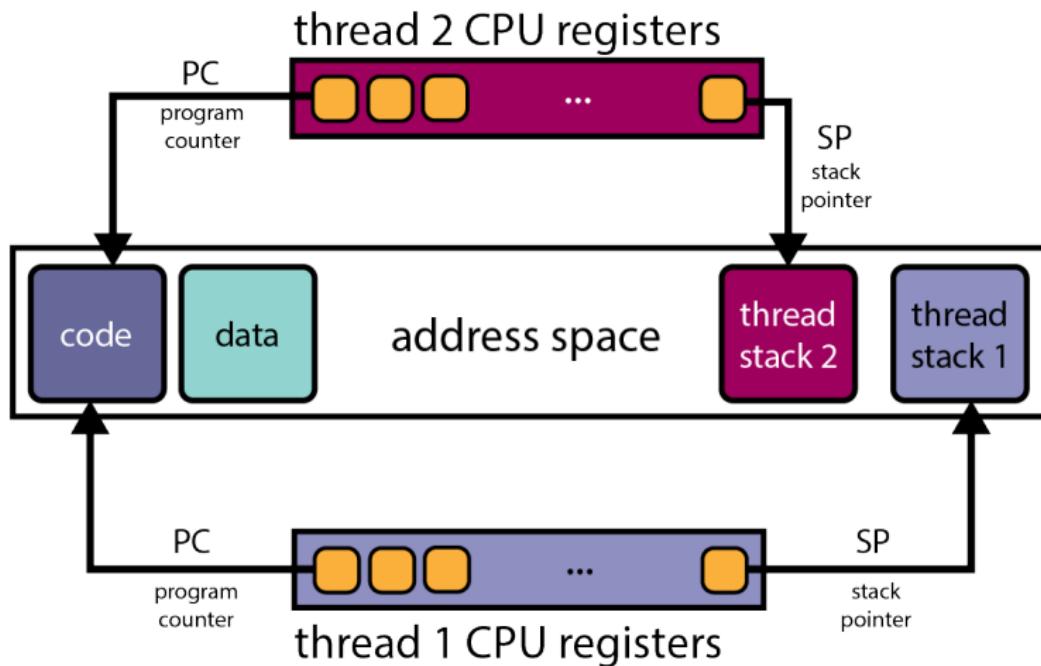
```
FuncA() {  
    ...  
    FuncB();  
    ...  
}  
  
FuncB() {  
    ...  
    FuncC();  
    ...  
}
```

# Review: The Stack

## OS/161's Stack Frame

- Note here the stack is drawn upside down from how you would likely see it in CS136 or CS241 ⇒ the “top” of the stack is shown at the bottom of the diagram, so *pushes and pops happen at the bottom of the slide.*
- When FuncA calls FuncB, the first thing FuncB does is create a stack frame which holds
  - any *additional arguments* to FuncB beyond the four that are passed in registers a0–a3,
  - any FuncB's *local variables*,
  - any *callee-save registers* that FuncB uses.
- If FuncB then calls FuncC, the first thing FuncC does is create its own stack frame.

# Concurrent Program Execution (Two Threads)



Conceptually, each thread executes sequentially using its private register contents and stack.

# Concurrent Program Execution (Two Threads)

**Key Question:** What is shared between threads?

- They are both executing (perhaps at the same time on different cores) the same program.
- *Both threads share* the same code, read-only data, global variables and heap.
- But each thread may be executing different functions within the code.
- E.g. one thread may be executing some Javascript while another is playing a video on the same browser tab.
- *Each thread has its own* stack and program counter (PC).
- E.g. one thread has a lot of Javascript functions' stack frames in its stack while the other has a lot of video decoding stack frame in its stack.

# Implementing Concurrent Threads

What options exist?

- *Hardware support:*  $P$  processors,  $C$  cores,  $M$  multithreading per core  $\Rightarrow$   $PCM$  threads can execute **simultaneously**.
- *Timesharing:* Multiple threads take turns on the same hardware; rapidly switching between threads so all make progress.
- *Hardware support + Timesharing:*  $PCM$  threads running simultaneously with timesharing.

Example: Intel i9-9900X

... 10 cores, each core can run 2 threads (multithreading degree). Therefore,  $P = 1$ ,  $C = 10$ , and  $M = 2$ , so  $PCM = 20$  threads can run simultaneously.

- **Multicore Processor:** a single die (or computer chip) can contain more than one processor unit. These units can share some components such as L2 and L3 cache, but execute code separately.
- **Multithreading:** in terms of hardware means having multiple threads run on the same core.
  - If one thread executes an instruction that can take many clock cycles (such as a load word that results in a cache miss), while waiting to get the data from RAM the core will execute code from another thread.
  - You would need specialized hardware to do this, such as two sets of registers, one for each thread.

# Timesharing and Context Switches

- When **timesharing** the switch from one thread to another is called a **context switch**.
- What happens during a context switch:
  - decide which thread will run next (scheduling)
  - save register contents of current thread
  - load register contents of next thread
- **Thread context** (register values) must be saved/restored carefully, since thread execution continuously changes the context.

## Timesharing

... each thread gets a small amount of time to execute on the CPU, when it expires, a context switch occurs. Threads **share** the CPU, giving the user the illusion of multiple programs running at the same time.

# Context Switch on the MIPS (1 of 2)

```
/* See kern/arch/mips/thread/switch.S */

switchframe_switch:
    /* a0: address of switchframe pointer of old thread. */
    /* a1: address of switchframe pointer of new thread. */

    /* Allocate stack space for saving 10 registers. 10*4 = 40 */
    addi sp, sp, -40

    sw    ra, 36(sp)      /* Save the registers */
    sw    gp, 32(sp)
    sw    s8, 28(sp)      /* a.k.a. frame pointer */
    sw    s6, 24(sp)
    sw    s5, 20(sp)
    sw    s4, 16(sp)
    sw    s3, 12(sp)
    sw    s2, 8(sp)
    sw    s1, 4(sp)
    sw    s0, 0(sp)

    /* Store the old stack pointer in the old thread */
    sw    sp, 0(a0)
```

## Context Switch on the MIPS (2 of 2)

```
/* Get the new stack pointer from the new thread */
lw    sp, 0(a1)
nop           /* delay slot for load */

/* Now, restore the registers */
lw    s0, 0(sp)
lw    s1, 4(sp)
lw    s2, 8(sp)
lw    s3, 12(sp)
lw    s4, 16(sp)
lw    s5, 20(sp)
lw    s6, 24(sp)
lw    s8, 28(sp)      /* a.k.a. frame pointer */
lw    gp, 32(sp)
lw    ra, 36(sp)
nop           /* delay slot for load */

/* and return. */
j    ra
addi sp, sp, 40      /* in delay slot */
.end switchframe_switch
```

# Context Switch on the MIPS

- A context switch is similar to calling a subroutine except *now you are switching from one thread's stack to another.*
- We use the frame pointer (s8 a.k.a. fp) as the base register for local variables.
- The gp (global pointer) is the base register for global variables.
- You save the return address (ra), frame pointer, global pointer, and registers whose values are expected to be preserved.
- Recall the concepts of **pipelining** and **delay slots** from CS251.
- If I'm loading a value into sp (top of slide 14), then I cannot use it in the next instruction because of a **Load-use Hazard**.
- In this case we solve that problem by introducing a nop (no operation) instruction so the use of sp does not occur right after the load.

# Context Switch on the MIPS

- Near the bottom of slide 14 there is a jump register instruction. (In CS241's dialect it was a **jr \$31** instruction). Here it is a **j ra** instruction.
- This is a **Control Hazard**, i.e. we don't know which instruction to fetch next so we set it up so that the next instruction (in the delay slot) always gets executed while fetching the instruction whose address is stored in mgcra.
- So the **addi sp, sp, 40** instruction gets executed next even though it occurs after the **j ra** instruction.

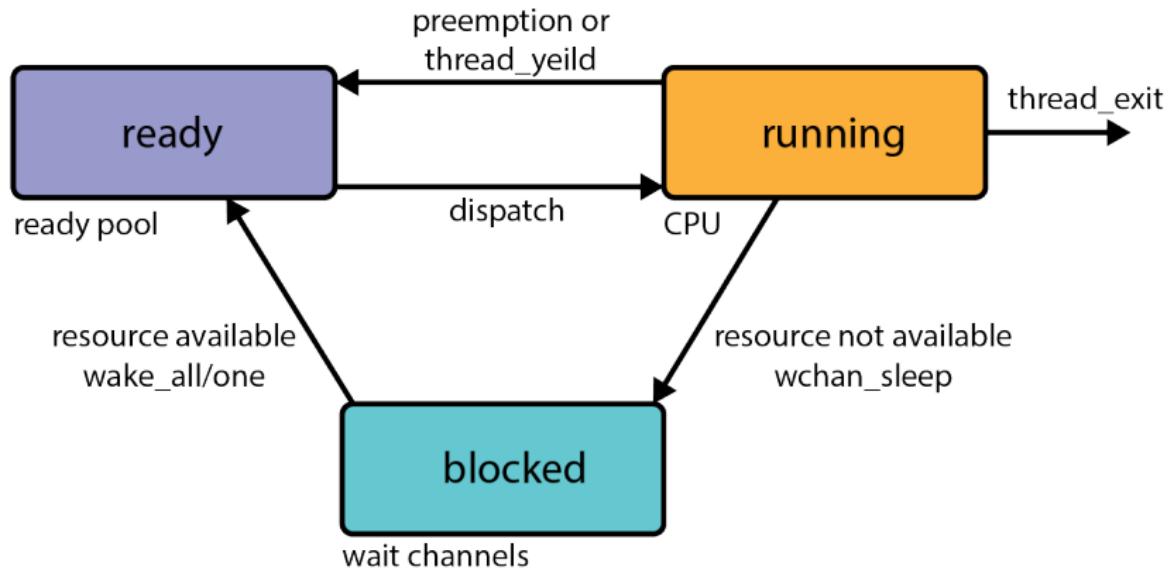
# What Causes Context Switches?

- the running thread calls **thread\_yield**
  - running thread *voluntarily allows other threads to run*
- the running thread calls **thread\_exit**
  - running thread *is terminated* (i.e. has completed its task)
- the running thread **blocks**, via a call to **wchan\_sleep**
  - it is waiting for some resource (such as network access) or for some event to happen
  - more on this later ...
- the running thread is **preempted**
  - running thread *involuntarily stops running* (because the thread scheduler stopped it)

## The OS

... strives to maintain high CPU utilization. Hence, in addition to timesharing, context switches occur whenever a thread ceases to execute instructions.

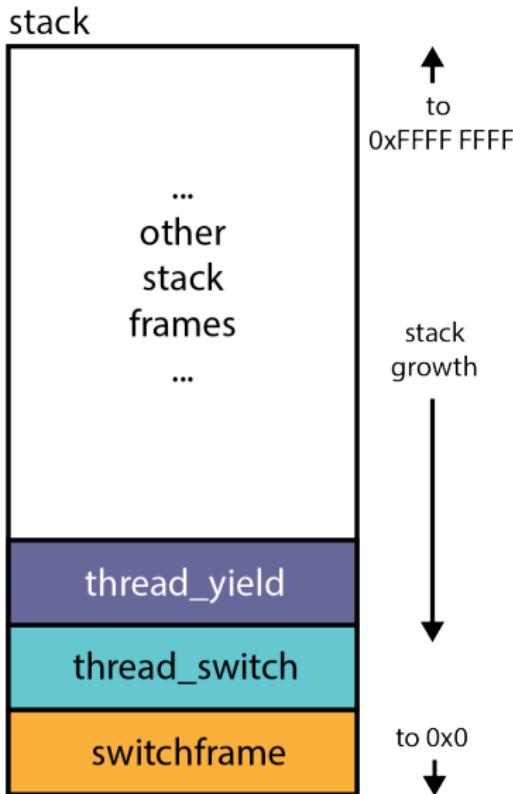
# Thread States



A thread can be in one of three states

1. **running:** currently executing
2. **ready:** ready to execute
3. **blocked:** waiting for something, so not ready to execute.

# OS/161 Thread Stack after Voluntary Context Switch



- program calls `thread_yield`, to yield the CPU
- `thread_yield` calls `thread_switch`, to perform a context switch
- `thread_switch` chooses a new thread, calls `switchframe_switch` to perform low-level context switch

# Timesharing and Preemption

- **timesharing:** concurrency achieved by rapidly switching between threads
  - how rapidly? impose a limit on processor time, the **scheduling quantum**
  - the quantum is an *upper bound on how long a thread can run before it must yield* the CPU
- how do you stop a running thread, that never yields, blocks or exits when the quantum expires?
  - preemption forces a running thread to stop running, so that another thread can have a chance
  - to implement preemption, the thread library must have a means of “getting control” (causing thread library code to be executed) even though the running thread has not called a thread library function
  - this is normally accomplished using **interrupts**

## Review: Interrupts

- an **interrupt** is an event that occurs during the execution of a program
- interrupts are caused by system devices (hardware), e.g., a timer, a disk controller, a network interface card
- when an interrupt occurs, the hardware automatically transfers control to a fixed location in memory (specified by the designer of the CPU)
- at that memory location, the thread library must place a procedure called an **interrupt handler**
- the interrupt handler normally:
  - creates a **trap frame** to record the thread context at the time of the interrupt
  - determines which device caused the interrupt and performs device-specific processing
  - restores the saved thread context from the trap frame and resumes execution of the thread

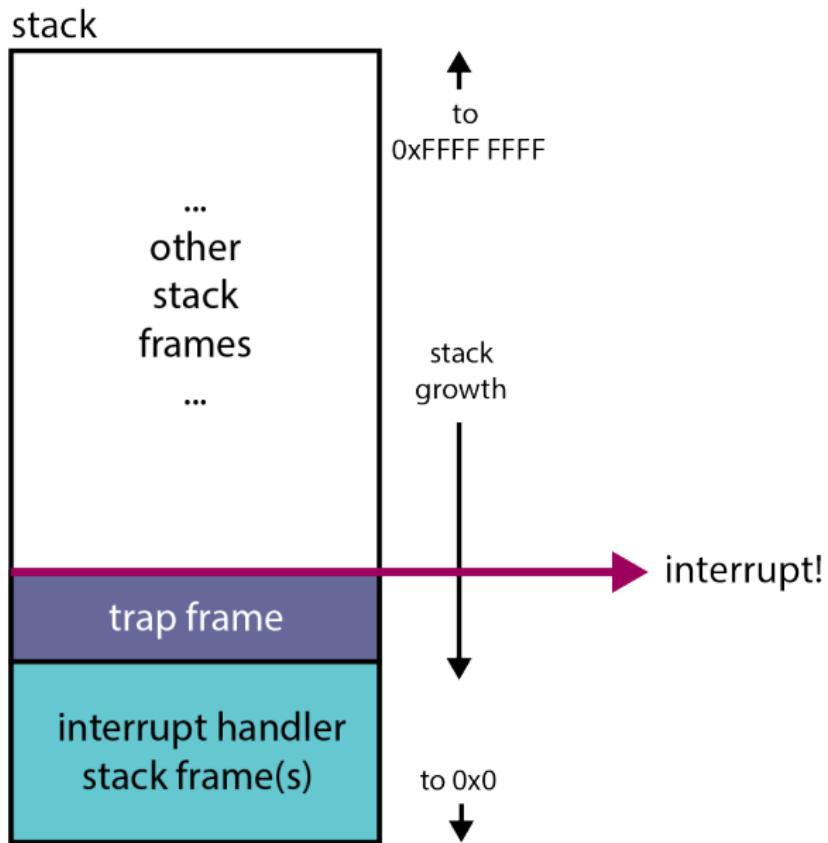
# Voluntary vs. Involuntary Context Switches

- When calling a *subroutine* the registers **ra**, **fp** and any of **s0—s6** that will be modified, are stored in a *stack frame*.
- For a *voluntary context switch* all of **ra**, **fp**, **s0—s6** and **gp** are stored in a *switch frame*.
- For an *involuntary context switch* all the registers are saved (including a few you have never heard of) in a *trap frame*.

Refs:

- `thread_yield()` and `thread_switch()` are in  
`/kern/thread/thread.c`
- `switchframe_switch` is in `/kern/arch/mips/thread/switch.S`
- `trapframe` struct in `/kern/arch/mips/include/trapframe.h`

# OS/161 Thread Stack after in Interrupt

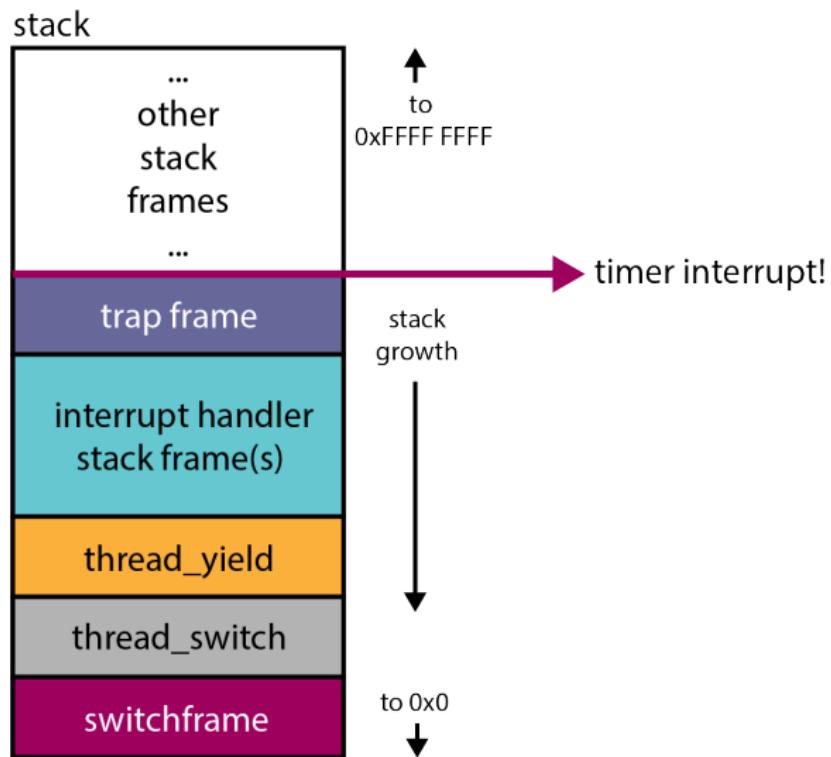


# Preemptive Scheduling

- A preemptive scheduler uses the **scheduling quantum** to impose a time limit on running threads
- Threads may block or yield before their quantum has expired.
- Periodic timer interrupts allow running time to be tracked.
- *If a thread has run too long, the timer interrupt handler preempts the thread by calling `thread_yield`.*
- The preempted thread changes state from running to ready, and it is placed on the *ready queue*.
- Each time a thread goes from ready to running, the runtime starts out at 0. Runtime does not accumulate.

OS/161 threads use *preemptive round-robin scheduling*.

# OS/161 Thread Stack after Preemption

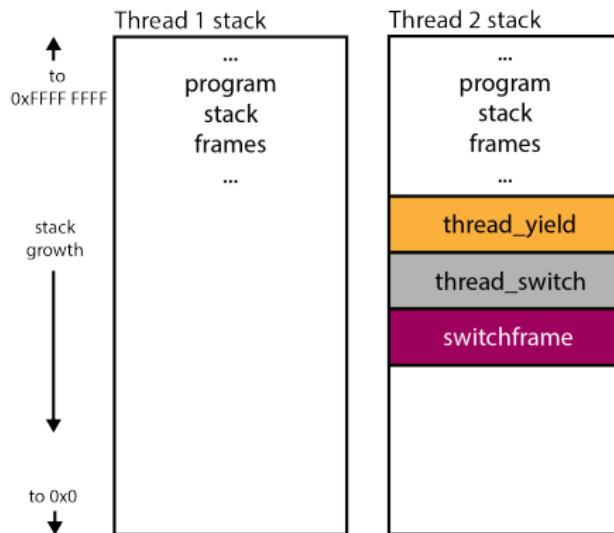


- **Scheduling** means deciding which thread should run next
- Scheduling is implemented by a **scheduler**, which is part of the thread library
- **Preemptive round-robin** scheduling:
  - scheduler maintains a queue of threads, often called the **ready queue**
  - the first thread in the ready queue is the running thread
  - on a context switch the running thread is moved to the end of the ready queue, and new first thread is allowed to run
  - newly created threads are placed at the end of the ready queue
- threads can be migrated to other processors or interrupted by device so the order they run is *nondeterministic*
- more on scheduling later ...

## Two Threads Example

- The follow example considers two threads.
- Thread 1 will get interrupted and Thread 2 will run, i.e an *involuntary context switch* will occur.
- Then Thread 2 will preform a *voluntary context switch*.

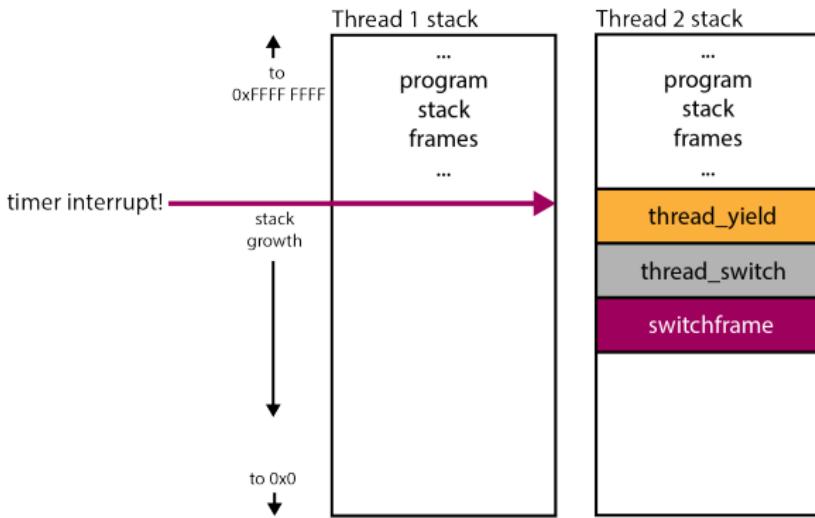
## Two-Thread Example - 1



Thread 1 is **running**.

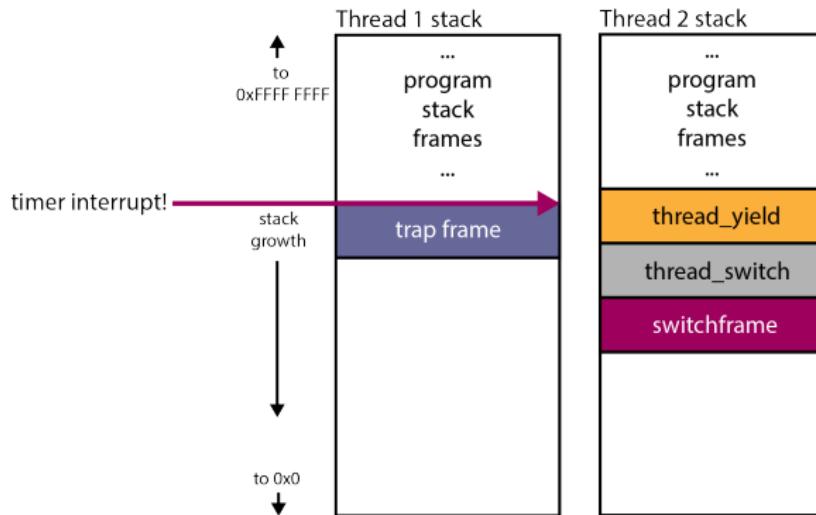
Thread 2 is **ready**, having called `thread_yield` previously.

## Two-Thread Example - 2



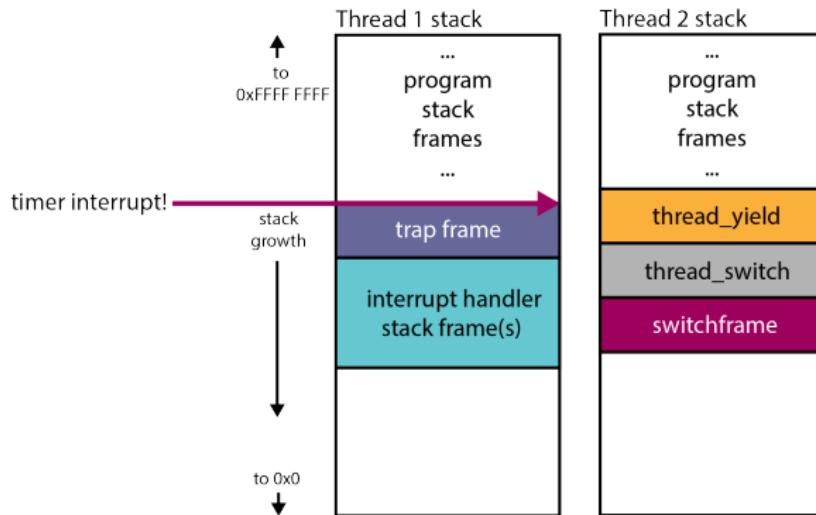
A timer interrupt occurs.

## Two-Thread Example - 3



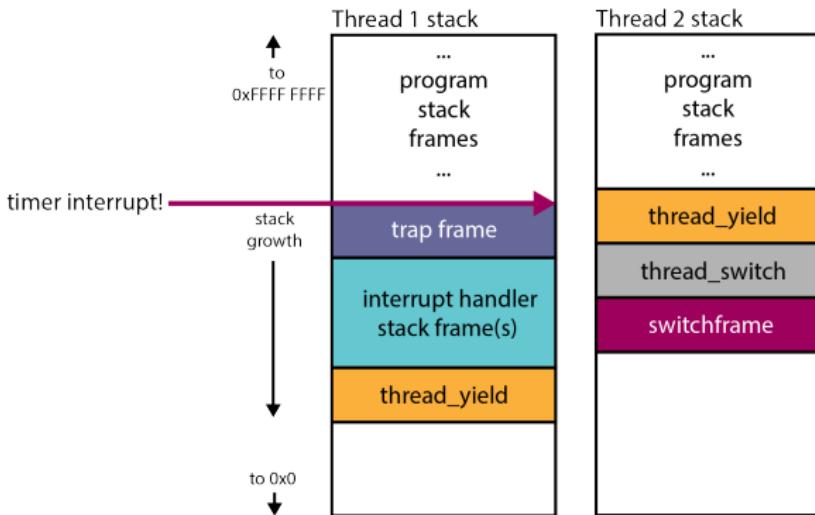
Thread 1 is preempted.  
A trap frame is created to save Thread 1's context.

## Two-Thread Example - 4



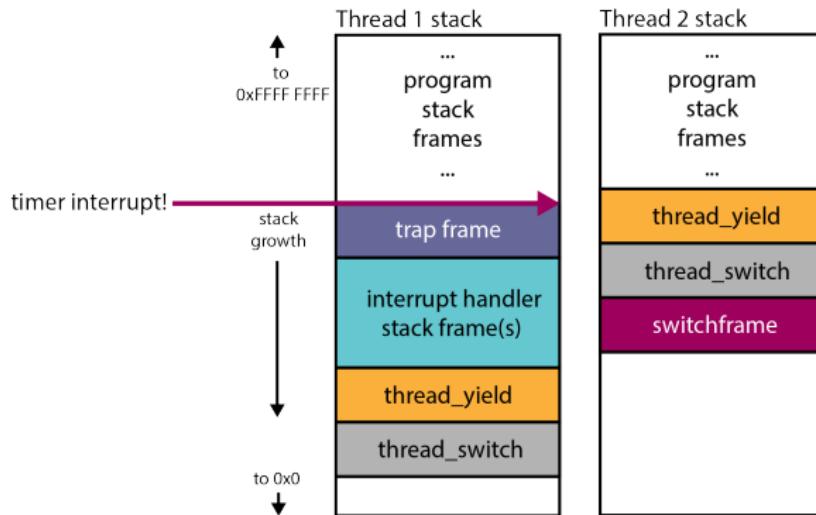
The timer interrupt handler determines what happened and calls the appropriate handler.

## Two-Thread Example - 5



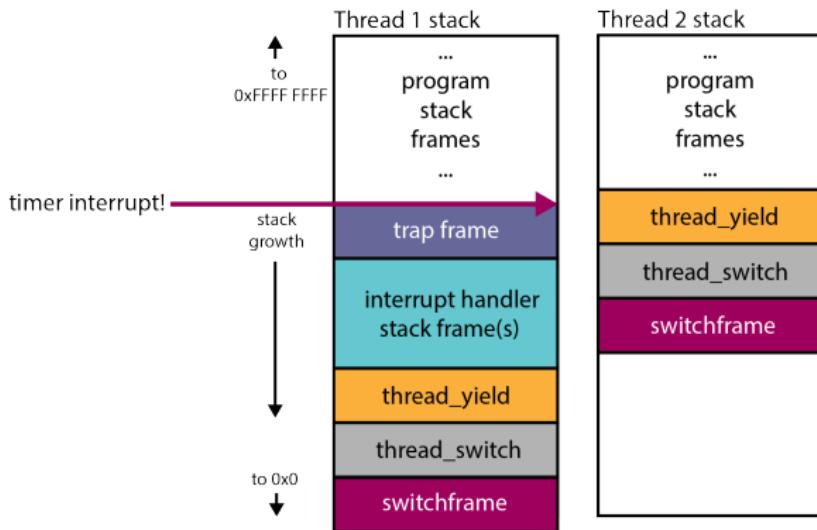
Thread 1 has exceeded its quantum.  
Hence it yields the CPU to another thread, i.e. call `thread_yield`.

## Two-Thread Example - 6



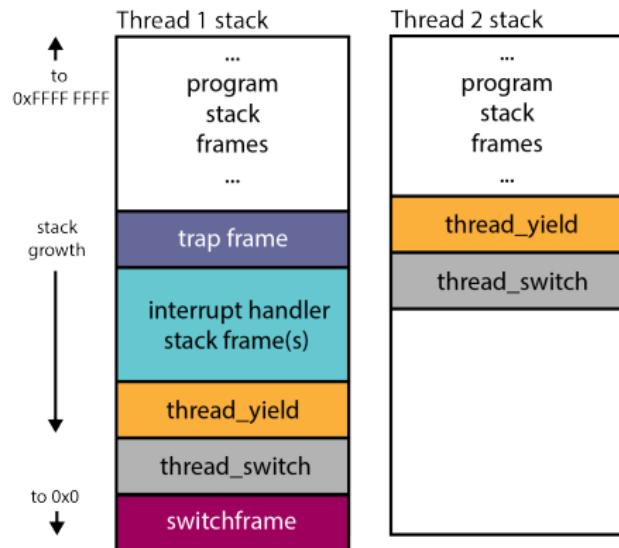
High-level context switch:  
choose new thread, save caller-save registers.

## Two-Thread Example - 7



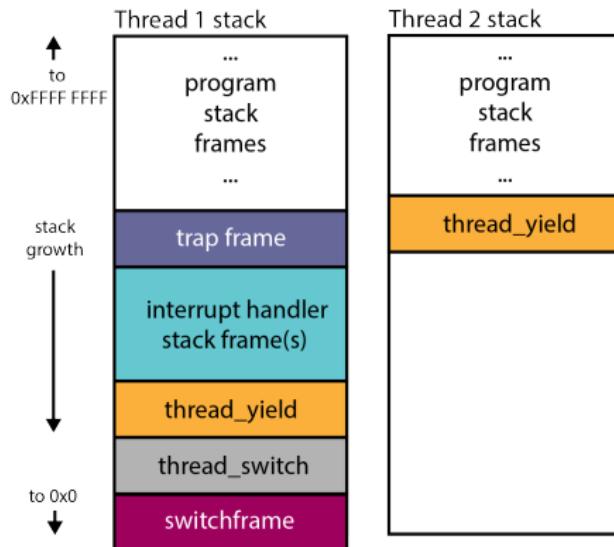
Low-level context switch:  
Save callee-save registers.

## Two-Thread Example - 8



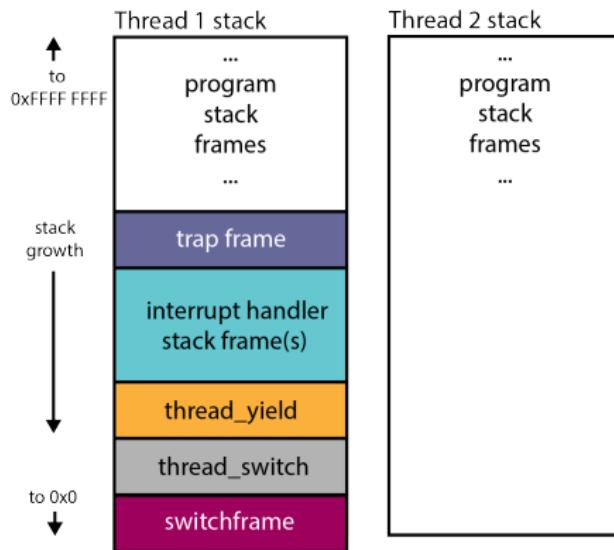
Thread 2 is now **running**, Thread 1 is now **ready**. Thread 2 returns from low-level context switch, restoring callee-save registers.

## Two-Thread Example - 9



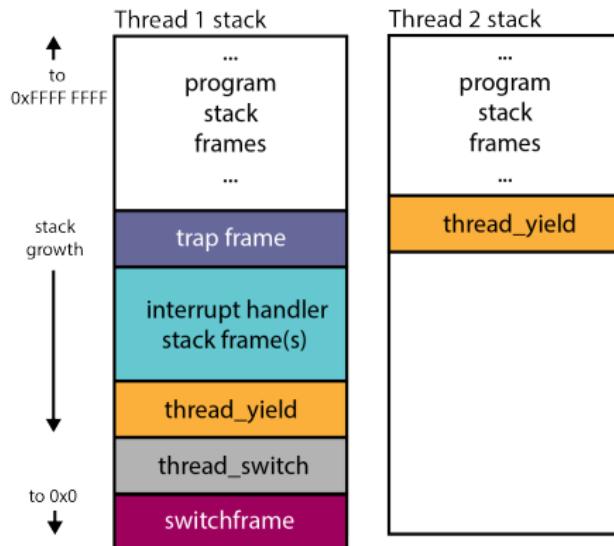
Return from high-level context switch, restoring caller-save registers.

## Two-Thread Example - 10



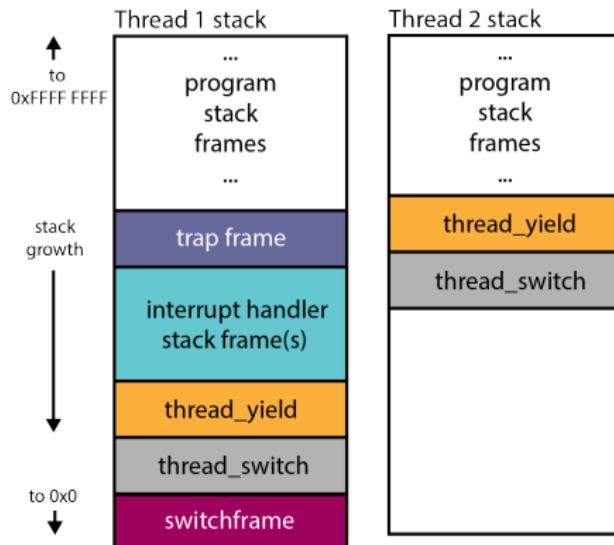
Return from yield. Context is fully restored. Thread 2 is now running its regular program.

## Two-Thread Example - 11



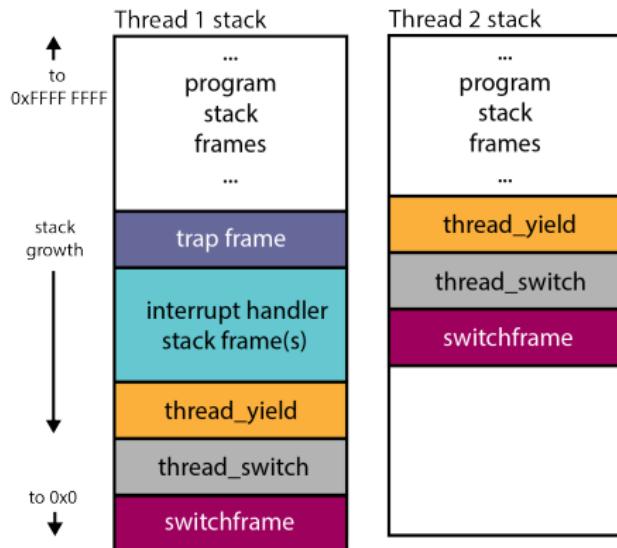
Thread 2 yields.

## Two-Thread Example - 12



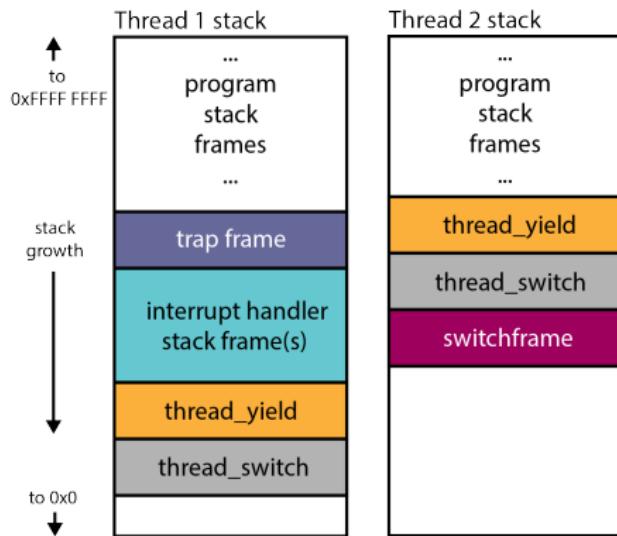
High-level context switch.

## Two-Thread Example - 13



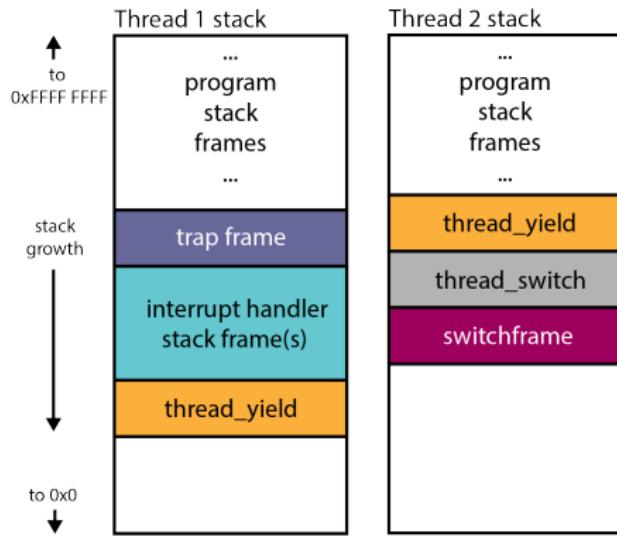
Low-level context switch.

## Two-Thread Example - 14



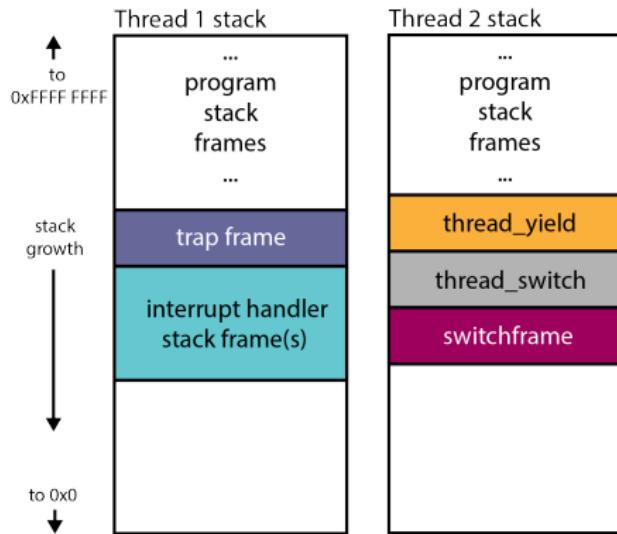
Thread 1 is now **running**. Thread 2 is now **ready**. Return from low-level context switch.

## Two-Thread Example - 15



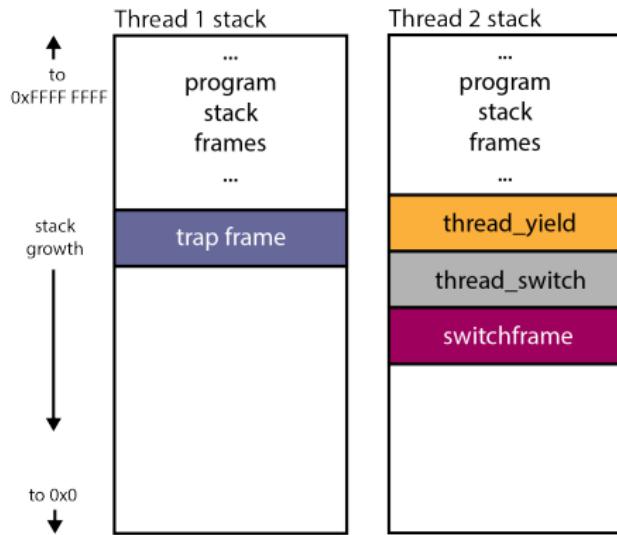
Return from high-level context switch.

## Two-Thread Example - 16



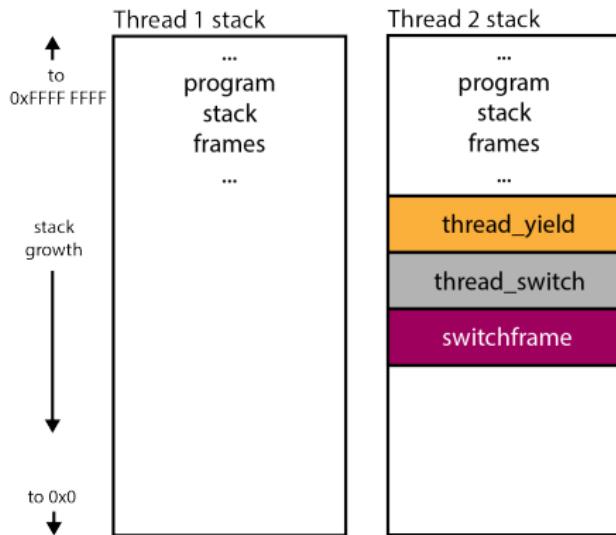
Return from yield.

## Two-Thread Example - 17



Return from interrupt handling functions.

## Two-Thread Example - 18



Restore thread 1's context (stored in the trap frame), return to regular program.

# Summary

1. A **thread** is a sequence of instructions. [1]
2. **Concurrency** (or concurrent execution) is when multiple programs or sequences of instructions make progress, i.e. *run or appearing to run*, at the same time. [2]
  - a) **Parallelism** is when multiple programs or sequences of instructions can *run* at the same time (because there are multiple processors or multiple cores). [2]
  - b) **Timesharing** is when multiple threads *take turns* on the same hardware by rapidly switching between them. [11]
3. Threads share access to the program's *code, global variables and heap* but each thread has its own *stack* and register values. [4]
4. OS/161 uses **thread\_fork** to create a new thread. [5]
5. OS/161 distinguishes between *caller-save* registers and *callee-save* registers. [8]

6. A **context switch** is when a processor/core switches from executing one thread to executing another. [12]
7. **Thread context** is all the information (i.e. register values) needed to resume executing a thread after it has been suspended. [12–14]
8. Thread context is stored in a **switch frame**. [12–14]
9. **Preemption** is when a running thread involuntarily stops running because the thread scheduler stopped it. [15]
10. When a thread is interrupted *all* the register values are stored in a **trap frame**. [19]
11. A thread can be one of the following states **running**, **ready**, **blocked**. [16]
12. Timesharing can be implemented using a **scheduling quantum**, a timer and interrupts. [18–21]

13. In a **voluntary context switch** the thread calls **thread\_yield** which calls **thread\_switch**, which stores a **switch frame** on the stack. [33–35]
14. In an **involuntary context switch** the thread is interrupted (resulting in a trap frame) followed by a stack frame for an interrupt handler, which calls **thread\_yield** which calls **thread\_switch**, which stores thread context in a **switch frame** on the stack. [23–29]

# Synchronization

**key concepts:** critical sections, mutual exclusion, test-and-set, spinlocks, blocking and blocking locks, semaphores, condition variables, deadlocks

Lesley Istead, Kevin Lanctot

David R. Cheriton School of Computer Science  
University of Waterloo

Winter 2019

# Thread Synchronization

## The Traffic Analogy

- On a divided highway, there is *little danger* of an accident caused from the cars going other direction (concurrent activities) because there is *no interaction*.
- At an intersection, *there is a greater danger* of an accident because cars going in different directions *can interact*.

## The Bowl of Fruit Analogy (From the course text, Ch 25 Dialog)

- We want to avoid the two extremes, i.e. all grab at once or each grabs one at a time.
- The goal is *efficiency and correctness*.

## Safety and Liveness Properties

- Safety: nothing bad will happen (i.e. correctness)
- Liveness: something good will happen (similar to efficiency).

# Thread Synchronization

- *Concurrent threads interact with each other* in a variety of ways:
  - All threads in a concurrent program *share access* to the program's global variables and heap.
  - Threads share access, through the operating system, to system devices, such as the hard drive, the display, etc. (More on this topic later in the course.)
- Returning to our traffic analogy, *when there is interaction* (e.g. an intersection) *there is more danger* of an accident than when there is no interaction.
- The danger can be *minimized by coordinating the interaction* (e.g. traffic lights).

# Thread Synchronization

- A common synchronization problem is to enforce **mutual exclusion**, which means ensuring that only one thread at a time accesses a shared resource.
- The part of a concurrent program in which a shared object is accessed (e.g. the intersection) is called a **critical section**.
- This coordination of access to a shared resource is called **synchronization**.
- *Key Question:* What happens if several threads try to access the same global variable or heap object at the same time?

# Synchronization

- Before we discuss how to synchronize and enforce mutual exclusion we are first going to remove the traffic lights at the intersection, so to speak, and see what happens.
- In the code on the next few slides, *the shared resource* that two different threads will be accessing is the variable `total`.
- One thread, running `add()`, will be incrementing `total` and another thread, running `sub()`, will be decrementing it.
- The code is available as `add_sub.c` in Lecture Supplements section of Learn.
- Compile it in the `linux.student.cs` environment with the following command: `gcc -pthread add_sub.c`

## Critical Section Example

```
/* Note the use of volatile; revisit later */
int volatile total = 0;

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

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

If one thread executes `add` and another executes `sub` what is the value of `total` when they have finished?

## Critical Section Example: assembly language detail

```
/* Note the use of volatile */
int volatile total = 0;

void add() {
    loadaddr R8 total
    for (i=0; i<N; i++) {
        lw R9 0(R8)
        add R9 1
        sw R9 0(R8)
    }
}

void sub() {
    loadaddr R10 total
    for (i=0; i<N; i++) {
        lw R11 0(R10)
        sub R11 1
        sw R11 0(R10)
    }
}
```

In order to understand what is going on, consider the code  
*at the (pseudo) assembly language level.*

## Critical Section Example: Trace 1

;; Thread 1

```
loadaddr R8 total  
lw R9 0(R8) ; R9=0  
add R9 1      ; R9=1  
sw R9 0(R8) ; total=1
```

Thread 2

----- Preemption ----->

```
loadaddr R10 total  
lw R11 0(R10) ; R11=1  
sub R11 1      ; R11=0  
sw R11 0(R10) ; total=0
```

One possible order of execution is sequential.  
Here the final value of **total** is 0.

## Critical Section Example: Trace 2

```
;; Thread 1           Thread 2
loadaddr R8 total
lw R9 0(R8) ; R9=0
add R9 1     ; R9=1
----- Preemption ----->
                                loadaddr R10 total
                                lw R11 0(R10) ; R11=0
                                sub R11 1      ; R11=-1
                                sw R11 0(R10) ; total=-1
<----- Preemption ----->
sw R9 0(R8) ; total=1
```

Another possible order of execution is interleaved.  
Here the final value of **total** is 1.

## Critical Section Example: Trace 3

<pre>;; Thread 1 loadaddr R8 total lw R9 0(R8) ; R9=0 add R9 1     ; R9=1 sw R9 0(R8) ; total=1</pre>	<pre>Thread 2 loadaddr R10 total lw R11 0(R10) ; R11=0 sub R11 1      ; R11=-1 sw R11 0(R10) ; total=-1</pre>
---	---

Yet another possible order of execution, this time on two processors. The final value of **total** is -1.

## Comments on the Critical Section Examples

Use of `volatile` is not enough.

- Without the keyword `volatile` the compiler could optimize the code. Using `volatile` forces the compiler to load and store the value on every use (rather than storing the value in a register).
- Even with the use of `volatile`, the programmer does not know when preemption will occur or if the threads will timeshare or run in parallel.
- *Key Point:* We require that concurrent programs will run correctly regardless of which order the threads are executed.

# Race Condition

- A **race condition** is when the program result depends on the order of execution.
- The previous program `add_sub` had a race condition.
- Race conditions occur when multiple threads are reading and writing the same memory at the same time.
- Sources of race conditions:
  1. implementation
  2. ...
  3. ...

... more sources of race conditions to come!

## Is there a race condition?

The next few slides show two functions involved in the implementation of a singly linked list of integers.

- `list *lp` is a pointer to the list.
- It points to a structure with
  - a pointer to the first item of the list, `lp->first`,
  - a pointer to the last item of the list, `lp->last`,
  - a count of the number of elements in the list, `lp->num_in_list`.
- If the list is empty then `lp->first` and `lp->last` will be `NULL`.
- Each element of the list is a structure with two fields
  - an `item`, which is an integer,
  - a pointer to the next element in the list `next`.

# Is there a race condition?

## Example 1

```
// Remove the first element from the list *lp
// and return the integer value stored there.
1. int list_remove_front(list *lp) {
2.     int num;                                // value stored in list
3.     list_element *element;                  // element to remove
4.     assert(!is_empty(lp));
5.     element = lp->first;
6.     num = lp->first->item;
7.     if (lp->first == lp->last) { // only item in list
8.         lp->first = lp->last = NULL;
9.     } else {                               // many items in list
10.        lp->first = element->next;
11.    }
12.    lp->num_in_list--;
13.    free(element);
14.    return num;
15. }
```

## Is there a race condition?

Imagine if two threads, **T1** and **T2**, call `list_remove_front` on a list with two (or more) items in it.

- Say **T1** got preempted after line 6 (i.e. it has a pointer to the first element but has not yet removed it from the list).
- Then **T2** executes the entire function (i.e. removes the first element from the list)
- Finally **T1** continued from line 7 and completed the function.
- Then they both would have a copy of the same element (the first element in the list).
- Line 12 would be executed twice but only one element would have been removed so `lp->num_in_list` would be one less than the actual size of the list.
- The function `free` on line 13 would be called twice on the same pointer value which would likely cause a crash.

## Is there a race condition?

Imagine if two threads **T1** and **T2** call `list_remove_front` on a list with just one item in it.

- Say **T1** got preempted after executing lines 1–4 (i.e. it has verified that the list is not empty).
- Then **T2** successfully completed the entire function (removing the only item from the list).
- Finally **T1** continued from line 5 to the end.
- Then **T2** would have removed the only element from the list and `lp->first` would now be **NULL**.
- So when **T1** would start running again on line 5 it would crash on line 6 because it would be dereferencing a **NULL** pointer.

There are many other scenarios that would cause race conditions.

# Is there a race condition?

## Example 2

```
// Add new_time to the end of the list *lp.  
// The list should not contain duplicate values.  
1. void list_append(list *lp, int new_item) {  
2.     list_element *element = malloc(sizeof(list_element));  
3.     element->item = new_item  
4.     assert(!is_in_list(lp, new_item));  
5.     if (is_empty(lp)) {  
6.         lp->first = element;  
7.         lp->last = element;  
8.     } else {  
9.         lp->last->next = element;  
10.        lp->last = element;  
11.    }  
12.    lp->num_in_list++;  
13. }
```

Is there a scenario where the same item can be added twice?

## Is there a race condition?

Imagine if two threads, **T1** and **T2**, call `list.append` with the same value for `new_item`, say 350.

- Say **T1** got preempted after executing line 4 (where it has just verified that it is adding a new item).
- Then **T2** completes the entire function adding 350 to the list.
- Then **T1** continues executing at line 5 and adds 350 to the list.

Now the value 350 is in the list twice.

# Tips for identifying race conditions

- To find the critical sections...
  - Inspect each variable and ask *is it possible for multiple threads to read and write it concurrently?*
  - Constants and memory that all threads *only read* do not cause race conditions.
- The course text defines a **critical section** as

*A critical section is a piece of code that accesses a shared variable (or more generally, a shared resource) and must not be concurrently executed by more than one thread.*
- **Key Question:** After identifying the critical sections, how can you prevent race conditions?

# Enforcing Mutual Exclusion With Locks

```
int volatile total = 0;

void add() {
    int i;
    for (i=0; i<N; i++) {
        ----- start mutual exclusion -----
        total++;
        ----- stop mutual exclusion -----
    }
}

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

Need a method to provide mutual exclusion a.k.a. **mutex**, i.e. ensure that only one thread can access this region at a time.

# Enforcing Mutual Exclusion With Locks

```
int volatile total = 0;
bool volatile total_lock = false; // false means unlocked
                                // true means locked

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

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

- `Acquire/Release` must ensure that only one thread can hold the lock at a time, even if both attempt to acquire it simultaneously.
- If a thread cannot acquire the lock immediately, it must wait until the lock becomes available.

## Lock Acquire and Release

```
Acquire(bool *lock) {  
    while (*lock == true) ; // spin until the lock is free  
    *lock = true;          // grab the lock  
}  
  
Release(bool *lock) {  
    *lock = false;         // give up the lock  
}
```

Does this work?

## Lock Acquire and Release

```
Acquire(bool *lock) {  
    while (*lock == true) ; // spin until the lock is free  
    *lock = true;          // grab the lock  
}  
  
Release(bool *lock) {  
    *lock = false;         // give up the lock  
}
```

It does not! Why?  
How could you fix it?

# Hardware-Specific Synchronization Instructions

- *One Solution:* H/W provides an **atomic** (i.e. indivisible or uninterruptable) **test-and-set** operation used to implement synchronization primitives such as locks.
- Example: the x86-64 (and x86) **xchg** instruction:

```
xchg src, addr
```

where **src** is a register and **addr** is a memory address.

- This instruction swaps the value stored in register **src** with the value stored at the address **addr**.
- It logically behaves like the following function, executed atomically.

```
xchg(value,addr) {  
    old = *addr;  
    *addr = value;  
    return(old);  
}
```

## x86 - Lock Acquire and Release with xchg

```
Acquire(bool *lock) {  
    while (xchg(true,lock) == true) {};    // spin  
}  
  
Release(bool *lock) {  
    *lock = false;                      // give up lock  
}
```

- Again, here `true` means locked and `false` means unlocked.
- If `xchg` returns `true`, then the lock was already set and you have not changed its value  $\Rightarrow$  you continue to loop.
- If `xchg` returns `false`, then the lock was free and you have changed its value to `true`  $\Rightarrow$  you have now acquired the lock.
- This construct is known as a **spin lock**, since a thread busy-waits (loops or spins) in `Acquire` until the lock is free.

# ARM Synchronization Instructions

Rather than provide a single instruction, ARM *provides a pair of instructions* namely the *exclusive* load in register (**LDREX**) and store from register (**STREX**), which are meant to be used together.

- **LDREX** loads a value from address *addr*
- **STREX** will attempt to store a value at address *addr*
  - it will fail to store the value if address *addr* was touched between the execution of the **LDREX** and **STREX** instructions.
  - **STREX** will report if it has been successful or not
  - **STREX** may fail even if the distance between **LDREX** and **STREX** is small, but should succeed after a few attempts.
- It is recommended that these instructions are placed close together (i.e. within 128 bits).

## Lock Acquire with LDREX and STREX

Logically, these two instructions behave as follows.

```
ARMTestAndSet(addr, new_val) {  
    old_val = LDREX addr          // load old value  
    status = STREX new_val, addr   // try store new value  
    if (status == SUCCEED) return old_val  
    return TRUE  
}  
  
Acquire(bool *lock) {           // spin until hold lock  
    while( ARMTestAndSet(lock, true) == true ) {};  
}
```

- **ARMTestAndSet:** **if** the lock is already held **or** **STREX** fails  
**then** return **true** so that we keep trying to acquire the lock.
- **ARMTestAndSet:** **if** the lock is available **and** **STREX** succeeds **then**  
return **false**

# MIPS Synchronization Instructions

Similar to ARM, two instructions are used, namely `ll` and `sc`.

- `ll` (load linked): load value at address *addr*.
- `sc` (store conditional): store new value at *addr* if the value at *addr* has not changed since the instruction `ll` was executed.
- `sc` returns SUCCESS if the value stored at the address has not changed since `ll`.
- The value stored at the address can be any 32-bit value.
- `sc` *does not check what that value at the address is. It only checks if it has changed.*
- So there can be success or failure.
- If there is success, the lock can be
  - acquired or
  - not acquired (i.e. it is currently held by another thread).

## Lock Acquire with ll and sc

```
MIPSTestAndSet(addr, new_val) {
    old_val = ll addr           // load old value
    status = sc addr, new_val   // store conditionally
    if ( status == SUCCEED ) return old_val
    return true
}

Acquire(bool *lock) {           // spin until hold lock
    while( MIPSTestAndSet(lock, true) == true ) {};
}
```

Lock value ...

at ll	at sc	after sc	sc returns	Lock State
false	false	true	succeed	own lock
false	true	true	fail	keep spinning, no lock
true	true	true	succeed	keep spinning, no lock
true	false	false	fail	keep spinning, no lock

# Spinlocks in OS/161

- 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. it *uses the CPU while they wait* for the lock.
- In OS/161, spinlocks are already defined.
- Used by OS/161
  - when needing to update a shared variable or data structure.
  - to implement other synchronization primitives: i.e. locks, semaphores and condition variables (more on these later).
- Typically *interrupts are disabled while the spinlock is being held* and the thread holding the lock will only execute a few instructions before releasing the spinlock. ⇒ Therefore any other threads waiting for the spinlock will only wait a short time.
- Spinlocks are efficient for short waiting times (a few instructions).

# Spinlocks in OS/161

```
struct spinlock {  
    volatile spinlock_data_t lk_lock;  
    struct cpu *lk_holder;  
};  
  
void spinlock_init(struct spinlock *lk}  
void spinlock_acquire(struct spinlock *lk);  
void spinlock_release(struct spinlock *lk);
```

`spinlock_acquire` calls `spinlock_data_testandset` in a loop until the lock is acquired.

- Ref: /kern/include/spinlock.h
- Ref: /kern/arch/mips/include/spinlock.h

# Using Load-Linked / Store-Conditional

- MIPs supports Load-Linked/Stored-Conditional (LL/SC) but not test-and-set.
- *OS/161 uses LL/SC to implement test-and-set.*
- It loads the value in `sd` into `x`, and uses `y` to store 1 (i.e. locked).
- After the Store-Conditional, `y` contains 1 if the store succeeded, 0 if it failed.
- On failure, return 1 to simulate that the spin lock was already held.
- On success, return the old value of the lock.
  - If old value was 0  $\Rightarrow$  the lock has been acquired.
  - If old value was 1  $\Rightarrow$  the lock still held by another thread.

# Spinlocks in OS/161

```
/* return value 0 indicates lock was acquired */
spinlock_data_testandset(volatile spinlock_data_t *sd)
{
    spinlock_data_t x,y;
    y = 1;                      // value to store (i.e. locked)
    __asm volatile(              // begin assembly language
        ".set push;"           // save assembler mode
        ".set mips32;"          // allow MIPS32 instructions
        ".set volatile;"         // avoid unwanted optimization
        "ll %0, 0(%2);"         // get value of lock x = *sd
        "sc %1, 0(%2);"         // *sd = y; y = success?
        ".set pop"               // restore assembler mode
        : "=r" (x), "+r" (y) : "r" (sd));
    if (y == 0) {return 1;} // if unsuccessful, still locked
    return x;                // if success return lock value
}
```

# Inline Extended Assembler Instructions

The previous instructions are in the following format:

```
__asm volatile(  
    "assembly language instructions ..." : outputs : inputs ;  
}
```

- *This format matches up the variables in C to the inputs and outputs of the assembly language code.*
- The outputs are x (%0) and y (%1) and the input is sd (%2).
- Switches describe how a value is handled:
  - “=r” means write only, stored in a register,
  - “+r” means read and write, stored in a register,
  - “r” means input, stored in a register.
- With C11 and C++11, these languages now support atomic operations, eliminating the need for this step.

- In addition to spinlocks, OS/161 also has **locks**.
- Like spinlocks, locks are used to enforce mutual exclusion, i.e. they are a type of mutex.

```
struct lock *mylock = lock_create("LockName");
lock_acquire(mylock);
    // critical section
lock_release(mylock);
```

- *spinlocks spin whereas locks block*: i.e. a thread that calls
  - `spinlock_acquire` spins until the lock can be acquired
  - `lock_acquire` blocks until the lock can be acquired
- *Locks can be used to protect larger critical sections without being a burden on the CPU.*
- Locks have owners.

# Spin Locks and Locks

## Spin Locks and Locks have Similar Semantics

- must be initialized or created before using them
- *only one thread can hold the lock* at a time
- a thread attempts to hold the lock by calling `acquire`
- it would then perform an operation where it need exclusive access to a variable, data structure, or device
- when the thread is finished that operation, it would `release` the lock
- *key constraint*: the thread that releases a lock must be the same thread that most recently acquired it
- a thread can also ask `do-i-hold` a lock
- typically you would declare one lock for each variable, data structure, or device you want exclusive access to

## Spin Locks

- void spinlock\_init(struct spinlock \*lk)
- void spinlock\_acquire(struct spinlock \*lk)
- void spinlock\_release(struct spinlock \*lk)
- bool spinlock\_do\_i\_hold(struct spinlock \*lk)
- void spinlock\_cleanup(struct spinlock \*lk)

## Locks

- struct lock \*lock\_create(const char \*name)
- void lock\_acquire(struct lock \*lk)
- void lock\_release(struct lock \*lk)
- bool lock\_do\_i\_hold(struct lock \*lk)
- void lock\_destroy(struct lock \*lk)

# Thread Blocking

- Sometimes a thread will need to wait for something, e.g.:
  - wait for a lock to be released by another thread
  - wait for data from a (relatively) slow device
  - wait for input from a keyboard
  - wait for busy device to become idle
- *When a thread blocks, it stops running*, i.e. it stops using the processor:
  - the scheduler chooses a new thread to run
  - a context switch from the blocking thread to the new thread occurs,
  - the blocking thread is queued in a *wait queue* (not on the ready list)
- Eventually, a blocked thread is signaled and awakened by another thread.

# Wait Channels in OS/161

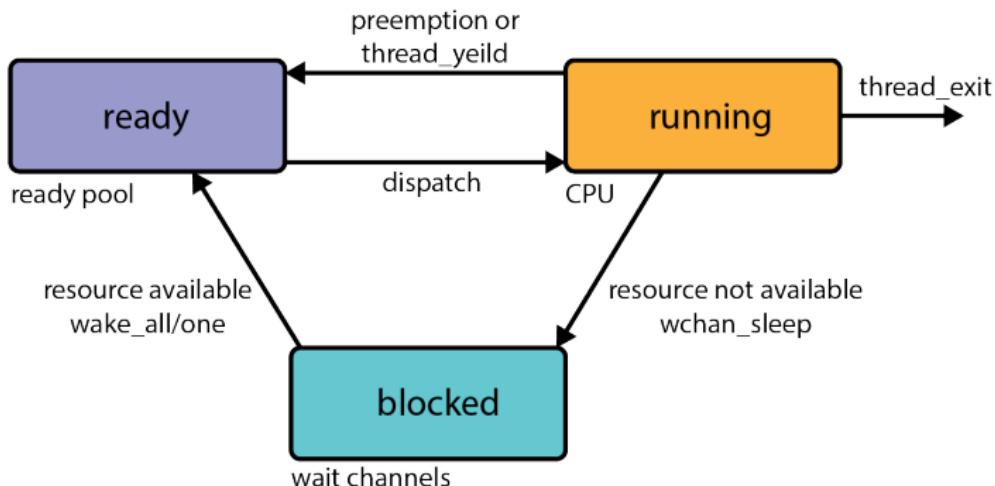
Wait channels are used to implement thread blocking in OS/161

- `void wchan_sleep(struct wchan *wc);`
  - blocks calling thread on wait channel `wc`
  - causes a context switch, like `thread_yield`
- `void wchan_wakeall(struct wchan *wc);`
  - unblock all threads sleeping on wait channel `wc`
- `void wchan_wakeone(struct wchan *wc);`
  - unblock one thread sleeping on wait channel `wc`
- `void wchan_lock(struct wchan *wc);`
  - prevent operations on wait channel `wc`
  - more on this later!

There can be many different wait channels, holding threads that are blocked for different reasons.

In OS/161 wait channels are implemented with queues.

# Thread States, Revisited



**running:** currently executing

**ready:** ready to execute

**blocked:** waiting for something, not ready execute

## Key Point:

- ready threads are queued on the ready queue,
- blocked threads are queued on wait channels

# Semaphores

- A **semaphore** is a synchronization primitive that can be used to enforce mutual exclusion requirements. It can also be used to solve other kinds of synchronization problems.
- A semaphore is an object that *has an integer value* and that supports two operations:
  - **P**: *if the semaphore value is greater than 0, decrement it.*  
Otherwise, wait until the value is greater than 0 and then decrement it.
  - **V**: *increment the value* of the semaphore
- Can think of the operations as
  - **P**: *procure*, down, wait, acquire, conditional decrement
  - **V**: *vacate*, up, signal, release, increment
- By definition, the **P** and **V** operations of a semaphore are atomic.

# Types of Semaphores

There are two types of semaphores:

1. **binary semaphore:** a semaphore with a single resource; behaves like a lock, but does not keep track of ownership
2. **counting semaphore:** a semaphore with an arbitrary number of resources

*Differences between a lock and a semaphore.*

- **V** does not have to follow **P**.
- A semaphore can start with 0 resources.
- Then **V** is called incrementing the semaphore to 1.
- This sequence of operations forces a thread to wait until resources are produced before continuing.

# Mutual Exclusion Using a Semaphore

```
volatile int total = 0;
struct semaphore *total_sem;
total_sem = sem_create("total mutex",1); // initial value is 1

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

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

The first thread to *procure* the semaphore, i.e. call `P(sem)`, will block the other thread from accessing the critical region until the first thread *vacates* it by calling `V(sem)`.

# Producer/Consumer Synchronization with Bounded Buffer

In some situations we can have two types of threads

1. **producers:** threads that (create and) *add* items to a buffer
  2. **consumers:** thread that *remove* (and process) items from the buffer
- *Constraint 1:* we want to ensure that consumers do not consume if the buffer is empty  $\Rightarrow$  instead they must wait until the buffer has something in it
  - *Constraint 2:* if the buffer has a finite capacity ( $N$ ), we need to ensure that producers must wait if the buffer is full
  - This requires synchronization between consumers and producers.
  - Semaphores can provide the necessary synchronization.
  - e.g. packets of video data arrive in a Wi-Fi buffer (producer) and the software that displays the video (consumer)

# Bounded Buffer Producer/Consumer with Semaphores

```
struct semaphore *Items,*Spaces;  
Items = sem_create("Buffer Items", 0); // initially = 0  
Spaces = sem_create("Buffer Spaces", N); // initially = N
```

Producer's Pseudo-code:

```
P(Spaces); // block if there is no space in buffer  
add item to the buffer  
V(Items); // increase the number of items available
```

Consumer's Pseudo-code:

```
P(Items); // block if there are no items to consume  
remove item from the buffer  
V(Spaces); // increase the amount of buffer space available
```

There is still a race condition in this code.  
What is it? How can you fix it?

# Bounded Buffer Producer/Consumer with Semaphores

## Discussion:

- Consumers will block i.e. wait for items to be produced.
- Producers will block i.e. wait for spaces to be available.
- But the two semaphores do not prevent the producers and consumers from *both accessing* the bounded buffer data structure at the *same* time, causing a race condition.
- Solution:
  - A third synchronization primitive is required to protect the buffer
  - e.g. a lock or binary semaphore is sufficient

# Semaphore Implementation

```
struct semaphore {  
    char *sem_name;          // for debug purposes  
    struct wchan *sem_wchan; // where threads wait  
    struct spinlock sem_lock; // to synch access to this struct  
    volatile int sem_count; // value of the semaphore  
};
```

*Each semaphore has* its own

- name to identify it for debugging purposes,
- wait channel for threads waiting for this semaphore,
- spinlock to synchronize updates to `sem_count` and `sem_wchan`,
- value for the semaphore, i.e. `sem_count`

But the semaphore *does not have* an “owner.”

Ref: kern/include/synch.h

# Semaphore Implementation

```
1. P(struct semaphore * sem) {
2.     spinlock_acquire(&sem->sem_lock);
3.     while (sem->sem_count == 0)           // prep to go to sleep
4.         wchan_lock(sem->sem_wchan);
5.         spinlock_release(&sem->sem_lock);
6.         wchan_sleep(sem->sem_wchan);    // go to sleep here
7.         spinlock_acquire(&sem->sem_lock); // wake up here
8.     }
9.     sem->sem_count--;                  // claim one resource
10.    spinlock_release(&sem->sem_lock);
11. }
```

Ref: kern/thread/synch.c

# Semaphore Implementation

- 2 You need to acquire a spinlock to access the semaphore structure.
- 3 If there are no resources available, so go to sleep.
- 4 – 6: Only need to modify the wchan, so lock it, release the semaphore spinlock and then go to sleep.
- 6 `wchan_sleep` will release the wchan spinlock.
- 7 Another thread may have gotten to the spin lock between the time that you woke up and tried to acquire the lock.
- 9 If you've broken out of the while-loop, you have the lock and `sem_count` is greater than 0, so claim a resource.

# Semaphore Implementation

```
V(struct semaphore * sem) {  
    spinlock_acquire(&sem->sem_lock);  
    sem->count ++;  
    wchan_wakeone(sem->sem_wchan);  
    spinlock_release(&sem->sem_lock);  
}
```

Vacating is simple.

- Acquire the lock, increase the count, and wake up the first thread (if any) sleeping on the semaphore's wait channel.

Ref: kern/thread/synch.c

# Designing Multithreaded Code

When designing or debugging multithreaded code, consider what could possibly go wrong if your code was interrupted at each step of the procedure. E.g.

```
int aFunc(int i) {  
    // What if aFunc was interrupted here?  
    f1(i);  
    // What if aFunc was interrupted here?  
    f2(i);  
    // What if aFunc was interrupted here?  
    f3(i);  
    // What if aFunc was interrupted here?  
    return 0;  
}
```

For example, the next seven slides are an example of why the semaphore function `P` must call `wchan_lock` before `spinlock_release`.

# Incorrect Semaphore Implementation Trace

Suppose `spinlock_release` preceeds `wchan_lock`.

Consider the case when `sem_count=0`.

Thread 1	Thread 2
calls <code>P()</code>	...
...	...
<code>sem_count==0</code>	
<code>spinlock_release</code>	
<i>context switch →</i>	

The semaphore has no resources, so Thread 1 will block.  
But before Thread 1 sleeps, there is a context switch.

# Incorrect Semaphore Implementation Trace

Thread 1	Thread 2
calls <code>P()</code>	...
...	...
<code>sem_count==0</code>	...
<code>spinlock_release</code>	...
<i>context switch →</i>	calls <code>V()</code>
	...
	<code>sem_count++</code>
	<code>wchan_wakeone</code>
	...
	← <i>context switch</i>

Thread 2 produces a resource by calling `V`. At this point,  
`sem_count=1`.

# Incorrect Semaphore Implementation Trace

Thread 1	Thread 2
calls P()	...
...	
sem_count==0	
spinlock_release	
context switch →	calls V()
	...
	sem_count++
	wchan_wakeone
	...
wchan_lock	← context switch
wchan_sleep	

Thread 1 is *blocked on a semaphore that has resources.*

# Correct Semaphore Implementation Trace

Again suppose `spinlock_release` preceeds `wchan_lock`. Consider the case when `sem_count=0`.

Thread 1	Thread 2
calls <code>P()</code>	...
<code>spinlock_acquire</code>	calls <code>V()</code>

The semaphore has no resources, so Thread 1 will block. But before Thread 1 sleeps, Thread 2 calls `V()`.

# Correct Semaphore Implementation Trace

Thread 1	Thread 2
calls P()	...
spinlock_acquire	calls V()
sem_count==0	spinlock_acquire
wchan_lock	spins
spinlock_release	spins

Thread 1 holds the spinlock and continues on while Thread 2 spins.

# Correct Semaphore Implementation Trace

Thread 1	Thread 2
calls P()	...
spinlock_acquire	calls V()
sem_count==0	spinlock_acquire
wchan_lock	spins
spinlock_release	spins
wchan_sleep	sem_count++

Thread 1 releases the spin lock and then goes to sleep.  
Thread 2 becomes unblocked.

# Correct Semaphore Implementation Trace

Thread 1	Thread 2
calls P()	...
spinlock_acquire	calls V()
sem_count==0	spinlock_acquire
wchan_lock	spins
spinlock_release	spins
wchan_sleep	sem_count++
	wchan_wakeone
spinlock_acquire	spinlock_release
...	

Thread 2 increments `sem_count` and wakes Thread 1 which tries to acquire the lock.

# Condition Variables

- OS/161 supports another common synchronization primitive:  
**condition variables**.
- Each condition variable is intended to work together with a lock:  
condition variables are only used *from within the critical section that is protected by the lock*.
- Three operations are possible on a condition variable:
  1. **wait:** This causes the *calling thread to block*, and it releases the lock associated with the condition variable. Once the thread is unblocked it reacquires the lock.
  2. **signal:** If threads are blocked on the signaled condition variable, then *one* of those threads *is unblocked*.
  3. **broadcast:** Like signal, but *unblocks all* threads that are blocked on the condition variable.

# 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)
  - $count < N$  (there is free space in the buffer)
- When a *condition is not true*, a thread can *wait* on the corresponding condition variable until it becomes true.
- When a thread detects that a *condition is true*, it uses *signal* or *broadcast* to notify any blocked threads.
- Note that signalling (or broadcasting to) a condition variable that has no waiters has *no effect*.  $\Rightarrow$  Signals do not accumulate.

## Condition Variable Example

```
int volatile numberOfGeese = 100;  
lock geeseMutex;  
  
int SafeToWalk() {  
    lock_acquire(geeseMutex);  
    while (numberOfGeese > 0) {  
        ... wait? ...  
    }  
    GoToClass();  
    lock_release(geeseMutex);  
}
```

The thread must wait while `numberOfGeese` > 0.

*But the thread holds `geeseMutex`, which prevents other threads from accessing `numberOfGeese`.*

## Condition Variable Example - Solution 1

```
int SafeToWalk() {
    lock_acquire(geeseMutex);
    while (numberOfGeese > 0) {
        lock_release(geeseMutex); // allow access
        lock_acquire(geeseMutex); // restrict access
    }
    GoToClass();
    lock_release(geeseMutex);
}
```

- Releasing and reacquiring `geeseMutex` provides an opportunity for a context switch to occur and another thread might then acquire the lock and modify `numberOfGeese`.
- But the thread should not be waiting for the lock, it should be *waiting for the condition to be true.*

## Condition Variable Example - Solution 2

```
int volatile numberOfGeese = 100;  
lock geeseMutex;  
cv zeroGeese;  
  
int SafeToWalk() {  
    lock_acquire(geeseMutex);  
    while (numberOfGeese > 0) {  
        cv_wait(zeroGeese, geeseMutex);  
    }  
    GoToClass();  
    lock_release(geeseMutex);  
}
```

*Better Approach:* Use a condition variable (cv).

- `cv_wait` will handle releasing and reacquiring the lock passed in.
- It will put the calling thread in the cv's wait channel to block.
- `cv_signal` and `cv_broadcast` wake threads waiting on the cv.

# Waiting on Condition Variables

- When a blocked thread is unblocked (by `signal` or `broadcast`), it reacquires the lock before returning from the `wait` call.
- A thread is in the critical section when it calls `wait` and it will be in the critical section when `wait` returns.
- However, in between the call and the return, while the caller is blocked, the caller is out of the critical section, and other threads may enter.

This describes Mesa-style condition variables, which are used in OS/161. There are alternative condition variable semantics (Hoare semantics), which differ from the semantics described here.

# Bounded Buffer Producer Using Locks and Condition Variables

```
int volatile count = 0;          // must be 0 initially
struct lock *mutex;            // for mutual exclusion
struct cv *notfull, *notempty; // condition variables

/* Note: the lock and cv's must be created using lock_create()
 * and cv_create() before Produce() and Consume() are called */

Produce(itemType item) {
    lock_acquire(mutex);
    while (count == N) {
        cv_wait(notfull, mutex); // wait until buffer is not full
    }
    add_item(item, buffer);    // add item to buffer
    count = count + 1;
    cv_signal(notempty, mutex); // signal that buffer is not empty
    lock_release(mutex);
}
```

# Bounded Buffer Consumer Using Locks and Condition Variables

```
itemType Consume() {
    lock_acquire(mutex);
    while (count == 0) {
        cv_wait(notempty, mutex); // wait until buffer is not empty
    }
    item = remove_item(buffer); // remove item from buffer
    count = count - 1;
    cv_signal(notfull, mutex); // signal that buffer is not full
    lock_release(mutex);
    return(item);
}
```

Both `Produce()` and `Consume()` call `cv_wait()` inside of a `while` loop. Why?

# Volatile and Other Sources of Race Conditions

- Notice throughout these slides that shared variables were declared **volatile**.
- Race conditions can occur for reasons *beyond the programmer's direct control*, specifically both:
  - the **compiler** and
  - the **CPU**can introduce race conditions.
- In both cases, the race conditions due to optimizations that try have the code execute faster.
- **Memory models** describe how thread access to memory in shared regions behave.
  - A memory model suggests which optimizations can be performed.

# Volatile

- E.g. it is *faster to access values from a register*, than it is from main memory.
- Compilers optimize for this difference; storing values in registers for as long as possible.
- Consider the following snippets of code:

```
int sharedTotal = 0;  
int FuncA() { ... code that modifies sharedTotal ... }  
int FuncB() { ... code that modifies sharedTotal ... }
```

If the compiler optimization stores `sharedTotal` in register `S1` for `FuncA` and register `S2` for `FuncB`, then which register has the correct value for `sharedTotal`?

- `volatile` disables this optimization, *forcing the value to be loaded from and stored to memory with each use*, it also prevents the compiler from re-ordering loads and stores for that variable.
- *Shared variables should be declared `volatile` in your code.*

## Other Language and Instruction Level Instructions

- Many languages support multi-threading with memory models and language-level synchronization functions (i.e., locks).
  - The *compiler is aware of critical sections via language-level synchronization functions* and does not perform optimizations which can cause race conditions.
  - The version of C used by OS/161 does not support this.
- The CPU also has a memory model as it also *re-orders loads and stores* to improve performance.
  - Modern architectures provide barrier or fence instructions to disable and reenable these CPU-level optimizations to prevent race conditions at this level.
  - The MIPS R3000 CPU used in this course does not have or require these instructions.

# Deadlocks

Consider the following pseudocode:

```
lock lockA, lockB;  
int FuncA() {  
    lock_acquire(lockA)  
    lock_acquire(lockB)  
    ...  
    lock_release(lockA)  
    lock_release(lockB)  
}  
  
int FuncB() {  
    lock_acquire(lockB)  
    lock_acquire(lockA)  
    ...  
    lock_release(lockB)  
    lock_release(lockA)  
}
```

What happens if the instructions are interleaved as follows:

- Thread 1 executes `lock_acquire(lockA)`
- Thread 2 executes `lock_acquire(lockB)`
- Thread 1 executes `lock_acquire(lockB)`
- Thread 2 executes `lock_acquire(lockA)`

# Deadlocks

Consider the following pseudocode:

```
lock lockA, lockB;  
int FuncA() {  
    lock_acquire(lockA)  
    lock_acquire(lockB)  
    ...  
}  
  
int FuncB() {  
    lock_acquire(lockB)  
    lock_acquire(lockA)  
    ...  
}
```

This is what happens...

- Thread 1 executes `lock_acquire(lockA)` and holds it.
- Thread 2 executes `lock_acquire(lockB)` and holds it.
- Thread 1 executes `lock_acquire(lockB)` and blocks because `lockB` is being held by Thread 2.
- Thread 2 executes `lock_acquire(lockA)` and blocks because `lockA` is being held by Thread 1.

The threads are **deadlocked**, i.e. neither can make progress. Waiting will not resolve the deadlock. They *are permanently blocked*.

## Two Techniques for Deadlock Prevention

**No Hold and Wait:** *prevent a thread from requesting resources if it currently has resources allocated to it.*

A thread may hold several resources, but to do so it must make a single request for all of them.

**Resource Ordering:** *Order (i.e., number) the resource types and require that each thread acquire resources in increasing resource type order.*

That is, a thread may make no requests for resources of type less than or equal to  $i$  if it is holding resources of type  $i$ .

## Summary: Synchronization and Locks

- A **race condition** is when the program result depends on the order of execution. [2.2–8]
- The coordination of access to a shared resource is called **synchronization**. [2.1]
- A common method of synchronization is to enforce **mutual exclusion** to a **critical section**. [2.1, 11.1]
  - I.e. a thread must **Acquire** a lock before entering the critical section and **Release** the lock after leaving it. [12-14]
- Processors often provide instructions that can be used to implement the **test-and-set** portion of a lock implementation.
  - x86-64 provides the **xchg**, [15-16]
  - ARM provides the **LDREX** and **STREX**, [17-18]
  - MIPS32 provides the **ll** and **sc**, [19-20]

# Summary: Synchronization Primitives

- Four types of synchronization primitives are covered in this course.
  - **spinlocks**, [21–22.1]
  - **locks** which use `lock_acquire` and `lock_release`, [23–24]
  - **semaphores** which use `P` and `V`, [27-40]
  - **condition variables**. [41-48]
- Spinlocks **busy-wait**, i.e. they actively use the CPU while waiting. [16, 18, 20-21]
- Locks, semaphores and condition variables **block** and use **wait channels** while waiting. [24-25]
- Recall, a thread can be: running, ready and blocked. [26]
- There are two types of **semaphores**: [28]
  1. **binary semaphore**: has value 0 or 1,
  2. **counting semaphore**: has a non-negative value.

## Summary: Semaphores

- The semaphore has two operations: [27]
  - **P** (procure) will decrement the semaphore if it is greater than 0 or block until it is.
  - **V** (vacate) will increment the semaphore.
- Unlike locks, semaphores do not have an owner. [27]
- Semaphores are often used in **Producer Consumer** problems where some threads are producing items and others are consuming (i.e. processing) them. [30-32]
- In order to debug multithreaded code, think of all the possible interleavings of threads executing the code. [34-39]
- **Condition Variables** (CVs) are used to check wait for a condition to be true while in a critical section. [41]

## Summary: Condition Variables

- CVs have three operations: [41]
  - `wait` causes a thread to block and release the lock
  - `signal` unblocks a single thread
  - `broadcast` unblocks all blocked threads
- Compiler and CPU optimizations can cause race conditions unless a shared variable is declared **volatile**. [49–51]
- A **deadlock** is when a group of threads (or processes) are waiting for an event that can only be completed by another member the group. [52–53]
- Deadlocks can be prevented by: [54]
  - **No Hold and Wait** prevent threads from making multiple requests for resources
  - **Resource Ordering** acquire resources in a fixed order

# Processes and the Kernel

**key concepts:** process, system call, processor exception, fork/execv, multiprocessing

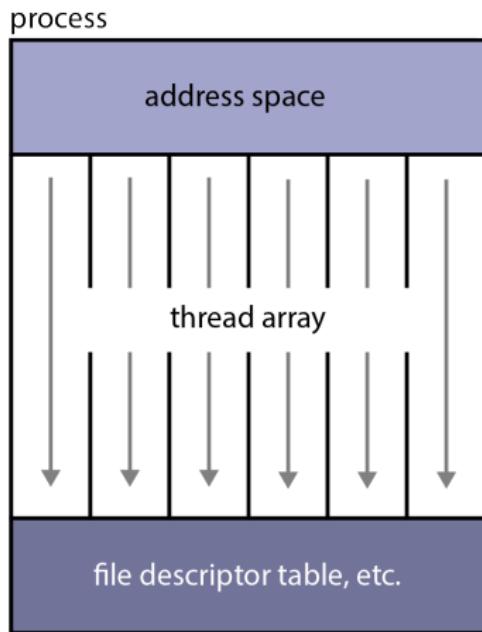
Lesley Istead, Kevin Lanctot

David R. Cheriton School of Computer Science  
University of Waterloo

Winter 2019

# What is a Process?

A **process** is an environment in which an application program runs (as opposed to a program which is a file in secondary storage e.g. HDD). You can think of it as *a program that is running*.



- The *process's environment includes virtualized resources* that its program can use:
  - one (or more) threads
  - an address space used for the program's code and data
  - other resources, e.g., file and socket descriptors
- processes are created and managed by the kernel
- each program's process *isolates* it from other programs in other processes

# What is a Process?

- A **virtualized resource** is to take a physical resources (e.g. a Toshiba Model MQ01ABD100 1TB Hard Disk Drive) and create a more abstract version of it (e.g. a file system).
- **Virtual memory** is an example of virtualization involving RAM and secondary storage, which is made to appear as one large primary storage (RAM). More on this topic soon.
- A **socket** is a endpoint for exchanging data between two processes (often on different machines).
- Processes are isolated from one another in the sense that when you run a program it is as if that program has exclusive access to the processor, RAM and I/O devices when in fact they are shared. A bug in one program should not affect the stability or outcome of other processes.

# Process Management Calls

*Processes can be created, managed, and destroyed.* Each OS supports a variety of functions to perform these tasks.

	Linux	OS/161
Creation	fork,execv	fork,execv
Destruction	_exit,kill	_exit
Synchronization	wait,waitpid,pause,...	waitpid
Attribute Mgmt	getpid,getuid,nice,getrusage,...	getpid

In CS350 A2a and A2b, you will be implementing part of OS/161's process management.

# fork

The system call `fork` creates a new process, the **child** (C), *that is a clone* of the the **parent** (P) (i.e. the orginal process).

- It copies P's address space (code, globals, heap, stack) into C's.
- It makes a new thread for C.
- It copies P's trap frame to the C's (kernel) stack, (more on this topic later).
- The address space (code, globals, heap, stack etc) of the parent and child are identical at the time of the fork, but may diverge afterwards.
- `fork` is called by the parent, but returns in *both* the parent and child
- The parent and child *see different return values* from `fork` ⇒
  - it returns 0 to the child process
  - it returns the child's pid to the parent process.

- `_exit` terminates the process that calls it
  - a process can supply an exit status code when it exits
  - the kernel records the exit status code in case another process asks for it (via `waitpid`)
- `waitpid` lets a process wait for another to terminate, (i.e. block) and then retrieve its exit status code

# The fork, \_exit, getpid and waitpid system calls

```
// sample code that uses fork, getpid, waitpid and _exit

1. main() {
2.     rc = fork();           // returns 0 to child, pid to parent
3.     if (rc == 0) {        // child executes this code
4.         my_pid = getpid();
5.         x = child_code();
6.         _exit(x);
7.     } else {              // parent executes this code
8.         child_pid = rc;
9.         parent_pid = getpid();
10.        parent_code();
11.        p = waitpid(child_pid,&child_exit,0);
12.        if (WIFEXITED(child_exit))
13.            printf("child exit status was %d",
14.                      WEXITSTATUS(child_exit))
15.    }
}
```

## The fork, \_exit, getpid and waitpid system calls

- lines 2, 3 and 7: the parent and child receive two different return values, which can be used to differentiate the child (`rc=0`) from the parent (`rc=pid of child`)
- line 4 and 9: `getpid()` is like checking the student number of two identical twins: they look the same but have different pids (process identifiers)
- line 5 and 10: even though the two processes are running identical code, they can be set up to execute two different functions, e.g. `parent_code()` vs. `child_code()`
- line 6: `_exit(x)` ends the process and `x` is the return value
- line 11: the parent waits for the child to exit and gets its exit status
- line 12: `WIFEXITED` returns true if the child called `_exit()`
- line 13: `WEXITSTATUS` returns the exit return value

The macros `WIFEXITED` and `WEXITSTATUS` are defined in `kern/include/kern/wait.h`

# The execv and getpid system calls

## getpid

- returns the **process identifier (pid)** of the current process
- each existing process has its own (unique) pid to identify it

## execv

- `execv` changes the program that a process is running.
- The calling process's current virtual memory is destroyed.
- The process gets a new virtual memory (code, heap, stack, etc) initialized with the code and data of the new program to run.
- After `execv`, the new program starts executing.
- The `pid` remains the same.
- `execv` can pass arguments to the new program, if required

## execv example

```
//      sample code that uses execv to execute the command:  
//      /testbin/argtest first second  
  
int main()  {  
1.      int status = 0;    // status of execv function call  
2.      char *args[4];    // argument vector  
  
            // prepare the arguments  
3.      args[0] = (char *) "/testbin/argtest";  
4.      args[1] = (char *) "first";  
5.      args[2] = (char *) "second";  
6.      args[3] = 0;       // end of args  
  
7.      status = execv("/testbin/argtest", args);  
8.      printf("If you see this output then execv failed");  
9.      printf("status = %d errno = %d", status, errno);  
10.     exit(0);  
}
```

## execv example

- lines 2–6: set up the arguments for the `execv` system call
- line 7: call `execv` and keep the return value
- line 8: if `execv`
  - fails  $\Rightarrow$  the current program will continue executing
  - succeeds  $\Rightarrow$  the current program will be replaced with the program `argtest`
- line 9: `errno` is a global variable that holds the value of the last error number
- line 10: exit with status 0

## Combining fork and execv

```
// the child executes execv while the parent waits

main() {
1.     int rc = 0;          // return code
2.     // declare and set args here
3.     rc = fork();        // returns 0 to child, pid to parent

4.     if (rc == 0) {      // child's code
5.         status = execv("/testbin/argtest",args);
6.         printf("If you see this output, then execv failed");
7.         printf("status = %d errno = %d", status, errno);
8.         exit(0);

9.     } else {            // parents's code
10.        child_pid = rc;
11.        parent_code();
12.        p = waitpid(child_pid,&child_exit,0);
13.    }
}
```

## execv example

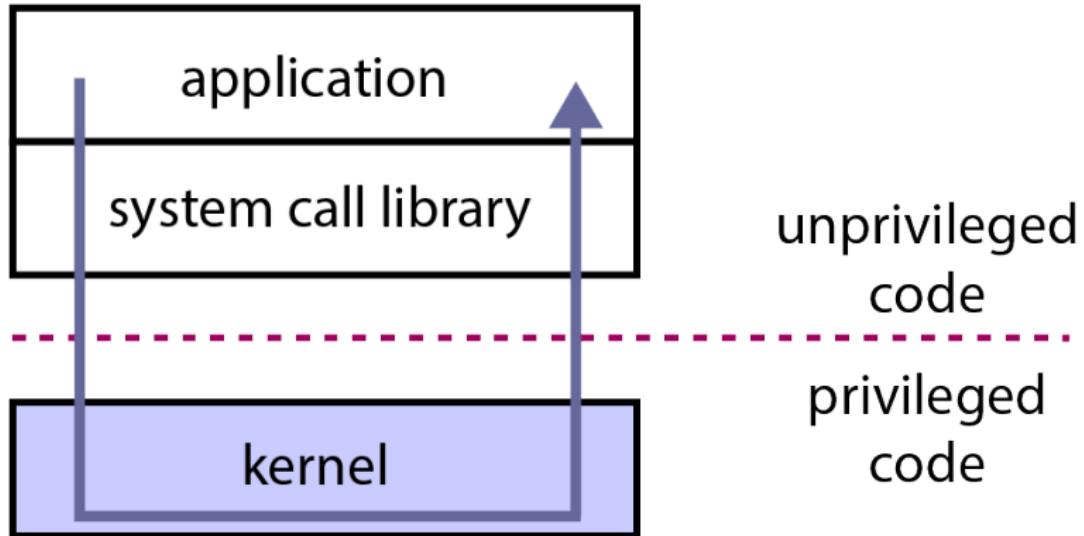
- lines 2: sets up the arguments for the `execv` system call, as was done on lines 2–6 in the previous example
- line 5: executes the `execv` system call in the child, which runs the `argtest` program
- lines 5–8: exactly the same code as the call to `execv` in the previous example
- lines 10–12: parent's code, which executes `parent_code()` and then waits for the child to exit (i.e. it blocks).

# System Calls

- Process management calls, e.g., `fork`, are called by user programs.
- They are also system calls.
- **System calls** are the interface between user processes and the kernel.

Services	OS/161 Examples
create, destroy, manage processes	<code>fork, execv, waitpid, getpid</code>
create, destroy, read, write files	<code>open, close, remove, read, write</code>
manage file system and directories	<code>mkdir, rmdir, link, sync</code>
interprocess communication	<code>pipe, read, write</code>
manage virtual memory	<code>sbrk</code>
query, manage system	<code>reboot, __time</code>

# System Call Software Stack



- The CPU implements different levels (or rings) of **execution privilege** as a security and isolation mechanism.
- *The kernel code runs at the highest privilege level*, where any CPU instructions can be executed.
- Application (i.e. user) code runs at a lower privilege level because user programs *should not* be permitted to perform certain tasks such as:
  - modifying the page tables that the kernel uses to implement process virtual memories (address spaces), i.e. access RAM that has not been allocated to this process
  - halting the CPU

# Kernel Privilege

- *Programs in user space cannot execute code or instructions belonging to a higher-level of privilege.*
- These restrictions allow the kernel to keep processes isolated from one another — and from the kernel.
- Application programs cannot directly call kernel functions or access kernel data structures.

The Meltdown vulnerability found on Intel chips allows user applications to bypass execution privilege and access any address in physical memory.

# How System Calls Work (Part 1)

**Key Question:** Since application programs cannot directly call the kernel, how does a program make a system call such as `fork`?

There are only two mechanisms to execute kernel code:

1. **Interrupts**

Interrupts are *generated by devices* when they need attention.

2. **Exceptions**

Exceptions are *caused by instruction execution* when a running program needs attention.

- **Interrupts** are raised by devices (hardware)  
e.g. system clock, WiFi card received a packet, an HDD has accessed some data.
- An interrupt causes the CPU to transfer control to a fixed location in memory (specified by the designers of the CPU) where an **interrupt handler** must be located.
- *Interrupt handlers are part of the kernel.*
  - If an interrupt occurs while an application program is running, control will jump from the application to the kernel's interrupt handler routine
- When an interrupt occurs, the processor switches to privileged execution mode when it transfers control to the interrupt handler
  - This is how the kernel gets its execution privilege

# Exceptions

- **Exceptions** are *conditions that occur during the execution of a program instruction.*
  - Examples: arithmetic overflows, illegal instructions, illegal addresses or page faults (to be discussed later).
- Exceptions are detected by the CPU during instruction execution
- The CPU handles exceptions like it handles interrupts:
  - control is transferred to a fixed location, where an **exception handler** is located
  - the processor is switched to privileged execution mode
- The exception handler is part of the kernel.
- In OS/161 *the same routine is used to handle exceptions and interrupts.*
- When an exception or interrupt happens, the CPU stores the exception code in the **cause** register.

# MIPS Exception Types

EX_IRQ	0	// Interrupt
EX_MOD	1	// TLB Modify (write to read-only page)
EX_TLBL	2	// TLB miss on load
EX_TLBS	3	// TLB miss on store
EX_ADEL	4	// Address error on load
EX_ADES	5	// Address error on store
EX_IBE	6	// Bus error on instruction fetch
EX_DBE	7	// Bus error on data load *or* store
EX_SYS	8	// Syscall
EX_BP	9	// Breakpoint
EX_RI	10	// Reserved (illegal) instruction
EX_CPU	11	// Coprocessor unusable
EX_OVF	12	// Arithmetic overflow

- The handler, `mips_trap`, uses these codes to determine what triggered it to run (EX\_IRQ is interrupt, rest are exceptions).
- Ref: `kern/arch/mips/include/trapframe.h`

## How System Calls Work (Part 2)

- *To perform a system call, the application program needs to cause an exception to make the kernel execute:*
  - on the MIPS processor, EX\_SYS is the system call exception
- To cause this exception on the MIPS processor, the application executes a special purpose instruction: `syscall`
  - Other processor instruction sets include similar instructions, e.g., it is also `syscall` on x86-64.
- The kernel's exception handler checks the exception code (set by the CPU when the exception is generated) to distinguish system call exceptions from other types of exceptions.

# Which System Call?

- **Key Question:** There is only one `syscall` exception, `EX_SYS`. So how does the OS distinguish between a `fork` and `getpid` system call?
- **Answer: system call codes**
- The kernel defines a code for each system call it understands.
- The kernel expects the application to place the appropriate system call code in a specified location before executing the `syscall` instruction.
  - for OS/161 on the MIPS processor, it goes in register `v0`
  - E.g. `li v0, 0` loads the system call code for `fork` into `v0`.
- The kernel's exception handler checks this code to determine which system call has been requested.
- The codes and code location are part of the kernel's **ABI**. (Application Binary Interface).

# Some OS/161 System Call Codes

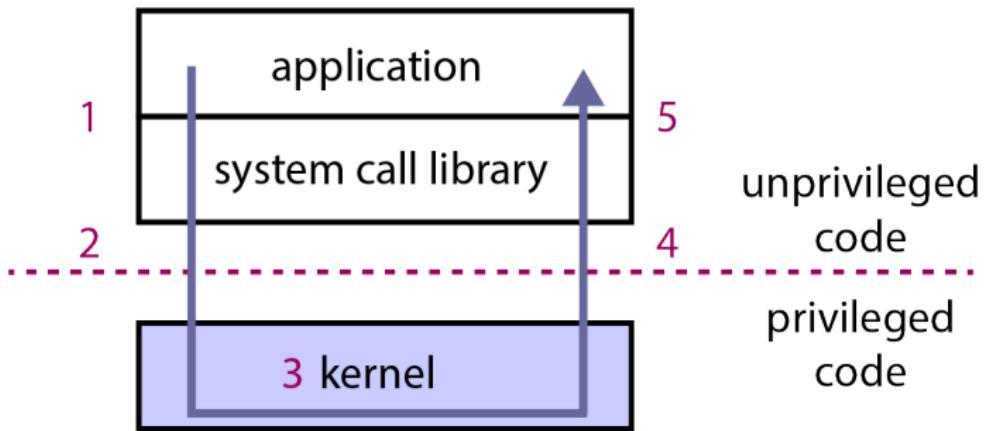
```
#define SYS_fork          0
#define SYS_vfork          1
#define SYS_execv          2
#define SYS__exit           3
#define SYS_waitpid         4
#define SYS_getpid          5
...
...
```

- These constants come from `kern/include/kern/syscall.h`
- There are roughly 120 of them. One for each system call.
- The files in `kern/include/kern` define things (like system call codes) that must be known by both the kernel and applications.

## System Call Parameters

- System calls take arguments and return values, like function calls.
- *Key Question:* How does this work, since system calls are really just exceptions?
- *Answer:* *The application places the arguments in kernel-specified locations* before the `syscall`, and receives the return values in kernel-specified locations after the exception handler returns.
  - The locations are specified in the kernel Application Binary Interface (ABI)
  - Parameter and return value placement is handled by the application's system call library functions
  - On MIPS, arguments go in registers `a0,a1,a2,a3`
    - result success/fail code is in `a3` upon return
    - return value or error code is in `v0` upon return

# System Call Software Stack (again)



System calls are expensive.

- Which is faster?

$N$  separate `print` calls, or forming a string of  $N$  numbers and a single `print`.

# System Call Software Stack (again)

1. *application* calls a library wrapper function for the desired system call
2. *library function* executes a `syscall` instruction causing ...
3. the kernel's *exception handler* to run. It
  - a) creates trap frame to save application program state
  - b) determines that this is a system call exception
  - c) determines which system call is being requested
  - d) does the work for the requested system call
  - e) restores the application program state from the trap frame
  - f) returns from the exception
4. then the *library wrapper function* finishes and returns from its call
5. the *application* continues execution

Every OS/161 process thread has two stacks, although it only uses one at a time.

1. **User (Application) Stack:** used while *the thread is executing application code*

- you saw this stack in CS241
- this stack is located in the application's virtual memory
- it holds the stack frames (a.k.a activation records) for the application's functions
- the kernel creates this stack when it sets up the virtual address memory for the process

2. **Kernel Stack:** used while *the thread is executing kernel code*, i.e. after an exception or interrupt.
  - this stack is a kernel structure (i.e. it is located in the kernel's address space)
  - in OS/161, the `t_stack` field of the `thread` structure points to this stack
  - this stack holds stack frames (a.k.a. activation records) for the kernel functions
  - this stack also holds *trap frames* and *switch frames* (because it is the kernel that creates trap frames and switch frames)

- (1) An application calls (2) a library function which makes a syscall causing an exception. To handle the exception the
- first to run is assembly code, `common_exception`, that
    - saves the application's stack pointer
    - switches the stack pointer to point to the thread's kernel stack
    - carefully saves application state and the address of the instruction that was interrupted in a trap frame on the thread's kernel stack
    - calls `mips_trap`, passing a pointer to the trap frame as a parameter
  - after `mips_trap` is finished, the `common_exception` handler will
    - restore application's state (including the application stack pointer) from the trap frame on the thread's kernel stack
    - jump back to the application instruction that was interrupted and switch back to unprivileged execution mode
  - see `kern/arch/mips/locore/exception-mips1.S`

- The C function `mips_trap` *determines what type of exception it is by looking at the exception code*: is it an interrupt? a system call? or something else?
- There is a separate handler in the kernel for each type of exception:
  - interrupt ⇒ call `mainbus_interrupt`
  - address translation exception ⇒ call `vm_fault`  
(important for later assignments!)
  - system call ⇒ call `syscall` (a kernel function), passing it the trap frame pointer
  - `syscall` is in `kern/arch/mips/syscall/syscall.c`
- see `kern/arch/mips/locore/trap.c`

# Why Have System Calls?

## A Key Challenge

- *Observation 1:* The kernel is a program, just like any other program, i.e. a sequence of instructions.
- *Observation 2:* We want the kernel to be protected and have privileges that no other program has (i.e. the ability to manage the hardware).
- *Key Challenge:* How to keep the kernel isolated from application processes and yet allow the application processes to use the services of the kernel.
- *Key Idea:* Have two different types of calls
  1. *procedure calls:* used by applications to execute other application code
  2. *system calls:* used by applications to execute kernel code

# Multiprocessing

- **Multiprocessing** (or **multitasking**) means having multiple processes existing at the same time
- Recall (from slide 2) the resources that a process has access to:
  - the processor (accessed via an executing thread)
  - memory
  - storage and I/O devices
- For multitasking: *all processes must share the available hardware resources.* ⇒ *This sharing is coordinated by the operating system.*
  - Each process's threads are scheduled to execute on available CPUs (or CPU cores) by the OS.
  - Each process's virtual memory is implemented using some of the available physical memory. The OS decides how much memory each process gets.
  - Processes share access to other resources (e.g., disks, network devices, I/O devices) by making system calls. The OS controls this sharing.

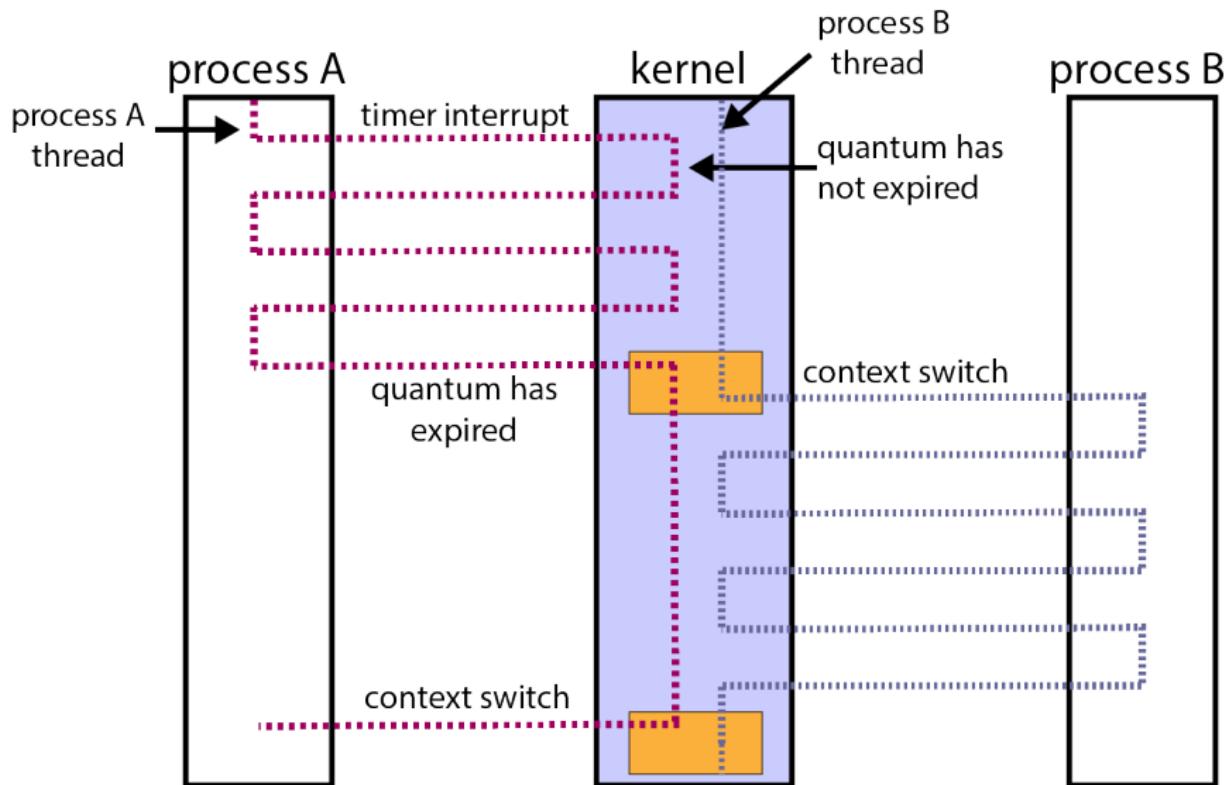
# Multiprocessing

- The OS ensures that *processes are isolated from one another*.
  - I.e. a bug in one process should not affect another process.
- Interprocess communication should be possible, but only at the explicit request of the processes involved.
  - i.e. no interprocess communication unless both processes have set up the communication channel.
- Note: Processes may have many threads, but must have at least one thread to execute.

OS/161

- OS/161 allows multithreaded code in kernel space.
- But it currently only supports a single thread per process in user-space.

## Two-Process Example



## Two-Process Example

The diagram illustrates timesharing between two processes A and B.  
Note: time is moving from top (earlier) to bottom.

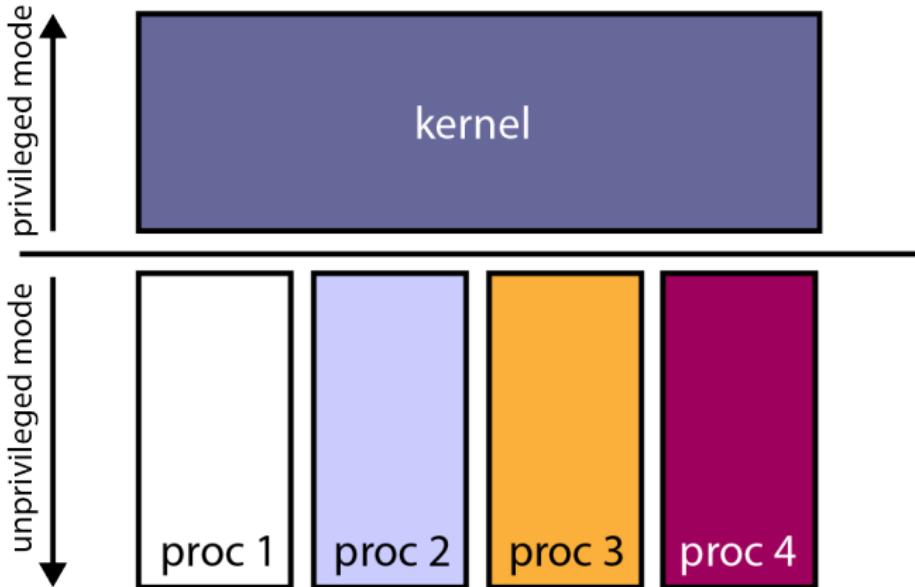
1. Initially, process A is running and process B is on the ready queue.
2. When the dotted line corresponding to a thread is in the kernel space, it means that thread is waiting in the kernel's ready queue.
3. Then a *timer interrupt* occurs, so the kernel checks if A's quantum has expired (i.e. has it used up its allotted time to run).
4. It *has not expired*, so the kernel allows A to continue to run.
5. Eventually A's quantum *does expire* and B is allowed to run.
6. Timer interrupts will happen while B is running
7. After three timer interrupts, A is scheduled to run again.

## Example: System Calls

The following sequence of diagrams will illustrate a system call (for `fork`) followed by a timer interrupt.

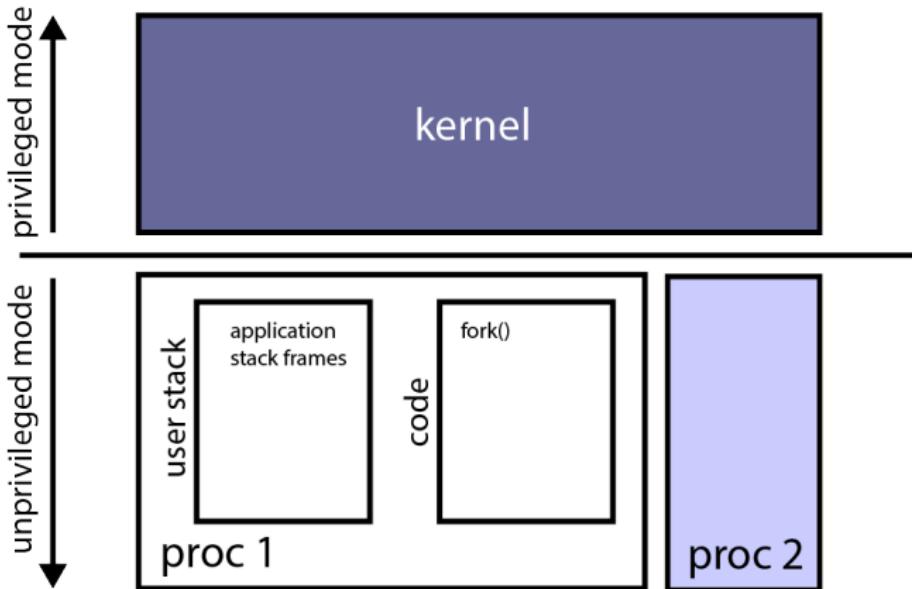
- The user-space contains four processes, which all run in unprivileged mode.
- The kernel space runs in privileged mode.
- Unlike the previous illustration of dealing with a timer interrupts (in Threads and Concurrency, slides 22–40), here we will distinguish what frames go on a thread's *user stack* vs. its *kernel stack*.

## Example: System Calls (1/27)



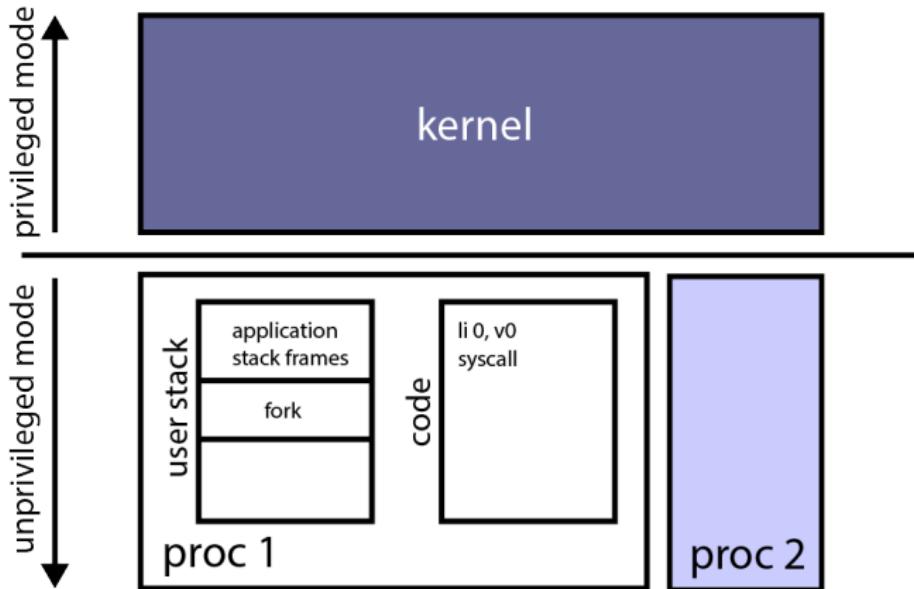
The initial configuration: four process and the kernel.

## Example: System Calls (2/27)



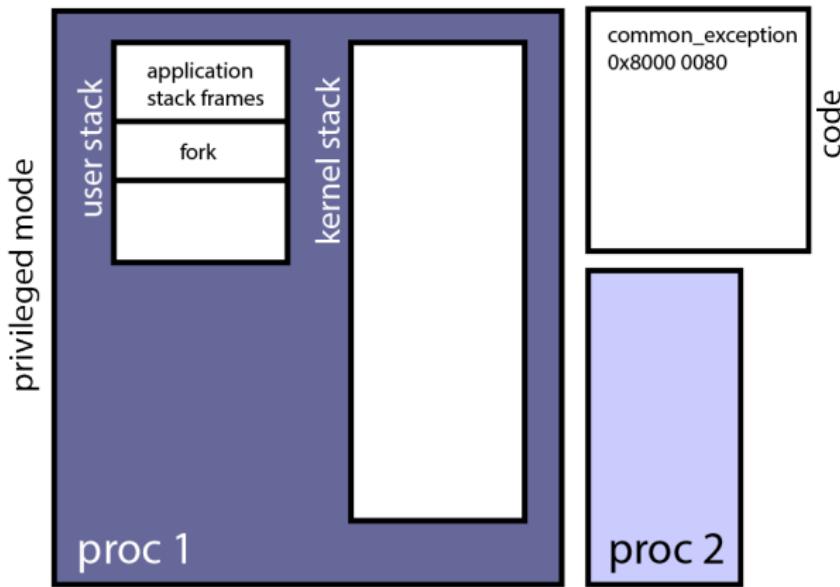
proc1 calls `fork` (a system call library function).

## Example: System Calls (3/27)



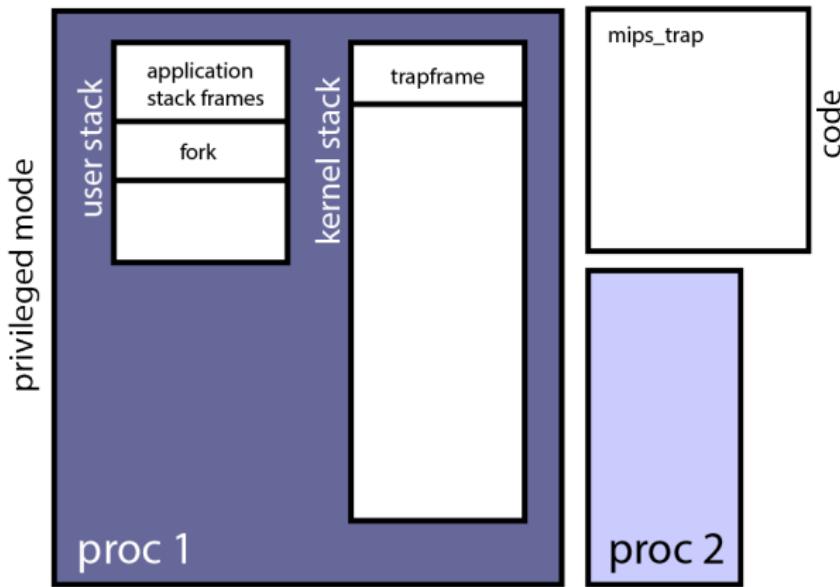
`fork` puts the system call code in register `v0` and executes the MIPS instruction `syscall` which raises an exception.

## Example: System Calls (4/27)



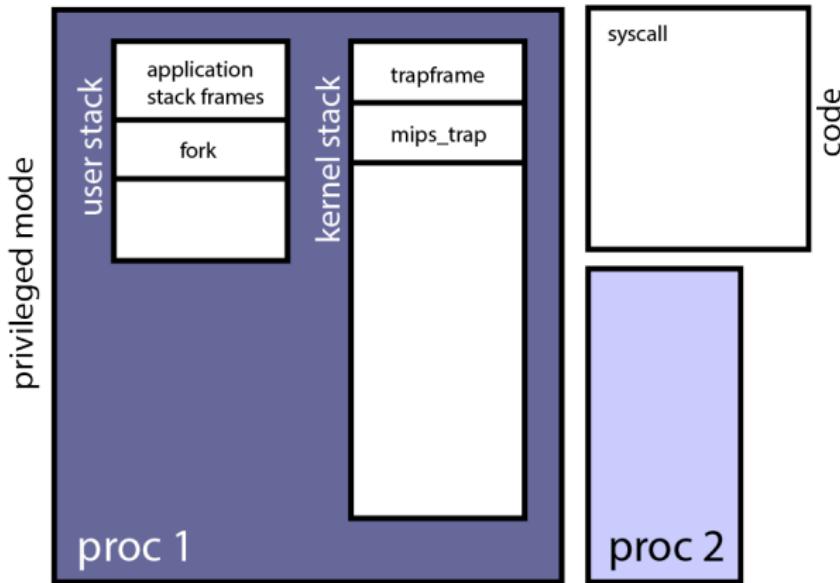
*When the exception is raised*, the CPU goes into privileged mode and interrupts are turned off. The kernel jumps to `common_exception` and switches from the user to the kernel stack and saves the trapframe.

## Example: System Calls (5/27)



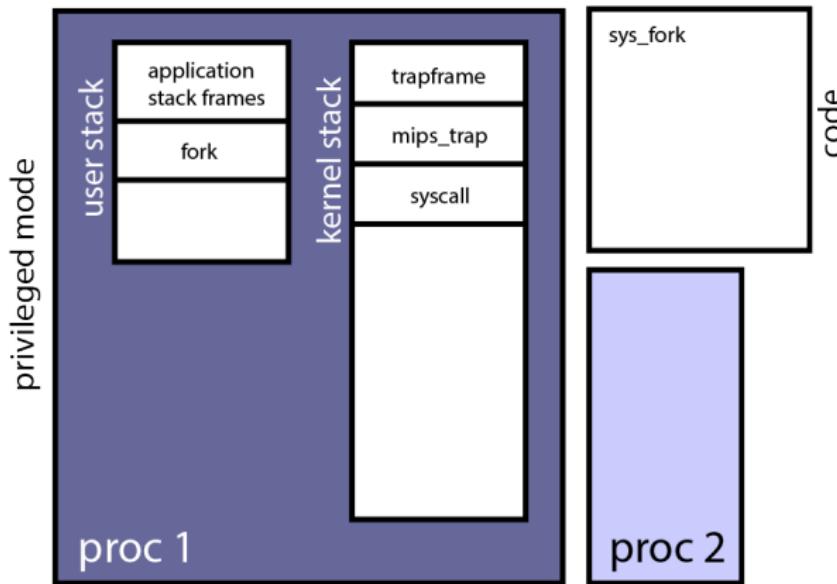
After saving the state, `common_exception` calls `mips_trap` to determine what kind of exception was raised. `mips_trap` determines that the exception is a system call.

## Example: System Calls (6/27)



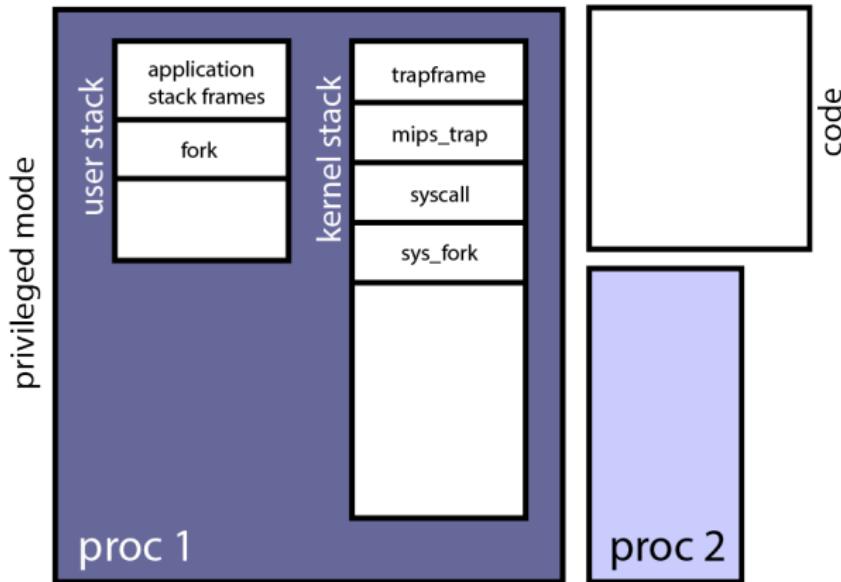
For a system call, interrupts are turned back on. `mips_trap` calls `syscall` (the kernel function not the MIPS instruction) to dispatch the correct function.

## Example: System Calls (7/27)



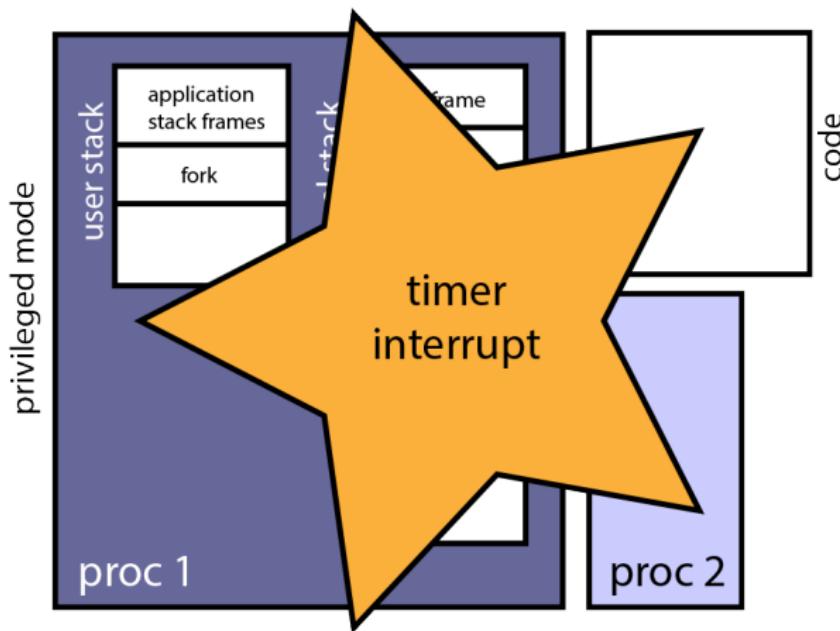
**syscall**, the system call dispatcher, calls the appropriate handler for the system call code provided in **v0**. In this case, **sys\_fork** is called.

## Example: System Calls (8/27)



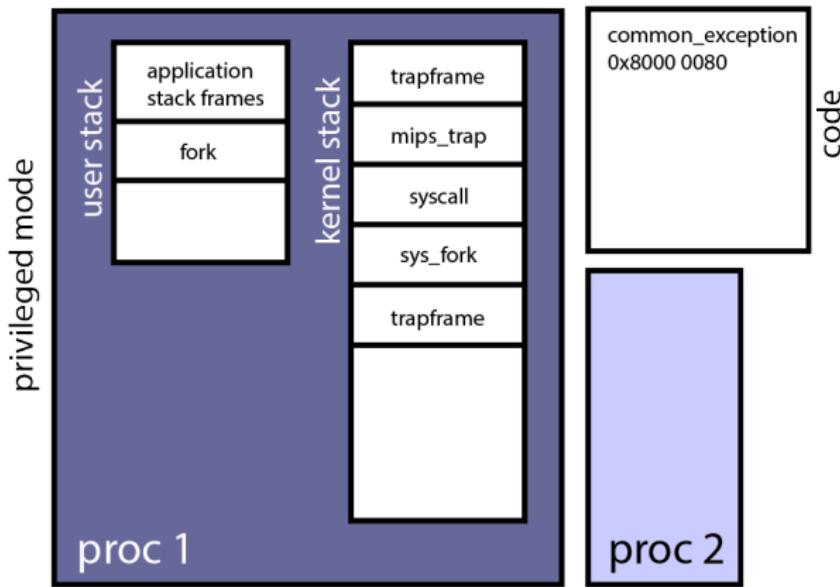
The system call `sys_fork` is executed by the kernel.

## Example: System Calls (9/27)



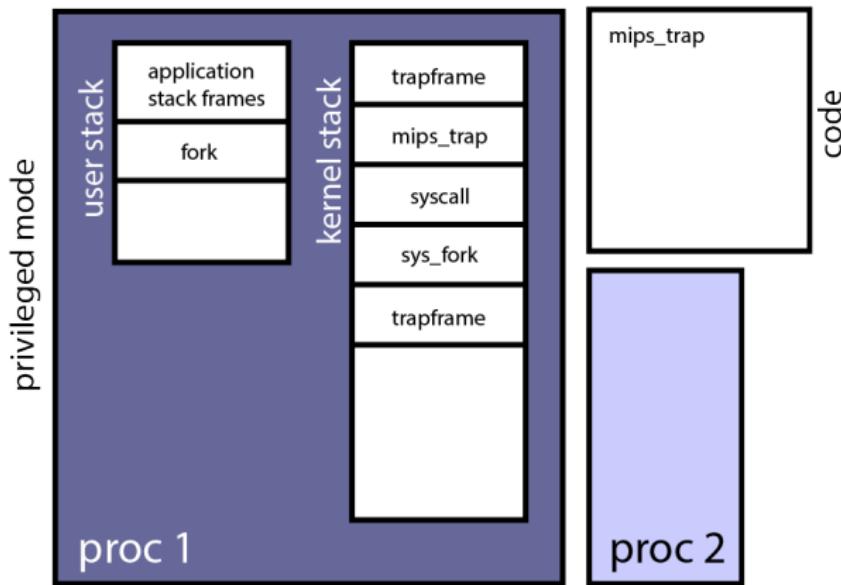
**Question:** What happens if a timer interrupt occurs at this point?

## Example: System Calls (10/27)



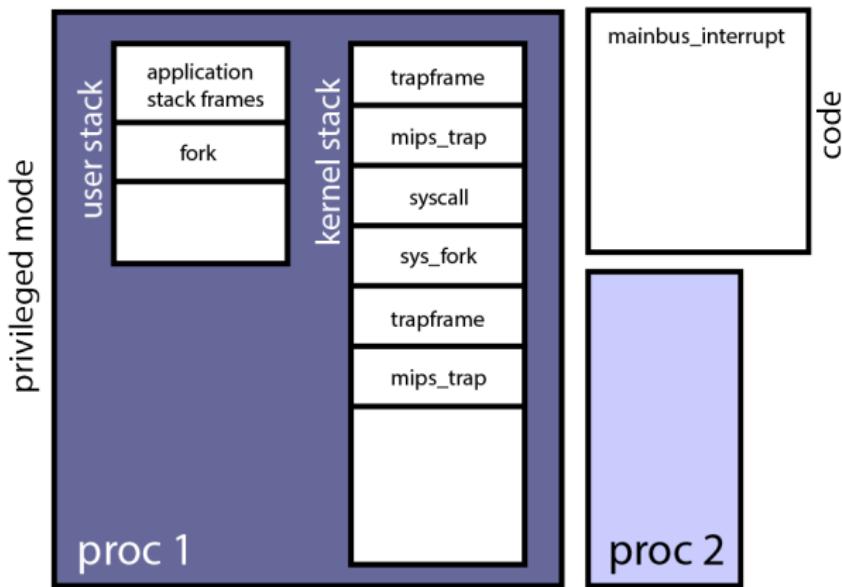
The CPU jumps to `common_exception` and interrupts are turned off. The kernel subroutine `common_exception` saves another trapframe and calls `mips_trap`.

## Example: System Calls (11/27)



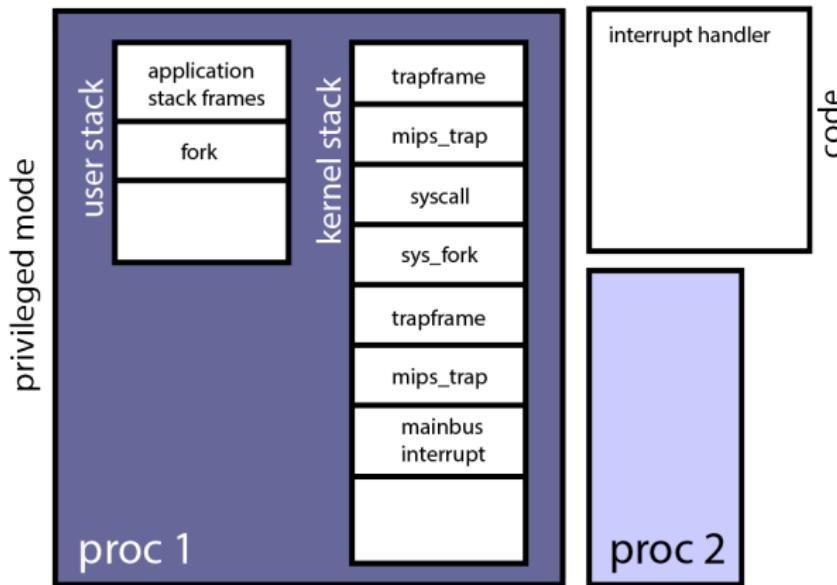
**mips\_trap** determines which exception has been raised. In this case, it is an interrupt.

## Example: System Calls (12/27)



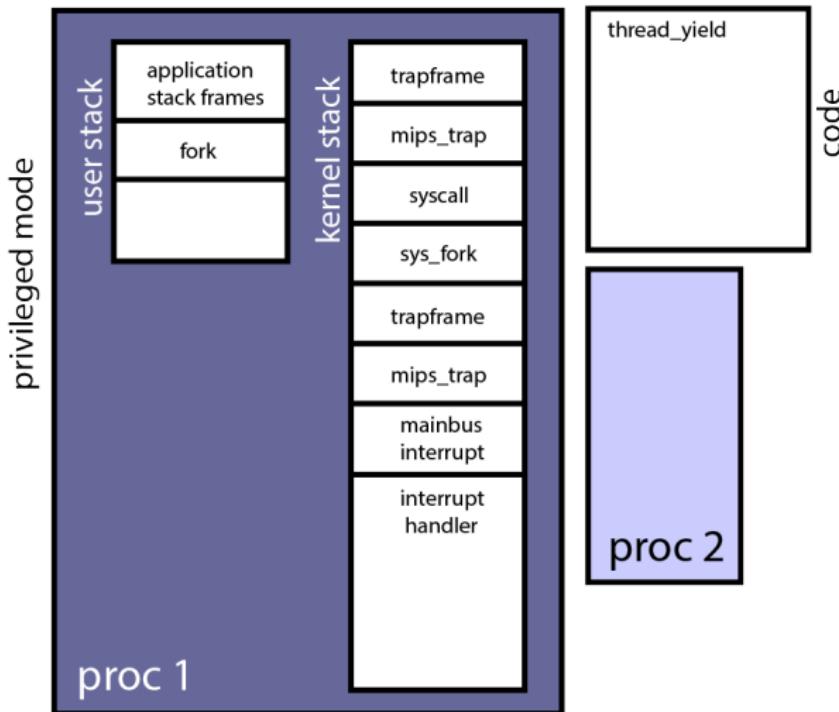
`mainbus_interrupt` determines which device threw the interrupt (by examining the cause register), then calls the appropriate handler.

## Example: System Calls (13/27)



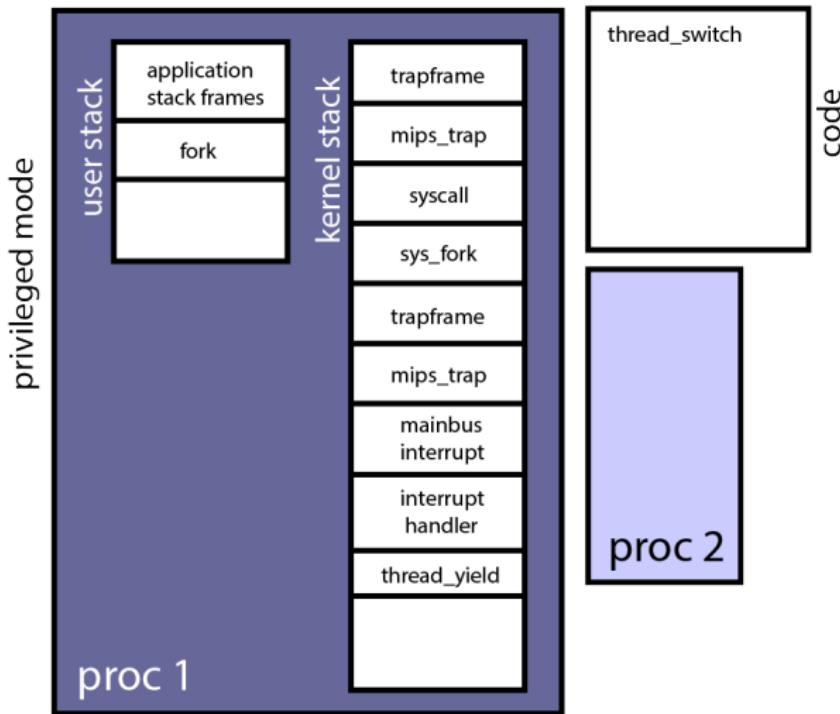
The device interrupt handler runs and determines that the thread's quantum has expired.

## Example: System Calls (14/27)



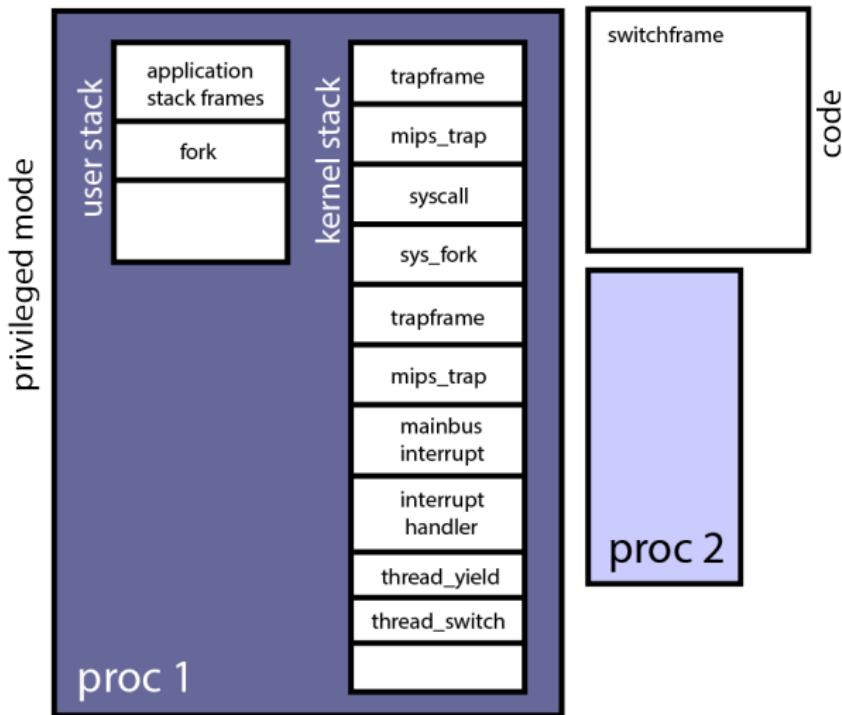
`thread_yield` is called to perform context switch.

## Example: System Calls (15/27)



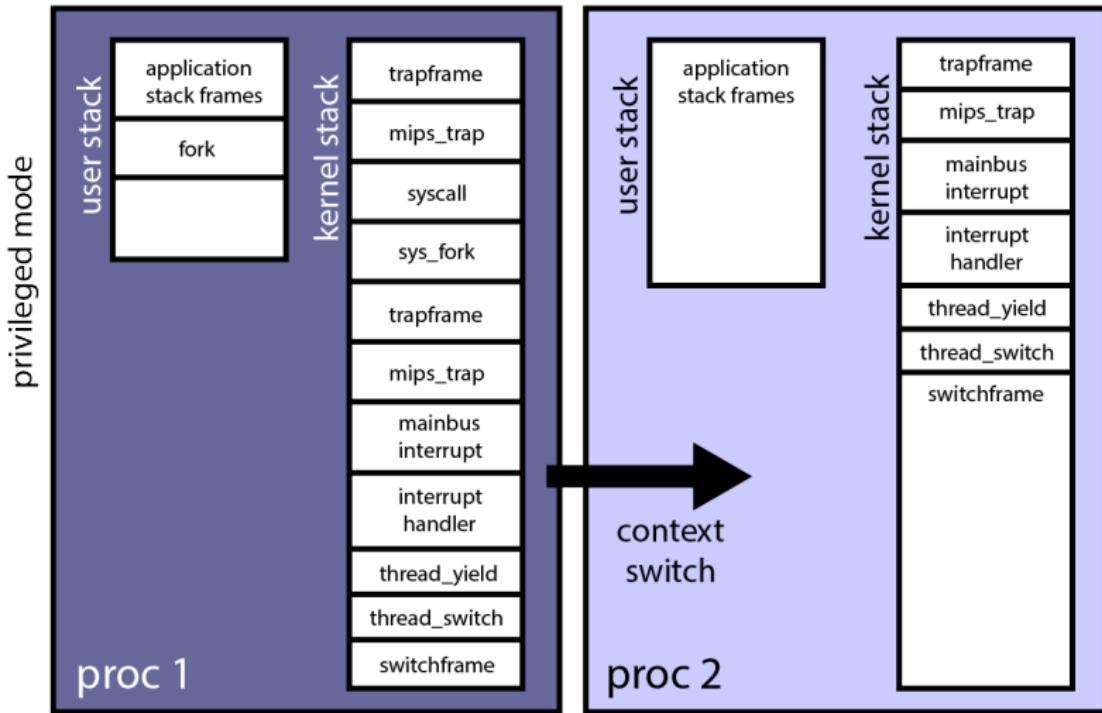
`thread_yield` calls `thread_switch`.

## Example: System Calls (16/27)



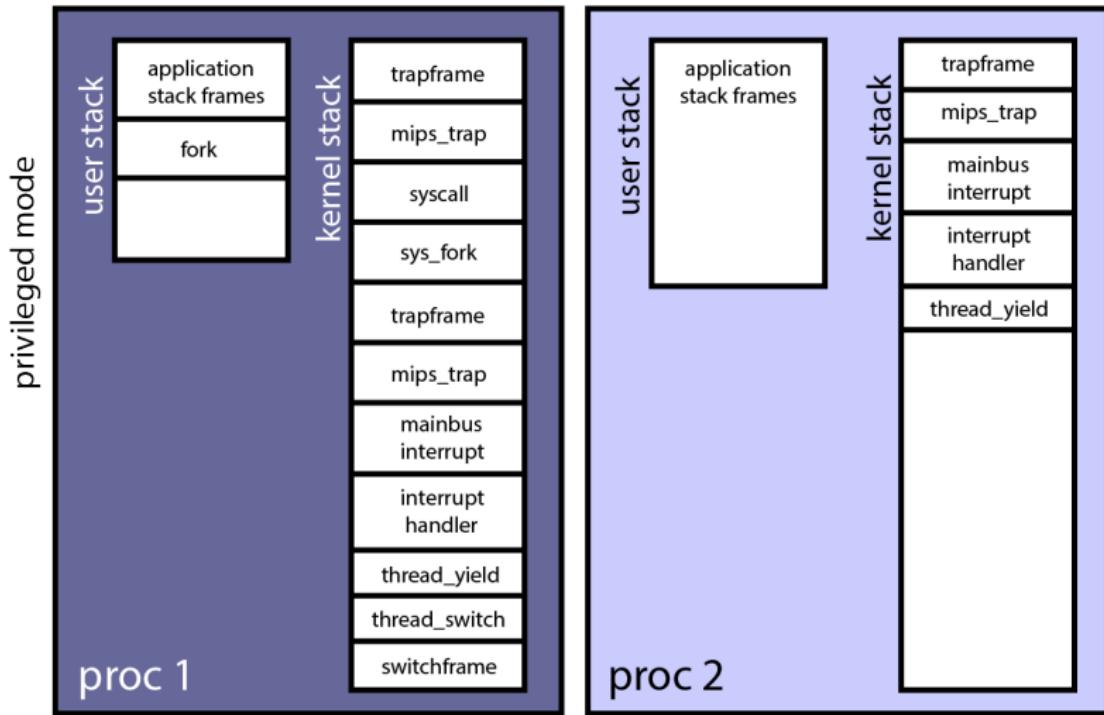
`thread_switch` calls `switchframe_switch`.

## Example: System Calls (17/27)



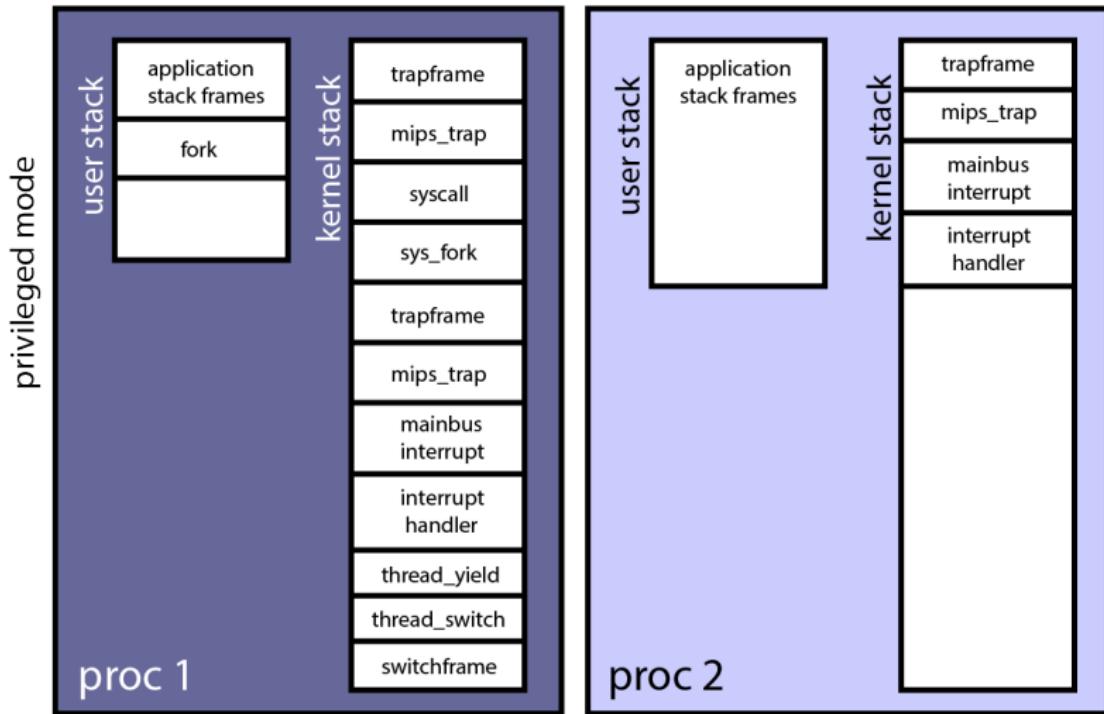
The current thread's state is saved and a context switch occurs.

## Example: System Calls (18/27)



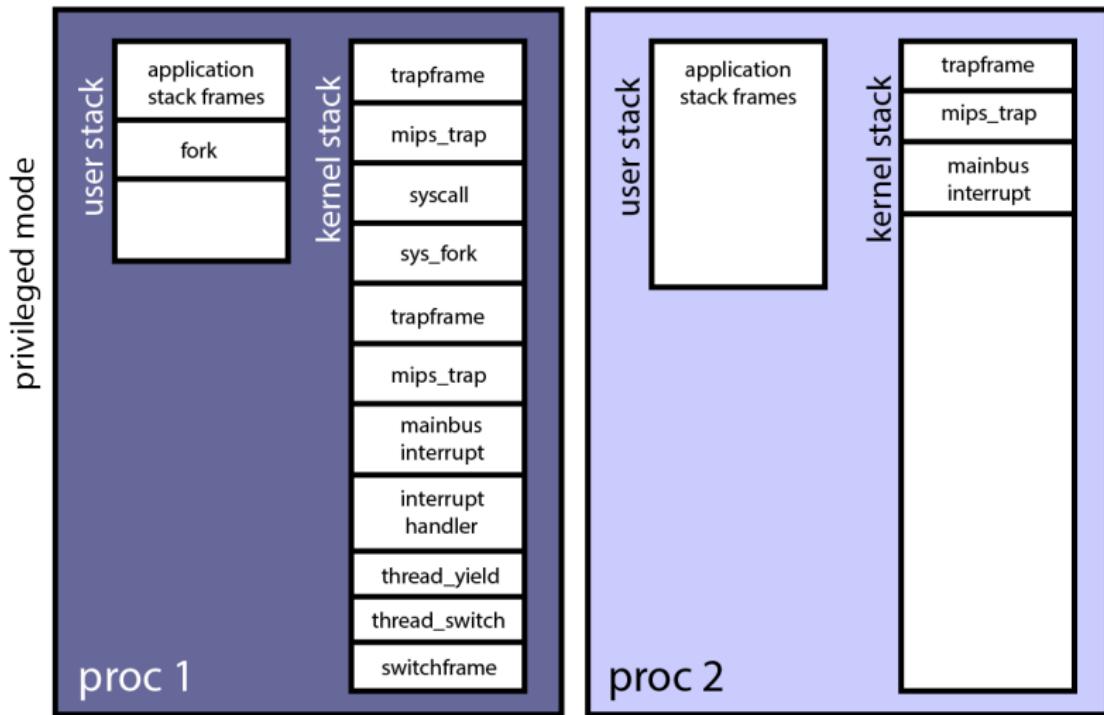
The new thread's state is restored, so return to `thread_yield`.

## Example: System Calls (19/27)



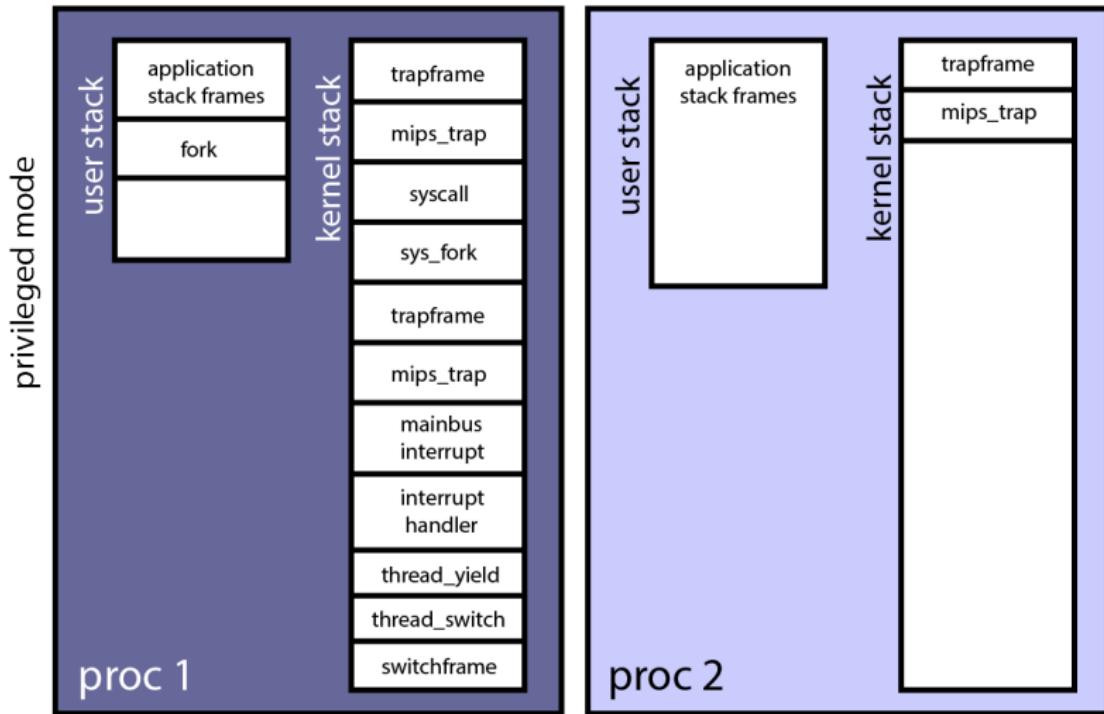
`thread_yield` returns to interrupt handler.

## Example: System Calls (20/27)



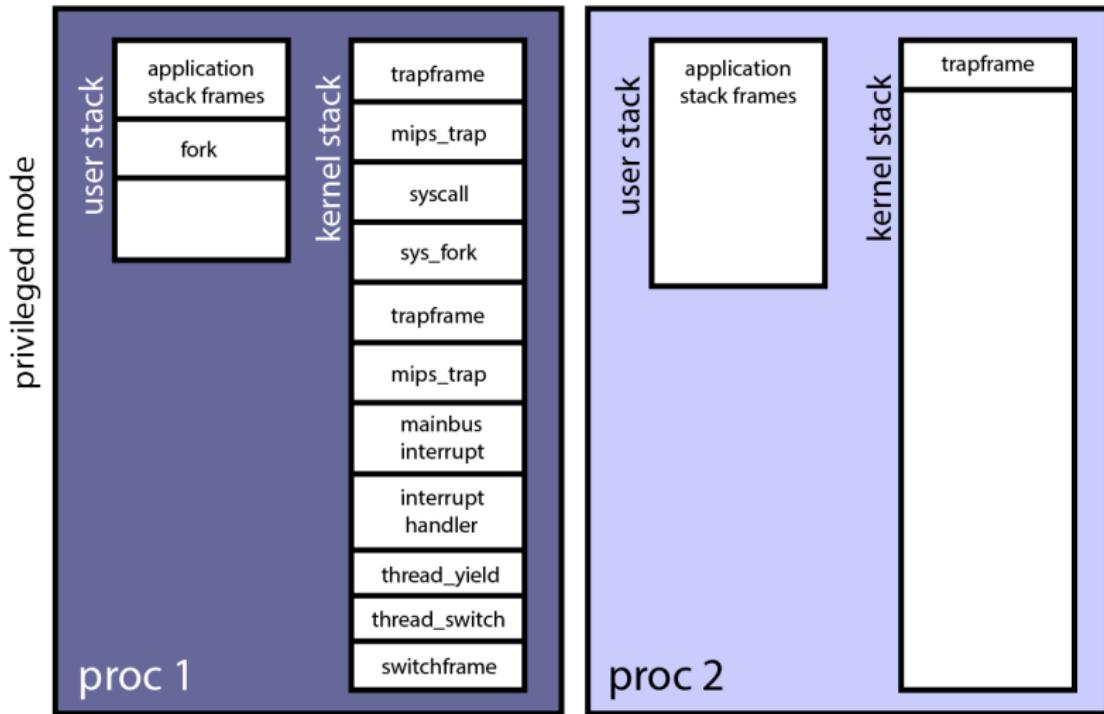
The interrupt handler returns to `mainbus_interrupt`.

## Example: System Calls (21/27)



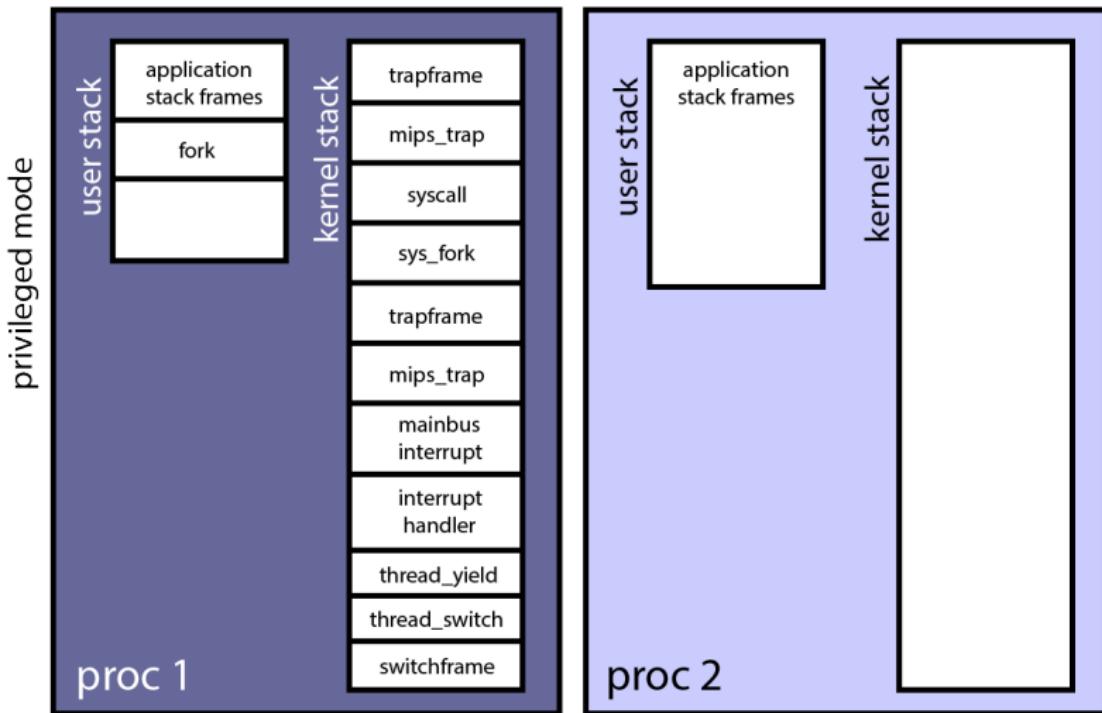
`mainbus_interrupt` returns to `mips_trap`.

## Example: System Calls (22/27)



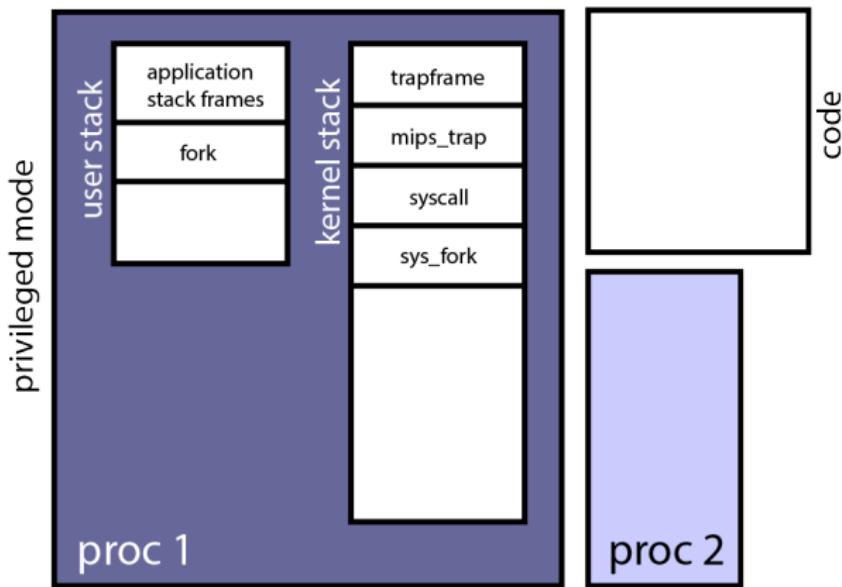
`mips_trap` returns to `common_exception`.

## Example: System Calls (23/27)



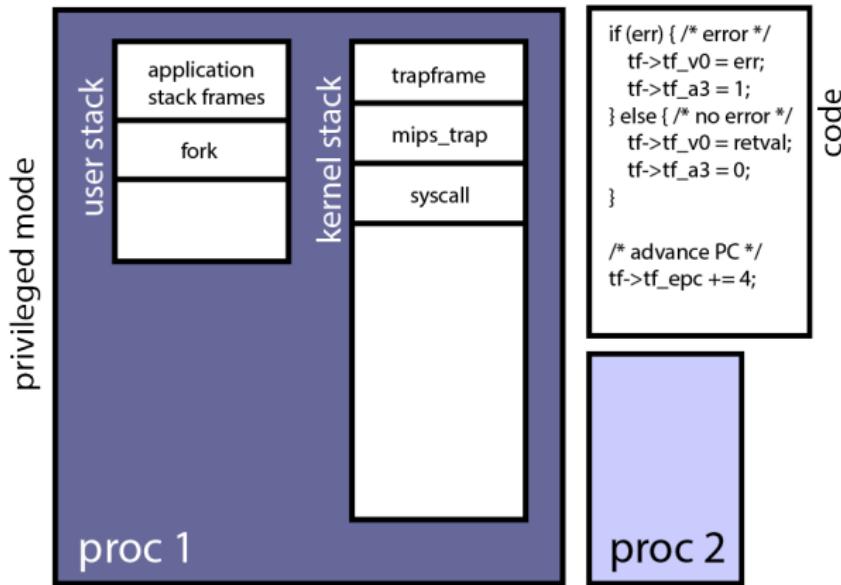
Thread context is restored from trapframe. Switch from kernel to user stacks. Switch to unprivileged mode. User code continues execution.

## Example: System Calls (24/27)



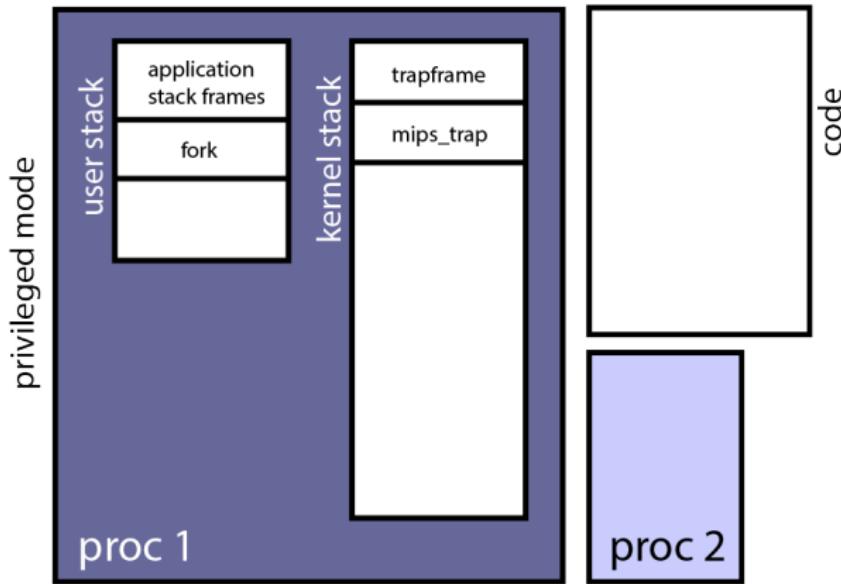
Suppose the timer interrupt (on slide 34) *did not occur* and instead the kernel finished executing `sys_fork`.

## Example: System Calls (25/27)



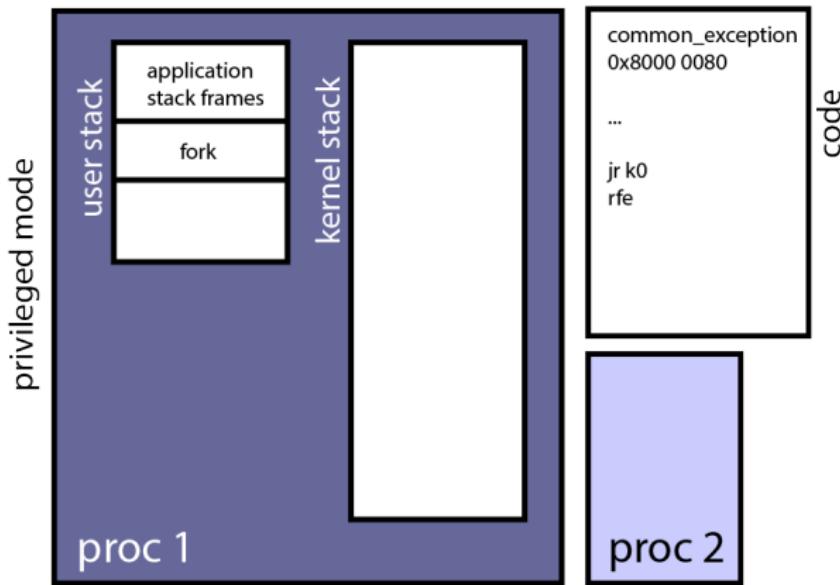
`sys_fork` returns to `syscall`. `syscall` sets up the return value/error code and result. It also increments the PC.

## Example: System Calls (26/27)



syscall returns to `mips_trap`.

## Example: System Calls (27/27)



`mips_trap` returns to `common_exception`. The trapframe data is restored to the registers. Switch from kernel to user stack and switch to unprivileged mode with (`rfe`). User code continues execution.

- The exception handler `common_exception` is in `kern/arch/mips/locore/exception-mips1.S`
- It calls `mips_trap` which is in `kern/arch/mips/locore/trap.c`
- If it is an exception `mips_trap` calls
  - `syscall` which is in `kern/arch/mips/syscall/syscall.c`
  - It calls `sys_fork` which is in `kern/syscall/proc_syscalls.c`
- If it is an interrupt, `mips_trap` calls
  - `mainbus_interrupt` which is in `kern/arch/sys161/dev/lamebus_machdep.c`

# Summary: Processes and Process Management

- A **process** is an environment in which an application program runs. [2]
- A **process's environment** includes: (one or more) threads, an address space, secondary storage and I/O devices. [2]
- A process can
  - be *created* using `fork` [4]
  - *terminate* using `_exit` [5]
  - *wait* for another process to terminate using `waitpid` [5]
  - *obtain its pid* using `getpid` [6]
  - *change the program* that is running using `execv` [6]

## Summary: System Calls

- **System calls** are the *interface* between user processes and the kernel. [10]
- The CPU implements different levels of **execution privilege** as a *security and isolation mechanism*. [11]
- The kernel runs at the highest privilege level. [11]
- **Interrupts** are *generated by devices* when they need attention. [12]
- **Exceptions** are *caused by instruction execution* when a running program needs attention. [12]
- **Interrupt handlers** and **exception handlers** are part of the kernel. [13–14]
- A *system call is implemented* by putting a **system call code** in a particular register and then raising an exception with the assembly language instruction **syscall**. The processor jumps to a fixed location, to handle the exception. [17–19]

- When an exception is raised, the *exception handler*
  1. creates trap frame to save application program state,
  2. determines that this is a system call exception,
  3. determines which system call is being requested,
  4. does the work for the requested system call,
  5. restores the application program state from the trap frame,
  6. returns from the exception. [20]
- Each OS/161 process thread has two stacks
  1. **User (Application) Stack:** used while the thread is *executing application code*. [21]
  2. **Kernel Stack:** used while *executing kernel code*, i.e. after an exception or interrupt. [21.1]

- **Multiprocessing** (or **multitasking**) means having multiple processes existing at the same time. [24]
- For multitasking: all processes must *share the available hardware resources*. [24]
- This resource sharing is coordinated by the kernel. [24]
- The OS ensures that *processes are isolated from one another*. [24.1]
- Sharing access to the CPU is achieved by performing a context switch from the thread of one process to the thread of another process. [25]

# Virtual Memory

**key concepts:** virtual memory, physical memory, address translation, MMU, TLB, relocation, paging, segmentation, executable file, swapping, page fault, locality, page replacement

Lesley Istead, Kevin Lanctot

David R. Cheriton School of Computer Science  
University of Waterloo

Winter 2019

# Virtual Memory

*Key Question for this topic:* How can we manage RAM so that

1. multiple processes can be loaded into RAM,
2. a bug in program that is being executed in one process (say a bad pointer value) cannot affect another process
3. the implementation details are largely hidden from the application programmer

*Hint:* Create a virtual environment for RAM, **virtual memory**, that looks the same for all processes.

# Virtual Memory

When running the programs that print out pointer values for the code, global variables and stack, a number of curious characteristics were observed.

- *The memory allocated to the process is quite large* approximately 128 TB of memory (0 – 7fff ffff ffff) which is more than could possibly be there physically.
- This large amount seems like a lot for our humble little process. Even a large program like Chrome only needs 400 MB of memory space.
- *The memory locations* of the code and global variables *are the same* in both process A and B.

What is going on? ⇒ There are *two types of addresses*.

# Virtual and Physical Addresses

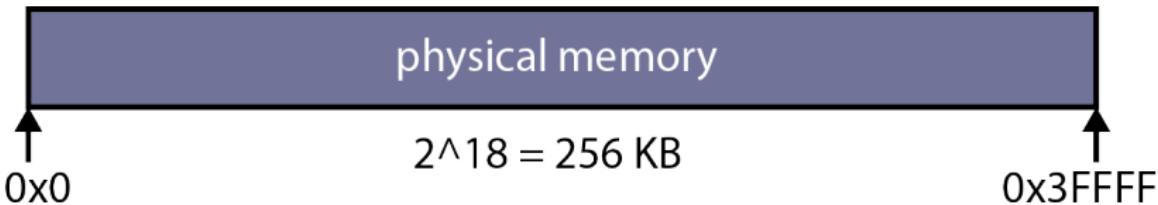
- **Physical addresses** are *provided directly by the hardware*, i.e. the amount of installed RAM.
  - One physical address space *per computer*.
  - The size of a physical address space (in bytes) determines the maximum amount of addressable physical memory (i.e. RAM).
- **Virtual addresses** (or logical addresses) are addresses *provided by the OS* to the processes.
  - One virtual address space *per process*.
- Processes use virtual addresses. As a process runs, the hardware (with help from the operating system) converts each virtual address to a physical address.
- The conversion of a virtual address to a physical address is called **address translation**.

# Why Have Virtual Addresses

- Remember from Lecture 1:  
the role of the OS is part facilitator and part cop.
- The OS provides each process with the *illusion* that it has a large amount of contiguous memory available exclusively to itself, i.e. an **address space**.
- The addresses that the processes see (e.g. stack pointer, program counter) are all *virtual addresses*.
- With processes, we *share the CPU over time* and *share the RAM in space*, i.e. different processes share the same RAM.
- The goals are
  - to *efficiently translate* between virtual and physical addresses,
  - to provide *transparency* so the programmer does not need to worry about the difference,
  - to *protect* one process's address space from other (perhaps buggy) processes.

# Physical Memory

- If it takes  $P$  bits to specify the physical address of each byte, then the maximum amount of *addressable physical memory* is  $2^P$  bytes.
  - Sys/161 MIPS CPU uses 32-bit physical addresses ( $P = 32$ ) so the maximum physical memory size is  $2^{32}$  bytes (i.e. 4GB).
  - On modern CPUs, typically  $P = 48$  so the maximum physical memory size is  $2^{48}$  bytes (i.e. 256 TB).
  - The examples in *these slides* use  $P = 18$

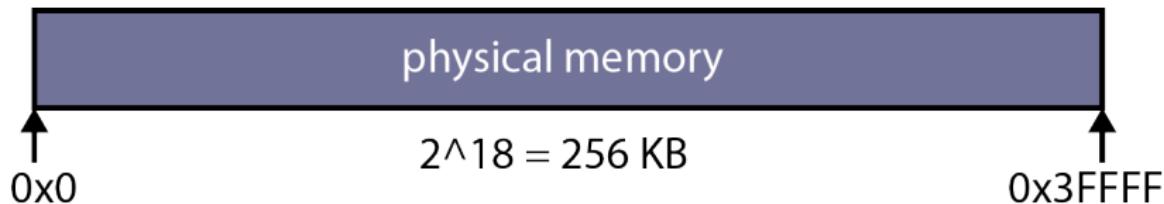
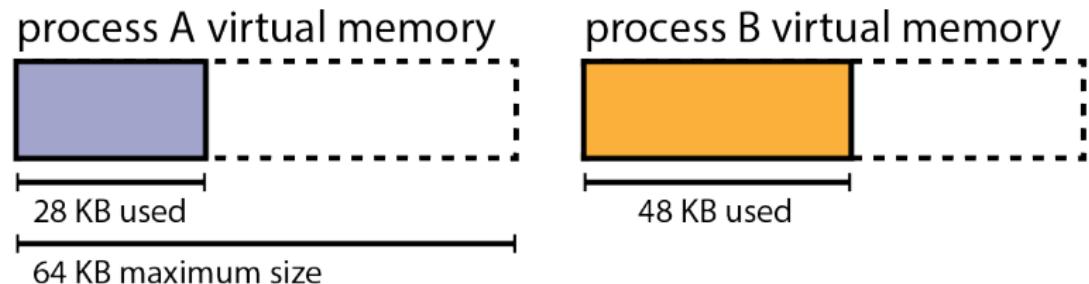


- The *actual amount* of physical memory on a machine may be less than the *maximum amount* that can be addressed.

# Virtual Memory

- The OS provides a separate, private virtual memory for each process.
- The virtual memory of a process holds the code, data, heap and stack for the program that is running in that process.
- If the virtual addresses of each byte is  $V$  bits, then the maximum amount of *addressable virtual memory* is  $2^V$  bytes.
  - For the MIPS,  $V = 32$  (and from previous slide  $P = 32$ )
  - In our examples,  $V = 16$  (and from previous slide  $P = 18$ ).
- Running applications *only see virtual addresses*, e.g.,
  - program counter and stack pointer hold virtual addresses of the next instruction and the top of the stack
  - pointers to variables are virtual addresses
  - jumps refer to virtual addresses
- Each process is isolated in its virtual memory and *cannot access another process's virtual memory*.

# Virtual Memory



Virtual addresses are 16 bits so the maximum possible virtual memory size is  $2^{16} = 64\text{KB}$ .

# Why virtual memory?

Virtual memory is used to

- *isolate* processes from each other and from the kernel,
- potentially support virtual memory that *is larger* than physical memory. I.e.
  - The total size of the virtual memory of all process can be larger than physical memory.
  - This feature allows for greater support for multiprocessing.

The concept of virtual memory dates back to a doctoral thesis in 1956. Burroughs (1961) and Atlas (1962) produced the first commercial machines with virtual memory support.

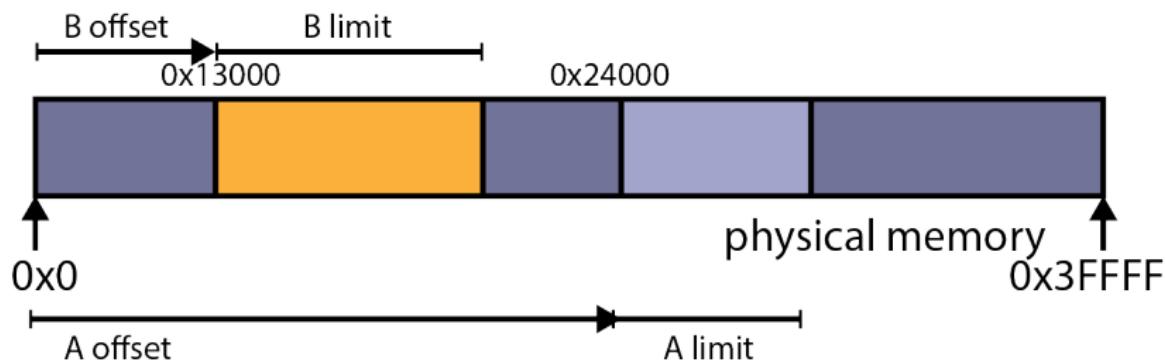
## Address Translation

- *Each virtual memory address is mapped to a different part of physical memory.*
- Since virtual memory is not real, when a process tries to access (load from or store at) a virtual address, that address is *translated* (mapped) to its corresponding physical address and the load or store is performed in physical memory.
- Address translation is performed in hardware, using information provided by the kernel.
- Even the program counter (PC) is a virtual address.
  - Each instruction requires at least one translation.
  - Hence, the translation is done in hardware, which is faster than software.

We will consider a series of five increasingly more sophisticated methods of address translation.

1. Dynamic Relocation
2. Segmented Addresses
3. Paging
4. Two-Level Paging
5. Multi-Level Paging

# 1. Dynamic Relocation



The virtual address of each process is translated using two values:

1. **offset:** addr in physical memory where the process's memory begins
2. **limit:** the amount of memory used by the process.

## 1. Dynamic Relocation

In the diagram on the previous slide, for process A

1. **offset**: 0x24000
2. **limit**: 0x7000

So process A resides in physical locations 0x24000 – 0x2AFFF.  
Recall that decimal 10 is 0xA in hexadecimal.

For process B

1. **offset**: 013000
2. **limit**: 0xC000

So process B resides in physical locations 0x13000 – 0x1EFFF.  
Recall that hexadecimal 0xC is 12 in decimal.

# 1. Address Translation for Dynamic Relocation

- The CPU includes a **memory management unit (MMU)** with a **relocation register** (i.e. the offset) and a **limit register**.
- The relocation register holds the physical offset for the running process' virtual memory and the limit register holds the size of the running process' virtual memory.
- To translate a virtual address  $v$  to a physical address  $p$ , the MMU does the following

```
if ( $v \geq \text{limit}$ )
then
    raise memory exception
else
     $p \leftarrow v + \text{offset}$ 
```

- The kernel maintains separate relocation and limit values for each process and changes these values in the MMU registers when there is a context switch *between processes*.

# 1. Properties of Dynamic Relocation

- In this simple version of virtual memory, each virtual address space corresponds to a *contiguous range of physical addresses*.
- The kernel is responsible for deciding where each virtual address space should map in physical memory.
  - The OS must track which parts of physical memory are in use and which parts are free.
  - Since different address spaces may have different sizes, the OS must allocate/deallocate variable-sized chunks of physical memory.
  - This creates the potential for *fragmentation of physical memory*.

## Fragmentation Example

A system has 100MB of free, but not contiguous, physical memory. A program requires 100MB of memory for its address space. Although the space is available the program cannot be loaded.

# 1. Dynamic Relocation Example

Process A	Process B
Limit Register: 0x0000 7000	Limit Register: 0x0000 C000
Relocation Register: 0x0002 4000	Relocation Register: 0x0001 3000
$v = 0x0000 \quad p = ?$	$v = 0x0000 \quad p = ?$
$v = 0x102C \quad p = ?$	$v = 0x102C \quad p = ?$
$v = 0x8000 \quad p = ?$	$v = 0x8000 \quad p = ?$

Recall

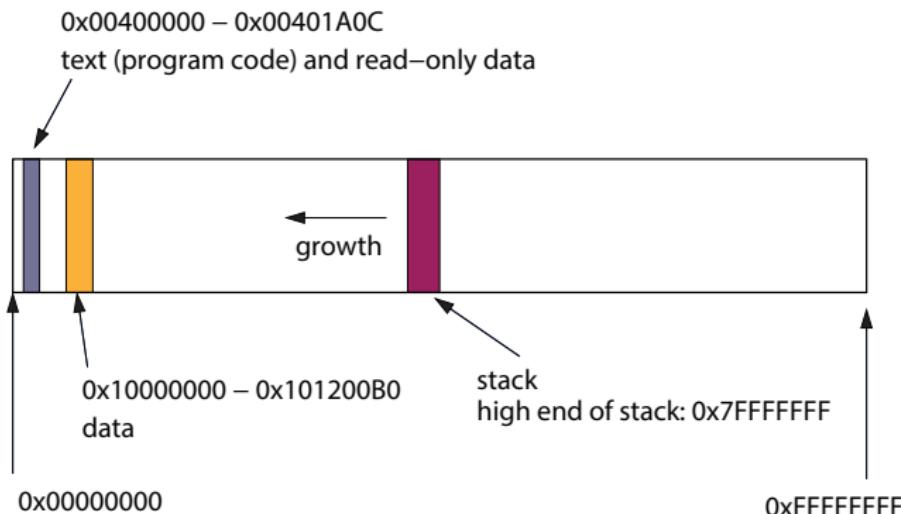
Addresses that cannot be translated produce *exceptions*.

# 1. Dynamic Relocation Properties

- Using just two registers per process, dynamic relocation
  1. allows multiple processes to *share RAM* and
  2. *isolates each process' address space* from all other processes.
- For example, if process B had an error in a pointer value,  
 $p = 0x12000;$   
Then writing to that virtual address  
 $*p = 350;$   
would be writing to physical address  $0x13000 + 0x12000 = 0x25000$   
which is in process A's address space.
- But since each use of an address (e.g. for load or store) is checked against that process's limit register, and  $0x12000$  exceeds process B's limit register ( $0xC000$ ), a memory exception would be raised and that error would be reported.

But dynamic relocation *does not use physical memory efficiently...*

## 2. A More Realistic Virtual Memory



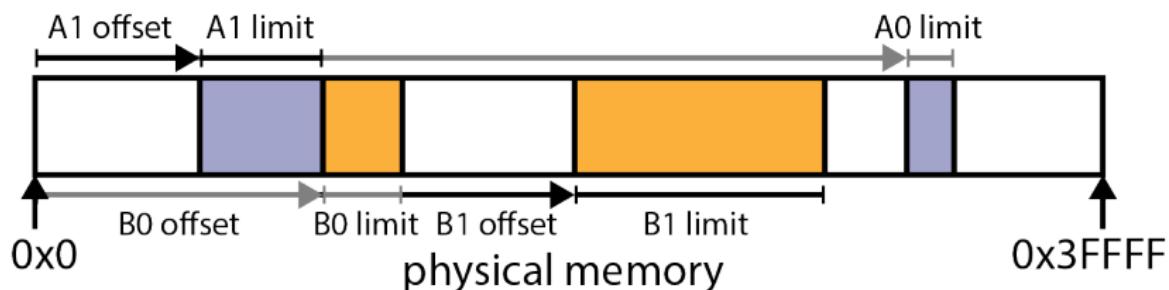
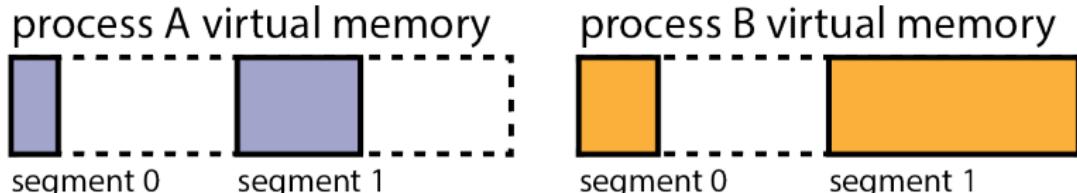
The OS/161 virtual address space for the application `sort`.

- Note that the address space only requires 1.2MB of space but dynamic relocation would require 2GB of space for `sort`.
- *Key Observation:* Virtual memory may be large yet the process's address space may be very small and discontiguous.

## 2. Segmentation

- **Key Idea:** Instead of mapping the entire virtual memory to physical memory, we map each **segment** of the address space separately.
- The kernel maintains an offset and limit value *for each segment*.
- With segmentation, a virtual address can be thought of as having two parts: (segment ID, offset within segment)
- With  $K$  bits for the segment ID, we can have up to:
  - $2^K$  segments
  - $2^{V-K}$  bytes per segment
- E.g. if there are 4 segments, then  $\log(4) = 2$  bits are required to represent the segment number and the maximum size of each segment is  $2^{V-2}$  bytes
- The kernel decides where each segment is placed in physical memory.
- Fragmentation of physical memory is still possible.

## 2. Segmented Address Space Diagram



While the segments for process A and B are separate in virtual memory (top), they mixed together in physical memory (bottom).

## 2. Translating Segmented Virtual Addresses

Many different approaches for translating segmented virtual addresses.

- Approach 2a): For each segment  $i$ , the MMU has
  - a relocation offset register: `offset[i]` and
  - a limit register: `limit[i]`
- To translate virtual address  $v$  to a physical address  $p$ :

```
s ← SegmentNumber(v)
a ← AddressWithinSegment(v)
if (a ≥ limit[s])
then
    raise memory exception
else
    p ← a + offset[s]
```

- As with dynamic relocation, the kernel maintains a separate set of relocation offsets and limits for each process and changes the values in the MMU when there is a context switch between processes.

## 2. Segmented Address Translation Example A

**Process A**

Segment	Limit Register	Relocation Register
0	0x2000	0x38000
1	0x5000	0x10000

**Process B**

Segment	Limit Register	Relocation Register
0	0x3000	0x15000
1	0xB000	0x22000

Translate the following for process A and B:

v	Segment	Offset	Physical Address
0x1240			
0xA0A0			
0x66AC			
0xE880			

## 2. Segmented Address Translation Example A

Split the virtual address into two parts:

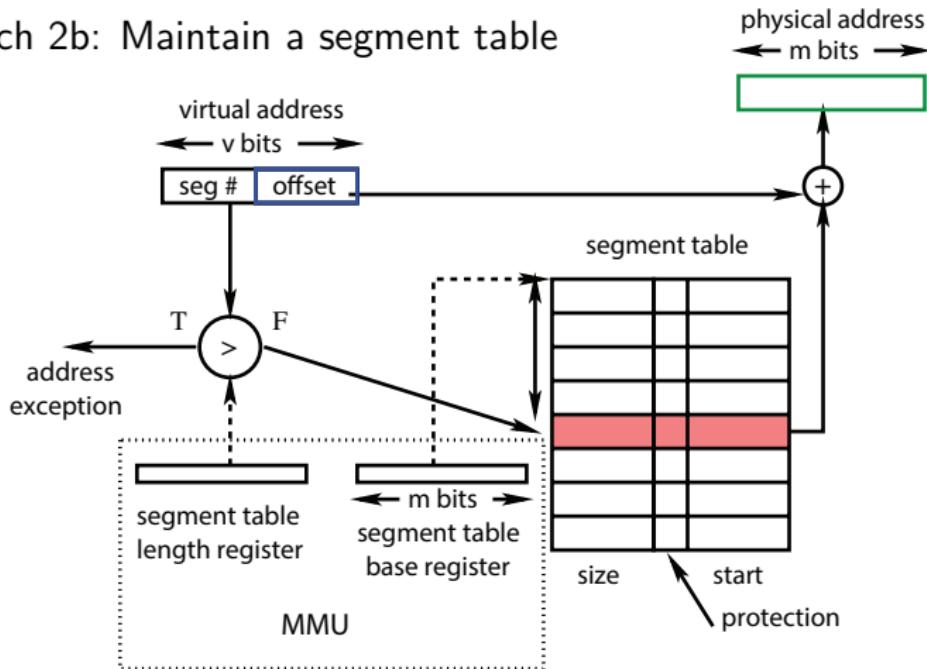
1. The first bit of the virtual address specifies the segment.
2. The next *three bits* specifies the *first hexdigit* of the offset and the rest the virtual address specifies the rest of the offset.

The first hex digit of the virtual address

- 0x1240 is 1 which is 0001 in binary, so the address is in segment 0. Since the binary value 001 is 1 in hex, the offset is 1240.
- 0xA0A0 is A which is 1010 in binary, so the address is in segment 1. Since the binary value 010 is 2 in hex, the offset is 20A0.
- 0x66AC is 6 which is 0110 in binary, so the address is in segment 0. Since the binary value 110 is 6 in hex, the offset is 66AC.
- 0xE880 is E which is 1110 in binary, so the address is in segment 1. Since the binary value 110 is 6 in hex, the offset is 6880.

## 2. Translating Segmented Virtual Addresses

- Approach 2b: Maintain a segment table



- If the segment number in  $v$  is greater than the number of segments then throw an exception else use the segment number to lookup the limit and relocation values from the segment table.

## 2. Segmented Address Translation Example B

Virtual addr = 32 bits, Physical addr = 32 bits, Offset = 28 bits

Segment Table base reg 0x0010 0000

Segment Table len reg 0x0000 0004

Seg	Size	Prot	Start
0	0x6000	X-	0x7 0000
1	0x1000	-W	0x6 0000
2	0xC000	-W	0x5 0000
3	0xB000	-W	0x8 0000

Virtual address 0x0000 2004

Physical address = ----- ?

Virtual address 0x2000 31A4

Physical address = ----- ?

## 2. Segmented Address Translation Example B

1. *Split* the virtual address 0x0000 2004 into the
  - segment number 0x0 and
  - the offset 0x000 2004.
2. *Check* that the segment number is not too large:  $0x0 < 0x4$ .
  - E.g. the virtual address 0x4000 2004 would have a segment number that is too large.
3. *Check* that the offset is not too large:  $0x000 2004 < 0x6000$ .
  - E.g. the virtual address 0x0000 6004 would have an offset that is too large.
4. *Translate* the segment number into a physical base address:  
 $\text{SegmentTable}[0x0] = 0x0007 0000$ .
5. *Add* the offset to physical base address to get the physical address:  
 $0x0007 0000 + 0x000 2004 = 0x0007 2004$ .

## 2. Segmented Address Translation Example B

1. *Split* the virtual address 0x2000 31A4 into the
  - segment number 0x2 and
  - the offset 0x000 31A4.
2. *Check* that the segment number is not too large:  $0x2 < 0x4$ .
3. *Check* that the offset is not too large:  $0x000 31A4 < 0xC000$ .
4. *Translate* the segment number into a physical base address:  
 $\text{SegmentTable}[0x2] = 0x0005 0000$ .
5. *Add* the offset to physical base address to get the physical address:  
 $0x0005 0000 + 0x000 31A4 = 0x0005 31A4$ .

# Address Translation: Mini Review

Recall: We are considering a series of five increasingly more sophisticated methods of address translation.

## 1. *Dynamic Relocation*

Benefit: processes share RAM and the OS provides isolation

Limitation: address spaces can be sparse and so physical memory can be wasted, so we introduced...

## 2. *Segmented Addresses*

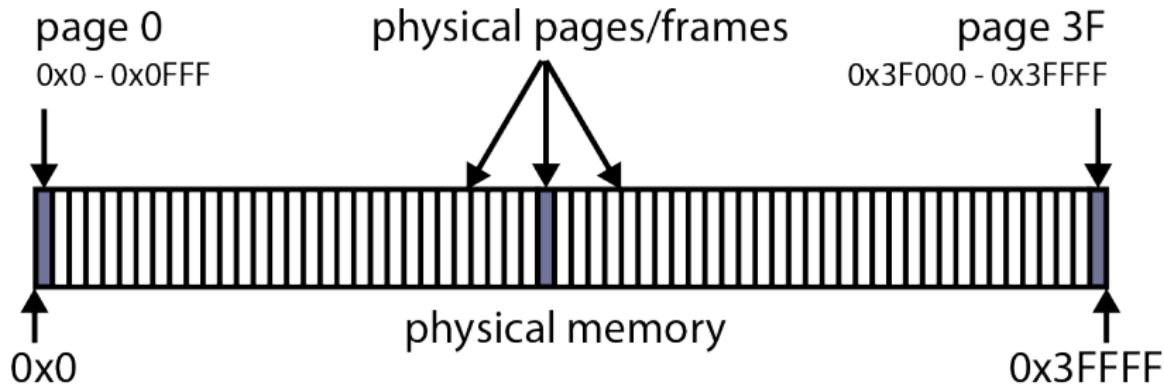
Limitation: space for heap and stack is cannot grow beyond a certain point, even though there is more memory available or space between various segment gets wasted because it is too small to fit anything (external fragmentation). So we will introduce ...

## 3. *Paging*

### 4. Two-Level Paging

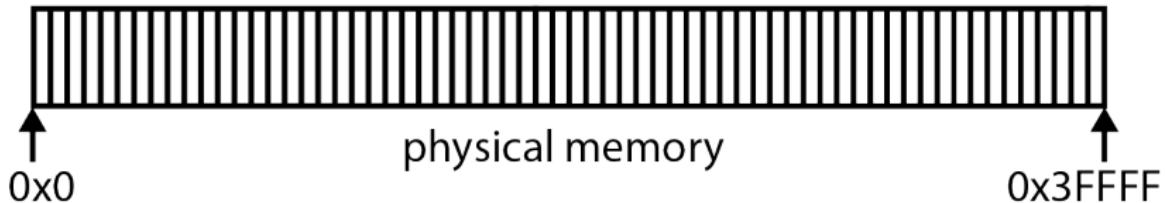
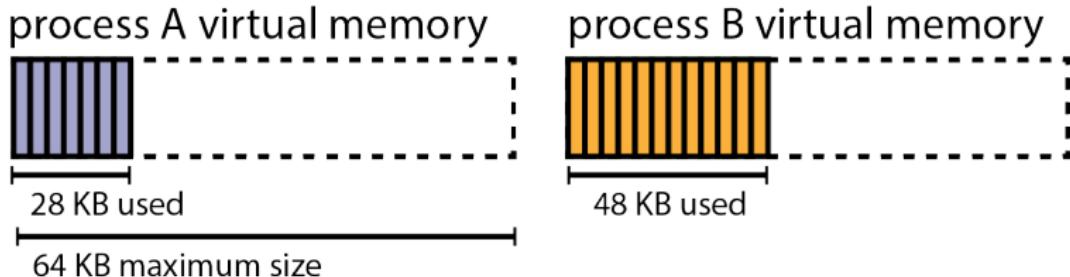
### 5. Multi-Level Paging

### 3. Paging: Physical Memory



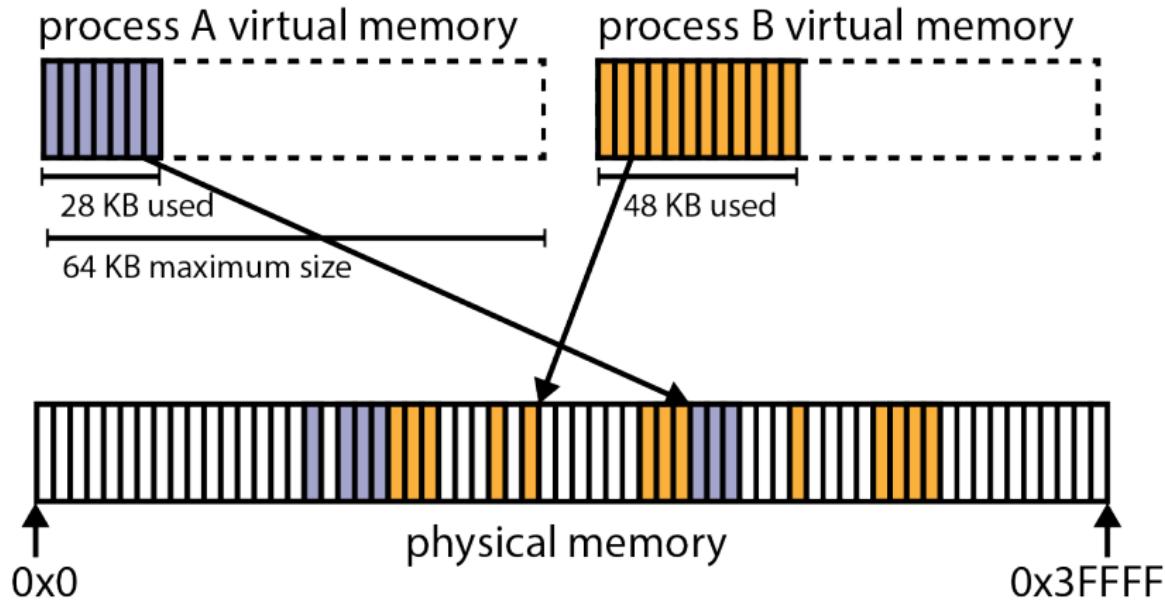
- *Key Idea:* Divide *Physical memory* into fixed-size chunks called **frames** (a.k.a. **physical pages**).
- In this example
  - physical addresses are 18 bits,
  - the size of physical memory is 256 KB,
  - The frame size is 4KB, so...
  - physical memory consists of 64 (0x40) frames.

### 3. Paging: Virtual Memory



- Divide *virtual memory* into fixed-sized chunks called **pages**.
- Page size (virtual memory) equals the frame size (physical memory).
- Virtual addresses, in this example, are 16 bits (for a max size of 64 KB) and so a process can use up to 16 pages.

### 3. Paging: Address Translation



- Each page (in virtual memory) maps to a different frame (in physical memory).
- Any page can map to any frame  $\Rightarrow$  we will need a method to track this mapping, i.e. **page tables**.

### 3. Page Tables Example

Process A Page Table

Page	Frame	Valid?
0x0	0x0F	1
0x1	0x26	1
0x2	0x27	1
0x3	0x28	1
0x4	0x11	1
0x5	0x12	1
0x6	0x13	1
0x7	0x00	0
0x8	0x00	0
...	...	...
0xF	0x00	0

Process B Page Table

Page	Frame	Valid?
0x0	0x14	1
0x1	0x15	1
0x2	0x16	1
0x3	0x23	1
...	...	...
0xA	0x33	1
0xB	0x2C	1
0xC	0x00	0
0xD	0x00	0
0xE	0x00	0
0xF	0x00	0

- Each process has its own **page table**.
- The page table maps pages (from virtual memory) to frames (in physical memory).
- Each row of a page table is called a **page table entry (PTE)**.

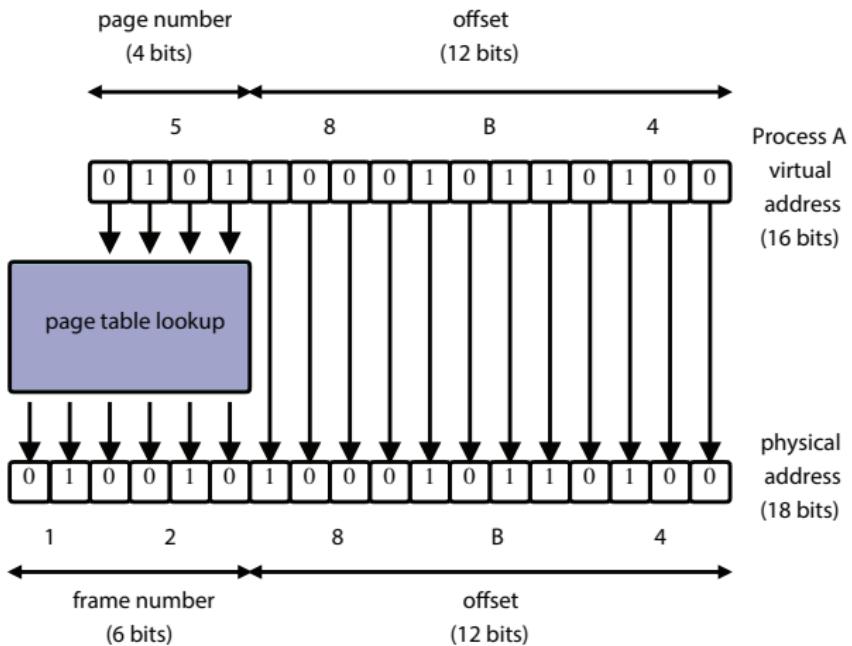
### 3. Page Tables Example

- For a small process, *not all available pages are used*.
- E.g. say Process A is using 28 KB of memory then it will only need 7 pages.
- The **valid bit** is used to indicate if the PTE is used or not.
  - If it is 1, then the PTE maps that page to physical memory.
  - If it is 0, the PTE does not correspond to a page in virtual memory.
- In Process A's table, pages
  - 0x0 to 0x6 maps onto frames (since their valid bit is 1).
  - 0x7 to 0xF do not map onto a frame yet (since the valid bit is 0).
- Number of PTEs = Maximum Virtual Memory Size / Page Size
  - In our example  $16 = 64\text{ KB} / 4\text{ KB}$ .

### 3. Paging: Address Translation in the MMU

- The MMU includes a **page table base register** which points to the page table for the current process.
- To translate a virtual address the MMU
  1. *determine* the **page number** and **offset** of the virtual address
    - page number is the virtual address divided by the page size
    - offset is the virtual address modulo the page size
  2. *look up* the page's entry (PTE) in the current process page table, using the page number
  3. if the PTE is *not valid*, raise an exception
  4. otherwise, *combine* the corresponding frame number (from the PTE) with the offset to determine the physical address
    - physical address = (frame number  $\times$  frame size) + offset

### 3. Paging: Address Translation Illustrated



- Number of Bits for Offset =  $\log(\text{Page Size}) = \log(4096) = 12$
- Number of PTEs = Maximum Virtual Memory Size / Page Size
- Number of Bits for Page Number =  $\log(\text{Number of PTEs})$

### 3. Paging: Address Translation Example

Process A Page Table		
Page	Frame	Valid?
0x0	0x0F	1
0x1	0x26	1
0x2	0x27	1
0x3	0x28	1
0x4	0x11	1
0x5	0x12	1
0x6	0x13	1
0x7	0x00	0
0x8	0x00	0
...	...	...
0xF	0x00	0

Process B Page Table		
Page	Frame	Valid?
0x0	0x14	1
0x1	0x15	1
0x2	0x16	1
...	...	...
0x9	0x32	1
0xA	0x33	1
0xB	0x2C	1
0xC	0x00	0
0xD	0x00	0
0xE	0x00	0
0xF	0x00	0

Exercise: Translate the following virtual addresses

Virtual Address	Process A	Process B
v = 0x102C	p =	p =
v = 0x9800	p =	p =
v = 0x0024	p =	p =

### 3. Paging: Address Translation Example

1. Determine (i.e. break address up into) page number and offset.
  - Here the page number is 1 hex digit and the offset is 3 hex digits.
  - E.g. for 0x102C it would be 1 02C
2. Look up the page's entry (PTE) in the current process page table, using the page number.
  - E.g. For Process A, the PTE for 1 is Frame = 26 and Valid = True
3. If the PTE is not valid, raise an exception.
  - It is valid.
4. Otherwise, combine the corresponding frame number (from the PTE) with the offset.
  - Combine 26 with 02C to get 2602C.

The physical address 0x102C is 0x2602C.

### 3. Paging: Address Translation Example

1. Determine (i.e. break address up into) **page number** and **offset**.
  - Again, the page number is 1 hex digit and the offset is 3 hex digits.
  - E.g. for 0x9800 it would be **9** **800**
2. Look up the page's entry (PTE) in the current process page table, using the **page number**.
  - E.g. For Process A, the PTE for **9** is **Frame = ???** and **Valid = False**
3. If the PTE is **not valid**, raise an exception.
  - It is **not valid**  $\Rightarrow$  Segmentation Violation.

*Key Point:* The valid pages are contiguous in the page table and come first. As the process uses more physical memory, more pages are used.

### 3. Other Information Found in PTEs

- *Key Point:* PTEs may contain other fields, in addition to the frame number and valid bit
- Example 1: *write protection bit*
  - can be set by the kernel to indicate that a page is read-only
  - if a write operation (e.g., MIPS `sw`) uses a virtual address on a read-only page, the MMU will raise an exception when it translates the virtual address
- Example 2: *bits to track page usage*
  - reference (use) bit: has the process used this page recently?
  - dirty bit: have contents of this page been changed?
  - these bits are set by the MMU and read by the kernel
  - more on this topic later!

### 3. Page Tables: How Big?

- *Key Point:* A page table has one PTE for each page in the virtual memory.
- Page Table Size = (Number of Pages) × (Size of PTE)
- Recall: Number of Pages = Maximum Virtual Mem Size / Page Size
- The Size of a PTE is typically provided.
  
- Example: What is the page table size for 64KB of virtual memory with 4KB pages where each PTE is 32 bits (i.e. 4 bytes)?
  - Number of Pages =  $64\text{ KB} / 4\text{ KB} = 16\text{ pages}$
  - Page Table Size =  $16 \times 4\text{ bytes} = 64\text{ bytes}$ .
  - Therefore the pages table is 64 bytes.

### 3. Page Tables: How Big?

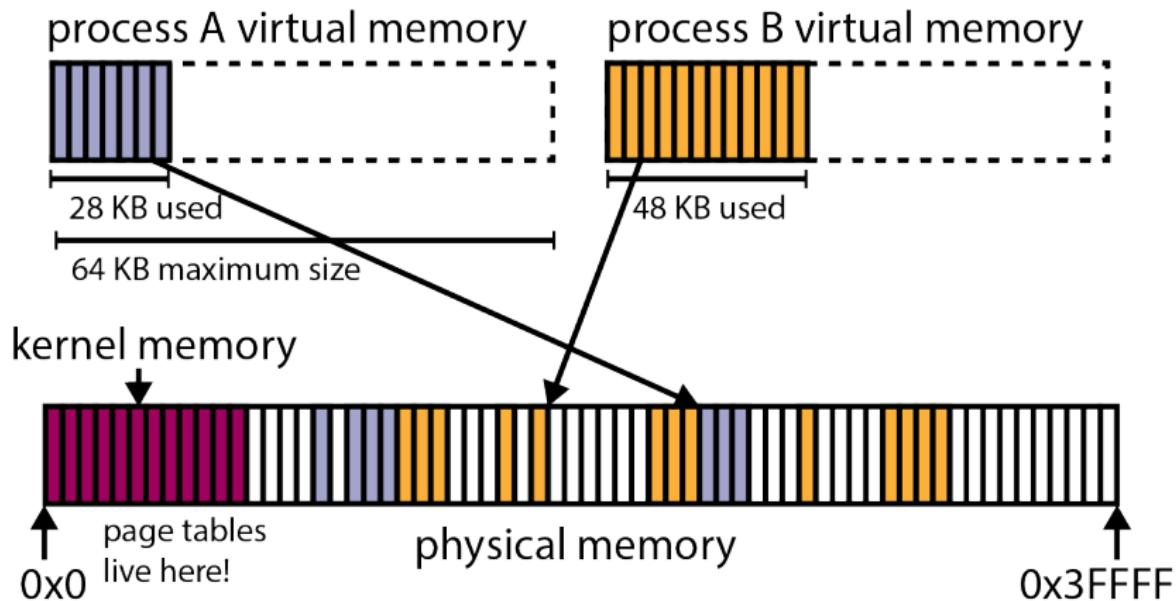
Larger Virtual Memory (i.e. 32-bit architecture, common 10 years ago)

- If  $V = 32$  bits (i.e. the size of a virtual memory address)  
⇒ the maximum virtual memory size is  $2^{32} = 4\text{GB}$ .
- Assuming the page size is 4KB and PTE size is 32 bits (4 bytes)  
⇒ the number of pages (and PTEs) would be  $2^{32}/2^{12} = 2^{20}$   
⇒ the size of the page table would be  $2^{20} \times 4$  bytes = 4 MB.

Larger Virtual Memory (i.e. current 64-bit architecture)

- If  $V = 48$  bits (i.e. the size of a virtual memory address)  
⇒ the maximum virtual memory size is  $2^{48} = 2^8 \text{ TB} = 256 \text{ TB}$
- Assuming the page size is 4KB and PTE size is 32 bits (4 bytes)  
⇒ the number of pages (and PTEs) would be  $2^{48}/2^{12} = 2^{36}$   
⇒ the size of the page table would be  $2^{36} \times 4$  bytes = 256 GB.
- Conclusion: Page tables can get very, very large.

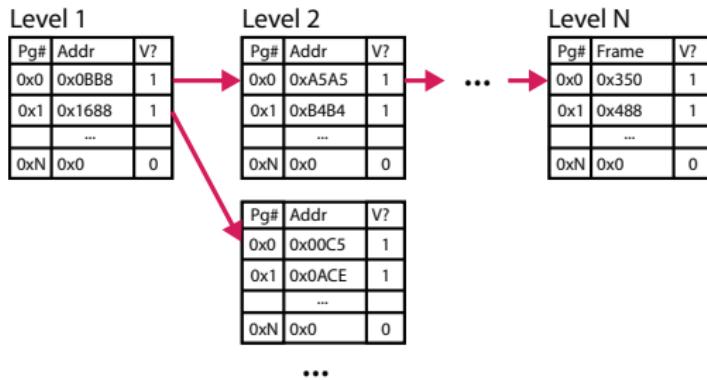
### 3. Page Tables: Where?



- Page tables are kernel data structures.
- They live in the kernel's memory.
- If  $P = 48$  and  $V = 48$ , how many page tables can fit into the kernel's memory?

# Shrinking the Page Table: Multi-Level Paging

- Instead of having a single page table to map an entire virtual memory, we can organize it and *split the page table into multiple levels*.
  - a large, contiguous table is replaced with multiple smaller tables, each fitting onto a single page
  - if a table contains no valid PTEs, do not create that table*



- The lowest-level page table ( $N$ ), contains the frame number.
- All higher level tables contain pointers to tables on the next level.

# Two-Level Paging Example (Process A)

Single-Level Paging

Page	Frame	V?
0x0	0x0F	1
0x1	0x26	1
0x2	0x27	1
0x3	0x28	1
0x4	0x11	1
0x5	0x12	1
0x6	0x13	1
0x7	0x00	0
0x8	0x00	0
...	...	...
0xE	0x00	0
0xF	0x00	0

Two-Level Paging

Directory

Page	Address	V?
0x0	Table 1	1
0x1	Table 2	1
0x2	NULL	0
0x3	NULL	0

Table 1

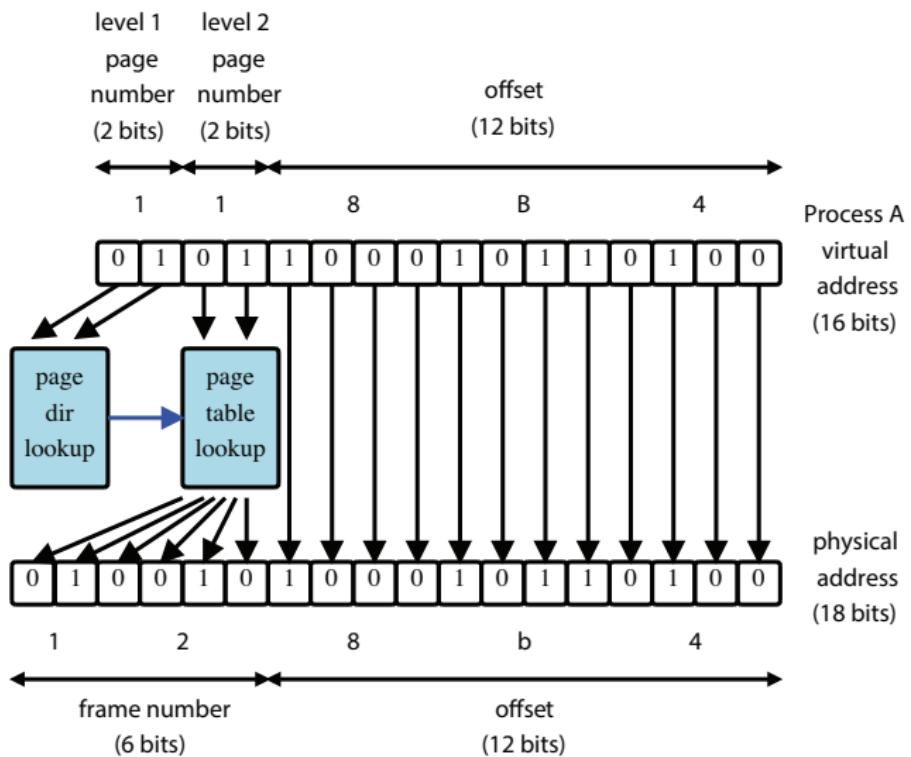
Page	Frame	V?
0x0	0x0F	1
0x1	0x26	1
0x2	0x27	1
0x3	0x28	1

Table 2

Page	Frame	V?
0x0	0x11	1
0x1	0x12	1
0x2	0x13	1
0x3	NULL	0

- V? means “Is this entry valid?”
- If a PTE is not valid, it does not matter what the frame or address is.
- The *address translation is the same* as single-level paging  
(i.e. Physical Address = Frame Number × Page Size + offset)  
but the *lookup is different*.

# Two-Level Paging: Address Translation



Translating virtual address 0x58B4 to physical address 0x128B4.

# Multi-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 case there are 2 levels, each page number is 2 bits.
- So in this case the virtual address  $v$  has 3 parts:  $p_1$   $p_2$  offset
- The first hex digit 0x5 is 0101 in binary.
- So 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.

## Multi-Level Paging: Address Translation

- The MMU's **page table base register** points to the page table directory for the current process.
- Each virtual address  $v$  has  $n$  parts:  $(p_1, p_2, \dots, p_n, o)$
- How the MMU translates a virtual address:
  1. index into the *page table directory* using  $p_1$  to get a pointer to a 2nd level page table
  2. if the directory entry is not valid, raise an exception
  3. index into the *2nd level page table* using  $p_2$  to find a pointer to a 3rd level page table
  4. if the entry is not valid, raise an exception
  5. ...
  6. index into the *n-th level page table* using  $p_n$  to find a PTE for the page being accessed
  7. if the PTE is not valid, raise an exception
  8. otherwise, combine the frame number from the PTE with  $o$  to determine the physical address (as for single-level paging)

# How Many Levels?

- *Goal of multi-level paging* is to reduce the size of individual page tables.
- *Key idea:* have each table fit on a single page (think B-trees).
- As  $V$  increases, so does the need for more levels, e.g.
  - if  $V = 40$  (i.e. a 40-bit virtual addresses),
  - the page size = 4KB ( $2^{12}$ ) and
  - the PTE size = 4 bytes then ...
    - *# of pages (and PTEs) needed* for this virtual memory size is  
(virtual memory size) / (page size)  
 $= 2^{40}/2^{12} = 2^{28}$  pages.
    - *# of PTE's that can be stored on each page* is  
(page size)/(PTE size)  
 $= 2^{12}/2^2 = 2^{10}$  PTEs.

# How Many Levels?

- How many page tables are needed to store all  $2^{28}$  PTEs?
  - the **# page tables needed** is:  
$$(\# \text{ of PTEs}) / (\text{PTE's per page})$$
$$= 2^{28} / 2^{10} = 2^{18} \text{ page tables.}$$
- So the top level page table (called the **directory**)
  - must hold  $2^{18}$  references to page tables
  - requires  $2^{18} \times 2^2 = 2^{20}$  or 1MB of space.
- **Key Point:** When the number of entries required in the directory is so large (in this case 1 MB) that they no longer fit on a page (in this case 4 KB)  $\Rightarrow$  add more levels to map larger virtual memories efficiently.

# Summary: Roles of the Kernel and the MMU

Managing virtual memory requires both software and hardware.

- *Kernel (software):*
  - Manages MMU registers on address space switches (context switch from thread in one process to thread in a different process)
  - Creates and manages page tables
  - Manages (allocate/deallocate) physical memory
  - Handles exceptions raised by the MMU
- *MMU (hardware):*
  - Translates virtual addresses to physical addresses
  - Checks for and raises exceptions when necessary

- *Keep in mind:* Each assembly instruction requires a minimum of one memory operation
  - one to fetch instruction, one or more for instruction operands
- Address translation through a page table adds a *minimum of one* extra memory operation (for page table entry lookup) for each memory operation performed during instruction execution.
- This extra step can be slow!
- Solution: include a **Translation Lookaside Buffer (TLB)** (a cache for PTEs) in the MMU
  - TLB is a small, fast, dedicated cache of address translations in the MMU
  - Each TLB entry stores a single ( $\text{page}\# \rightarrow \text{frame}\#$ ) mapping.
  - I.e. store the pair  $(p, f)$  which means translate  $p$  to  $f$

# TLB Use

- *Question:*  
How does the MMU does to translate a virtual address on page p?
- There are two types of TLBs: **Hardware-Managed TLBs** and **Software-Managed TLBs**
- *Algorithm for a Hardware-Managed TLB:*  

```
if (p has an entry (p,f) in the TLB) then
    return f      // TLB hit!
else
    find p's frame number (f) from the page table
    add (p,f) to the TLB, evicting another entry if full
    return f      // TLB miss
```
- If the MMU cannot distinguish TLB entries from different address spaces, then the kernel must clear or invalidate the TLB on each context switch from one process to another.

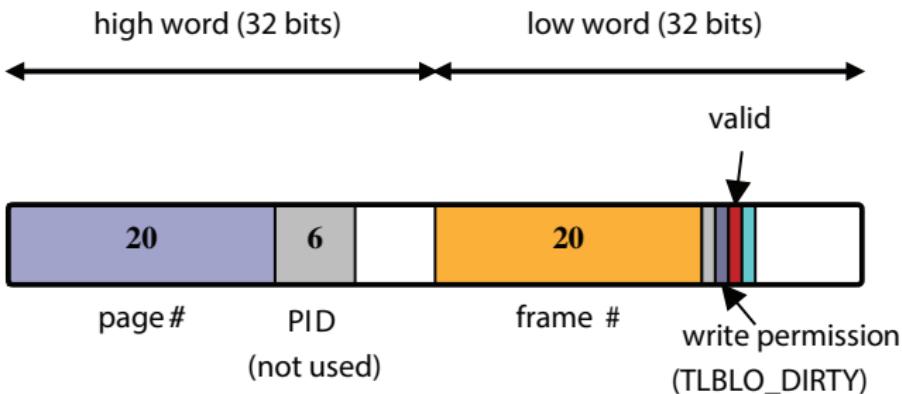
## TLB Use

- For a *hardware-managed TLB* the MMU handles TLB misses, including page table lookup and replacement of TLB entries.  
⇒ MMU must understand the kernel's page table format.
- MIPS has a *software-managed TLB*, which translates a virtual address on page  $p$  like this:

```
if (p has an entry (p,f) in the TLB) then
    return f           // TLB hit!
else
    raise exception   // TLB miss
```

- In case of a TLB miss, the kernel must
  1. determine the frame number for  $p$
  2. add  $(p,f)$  to the TLB, evicting another entry if necessary
- After the miss is handled, the instruction that caused the exception is re-tried

# The MIPS R3000 TLB



- The MIPS TLB has room for 64 entries.
  - Each entry is 64 bits (8 bytes) long, as shown.
  - If TLBLO\_DIRTY (a.k.a. write permission) is set (i.e. is 1) it means that you can write to this page.
  - See `kern/arch/mips/include/tlb.h`

# The MIPS R3000 TLB

For OS/161's Address Space

- virtual and physical addresses are 32 bits,
- the page size is 4KB (requiring 12 bits) so
  - both frame number and page number are 20 (i.e. 32-12) bits each.

For the TLB

- The PID field can be used to distinguish mappings from different processes, but OS/161 does not use it.
- The valid bit indicates that the mapping in the TLB entry is valid (otherwise the TLB entry is ignored).
- The write permission bit indicates whether the mapped page can be written to. If this TLBL\_DIRTY is clear (i.e. 0) an attempt to translate an address on this page for a store instruction (e.g. sw or sb) will result in an exception.

# Paging - Conclusion

## Benefits

- paging does not introduce external fragmentation
- multi-level paging reduces the amount of memory required to store page-to-frame mappings

## Costs

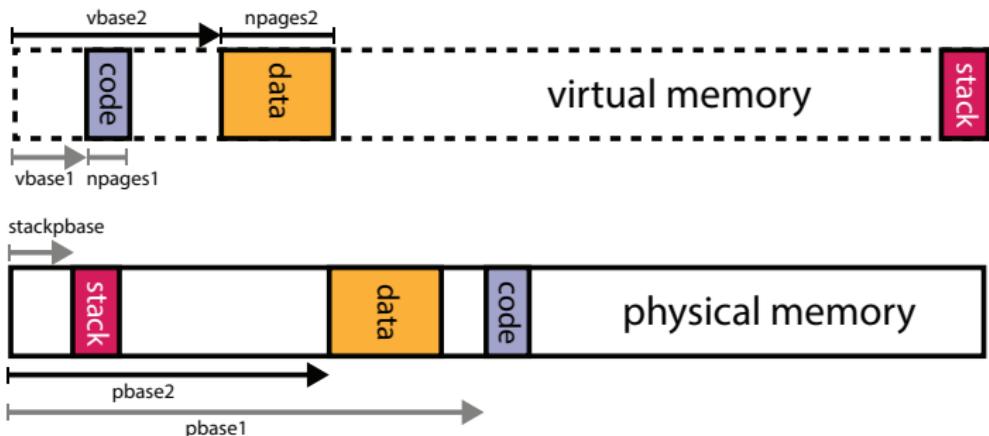
- TLB misses are increasingly expensive with deeper page tables
  - To translate an address causing a TLB miss for a *three-level page table* requires *three* memory accesses, i.e. one for each page table.

- Paging originates in the late 1950s/early 1960s.
- Current Intel CPUs support 4-level paging with 48-bit virtual addresses.
- Support for 5-level paging with 57-bit virtual addresses is coming and Linux already supports it.

- *Recall:* MIPS uses 32-bit paged virtual and physical addresses and MIPS has a software-managed TLB
  - software-managed TLBs raise an exception on every TLB miss
  - kernel is free to record page-to-frame mappings however it wants to
  - TLB exceptions are handled by a kernel function called `vm_fault`
- `vm_fault` *uses information from an addrspace structure to determine a page-to-frame mapping* to load into the TLB
  - there is a separate addrspace structure for each process
  - each addrspace structure describes where its process's pages are stored in physical memory
  - an addrspace structure does the same job as a page table, but the addrspace structure is simpler because *OS/161 places all pages of each segment contiguously* in physical memory

# The addrspace Structure

```
struct addrspace {  
    vaddr_t as_vbase1; /* base virtual address of code segment */  
    paddr_t as_pbase1; /* base physical address of code segment */  
    size_t as_npages1; /* size (in pages) of code segment */  
    vaddr_t as_vbase2; /* base virtual address of data segment */  
    paddr_t as_pbase2; /* base physical address of data segment */  
    size_t as_npages2; /* size (in pages) of data segment */  
    paddr_t as_stackpbase; /* base physical address of stack */  
};
```



# The addrspace Structure

```
vaddr_t as_vbase1; /* base virtual address of code segment */  
paddr_t as_pbase1; /* base physical address of code segment */  
size_t as_npages1; /* size (in pages) of code segment */
```

For the *code and data segments* you keep track of the

- the starting address of each segment in *virtual* memory, `as_vbase1`, (unlike dynamic relocation or segmented addresses).
- the starting address of each segment in *physical* memory, `as_pbase1`, (similar to the `relocation` register as in segmented addresses).
- the size in pages, `as_npages1`, (rather than a `limit` register as in segmented addresses).

For the *stack*

- just keep track location of the base of the stack in physical memory, `stackpbbase`, as the size, top and bottom of the stack in virtual memory *is fixed for all processes*.

## dumbvm Address Translation

First calculate the **top** and **bottom** address for each segment.

```
vbase1 = as->as_vbase1;  
vtop1 = vbase1 + as->as_npages1 * PAGE_SIZE;  
vbase2 = as->as_vbase2;  
vtop2 = vbase2 + as->as_npages2 * PAGE_SIZE;  
stackbase = USERSTACK - DUMBVM_STACKPAGES * PAGE_SIZE;  
stacktop = USERSTACK;
```

Where the following are predefined constants.

USERSTACK = 0x8000 0000

DUMBVM\_STACKPAGES = 0xC // decimal 12

PAGE\_SIZE = 0x1000 // decimal 4096 or 4K

Together they define the top, bottom and size of the stack in virtual memory.

See: kern/arch/mips/include/vm.h

and: kern/include/addrspace.h

and: kern/arch/mips/vm/dumbvm.c

# dumbvm Address Translation

```
if (faultaddress >= vbase1 && faultaddress < vtop1)
    paddr = (faultaddress - vbase1) + as->as_pbase1;

else if (faultaddress >= vbase2 && faultaddress < vtop2)
    paddr = (faultaddress - vbase2) + as->as_pbase2;

else if (faultaddress >= stackbase && faultaddress < stacktop)
    paddr = (faultaddress - stackbase) + as->as_stackpbase;

else
    returnEFAULT;
```

**Key Task:** To perform the address translation of faultaddress

1. *Determine if the address is valid* by checking if it is in the proper range for any of the code, data or stack segments.
2. If so then (a) *calculate its offset* from the virtual base address for that segment. Then (b) *add the physical base address* to that offset.

# Address Translation: OS/161 dumbvm Example

Variable/Field	Process 1	Process 2
as_vbase1	0x0040 0000	0x0040 0000
as_pbase1	0x0020 0000	0x0050 0000
as_npages1	0x0000 0008	0x0000 0002
as_vbase2	0x1000 0000	0x1000 0000
as_pbase2	0x0080 0000	0x00A0 0000
as_npages2	0x0000 0010	0x0000 0008
as_stackpbase	0x0010 0000	0x00B0 0000

	Process 1	Process 2
Virtual addr	0x0040 0004	0x0040 0004
Physical addr =	----- ?	----- ?
Virtual addr	0x1000 91A4	0x1000 91A4
Physical addr =	----- ?	----- ?
Virtual addr	0x7FFF 41A4	0x7FFF 41A4
Physical addr =	----- ?	----- ?
Virtual addr	0x7FFF 32B0	0x2000 41BC
Physical addr =	----- ?	----- ?

## Address Translation: OS/161 dumbvm Example

Variable/Field	Process 1	Process 2
as_vbase1	0x0040 0000	0x0040 0000
as_pbase1	0x0020 0000	0x0050 0000
as_npages1	0x0000 0008	0x0000 0002

Translate Process 1 virtual address 0x40 0004. First check if it belongs to any segments.

```
vbase1 = as->as_vbase1 = 40 0000  
vtop1 = vbase1 + as->as_npages1 * PAGE_SIZE  
= 40 0000 + 8x1000 = 40 8000
```

0x40 0004 is between vbase1 and vtop1 so translate it.

```
paddr = (faultaddress - vbase1) + as->as_pbase1  
= (40 0004 - 40 0000) + 20 0000 = 20 0004
```

# Address Translation: OS/161 dumbvm Example

Variable/Field	Process 1	Process 2
as_vbase2	0x1000 0000	0x1000 0000
as_pbase2	0x0080 0000	0x00A0 0000
as_npages2	0x0000 0010	0x0000 0008

Translate Process 1 virtual address 0x1000 91A4.

It is not in range 40 0000 to 40 8000 (from previous slide) so it is not in segment1. Try segment2.

$$\text{vbase2} = \text{as->as\_vbase1} = 1000\ 0000$$

$$\begin{aligned}\text{vtop2} &= \text{vbase2} + \text{as->as\_npages2} * \text{PAGE\_SIZE} \\ &= 1000\ 0000 + 10 \times 1000 = 1001\ 0000\end{aligned}$$

0x1000 91A4 is between vbase2 and vtop2 so translate it.

$$\begin{aligned}\text{paddr} &= (\text{faultaddress} - \text{vbase2}) + \text{as->as\_pbase2} \\ &= (0x1000\ 91A4 - 1000\ 0000) + 80\ 0000 = 80\ 91A4\end{aligned}$$

## Address Translation: OS/161 dumbvm Example

Variable/Field	Process 1	Process 2
as_stackpbase	0x0010 0000	0x00B0 0000

Translate Process 1 virtual address 0x7FFF 41A4.

It is not in range 40 0000 to 40 8000  $\Rightarrow$  not in segment1.

It is not in range 1000 0000 to 1001 0000  $\Rightarrow$  not in segment2.

Try stack. *Recall:* the base and top of the stack is fixed.

```
stackbase = USERSTACK - DUMBVM_STACKPAGES * PAGE_SIZE;  
          = 8000 0000 - C x 1000 = 7FFF 4000  
stacktop = USERSTACK = 8000 0000
```

0x7FFF 41A4 is between `stackbase` and `stacktop` so translate it.

```
paddr = (faultaddress - stackbase) + as->as_stackpbase  
       = (7FFF 41A4 - 7FFF 4000) + 10 0000 = 10 01A4
```

## Address Translation: OS/161 dumbvm Example

Variable/Field	Process 1	Process 2
as_stackpbase	0x0010 0000	0x00B0 0000

Translate Process 1 virtual address 0x7FFF 31A4.

It is not in range 40 0000 to 40 8000 ⇒ not in segment1.

It is not in range 1000 0000 to 1001 0000 ⇒ not in segment2.

It is not in range 7FFF 4000 to 8000 0000 ⇒ not in the stack.

It is not in the range of any segment so return EFAULT.

*Conclusion:* We now know how to translate a virtual address.

*Next Question:* But where do those virtual addresses come from?

*Answer:* They come from the program's object file...

## Initializing an Address Space

- When the kernel *creates a process* to run a particular program,
  - it must *create an address space* for the process and
  - it *loads the program's code and data* into that address space.
- OS/161 *pre-loads* the address space before the program runs.
- Many other OS load pages *on demand*. (Why?) ⇒
  - the program will load faster and
  - it will not use up physical memory for unused features.
- A program's code and data is described in an **executable file** (i.e. an **object file**).
- OS/161 (and some other operating systems) expect executable files to be in **ELF** (**E**xecutable and **L**inking **F**ormat) format.
- This object format has more features than the MERL object file format you saw in CS241.

# Initializing an Address Space

For example (from Assignment 2b)

- The OS/161 execv system call re-initializes the address space of a process

```
int execv(const char *program, char **args)
```

- The `program` parameter of the `execv` system call should be the name of the ELF file for the program that is to be loaded into the address space.

*Key Questions:* What exactly is ELF? and how is it different from MERL?

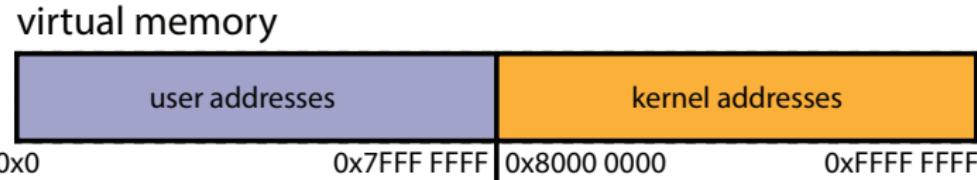
- ELF files contain *address space segment descriptions*.
  - The ELF header describes the segment **images**:
    - the virtual address of the start of the segment
    - the length of the segment in the virtual address space
    - the location of the segment in the ELF file
    - the length of the segment in the ELF file
- Note in CS241, the MERL format only described the text segment as we did not have global variables.
- The ELF file identifies the (virtual) address of the program's first instruction (i.e. the **entry point**).
- The ELF file also contains lots of other information (e.g., section descriptors, symbol tables, relocation tables, external symbol references) that is useful to compilers, linkers, debuggers, loaders and other tools used to build programs.

- OS/161's dumbv<sub>m</sub> implementation assumes that an ELF file contains two segments:
  1. a *text segment*, containing the program code and any read-only data
  2. a *data segment*, containing any other global program data
- The images in the ELF file are an exact copy of the binary data to be stored in the address space.
- But the ELF file does not describe the stack (why not?)
  - Because the initial contents of the stack (command line arguments) are unknown until run time.
- dumbv<sub>m</sub> creates a *stack segment* for each process.
  - It is 12 pages long, running from virtual address 0x7FFF C000 to 0x7FFF FFFF.

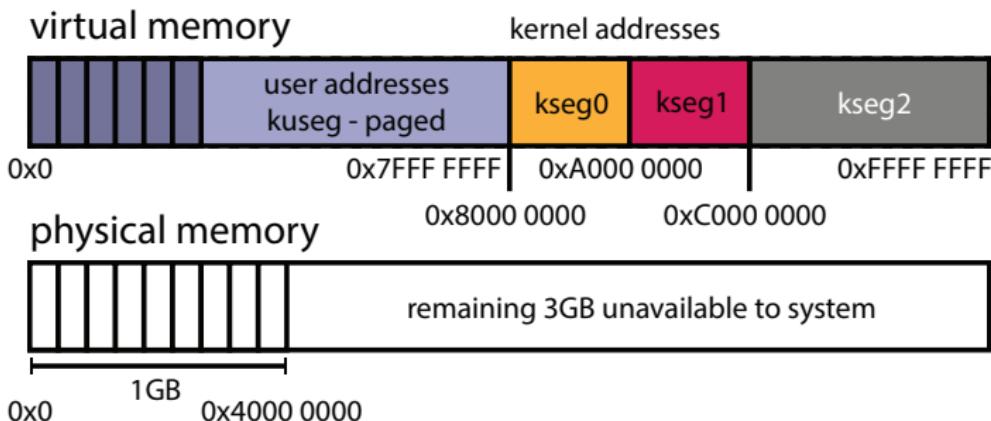
- The image (binary data) in the ELF *may be smaller* than the segment it is loaded into (in the address space) because the segment has an integer number of pages.
- When the image is smaller than the segment the rest of the address space segment is expected to be *zero-filled*.
- Look at `kern/syscall/loadelf.c` to see how OS/161 loads segments from ELF files.

# Virtual Memory for the Kernel

- We would like the kernel to live in virtual memory, but there are some challenges:
  - *Bootstrapping*: Since the kernel helps to implement virtual memory, how can the kernel run in virtual memory when it is just starting?
  - *Sharing*: Sometimes data need to be copied between the kernel and application programs? How can this happen if they are in different virtual address spaces?
- The sharing problem can be addressed by making the kernel's virtual memory *overlap* with process' virtual memories.
- Solutions to the bootstrapping problem are architecture-specific.

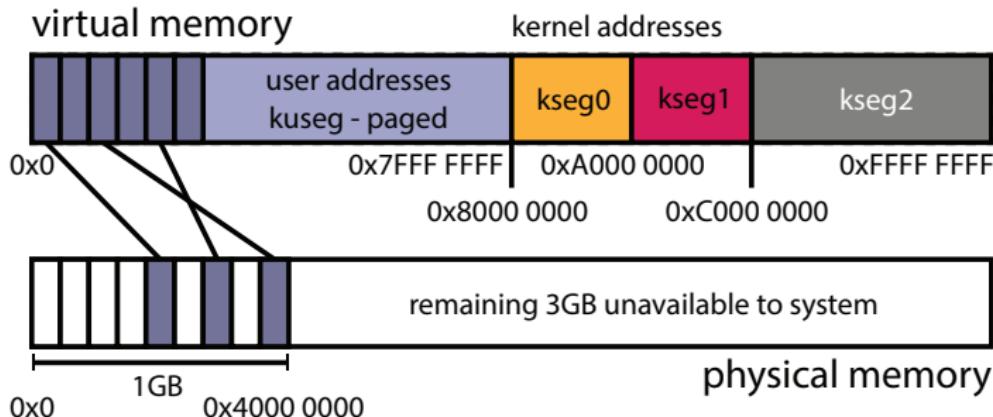


# OS/161 Memory



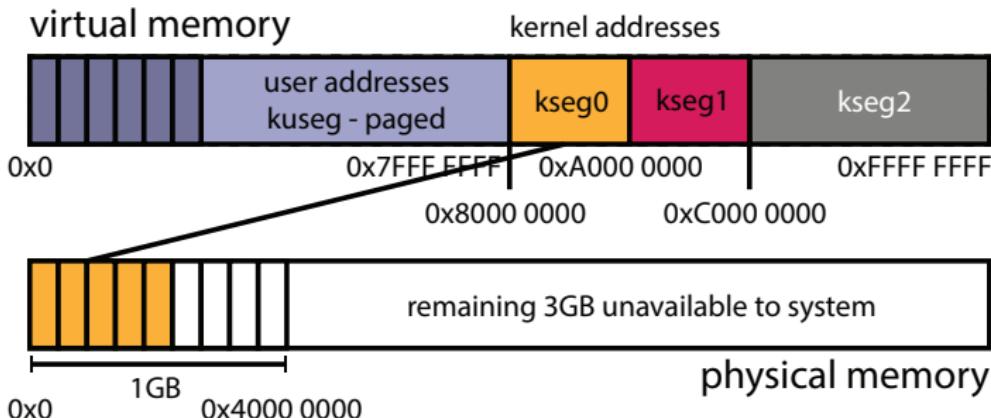
- **Key Point:** Sys/161 only supports 1GB of physical memory. The remaining 3GB are not available (i.e. not usable).
- The kernel's virtual memory is divided into three segments:
  - **kseg0** (512MB): for kernel data structures, stacks, etc.
  - **kseg1** (512MB): for addressing devices
  - **kseg2** (1GB): not used
- Physical memory is divided into frames which are managed by the kernel in the **coremap**.

# OS/161 Memory: kuseg



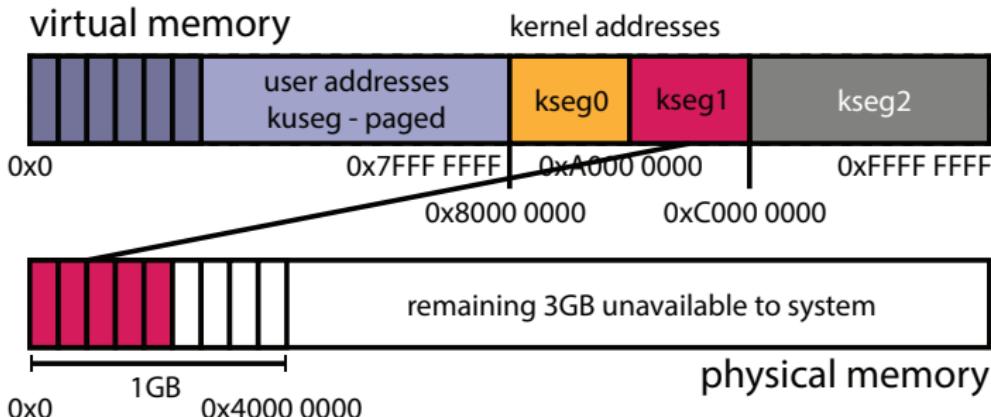
- User virtual memory, **kuseg**, is paged.
- The kernel maintains the page-to-frame mappings for each process.
- The TLB is used to translate **kuseg** virtual addresses to physical ones.

# OS/161 Memory: kseg0



- Addresses within **kseg0** are for the kernel's data structures, stacks, and code.
- *To translate* kseg0 addresses to physical ones *subtract 0x8000 0000* from the virtual address. I.e. the TLB is not used.
- **kseg0** maps to the first 512MB of physical memory, though it may not use all of this space.

# OS/161 Memory: kseg1



- Addresses within **kseg1** are for accessing devices, such as the hard drive or the timer.
- *To translate kseg1 addresses to physical ones subtract 0xA000 0000 from the virtual address.* Again, the TLB is not used.
- **kseg1** also maps to the first 512MB of physical memory, though it does not use all of this space.
- **kseg2** is not used.

# Exploiting Secondary Storage

## Goals:

- Allow virtual address spaces that are *larger than the physical address space* (i.e. installed RAM).
- Allow greater multiprogramming levels by using less of the available primary memory (i.e. installed RAM) for each process.

## Method:

- Allow pages from virtual memories to be *stored in secondary storage*, i.e., on a hard disk drive (HHD) or solid state drive (SSD).
- **Swap** pages (or segments) between secondary storage and primary memory so that they are in primary memory when they are needed.

# Resident Sets and Present Bits

**Key Question:** What additional information needs to be tracked to allow us to exploit secondary storage?

- When swapping is used, some pages of virtual memory will be in primary memory and others will not be in primary memory.
  - The set of virtual pages present in physical memory is called the **resident set** of a process.
  - A process's resident set will change over time as pages are swapped in and out of physical memory
- To track which pages are in physical memory, each PTE needs to contain an extra bit, called the **present bit**:
  - `valid=1, present=1` ⇒ page is valid and in memory
  - `valid=1, present=0` ⇒ page is valid, but not in memory
  - `valid=0` ⇒ invalid page

# Page Faults

- When a process *tries to access a page that is not in memory*, the situation is detected because the page's **present** bit is zero.
  - On a machine with a *hardware-managed TLB*
    - the MMU detects this situation when it checks the page's PTE,
    - it generates an exception, which the kernel must handle.
  - On a machine with a *software-managed TLB*
    - the kernel detects this situation when it checks the page's PTE after a TLB miss
    - i.e., the TLB should not contain any entries that are not present in RAM.
- This event (attempting to access a non-resident page) is called a **page fault**.

# Page Faults

When a page fault happens, it is the kernel's job to:

1. Swap the page into memory from secondary storage, possibly evicting another page (move it from RAM to secondary storage) if necessary.
2. Update the PTE for that page (i.e. set the **present** bit)
3. Return from the exception so that the application can retry the virtual memory access that caused the page fault.

Recall the memory hierarchy from CS251: Access time (latency) for

- primary memory is approximately 1 (L1 cache) to 100 *nanoseconds* for RAM
- secondary storage is approximately 15,000 ns (15 *microseconds*) for an SSD
- secondary storage is approximately 15,000,000 ns (15 *milleseconds*) for a HDD

## Page Faults are Slow

- The impact of swapping on the average memory access time depends on the fault frequency.
- If secondary storage access is 1000 times slower than ...

Fault Frequency	Average Memory Access Time
1 in 10 memory accesses	100 times slower
1 in 100	10 times slower
1 in 1000	2 times slower

*Key Idea:* to improve the performance of virtual memory with on-demand paging, *reduce the occurrence of page faults*.

- Limit the number of processes, so that there is enough physical memory per process.
- Try to be smart about which pages are kept in physical memory and which are evicted, i.e. have a **replacement policy**.
- Hide latencies, e.g., by **prefetching** pages before a process needs them.

# A Simple Replacement Policy: FIFO

- A **replacement policy**: specifies how to determine which page to evict from physical memory.
- The **FIFO policy**: replace the page that has been in memory the longest
- For simplicity, below there are only three frames in physical memory.

	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 1	a	a	a	d	d	d	e	e	e	e	e	e
Frame 2		b	b	b	a	a	a	a	a	c	c	c
Frame 3			c	c	c	b	b	b	b	b	d	d
Fault ?	x	x	x	x	x	x	x			x	x	

## A Simple Replacement Policy: FIFO

- **Refs** means which page has been referenced / swapped in from secondary storage.
- Fault? has an 'x' when the request resulted in a page fault.
- **1–3:** initially physical memory is empty so each page request results in a page fault.
- **4:** with physical memory full, in order to swap in page **d** the kernel must evict a page. By the FIFO policy, since **a** has been there the longest, evict it.
- **5:** since **b** has been there the longest (since 2), evict it and add **a**.
- **6:** since **c** has been there the longest (since 3), evict it and add **b**.
- **7:** since **d** has been there the longest (since 4), evict it and add **e**.
- **8–9** no need to evict any pages since **a** and **b** are already in physical memory.
- and so on ...

# Optimal Page Replacement

There is an *optimal page replacement policy for demand paging*, called MIN: replace the page that will not be referenced for the longest time.

	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 1	a	a	a	a	a	a	a	a	a	c	c	c
Frame 2		b	b	b	b	b	b	b	b	b	d	d
Frame 3			c	d	d	d	e	e	e	e	e	e
Fault ?	x	x	x	x			x			x	x	

- MIN requires knowledge of the future so it is only a theoretical policy.
- Other practical policies can be compared to MIN's performance.
- MIN differs from FIFO at 3: evict **c** because **a** will be referenced at 5 and **b** will be referenced at 6 but **c** will not be referenced again until 10.

# Locality

- Real programs do not access their virtual memories randomly. Instead, they exhibit **locality**.
- There are two types of locality.
  1. **temporal locality:** programs are more likely to access pages that they have accessed recently than pages that they have not accessed recently.
  2. **spatial locality:** programs are likely to access parts of memory that are close to parts of memory they have accessed recently.
- Locality helps the kernel keep the page fault rates low.

# Least Recently Used (LRU) Page Replacement

Using the same three-frame example as before, **Least Recently Used (LRU)** Page Replacement would evict the page that has not been used in the longest period of time.

	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 1	a	a	a	d	d	d	e	e	e	c	c	c
Frame 2		b	b	b	a	a	a	a	a	a	d	d
Frame 3			c	c	c	b	b	b	b	b	b	e
Fault ?	x	x	x	x	x	x	x			x	x	x

LRU differs from FIFO starting at column 10.

- FIFO would evict **a** since it was in memory before **e** or **b**.
- LRU would evict **e** since it was last referenced at 7, before **a** or **b**.

# Measuring Memory Accesses

## *Challenges with using LRU*

- It must track usage and find a maximum value which is expensive.
- The kernel is not aware which pages a program is using unless there is an exception.
- This makes it difficult for the kernel to exploit locality by implementing a replacement policy like LRU.

*Solution:* Have the MMU track page accesses in hardware.

- Simple scheme: add a **use** bit (or **reference** bit) to each PTE.  
This bit:
  - is set by the MMU each time the page is used, i.e., each time the MMU translates a virtual address on that page
  - can be read and cleared by the kernel.
- The **use** bit provides a small amount of memory usage information that can be exploited by the kernel.

# The Clock Replacement Algorithm

- The **Clock Algorithm** (also known as **Second Chance**) is one of the simplest algorithms that exploits the use bit.
- The clock algorithm can be visualized as a victim pointer that cycles through the page frames. The pointer moves whenever a replacement is necessary:

```
1. while (use bit of victim is set) {  
2.     clear use bit of victim           // get a 2nd chance  
3.     victim = (victim + 1) % num_frames  
4. }  
5. evict victim                      // its use bit is 0  
6. victim = (victim + 1) % num_frames
```

- Lines 1–4: skip over any pages whose **use** bit is set.
- Line 5: To get to this line, you've found a page whose **use** bit is not set, so evict this page.
- Line 6: Go to next frame.

# The Clock Replacement Algorithm

	1	2	3	4	5	6	7	8	9	10	11	12
Refs	a	b	c	d	a	b	e	a	b	c	d	e
Frame 1	a	a	a	d	d	d	e	e	e	e	e	e
Frame 2		b	b	b	a	a	a	a	a	c	c	c
Frame 3			c	c	c	b	b	b	b	b	d	d
Fault ?	x	x	x	x	x	x	x			x	x	
victim				1	2	3	1			2	3	

- 4: Cycle through and clear each **use** bit and then select Frame 1.
- 5–7: Evict next victim.
- 8–9: Set **use** bits for Frame 2 and 3.
- 10: Cycle through and clear each **use** bit and then select Frame 2.
- 11: Evict next victim.
- ...

## Summary: Physical vs. Virtual Memory

- **Physical addresses** are provided directly by the hardware. [1.3]
- **Virtual addresses** are addresses provided by the OS. [1.3]
- There is one virtual address space per process. [1.3]
- **Address translation** is the conversion of a virtual address to a physical one. [1.3]
- If there are  $s$  addresses  $\Rightarrow$  it takes  $\log_2 s$  bits to specify them. [2]
- If it takes  $b$  bits to specify them  $\Rightarrow$  there are up to  $2^b$  addresses. [2]
- Virtual memory is used to
  - *isolate* processes from each other and from the kernel,
  - potentially support virtual memory that *is larger* than physical memory. [5]

## Summary: Address Translation

- **Dynamic Relocation** uses a **limit** register to check the size of each address and an **offset** register to translate the virtual address to a physical one. [7–8]
  - Since physical addresses are contiguous, dynamic relocation causes fragmentation of physical memory. [9]
- **Segmented Addresses** attempts to reduce fragmentation by mapping each segment of a process using its own **limit** and **offset** values. [11–16.3]
  - Each virtual address is broken up into a **segment number** and an **offset**. [14–16.3]
- **Paging** divides up physical memory into **frames** and virtual memory into **pages** and maps each page to a frame using a **page table**. [17–24]
- Each row of a page table is called a **page table entry (PTE)**. [20]

## Summary: Address Translation and TLBs

- A PTE has a **valid** bit to track if the mapping is used or not. [20.1]
- A virtual address is broken up into a **page number** and an **offset**. [21–24]
- Page tables can get quite large. [25–26]
- Multi-level paging reduces the amount of space used for a page table by not creating tables if they do not contain valid PTEs. [27–28, 30–30.1]
- $n$ -level paging will break a virtual address into  $n$  page numbers,  $p_1 \dots p_n$  and one offset. Each  $p_i$  is an index into the  $i$  level page table. [29–30]
- A **TLB** is a cache that stores recent address translations in the MMU. [33]
- A TLB can be **Hardware-Managed** or **Software-Managed**. [34–35]
- OS/161 uses segmented addresses but the size of each segment is specified in 4KB pages. [38–41.4]

# Summary: Virtual Memory and Swapping

- In OS/161, the text and data segments of virtual memory are specified in ELF (object) files. [42–44.1]
- In OS/161 kernel space is divided into three segments kseg0, kseg1 and kseg2 (unused). [45–49]
- Pages can also be stored in secondary storage. [50]
- **Primary storage** is memory that the processor can directly access, i.e. RAM. [50]
- **Secondary storage** is memory that can store a large amount of data permanently, i.e. a HHD or a SSD. [50]
- The **resident set** is the set of all virtual pages that are present in physical memory. [51]
- A **page fault** is an attempt to access a location on a non-resident page. [52–52.1]

## Summary: Page Replacement Policies

- Page faults slow down computation and there are a number of methods, called **replacement policies** for determine which pages to keep in primary memory. [53]
  - **FIFO** evicts the oldest page. [54–54.1]
  - The optimal policy evicts the page that will not be references for the longest time. [55]
  - **LRU** evicts the page that has not been referenced for the longest time. [57–58]
  - The **clock algorithm** uses a **use** bit to give each page a second chance. [59–59.1]
- Page replacement policies try to take advantage of
  - **temporal locality**: programs are more likely to access pages that they have accessed recently. [56]
  - **spatial locality**: programs are likely to access parts of memory that are close to parts of memory they have accessed recently. [56]

# Scheduling

**key concepts:** round robin, shortest job first, MLFQ, multi-core scheduling, cache affinity, load balancing

Lesley Istead, Kevin Lanctot

David R. Cheriton School of Computer Science  
University of Waterloo

Winter 2019

# Simple Scheduling Model

First consider a simplified version of the **Job Scheduling Problem**.

- A set of **jobs**  $\{J_1, J_2, J_3, \dots\}$  needs to be executed using a single server.
  - Only one job can run at a time.
  - The server can switch from executing one job to another instantly.
  - The server can switch from executing one job to another at any point in time.

## 1. Inputs to the Scheduling Algorithm

- For the  $i^{\text{th}}$  job, there are two parameters that characterize it,
  - an **arrival time**  $a_i$ , when the  $i^{\text{th}}$  job becomes available to run,
  - a **run time**  $r_i$ , the total length of time (i.e. total amount of processing time) required to complete the  $i^{\text{th}}$  job.

## 2. Outputs of the Scheduling Algorithm

- A job **scheduler** decides which job should be running on the server at each point in time.
- These decisions determine two times for each job,
  - a **start time**  $s_i$ , when the  $i^{\text{th}}$  job starts running,
  - a **finish time**  $f_i$ , when the  $i^{\text{th}}$  job finishes running.

## 3. Performance Metrics

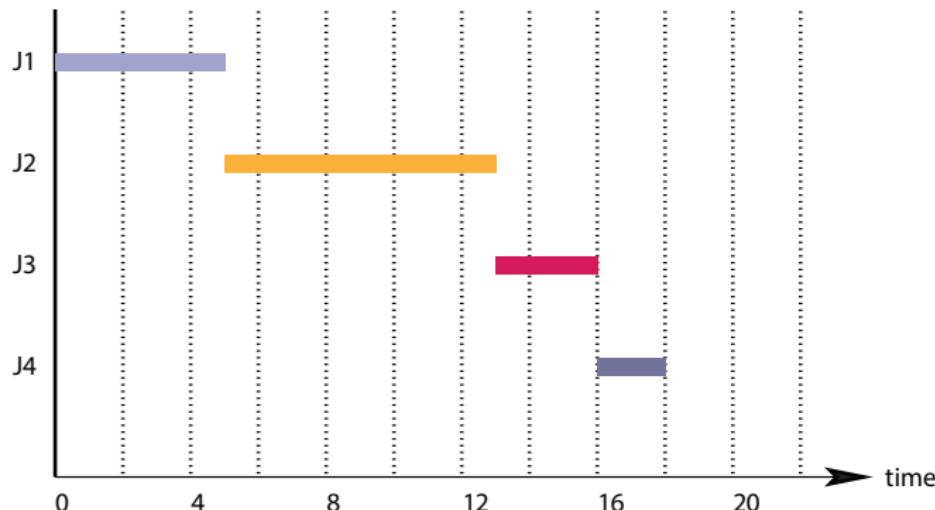
- The *performance of the scheduler* is characterized using two times,
  - the **response time**:  $s_i - a_i$ , i.e. how long it takes from the arrival of the job until it *starts running*,
  - the **turnaround time**:  $f_i - a_i$ , i.e. how long it takes from the arrival of the job until it *finishes running*.
- For example, think of lining up at a grocery store check out.

# Four Basic Schedulers

- **First Come First Serve (FCFS):** runs jobs in arrival time order.
  - simple, avoids starvation
  - pre-emptive variant: **Round-Robin (RR)**
- **Shortest Job First (SJF):** run jobs in increasing order of  $r_i$ 
  - minimizes average *turnaround* time
  - long jobs may starve
  - pre-emptive variant: **Shortest Remaining Time First (SRTF)**
- A **Gantt Chart** will be used to illustrate how four scheduling disciplines differ.

# First Come, First Served

- *Strategy:* jobs run in order of arrival
- *Attributes:* simple, avoids starvation



Job	J1	J2	J3	J4
arrival ( $a_i$ )	0	0	0	5
run time ( $r_i$ )	5	8	3	2

## Explaining the Gantt Chart

- The table below is the *input* to the scheduling algorithm.
- It consists of the arrival time,  $s_i$ , and the run time,  $r_i$ , for each job.

Job	J1	J2	J3	J4
arrival ( $a_i$ )	0	0	0	5
run time ( $r_i$ )	5	8	3	2

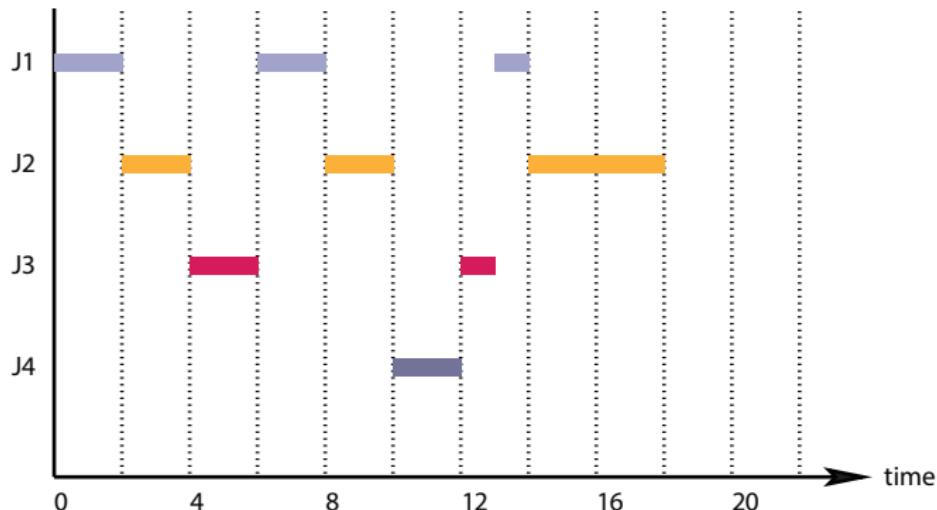
- Three jobs are added to the *Ready* queue at time 0, i.e. their arrival time  $a_i = 0$ , say due to a `cv.broadcast()`.
- They have been added in the order J1, J2, J3.
- Job J4 was added to the *Ready* queue at time 5.
- Each job has a run time of 5, 8, 3 and 2 respectively.
- Typically the run time for a job is not known ahead of time (this is a simplification for now).

# Explaining the Gantt Chart

- The **Gantt Chart** shows the *output* of the scheduling algorithm, i.e. the schedule it created.
- In a Gantt Chart
  - the  $x$ -axis represents time,
  - the  $y$ -axis represents the various jobs,
  - the bars represent when specific jobs were run.
- The beginning of the bar for job  $J_i$ , marks the start time,  $s_i$ .
- The end of the bar marks the finish time,  $f_i$ .
- The total length of the bar (not including blank spaces) is the run time,  $r_i$ .
- The arrival time is not shown on the Gantt chart.

# Round Robin

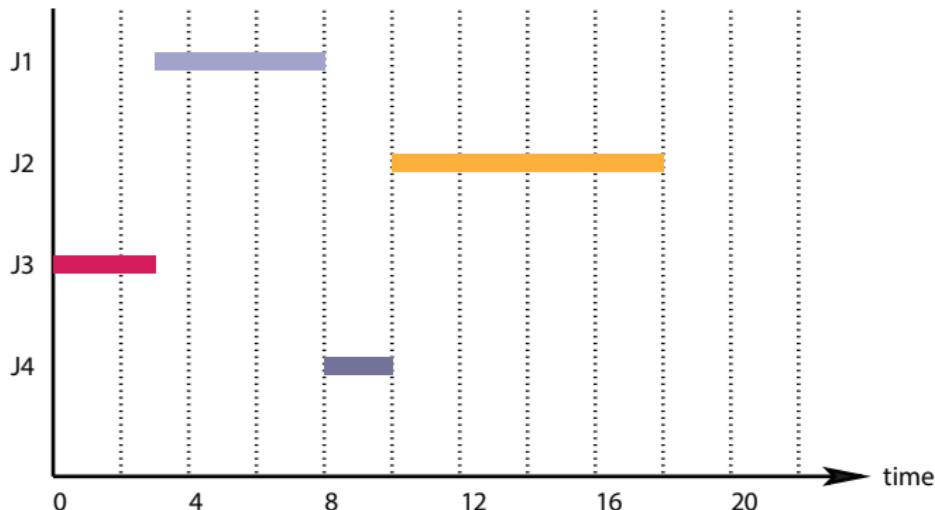
- *Strategy:* preemptive FCFC
- OS/161's scheduler



Job	J1	J2	J3	J4
arrival ( $a_i$ )	0	0	0	5
run time ( $r_i$ )	5	8	3	2

# Shortest Job First

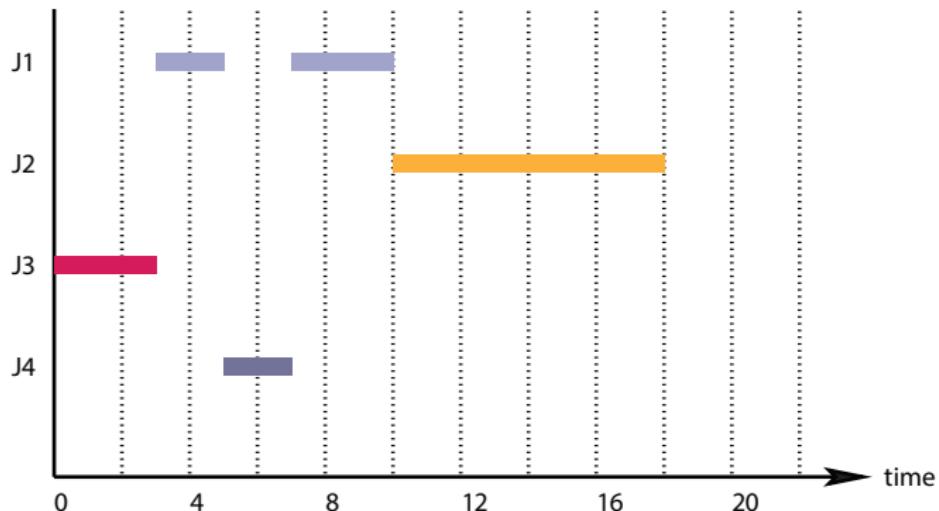
- *Strategy:* run jobs in increasing order of runtime
- *Attributes:* minimizes average *turnaround* time but *starvation is possible*



Job	J1	J2	J3	J4
arrival ( $a_i$ )	0	0	0	5
run time ( $r_i$ )	5	8	3	2

# Shortest Remaining Time First

- *Strategy*: preemptive variant of SJF: select one with shortest remaining time but arriving jobs preempt a running job
- *Attribute*: starvation still possible



Job	J1	J2	J3	J4
arrival ( $a_i$ )	0	0	0	5
run time ( $r_i$ )	5	8	3	2

# Comparing Performance Metrics

*Turnaround Time* (i.e. finishing time - arrival time) for jobs J1–J4

- *FCFS*  $(5-0) + (13-0) + (16-0) + (18-5) = 47$ . Average  $= 47/4 = 11.75$
- *SJF*  $(3-0) + (8-0) + (18-0) + (10-5) = 34$ . Average  $= 34/4 = 8.5$
- *RR*  $(14-0) + (18-0) + (13-0) + (12-5) = 52$ . Average  $= 52/4 = 13.0$
- *SRTF*  $(10-0) + (18-0) + (3-0) + (7-5) = 33$ . Average  $= 33/4 = 8.25$

*Response Time* (i.e. start time - arrival time) for jobs J1–J4

- *FCFS*  $(0-0) + (5-0) + (13-0) + (16-5) = 29$ . Average  $= 29/4 = 7.25$
- *SJF*  $(3-0) + (10-0) + (0-0) + (8-5) = 16$ . Average  $= 16/4 = 4.0$
- *RR*  $(0-0) + (2-0) + (4-0) + (10-5) = 11$ . Average  $= 11/4 = 2.75$
- *SRTF*  $(3-0) + (10-0) + (0-0) + (5-5) = 13$ . Average  $= 13/4 = 3.25$

# CPU Scheduling

- In CPU scheduling, *the jobs to be scheduled are the threads*.
- CPU scheduling typically differs from the simple scheduling model:
  - the run times of threads are normally not known
  - threads are sometimes not runnable: when they are blocked
  - threads may have different priorities
- The objective of the scheduler is normally to achieve a balance between
  - *responsiveness*: ensure that threads get to run regularly
  - *fairness*: sharing of the CPU
  - *efficiency*: account for the fact that there is a cost to switching

How would FCFC, Round Robin, SJF, and SRTF handle blocked threads? Priorities?

- CPU schedulers are often *expected to consider process and thread priorities.*
- Priorities may be
  - specified by the application or user,
  - chosen by the scheduler,
  - some combination of these two.
- There are two approaches to scheduling with priorities
  1. schedule the highest priority thread
  2. weighted fair sharing
    - let  $p_i$  be the priority of the  $i^{\text{th}}$  thread
    - try to give each thread a share of the CPU in proportion to its priority:

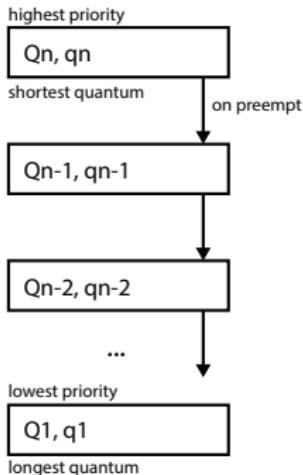
$$\frac{p_i}{\sum_j p_j}$$

# Multi-level Feedback Queues

- **Multi-level Feedback Queues (MLFQ)** is currently the most commonly used scheduling algorithm.
- *Objective:* good responsiveness for interactive threads (e.g. threads that interact with people via the keyboard, mouse and display).
- Non-interactive threads make as much progress as possible.
- *Key Challenge:* how to determine which threads are interactive and which are not?
- *Key Observation:* interactive threads are frequently blocked, waiting for user input, packets, etc.
- *Approach:* give higher priority to threads that block frequently (typically interactive ones), so they run whenever they are ready.

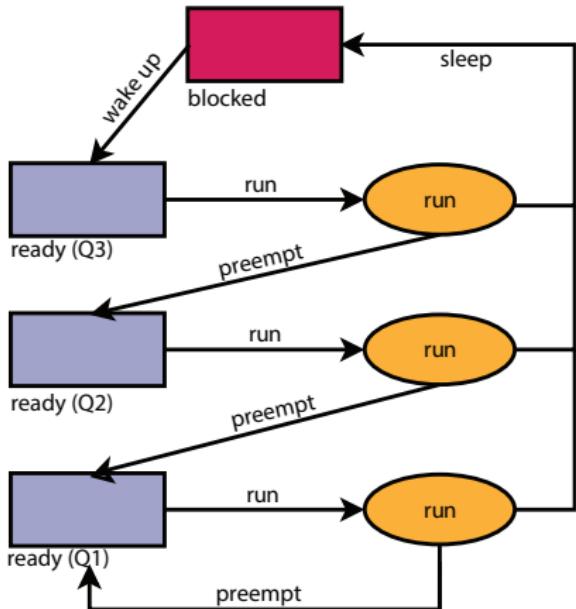
MLFQ is used in Microsoft Windows, Apple macOS, Sun Solaris, and many more. It was used in Linux, but no longer is.

# MLFQ Algorithm



- $n$  round-robin ready queues where the priority of  $Q_i > Q_j$  if  $i > j$  (i.e. *higher level  $\Rightarrow$  higher priority*)
- Threads in  $Q_i$  use quantum  $q_i$  and  $q_i \leq q_j$  if  $i > j$  (i.e. *higher level  $\Rightarrow$  smaller quantum*).
- The scheduler selects threads from the highest priority queue to run (i.e. threads in  $Q_{n-1}$  are only selected if  $Q_n$  is empty).
- *Preempted threads* are put at the back of the next lower-priority queue (i.e. a thread from  $Q_n$  is preempted, it is pushed onto  $Q_{n-1}$ )
- When a *thread wakes after blocking*, put it into the highest-priority queue.
  - Since interactive threads tend to block frequently, they will tend to stay in the higher-priority queues.
  - Non-interactive threads will tend to sift down towards the bottom.

# 3-Queue MLFQ Example

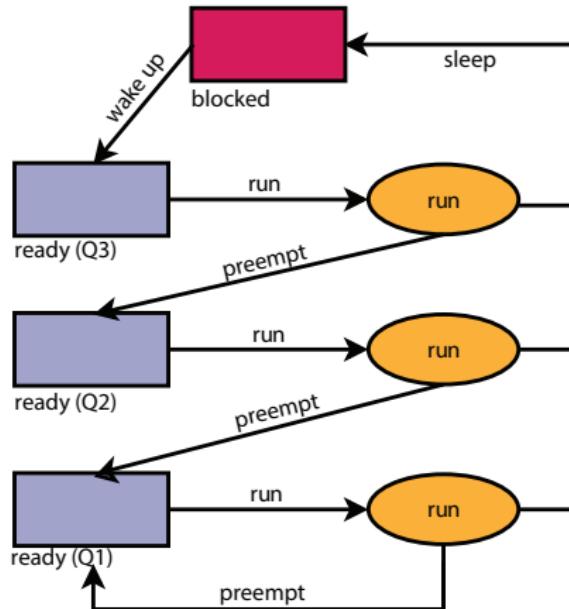


## A 3 Queue Example

- Blocked threads start at the highest level.
- Q3: Top level  
*highest priority  
smallest quantum.*
- When preempted, threads drop down to the next level.
- Q1: Bottom level  
*lowest priority  
largest quantum.*

Note: each level has it's own **Run** queue and **Ready** queue, but they all share the **Blocked** queue.

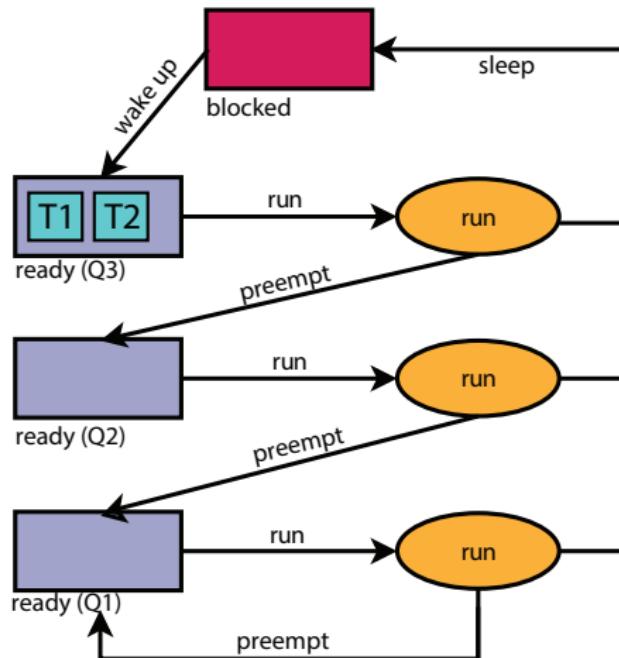
# 3-Queue MLFQ Example



**Question:** When do threads in Q1 run if Q3 is never empty?

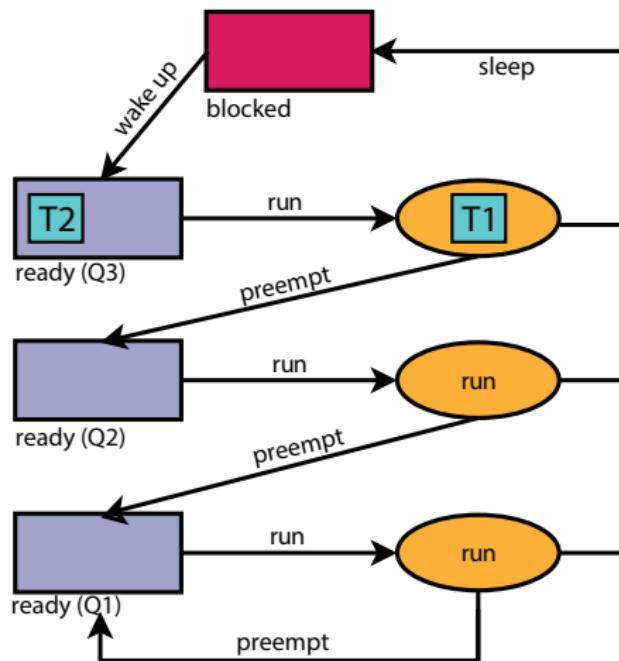
**Answer:** To prevent starvation, all threads are periodically placed in the highest-priority queue.

# 3-Queue MLFQ Example



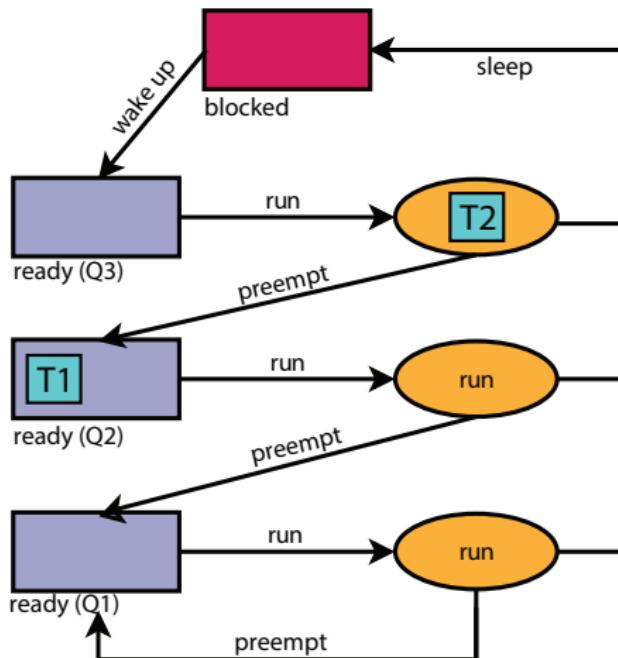
Two threads, T1 and T2, start in Q3.

# 3-Queue MLFQ Example



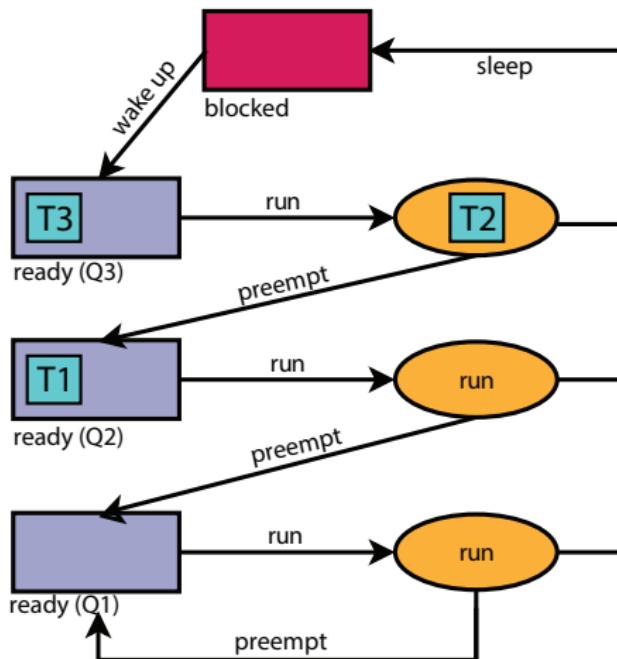
T1 is dispatched (i.e. selected to run).

# 3-Queue MLFQ Example



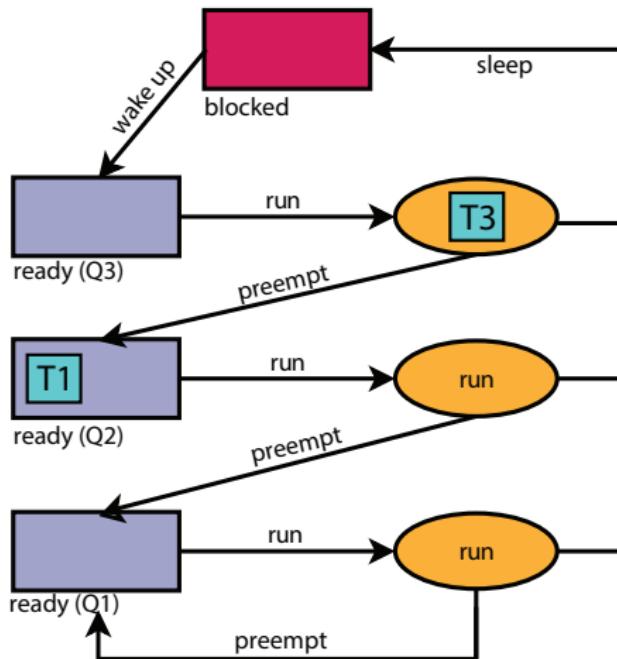
T1 is preempted and pushed to the back of Q2.  
T2 is dispatched (i.e. selected to run).

# 3-Queue MLFQ Example



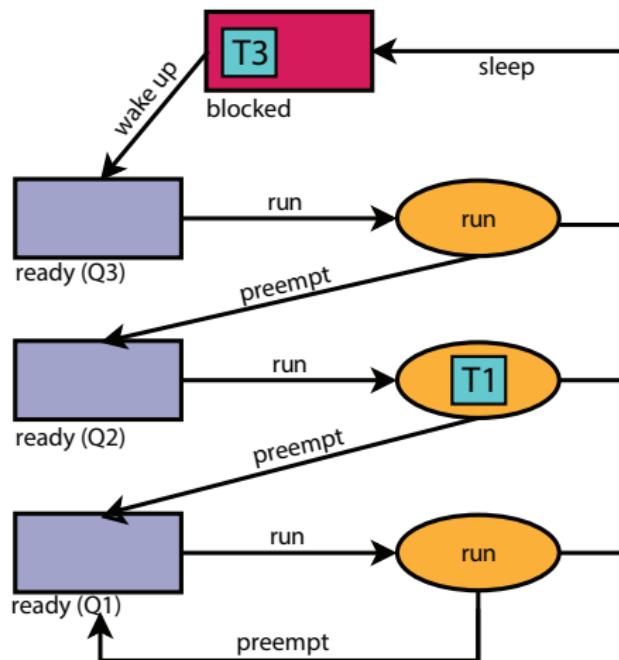
While T2 is running a new thread, T3, arrives.

# 3-Queue MLFQ Example



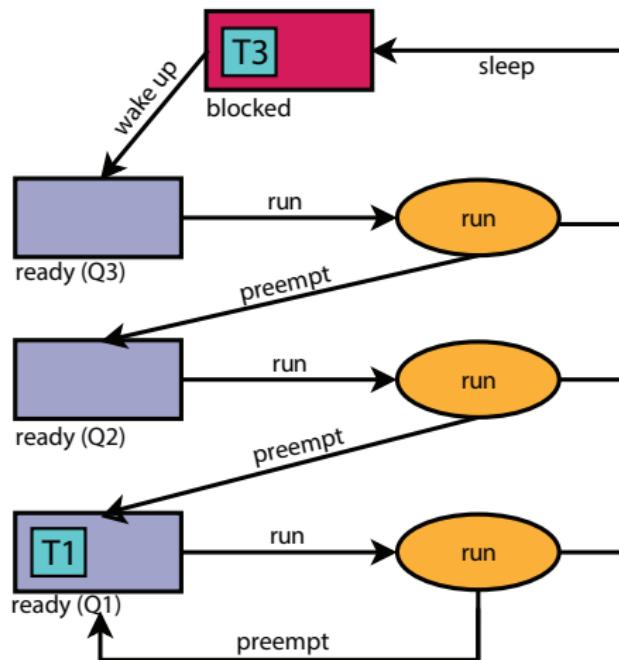
T2 terminates. T3 is dispatched.

# 3-Queue MLFQ Example



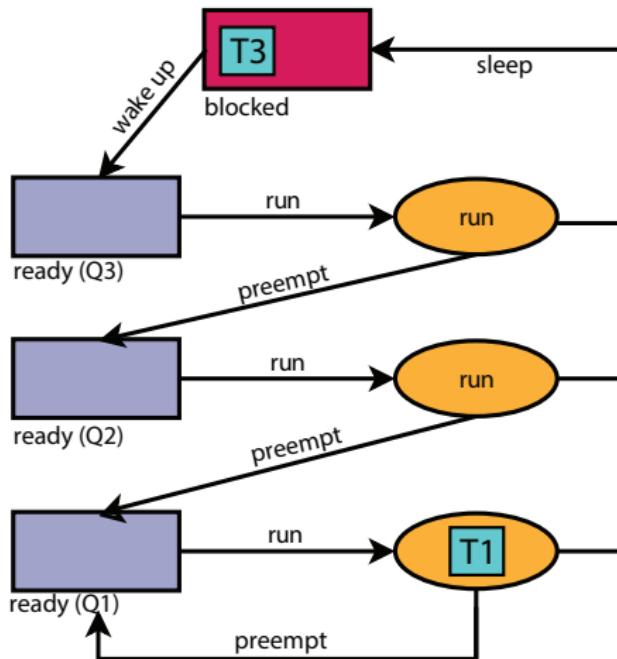
T3 blocks. T1 (on level 2) is dispatched.

# 3-Queue MLFQ Example



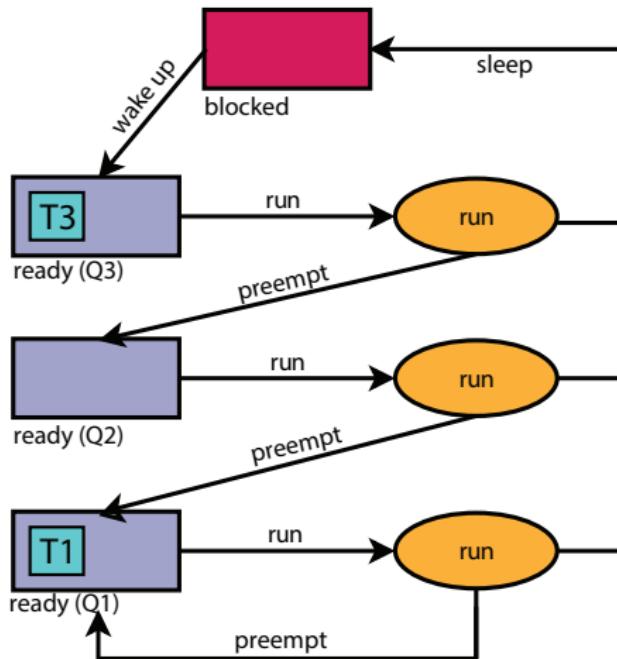
T1 is preempted and pushed to the back of Q1.

# 3-Queue MLFQ Example



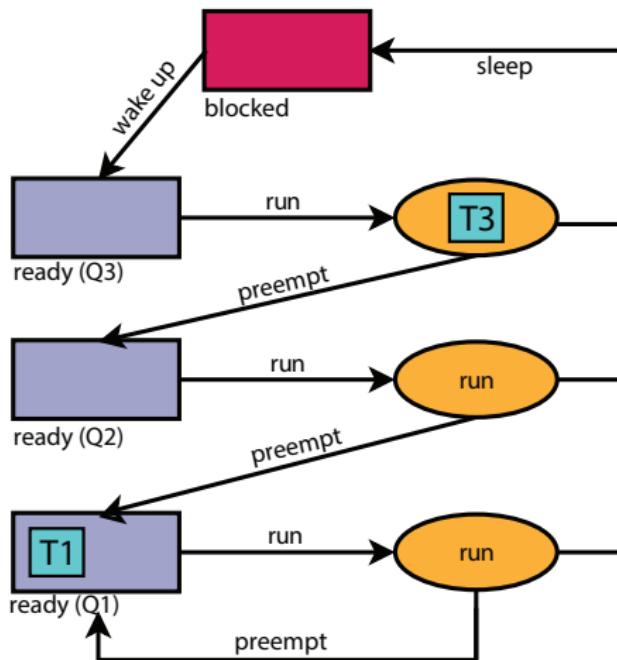
T3 is still blocked so T1 is dispatched.

# 3-Queue MLFQ Example



T3 is woken by T1 (which causes T1 to be preempted).

# 3-Queue MLFQ Example



T3 is dispatched.

## 3-Queue MLFQ Variants

- There are many variants of MLFQ. Often these variants will preempt low-priority threads when a thread wakes to ensure a fast response to an event (as was done in the previous example).
- Some realistic values
  - Six levels
  - Quanta ranging from 40ms for  $Q_6$  to 200ms for  $Q_1$ .
  - Threads are raised up to  $Q_6$  about once per second to avoid starvation.

While many OS (e.g. Windows and macOS) use MLFQs for thread scheduling, Linux uses a different approach. It uses the **Completely Fair Scheduler (CFS)** which is a weighted fair sharing approach...

# Linux Completely Fair Scheduler (CFS) - Main Ideas

- *Key Idea:* each thread is assigned a **weight**.
- The goal of the scheduler is to ensure that each thread gets *a share of the processor proportional to its weight*.
- For example, suppose there are two threads,
  - $T_1$  with weight  $w_1=1$ ,
  - $T_2$  with weight  $w_2=4$ .
  - Then after 5 seconds we would expect  $T_1$  to have run for 1 second and  $T_2$  to have run for 4 seconds,
  - i.e. run time is proportional to their weights, namely  $\frac{1}{1+4}$  and  $\frac{4}{1+4}$ .
- Suppose that  $a_i$  is the actual amount of time that the scheduler has allowed thread  $T_i$  to run.
  - Ideally we would expect  $a_1 \frac{\sum_j w_j}{w_1} = a_2 \frac{\sum_j w_j}{w_2}$ .
  - For simplicity, factor out the  $\sum_j w_j$  term yielding  $\frac{a_1}{w_1} = \frac{a_2}{w_2}$ .

## Linux Completely Fair Scheduler (CFS) -Example

For example, if out of 10 seconds

- $T_1$  ran for 2 seconds and ( $a_1 = 2$ ) and  $T_2$  ran for 8 seconds ( $a_2 = 8$ )  
then  $\frac{a_1}{w_1} = \frac{2}{1} = 2$  and  $\frac{a_2}{w_2} = \frac{8}{4} = 2$  which would be fair.
- $T_1$  ran for 1 second and  $T_2$  ran for 9 seconds  
then  $\frac{a_1}{w_1} = \frac{1}{1} = 1$  and  $\frac{a_2}{w_2} = \frac{9}{4} = 2.25$  so run  $T_1$  to increase its share.
- $T_1$  ran for 3 seconds and  $T_2$  ran for 7 seconds  
then  $\frac{a_1}{w_1} = \frac{3}{1} = 3$  and  $\frac{a_2}{w_2} = \frac{7}{4} = 1.75$  so run  $T_2$  to increase its share.
- *Our goal* is to have  $\frac{a_i}{w_i} = \frac{a_j}{w_j}$  for each pair of threads
- So the thread with the **smallest  $\frac{a_i}{w_i}$  ratio** should run next to increase its share of the processor time.

# Linux Completely Fair Scheduler (CFS) - Scheduling

To schedule threads

- Track the **virtual run time** of each runnable thread.
  - The virtual run time of  $T_i$  is  $a_i \frac{\sum_j w_j}{w_i}$ .
  - The virtual run time is the actual run time ( $a_i$ ) adjusted by the thread weights.
- Always *run the thread with the lowest virtual run time*.
  - The virtual runtime advances slowly for threads with high weights. It advances quickly for threads with low weights.
- When a *thread becomes runnable*, its virtual run time is initialized to some value between the min and max virtual run times of the threads that are already runnable.

MLFQ vs. CFS

- In MLFQ the quantum depended on the thread priority.
- In CFS, *the quantum is the same for all threads and priorities*.

## CFS Example

Suppose the total weight of all threads in the system is 50 and the quantum is 5.

Time	Thread	Weight	Actual	Virtual
			Run time	Run time
$t$	1	25	5	
	2	20	5	
	3	5	5	

$t + 5$	1	25		
	2	20		
	3	5		

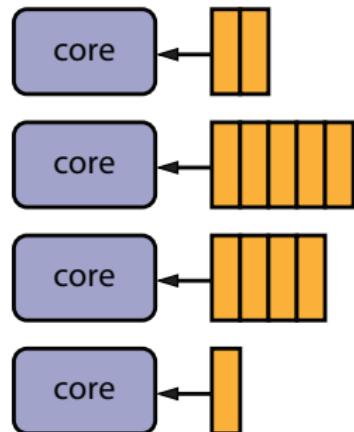
Which thread is selected at time  $t$ ? Which thread at  $t + 5$ ?

## CFS Example

Suppose the total weight of all threads in the system is 50 and the quantum is 5.

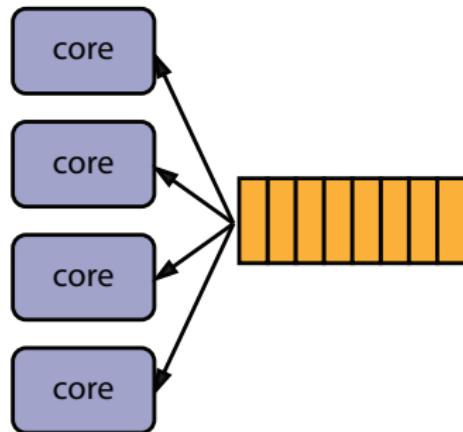
Time	Thread	Weight	Actual Run time		Virtual Run time
$t$	1	25	5	5	$5 * 50/25 = 10$
	2	20	5	5	$5 * 50/20 = 12.5$
	3	5	5	5	$5 * 50/5 = 50$
			T1 is selected		
$t + 5$	1	25	10	10	$10 * 50/25 = 20$
	2	20	5	5	12.5
	3	5	5	5	50
			T2 is selected		

# Scheduling on Multi-Core Processors



Per Core Ready Queue

vs.



Shared Ready Queue

Questions:

- Which offers better performance?
- Which one scales better?

# Scalability and Cache Affinity

## Contention and Scalability

- Access to shared ready queue is a *critical section* ⇒ mutual exclusion is required.
- As the number of cores grows, contention for the ready queue becomes more of a problem.
- *Conclusion: per core design scales* to a larger number of cores.

## CPU Cache Affinity

- As thread runs, data is loaded into CPU cache(s).
- Typically each core will have some memory cache of its own (e.g. L1 and L2) and a cache it shares with the other cores (e.g. L3 cache).
- Moving the thread to another core means data must be reloaded into that core's caches.
- As thread runs, it acquires an **affinity** for one core because of the cached data.

# Cache Affinity and Load Balancing

## CPU Cache Affinity (continued)

- *Conclusion: per core design benefits from affinity* by keeping threads on the same core.

## Load Balancing

- In per-core design, queues often have different lengths.
- This results in **load imbalance** across the cores
  - Some cores may be idle while others are busy.
  - Threads on lightly loaded cores get more CPU time than threads on heavily loaded cores.
- This difference is not an issue in shared queue design.
- Per-core designs typically need some mechanism for **thread migration** to address load imbalances.
  - Thread migration means moving threads from heavily loaded cores to lightly loaded cores.

## Summary: Job Scheduling

For a job scheduling algorithm, for the  $i$ th job

- The *input* is the arrival time ( $a_i$ ) and the run time ( $r_i$ ). [2]
- The *output* is the start time ( $s_i$ ) and the finish time ( $f_i$ ). [2.1]
- The *performance* is measured by the **response time** ( $s_i - a_i$ ) and the **turnaround time** ( $f_i - a_i$ ). [2.1]

The basic scheduling algorithms are

- **First Come First Serve (FCFS)**: runs jobs in arrival time order. [3]
  - simple, avoids starvation
  - pre-emptive variant: **Round-Robin (RR)** [4]
- **Shortest Job First (SJF)**: run jobs in increasing order of  $r_i$ . [5]
  - minimizes average *turnaround* time
  - long jobs may starve
  - pre-emptive variant: **Shortest Remaining Time First (SRTF)** [6]

# Summary: CPU Scheduling

- With CPU scheduling we look for
  - *responsiveness*: ensure that threads get to run regularly
  - *fairness*: sharing of the CPU
  - *efficiency*: account for the fact that there is a cost to switching [7]
- CPU schedulers are *expected to consider process and thread priorities*. [7.1]
- There are two approaches to scheduling with priorities
  1. schedule the highest priority thread: **Multi-level Feedback Queues (MLFQ)** [8–20.1]
  2. weighted fair sharing: **Completely Fair Scheduler (CFS)** [21–23]

# Summary: Multi-level Feedback Queues (MLFQ)

With Multi-level Feedback Queues [8–20.1]

- There are  $n$  levels.
- Each level has its own **Ready** and **Run** queue.
- *Blocked threads* start at the highest level.
- The *highest level* ( $Q_n$ ) has the *highest priority* and then *smallest quantum*.
- When *preempted*, threads drop down to the next level.
- The lowest level has the *lowest priority* and the *largest quantum*.
- To prevent starvation, all threads are periodically placed in the highest-priority queue.
- Since *interactive threads tend to block frequently*, they will tend to stay in the higher-priority queues.
- Non-interactive threads will tend to sift down towards the bottom.

# Summary: Linux Completely Fair Scheduler (CFS)

With the Linux Completely Fair Scheduler [21–23]

- Each thread is assigned a **weight**,  $w_i$ .
- The goal of the scheduler is to ensure that each thread's actual share of the processor ( $a_i$ ) is *proportional to its weight*, i.e.  $\frac{a_i}{w_i}$  is the same for all threads.
- The goal is met by tracking the **virtual run time** of each runnable thread, namely  $a_i \frac{\sum_j w_j}{w_i}$  and then dispatching the thread that has *the lowest virtual run time to run next*.
- When a *thread becomes runnable*, its virtual run time is initialized to some value between the min and max virtual run times of the threads that are already runnable.
- The *quantum is the same* for all threads and priorities.

# Summary: Scheduling on Multi-Core Processors

For multi-core processors, you can have

- one **Ready** queue per core or
- one shared **Ready** queue for all the cores. [24]

Per Core designs

- *scale* to a larger number of cores, [25]
- *benefit from cache affinity*, [25-26]
- need some mechanism for **thread migration** to address load imbalances. [26]

# Devices and I/O

**key concepts:** device registers, device drivers, program-controlled I/O, DMA, polling, disk drives, disk head scheduling

Lesley Istead, Kevin Lanctot

David R. Cheriton School of Computer Science  
University of Waterloo

Winter 2019

# Devices

- **devices** are how a computer *receives input and produces output*
  - a *keyboard* is an input device
  - a *printer* is an output device
  - a *touch screen* is both input and output
- sys/161 example devices:
  - timer/clock - current time, timer, beep
  - disk drive - persistent storage
  - serial console - character input/output
  - text screen - character-oriented graphics
  - network interface - packet input/output

keyboards, mice, WiFi cards, speakers, printers, graphics cards, USB drives, joysticks, keyloggers, DVD drives, card readers, sound cards, ... are all devices.

# Device and Buses

- A **bus** is a *communication pathway between various devices* in a computer.
- There are two major types of buses in a computer.
  1. The **internal bus**, memory bus or front side bus, is for communication between the processors and RAM. It is very fast and must be close to the processor(s).
  2. **Peripheral**, or expansion buses, allow other devices in the computer (such as the network interface card, optical disk drive) to communicate.
- A **bridge** connects two different buses.
- There are a variety of ways of arranging these buses. For a (constantly changing) view of these arrangements, just google “intel chipset” or “intel chipset diagram.”

## Device Register Example: Sys/161 timer/clock

- Communication with devices carried out through **device registers**
- There are three primary types of device registers
  1. **status**: tells you something about the device's current state.  
Typically, a status register is *read*.
  2. **command**: issue a command to the device by *writing* a particular value to this register.
  3. **data**: used to transfer larger blocks of data to/from the device.
- Some device registers are combinations of primary types.
  - A **status and command** register is read to discover the device's state and written to issue the device a command.
  - A **data buffer** is sometimes combined or other times it is separated into data in and data out buffers.

## Device Register Example: sys/161 timer/clock

Offset	Size	Type	Description
0	4	status	current time (seconds)
4	4	status	current time (nanoseconds)
8	4	command	restart-on-expiry
12	4	status and command	interrupt (reading clears)
16	4	status and command	countdown time (microseconds)
20	4	command	speaker (causes beeps)

Where Size is in bytes.

The clock is used in preemptive scheduling.

## Device Register Example: Serial Console

Offset	Size	Type	Description
0	4	command and data	character buffer
4	4	status	writeIRQ
8	4	status	readIRQ

If a write is in progress, the device exhibits **undefined** behaviour if another write is attempted.

# Device Register Example: Serial Console

## Sys/161 timer/clock

- Write a time into offset 16, `countdown time`.
- Write 1 into offset 8, `restart-on-expiry`.
- *A timer interrupt will occur* when `countdown time` reaches 0.

## Serial Console

- The buffer is used to *read or write one character at a time*.
- Writing to the `character buffer` initiates a write operation.
- The device uses a `writeIRQ` (interrupt request) register to indicate when a write is complete.
- The driver clears `writeIRQ` to acknowledge completion.
- The device generates an interrupt when an incoming character becomes available or when a write operation completes.

Ref:

<http://os161.eecs.harvard.edu/documentation/sys161-2.0.8/devices.html>

- A **device driver** is a part of the kernel that interacts with a device,
  - e.g. write a char to a serial output device
- A significant portion of an OS (70% in the case of Linux) are the device drivers.
- *Communication happens by reading from or writing to* the command, status and data registers.
- Two methods for interacting with devices are...
  1. **Polling:** the kernel driver repeatedly checks the device status
  2. **Interrupts:** the kernel does not wait for the device to complete the command. Instead, request completion is taken care of by the interrupt handler.

I.e. the device updates a status register to indicate whether or not it was successful and then generates an interrupt.

# Device Drivers

An example of writing a character to a serial output device *using polling*.

```
// only one writer at a time
P(output device write semaphore)
// trigger the write operation
write character to device data register
repeat {
    read writeIRQ register
} until status is "completed"
// make the device ready again
clear writeIRQ register to acknowledge completion
V(output device write semaphore)
```

Although the majority of device drivers are a (dynamically loadable) part of the kernel, some exist in user-space.

# Using Interrupts to Avoid Polling

An example of writing a character to a serial output device *using interrupts*. There are two separate routines for writing a character.

## 1. Device Driver Write Handler:

```
// ensure only one writer at a time  
P(output device write semaphore)  
// trigger write operation  
write a character to the device data register
```

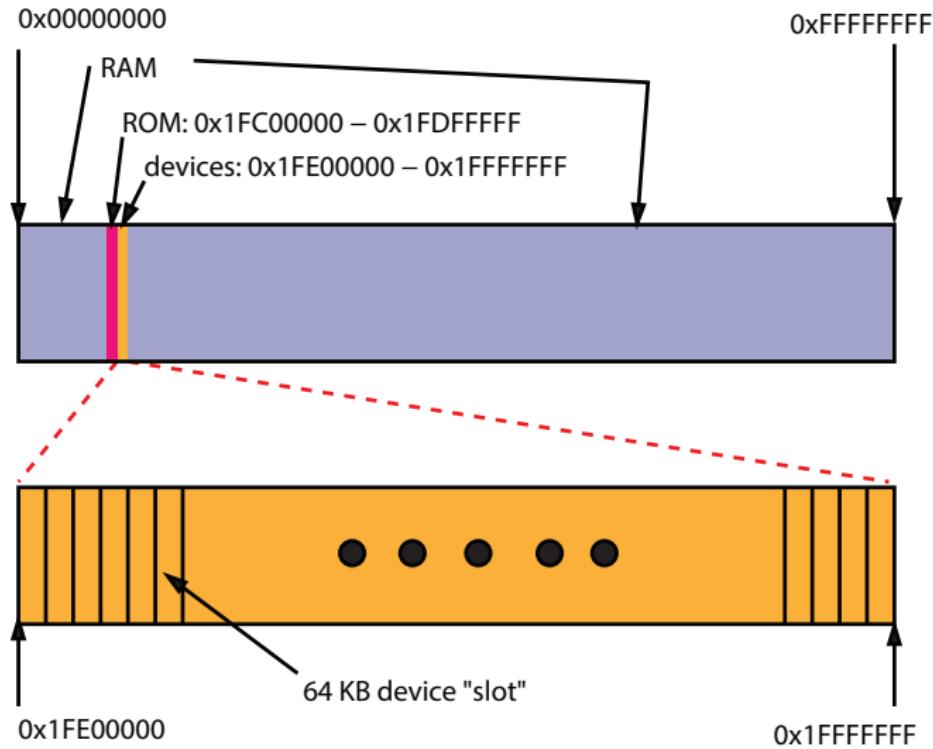
## 2. Interrupt Handler for Serial Device:

```
// make the device ready again  
clear writeIRQ register to acknowledge completion  
V(output device write semaphore)
```

Here the kernel does not wait for the device to complete the command. A semaphore is used to ensure the device is ready to accept input.

- *Key Question:* How can a device driver access the device registers?
- Option 1: **Port-Mapped I/O**
  - *Uses special assembly language I/O instructions*
  - Device registers are assigned *port* numbers, which correspond to regions of memory in a separate, smaller address space.
  - Special I/O instructions, such as the `in` and `out` instructions on x86 are used to transfer data between a specified port and a CPU register.
- Option 2: **Memory-mapped I/O**
  - *Each device register has a physical memory address.*
  - Device drivers can read from or write to these device registers using normal load and store instructions, as though accessing memory.
- A system may use both port-mapped and memory-mapped I/O.

# MIPS/OS161 Physical Address Space



- *A range of physical memory is reserved for devices*, e.g. 0x1FE0 0000 to 0x1FFF FFFF for OS/161.
- This range is further divided up into 32 *slots* each of 64KB in size.
- Each device is assigned to one of the 32 device *slots*.
- A device's registers and data buffers are memory-mapped into its assigned slot.
  - A simple device such as a timer will only need a few bytes.
  - A devices that transfers lots of data, such as a HDD or a WiFi card will use a lot more of that range.
  - Often devices can store (or buffer) several items.
  - E.g. store up to 16 key presses from the keyboard.

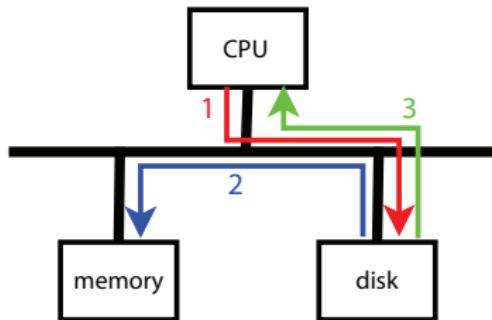
# Large Data Transfer To/From Devices

In addition to port and memory mapped I/O, large data blocks can be transferred using other strategies.

- **program-controlled I/O (PIO)**
  - The device driver moves the data between memory and a buffer on the device.
  - The CPU is *used to transfer the data.*
- **direct memory access (DMA)**
  - The device itself is responsible for moving data to/from memory.
  - The CPU is *not used to transfer the data.* and is free to do something else.
- **SYS/161** LAMEbus devices use program-controlled I/O.

# Large Data Transfer To/From Devices

DMA is used for block data transfers between devices (e.g., a disk controller) and memory.



1. The processor initiates the data transfer (i.e. issues a DMA request to the disk controller).
2. *The controller (not the processor) directs the data transfer.*
3. The controller interrupts the processor when the transfer is complete.

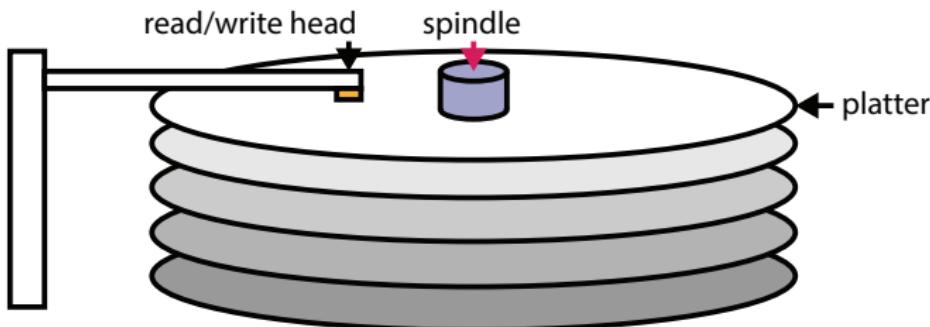
# Persistant Storage Devices

- **Persistant storage** is any device where *data persists even when the device is without power.*
  - physical memory is not persistant
  - a hard disk is persistant
  - also referred to as **non-volatile**
- persistant storage comes in many forms
  - punched cards of metal or paper (1700s-1970s)
  - magnetic drums (1930s-1960s), tapes (1920s)
  - floppy disks (1970s-2000s) and hard disks (1950s)
  - CDs (1980s), DVDs (1990s), Blu-ray (2000s)
  - solid state memory (1970s, 1990s)

One of the earliest form of persistant storage was punched cards which held the "programs" for Jacquard weaving looms in the 1800s.

# Hard Disks

- HDDs are a commonly used persistent storage device.
- They contain a number of spinning, ferromagnetic-coated platters read/written by a moving R/W head.



- Often called mechanical disks, both platter and read/write head must move to perform a read or write operation.
- This motion is takes on the order of milliseconds.

# Physical View of a Hard Drive

A hard disk drive (HDD), a.k.a. a hard drive (HDD), consists of ...

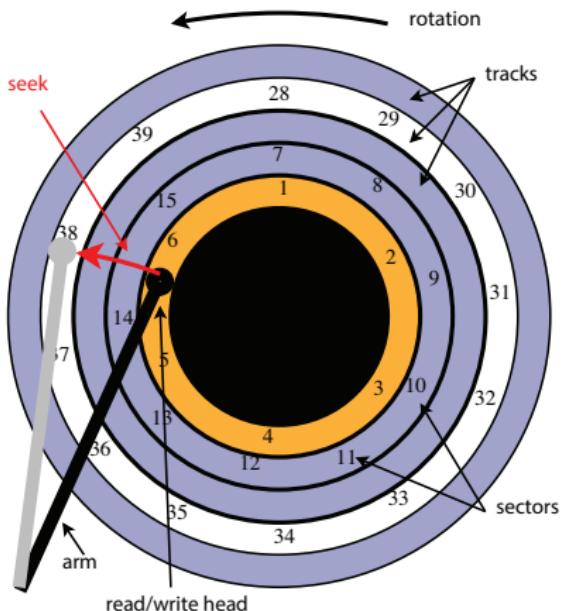
- A small number (e.g. 1 to 4) of **platters** (or disks) made of glass or porcelain.
- Each platter has two **surfaces**.
- Each surface has a **disk head** (also called a **read/write head**) associated with it to read and write data from this surface.
- The disk head is put in position by a **disk arm** (also called an **actuator arm**).
- The surface is broken up into a series of concentric circles called **tracks** if you are talking about one surface or **cylinders** if you are talking about the same radius on each surface.
- Each track is broken up into a series of arcs called **sectors** or **blocks**.

# Physical View of a Hard Drive

Here are three YouTube videos that show different aspects of the operation of a hard drive.

- This video (0:00–2:15) shows the parts of a hard drive:  
<https://www.youtube.com/watch?v=kdmLvl1n82U>
- This video (0:00–0:55) shows a hard drive in action, e.g. booting up, deleting a folder, etc:  
<https://www.youtube.com/watch?v=9eMWG3fwEU>
- This video discusses some of the innovations behind hard drives:  
<https://www.youtube.com/watch?v=wteUW2sL7bc>

# Logical View of a Disk Drive



- Logically (i.e. virtually) a disk is an *array of numbered blocks* (or sectors).
- Each block is the same size (e.g., typically 512 bytes).
- Blocks are the unit of transfer between the disk and memory.
- Typically, one or more contiguous blocks can be transferred in a single operation.

Assume, for simplicity, that each track contains the same number of sectors. This is no longer true.

# Cost Model for Disk I/O

Moving data to/from a disk involves:

1. **seek time:** move the read/write heads to the appropriate track
  - depends on **seek distance**, the distance (in tracks) between previous and current request
  - value: 0 milliseconds to cost of max seek distance
2. **rotational latency:** wait until the desired sectors spin to the read/write heads
  - depends on rotational speed of disk
  - value: 0 milliseconds to cost of single rotation
3. **transfer time:** wait while the desired sectors spin past 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

- The disk has a total capacity of  $2^{32}$  bytes.
  - The disk has a single platter with  $2^{20}$  tracks.
  - Each track has  $2^8$  sectors.
  - The disk operates at 10000 RPM
  - It has a maximum seek time of 20 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

- The disk has a total capacity of  $2^{32}$  bytes.
- The disk has a single platter with  $2^{20}$  tracks.
- Each track has  $2^8$  sectors.
- The disk operates at 10000 RPM
- It has a maximum seek time of 20 milliseconds (ms).

d) What is the average seek time ?

$$\begin{aligned} \text{AverageSeek} &= \text{MaxSeek}/2 \\ &= 20/2 = 10 \text{ ms average seek time} \end{aligned}$$

e) What is the average rotational latency?

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

# Request Service Time Example

## Parameters

- The disk has a total capacity of  $2^{32}$  bytes.
- The disk has a single platter with  $2^{20}$  tracks.
- Each track has  $2^8$  sectors.
- The disk operates at 10000 RPM
- It has a maximum seek time of 20 milliseconds (ms).

f) What is the cost to transfer 1 sector?

$$\begin{aligned} \text{SectorLatency} &= \text{MaxLatency} / \text{NumSectorsPerTrack} \\ &= 6 / 2^8 = 6 / 256 = 0.0195 \text{ ms per sector} \end{aligned}$$

g) What is the cost to read 10 consecutive sectors from this disk?

$$\begin{aligned} \text{RequestServiceTime} &= \text{Seek} + \text{RotationalLatency} + \text{TransferTime} \\ &= 10 + 3 + 10(0.0195) = 13.195 \text{ ms} \end{aligned}$$

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

# Performance Implications of Disk Characteristics

## Some Observations

- *Larger transfers (many blocks) are more efficient* than smaller ones.  
That is, the cost (time) per byte is smaller for larger transfers.
- Why? because the time it takes to get there is amortized over many blocks of data.
- *Sequential I/O is faster* than non-sequential I/O
  - Sequential I/O operations eliminate the need for (most) seeks

## Implications

- While sequential I/O is not always possible, we can group requests to try and reduce average request time

- Historically, *seek time is the dominating cost*.
- High-end drives can have maximum seek times around 4 ms, whereas consumer grade drives typically have seek times of 9–12 ms.

# Disk Head Scheduling

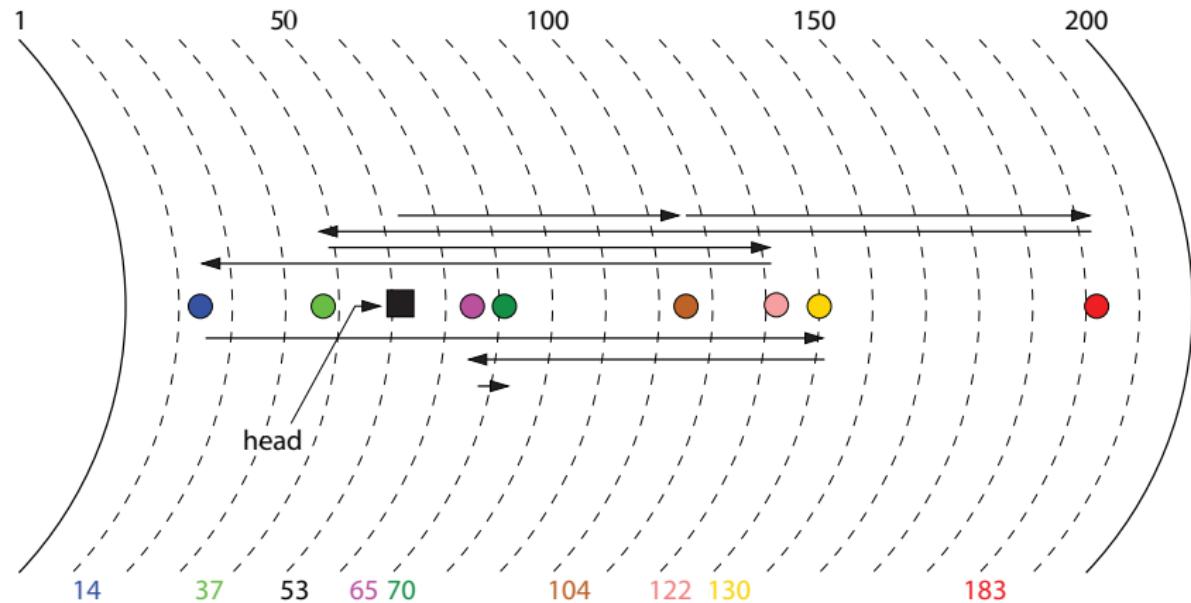
- *Goal: reduce seek times by controlling the order in which requests are serviced.*
- Disk head scheduling may be performed by the device, by the operating system, or by both.
- For disk head scheduling to be effective, there must be a queue of outstanding disk requests (otherwise there is nothing to reorder).
- In reality, the disk request queue is not static, i.e. new requests are being added before older requests have been processed.
- For the examples on the next few slides assume that
  - We are not concerned about rotational latency.
  - The disk request queue (in order of arrival) is:  
104 183 37 122 14 130 65 70

We will consider three methods...

# Disk Head Scheduling: First-Come, First Served (FCFS)

*Policy:* Service the requests in the order they arrive.

*Tradeoffs:* fair and simple, but offers no optimization for seek times.

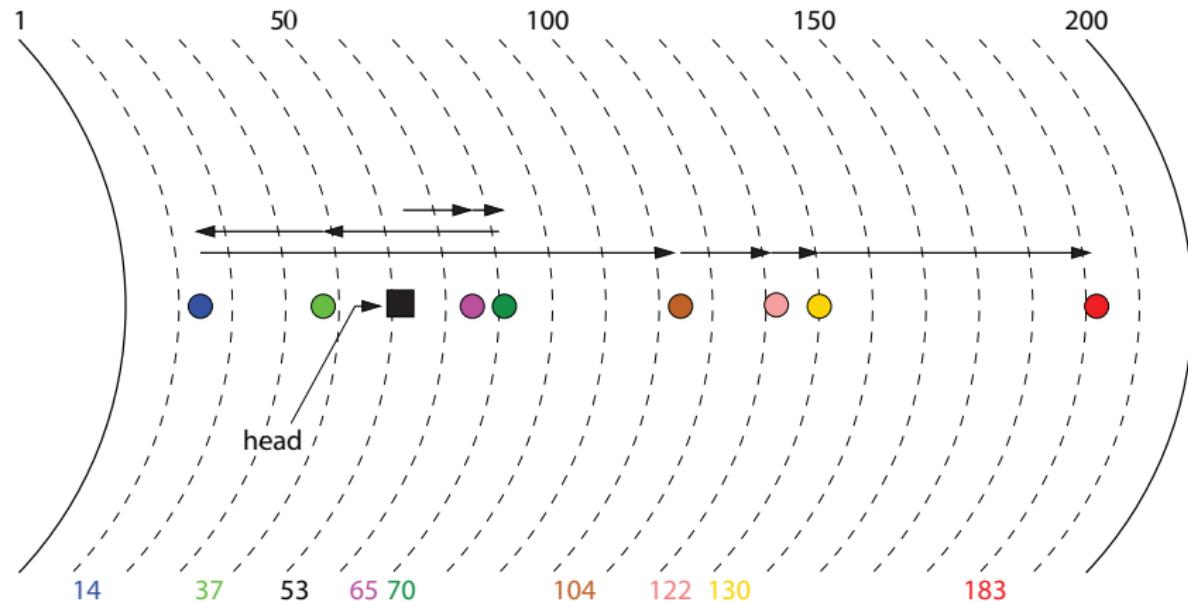


The service order is: 104 183 37 122 14 130 65 70

# Disk Head Scheduling: Shortest Seek Time First (SSTF)

*Policy:* Service the closest request first (a greedy approach).

*Tradeoffs:* Seek times are reduced, but some requests may starve.

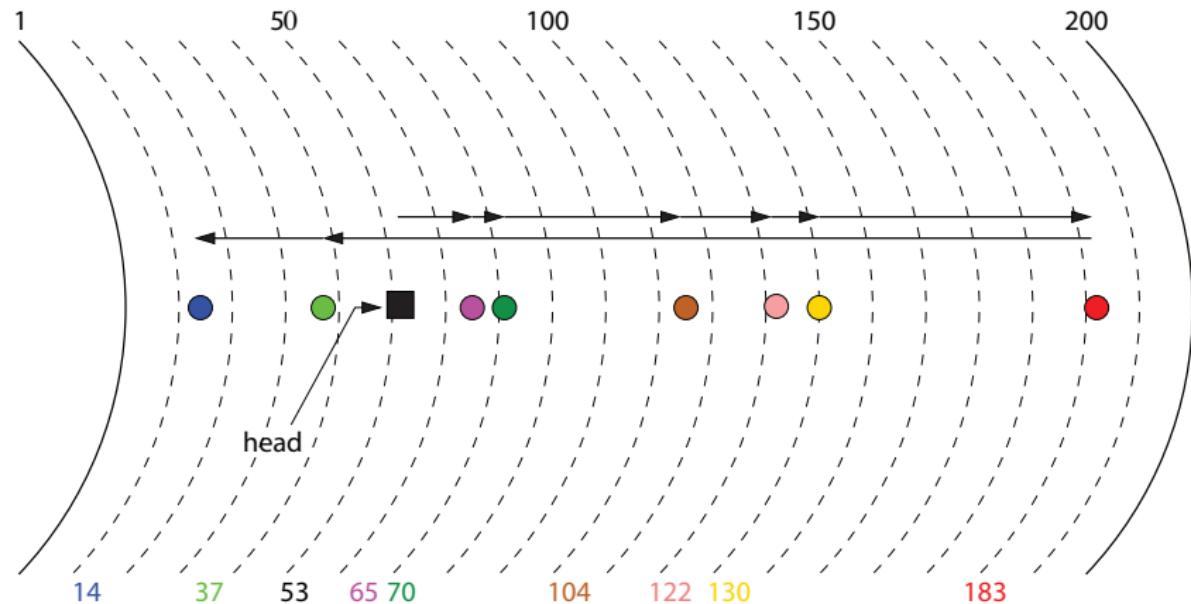


The service order is: 65 70 37 14 104 122 130 183

# Elevator Algorithms (SCAN)

*Policy:* The disk head moves in one direction until there are no more requests in front of it, then it reverses direction.

*Tradeoffs:* Reduces seek times (relative to FCFS) and avoids starvation.



The service order is: 65 70 104 122 130 183 37 14

## Device Register Example: Sys/161 disk controller

Offset	Size	Type	Description
0	4	status	number of sectors
4	4	status and command	status
8	4	command	sector number
12	4	status	rotational speed (RPM)
32768	512	data	transfer buffer

- *To perform an operation:* store the sector number into the **sector number** register and then write either the **read-in-progress** or **write-in-progress** value into the **status** register.
- The **status** register reports the present state of the disk.
- When the disk controller is reporting a completed operation, the IRQ line is raised.

# Writing to a Sys/161 Disk

## Device Driver Write Handler

```
// only one disk request at a time
P(disk semaphore)
copy data from RAM to device transfer buffer
write target sector number to sector number register
write write command to the status register
// wait for request to complete
P(disk completion semaphore)
V(disk semaphore)
```

## Interrupt Handler for Disk Device

```
// make the device ready again
write disk status register to ack completion
V(disk completion semaphore)
```

# Reading From a Sys/161 Disk

## Device Driver Read Handler

```
// only one disk request at a time
P(disk semaphore)
write target sector number to sector number register
write read command to the status register
// wait for request to complete
P(disk completion semaphore)
copy data from device transfer buffer to RAM
V(disk semaphore)
```

## Interrupt Handler for Disk Device

```
// make the device ready again
write disk status register to ack completion
V(disk completion semaphore)
```

- **Key Point:** Use integrated circuits for persistent storage instead of magnetic surfaces ⇒ no mechanical parts.
- DRAM (dynamic RAM) requires constant power to keep values but **Flash Memory** uses quantum properties of electrons.
- Flash memory is divided into blocks and pages
  - 2, 4 or 8KB pages
  - 32KB–4MB blocks
- For flash memory *reads/writes at page level*
  - Pages are initialized to 1s.
  - Transition 1 → 0 can occur at the page level (i.e., write new page).
- but *overwriting/deleting must be done at the block level.*
  - A high voltage is required to switch 0 → 1.
  - Can only apply high voltage at the block level.

# Writing and Deleting from Flash Memory

- Naive Solution (slow):
  - read whole block into RAM
  - re-initialize block (all page bits back to 1s)
  - update block in RAM and write back to SSD
- SSD controller handles requests (faster):
  - the page to be deleted/overwritten is *marked as invalid*
  - write to an unused page
  - update translation table
  - requires garbage collection
- SSD limitations
  - Each block of an SSD *has a limited number of write cycles* before it becomes read-only.
  - SSD controllers perform **wear leveling**, distributing writes evenly across blocks, so that the blocks wear down at an even rate.
  - Hence defragmenting can be harmful to the lifespan of an SSD.

# Persistent RAM

- New forms of solid state non-volatile memory are being developed.
- Data is preserved (persists) in the absence of power.
  - ReRAM: resistive RAM
  - 3D XPoint, Intel Optane
- They can be used to *improve the performance of secondary storage.*
  - Traditional CPU caches are small; not effective for caching many disk blocks worth of data.
  - Volatile RAM can cache file system information but is best reserved for the address spaces of processes.
  - *Key Idea: Cache file system information* in persistent RAM.
    - E.g. With Intel Optane, modules are 16-32GB, so many blocks can be cached giving big performance improvements when mechanical disks are used.

## Summary: Interacting with Devices

- **Devices** are how a computer receives input and produces output. [2]
- A **bus** is a communication pathway between various devices. [2.1]
- Communication with devices carried out through **device registers**. [3]
  - There are three types: **status**, **command**, and **data** registers. [3]
- A **device driver** is a part of the kernel that interacts with a device. [5.2]
- Two methods for interacting with devices are... [5.2]
  1. **Polling:** the kernel driver repeatedly checks the device status.
  2. **Interrupts:** the device generates an interrupt when done.

# Summary: Reading, Writing and Storing Data

- Two methods for accessing device registers are ... [8]
  1. **Port-Mapped I/O** using special assembly language I/O instructions.
  2. **Memory-mapped I/O** each device register has a physical memory address.
- Large data blocks can be transferred using ... [10]
  1. **program-controlled I/O (PIO)** the device driver (using the CPU) moves the data
  2. **direct memory access (DMA)** the device itself (without the CPU) moves the data
- **Persistent storage** (or **non-volatile storage**) is any device where the data persists even when the device is without power. [11]
- A Hard Disk Drive consists of **tracks**, **sectors** and **blocks**. [13]
- Logically a HDD is viewed as an *array of numbered blocks*. [13]

- Request Service Time  
= **Seek Time + Rotational Latency + Transfer Time** [14]
- Larger transfers are more efficient than smaller ones. [17]
- Sequential I/O is faster than non-sequential I/O. [17]
- The three methods of disk head scheduling are... [18–20]
  1. **First Come First Served** offers no optimization.
  2. **Shortest Seek Time First** seek times are reduced, but some requests may starve.
  3. **Elevator Algorithms** reduces seek times and avoids starvation.
- **Flash Memory** has no mechanical parts but requires **wear leveling** since each block can only be written to a limited number of times. [24–25]
- **Persistent RAM** can be used to improve the performance of secondary storage by caching file system information. [26]

# File Systems

**key concepts:** file, directory, link, open/close, descriptor, read, write, seek, file naming, block, i-node, crash consistency, journaling

Lesley Istead, Kevin Lanctot

David R. Cheriton School of Computer Science  
University of Waterloo

Winter 2019

# Files and File Systems

**files**: *persistent, named data objects*

- when you edit a file, the change is not *persistent* until you save it
- in the expression “ $c = 2*(a+b)$ ”
  - $a$  is a *named data object*
  - $(a+b)$  is not a named object. It is an *intermediate result*.
- data consists of a sequence of numbered bytes
  - i.e. you can ask for the  $i^{\text{th}}$  byte
  - no other assumptions are made about the data: it could be text, images, video, machine code, etc
- files may change size and content over time
- files have associated meta-data (e.g., owner, file type, date created, date modified, access controls)
  - e.g. “`ls -l`” in Linux or right-click Properties in Windows.

- **File systems:** the data structures and algorithms used to store, retrieve, and access files.
- A file system can be separated out into three different layers (although sometimes the first two are combined into one layer).
  1. **logical file system:**
    - High-level API, what a programmer sees (e.g. fopen, fscanf, fprintf, fclose).
    - Manages file system information, e.g. which files are open for reading and writing, which are read-only.
  - 2 **virtual file system:** abstraction of lower level file systems, presents multiple different underlying file systems (HDD, SDD, DVD, network drive) to the user as one.
  3. **physical file system:** how files are actually stored on physical media (e.g. track, sector, magnetic orientation).

# File Interface: Basics

Some common file operations are

- `open`
  - `open` *returns a file descriptor* (or identifier or handle),
  - other file operations (e.g., `read`, `write`, `seek`, `close`) for that process require this file descriptor as an parameter in order to identify the file
- `close`
  - `close` *invalidates a valid file descriptor* for a process
  - the kernel tracks which file descriptors are currently valid for each process
- `read`, `write`, `seek`
  - `read` copies data from a file into a virtual address space
  - `write` copies data from a virtual address space into a file
  - `seek` enables non-sequential reading/writing
- get/set file meta-data, e.g., Linux `fstat`, `chmod`

# File Position and Seek

- *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
  - update the current file position as bytes are read/written
- this makes sequential file I/O easy for an application to request
- seeks (`lseek`) are used for achieve non-sequential file I/O
  - `lseek` changes the file position associated with a descriptor
  - next read or write from that descriptor will use the new position

`open`, `close`, and `lseek` are Linux/UNIX system calls.

`fopen`, `fclose`, and `fseek` are ANSI C functions that are portable to other OSs.

## Sequential File Reading Example

```
char buf[512];
int i;
int f = open("myfile", O_RDONLY);
for(i=0; i<100; i++) {
    read(f, (void *)buf, 512);
}
close(f);
```

- Read the first  $100 * 512$  bytes of a file, 512 bytes at a time using the Linux system call `open`.
- The `open` system call provides a number of options in order to specify if the file is open for reading (`O_RDONLY`), writing (`O_WRONLY`), both (`O_RDWR`) or for appending (`O_APPEND`).

## File Reading Example Using Seek

```
char buf[512];
int i;
int f = open("myfile", O_RDONLY);
for(i=1; i<=100; i++) {
    lseek(f, (100-i)*512, SEEK_SET);
    read(f, (void *)buf, 512);
}
close(f);
```

- Read the first 51,200 bytes, 512 bytes at a time, *in reverse order*.
- `lseek` does not modify the file.
- `lseek` *does not check if the new file position is valid*, i.e. it will not return an error or throw an exception if the position is invalid.
- However, the subsequent read or write operation will.

# Directories and i-numbers

## Directories

- A **directory** *maps file names (strings) to i-numbers*
  - an **i-number** (index number) is a unique (within a file system) *identifier for a file or directory*
  - given an i-number, the file system can find the data and meta-data for the i-number's corresponding file
- Directories provide a way for applications to group related files

## Directories as Trees

- Since directories can be nested, a filesystem's *directories can be viewed as a tree*, with a single **root** directory.
- In a directory tree,
  - files are always leaves and
  - directories can be interior nodes (if they are non-empty) or leaves (if they are empty).

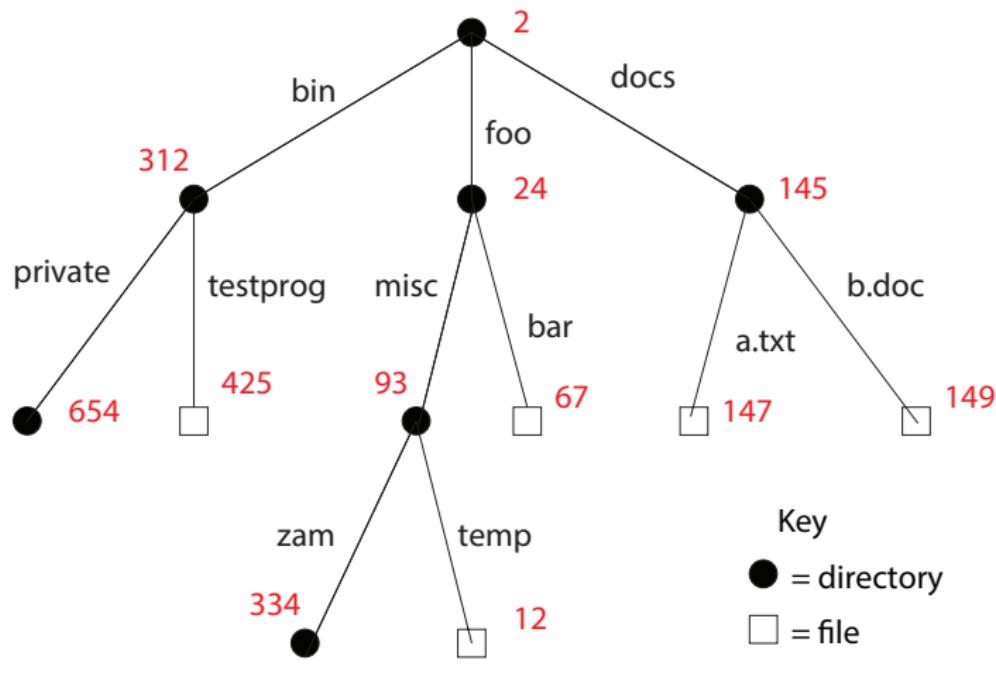
# File Names and Pathnames

- Files may be identified by **pathnames**, which *describe a path through the directory tree from the root directory to the file*, e.g.:  
`/home/user/courses/cs350/notes/filesys.pdf`
- Directories also have pathnames.
- Applications (and humans!) refer to files using pathnames, not i-numbers.

## Question

- Only the kernel is permitted to directly edit directories. Why?
- In general, the kernel should not trust the user with direct access to data structures that the kernel relies on.

# Hierarchical Namespace Example



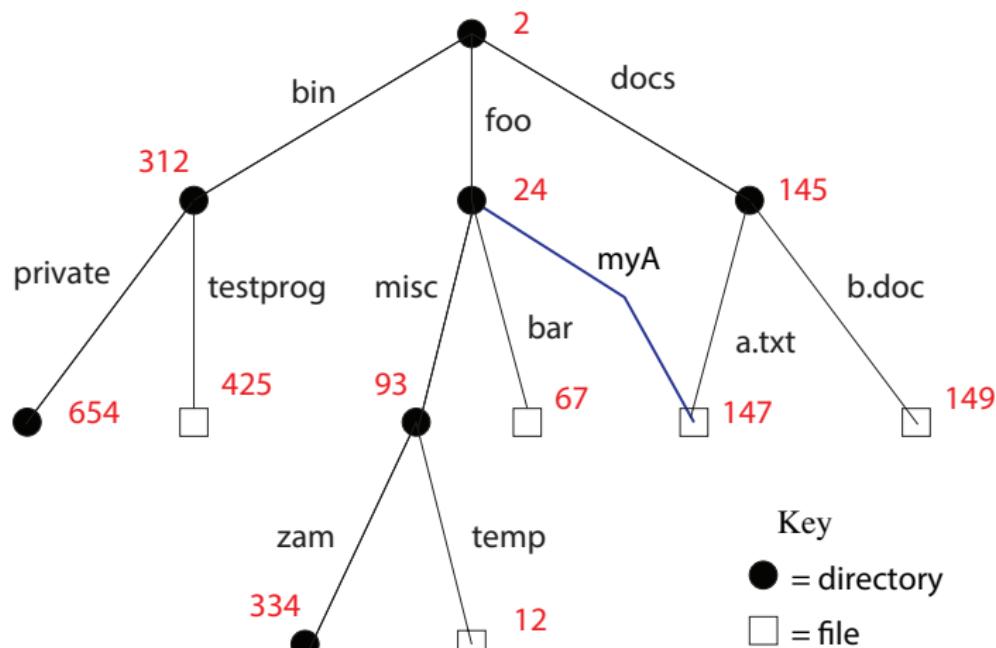
/docs/b.doc is the path for the file with i-number 149.

# Links

- A **hard link** is *an association between a name (string) and an i-number.*
  - Each entry in a directory is a hard link.
- When a file is created, a hard link to that file is also created,  
e.g.: `open("/docs/a.txt", "O_CREAT|O_TRUNC")`
  - This command opens the file and creates a hard link to that file in the directory `/docs`
    - `O_CREAT` means if this file does not exist then create it.
    - `O_TRUNC` means overwrite the existing contents (if the file is writable).

- Once a file is created, *additional hard links can be made to it.*
- E.g. in C: `link("/docs/a.txt", "/foo/myA")`
- E.g. in Linux : `link /docs/a.txt /foo/myA`
  - creates a new hard link `myA` in directory `/foo`.
  - The link refers to the i-number of file `/docs/a.txt` (**147**) which must exist.
- Linking to an existing file *creates a new pathname for that file.*
- Each file
  - has a unique i-number,
  - but may have multiple pathnames.
- In order to avoid cycles, it is not possible to `link` to a directory.

# Hierarchical Namespace Example



Key  
● = directory  
□ = file  
234 = i-number

/foo/myA and /docs/a.txt are two different paths to file 147

# Unlinking

- *Hard links can be removed:*
  - E.g. in C: `unlink("/docs/b.doc")`
  - E.g. in Linux: `unlink /docs/b.doc`
  - This removes the link `b.doc` from the directory `/docs`
- The file system ensures that hard links have **referential integrity**, which means that *if the link exists, then the file that it refers to also exists.*
  - When a hard link is created, it refers to an existing file.
  - The kernel keeps a count of how many hard links there are to an existing file.
  - A file is deleted when its last hard link is removed (and the count of the number of hard links become zero).
    - Since there are no links to the file, it has no pathname and could no longer be opened.

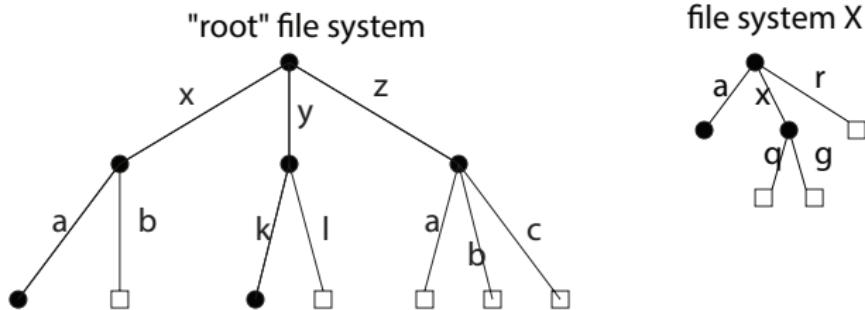
# Multiple File Systems

- It is not uncommon for *a system to have multiple file systems.*
  - For example, the `df -T` command in Linux.
  - Plugging in a USB flash drive into your laptop.
  - Accessing network drives or file servers.
- Some kind of **global file namespace** is required, i.e.
  - uniform across all the file system,
  - independent of physical location.
- Two Examples:
  - Windows:** Use two-part file names: file system name, pathname within file system, e.g.: `C:\user\cs350\schedule.txt`
  - Linux:** Create a single hierarchical namespace that combines the namespaces of multiple file systems, e.g. the `mount` system call.

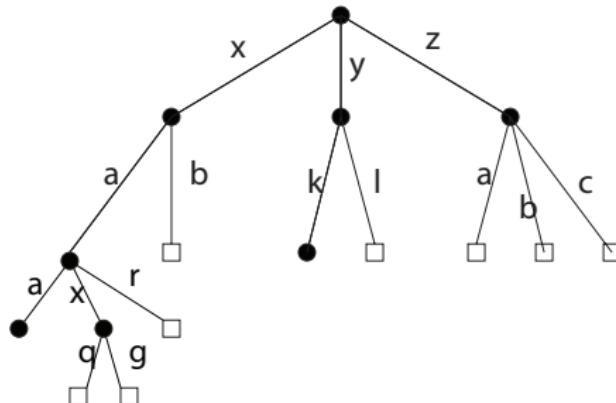
# Multiple File Systems

- Mounting does *not* make two file systems into one file system.
  - It merely *creates a single, hierarchical namespace* that combines the namespaces of two file systems.
  - The new namespace is temporary - it exists only until the file system is unmounted.
- In Windows, when you plug in a USB flash drive it is **auto-mounted** and assigned a letter.
- Linux is more flexible: you may auto-mount or specify where it is mounted, e.g. mount the file system X in the directory `/x/a`.
  - Before mounting: `/x/a` is in the root file system and `/x/g` is in the X file system.
  - After mounting: `/x/a/x/g` is in the resulting file system.
  - See next slide for a diagram illustrating this mount.

# Unix mount Example



result of mount (file system X, /x/a)



# File System Implementation

## Key Questions

- What needs to be *stored persistently*?
  - file data (i.e. the actual contents of the file)
  - file meta-data (e.g. creating data, access permissions)
  - directories and links
  - file system meta-data
- What information is *non-persistent*?
  - per process open file descriptor table
    - file descriptor
    - file position for each open file
  - system wide:
    - open file table
    - *cached* copies of persistent data

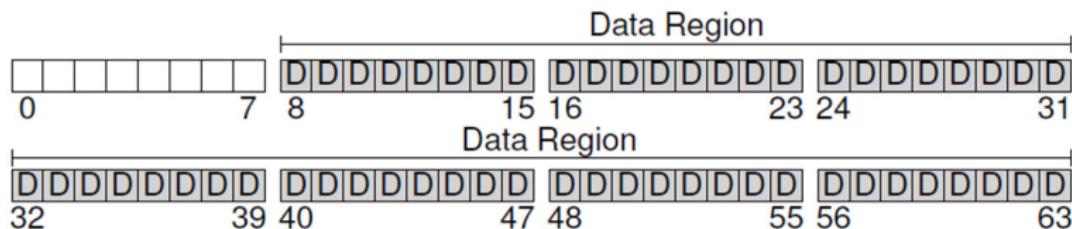
# File System Example

*Goal:* create a very small file system to get a sense of *what components are needed and how they are used.*

- Use an extremely small disk as an example: i.e. a 256 KB disk!
  - This disk will have a sector size of 512 bytes (a typical value).
    - RAM is usually **byte addressable**,  
i.e. can read or write to any byte.
    - Disks are usually **sector addressable**,  
i.e. can read or write to any sector.
  - For this 256 KB disk we will have 512 total sectors.
- Group every 8 consecutive sectors into a block (of size 4KB)
  - Better spatial locality (fewer seeks).
  - Reduces the number of block pointers  
(we'll see what this means soon).
  - A 4 KB block is a convenient size for demand paging.
  - There are a total of 64 blocks on this disk.

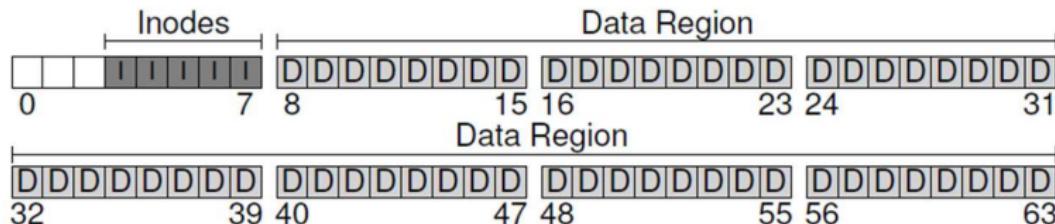
# VSFS: Very Simple File System (1 of 5)

- **Key Task:** decide (ahead of time) how much *space to reserve for data vs. meta-data* (e.g. owner, creation date).
- This decision will depend in part on how much meta-data you track for each file, etc.
- **Goal:** Most of the blocks should be for storing file's contents.
- In this case we'll reserve the first 8 blocks for meta-data and the last 56 blocks for data (i.e. the actual contents of the file).



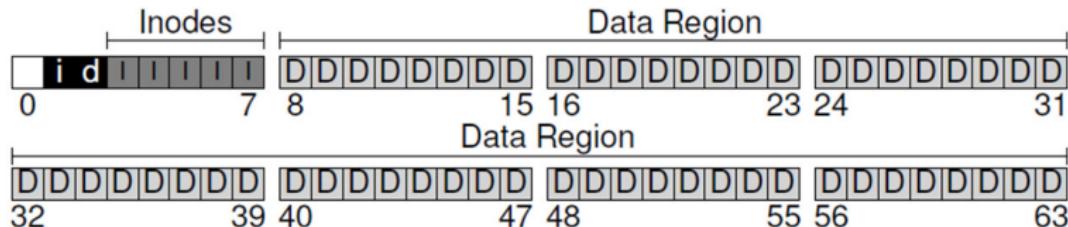
## VSFS: Very Simple File System (2 of 5)

- **Key Task:** Need some way to *map files to their data blocks*.
- Create an array of **i-nodes**, where each i-node contains the meta-data for a file (more on i-nodes later).
  - The index into the array is the file's index number (i-number).
- Use 256 bytes for each i-node and dedicate 5 blocks for i-nodes
  - This choice allows for 80 total i-nodes (and hence at most 80 files since there is exactly one i-node per file).



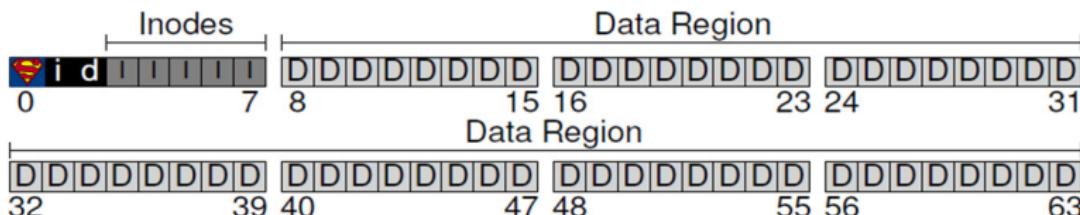
## VSFS: Very Simple File System (3 of 5)

- **Key Task:** We need to *track which i-nodes and data blocks are in use*.
- Many ways of doing this:
  - In VSFS, we will *use a bitmap* for each.
    - i:** block containing the bitmap tracking inodes in use
    - d:** block containing the bitmap tracking data blocks in use
  - Could also use a free list instead of a bitmap.
- Since there are 8 bits in a byte  $\Rightarrow$  a block size of 4 KB can track 32K i-nodes (or 32K data blocks).
  - This is far more than we actually need.
  - We only need to track at most 80 of them.



# VSFS: Very Simple File System (4 of 5)

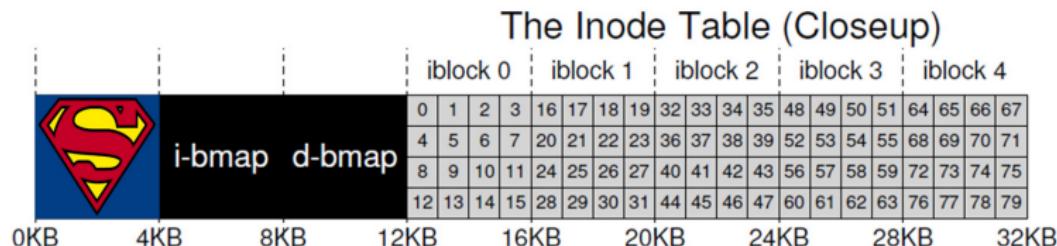
- Reserve the first block as the **superblock**.
- A superblock *contains meta-information about the entire file system.*
  - e.g., how many i-nodes and blocks are in the system, the location of the i-node bitmap, the location of the data block bitmap, the location of the i-node table, etc.



# VSFS: Very Simple File System (5 of 5)

*Summary of VSFS Format:* the first eight blocks of the file system are used for...

- 0-4KB: *superblock*: meta-data about the whole file system,
- 4-8KB: *i-bitmap* of used/free i-nodes,
- 8-12KB: *d-bitmap* of used/free data blocks,
- 12-32KB: *i-nodes*, i.e. meta-data for up to 80 files.



We could have used our space more efficiently, but we are trying to keep things simple.

- An **i-node** is a fixed size index structure (for a given file system they are all the same size) that *holds the file meta-data and pointers to the data blocks.*
- i-node fields may include:
  - file type
  - file permissions
  - file length (in bytes and number of file blocks)
  - time of last file access, last file update, last i-node update
  - number of hard links to this file
  - pointers to data blocks
- *The Challenge:* Since the i-node is a fixed size, how can it point to all the data blocks for the file?
  - For *small files*, pointers in the i-node are sufficient to point to all data blocks.
  - For *larger files* we need another approach.

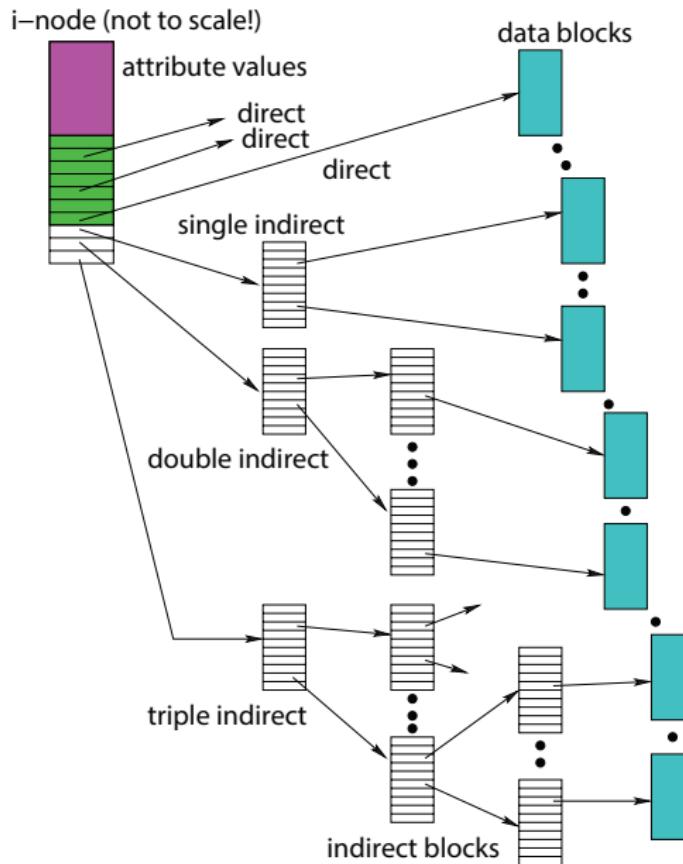
# VSFS: i-node

*Idea:* use direct and indirect block pointers.

- If disk blocks are referenced with a 4-byte (32-bit) address
  - then there are  $2^{32}$  blocks each of size 4 KB
  - so the maximum disk size is  $2^{32} \times 2^{12} = 2^{44} = 16 \times 2^{40} = 16\text{TB}$ .
- In VSFS, an i-node is 256 bytes
  - Reserve room for 12 pointers to blocks.
  - Call these **direct pointers**.
  - *Each pointer points to a different block for storing user data.*
  - Pointers are ordered:
    - 1st pointer points to the 1st block in the file, etc.
- What is the maximum file size if we only have direct pointers?  
 $12 \times 4\text{ KB} = 48\text{ KB}$
- Great for small files (which are common).
- Not so great if you want to store big files.

- In addition to the 12 direct pointers, we can also introduce an **indirect pointer**
  - An indirect pointer *points to a block full of direct pointers*.
- A 4 KB block of direct pointers holds 1024 (4-byte) pointers.
  - Maximum file size using 12 direct and 1 indirect pointers is:  
 $(12+1024) \times 4 \text{ KB} = 4,144 \text{ KB}$
- This is more than enough for any file that can fit on our tiny 256KB disk, but what if the disk was larger?
- Add a **double indirect pointer**.
  - A double indirect pointers points to a 4 KB block of indirect pointers.
  - Maximum file size using 12 direct, 1 indirect and 1 double indirect pointers is:  
 $(12+1024+1024^2) \times 4 \text{ KB} \approx 4 \text{ GB}$
- Is this enough? If not, consider using a **triple indirect pointer**.

# VSFS: Indirect Blocks



# Directories

Directories are implemented as a special type of file that contains directory entries, pairing up

- an *i-number* and
- a *file name* (the last component of a path name).

i-num	name
5	.
2	..
12	foo
13	cs341
15	cs350

- Directory files can be read by application programs (e.g., `ls`)
- Directory files are only updated by the kernel, in response to file system operations, e.g, create file, create link.

# In-Memory (Non-Persistent) Structures

In order to speed up file access, *the kernel keeps some information in RAM which is updated as processes access files.*

## *Per Process*

- Descriptor Table
  - Recall: that the system call `open` returns a file descriptor.
  - The **Descriptor Table** tracks
    - Which file descriptors does this process have open?
    - To which file does each open descriptor refer?
    - What is the current file position for each descriptor?

## *System Wide*

- **Open File Table:** files are currently open by any process
- **i-Node Cache:** in-memory copies of recently-used i-nodes
- **Block Cache:** in-memory copies of data blocks and indirect blocks

# Reading from a File (/foo/bar)

In order to understand how the various components of the VSFS work, consider the following example. **Task:** Read from the file `/foo/bar`

1. First, *the root i-node is read.*

operation	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
1. open(bar)			read							

- *Why:* The root's i-node will provide the location of root's data block which *stores the root directory*.
  - A file system will have a predefine i-node for the location of the root directory, e.g. 2.
  - In Linux i-node 1 is usually reserved for tracking bad blocks.

# Reading from a File (/foo/bar)

**Task:** Read from the file /foo/bar

1. First, the root i-node is read.
2. The root's data block is read *to find the i-number for foo.*

operation	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
1. open(bar)			read							
2.						read				

- **Why:** Recall that a directory associates a string (file or directory name) to an i-number.
- In this example, we assume that the directory fits into a single block.

# Reading from a File (/foo/bar)

**Task:** Read from the file `/foo/bar`

1. First, the root i-node is read.
2. The root's data is read to find the i-number for `foo`.
3. *Read `foo`'s i-node* which provides the location of `foo`'s data.

operation	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
1. open( <code>bar</code> )			read							
2.					read					
3.						read				

- Recall: `foo`'s i-node can be found by indexing into the array of i-nodes using `foo`'s i-number (which was obtained in step 2).

# Reading from a File (/foo/bar)

**Task:** Read from the file /foo/bar

1. The root i-node is read.
2. The root's data is read to find the i-number for `foo`.
3. Read `foo`'s i-node which provides the location of `foo`'s data.
4. *Read foo's data* (a directory) to find `bar`'s i-number.

operation	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
1. open(bar)			read							
2.					read					
3.						read				
4.							read			

- Again, for this example we assume that the data contained in directory `foo` fits into a single block.
- This fact may not always be true.

# Reading from a File (/foo/bar)

4. ...

5. bar's i-node is read and

- the permissions are checked
- a file descriptor is returned and added to the process's file

## Descriptor Table

- the file is added to the kernel's Open File Table.

operation	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
1. open(bar)			read							
2.					read		read			
3.										
4.								read		
5.					read					

- The file is now open and ready for reads and writes.
- The position of the file is byte 0.
- Opening this file required 5 disk reads!

# Reading from a File (/foo/bar)

*Task:* Reading data from `/foo/bar`, one block at a time.

6. `bar's i-node is read` and a pointer to `bar`'s 0<sup>th</sup> data block is found.

operation	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
1. open( <code>bar</code> )			read							
2.				read		read				
3.					read					
4.							read			
5.								read		
6. read()					read					

- If `bar`'s i-node is not in the `i-node cache`, it must be read from disk.

# Reading from a File (/foo/bar)

Reading data from `/foo/bar`, one block at a time.

6. `bar`'s i-node is read and a pointer to `bar`'s data block is found.
7. The data block for `/foo/bar` is read.

operation	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
1. open( <code>bar</code> )			read			read				
2.				read			read			
3.					read			read		
4.						read				
5.							read			
6. read()					read				read	
7.										

# Reading from a File (/foo/bar)

Reading data from `/foo/bar`, one block at a time.

6. `bar`'s i-node is read and a pointer to `bar`'s data block is found.
7. The data block for `/foo/bar` is read.
8. `bar`'s i-node is written to *update the access time*

operation	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
1. open( <code>bar</code> )			read			read				
2.				read			read			
3.					read			read		
4.						read				
5.							read			
6. read()					read			read		
7.							write			
8.										

# Reading from a File (/foo/bar)

9–14: *Two more data blocks are read.* I.e. the last three steps (6–8) are repeated two more times.

operation	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
1. open(bar)			read			read				
2.				read						
3.					read					
4.						read				
5.							read			
6. read()				read						
7.					read					
8.					write		read			
9. read()				read						
10.					read					
11.					write					
12. read()				read						
13.					read					
14.					write					read

- Even if the user wants a single byte out of the middle of a block, the entire block must be read.
- Disks typically do not permit byte-based addressing, only block or sector addressing.

For another example, consider how the file **bar** was created...

# Creating the File **bar** in the Directory **/foo**

operation	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
1. create(bar)			read			read				
2.				read			read			
3.					read					
4.						read				
5.							read			
6.								write		
7.									write	
8.										write
9.										
10.				write						
11. write()		read			read					
12.		write								
13.								write		
14.									write	
15						write				
16. write()		read			read					
17.		write								
18.										write
19.										
20.						write				
21. write()		read			read					
22.		write								
23.										
24.										write
25.						write				

When *writing a partial block*, that block must be read first but when *writing an entire block*, no read is required.

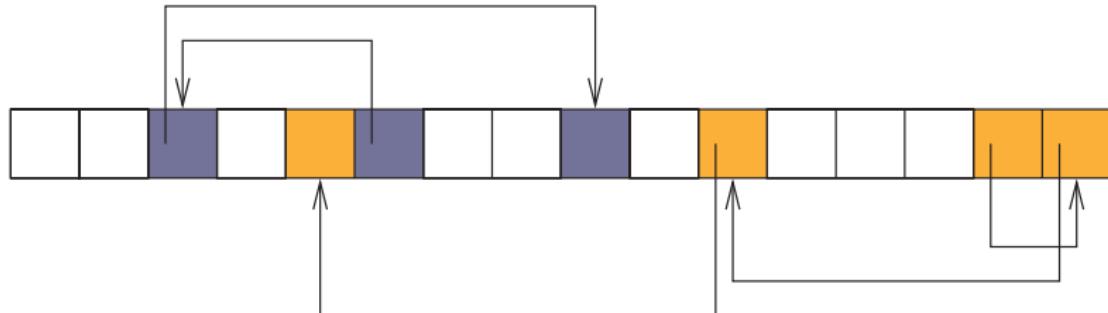
## Creating the File `bar` in the Directory `/foo`

- 1–3: these are the same as the previous case (Reading from a File), i.e. read the *root i-node*, *root data*, *foo i-node*, *foo data*.
- 4: Check that this process has write permission in `/foo`.
- 5–6: Read and then write the *i-node bitmap* to find and allocate a new i-node for `bar`.
- 7: Create a new entry in the `foo` directory by writing to `foo`'s data.
- 8–10: Initialize `bar`'s i-node and update the `foo`'s i-node with a new value for the *last time modified*.
- 11–25: For each block of data that is written, do the following
  - 12–13: Read and write to the data bitmap to *find and allocate a new data block*.
  - 11, 15: A read and write to `bar`'s i-node to *track the location of this new data block*.
  - 14: Write to the `bar`'s data block *to store the data*.

- VSFS uses a per-file index (direct and indirect pointers) to access blocks
- Two alternative approaches:
  1. **Chaining:**
    - *Each block includes a pointer to the next block.*
  2. **External Chaining:**
    - *The chain is kept as an external structure.*
    - Microsoft's File Allocation Table (FAT) uses external chaining.

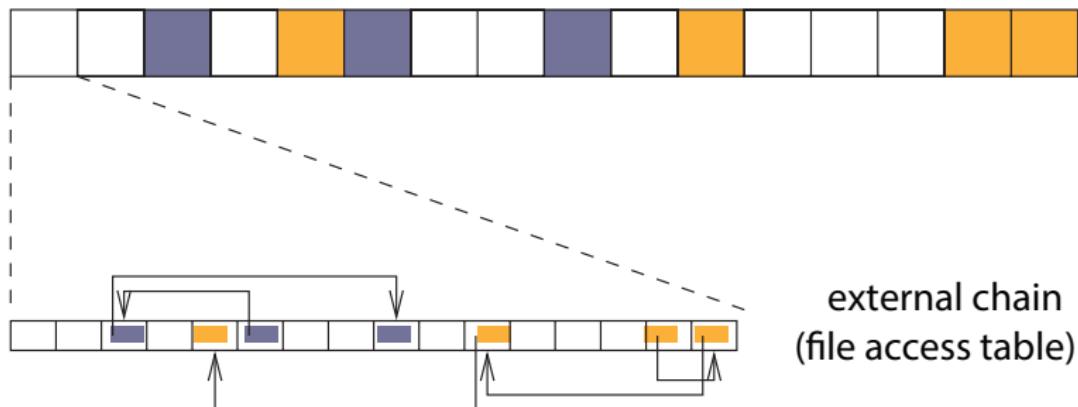
# Chaining

- *Implementation:* The directory table contains the name of the file paired with its starting block (and possibly its end block to facilitate appending).
- *Performance:* Acceptable for sequential access, very slow for random access (why?)  
⇒ You must go through the file sequentially (block by block) to get to a particular location.
- *Question:* Is there a way of improving performance?



# External Chaining

- *Idea:* Introduce a **file access table (FAT)** that specifies all of the file chains in one big table.
- *Key Benefit:* When doing random access, this approach reduces the number of blocks read by *keeping this table in its own block* (or blocks) rather than having one pointer per block.
- This table can be cached to improve performance.



# File System Design

When designing a file system there are a number of key questions to consider, such as values for the following parameters...

- How many files should the file system allow for?
- What is a good block size?
- How many direct and indirect blocks should an i-node have?

**Key Idea:** For a general purpose file system, design it to be efficient for the common case. Here are some observations...

- Most *files are small* ⇒ about 2KB but...
- the average *file size growing*.
- File systems *contain many files* ⇒ 100,000 on average.
- Typically *directories are small* ⇒ containing 20 files on average.
- Even as disks grow in capacity the *average file system usage is 50%*.

*Key Challenge:* But you should be able to handle exceptional cases.

- What if the files were mostly large, 50GB minimum?
- What if each file is less than 1KB?

I.e. we need to be able to handle a large number of small files or a few large ones.

But there is another key issue we need to consider...

# Problems Caused by Failures

- **Observation:** A single logical file system operation may require several disk I/O operations, e.g. deleting a file
  - remove entry from directory
  - remove file index (i-node) from i-node table
  - mark file's data blocks free in free space index
- **Key Challenge:** what if, because of a failure, some but not all of these changes are reflected on the disk?  
⇒ system failure could destroy in-memory file system structures
- **Key Constraint:** persistent structures should be **crash consistent**, i.e., *they should be consistent when a system restarts after a failure.*
- How can we achieve crash consistency in practice? There are two ways.
  1. **Special-purpose Consistency Checkers**
  2. **Journaling**

1. Special-purpose Consistency Checkers (e.g. `fsck` Linux ext2)
  - It runs after a crash but before normal operations resume.
  - It finds and *attempts to repair inconsistent file system data structures*, e.g.:
    - o file with no directory entry,
    - o free space that is not marked as free.
2. Journaling (e.g. NTFS, Linux ext3)
  - Use **write-ahead logging**
    - o First record the file system meta-data changes in a journal (i.e. a log file) so that the sequences of changes can be written to disk in a single operation.
    - o *After the changes have been journaled, update the disk data structures.*
  - After a failure, redo the journaled updates in case they were not completed before the failure.

# Summary: Files, Directories and File Systems

- A **file** is a persistent, named data object. [2]
- A **file system** is the data structures and algorithms used to store, retrieve, and access files. [2.1]
- The three views of a file system are [2.1]
  1. **logical file system**: the high-level API
  2. **virtual file system**: abstraction of lower level file systems, presents multiple underlying file systems as one.
  3. **physical file system**: how files are actually stored on physical media.
- A **file descriptor** is returned by open and is used subsequent operations to identify the file. [3]
- Each open file has an associated file position which can be modified by `lseek`. [4–6]
- A **directory** maps file names (strings) to **i-numbers**, a unique identifier within a file system. [7]

## Summary: Components of a File System

- A **hard link** is an association between a file name and an i-number. [9–11]
- File systems often provide a **global file namespace** which is uniform across the file system and independent of the physical location. [12]
- **Mounting** a file system creates a single, hierarchical namespace that combines the namespaces of two file systems. [12.1–13]
- A simple file system (such as VSFS) stores some information *persistently* such as the file data, the **i-nodes** (which stores the file's **meta-data**), available data blocks, available inodes, and file system meta-data. [14–21]
- A **direct pointer** points to data blocks and an **indirect pointer** points to a block full of direct pointers. [22–24]
- **Directories** pair up i-numbers with file names. [24.1]

- *Non-persistent* data structures include **Descriptor Tables**, an **Open File Table** and caches for i-Nodes and data blocks. [24.2]
- Two other methods for tracking a file's data blocks include **Chaining** and **External Chaining** using a **File Allocation Table (FAT)**. [35–37]
- Two methods for creating fault tolerant files sytems include [39]
  1. Special-purpose Consistency Checkers [40]
  2. Journaling (using **write-ahead logging**) [40]