CS 350: Operating Systems

University of Waterloo

Instructor: Kevin Lanctot

Spring 2023

Andrew Wang

Table of Contents

Introduction to Operating Systems **5**
- Three Views of an Operating System . . . 5
  - Application View . . . 5
  - System View . . . 6
  - Implementation View . . . 6
- Operating System and the Kernel . . . 6
  - Schematic View of an Operating System . . . 7
  - Types of Kernels . . . 7
  - Operating System Abstractions . . . 8

Multiprogramming **9**
- Threads . . . 9
  - OS/161's Thread Interface . . . 10
  - Low Level View of Threads . . . 10
  - Concurrent Threads . . . 12
  - Context Switch in MIPS . . . 13
  - Thread States . . . 14
  - Yielding . . . 15
  - Interrupts . . . 15
  - Scheduling . . . 16
  - Two Threads Example . . . 17

Synchronization **21**
- Mutual Exclusion . . . 21
  - Critical Section Example . . . 22
  - Spinlocks . . . 23
  - Locks . . . 25
  - Wait Channels . . . 26
  - Semaphores . . . 27
- Producer and Consumer . . . 28
  - Condition Variables . . . 29
- Other Synchronization Stuff . . . 31
  - Deadlocks . . . 31
  - Other Sources of Race Conditions . . . 32
  - Hardware-Specific Synchronization Instructions (TODO) . . . 33

Processes **36**
- Process Management . . . 37
  - _exit, waitpid, getpid . . . 37
  - fork . . . 37
  - execv . . . 38
  - Examples . . . 38
- System calls . . . 39
  - Kernel Privilege . . . 40
  - Interrupts and Exceptions . . . 41
  - How System Calls Work . . . 42
  - User and Kernel Stacks . . . 43
  - Exception Handling in OS/161 . . . 43
- Multiprocessing . . . 44

Two Process Timesharing Example . . . 44
System Call Example . . . 45
Inter-Process Communication (IPC) . . . 50

Virtual Memory **51**
Physical and Virtual Addresses . . . 52
Sizes . . . 53
Physical Memory . . . 53
Virtual Memory . . . 53
Address Translation . . . 54
Methods for Address Translation . . . 54
Dynamic Relocation . . . 55
Segmentation . . . 56
Paging . . . 58
Two-Level Paging . . . 61
Multi-Level Paging . . . 62
Transition Lookaside Buffer (TLB) . . . 63
Software and Hardware Managed TLBs . . . 63
MIPS R3000 TLB . . . 64
OS/161's dumbvm . . . 65
ELF Files . . . 67
Virtual Memory for the Kernel . . . 68
Secondary Storage . . . 70
Page Faults . . . 70
Optimal Page Replacement . . . 71
FIFO Page Replacement . . . 72
LRU Page Replacement . . . 72
Clock Replacement Algorithm . . . 73
Global vs Local Page Replacement . . . 73

Scheduling **75**
Basic Schedulers . . . 75
Gantt Chart . . . 76
First Come First Served (FCFS) . . . 76
Round Robin (RR) . . . 77
Shortest Job First (SJF) . . . 77
Shortest Remaining Time First (SRTF) . . . 77
Comparing Perforances . . . 78
Processor Scheduling . . . 78
Multi-Level Feedback Queue (MLFQ) . . . 79
Completely Fair Scheduler (CFS) . . . 80
Multi-Core Scheduling . . . 81

Devices and I/O **82**
Device Register . . . 82
Example: Serial Console . . . 83
Example: SYS/161 timer/clock . . . 83
Device Drivers . . . 84
Accessing Devices . . . 85
Large Data Transfer To/From Devices . . . 86

Persistent Storage Devices 86
Hard Disks 86
Solid State Drives (SSD) and Flash Memory 87

Files and File Systems **89**

Introduction to Operating Systems

"*This operating system gives my life meaning*"
Linux vs Windows

As prerequisites you were required to take the following two courses:

- CS 241: how to convert code to assembly/machine language
- CS 251: how the CPU can run machine language

Notice however that your computer can do much more, this course will discribe how you are able to run multiple program and interact with hardware beyond your CPU.

A program does not directly interact with hardware, rather it interacts with the OS.

Example: say you want to read the text file `aaaa.txt`

- The file is located on a Toshiba Model `MQ01ABD100` 1TB Hard Disk Drive with a NTFS file system
- Which is connected via a SATA interface to an Intel 7 Series Chipset Family SATA AHCI Controller

In, C you can just call `fopen` and the specifics are abstracted away.

Definition: an *operating system* (OS) is a system that:

- Manages resources (e.g. processor, memory, non-volatile storage, I/O)
- Creates executation environments (i.e. interfaces to resources)
- Loads programs
- Provides common services and utilties

Three Views of an Operating System

We will look over three views of an operating system

- *Application view*: what services does it provide?
- *System view*: what problems does it solve?
- *Implementation view*: how is it built?

Application View

From a programmer's point of view the application view is that the OS is:

- *part cop*: provides protection from program errors
- *part facilitator*: provides an abstracted interface to the underlying system

More specifically we say that an OS provides an execution environment for running programs:

- Allocates *resources* that a program needs to run
  - e.g. processor time, memory, storage, access to I/O devices like the keyboard and monitor

- Provides *interfaces* for a program to use networks, storage, I/O devices, and other hardware
  - These interfaces are a much simplified abstract view of these hardware
- Performs *isolation* of running programs from each other
  - Prevents the crash/errors of one program from affecting the other programs
    * Without isolation one program crashing can bring down the entire system
    * Prevents a program from making invalid calls to another (e.g. wrong number of arguemnts)
  - Nowadays it is really hard to write a program that causes the entire system to crash

System View

The problems that the OS solves are:

- *management* of resources
- *allocation* of resources among running programs
- *controls* access to or the sharing of resources among programs

Resources include processor time, memory, storage, network, I/O such as keyboard(s), mouse, monitor.

The OS itself requires resources and must share with applictions running under it.

Implementation View

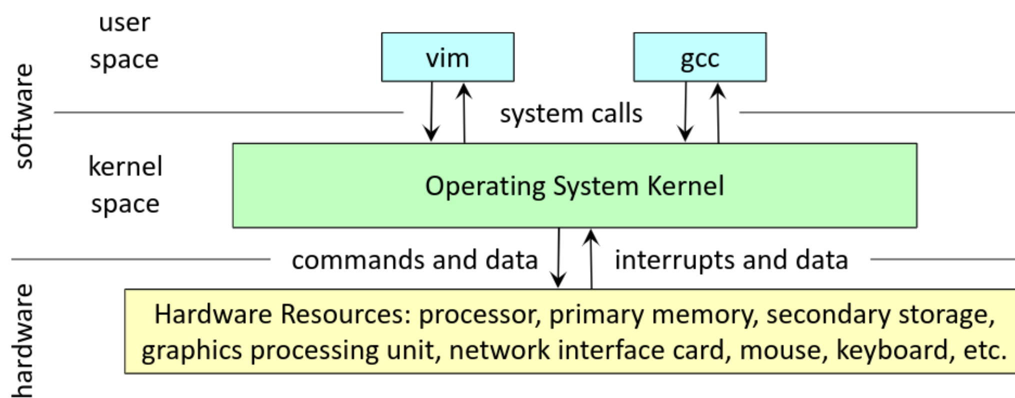
The OS itself is a concurrent, real-time program

- *concurrent*: many programs or sequences of instructions running, or appearing to run, at the same
  - arises naturally since an OS should support concurrent applications
  - also the OS must interact directly with many hardware
- *real-time*: program must respond to events in a specific amount time
  - hardware interactions impose timing constraints

Operating System and the Kernel

- *Kernel*: the core of the OS that responds to system calls, interrupts, and exceptions
- *Operating system*: includes the kernel and other related programs
  - utility programs (e.g. disk defragmenter, task manager)
  - command interpreters (e.g. cmd.exe on Windows, bash on Linux)
  - programming libraries (e.g. POSIX threads in Linux)

Schematic View of an Operating System



- **User Space**: all programs that run outside the kernel
  - These programs do not interact with the hardware directly
- **Kernel Space**: the region of memory where the kernel runs
- **User Programs**: programs that the user interacts with directly
  - e.g. desktop environment, web browser, games, editors, compilers, etc.
- **System Calls**: how a user process interacts with the OS
  - e.g. the C function `fopen` is implemented by making a system call to the OS

Key Point: there is a significant cost to pass arguments or copy data between user space and kernel space using a system call, than just executing a function call within user space.

Types of Kernels

- **Monolithic Kernel**: the entire OS (e.g. drivers, file system, etc) is in the kernel (e.g. Linux)
- **Microkernel**: only absolutely necessary components are part of the kernel
- **Hybrid kernel**: somewhere between monolithic and microkernel (e.g. Windows and macOS)
- **Real-time OS**: an OS with hard event response times, guarantees, and preemptive scheduling
  - Used for self-driving cars, air-traffic control, manufacuring, nuclear power plant, etc
  - Windows and macOS are real-time systems but are *not* a real-time OS

Differences between types of kernels

- *Monolithic kernels are faster* (than a microkernel)
  - Monolithic kernel: program calls the kernel which calls the driver which is also in kernel-space
    - 2 passes between user-kernel space barrier: 1 call and 1 return
  - Microkernel: program calls the kernel which calls the driver in user-space
    - 4 passes between user-kernel space barrier: 2 calls and 2 returns

- *Microkernels are more stable* (than a monolithic kernel)
  - Monolithic kernel: a fatal bug in a driver would case the entire kernel to crash
  - Microkernel: a fatal bug in a driver would only cause the caller and driver to crash

Operating System Abstractions

The *execution environment* provided by the OS included many *abstract entities* such as:

- files and file systems → secondary storage (HDD or SSD)
- Address spaces → primary memory (RAM)
- processes, threads → program execution (processor cores)
- sockets, pipes → networking and interprocess communications

This course will attempt to cover the design, implementation, and usage of these abstractions.

Multiprogramming

Definition: *multiprogramming* in the context of an operating system is the ability to multitask between multiple programs while maximizing resource utilization and maintaining a decent response time.

- **Concurrency**: multiple programs or sequences of instructions make progress at the same time
  - i.e. *run or appear to run* at the same time
- **Parallelism**: multiple programs of sequences of instructions can run at the same time
  - multiple processor cores or multiple processors
- **Timesharing**: multiples of sequences or instructions are given a fixed time slot to run in
  - e.g. round robin of all the jobs users submitted switching rapidly so all make progress

Threads

Definition: a *thread* is a *sequence of instructions*

- *Sequential programs* (CS241) have a single thread of execution (one location in code being executed)
- Consider programs with mutliple threads (multiple locations in code being executed concurrently)
  - Different thread can be responsible for different roles
    - e.g. displaying HTML, playing a video, audio, etc
  - Multiple threads can be responsible for the same roles
    - e.g. a webserver may have one thread for each person that is currently connected
- Analogous to a DFA (single current state) versus a NFA (set of current states)

A thread can **block**, ceasing execution until some condition is met to allow for another thread to run. (e.g. decode video while waiting on website response or for the user to type something)

Some reasons why we use threads:

- More efficient use of resources: while one threads is waiting another thread can be run
- Parallelism: threads can be distributed to different cores/processors to increase throughput
- Responsiveness and prioritization: some threads can run at higher priority
- Modular code: e.g. separate out the user inferface code from web page loading code

Intel and Hyper-threading (called Simultaneous Multi-Threading for AMD)

- This allows for two threads to be run on a single core by interweaving the instructions
- e.g. thread 1 is loading a word then thread 2 can do something else
- Improves the throughput of a core by around 30%
- This is why CPUs are marketed as something like “4 core 8 thread”

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
);
```

  - create a copy of the stack then place the function specified with arguments on the copied stack then resume execution on both stacks
  - notice that threads are not suppose to return anything (we are calling a void function)

- *Terminate* the calling thread: `void thread_exit(void);`
  - if you `exit();` a thread, the os will call `thread_exit();`
- *Volutarily yield* execution: `void thread_yield(void);`

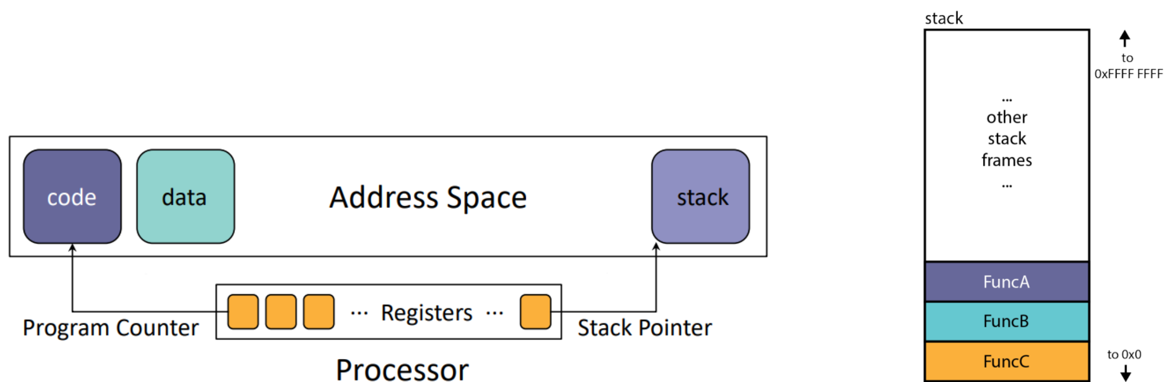
See `kern/include/thread.h` for more details.

Important: the threads will run in effectively random order if the programmer does not apply tools like synchronization primitives (i.e. thread sceduling is nondeterministic)

however there are tools (like locks) to force the threads into some order

Low Level View of Threads

When performing sequential program execution all we need is a single set of registers and a single stack:



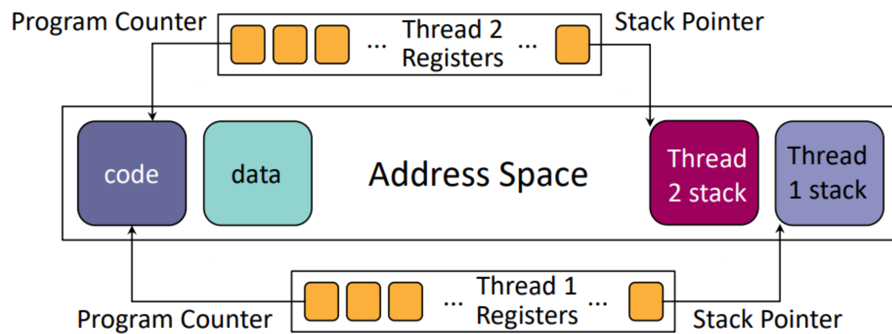
During the fetch/execute cycle we:

- *fetch* the instruction from the code that the PC points to
- then decode and *execute* it before incrementing the PC

Recall also that functions push their arguements, return address, local variables, and temp-use registers onto the stack (before they make a function call in order to save them)

Also recall how the stack grows downwards to lower numbers.

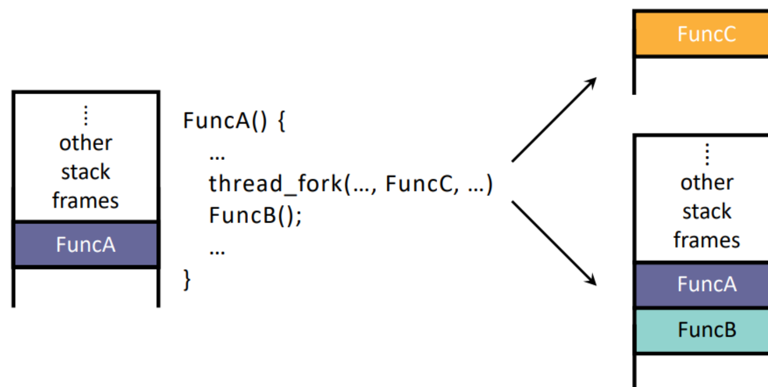
In OS/161 we conceptually think about having multiple independent sets of registers and stacks:



Notice that the two threads don't actually need to execute at the same time:

- Each thread *shares* access to the same global data, heap, code
- Each thread has its *own* stack and registers (PC is a register)
  - Each stack has a fixed size (typically around 2 MB)

Whenever `thread_fork` is executed, a new stack is created:



OS/161 also follows a more advanced register convention (and different register names)

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)			

- *Passing Arguments*: pass first 4 args in registers a0-a3 then rest (if more) on the stack
- *Return Value(s)*: use registers v0 and v1 for the return value
- *Caller-save*: registers *are not* preserved across subroutine calls
  - Caller should save the value onto the stack if it still needs the value after the subroutine

- *Callee-save*: registers *are* preserved across subroutine calls
  - Callee must restore register to original value if they change it

Concurrent Threads

To implement concurrent threads we have:

- *Hardware*: P processors, C cores, M multithreading per core $\implies$ PCM threads can execute simultaneously
  - e.g. Intel i9-9900X has 10 cores which can each run 2 threads (via hyperthreading) so

$$P = 1,\ C = 10,\ M = 2 \implies PCM = 20$$

- *Timesharing*: mutliple threads take turns on the same hardware; rapidly switching between threads so they all make progress

We use both hardware and timesharing: PCM threads running simultaneously with timesharing.

Definitions:

- *Multicore Processor*: a single die (or computer chip) containing more than one processor unit
  - Each core will have their own components but usually share the largest cache (L3)
- *Multithreading*: having multiple threads run on the same core
  - While one thread executes an slow instruction (e.g. load word with possible cache miss) we can execute instructions from the other thread
  - Require specialized hardware, such as two sets of registers, but the two threads can share most of the other hardware resources (e.g. ALU, Data, Memory)
- *Context Switch*: during timesharing this is switching from running one thread to another
  - The scheduler (not the programmer) decides which thread will run next
  - Saves register contents of the current thread then loads the register contents of next thread
- *Thread Context* (register values): must be saved/restored carefully, since thread execution continuously changes the context

The goal of timesharing is to give the user the *illusion* of multiple programs running at the same time.

Context Switch in MIPS

```
/* 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)

    /* 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
```

The C function `thread_switch` calls this assembly language subroutine `switchframe_switch`

- Context switch is similar to calling another subroutine except we also need to change the stack
- `thread_switch` is the *caller*
  - it will save and restore the caller-save registers, including the return address `ra`
- `switchframe_switch` is the *callee*
  - it must save and restore callee-save registers (frame pointer is callee saved in OS/161)
  - callee-save registers are saved to the old thread's stack

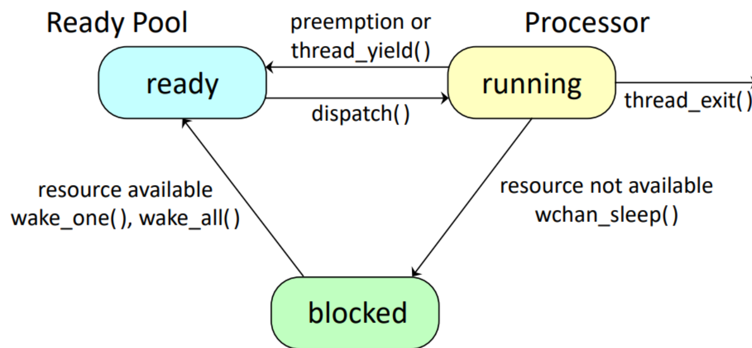
– restores callee-save registers from the new thread's stack

Since we are dealing with registers, context switching cannot be done in *high-level* C and must be written as a *low-level* subroutine in assembly langauge

Notice also that MIPS R3000 is pipelined so we need to manually insert *delay-slots* to protect against

- `load-user hazards`: value is loaded then used in next instruction
  – In this case we avoid this by using a `nop` (no operation) (line 39)
- `control hazard`: doesn't know which instruction should be fetched next
  – This MIPS will always fetch the next instruction (delay slot) so we can place an instruction after to always be executed (line 43)

Thread States



A thread can be in one of three states:

- *Running*: currently executing on a processor core
- *Ready*: ready to execute in the ready pool
- *Blocked*: waiting one something so not ready to make progress

we have four possible causes of a context switch:

- *Yields* (`thread_yield`) is the thread voluntarily allowing other threads to run
- *Exits* (`thread_exit`) is the thread terminating (i.e. it has completed its task)
- *Blocks* (`wchan_sleep`) is the thread waiting for some resource or event
- *Preempted* is when the thread exhausts its scheduling quantum so scheduler stops it

We then *dispatch* the next thread that can be run using `dispatch`.

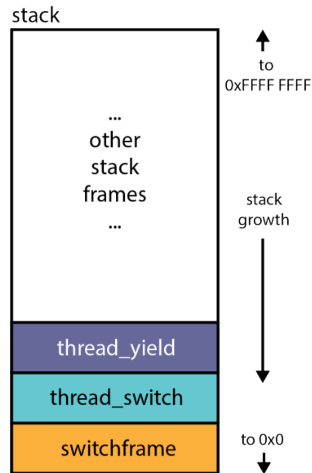
The concurrency in *timesharing* is achieved by rapidly switching between threads

- *Scheduling quantum*: the limit the thread can run before it will be preempted (e.g. 20 ms)
- *Preemption*: forces a running thread to stop running so that another thread can run

To implement preemption the thread library uses *interrupts* to "get control".

Yielding

Thread *yielding* is performing a voluntary context switch.



Suppose a thread calls `thread_yield` to yield the processor the call stack becomes

1. `thread_yield` calls another function to perform a high-level context switch
2. `thread_switch` chooses a new thread to run, then calls the assembly language subroutine
3. `switchframe_switch` performs the low-level context switch and creates the switchframe
    - See `/kern/arch/mips/thread/switch.S`

When performing a voluntary context switch all of `ra`, `fp`, `gp` and `s0-s6` are saved to the switchframe.

Control is given back by loading the values in the switchframe then continuing after `thread_switch` called `switchframe_switch` (much more subtleties, see: `/kern/thread/thread.c`)

However for involuntary context switches we need to save all the registers to a *trap frame*
(this includes `lo`, `hi`, `epc`, `cause`, `status`)

Interrupts

Thread *preemption* is when the execution of a thread is interrupted for something else to run.

Each thread actually has two stacks

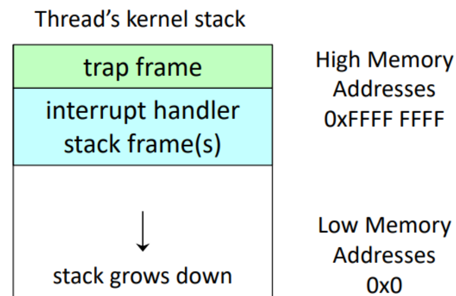
- *User space* stack which is the one you use in CS241
- *Kernel space* stack is used when the kernel calls kernel functions for the thread (e.g. `thread_yield`)

The kernel cannot trust any data structure in user space (e.g. stack could be full from infinite recursion)

An **interrupt** is specific events that forcefully give control to the interrupt handler

- Interrupts are caused by system devices (hardware), e.g. a timer, disk controller, network card, etc

- When any interrupt occurs the hardware transfers to a fixed location in memory
  - The location in memory is specified by the processor designer
  - It should be populated by a procedure to handle exceptions
    - see `/kern/arch/mips/locore/exception-mips1.S` for `common_exception`



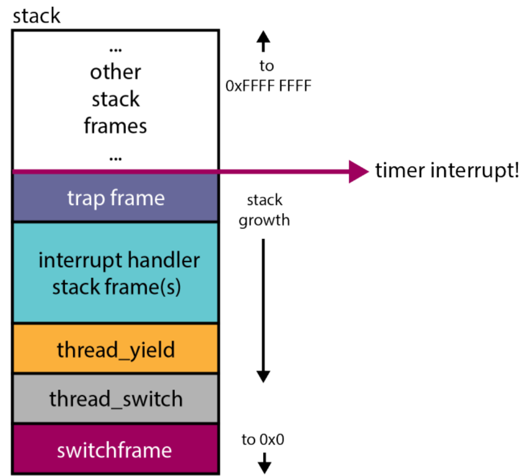
An **interrupt catcher** (`common_exception`) uses the *kernel stack* to

- Create a *trap frame* to record the thread context at the time of interrupt
  - the trap frame records every register including `lo`, `hi`, `epc`, `cause`, `status`
  - `trapframe` struct in `/kern/arch/mips/include/trapframe.h`
- Determine which device caused the interrupt and perform device-specific *interrupt handlers*
- Restore the saved thread context from the trap frame the resume execution of the thread

Scheduling

The preemptive scheduler manages the running threads

- Uses a *scheduling quantum* to impose a time limit on running threads
  - threads may block or yield before their quantum has expired
- Periodic *timer interrupts* are used to check how much time has elapsed
  - control is given to timer interrupt handler to check how much time has passed
  - if thread is has run too long then thread is preempted using `thread_yield`



everything after timer interrupt is done on the kernel stack

- Preempted threads changes state from running to ready and placed in *ready queue*
- Each time thread goes from ready to running runtime starts at 0

The scheduler is also responsible for deciding which thread shoulid 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*
  - after a context switch the running thread is moved to the end of the ready queue and the first thread on the queue is allowed to run
  - newly created threads are placed at the end of the ready queue
- Nondeterminstic because thread can be migrated to other cores or interrupted by some device

Two Threads Example

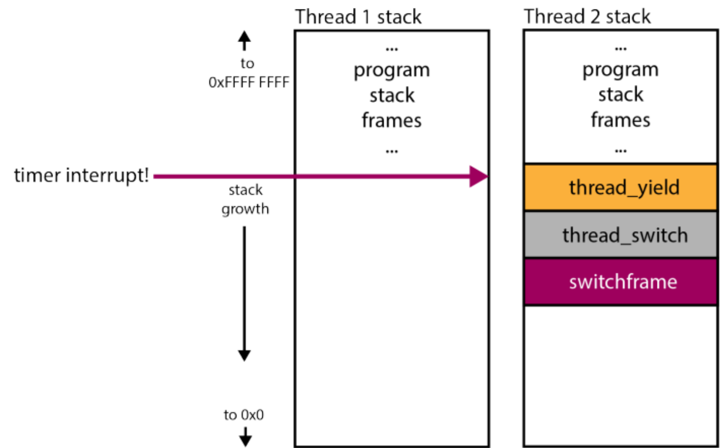
The following example considers two threads

- Thread 1 will get interrupted and Thread 2 can run (involuntary context switch)
- Then Thread 2 will preform a voluntary context switch

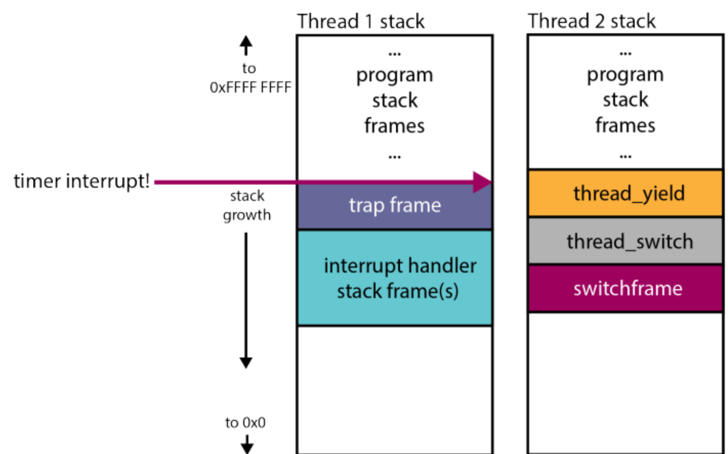
In summary when an interrupt occurs (notice this stack is in kernel space rather than user space)

- A trap frame is created
- Then the interrupt handler is called
- For an involuntary context switch `thread_yield` is called
- Which calls `thread_switch` to picks a new thread and
- Calls `switchframe_switch` to create a switchframe

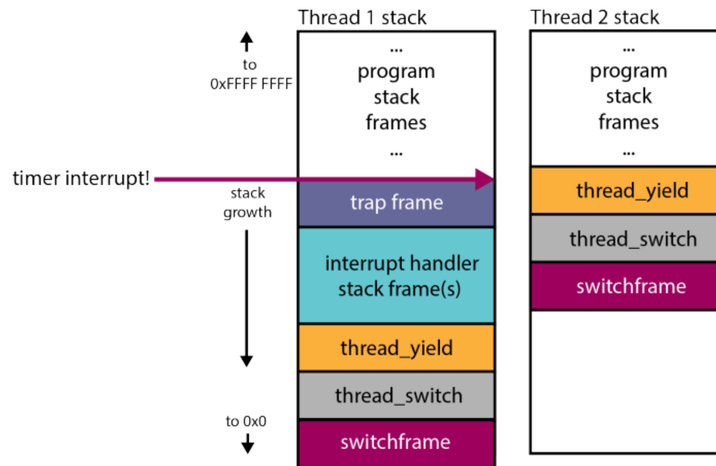
Timer interrupt occurs on the running Thread 1 and Thread 2 ready (called `thread_yield` before)



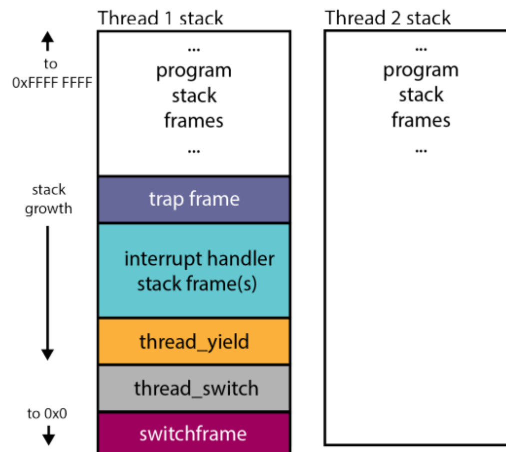
- Thread 1 is preempted then the exception handler will
  - create a trap frame to save Thread 1's context
  - call the timer interrupt handler



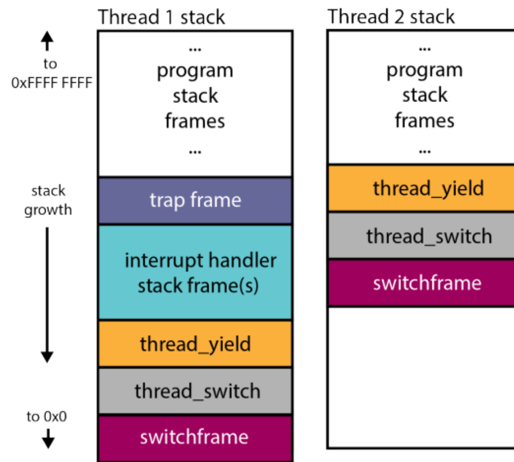
- In this case the timer interrupt handler determines that Thread 1 has exceeded its quantum
  - kernel calls `thread_yield` which calls
  - `thread_switch` (high-level context switch) choose a new thread then calls
  - `switchframe_switch` (low-level context switch) to store callee-save regsiter to switchframe
    * as computation has occured since trap frame was created (e.g. return address has changed)



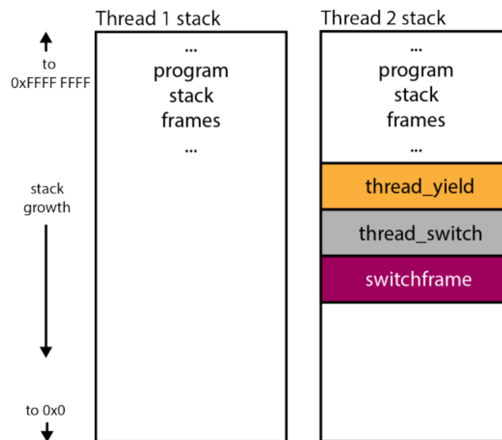
- Thread 1 is now ready and Thread 2 is running so begins by
  - completing low-level context switch `switchframe_switch` to restore callee-save registers
  - popping high-level context switch `thread_switch` stack frame
  - return from `thread_yield` and resume running with a fully restored context



- Now Thread 2 voluntarily yields by calling `thread_yield` which
  - for the high-level context switch calls `thread_switch` which chooses a new thread and
  - calls low-level context switch `switchframe_switch` to save callee-save registers in a switchframe



- Thread 1 is now running and Thread 2 is ready which
  - returns from low-level context switch `switchframe_switch`
  - returns from high-level context switch `thread_switch`
  - returns from `thread_yield`
  - return from timer interrupt handler
  - restore the Thread 1 context and pop off the trap frame and resume regular program



Synchronization

There are two main problems that we will need synchronization for:

- *Mutual Exclusion*: ensuring that only one thread can access the shared resource at a time
- *Producer and Consumer*: ensure that producer has made enough resources for the consumer to run

Concurrent threads interact with each other in different ways:

- All threads *share access* to the process's code, global variables, and heap
- Threads share access, through the os, to system devices such as hard drive, display, etc

The goal of synchronization is to maximize liveness and fairness while maintaining safety:

- *Safety* (correctness): ensuring that nothing bad will happen
- *Liveness* (efficiency): making sure that we are still making good use of resources
- *Fairness* (no starvation): making sure all threads are fairly granted a chance to execute

Mutual Exclusion

Definitions:

- **Mutual exclusion**: ensuring that only one thread at a time accesses a shared resource
- **Critical section**: the part of a concurrent program in which a shared object is accessed
- **Synchronization**: the coordination of access to a shared resource
- **Race condition**: when the programs result depends on the order of execution of the threads

Important: `volatile` keyword tells the compiler to not optimize the code

- Forces compiler to load/store the value on every use rather than keeping it in a register
- Threads don't share same regsiters but do share the same memory space

Example: mutual exclusion is best explained in the traffic analogy

- On highways as long as each car says in their lane there is no danger of collision
- At an intersection even if the cars just go straight there can be collision
- We need to maintain safety and ensure liveness and fairness otherwise:
  - no safety: cars can collide
  - no liveness: just allow a single car in the intersection at a time
  - no fairness: north and south can always go straight but no one else can go

Critical Section Example

Consider the following procedures along with their corresponding (pseudo) assembly language code.

```
/* code is available add_sub.c */
/* compile using: gcc -pthread add sub.c */

/* note the use of volatile */
int volatile total = 0;

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

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

Suppose that one thread executes `add` and another executes `sub` for a large N.

We have a couple of possible ways that the computer will execute this:

1. One possible order of execution is sequential (correct final value of `total=0`)

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

2. Another possible order is interleaved (incorrect final value of `total=1`)

```
;; Thread 1                      Thread 2
loadaddr R8 total
lw R9 0(R8)     ; R9= 0
add R9 1        ; R9= 1
```

```
5                   ----- Preemption ----->
6                                   loadaddr R10 total
7                                   lw R11 0(R10)   ; R11=0
8                                   sub R11 1       ; R11=-1
9                                   sw R11 0(R10)   ; total=-1
10                  <----- Preemption -----
11  sw R9 0(R8)     ; total= 1
```

3. Another possible order of execution on two cores (incorrect final value of `total=-1`)

```
1  ;; Thread 1                     Thread 2
2  loadaddr R8 total
3  lw R9 0(R8)     ; R9=0          loadaddr R10 total
4  add R9 1        ; R9=1          lw R11 0(R10)   ; R11=0
5  sw R9 0(R8)     ; total=1       sub R11 1       ; R11=-1
6                                  sw R11 0(R10)   ; total=-1
```

In this case example the critical section is when the thread reads, updates, then writes to `total`.

- Thread 1 loads `total` and increments then get preempted
- Thread 2 loads `total` and increments then writes back `total-1`
- Control is given back to Thread 1 which writes `total+1`

After one thread reads the value the other thread should not be able to read until the first thread finishes.

To enforce mutual exclusion we must use a lock:

```
1  int volatile total = 0;
2  bool volatile total lock = false;   // false means unlocked
3                                      // true means locked
4
5  void add() {                            void sub() {
6      int i;                                  int i;
7      for (i=0; i<N; i++) {                   for (i=0; i<N; i++) {
8          Acquire(&total lock);                   Acquire(&total lock);
9          ------------------ start mutual exclusion ------------------
10         total++;                                total--;
11         ------------------ stop mutual exclusion  ------------------
12         Release(&total lock);                   Release(&total lock);
13     }                                       }
14 }                                       }
```

We use `Acquire`/`Release` to ensure that only one thread can hold the lock

- Even if both threads try to aquire the lock simultaneously only one can get it
- The thread without the lock must wait until the lock is available

Spinlocks

```
1  Acquire(bool *lock) {
2      while (*lock == true) {};   // spin until lock is free
```

```
3     *lock = true;               // grab the lock
4 }
5
6 Release(bool *lock) {
7     *lock = false;              // give up the lock
8 }
```

Implemented directly this approach fails because it is possible to get preempted between line 2 and 3

- Thread is stopped just before it can acquire the lock
- Another thread can come along and set `*lock = true`
- Then when control is given back the first thread will acquire the same lock

As a result leading to two threads being in the critical section.

In order to fix this issue we will use the *test-and-set* approach that requires specific assembly instructions

- It needs to ensure between when `*lock` is *lested* and *set*, no other thread can change its value
- This will be discussed in depth later

Spinlocks should only be used for short waiting times (a few instructions) since:

- Other thread will *spin* (repeatedly test) until they can acquire the lock
- This is a form of *busy-waiting* where the processor is basically wasted on waiting on a spinlock

Important: *interrupts are disabled* on the CPU that holds the spinlock

- This is to ensure that other instructions waiting for the spinlock only wait for a short time
- This is why only the kernel can use spinlocks as a bug can cause an uninterruptable deadlock
- However other threads can still be preempted (reducing the amount of spinning they do)

Spinlocks are used to implement other synchronization primitives (lock, semaphore, conditional variable)

Spinlocks are already defined for you in OS/161

```
1 struct spinlock {
2     volatile spinlock_data_t lk_lock;
3     struct cpu *lk_holder;
4 };
5
6 void spinlock_init(struct spinlock *lk}
7 void spinlock_acquire(struct spinlock *lk);
8 void spinlock_release(struct spinlock *lk);
```

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

- `/kern/include/spinlock.h`
- `/kern/arch/mips/include/spinlock.h`

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);
```

The major difference is the *spinlocks spin*, while *locks block*, i.e. when a thread calls

- `spinlock_acquire` spins until the lock can be acquired
- `lock_acquire` blocks until the lock can be acquired

Locks are used to protect larger critical sections without being a burden on the processor.

Another difference is that while they both have *owners* and can only be released by their owner:

- A spinlock is owned by a CPU
- A lock is owned by a thread

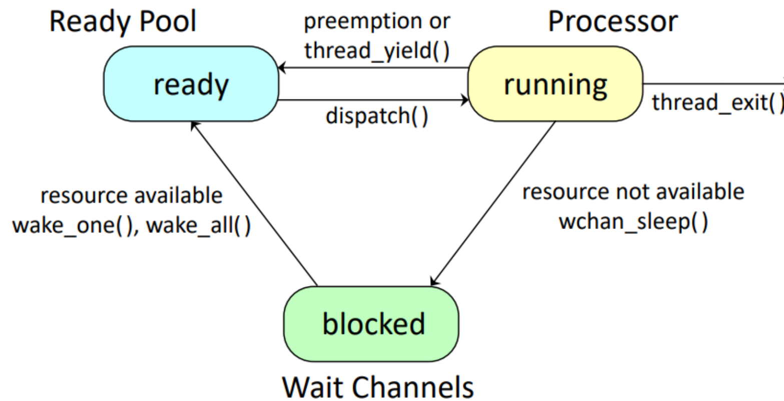
This is because interrupts need to be disabled on the CPU hardware.

Spinlock and lock API in OS/161 are quite similar:

- Spinlocks
  - `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)`

`do_i_hold` is important because we deadlock if a thread tries to acquire a spinlock/lock they already have.

Wait Channels



Recall the thread states

- *Running*: currently executing
- *Ready*: ready to execute
- *Blocked*: waiting for something, not ready to execute

When a thread *blocked* it will stop running and will not even be in the queue of threads to run.

Blocked threads need to be signaled by another thread to awaken so every lock has a *wait channel*.

```
struct lock {
        char *lk_name;
        struct wchan *lk_wchan;
        struct spinlock lk_lock;
        volatile struct thread *lk_owner;
};
```

- Blocked threads are queued on this wait channel
- After the lock is released, the wait channel is signaled and one thread is awakened to the ready pool

The wait channel API that will be used to implement thread blocking in OS/161

- `void wchan_lock(struct wchan *wc)`
  - prevents operations on wait channel `wc` by other threads
  - can't allow any wakeup to go through until we are finished going to sleep
- `void wchan_sleep(struct wchan *wc)`
  - blocks the current calling thread on wait channel `wc`
  - causes a context switch, like `thread_yield`
  - unlocks the wait channel `wc`
- `void wchan_wakeall(struct wchan *wc)`
  - unblocks all threads sleeping on wait channel `wc`

- `void wchan_wakeone(struct wchan *wc)`
  - unblocks one thread sleeping on wait channel `wc`

In OS/161 wait channels are implemented using queues.

Semaphores

A *semaphore* is another synchronization primitive that can act as a mutex.

They are an object with an integer value and has two operations

- `P` (*procure*): *wait* until semaphore value is greater than 0 then decrement it
- `V` (*vacate*): increment the value of the semaphore

By definition, these operations are atomic (note it actually `P`*roberen* and `V`*erhogen* from Dutch origin)

We have three types of semaphores:

- *Binary semaphore*: semaphore with a single resource; behaves like a lock, but *without owner*
- *Counting semaphore*: a semaphore with an arbitrary number of resourcres
- *Barrier semaphore*: used to force one thread to wait for others to complete, inital count of 0

Note: `V` does not have to follow `P` (can start with 0 resources and call to `V` to add resources)

Semaphores are already implementated in OS/161:

```
/* kern/include/synch.h */

struct semaphore {
    char *sem_name;             // for debug purposes
    struct wchan *sem_wchan;    // queue where threads wait
    struct spinlock sem_lock;   // to synchronize updates to sem_count and sem_wchan
    volatile int sem_count;     // value of the semaphore
};                              // notice that a semaphore does not have an owner
```

```
/* kern/thread/synch.c */

P(struct semaphore * sem) {
    spinlock_acquire(&sem->sem_lock);       // acquire spinlock before accessing
    semaphore
    while (sem->sem_count == 0) {           // check that sem_count == 0
        wchan_lock(sem->sem_wchan);         // prepare to sleep
        spinlock_release(&sem->sem_lock);

        wchan_sleep(sem->sem_wchan);        // sleep and unlock the channel

        spinlock_acquire(&sem->sem_lock);   // acquire spinlock
    }                                       // if sem_count > 0
    sem->sem_count--;                       // set
    spinlock_release(&sem->sem_lock);
}

V(struct semaphore * sem) {
```

```
    spinlock_acquire(&sem->sem_lock);
    sem->sem_count++;
    wchan_wakeone(sem->sem_wchan);
    spinlock_release(&sem->sem_lock);
}
```

In `P` during the time between waking up and aquiring the spin lock

- Possible to get preempted then another thread can change the value of `sem_count`
- We always need to double check under the safety of the critical section

A semaphore can be used as a mutex for mutual exclusion of a critical section:

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

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

The first thread to call `P(sem)` will block the other from accessing the critical section until `V(sem)`.

Producer and Consumer

In some situations we can have two types of threads:

- *Producers*: threads that (create and) *add* items to a buffer
- *Consumers*: threads that *remove* (and process) items from the buffer

This requires synchronization between consumers and producers which can be done using semaphores:

- Constraint 1: need to ensure that consumers only consume if the buffer has something in it
- Constraint 2: if buffer has finite capacity (N) then producers must wait when the buffer is full

e.g. packets of video data arrive from Wi-Fi (producer) into a buffer, then software displays it (consumer)

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

Producer’s Pseudo-code:
    P(Spaces);              // block if there is no space in buffer
    lock_acquire(Access);
```

```
    add item to the buffer
    lock_release(Access);
    V(Items);               // increase the number of items available

Consumer's Pseudo-code:
    P(Items);               // block if there are no items to consume
    lock_acquire(Access);
    remove item from the buffer
    lock_release(Access);
    V(Spaces);              // increase the amount of buffer space available
```

- Consumers will block i.e. wait for items to be produced
- Producers will block i.e. wait for spaces to available
- Only one thread can access the buffer data structure at a time

For multithreaded code we always consider what could go wrong if code was preempted at each step

```
int aFunc(int i) {
    f1(i);
            // What if it is preempted here?
    f2(i);
            // What if it is preempted here?
    f3(i);
            // What if it is preempted here?
    return 0;
}
```

Condition Variables

OS/161 support another synchronization primitive: *condition variables*

- Each condition variable is intended to work together with a lock
- They are used *within* the critical section protected by the lock

We have three operations we can do with a condition variable:

- *Wait*: causes calling thread to block
  - releases the lock associated with the condition variable
  - once thread is unblocked it reacquires the lock
- *Signal*: if threads are blocked on the signaled condition variable, then *one* of the threads is unblocked
- *Broadcast*: like signal, but unblocks *all* threads that are blocked on the condition variable

Conditional variable allow threads to wait on arbitrary conditions while in a critical section.

- If condition is not true, a thread can `wait` on corresponding condition variable until it becomes true
- If condition is true, a thread uses `signal` or `broadcast` to notify any blocked threads

- Usually, each condition variable corresponds to a condition of interest to an application

Note that signal/broadcasting to a condition variable with no threads waiting has *no effect*

Example: we want to wait while `numberOfGeese > 0`.

However, the thread cannot hold `geeseMutex` as other threads need access to `numberOfGeese`.

- Without condition variable:
  - we can release and yield to give another thread the chance to get `geeseMutex`
  - issue: thread is keeps trying to get the lock while it should just block until condition is true

```
1  int volatile numberOfGeese = 100;
2  lock geeseMutex;
3
4  int SafeToWalk() {
5      lock_acquire(geeseMutex);
6      while (numberOfGeese > 0) {
7          lock_release(geeseMutex); // allow access
8          thread_yield();
9          lock_acquire(geeseMutex); // restrict access
10     }
11     GoToClass();
12     lock_release(geeseMutex);
13 }
```

- With condition variable:
  - `cv_wait` will handle releasing and reacquiring `geeseMutex`
  - Calling thread will be placed on the cv's wait channel to block
  - `cv_signal` and `cv_broadcast` will wake threads waiting on the cv

```
1  int volatile numberOfGeese = 100;
2  lock geeseMutex;
3  cv zeroGeese;
4
5  int SafeToWalk() {
6      lock_acquire(geeseMutex);
7      while (numberOfGeese > 0) {
8          cv_wait(zeroGeese, geeseMutex);
9      }
10     GoToClass();
11     lock_release(geeseMutex);
12 }
```

Notice that we still need to check that `numberOfGeese == 0` even after `cv_wait`

- The other thread calls `cv_signal`/`cv_broadcast` on `zeroGeeze` then releases `geeseMutex`
- Now it becomes just a matter for which thread get acquire `geeseMutex` first

By the time `cv_wait` gets `geeseMutex` the `numberOfGeese` could have been updated.

Remark: we almost always signal the cv before releasing the lock

- This gives time to allow threads waiting on the cv to move to waiting on the lock
- This gives a better chance for one of them to get the lock

Example: producer consumer with 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);
}

itemType Consume() {
    lock_acquire(mutex);
    while (count == 0) {
        cv_wait(notempty, mutex);   // wait until buffer is not emtpy
    }
    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);
}
```

Other Synchronization Stuff

Deadlocks

Consider the following pseudocode:

```
lock lock1, lock2;

int FuncA() {                   int FuncB() {
    lock acquire(lock1)             lock acquire(lock2)
    lock acquire(lock2)             lock acquire(lock1)
    ...                             ...
```

It is possible that

- Thread 1 executes `lock_acquire(lock1)` and holds `lock1`

- Thread 2 executes `lock_acquire(lock2)` and holds `lock2`
- Thread 2 executes `lock_acquire(lock1)` and blocks
- Thread 1 executes `lock_acquire(lock2)` and blocks

The threads are now *deadlocked* (i.e. neither can make progress and are permanently blocked)

Two techniques for deadlock prevention:

- *No hold and wait*: thread should not request resources if it currently has resources allocated to it

```
lock_acquire(lock1);        // try get both locks
while(!try_acquire(lock2)) {
    lock_release(lock1);    // didn't get lock2 so try again
    lock_acquire(lock1);
}                           // can now obtain both resources
```

  `try_acquire` will return false if calling thread did not acquire the lock (not in OS/161)

- *Resource ordering*: order (i.e. number) the resource types and require threads to acquire resources in increasing order (if thread holds i it may not request resources of order less than or equal to i)

```
lock lock1, lock2;

int FuncA() {                   int FuncB() {
    lock acquire(lock1)             lock acquire(lock1)
    lock acquire(lock2)             lock acquire(lock2)
    ...                             ...
```

Other Sources of Race Conditions

Race conditions can occur of reasons beyond the programmer's direct control, specifically by

- the compiler
- then CPU

which may introduce race conditions due to optimizations to try to make the code execute faster.

Registers are must faster to access than RAM

- Compilers optimize for this by storing values in register for as long as possible
- Consider if one thread runs `FuncA` and another runs `FuncB`

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

  recall: threads each get their own set of registers but share the memory

- `volatile` disables this optimization
  - it forces values to be loaded and stored to memory with each use
  - it also prevents the compiler from re-ordering the loads the stores for that variable
  - All shared variables should be declared `volatile`

Many languages have multi-threading with memory models and language-level synchronization functions

- Their compilers are aware of critical sections via these language-level synchronization functions and optimize such to not cause race conditions
- Version of C used by OS/161 does not support this

CPU also has a memory model and may re-order loads and stores to improve performance

- Modern architectures provide barrier/fence instructions to disable these CPU-level optimizations as they cna create race conditions
- Our MIPS R3000 CPU does not have such optimizations so doesn't need these instructions

Hardware-Specific Synchronization Instructions (TODO)

For spinlocks we need to ensure no other thread can change `*lock` between when it is tested and set

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

We will look at two ways to solve this problem

1. x64's (a.k.a. AMD64, x86-64) `xchg scr, addr`
   - this *test-and-set* instruction is *atomic*
   - this swaps the values stored in regsiter `src` with value stored at address `addr`

   ```
   // logically behaves like this, executed atomically

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

   - spinlock Acquire and Release with `xchg`

   ```
   Acquire(bool *lock) {
       while (xchg(lock, true) == true) {}; // test and set
   }

   Release(bool *lock) {
       *lock = false; // give up lock
   ```

```
7 }
```

- If `xchg` returns `true` then lock is already taken and you have not changed the value
  - keep spinning
- If `xchg` returns `false` then lock was free and you have changed the value
  - stop spinning

2. MIPS uses two instructions `ll addr` and `sc addr new_val`
   - `ll` (load linked): loads value at address `addr`
   - `sc` (store conditional): store `new_val` at `addr` if it has not been changed since `ll` was executed
     - returns FAIL if value has changed since `ll`
     - returns SUCCESS if value hasn't changed since `ll`

TODO

```
1  MIPSTestAndSet(addr, new_val) {
2      old_val = ll addr           // test if someone holding lock
3      status = sc addr, new_val   // try to set the lock
4      if ( status == SUCCESS ) return old_val
5      else return true            // lock is being held
6  }
7
8  Acquire(bool *lock) { // spin until hold lock
9      while( MIPSTestAndSet(lock, true) == true ) {};
10 }
```

if the lock value at `ll` and before `sc`

- *stays false* then return false (is is acquired)
- *stays true* return true (owned by someone else)
- *change* return true (someone else is doing something?)

Spinlock in OS/161

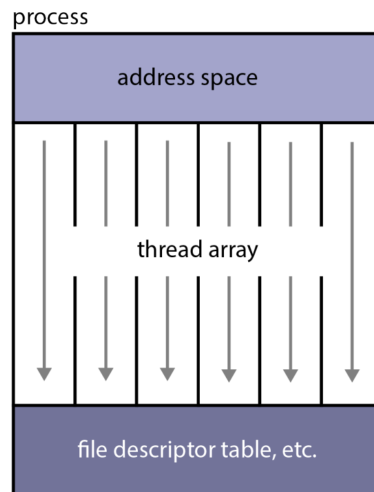
```
1  /* return value 0 indicates lock was acquired */
2  spinlock_data_testandset(volatile spinlock_data_t *sd)
3  {
4      spinlock_data_t x,y;
5      y = 1; // value to store (i.e. locked)
6      __asm volatile( // begin assembly language
7          ".set push;" // save assembler mode
8          ".set mips32;" // allow MIPS32 instructions
9          ".set volatile;" // avoid unwanted optimization
10         "ll %0, 0(%2);" // get value of lock x = *sd
11         "sc %1, 0(%2);" // *sd = y; y = success?
12         ".set pop" // restore assembler mode
13         : "=r" (x), "+r" (y) : "r" (sd));
14         // above: read only, read + write, input
```

```
15      if (y == 0) {return 1;} // if unsuccessful, still locked
16      return x; // if success return lock value
17  }
```

Processes

Definition: a *process* is an environment in which a program runs

- The *process's environment* includes *virtualized resources* that the program can use:
  - address space for the code and data
  - threads for executing code on processors
  - I/O such as keyboard, display, file system, etc.



- Processes are created and managed by the kernel
- Each program's process *isolates* it from other programs running in other processes

Definition: *virtualized resources* are abstract versions of physical resources

- An address space, called *virtual memory*, is an abstraction involving RAM and secondary storage
  - the goal is to make it appear like we have a large amount of primary memory (RAM)
  - from the process's perspective it has exclusive access to this memory
- Every physical resource (e.g. processor, RAM, I/O) are in shared by all processes
  - virtualization isolates the processes from each other
  - each process feels that they have exclusive access
  - a bug in one program does not effect stability or outcome of another program

Process Management

Processes can only be created, managed, or destroyed by the kernel

- you will be implementing OS/161's process management in A2a and A2b

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

_exit, waitpid, getpid

Definition: the *process identifier* (pid) is a unique identifier given to every process.

- `_exit` terminates the process that calls it
  - the process can give an status code for it to exit on
  - kernel records this exit status code in case another process asks for it (waitpid)
- `waitpid` blocks until the process being waited on terminates, then it returns its exit status code
- `getpid` returns the pid of the current process

fork

System call `fork` creates a new process *child* (C) that is a clone of the of the original process *parent* (P).

- Copies P's address space (code, globals, heap, stack) in C's address space
- Creates a new thread for C
- Copies P's trap frame onto C's (kernel) stack
  - code in P is reempeted until all copying is done

The address space of the parent and child are identical at the time of the fork.

`fork` was called by the parent and returns in *both* the parent and child.

- The parent and child will see different return values from `fork`
  - parent process P will get the C's pid
  - child process C will get 0
- This provides a way to distinguish the child from the parent

execv

System call `execv` changes the program that the process is running.

- The calling process's current virtual memory is destroyed
- Then the process gets a new virtual memory initialized with code and data of new program to run
- Extra arguments given to `execv` will be passed to the new program
- After `execv` is done, the new program starts executing
- `pid` and permissions remain the same as before

Example: running `htop` in `bash`, `bash` will fork itself and use execv to load and run the `htop` executable

Examples

Example: `_exit`, `waitpid`, `getpid`, `fork`

```
main() {
    int rc = fork();        // returns 0 to child, pid to parent

    if (rc == 0) {          // child executes this code
        my_pid = getpid();
        x = child_code();
        _exit(x);
    }
    else {                  // parent executes this code
        child_pid = rc;
        parent_pid = getpid();
        parent_code();
        p = waitpid(child_pid, &child_exit, 0);
        if (WIFEXITED(child_exit))
            printf("child exit status was %d", EXITSTATUS(child_exit));
    }
}
```

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

- `WIFEXITED` returns true if the child called `_exit()`
- `WEXITSTATUS` returns the exit return value

The exit status code `child_exit` must store both these values

Example: using `execv` to execute: `/testbin/argtest first second`

```
int main() {
    int status = 0;     // status of execv function call
    char *args[4];      // argument vector
                        // prepare the arguments
    args[0] = (char *) "/testbin/argtest";
    args[1] = (char *) "first";
    args[2] = (char *) "second";
    args[3] = 0;        // null terminate the end of args

    status = execv("/testbin/argtest", args);
```

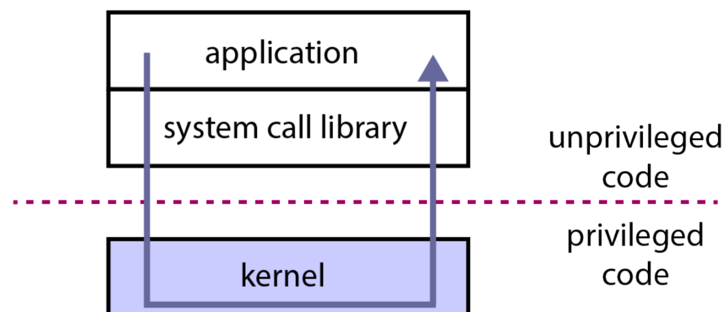
```
    printf("If you see this output then execv failed");
    printf("status = %d errno = %d", status, errno);
    exit(0);
}
```

Example: fork then have child execute `execv` while the parent waits

```
main() {
    char * args[4];     // args for argtest

    /* <--- set args here ----> */

    int rc = fork();    // returns 0 to child, pid to parent

    if (rc == 0) {      // child's code
        status = execv("/testbin/argtest", args);
        printf("If you see this output, then execv failed");
        printf("status = %d errno = %d", status, errno);
        exit(0);
    }
    else {              // parents's code
        child_pid = rc;
        parent_code();
        p = waitpid(child_pid, &child_exit, 0);
    }
}
```

System calls

Definition: *system calls* are the interface between user processes and the kernel



- The *applications* are programs like you wrote in CS246 or CS241
- The *system call library* is provided by the programming language for using the OS's services
  - e.g. C has `fopen` while Python has `open` to interface with the `open` system call
- Process management calls (e.g. `fork`) are system calls
  - they are called by user programs
  - to have to kernel perform some action

- System calls are used to perform many services

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

  - interprocess communication: copy and paste, anything over the internet
  - `sbrk` (sbreak) is the process asking the OS for more memory

Kernel Privilege

The processor implements different levels (rings) or *execution privilege* for security and isolation.

- Kernel code runs at the highest privilege level, where all instructions given to processor are executed
- Application code runs at a lower privilege level to prohibit it from performing certain tasks
  - modifying page tables that kernel is using to implement address spaces
  - access RAM that has not been allocated to the process
  - outright halting the processor and many more
- User space programs cannot execute code belonging to a higher privilege level than they have

So application programs cannot directly call kernel functions or access kernel data structures.

Remark: the Meltdown vulnerability found in Intel chips allowed user applications to bypass execution privilege and access any address in physical memory.

So how do we keep the kernel isolated from application processes yet allow them to use kernel services?

- The kernel is a program, just like any other program
- However we want the kernel to have privileges no other programs may get

The key idea is to have two different types of calls:

- *Procedure calls*: used by applications to execute other application code
- *System calls*: used by applications to execute kernel code

Interrupts and Exceptions

There are two times when we forcefully switch to executing kernel code:

- **Interrupts**: *generated by devices* when they need attention
  - *hardware interrupt*
  - e.g. timer interrupt, new network traffic received, SSD got data, keyboard press, etc.
- **Exceptions**: *caused by instruction execution* when a running program needs attention
  - *software interrupt*
  - e.g. division by zero, illegal instruction, page faults, system calls, etc.

Remark: there is ambiguity in these terms as some use interrupt is used to refer to both.

In both cases the following occurs:

- The processor stores the cause in the `cause` register
- Control is transferred to a fixed location where (depending on if interrupt or exception):
  - a *interrupt handler* must be located
  - a *exception handler* must be located
- The processor switches to privileged execution mode

Both the interrupt handler and exception handler are part of kernel.

OS/161 uses the same routine `mips_trap` to handle both exceptions and interrupts

- The handler matches the `cause` register with codes set by MIPS to determine what triggered it
- `EX_IRQ` is interrupt and the rest are exceptions
- `kern/arch/mips/include/trapframe.h`

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

How System Calls Work

On the MIPS processor `EX_SYS` is the system call exception

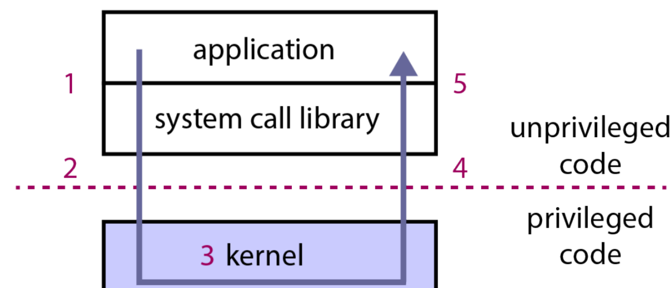
- To cause this exception on MIPS the application executes the assembly instruction: `syscall`
- To distinguish which system call was made the kernel defines a code for each system call

Definition: *system call codes* are codes the kernel defines for each system call it understands

- Kernel expects application to place the system call code in a location before executing `syscall`
  - for OS/161 the system call code goes in register `v0`
  - other arguments go in registers `a0`, `a1`, `a2`, `a3` (e.g. `execv` passes program path and args)
    * success or fail code is placed in `a3` upon return
    * return value or error code is placed in `v0` upon return
- If you pass something invalid the system will probably just terminate your program
- `kern/include/kern/syscall.h`

```
#define SYS_fork    0
#define SYS_vfork   1
#define SYS_execv   2
#define SYS__exit   3
#define SYS_waitpid 4
#define SYS_getpid  5
// over 100 more
// ...
```

These handler codes and code location are part of the kernel's *ABI* (*Application Binary Interface*).



1. When an application calls a library wrapper function for a system call
2. The library function executes a `syscall` instruction
3. This causes the kernel's exception handler to run
   - creates a trap frame to save application program state
   - matches `cause` with `EX_SYS` and determines which system call was made
   - does the work, restores application program state from trap frame, and returns
4. The library wrapper function finishes and returns, allowing the application to continue execution

User and Kernel Stacks

Every OS/161 process thread has 2 stacks, but only uses one at a time:

- **User (Application) Stack**: used when the thread is executing application code
  - located in application's user space
  - stores stack frames for application's functions (as seen in CS241)
  - created by the kernel when it set up the virtual address for the process
- **Kernel Stack**: used while the thread is executing kernel code (i.e. for exception or interrupt)
  - is a kernel structure (i.e. located in the kernel's address space)
  - stores stack frames for kernel's functions
  - also holds *trap frames* and *switch frames*
  - in OS/161 the `t_stack` field of the `thread` structure points to this stack

We don't trust the user stack because of bugs, stack overflow, etc.

Exception Handling in OS/161

When the library function executes `syscall` to handle the exception:

- First to run is `common_exception` that
  - saves the application stack pointer
  - switches the stack pointer to point to the thread's kernel stack
  - saves the application state into a trap frame on the thread's kernel stack
  - calls `mips_trap` passing a pointer to the trap frame as an argument
- After `mips_trap` finishes, `common_exception` will
  - retores the application's state from the trap frame and stack pointer
  - switch back to unprivileged execution mode
  - jump back to the application instruction that was interrupted
- `kern/arch/mips/locore/exception-mips1.S`

The C function `mips_trap` is what actually looks at the exception code:

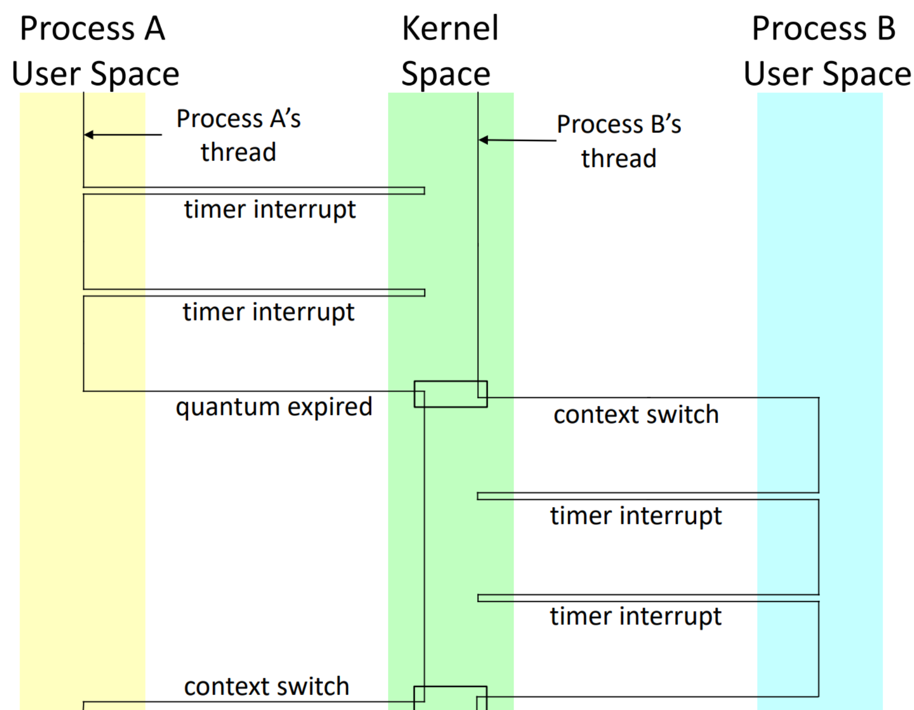
- interrupt → call `mainbus_interrupt`
- address translation exception → call `vm_fault`
- system call → call `syscall` (a kernel function), passing it the trap frame pointer
  - `kern/arch/mips/syscall/syscall.c`
- `kern/arch/mips/locore/trap.c`

Multiprocessing

Definition: *multiprocessing* (or *multitasking*) means having multiple processes existing at the same time

- All processes must share the available hardware resources as coordinated by the kernel
  - virtual memory is implemented using parts of the physical memory
  - threads are scheduled to execute on available processor cores
  - processes share access to other resources (e.g. disks, network, etc) by making system calls
- OS ensures that processes are isolated from each other
  - bugs in one process should not affect another
  - interprocess communication should be possible but only at explicit request of both processes
    - both processes have to set up the communication channel
- Processes may have many threads, but must have at least one thread to execute
- OS/161 supports multithreaded code only in kernel space (one thread or user space process)

Two Process Timesharing Example



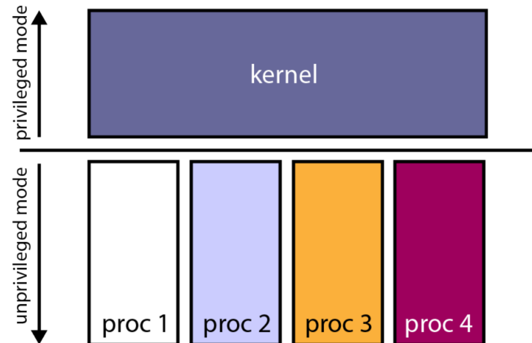
- Initially process `A` is running and process `B` is on the ready queue
- Periodic timer interrupts occur to give control to the kernel to check if `A`'s quantum has expired
- When the quantum has expired kernel switches to running process `B`

One way threads are given higher priority is to allowed to run for longer.

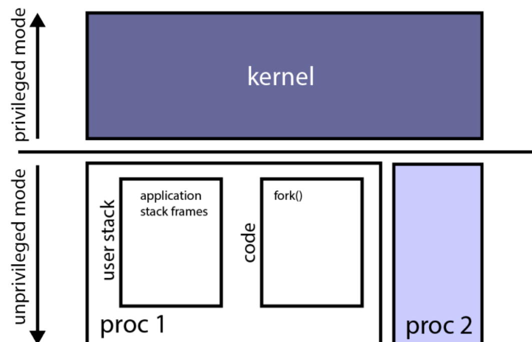
System Call Example

The following is what happens under a system call for `fork` followed by a timer interrupt

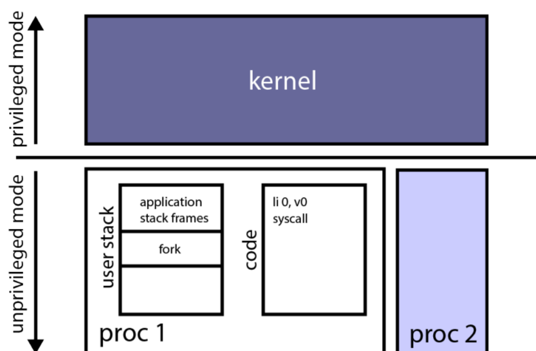
- User space contains four processes all running in unprivileged mode
- Kernel space runs in privileged mode



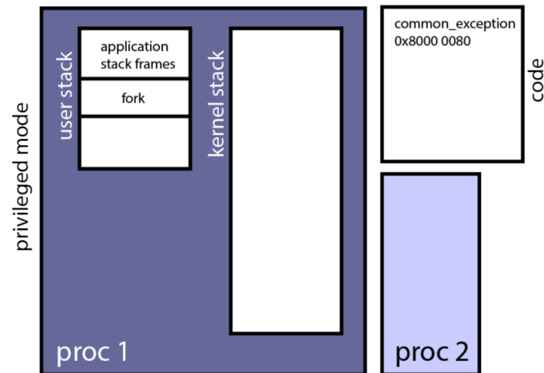
- First `proc1` calls `fork` (system call library function)



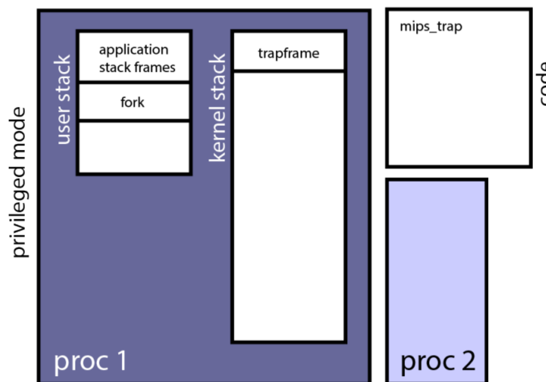
- The library routine `fork` then performs the following:
  - stores the system call code `SYS_fork` into register `v0` (`li 0, v0`)
  - then executes the `syscall` (MIPS assembly instruction) to raise an exception



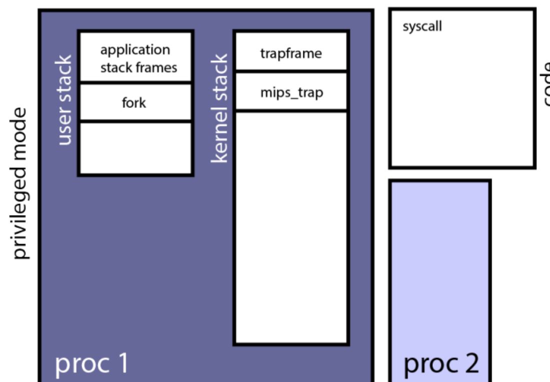
- When the exception is raised the processor will:
  - trap (i.e. go) into privileged mode and turn interrupts off
  - jump to `0x8000 0080` which is where the kernel put the `common_exception` routine



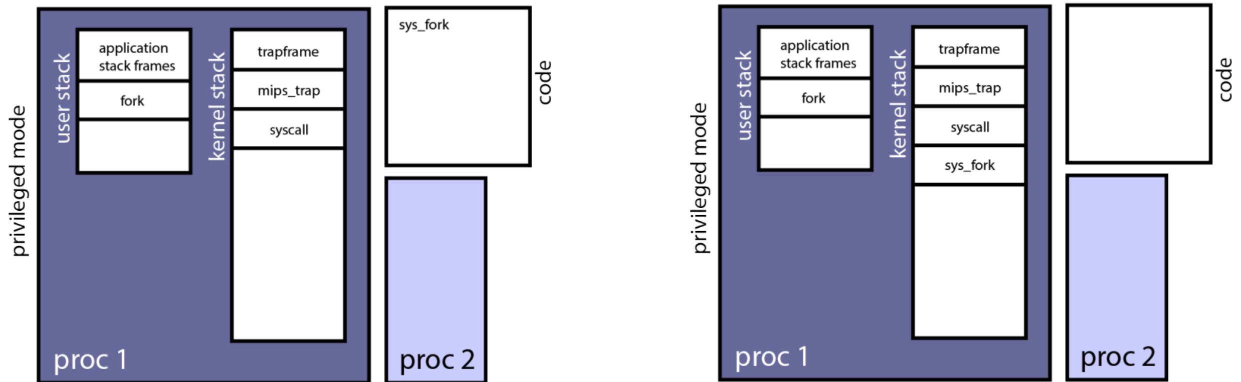
- `common_exception` then runs which:
  - since we came from user mode, switch from user stack to kernel stack
  - save the trapframe then call `mips_trap`



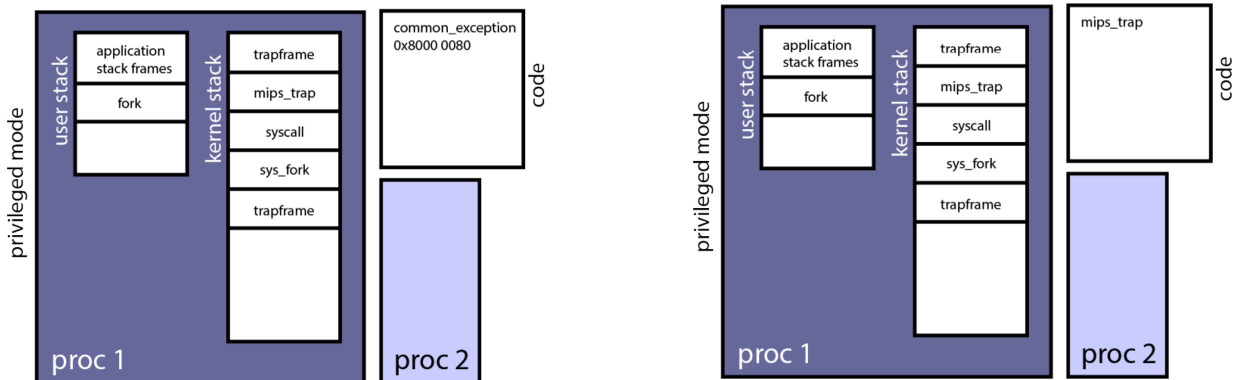
- `mips_trap` determines what kind of exception was raised (in this case `EX_SYS`)
  - for a system call, interrupts are turned back on and `syscall` (kernel function) is called



- `syscall` (system call dispatcher) calls the appropriate handler based on system call code in `v0`
  - in this case `sys_fork` is called which is executed by the kernel



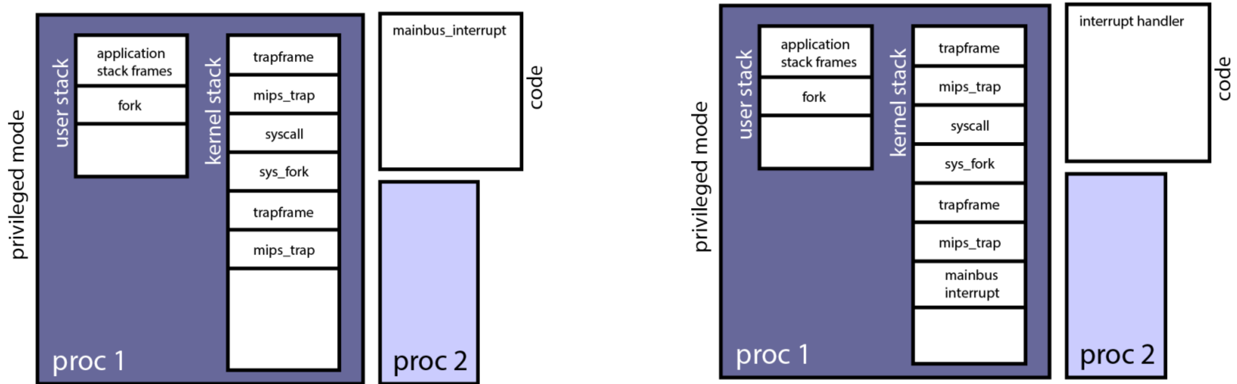
- Suppose during the execution of `sys_fork` a timer interrupt occurs
  - the processor disables interrupts and jumps to `common_exception` again
  - `common_exception` saves another trapframe and calls `mips_trap`



– since the exception was an interrupt `mainbus_interrupt` is called to handle it

– `mainbus_interrupt` determines which device threw the interrupt by the `cause` register

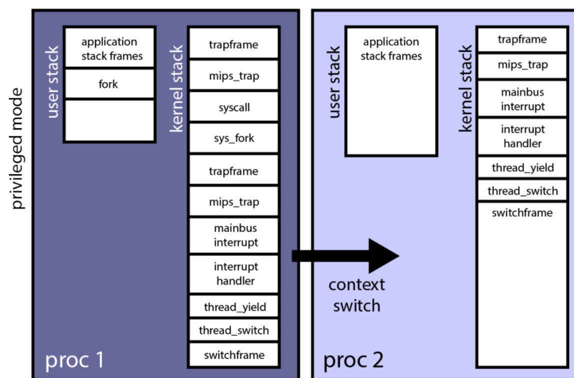
  * more information other than the exception type is passed as `cause`
  * in this case it is a timer interrupt so `mips_timer_set` is called



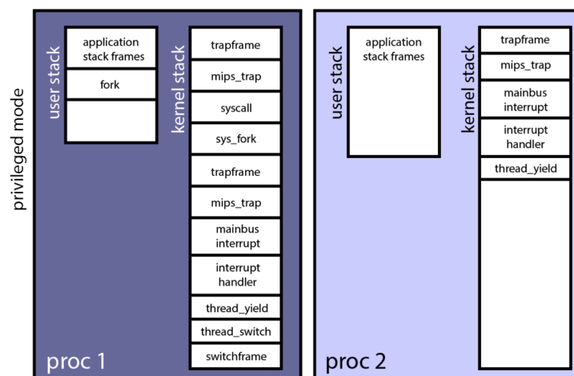
– since the thread's quantum has expired, `thread_yield` is called

– which calls `thread_switch` to pick a new thread and

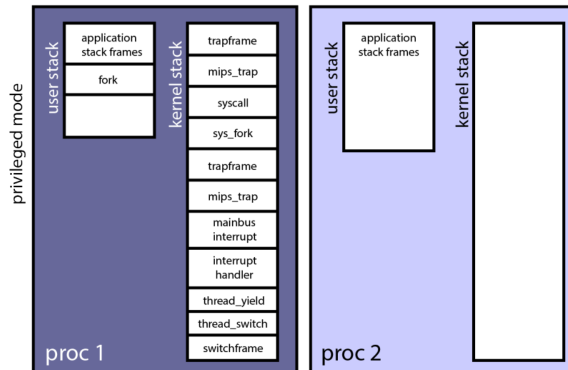
– call `switchframe_switch` to create a switchframe and switch threads



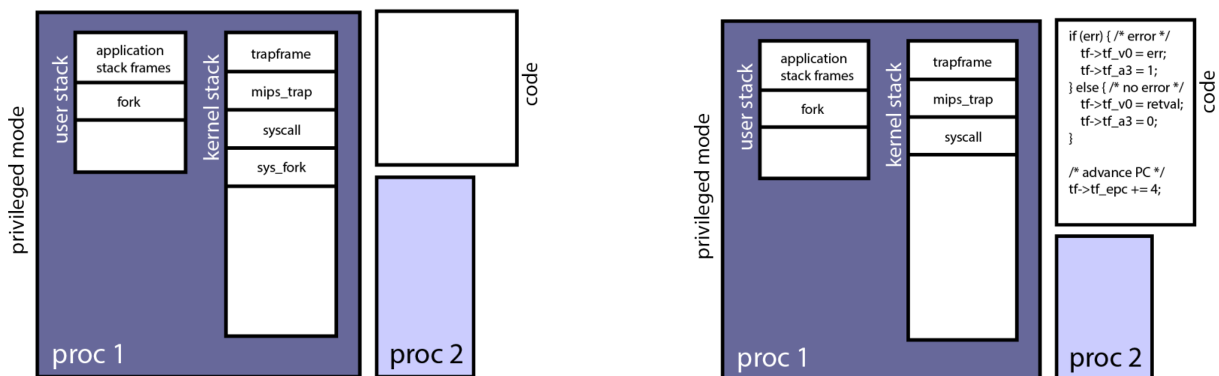
– in the new thread, `switchframe_switch` is completed to restore its execution state



- `thread_yield` returns to interrupt handler
- interrupt handler returns to `mainbus_interrupt`
- `mainbus_interrupt` returns to `mips_trap` which enables interrupts again
- `mips_trap` returns to `common_exception`
- thread context is restored from trapframe, switch back to user stack and unprivileged mode



- Now assume that timer interrupt did not occur and kernel finishes executing `sys_fork`
  - `sys_fork` returns to `syscall` which sets up the return value/error code `v0` and the result `a3`
  - it also increments the PC so we execute the next instruction when we return



- `syscall` returns to `mips_trap`
- `mips_trap` returns to `common_exception` and restores registers using trapframe
  - also switches from kernel to user stack and back to unprivileged mode (`rfe`)
- Finally the user code continues execution

References:

- `common_exception` is located in: `kern/arch/mips/locore/exception-mips1.S`
- `mips_trap` is located in: `kern/arch/mips/locore/trap.c`
  - `syscall` is located in: `kern/arch/mips/syscall/syscall.c`
  - `mainbus_interrupt` is located in: `kern/arch/sys161/dev/lamebus_machdep.c`
- `sys_fork` is probably in `kern/syscall/proc_syscalls.c` (Assignment A2a)

Inter-Process Communication (IPC)

Definition: *inter-process communication* (IPC) are methods used to send data between processes

- *File*: data to be shared is written to a file, accessed by both processes
- *Socket*: data is send via network interface between processes
- *Pipe*: data is send unidirectionally, from one process to another via OS-managed data buffer
- *Shared Memory*: data is send via block of shared memory visible to both processes
- *Message Passing/Queue*: queue/data stream provided by the OS to send data between processes

Virtual Memory

We have three requirements for memory:

1. Multiple processes can be loaded into RAM
2. Bugs in one program (e.g. bad pointer value) cannot affect another process
3. Implementation details can be largely hidden from the application programmer

We create a virtual environment for RAM, *virtual memory*, that looks the same for all processes.

The best demonstration of what we mean by this is to consider what happens when we `fork()`

```
#include <stdio.h>
#include <unistd.h>     // provides fork()
#include <sys/types.h>
#include <sys/wait.h>   // provides sleep()
#include <stdlib.h>

int main() {
    int status;
    pid_t pid;

    int j = 0;                      // j is stored in stack
    int* i;                         // i is stored in the heap
    i = (int *) malloc(sizeof(int));
    *i = 0;

    sleep(1);
    pid = fork();                   // Fork into parent and child

    if (pid == 0) {                 // Child Process
        sleep(2);
        printf("\nChild pid  %d\n", (int) getpid());
    }
    else {                          // Parent Process
        sleep(3);
        waitpid(pid,&status,0);
        printf("\nParent pid  %d\n", (int) getpid());
    }

                                    // Print out memory addresses
    printf("code      %14p\n", (void *) (& main) );
    printf("heap (i)  %14p\n", (void *) i);
    printf("stack (j) %14p\n", (void *) &j);


    ++(*i);                         // Increment and print values
    ++j;
    printf("i = %d\n", *i);
    printf("j = %d\n", j);
    free(i);
    return 0;
}
```

Running this code we get the output:

```
Child pid  84976
code       0x5627a485b1a9
heap (i)   0x5627a50262a0
stack (j) 0x7fffab6260e8
i = 1
j = 1

Parent pid  84975
code       0x5627a485b1a9
heap (i)   0x5627a50262a0
stack (j) 0x7fffab6260e8
i = 1
j = 1
```

Notice that the code, heap, and stack addresses are the same in both processes.

- The parent process sleeps one second longer than the child
- This should imply that that `i=2` and `j=2` because they have the same address
- However incrementing one process's variables does not affect another other's variables

Also notice that the memory addresses are much larger than how much RAM the computer can have.

This is only possible because the addresses we are seeing here are *virtual*.

Physical and Virtual Addresses

- **Physical addresses**: are provided directly by the hardware
  - one physical address space *per computer*
  - size of physical address space determines maximum amount of addressable physical memory
    * e.g. 32 bit addresses can only index 2^{32} bytes = 4GB of RAM
- **Virtual Addresses**: (or logical addresses) are addresses provided by the OS to the process
  - one virtual address space *per process*
  - the hardware (with help from OS) converts virtual addresses to physical ones for a process

The conversion of virtual addresses to a physical address is called *address translation*.

The goal of this differentiation is for the OS:

- part facilitator: gives each process the illusion of a large amount of exclusive contiguous memory
- part cop: isolate processes from each other and from the kernel

Sizes

Definition: B always means byte, b always means bit and 8b = 1B

- K, M, G are ambiguous
  - powers of 10 bits: $K = 1000,\ M = 1000^2,\ G = 1000^3$ for secondary storage or bandwidth
  - powers of 2 bits: $K = 1024,\ M = 1024^2,\ G = 1024^3$ for primary storage
- For this course K, M, G will always be powers of 2 bits

$$1K = 2^{10}$$
$$1M = 2^{30}$$
$$1G = 2^{30}$$

Examples:

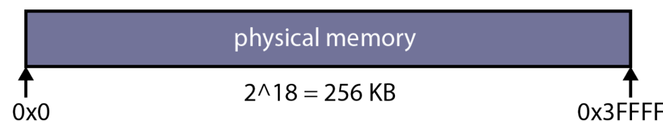
- A typical page size is $2^{12} = 2^2 \times 2^{10} = 4K$
- If we had $8M$ how many pages could we fit into it

$$8M/4K = (2^3 \times 2^{20})/(2^2 \times 2^{10}) = 2^{11} = 2K \approx 2000$$

Physical Memory

If we have P bits to specify physical addresses of each byte then we can have at max 2^P bytes.

- SYS/161 MIPS processes uses 32-bit physical addresses ($P = 32$)
  - so maximum physical memory size is 2^{32} bytes (4GB)
- Modern processes typically use $P = 48$ for maximum physical memory size of 256TB
- The examples in these notes will use $P = 18$ or 256KB

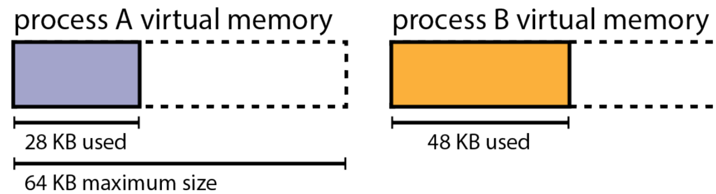


The *actual amount* of physical memory is typically less than *maximum amount* that can be addressed.

Virtual Memory

The OS provides a private virtual memory for each process.

- Virtual memory holds the code, data, heap, and stack for the program running in the process
- If using V bits to address a byte then we can have at max 2^V bytes
  - For MIPS $V = 32$ (also $P = 32$)
  - For our examples $V = 16$ (with $P = 18$ so physical is larger than virtual which is a little weird)



- Running applications only see virtual addresses
  - program counter and stack poitner hold virtual addresses of next instruction and top of stack
  - pointers to varaibles are virtual addresses
  - jumps refer to virtual addresses
- Each process is isolated by its virtual memory and cannot access another process's virtual memory

Address Translation

Definition: address translation is performed in hardware by the *Memory Management Unit* (MMU)

- The MMU works using information provided by the kernel
- When a process tries to access a virtual address is translated to a physical address
- Recall how even the program counter is a virtual address, and
  - each instruction requires at least one translation
  - software translation is much slower than just doing it with hardware

Remark: can store things at physical address 0 but not virtual address 0 (reserved as `nullptr`)

It is very wasteful to have an entry in the page table for every virtual address

- Instead of indexing by bytes the page tables translates by the *block* (typically $2^{12} = 4\text{KB}$)
- For a virtual address of 32 bits we will use the last 12 for the *offset* and the rest to index the block
  - this means that MMU will only translate the first 20 bits as so

$$\text{Virtual Page Number} \,||\, \text{Block Offset} \quad \rightarrow \quad \text{Physical Page Number} \,||\, \text{Block Offset}$$

  - by this method we can save a lot of space for the page table (2^{20} vs 2^{32} entries)

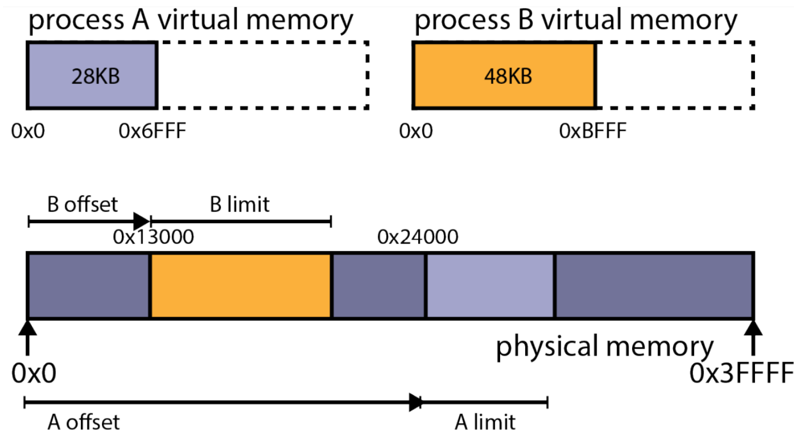
Methods for Address Translation

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

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

Dynamic Relocation

The most basic method is to just give each process some portion of physical memory.



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

- *Offset*: address in physical memory where the process's memory begins
- *Limit*: the amount of memory allocated to the process

The memory management unit (MMU) then has:

- *Relation register*: (`offset`) to hold the physical offset of the running process's virtual memory
- *Limit register*: (`limit`) to hold the size of the running process's virtual memory

Then to tranlate a *virtual* address v to a *physical* address p the MMU does the following computation:

```
1  if (v < limit) then
2      p ← v + offset
3  else
4      raise memory exception
```

The kernel maintains `offset` and `limit` values for each process and updates the values in the MMU registers when there is a context switch.

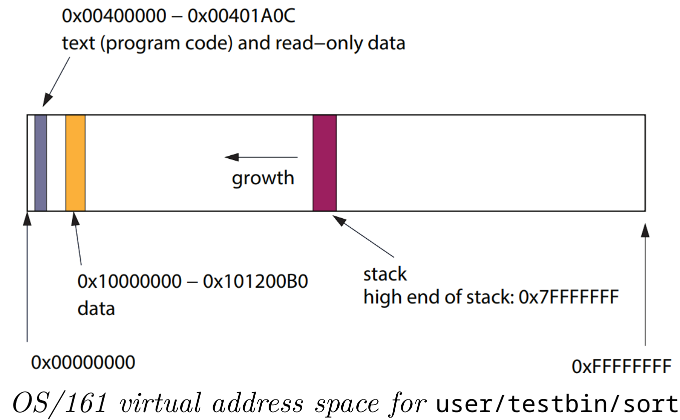
Properties of dynamic relocation:

- Each virtual address space corresponds to a *contiguous range* of physical addresses
- Kernel decides where each virtual address space should map in physical memory
  - OS must track which parts of physical memory are in use and where is free
  - Since different address spaces can have different sizes OS must allocate/deallocate variable-sized chunks of physical memory
  - This creates the potential for fragmentation of physical memory

Example: fragmentation example

- Say a system has 100MB of free, but not contiguous, physical memory
- If a program requires 100MB of memory we would not be able to allocate it
- Although we have 100MB free, we don't have 100MB of contiguous space

The most critical issue of dynamic relation is that it does not use physical memory efficiently:

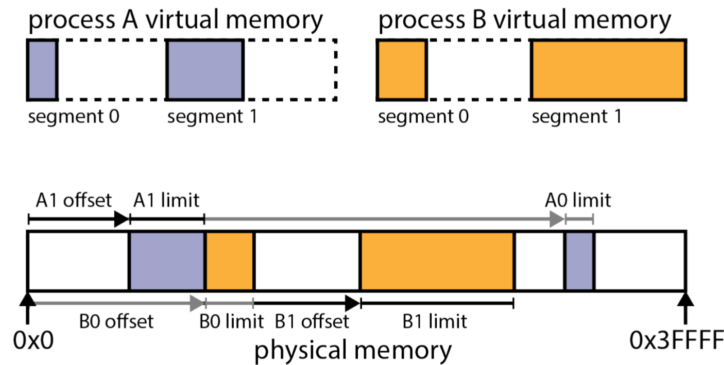


OS/161 virtual address space for `user/testbin/sort`

- While the address space uses 1.2MB its dynamic reloation would require 2GB
- Virtual memory may be large but the process's address space may be very small and discontiguous

Segmentation

Instead of mapping the entire virtual memory to physical memory, map each *segment* separately.



- Kernel maintains an offset and limit value for each segment (instead of each process)
- The virtual address has 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

$$\text{Segment ID} \mid\mid \text{Segment Offset} \quad \rightarrow \quad \text{Physical Offset} + \text{Segment Offset}$$

- The kernel decides where each segment is placed in physical memory
- Fragmentation is still possible as the segments can be of any size

To translate virtual adddress v to physical address p we have:

```
1  s ← SegmentNumber(v)
2  a ← OffsetWithinSegment(v)
3
4  if (a ≥ limit[s]) then
5      raise memory exception
6  else
7      p ← a + offset[s]
```

The kernel maintains a set of relation offsets and limits for each process and updates the values in the MMU when there is a context switch.

Example: say we have a process with two processes

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

- To translate a virtual address we need to split it into two parts
  - first bit specifies its segment
  - next three bits specifies first hexdigit of offset and rest specifies rest of the offset
- `0x1240` starts with `0x1 = 0001`
  - address is in segment 0 so offset is `0x38000`
  - `001 = 0x1` so offset is `0x1240` (which is within limit of `0x2000`)
  - adding these together gets us a physical address of

$$\texttt{0x01240} + \texttt{0x38000} = \texttt{0x39240}$$

- `0xA0A0` starts with `0xA = 1010`
  - address is in segment 1 so offset is `0x10000`
  - `010 = 0x2` so offset is `0x20A0` (which is within limit of `0x5000`)
  - adding these together gets us a physical address of

$$\texttt{0x020A0} + \texttt{0x10000} = \texttt{0x120A0}$$

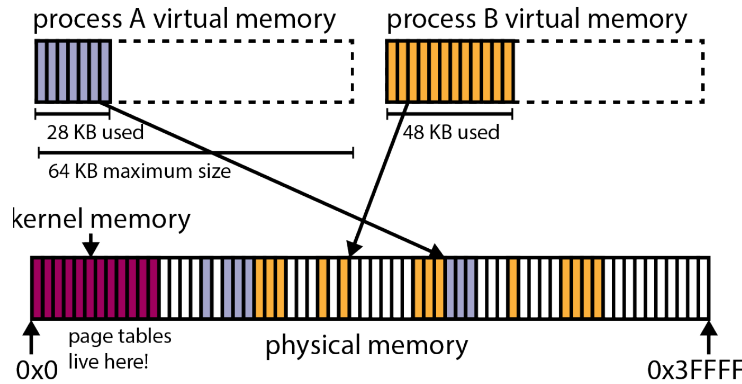
If the segment number or offset is too large then throw an exception (segmentation fault).

Paging

To rid of fragmentation we need our segments to be of fixed size.

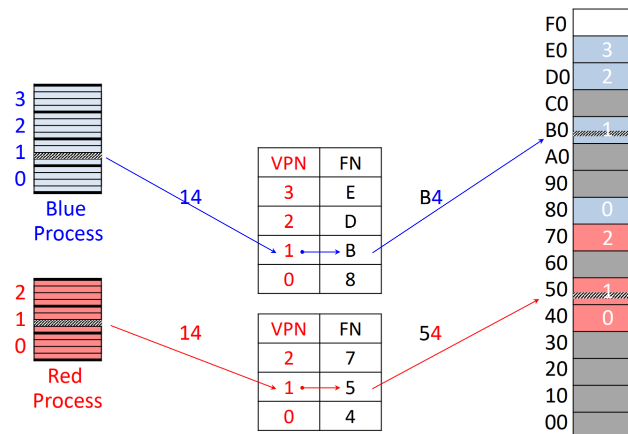
- *Frames*: we divide physical memory into fixed-sized chunks called frames (physical pages)
- *Pages*: we divide virtual memory into fixed chunks called pages

Each frame and page must be off the same size (typically 4KB)



- *Page Table*: mapping of pages (virtual memory) different frames (physical memory)
  - each process has its own page table
  - a row of a page table is called a *page table entry* (PTE)

Example: *virtual page number* (VPN) to *frame number* (FN) translation



- Assume that RAM has been divided into 16 frames (physical pages) of 16 addresses each
- Blue process: virtual address `0x14`
  - VPN of `1` is translated to FN of `B` by page table
  - using the offset of `4` we get physical address `0xB4`
- Red process: virtual address `0x14`
  - VPN of `1` is tranlated to FN of `5` by page table
  - using the offset of `4` we get physical address `0x54`

For fast random access we make the page table an array of PTE indexed by VPNs

- *Valid bit*: used to indicate if the PTE is actually used
  - if 1, then PTE maps that page number to a frame number
  - if 0, then PTE is not valid
- Number of PTEs = Max Virtual Memory Size / Page Size

Example: translate `0x102C` and `0x9800` for process A

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

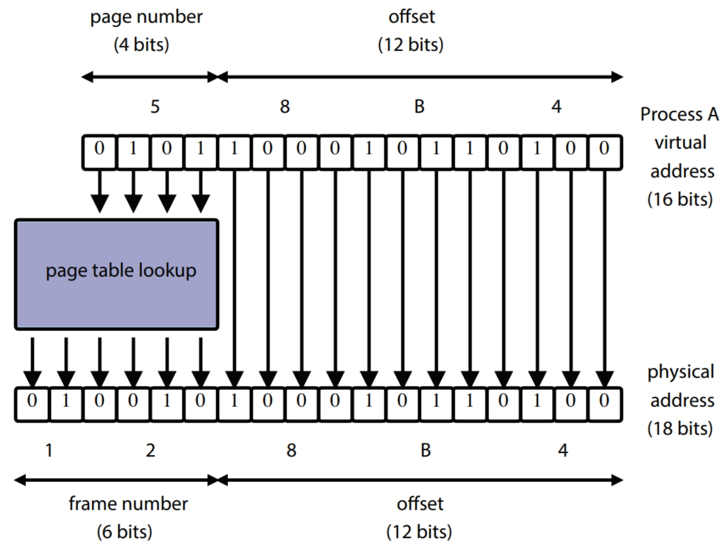
- `0x102C` is split into page number `0x1` and offset `0x02C`
  - in the page table `0x1` is valid and is translated to frame number `0x26`
  - concatenating with the offset we have `0x2602C`
- `0x9800` is split into page number `0x9` and offset `0x800`
  - in the page table `0x9` is invalid $\implies$ segmentation violation

The MMU includes a *page table base register* which points to the page table for the current process

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. Check if the PTE is not valid then raise an exception
4. Otherwise, combine corresponding frame number with offset to determinet he physical addresss

$$\text{physical address} = (\text{frame size} \times \text{frame size}) + \text{offset}$$

Notice that these operations can be done using bit manipulations since they are all powers of 2



Page table entries (PTE) can contain other fields

- Write protection bit: set by kernel to indicate that a page is read-only
  - If a write operation uses a virtual address on a read-only page the MMU will raise an exception
- Reference bit: set when page is accessed and cleared periodically
  - used to gauge if a page has been recently used
- Dirty bit: used to indicate contents of page has been changed
  - used later to check if we need to flush the data to drive

Page tables are kernel data structures so are stored in the kernel memory, but how big are they?

- $V = 32$ bits (common 20 years ago) leads to max virtual memory size of $2^{32} = 4$GB
  - assuming page sizes are 4KB and each PTE is just 32 bits (4 bytes)
  - the number of pages would be $2^{32}/2^{12} = 2^{20}$ so the page table (per process) would be

$$2^{20} \times 4 \text{ bytes} = 4\text{MB}$$

- $V = 48$ bits (current 64-bit architecture) leas to maximum memory size of $2^{48} = 2^8$ TB = 256TB
  - assuming page size is 4KB each PTE size is 32 bits (4 bytes)
  - the number of pages would be $2^{36}/2^{12} = 2^{36}$ so the page table (per process) would be

$$2^{36} \times 4 \text{ bytes} = 2^{38} = 256\text{GB}$$

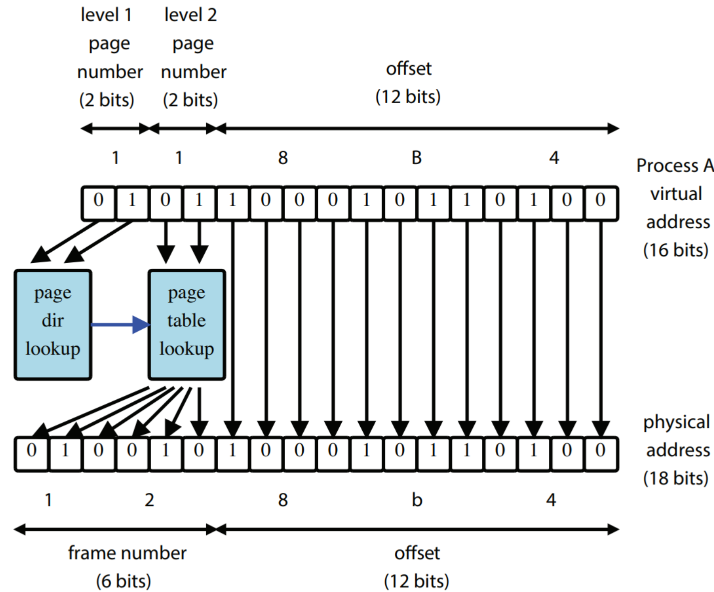
So with our current page table most people would not be able to start even one process.

Two-Level Paging

We can split up the page number and organize the page table into multiple levels.

- The higher level tables contain pointers to tables on the next level
- The lowest-level page table contains the frame number

If a table contains no valid PTEs do not create it and set its pointer in the higher level table to invalid.



Example: we will translate the virtual address `0x58B4` using the two-level paging

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	NULL	0
0x8	NULL	0
...	...	...
0xE	NULL	0
0xF	NULL	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

- There are 2 levels and each level uses 2 bits so virtual address has 3 parts:

$$p_1 \mid\mid p_2 \mid\mid \text{offset}$$

- The page number is `0x5 = 0101` which we split into $p_1 = 01$ and $p_2 = 01$
  - using $p_1 = 01 =$ `0x1` we go to Table 2
  - using $p_2 = 01 =$ `0x1` we have frame number `0x12`
- Combining this with offset of `0x8B4` we get the physical address `0x128B4`

Need to know the number of levels and number of bits for each level's translation.

Multi-Level Paging

We can extend our 2-level page table to an n-level page table so each virtual address has $n+1$ parts:

$$p_1 \mid\mid p_2 \mid\mid \cdots \mid\mid p_n \mid\mid \text{offset}$$

MMU's page table base register points to the page table directory for the current process

- Index into the page table directory (level 1) using p_1 to get a pointer to a 2nd level page table
  - if entry is not valid, raise an exception
- Index into the 2nd level page table using p_2 to find a pointer to a 3rd level page table
  - if entry is not valid, raise an exception
- $\cdots$
- Index into n-th level page table using p_n to find a PTE for the page being accessed
  - if PTE is not valid, raise an exception
- Otherwise, frame number from the PTE with *offset* to determine the physical address

The overall goal of multi-level paging is to reduce the size of individual page tables.

How do we choose the number of levels? Key idea: have each table fit on a single page

- Say if $V = 40$, page size is $2^{12} = 4\text{KB}$, and PTE size is $2^2 = 4$ bytes
- The number of PTEs needed for this virtual memory size is

$$(\text{virtual memory size})/(\text{page size}) = 2^{40}/2^{12} = 2^{28} \text{ pages}$$

- The number of PTEs that can be stored on each page is

$$(\text{page size})/(\text{PTE size}) = 2^{12}/2^2 = 2^{10} \text{ PTEs}$$

- So how many page tables rae needed to store all 2^{28} PTEs?

$$(2^{10})^n = 2^{28} \quad \rightarrow \quad \lceil 28/10 \rceil = 3 \text{ levels}$$

- With format $p_0 \mid p_1 \mid p_2 \mid$ offset we have sizes

$$2^8 \mid 2^{10} \mid 2^{10} \mid 2^{12}$$

- The top level page (called the *directory*) holds 2^8 references to page tables
  - requires $2^8 \times 2^2 = 2^{10} = 1\text{KB}$ of space

Transition Lookaside Buffer (TLB)

We have seen how the kernel (software) and MMU (hardware) interact to mangage virtual memory:

- Kernel:
  - manages MMU registers on address space switches
  - creates and manages page tables
  - allocates/dellocates physical memory
  - handles exceptions raised by the MMU
- Memory management unit (MMU):
  - translates virtual addresses to physical addresses
  - checks for and raises exceptions as necessary

Notice that address translation adds a minimum of one extra memory operation (page table lookup)

- Every assembly language instruction requires at least one memory operation (fetch instruction)
- This extra step really slows down everything so we add a cache

Definition: *Translation Lookaside Buffer* (TLB) is a cache for PTEs in the MMU

- TLB is a small, fast, dedicated cache of recent address translations in the MMU
- Each TLB entry stores a single (page → frame) mapping stored as pair (p, f)

Software and Hardware Managed TLBs

Hardware-managed TLB: MMU handles TLB misses

```
1 if (p has an entry (p,f) in the TLB) then
2     return f        // TLB hit!
3 else
4     find p’s frame number (f) from the page table
5     add (p,f) to the TLB, evicting another entry if full
6     return f        // TLB miss
```

- This is what is actually done by Intel/AMD CPUs due to its speed
- The MMU must be able to perform a page table lookup (i.e. understand kernel’s page table format)
- One evication method that works quite well and is very fast (important!) is random eviction
- Kernel must invalidate all entires int he TLB during a address space switch
  - unless the MMU can distinguish TLB entries are from different address spaces

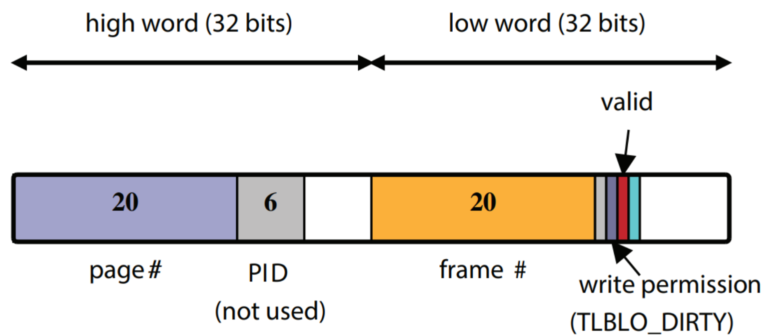
Software-mangaged TLB: kernel handles TLB misses

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

- In the case of a TLB miss exception, the kernel must
  1. determine the frame number for p (by page table lookup)
  2. add (p, f) to the TLB, evicting another entry if necessary
- After miss is handled, instruction that cause the exception is re-tried
- Kernel must invalidate all entires int he TLB during a address space switch

This is the method used by OS/161 and is much too slow to be used for actual machines.

MIPS R3000 TLB



The MIPS TLB has room for 64 entries

- Each entry is 64 bits (8 bytes) long
- If `TLBLO_DIRTY` (a.k.a. write permission) is set then you can write to this page
- Valid bit indicates that the mapping in the TLB entry is valid
- PID can be ued to distinguish mappings from different processes but we are not using it
  - instead we set all the valid bits to 0 when changing address spaces
- `kern/arch/mips/include/tlb.h`

OS/161's address space

- Virtual and physical addresses are 32 bits
- Page size is 4KB (requires 12 bits) so both frame number and page number are $32 - 12 = 20$ bits

Paging conclusions

- Benefits
  - paging avoids externel fragementation
  - multi-level paging reduces the amount of memmory reguired to store page-to-frame mappings
- Costs
  - TLB misses are increasingly expensive with deeper page tables

Normally have 4-level paging and 48-bit virtual addresses, servers will sometimes have 5-level paging with 57-bit virtual addresses.

OS/161's dumbvm

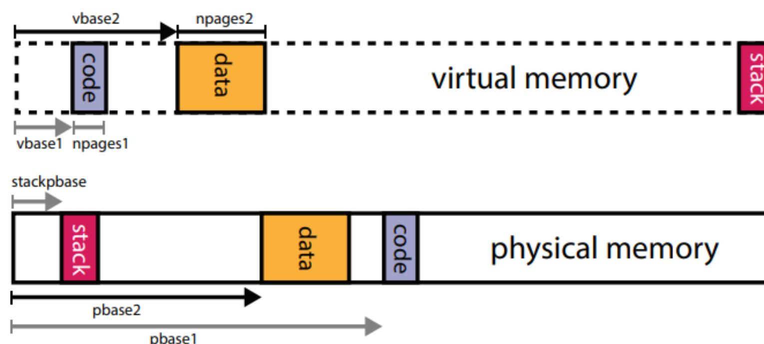
MIPS uses 32-bit paged virtual and phyiscal addresses and has a software-managed TLB.

- Software-managed TLB will raise an exception on every TLB miss
- Then the kernel mangages the page-to-frame mappings
- TLB exceptions are handled by a kernel function `vm_fault`

`vm_fault` uses information from a kernel `addrspace` structure to determine a page-to-frame mapping

- There is a separate `addrspace` strcture for each process
- Each `addrspace` structure describes where the process's pages are stored in physical memory
- The `addrspace` structure does the same job as a page table
  - is made simpler since OS/161 places all pages of each segment contiguously in physical memory

```
1  struct addrspace {
2      vaddr_t as_vbase1;      /* base virtual address of code segment */
3      paddr_t as_pbase1;      /* base physical address of code segment */
4      size_t as_npages1;      /* size (in pages) of code segment */
5      vaddr_t as_vbase2;      /* base virtual address of data segment */
6      paddr_t as_pbase2;      /* base physical address of data segment */
7      size_t as_npages2;      /* size (in pages) of data segment */
8      paddr_t as_stackpbase;  /* base physical address of stack */
9  };
```



For the code and data segments you keep track of the

- `as_vbase1` tracks the starting address of the segment in *virtual* memory
- `as_pbase1` tracks the starting address of the segment in *physical* memory
  - like the `relocation` register in segmented addresses
- `as_npages1` tracks the size in pages
  - like the `limit` register in segmented addresses

For the stack we just keep track of the physical location of the base (`stackpbase`) since in virtual memory the size, top, and bottom is fixed for all processes.

For `dumbvm` to perform address translation we use contants:

```
USERSTACK = 0x8000 0000
DUMBVM_STACKPAGES = 0xC      // decimal 12
PAGE_SIZE = 0x1000           // decimal 4096 or 4K
```

- First calculate the addresses of the top and bottom 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;
```

- Then translate to to physical address based on which virtual segment the address is in

```
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
    return EFAULT; // segmentation violation (outside valid ranges of segments)
```

- `kern/arch/mips/include/vm.h`
- `kern/include/addrspace.h`
- `kern/arch/mips/vm/dumbvm.c`

When the kernel creates a process to a run a program

- It needs to create an address space for that process and
- Load the program's code and data into that address space

OS/161 pre-loads the entire address space before the program runs

- The `execv` system call will re-initialize the address space of a process

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

- Most OS will load pages on demand (lazy loading)
  - faster program startup times
  - does 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*)

- CS241 used a MERL object file
- OS/161 (and Linux) expects the executable file to be in ELF (Exceutable and Linking Format)

The `program` parameter of the `execv` system call should be the name of the ELF file of the program

ELF Files

ELF files contain address space segment descriptions which describe the segment *images*:

- Virtual address of the start of the segment
- Length of the segment in the virtual address space
- Location of the segment in the ELF file
- Length of the segment in the ELF file
- Defines the global variables
- Also identifies the virtual address of the program's first instruction (i.e. *entry point*)
  - this is usually the `main` function
- They can also contain lots of other information:
  - e.g. section descriptors, symbol tables, relocation tables, external symbol references
  - useful for compilers, linkers, debuggers, loaders and other tools used to build programs

OS/161's `dumbvm` assumes that an ELF file contians two segments:

- *Text segment*: containing the program code and any read-only data
- *Data segment*: containing any other global program data

ELF file does not describe the stack as the initial contents of the stack (e.g. command line arguements passed to `main`) are unknown until run time

The image (binary data) in the ELF may be smaller than the segment it is loaded into (address space)

- This is because the segment has an integer number of pages
- The rest of the address space segment is expected to be zero-filled
- `kern/syscall/loadelf.c`

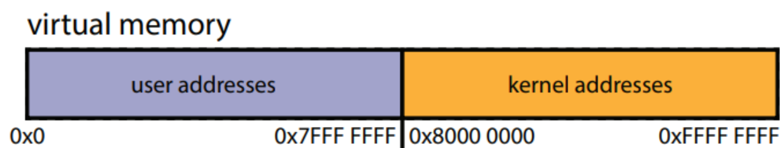
Can check out the some ELF files by running: `cs350-readelf -h widefork.o`

Virtual Memory for the Kernel

We would like the kernel to live in virtual memory but there are some challenges

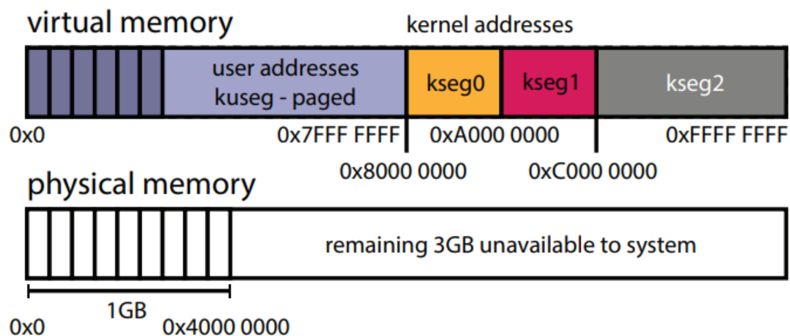
- *Bootstrapping*: the kernel implements virtual memory, so how can it run without virtual memory
- *Sharing*: sometimes data needs to be copied between kernel and user space
  - how can this happen if they are in different virtual address spaces?

Sharing is solved by making the kernel's virtual memory overlap with the process' virtual memories



The solutions to the bootstraping problem are architectre-specific but we will talk about OS/161.

SYS/161 only supports 1GB of physical memory (remaining 3GB are not usable)



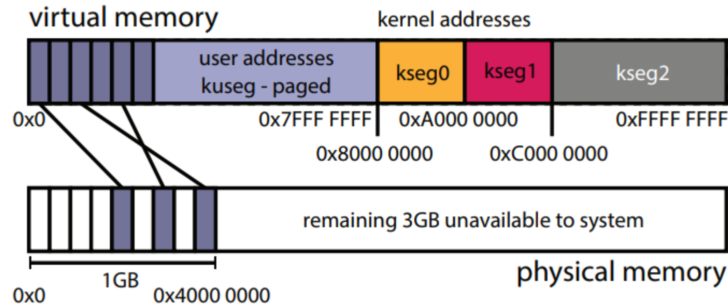
The kernel's virtual memory is divided into three segments:

- `kseg0` (512MB): for kernel data structures, stacks, etc.
- `kseg1` (512MB): for addressing devices
- `kseg2` (1GB): not used

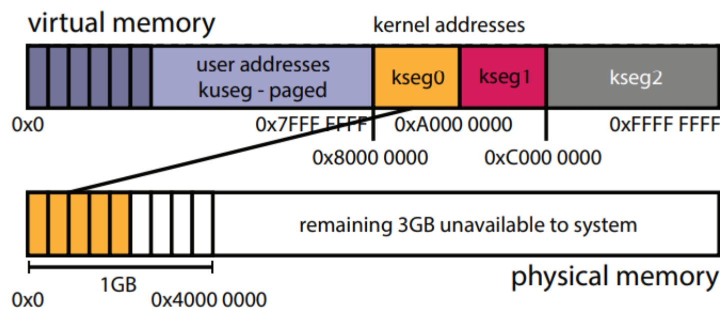
Physical memory is divided into frames which are managed by the kernel in the `coremap`.

Now we will take a closer look:

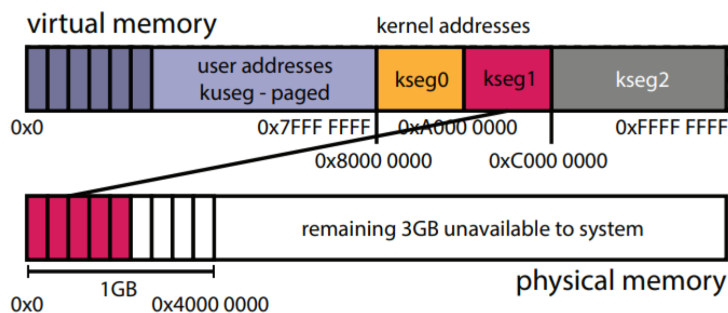
- `kuseg`: user virtual memory is paged
  - kernel maintains the page-to-frame mappings for each process
  - TLB is used to translate `kuseg` virtual addresses to physical ones



- `kseg0`: used for kernel's data structures, stacks, and code
  - to translate a `kseg0` address to a physical one we subtract `0x8000 0000` (i.e. TLB not used)
  - `kseg0` maps to the first 512MB of physical memory, though may not use all of this space



- `kseg1`: used for accessing devices, such as the hard drive or the timer
  - to translate a `kseg1` address to a physical one we subtract `0xA000 0000` (i.e. TLB not used)
  - `kseg1` also maps to the first 512MB of physical memory, though does not use all of this space



Secondary Storage

We want virtual address spaces to be larger then the physical address space (i.e. installed RAM):

- Allow pages from virtual to be stored on secondary storage (i.e. on HDD or SSD)
- `Swap` pages (or whole segments) between secondary storage and primary memory
  - attempt to make them already be in primary memory when they are needed

Definition: *resident set* is the set of virtual pages current present in physical memory

- This set can 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 a `present bit`
  - `valid=1`, `present=1` $\implies$ page is valid and in memory
  - `valid=1`, `present=0` $\implies$ page is valid but not in memory
  - `valid=0` $\implies$ invalid page

Page Faults

Definition: a *page fault* occurs when a process tries to access a valid page not in memory

- Hardware-managed TLB
  - MMU detects this situation when it checks the page's PTE
  - generates an exception, which the kernel must handle
- Software-managed TLB
  - Kernel detects this situation when it checks the page's PTE after a TLB miss
  - TLB should not contain any entires that are not present in RAM

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

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

One issue is that page faults are extremely slow:

- Primary memory is approximetely:
  - 1ns for L1 cache
  - 100ns for RAM
- Secondary storage is approximately:
  - $15,000$ns for SSD (15 microseconds)
  - $15,000,000$ns for HDD (15 milleseconds)

This means that the more improvements from more complex replacement algorithms could be worth it.

If secondary storage access is 1000 times slower than primary storage access then

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

Page faults are quite costly so we have a couple of ways of reducing the number of occurrences:

- *No swapping*: limiting the number of processes, so there is enough physical memory per process
- *Replacement policy*: being smart about which pages are evicted
- *Prefetching*: can hide the latency by getting the page before it is needed

The optimal replacement policy requires future knowledge and is the baseline for the other algorithms:

- MIN: (optimal) replace the page that will not be referenced for the longest time
- FIFO: first page in get evicted first
- LRU: the least recently used gets evicted
  - Clock replacement algorithm

Optimal Page Replacement

Optimal page replacement policy, MIN, replaces 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 0	a	a	a	a	a	a	a	a	a	c	c	c
Frame 1		b	b	b	b	b	b	b	b	b	d	d
Frame 2			c	d	d	d	e	e	e	e	e	e
Fault ?	x	x	x	x			x			x	x	

- This clearly requires knowledge of the future but can be proved to be optimal
- We will compare MIN to the performance of other more practical policies

FIFO Page Replacement

First In First Out (FIFO) says to replace the page that has been in memory for the longest

	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 0	a	a	a	d	d	d	e	e	e	e	e	e
Frame 1		b	b	b	a	a	a	a	a	c	c	c
Frame 2			c	c	c	b	b	b	b	b	d	d
Fault ?	x	x	x	x	x	x	x			x	x	

LRU Page Replacement

Least Recently Used (LRU) evicts the page that has not been used int he 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 0	a	a	a	d	d	d	e	e	e	c	c	c
Frame 1		b	b	b	a	a	a	a	a	a	d	d
Frame 2			c	c	c	b	b	b	b	b	b	e
Fault ?	x	x	x	x	x	x	x			x	x	x

The idea of this is to use the *locality* heuristics to attempt to predict the future:

- *Temporal locality*: programs are more likely to access pages they accessed recently
- *Spatial locality*: programs are likely to access parts of memory close to those they accessed recently

There are some issues with LRU

- Must track usage and find a maximum value which is expensive
- Kernel is not aware about which pages are being used unless there is an exception
- As a result it is difficult for the kernel to implement LRU for its replacement policy

The solution for this is to relax the problem a bit and have the MMU track accesses in hardware

- Add a `reference` bit (or `use` bit) to each PTE
  - MMU sets bit each time the page is used
  - bit can be read and cleared by the kernel
- We do also need to periodically clear the bit

Clock Replacement Algorithm

The clock algorithm (also known as *second chance*) is one of the simplest algorithms for LRU.

- Imagine a *victim* pointer that cycles through the page frames
- When eviction becomes necessary the pointer will cycle through page frames
  - if the `use` bit is set then clear it (set to 0) and move to next
  - otherwise evict the current victim

```
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
6  evict victim                    // its use bit is 0
7  victim = (victim + 1) % num_frames
```

So having the the `use` bit set gives the page frame a second chance in the round robin.

	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 0	a	a	a	d	d	d	e	e	e	c	c	c
Frame 1		b	b	b	a	a	a	a	a	a	d	d
Frame 2			c	c	c	b	b	b	b	b	b	e
Fault ?	x	x	x	x	x	x	x			x	x	x
victim				0	1	2	0			0	1	2

Global vs Local Page Replacement

- *Global replacement*: contents of any frame can be replaced
  - an *unlucky* process can completely lose residency
- *Local replacement*: only contents of frame already given to faulting process can be replaced
  - need to decide at process start up how many frame it needs and could estimate incorrectly
  - faster than global because we inspect less frames to decide a victim to evict
- *Mixed strategy*: do replacements locally then time to time re-distribute RAM allocations based on the observed page-fault frequency of each process

Principle of locality suggests some portions of process's virtual address space are more referenced

- *Work set model*: at any given time some portion of a program's address space will be heavily used and the remainder will not
- This portion of the address space is called the *working set* of the process
- One goal is to make a process's resident set include its working set

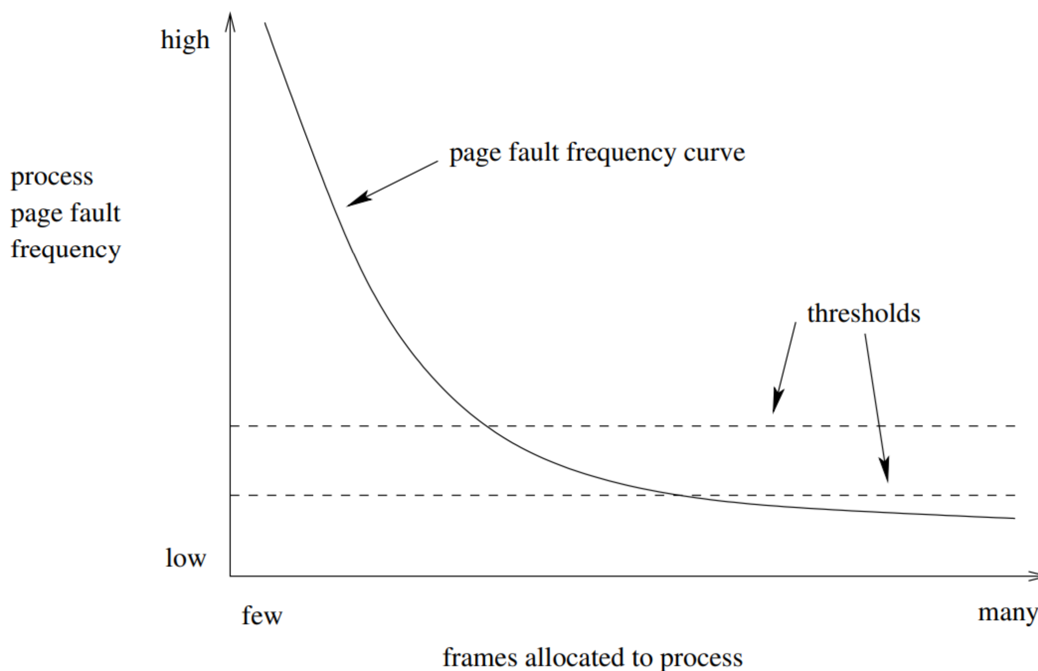
Using the Linux command `ps -o pid,vsz,rss,command`

```
PID VSZ   RSS   COMMAND
805 13940 5956  /usr/bin/gnome-session
831 2620  848   /usr/bin/ssh-agent
834 7936  5832  /usr/lib/gconf2/gconfd-2 11
838 6964  2292  gnome-smproxy
840 14720 5008  gnome-settings-daemon
851 34980 7544  nautilus
853 19804 14208 gnome-panel
857 9656  2672  gpilotd
867 4608  1252  gnome-name-service
```

- Resident Set Size (RSS) or working set is the amount of ram currently allocated to the process
- Virtual Memory Size (VSZ) or commit size is the total amount of meory (RAM + swap) allocated

When re-allocating memory between different processes we look at the number of page faults they generate.

- If a process's page fault frequency is too high we give it more memory
- If a process's page fault frequency is too low it may to give up some memory
- The working set model suggests a sharp *knee* in a page fault frequency plot:
  - big improvements in the beginning
  - diminishing returns over a certain point



It is the designers of the kernel to decide on this upper and lower threshold.

Scheduling

The simplified `Job Scheduling Problem`: say we have a set of jobs

$$\{J_1, J_2, J_3, \dots\}$$

that need to be executed on 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

The job *scheduler* decides which jobs should be running on the server at each point in time

- Inputs: for the i-th job we have two parameters that characterize it
  - *arrival time*: a_i is when the ith job becomes available to run
  - *run time*: r_i is the total length of time required to complete the ith job
- Outputs: for each i-th job we are given, determine two times:
  - *start time*: s_i is when the ith job starts running
  - *finish time*: f_i is when the ith job finishes running

The performance of the scheduler is characterized using two times:

- *Reponse time*: $s_i - a_i$ how long from arrival until it starts running
- *Turnaround time*: $f_i - a_i$ how long from arrival of a job until it finishes running
- e.g. lining up at a grocery store to check out:
  - Reponse time = time waiting in line
  - Turnaround time = time from joining line to paying for groceries

Basic Schedulers

Some basic ideas for scheduling algorithms are

- First Come First Served (FCFS): runs jobs in arrival time order
  - simple and avoids starvation
  - pre-emptive variant: Round-Robin (RR)
- Shortest Job first (SJF): run jobs in increasing order of r_i
  - minimizes average turnaround time but long jobs may starve
  - pre-emptive variant: Shortest Remaining Time First (SRTF)

Gantt Chart

A *Gantt Chart* show the output of a scheduling algorithm (i.e. schedule it created)

- x-axis represents time
- y-axis represents the various jobs
- Bars represent where specific jobs are run
  - beginning of the first bar of job J_i is the start time s_i
  - end of the last bar of job J_i is the finish time f_i
  - sum of all the bars of job J_i is the actual runtime r_i

The arrival time is not shown on the Gantt charts.

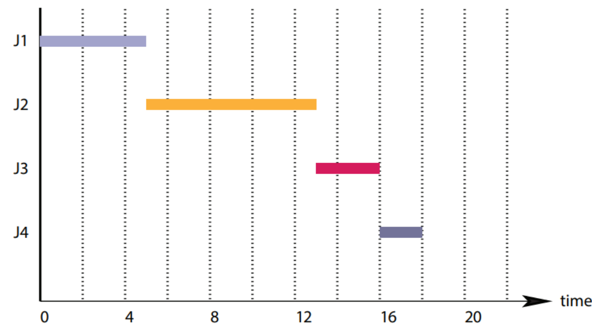
The following table is the input to each scheduling algorithm we will look at in this subsection.

Job	J_1	J_2	J_3	j_4
arrival (a_i)	0	0	0	5
runtime (r_i)	5	8	3	2

- Jobs J_1, J_2, J_3 (in that order) are added to the *Ready* queue at time 0
- Job J_4 is added to the *Ready* queue at time 5

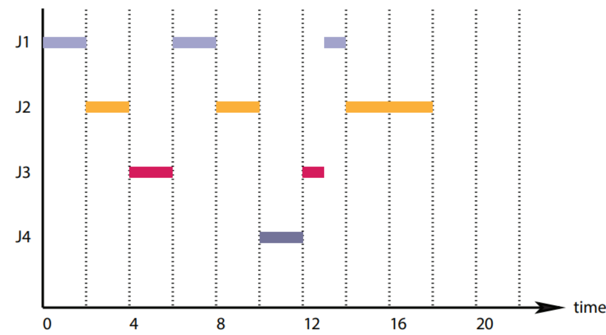
First Come First Served (FCFS)

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



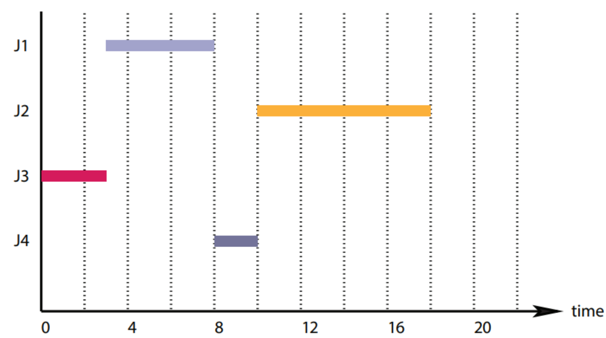
Round Robin (RR)

- Strategy: pre-emptive FCFS, each job runs for 2 units of time
- Scheduler for OS/161 (works even without prior knowledge of run times)



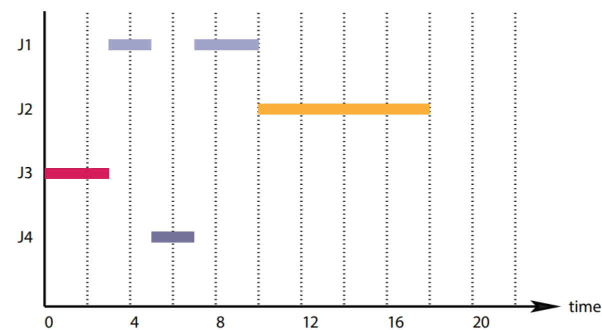
Shortest Job First (SJF)

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



Shortest Remaining Time First (SRTF)

- Strategy: pre-emptive SJF, select job with shortest remaining time
  - preempt if arriving job has shorter remaining time
- Attributes: starvation still possible



Comparing Perforances

Runaround time (i.e. finishing time − arrival time)

- FCFS: average = 47/4 = 11.75

$$(5 - 0) + (13 - 0) + (16 - 0) + (18 - 5) = 47$$

- RR: average = 52/4 = 13.0

$$(14 - 0) + (18 - 0) + (13 - 0) + (12 - 5) = 52$$

- SJF: average = 34/4 = 8.5

$$(3 - 0) + (8 - 0) + (18 - 0) + (10 - 5) = 34$$

- SRTF: average = 33/4 = 8.25

$$(10 - 0) + (18 - 0) + (3 - 0) + (7 - 5) = 33$$

Response time (i.e. start time − arrival time)

- FCFS: average = 29/4 = 7.25

$$(0 - 0) + (5 - 0) + (13 - 0) + (16 - 5) = 29$$

- RR: average = 11/4 = 2.75

$$(0 - 0) + (2 - 0) + (4 - 0) + (10 - 5) = 11$$

- SJF: average = 16/4 = 4.0

$$(3 - 0) + (10 - 0) + (0 - 0) + (8 - 5) = 16$$

- SRTF: average = 13/4 = 3.25

$$(3 - 0) + (10 - 0) + (0 - 0) + (5 - 5) = 13$$

Processor Scheduling

For processor scheduling, the jobs to be scheduled are threads and there are some other differences:

- Runtime of threads are normally not known
- Threads are sometimes not runnable (when they are blocked)
- Threads may also have different priorities
- Performing a context switch to a different thread has a small cost

The objective of this scheduler is to achieve a balance between

- *Responsiveness*: ensure that threads get to run regularly
- *Fairness*: sharing of the processor
- *Efficiency*: account for the cost of switching threads

Processor schedulers are 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. schedule using a weighted fair sharing
     - let p_i be the priority of the i thread
     - try to give each thread a share of the processor in proporition to its priority:

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

Multi-Level Feedback Queue (MLFQ)

Multi-level Feedback Queues (MLFQ) (used in Windows and MacOS)

- Good responsiveness for interactive threads (e.g. threads interacting with keyboard, mouse, display)
- Allow non-interactive threads to make as much progress as possible

The key issue is how do we determine which threads are interactive and which are not.

Key idea: interactive threads are frequently blocked waiting for user input, packets, etc

- Give higher priority to threads that block frequently so they can run whenever they are ready
- Give lower priority to non-interactive threads

MLFQ works off of having n round-robin ready queues: $Q_n, \ldots, Q_1$

- Threads in Q_i have quantum q_i and priority i
  - the higher the level the *higher the priority*
  - the higher the level the *shorter the quantum*
- Preempted threads (quantum expired) will be put at the back of the next lower-priority queue
  - i.e. if a thread from Q_n is preempted it is pushed into Q_{n-1}

- When a thread wakes after blocking it is put into the highest-priority queue
  - since interative threads will block frequenctly they will tend to be in higher-prioty queues
  - non-interactive threads will tend to shift down towards the bottom

To prevent starvation all threads are periodically moved to the highest-priority queue.

Example: L17 slide 372 to 383

Other variants of MLFQ will preempt running lowe-priority threads when a thread wakes to really ensure a fast response to the event (as was done in the previous example)

Some real values:

- Six levels
- Quanta ranging from 40ms for Q_6 to 200ms for Q_1
- All threads raised to Q_1 about once per second to avoid starvation

Completely Fair Scheduler (CFS)

Completely Fair Scheduler (CFS) (used in Linux)

- Assigns each thread a *weight*
- Ensures each thread gets a *fair share* of the processor proportional to its weight

For threads T_i with weights w_i with actual runtimes of a_i we want:

$$a_1 \frac{\sum_i w_i}{w_1} = \cdots = a_n \frac{\sum_i w_i}{w_n}$$

For simplicity we can factor out the $\sum_j w_j$ which gives us

$$\frac{a_1}{w_1} = \cdots = \frac{a_n}{w_n}$$

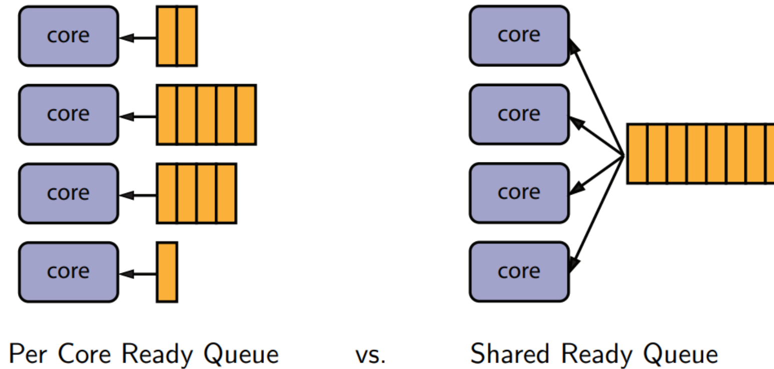
The thread with the lowest $\frac{a_i}{w_i}$ ratio is run next to increase its share of the processor time.

To schedule threads:

- Track the *virtual run time* of each runnable thread
  - the virtual runtime of is the actual run time a_i adjusted byt he thread weights
  - the virtual runtime of T_i is $a_i \frac{\sum_j w_j}{w_i}$
- Always run the thread with the lowest virtual runtime
  - virtual runtime advances more slowly for threads with high weights
- When a thread becomes runnable, its virtual runtime is initialized to some value between min and max virtual runtimes of the threads that are already runnable

Example: L17 slide 388 to 389

Multi-Core Scheduling



- *Per Core Ready Queue*: each core is given its own queue of threads
- *Shared Ready Queue*: every core the same queue of threads
  - accessing the shared ready queue is a critical section (requires mutual exclusion)
  - as the number of cores grows, the fight of the lock becomes problematic
  - as a result the *per core* design scales better to larger numbers of cores

One thing we need to tackle is when to send threads to other queues if they become freed up.

Core Cache Affinity:

- When a thread runs, data is loaded into that processor's caches
- Each core has some memory cache of its own (L1 and L2) and some it shares with other cores (L3)
- Moving a thread to another core means data must be reloaded into that core's cache
- As a thread runs on a core, it acquires an *affinity* for it

So if we do want to move a thread to another core it is a good idea to move one with a lower *affinity*.

Load balancing:

- *Load imbalance* occurs when the queues have different lengths
  - Some core may be idle while others are busy
  - Threads on lightly loaded cores get more processor time than those on heavily loaded cores
- *Thread migration* is moving threads from heavily loaded core to lightly loaded cores

Devices and I/O

Definition: *devices* are how computers *receive input* and *produce output* from/for the outside world

- Some examples of common devices:
  - *keyboard* is an input device
  - *printer* is an output device
  - *touch screen* is both an input and output device
- SYS/161 devices:
  - timer/clock - current time, timer, beep
  - disk drive - persistent / secondary storage
  - serial console - character input/output
  - network interface - packet input/output

Terminology:

- *Bus*: communication pathway between various devices in a computer
  - *internal bus* (a.k.a. memory bus) for communication between processor and RAM
  - *peripheral bus* (a.k.a. expansion bus) allows devices inside the computer to communicate
- *Bridge*: connects two different buses

Evolution of Buses (L18 slide 401) Optional

Device Register

Communication with devices are carried out by interactions with *device registers*.

- There are three primary types of device registers:
  - *status*: tells you something about the device's current state (output)
  - *command*: issue a command to device by writing a particular value to this (input)
  - *data*: used to transfer larger blocks of data from or to device (output/input)
- Some device registers can be a combination of these primary types:
  - *status and command*: read to discover device's state and written to to issue a command
  - *data buffer*: sometimes combined or other times is separated into data in and data out buffers

Example: Serial Console

The serial console has a character buffer

- can be used to read incoming characters from
- can also be used to write outgoing characters

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

- IRQ stands for interrupt request
- We write and read to/from these locations as if we were writing/reading memory
- *Undefined behaviour* occurs if a write is in progress and another write is attempted

To write a character:

- Write the ASCII code of the character to offset 0
- After it is done the serial device will issue an interrupt
- The driver will then check writeIRQ for success or failure
- Finally the driver will clear writeIRQ to acknowledge completion

During a write no other device should write to the serial console.

If the serial console is written to it will generate a interrupt.

Example: SYS/161 timer/clock

This clock is used for preemptive scheduling.

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)

- The offset and size are in bytes (i.e. every value is 4 bytes)
- The offset is relative to a base value (bus location for the device is decided by the OS)
  - e.g. if the base value is `0x1FE0 0000` then we get nanoseconds at `0x1FE0 0004`

How the timer interrupt works

- Write a time at offset 16 for `countdown time`
- Write 1 at offset 8 for `restart-on-expiry`
- Timer interrupt will occur when the `countdown time` reaches 0

http://www.os161.org/documentation/sys161-2.0.8/devices.html

Device Drivers

Definition: *device drivers* are parts of the kernel that know how to interact with a device

- Significant portions of the OS are jsut device drivers (70% for Linux)
- The two methods for interacting with devices are:
  - *polling*: kernel driver needs to repeatedly check device status
  - *interrupts*: kernel does not wait on the device to complete instead gets it to interrupt
    - the interrupt handler then takes care of the exception

Example: writing a charater to serial output 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)
```

- Need to use a semaphore to ensure that only one thread can write to the screen at a time
- Although majority of device drivers are (dynamically loadable) part of the kernel, some exist in user-space (e.g. printer drivers)

Example: writing a character to serial output using *interrupts* requires two separate routines

- 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
  ```

- Interrupt handler for serial device

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

Kernel does not wait for the device to complete and just gets the device to interrupt when it is done (e.g. reading from disk takes a relatively long time)

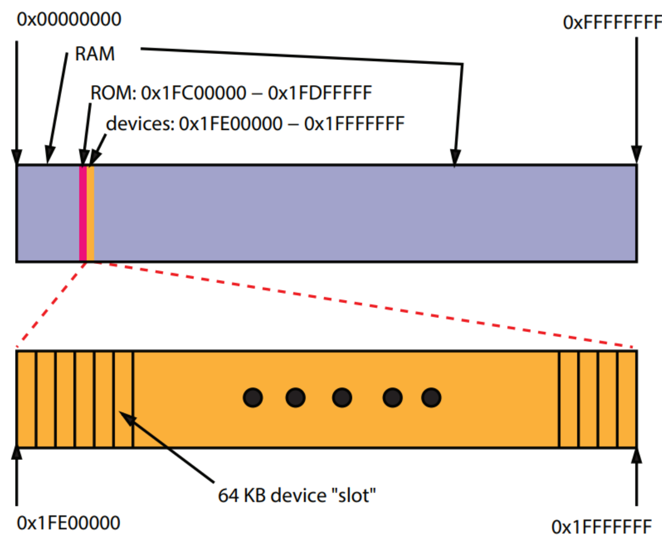
Accessing Devices

How does a device driver access these 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
  - e.g. `in` and `out` instructions on x86
- Option 2: *Memory-Mapped I/O*
  - each device register has a physical memory address
  - device drivers can read from or write to these device register using normal `sw` and `lw`

A system may use both port-mapped and memory-mapped I/O.

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 devices's register and data buffers memory-mapped into its assigned slot
  - a simple device like a timer will only need a few bytes

  - more complex devices that transfer lots of data will use more of that range
  - devices can use the space to store/buffer several item (e.g. keyboard buffering 16 key presses)

Large Data Transfer To/From Devices

Program-Controlled I/O (PIO)

- Device driver moves data between memory and a buffer on the device
- Processor is used to transfer the data

SYS/161 LAMEbus devices use program-conrolled I/O

Direct Memory Access (DMA)

- Device itself is responsible for moving data to/from meory
- Processor is not used to transfer the data and is free to do something else
- DMA is used for block data transfers between devices (e.g. a disk controller and primary memory)

For DMA the device needs its own controller to perform the data transfer.

Example: (L18 slide 413 to 414)

Persistent Storage Devices

Persistent (a.k.a. non-volatile) storage is any device where data persists even after device poweroff

- Primary memory (RAM) is non-persistent
- Secondary storage is persistent

Hard Disks

HDDs are still the most commonly used for persistent staorage because they are inexpensive for the volume of data they store

- store spinning ...

slide 416

for simplicy we will just say that every track will contain the same number of blocks (but in reality the ones futher from the center have more)

Definitions:

- *Seek time*: time it takes ot move read/write head to appropriate track
- *Rotational latency*: time it takes for the desired sector spin to read/write heads
- *Transfer time*: time it takes until desired sectors spin past the read/write heads

$$\text{Request Service Time} = \text{Seek Time} + \text{Rotational Latency} + \text{Transfer Time}$$

previously we had the same number of sectors per track but nowadays to fit more we put more sectors per track at further from the center

slide 421

we observe that

- larger transfers are more efficient
  - this is why we defrag our hard drives
- sequential I/O is faster
  - eliminate the ened for (most) seeks

sequential I/O is not always possible but we can group requests to try to reduce average requst time (historically seek time dominates the request time this is true for SSDs too??)

slide 426

slide 429 (sys/161 disk controller)

read from and to a sys/161 disk

- os/161 will only allow one thread at a time to access the disk
- thread that initates a write should wait until write is completed before continuing
- thread that initates a read must wait until that read is completed before continuing

to enforce this we will use two semaphores

- `disk`: used so that only one thread at a time can use the disk
- `disk_completion`: sempahore so that thread sleeps on after it has acquired exclusive access to the disk but is waiting for its request to be completed (calls `P(disk_completion)` and sleeps on it)
- once disk has completed the rest, device driver will call `V(disk_completion)`

slide 431 writing and reading from sys/161 disk

Solid State Drives (SSD) and Flash Memory

rather than magnetic surfaces we use integrated circuits so we have no mechnical parts

DRAM requires contsnta power to keep values but Flash memory uses quantum properties of electrons (to trap them?)

Flash memory is divided into blocks and pages

- 2,4,8KB pages
- 32KB-4MB blocks

for flash memory reading/writing can be done at page level

- pages are initialzed to 1
- transition $1 \to 0$ can occur at the page level (i.e. writing a page to place we never written before)

but overwriting/deleting must be done at the block level

- high voltage is required to switch $0 \to 1$
- we can only apply hihg voltage at the block level

to write to an SSD a naive solution

- read whole block into RAM
- re-initialiez block (set all page bits back to 1)
- update block in RAM then write back to SSD

and SSD control can handle these requests in a much better way

- page to be deleted/overwritten is marked as invalid
- write to an unused page
- update translation table
- requies garbage collection

each block of an SSD has a limited number of write cycles before it becomes read-only (roughly 1,000 times for consumer SSDs, 1,000,000 for enterprise)

- SSD controllers perfor wear leveling to discribute the writes more evenly accross blocks so that blocks wear down at a more even rate
- defragementation is harmful for lifespan of an SSD

Files and File Systems

Definition: *files* are persistent, named data objects

- when you edit a file the change is not *persistent* until you save it
- files may change size and content over time and have associated metadata
  - e.g. owner, file type, date created, date modified, acces controls
  - `stat file`

a variable like `b=10` or `c=5` has a name but `b + c` does not have a name (it is an intermeddiate result until we assign it to a varaible)

Definition: a *file systems* are the data structures and algorithms used to store, retrieve, and access files

- a file system can be separated into three different layer
  1. *logical file system*: the high level API used to manage the system information
  2. *virtual file system*: abstraction of the lower lvel file systems (presents different underlying file systems (HDD, SDD, DVD, network drive) to the user as one)
  3. *physical file system*: how files are actually stored on the physical media (e.g. track, sector, magnetic orientation)

some common file oeprations are (note that these are the linux operations)

- `open` returns a file descriptor (or identifier or handle)
  - other file operations will take this file descriptor as a paramter
- `close` invalidates a file descriptor for a process
  - 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
- to get or set file meta-data we can use `fstat`, `chmod`, `ls -la`

each open file (valid descriptor) has an assocaited file position

- hte position starts at byte 0 when the file is opened
- Read and write operations
  - start from the current file position and
  - update the current file position as bytes are read/written
- this makes sequential file I/O easy for an application to request
- seek (`lseek`) is used for non-sequential file I/O

- `lseek` changes the file position associated with a descriptor
- the next read or write from that descriptor will use the new position

databases are the expection as they allow for more random access from multiple parties

continuing with file systems

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);
```

file reading example using seek (for non-sequential reading)

```
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);
```

`lseek` does not check if new file position is valid (in fact it doesn't do anything except updating file position parameter in the kernel) however `read` will error

Definition: a *directory* (linux, folder for windows/mac) maps a file name (string) to *i-numbers*

- the *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 metadata for the i-numbers corresponding file

A directory provides 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
  - directories are interior nodes (if they are non-empty) or leaves (if they are empty)

files are identified by *pathnames* with descibe a path through the directory tree to reach the file

- absolute paths are relative to the root of the file system and always being with slash
- only the kernel has direct access to edit directoires since it is one of the data structures that the kernel relies on (so needs to perform many error checking)

Definition: a *hard link* is an association between a name (string) and a i-number (many to one relationship between name and i-number)

- every 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/new_aaaa.txt", "O_CREAT|O_TRUNC")`
  - we can `stat` the file and the Inode is the i-number???
- we can perform `ln new_aaaa.txt a.txt` to create a new hardlink to the same file
  - they share the same Inode
  - note is similar but not the same as `ln -s` (symbolic link)
  - when there are no more files that have some Inode the directory marks space as empty (to be written to) (lazy deletion)
    * the manufactours will usually give you some way to do a entire erase
    * even if you try to write all 0s the device will be "smart" and try to do wear leveling and may not overwrite all parts
- each file
  - has a unique i-number
  - but may have multiple pathnames (by hardlinking)
- in order to avoid cycles it not possible to `ln` (hardlink) a directory
- if there are at least one hardlink to the file then we cannot delete it

Definition: *symbolic link* or `soft link` is an association between a file name (string) and a pathname

- `ln -s /doc/aaa /doc/m` the link `/doc/m` references the pathname `/doc/aaa`
- notice if we `stat` a symbolic link we see its size is just the refrence
- referential integrity is not preserved (i.e. moving `/doc/aaa` will break the softlink)
- when we open the softlink we end up just opening the file it references
  - this means we can create a softlink to a file that does not exist
  - while the hardlink will take on the file's Inode so requires it to exist
- cycles?
- `ls -li` to also list the Inodes of the files

Multiple file systems

- it is not uncommon for a system to have multiple files systems
  - `df-T` command on Linux
  - plugging in a USB is another file system

- network drives are another file system

- some kind of *global file namespace* is required
  - uniform across all file systems
  - independent of physical location
- windows: uses two-part files names the file system name and pathname within file system, e.g: `C:\user\cs350\schedule.txt` (automounted)
- linux: creates a single hierarchical namespace that combines all the namespaces of multiple, requires manually mounting via `mount` syscall

Definition: *mounting* does not combine the two files systems into a single file system

- it 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
- todo

- modern hard drives are quite fault tolarent and have a small mount battery for last minate stuff even if the power goes out
- the actual file system takes a bit of space
- what needs tobe stored persistently?
  - ...
- what inforamtion is non-persistanct?
  - ...

- what needs tobe stored persistently?
  - file data
  - file meta-data
  - directories and links
  - file system meta-data (the data structures for the file system)
- what inforamtion is non-persistanct?
  - per process open file descriptor table
    * file descriptor
    * file position for each open file
  - system wide
    * open file table (info about which files are currently open)
    * *cached* copies of persistent data

Example: very small file system

- Use an extremely small disk (256 KB disk)
  - disk will have sector size of 512 bytes (typical value)
    * RAM is usually byte addressable
    * Disks are usually sector addressable
  - for this 256KB disk we have 512 total sectors
- Group every 8 consecutive sectors into a block fo size 4KB
  - better spatial locality (fewer seeks)
  - Reduces number of block pointers (more about this later)
  - 4KB block is convenient size of demand paging
  - total of 64 blocks on this disk
- we decide (ahead of time) how much space to reserve for data v meta-data
- This decision will depend on part on how much meta-data you will track for each file
- the goal is to maximize the same for storing file's contents
  - for our case we will reserve the first 8 blocks for meta-data and last 56 blocks for data
- next how to map files to their data blocks
  - create an array of i-nodes where each i-node contains the meta-data for a file
  - the index of the i-node array is the file's index 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 must be exactly one i-node per file)

    - this is why there is a limit of number of files (e.g. exFAT 2,796,202)
- next 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 $\implies$ a block size of 4KB to track 32K i-nodes (or 32K data blocks)
        * this is fare more then we actually need for this file system
        * we only need to track at most 80 of them
- next we reserve the first block as the superblock
    - superblock contains meta-information about the entire file system
        * e.g. how many i-nodes and blocks are in the system, locaiton of the i-node bitmap, location of hte data block bitmap, the locaiton of the i-node table, etc
        * this is what is read when you plug in a USB/external HD
- Summary of VSFS format (slide 467)

the i-node is a fixed size index structure (for a given file system they are all the same size) htat holds the file meta-data and pointers to the data blocks

- i-node field may include
    - file type
    - file permissions
    - file length
    - time of last file access/update, last i-node update
    - number of hard links t othis file
    - pointers to data blocks (the i-number)
- since the i-node is of a fixedsize, how can it point to all the data blocks for the file
    - for small files: pointers to the i-node are sufficient to point to all data blocks (i.e. direct data block pointers)
    - for larger files: we need single, double, or triple indirect data block pointers
    - (slide 469) for how this would work (first couple will be direct then later pointers are single, double, triple indirect)
    - notice that each data block can be fragemented into other places
    - this is why there are limits on file size (e.g. FAT32 4GB file size limit)

- this is simular to how page tables work

- if disk blocks are referenced with 8-byte (32-bit address)
  - then there are 2^{32} blocks eahc of size 4KB
  - so maximum disk size is $2^{32} \times 2^{12} = 16\text{TB}$
  - the maximum file size if we only use direct pointer is

$$12 \times 4\text{KB} = 48\text{KB}$$

  - this is great for small files but not good for big files
- indirect pointer
  - each indirect pointer points to a block full of direct pointers
  - 4KB block of direct pointers holds 1,024 (4-byte) pointers so max file size using 12 direct and 1 indirect pointer is

$$(12 + 1{,}024) \times 4\text{KB} = 4{,}144\text{KB}$$

  - if the disk were larger we can add
    * double indirect pointer: $(12 + 1024 + 1024^2) \times 4\text{KB}) \approx 4\text{GB}$
    * triple indirect pointer
  - notice it can take up to 3 pointer follows to get to the actual data but luckly since we are assuming sequenctial linear read by the time we get there we can preload

Directoies are implemeted as a special type of files that contains directoy entires

- contains many entry pairs of
  - i-number
  - a file name
- these directory files can read by application programs
- directory files can only be updated by the kernel, in response to syscalls (e.g. create file, create link, etc)

in order to speed up file access the kernel keeps some information in RAM which is updated as the prcesses access files

- per process (descriptor table)
  - system call open returns a file descriptor
  - the descriptor table tracks
    * the file descritpors this process have open
    * the files each open descriptor referes to
    * current file position for each descriptor

- system wide
  - open file table: files are currenlty 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`

- the superblock will contain the inode for the root (for Linux this always 2)
  - Linux i-node 1 is reserved for tracking bad blocks
- read the root i-node to get the location of root's data block which stores the root directoy
- use root's data block to find the i-number for foo
- read foo's i-node which provides the location of foo's data
- read foo's data (also a directoy to find bar's i-number)
- bar's i-node is read and
  - permissions are checked
  - file descriptor is returned and added to the rpocess's file descriptor table
  - file s added to kernel's open file table

it take 5 disk reads just to open the file!

now to read data from `/foo/bar` one block at a time

- bar's i-node is read and a pointer to bar's 0th data block is found
- data block for /foo/bar is read
- bar's i-node is written to update the access time but not the directoies foo or bar as a deisgn decision to speed up the filesystem
- two more data blocks are read ie. previous three steps are repated two more times

slide 483 notice the read/write on 8/9 (this is because there are multiple inodes on the same block so we need to read update then write when we update one of them) (16 inodes in one block, we can only read/update by the block)

Chaining: VSFS uses a per-file index (direct and indirect pointers) to access blocks

Two alternative approaches:

- Chaining: each block includes a pointer sto the next block
  - Implentation: directoy table contains the name of the file paired with its starting block (and possibly its end block to facilitate appending)
  - Performance: acceptable for sequential but very slow for random access
    - must be through the file sequentially (block by block) to get to a particular location

- External Chaining: chain is kept as an externel structure
  - Bill Gates invented this
  - Microsoft File Allocation Table (FAT) uses this and is very standard
  - ...

File system design ...

Btrfs (b-tree fs, whoa uses b-tree!)

File system failure tolerance

- one file system operation may require severl disk I/O operations, e.g. deleting a file
- what if, due to a failure, some but not all changes were written to disk
- we want to make sure the strucutres are *crash consistent*

Fault tolerance

- speical-purpose consistency checkers (e.g. fsck)
  - runs after a crash before the normal operations resume
  - attempts to repair inconcsistent file system data struectres such as
    * file with no directoy entry
    * free space that is not marked as free
- Journaling (e.g. NTFS, ext3/4)
  - write-ahead logging
    * first record the file system meta-data changes in a journel (logger) so that the squences of changes can be written to disk in a single operation
    * after changes have been journaled, updated disk data
    * these need to be written to secondary storage
  - after a failure, redo the journaled updates in case they were not completed before the failure (heyo recall databases logging)