

Quiz 4 sample questions	3
Quiz 3 sample questions	3
Quiz 2 sample questions	3
2019 midterm	6
2018 midterm	6
Lab 5	6
Lab 4	8
Lab 3	9
Lab 2	11
L1 intro to System Software	16
L2	18
L2 Think Time	22
L3	22
L3 Think time	26
L4	27
L4 think time	30
L5	31
L5 think time	35
L6	37
L6 think time	40
L7	40
L7 think time	46
L8	48
L8 think time	50
L9 threads API	51
L9 thinking time	56
L10 thread implementation	57
L10 think time	62
L11 kernel threads and user threads	63
L11 think time	64
L12 mutual exclusion	64
L12 think time	69

L13 Implementing mutex locks	69
L13 think time	72
L14 Using Mutex locks	73
L14 think time	75
L15 misusing mutex lock	75
L15 think time	77
L16 intro to synchronization	78
L16 think time	80
L17 monitors	81
L17 think time	84
L18 semaphores	84
L18 think time	88
L19 classic synchronization problems	90
L19 think time	93
L20 scheduling policies	94
L20 think time	95
L21, L22, L23 batch and interactive scheduling	96
L21 think time	101
L22 think time	102
L23 think time	102
L24 Unix Process	103
L24 Think time	104
L25 POSIX threads	105
L26 Memory Management Overview	106
L26 Think time	108
L27 Virtual Memory Hardware	109
L27 think time	113
L28 Translation Lookaside Buffer TLB	113
L28 think time	115
L29 Demand Paging	115
L29 think time	117
L30 virtual memory implementation	118
L30 think time	120
L31 process and virtual memory	120
L31 think time	123
L32	123

L33 Page Replacement	123
L33 think time	124
L34 Disk and Raid	124
L34 think time	128
L35 file system introduction	128
L35 file system design	133
L35 think time	141
L36 Unix file operations	141

Quiz 4 sample questions

Q1 TLB

PID = 1+1=2

Quiz 3 sample questions

Q4 paging 48bit virtual addr, PT=4KB, size of each entry = 8 bytes

a) Levels needed in page table?

Offset = 12bits

Page number = 48-12 = 36bits

Entries in each PT = 4KB/8bytes = $2^{12}/2^3 = 2^9$ entries

To denote these 2^9 entries, need 9 bit

In total: $36/9 = 4$ levels

b) Combined max number of PTE that can be used in topmost 2 levels of page table?

Each level: 2^9 entries

Each entries in topmost level matches to a PT in second topmost level

Total PTE: 2^9 (first level) + $2^9 * 2^9$ (second level) = $2^9 + 2^{18}$

c) Want to reduce 1 level and ensure the largest page table at any level is as small as possible, what is the new page table size?

Page level = 3

Page number = 36

Entries in each page = $36/3 = 12 \rightarrow 2^{12}$ entries

Each page entry size = 2^3

New page table size = $2^3 * 2^{12} = 2^{15} = 32KB$

Quiz 2 sample questions

1. Thread Switching

You know that implementing thread switching can be a little tricky.

a. To simplify matters, you start by implementing thread switching as shown below. Assume that the arguments to thread switch are initialized correctly and getcontext/setcontext do not return any errors. Clearly explain the problem with the code shown below.

```
thread_switch(ucontext_t *curr_context, ucontext_t *next_context){
    getcontext(curr_context);
    setcontext(next_context);
}
```

- Getcontext will save the current state of the CPU registers. The PC will be pointing at the next instruction, which is one line after getcontext i.e. setcontext. When setcontext is called, CPU registers are restored to the registers stored last time when getcontext is called, the program will run setcontext again because this is the next instruction that is stored previously. The program will run as an infinite loop here

b. Fill the partial code shown below to solve the problem in the original code listing. Make sure to initialize any variables.

```
thread_switch(ucontext_t *curr_context, ucontext_t *next_context){
    volatile int setcontext_called = 1;
    getcontext(curr_context);
    if (setcontext_called == 1){
        Setcontext_called = -1;
        setcontext(next_context);
    }
}
```

this keyword works as a condition variable that keeps track of whether the program has been at this instruction. If **setcontext_called** == 1, the program visits this for the first time, thus allowing it to proceed into get setcontext. If it == -1, that means we are coming back from setcontext, therefore should not setcontext again

2. Monitors and Semaphores

We have discussed how monitors and semaphores can be used to solve all synchronization problems. They are thus equally expressive, meaning that each can be used to implement the other. Your task is to implement semaphores using monitors. Write the code for the down and up semaphore functions below. You can assume that mutex locks and condition variables implementations are available to you. You are not allowed to disable interrupts in your code

Down:

```
lock(sem->mutex)
while (sem->value <= 0)
{
    wait(sem->cv,sem->mutex);
}
sem->value -= 1;
}
unlock(sem->mutex)
```

Up:

```

{
lock(sem->mutex)
em->value += 1;
signal(sem->cv,sem->mutex));
}

```

3. 3. East or West

The Bloor Viaduct Bridge is undergoing repairs and only one lane is open for traffic. To prevent accidents, traffic lights have been installed at either end of the bridge to synchronize the traffic going in different directions. A car can only cross the bridge if there are no cars going in the opposite direction on the bridge. Sensors at either end of the bridge detect when cars arrive and depart from the bridge, and these sensors control the traffic lights. Your aim is to ensure that cars don't crash and cross as efficiently and quickly as possible.

a. A skeleton implementation of the four routines, east arrive, east depart, west arrive and west depart, is shown below. You may assume that each car is represented by a thread, and cars arriving from the East side call the east arrive function when they arrive at the bridge (on the East side) and east depart function when they leave the bridge (on the West side), and vice versa for the cars arriving from the West side.

East car comes, hold bridge, make it down, then leave east_arrive and unlock crit sec. West car comes, hold lock, wait in down(). Since lock is held by west, east depart cannot release the bridge and thus causes a deadlock

b. In this implementation, we have modified the code shown in (a) by adding the unlock and lock functions before and after down, as shown in red font.

East comes, east == 0
 Before it downs the bridge
 Another east comes, east == 0 as well
 Down the bridge again
 In east depart, east -- = 0 will up the bridge
 And then first car can down the bridge
 Not efficient

3c. In this implementation, we have modified the code shown in (a) by using two different locks, as shown in red font.

- This will work
- Lock ensures no two cars waiting on same bridge, one can only pass after another passes so inefficient
- Up and down ensures bridge is only acquired by 1 resource
- BUT THIS IS NOT FAIR
 - Cars arriving from one side can block cars from the other side

4.a

T1: arrive at 0, run for 20s
 S = 1
 $N = (1\%3)+3 = 1+3 = 4$

Additional 4 threads arrive at 10, run for 10 sec
TS = 5, interrupt every second
Response time = arrive - start running for first time

- a. $N = 4$
- b. 20
- c. 6
- d. 6, assume all priorities are 0 at the beginning

2019 midterm

- 2.a: Race condition
- 2.b: yes, but this will cause GPU only running with 1thread, so no speed up
- 2.c: will not work. Another thread running another `l[j]` can still get in the crit sec
- 2.d: yes will work. Last iteration has `i == size`, so no lock, but has an unlock

2018 midterm

- 3.a: thread calling `thread_wait` waits until target thread exits
- 3.b: 15 14 13 12 ... 1 0 main
- 3.c: will run next thread before it is created in the main loop, then order will not be maintained
- 4.a:
 - 1. Not possible: since nothing wakes up T1, it will sleep forever
 - 2. Okay: T1 wait at sleep, not acquiring, context switch to T2, it makes count increment and wake up T1, then T1 runs line 8
 - 3. Not possible. If T2 calls thread wake up, when T2:5 runs, since interrupt is disabled, and T2 does not sleep, T1 should not come in until T2 finishes
 - 4. Problem: `Sem < 0` as both T1 and T5 are decrement sem
 - 5. Problem: T2 wakes up T1, but since T3 comes and make sem full again, (with while loop) T1 will not be able to be woken up and will sleep again (with if loop) sem will be decremented again, resource held by T1 T3 at the same time
 - 6. Not possible: T3 only wakes up 1, but both threads waking up

Lab 5

Caching web-server

- Benefits of using multithread for web requests:
 - 1. Using multiple cores for machine, so request can be processed concurrently
 - 2. Allows processing a request while another request is fetching file from disk
- Disadvantage:

- 1 thread fetch 1 file request slows down request process since disk accesses is slow
- Overall throughput is decreased since disk speed is the bottleneck for webserver
- Solution: caching
- 2 ways of implementing file caching
 1. Block granularity
 - a. Some file blocks are cached some others are not
 - b. If only parts of file is accessed, no need to cache entire file
 - c. Memory management is easier because block has fixed size
 - d. OS typically implement this
 2. File granularity
 - a. A file is either entirely cached or not cached at all
 - b. Webserver reads entire files
 - c. Need to ensure these file will fit in memory
 - d. Can use malloc() and free() to allo/deallocate memory
- To avoid double caching, caching of OS is disabled in this assignment
- Benefit of caching in web server:
 - Server program has complete control over files to be cached/evicted
 - Avoid caching large files or unlikely requested files
 - Evict larger files before smaller files
- In implementation:
 - Use a hash table to lookup cached files
 - Key = file name
 - Data = file data fetched from disk
- Cache must only be allowed to grow to a max limit
 - Max_cache_size == max amount of bytes used in file cache
 - 100000
 - Increase this value improves performance
 - Ensure cache does not grow beyond this limit
 - Evict another file before inserting a newly fetched file in cache
- cache_lookup(file), cache_insert(file), and cache_evict(amount_to_evict). You may add any other parameters to these functions.
- Locks needed to ensure mutual exclusion
 - Use a single lock that protects the entire cache
 - While reading a file/sending file to client, SHOULD NOT HOLD this global lock
- Eviction algorithm
 - Keep an LRU list of cached files
 - Update this list on cache lookup or insert operation
- Cache must not store multiple copies of a file
 - Busy-waiting
 - Solution:
 - 1. allow threads to read the same file concurrently If file is not cached currently

- When returned from a file read, thread check whether the file is already cached, if yes, avoid caching, free the buffer containing the file contents
- 2. Synchronization between requests for reading the same file from disk
- Make sure lock and synch are done
-

Lab 4

HTTP

- Web browsers and web servers interact through text-based protocol called HTTP
- Web browser opens internet connection to web server, then request content, web server responds with requested content, then closes connection, browser displays content on screen
- Each content on server is associated with a file
 - E.g client requests a specific disk file (static content)
 - Client request executable file to run and return its output (dynamic content)
- Each file requested from server: URL (Universal Resource Locator)
 - `http://www.eecg.toronto.edu:80/Welcome.html` identifies an HTML file called "index.html" on Internet host "www.eecg.toronto.edu" that is managed by a web server listening on port 80. The port number is optional and defaults to the well-known HTTP port of 80.

Multithreaded web server

- Single-thread web servers
 - Suffers from performance problem: only 1 HTTP request can be serviced at a time
 - Other clients have to wait until the process is finished
 - Problem
 - User-utilizing the machine
 - Request for file resident only on disk (not in memory), then request will be delayed until the file is fetched from disk
- Multi-threaded web servers
 - Spawn a new thread for serving each new HTTP request
 - OS will schedule threads according to its own policy
 - Adv:
 - Short requests will not need to wait for long request
 - 1 thread blocked waiting for file fetching, other threads continue to handle their requests
 - Disadv:

- 1thread/request: pays overhead of creating new thread on each request
- Preferred approach: multi-threaded server through creating fixed-size pool of worker threads when web server first started
 - Each thread is blocked until HTTP request for it to handle
 - If worker > active request
 - Some threads are blocked, waiting for new HTTP request to arrive
 - If worker < active request
 - Request be buffered until there is a ready thread
 - A master thread
 - Begins by creating a pool of worker thread
 - Specify number on command line
 - Master thread is responsible for accepting new HTTP connections, place socket descriptor for the connection into fixed-size request buffer
 - Place connection descriptor → fixed-size buffer
 - Return to accepting more connections
 - No read/performance
 - Num of element in buffer: spec by cmd line
 - Each worker thread wakes up when HTTP request is in the queue
 - Wake up 1 thread, process 1 HTTP by performing a READ on the network descriptor, obtains specified content by reading the file requested, return content to client by WRITING to descriptor
 - Then wait for another HTTP request
 - Request handled in order dep on how long the process takes + how OS schedules active threads
 - Master and worker threads == producer-consumer relationship
 - Require their accesses to share buffer be synch
 - Master thread block and wait in buffer is full
 - Worker thread wait if buffer is empty

Lab 3

Timer signals

- User-level codes cannot use hardware time interrupts
 - POSIX OS gives “signals” to simulate “interrupts” for users
 - E.g. ctrl-c kills a program
 - OS sends SIGINT signal to program, kills process
 - If want to save state of program, register a handler with OS, and then when user hits ctrl-c, OS calls handler, write out current state and then exit
- A form of asynchronous, inter-process communication mech
- 1 process → another process
- 1 process → itself
 - Use this to have process that invoke your user-level scheduler
 - Thread library functions
 - And deliver timer signals to itself

- OS delivers a signal to a target (recipient) process → interrupting normal flow of execution
 - Can happen anytime
 - Process register a signal handler that is invoked when signal is delivered
 - Signal handler finishes executing → normal flow of execution of process is RESUMED
- Signal handler can have race conditions
 - E.g. For example, if you increment a counter (counter = counter + 1) during normal execution as well as in your signal handler code, the increment operation may not work correctly because it is not atomic, and the signal may be delivered in between the instructions implementing the increment operation.
 - To avoid: disable signal delivery while counter is being updated

Interrupt.c functions

- *void register_interrupt_handler(int verbose):*
 - Install timer signal handler in calling program with
 - Sigaction system call
 - Is invoked when a timer signal fires
 - Verbose flag: a message is printed when function runs
- *int interrupts_set(int enabled):*
 - Enable = 1
 - Enables timer signals
 - Enable = 0
 - Disables timer signals
 - *Signal state*
 - Current enabled/disabled state
 - Returns whether signals were previously enabled or not
 - Allows “stacking” calls to this function
 - ```
fn() {
```

      - ```
/* disable signals, store the previous signal state in "enabled" */
```
 - ```
int enabled = interrupts_set(0);
```
      - ```
/* critical section */
```
 - ```
interrupts_set(enabled);
```
      - ```
}
```
 - First call *interrupts_set* disables the signal
 - Second call restores signal to its previous state rather than unconditionally enabling signal
 - Useful bc fn may be expecting signals to remain disabled after fn finishes exe
 - OS insures reading prev state & updating pre state are performed atomically
 - Sigprocmask system call is used
 - Use this to **disable signals** when running any code that is in a **crit section**
 - *int interrupts_enabled():*

- Returns whether signals are enabled or disabled currently
- Use this to check
- `void interrupts_quiet():`
 - Turns off printing signal handler messages

Context switches

- as a result of thread context switches, the thread that disables signals may not be the one enables them
- The signal state is saved when `getcontext` is called (recall the `show_interrupt` function in the `show_ucontext.c` file is Lab 2), and restored by `setcontext`.
- if you would like your code to be running with a specific signal state (i.e., disabled or enabled) when `setcontext` is called, make sure that `getcontext` is called with the same signal state.

Lab 2

Background on threads

- Threads provide illusions: different parts of the program that are running concurrently
- Threads share code, heap, runtime system
- Threads have separate stack and set of CPU registers
 - Such provides synchronization primitives so different threads can coordinate to access and share resource

User level v.s. Kernel threads

- This project is done at user level:
 - Construct user threads by implementing a set of functions that allow program to call directly to provide the illusion of concurrency
- Modern OS provide kernel thread:
 - User program invokes corresponding kernel thread function using system calls
- Both types of thread use the same core technique for providing concurrency abstraction, and both threads can be built the same way
- Differences between kernel and user threads:
 1. Multiprocessing
 - a. User:
 - i. Provide illusion of concurrency using multiple processors
 - ii. Kernel schedules user process on 1 CPU, and user-level thread package multiplexes the kernel thread associated with the process between 1+ user threads
 - b. Kernel:
 - i. Actual concurrency
 - ii. Kernel is aware of the different kernel threads
 - iii. Can simultaneously schedule these threads from the same process on different processors
 - c. In this project:
 - i. Allow programs to multiplex some (m) number of user-level threads on 1 kernel thread
 1. Only 1 user level threads is running @ a time
 2. The runtime system has complete control over the interleaving of user-level threads with each other

2. Asynchronous I/O

a. User:

- i. When Make a system call that blocks (e.g. reading file), the kernel scheduler move the process to BLOCKED state and do not schedule it until I/O has completed
 1. Even other user threads are within that process and are ready, they need to wait
- ii. When kernel thread blocks for a system call, kernel scheduler is aware that other threads are in the same process can run
 1. Thus some kernel threads can run while others are waiting for IO

3. Timer interrupts

a. Next lab:

- i. simulate timer interrupts that cause the scheduler to switch from one thread or process to another by using POSIX signals
- b. Threads library will turn off interrupts by blocking delivery of signals using system call
- c. User: But there is nothing to prevent the signals from turning off interrupts, and can be preempted until it calls yield, thus hogging the CPU
- d. Kernel: do not have the problem
 - i. Only privileged code in kernel can turn off real time interrupts

Threat context

- Per-thread state: represents working state of the thread
 - PC, local variables, stack
- Thread context is a subset of the state
 - Only include a pointer to the stack, not the entire stack
 - Library will store the thread context in a per-thread data structure
 - Called thread control block

saving/restoring thread context

- When a thread yields, threads library must:
 - save the current thread's context (contains the processor register values at the time of yield)
 - Restore the saved context later when this thread runs on the processor
 - Creates a fresh context and a new stack when it creates a thread
- C runtime system allows
 - Retrieve current context
 - Store it in a memory location
 - Set its current context to a pre-det value from memory location
 - Use library calls:
 - Getcontext
 - Saves current context to a struct (type struct ucontext of type ucontext_t)
 - When allocate a struct ucontext + pass a pointer to that memory to a call to getcontext
 - Current registers and other context will be stored in that memory
 - Setcontext

- Call setcontext later to copy that state from memory to processor, restore the saved state
- The struct ucontext is defined in /usr/include/sys/ucontext.h. Look at the fields of this struct in detail, especially the uc_mcontext and the uc_sigmask fields.
- 1. Suspend a currently running thread:
 - a. Save its state w getcontext
 - b. Restore state w setcontext
- 2. Create a new thread
 - a. Getcontext to create a valid context
 - i. Leave the current thread running
 - ii. The current thread will then change a few reg in its valid context to initialize it as a new thread
 - iii. Put the new thread to ready queue
 - b. New thread is chosen by schedule, and will run when setcontext is called on this new thread's context

Changing thread context

- When creating a thread, cannot make a copy of the current thread's context using getcontext
- Need to make a copy, then change
 1. Program counter to point to first function that thread will run
 2. Allocate a new stack
 3. Change the stack pointer, point to top of the stack
 4. Setup parameters to the first function
- Makecontext, swapcontext CANNOT BE USED
 - Must manipulate fields in the saved ucontext_t directly by:
 - Change the program counter --point→ a stub function
 - Is the first function the thread runs
 - Malloc a new per-thread stack
 - Change stack pointer --point→ top of the new stack
 - Initialize argument registers with arg that will be pass to the stub function

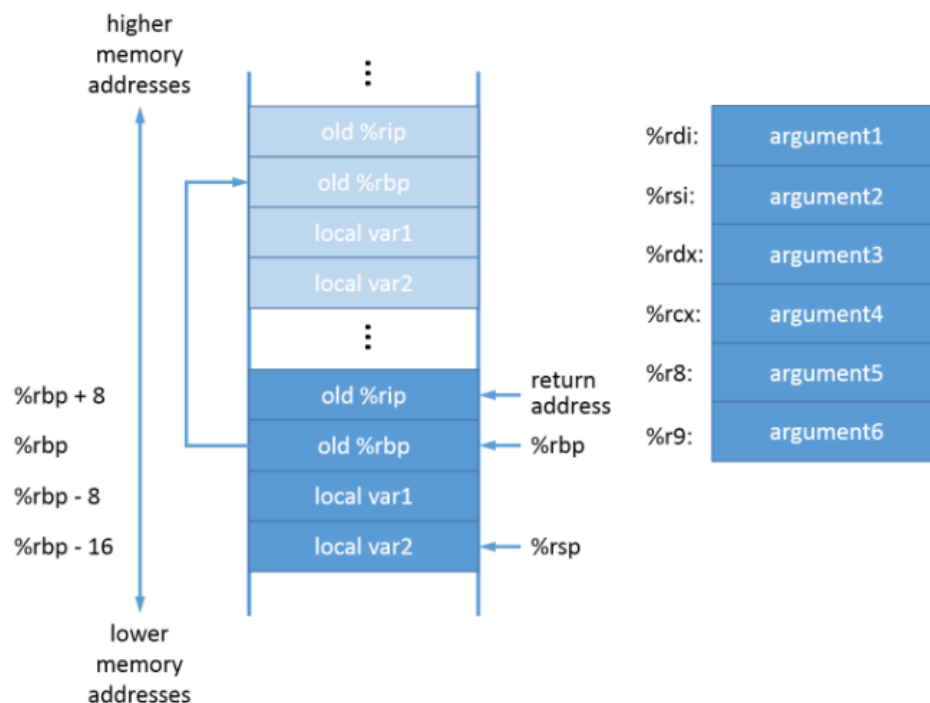
Stub function

1. Create a thread
 - a. Setup a thread that starts running thread_stub
 - b. So when thread runs, it will call thread_stub
 - c. Thread_stub calls thread_main
 - d. Thread_stub is input with a pointer to the thread_main which is the pointer to that function, second arg = argument to pass to thread_main
2. Run thread_main
 - a. Defines the work you want the thread do
 - b. Thread exits explicitly when it returns from its thread_main function
 - i. Thread is destroyed by OS when it returns to the main()
 - ii. To simulate this: begin by running thread_main directly
 - c. Will take argument that is a pointer to an arbitrary type
3. Then, your thread_main function can return to the stub function, should it return.

Context and calling conventions

- 4 fields in context structure

1. Stack pointer
 2. PC
 3. 2 argument regs
- Use var initialized through getcontext call
 - When a procedure executes
 - Allocate stack space by moving the SP pointer **down**
 - Can find local var, para, return addresses, old FP %rbp by relative to the frame pointer (%rbp)
 - When function needs to make a function call
 1. Copy argument of the callee function (function to be called)
 2. Put to the reg on right
 3. Push callee function parameter
 4. Push caller function current IP %rip (the return address)
 - IP → callee function address, callee starts ---
 5. Push the old FP (caller's) %rbp
 6. Set frame pointer = current pointer : current %rbp points to cell that stores old %rbp
 7. Callee push more local var to stack, %rsp continues moving down
 8. To return, %rsp = %rbp
 9. Pop old %rbp, assign to current %rbp
 10. Use ret to pop old IP off and assign (with leaveq, retq) to %rip
 11. Control is passed back to caller
 - E.g see ipad notes



- Space between %rbp (frame pointer) and %sp (current stack pointer) for local variables, saving/spilling other reg
 1. Push
 2. Mov

3. Sub

- Callee locates its var, para and return add using add related to %rbp
- Return to caller:
 - Assign %rbp to %rsp
 - Release current frame

Void thread_init(void):

- Initialization
- Custom create the first user thread in the system
 1. Setup kernel thread that is running with when your program begins (before any calls to thread create) as first user thread tid=0
 2. No need to allocate stack
 - a. Will run on user stack allocated for this thread by os

Tid thread_id():

- Returns thread identifier of currently running
- Between 0 - THREAD_MAX_THREADS

Tid thread_yield(Tid tid):

- Suspends the caller
- Activates the thread given by tid
- Caller is put to the ready queue, will later be run
 - Add caller to the tail
- Tid is the identifier of ANY available thread, or the constants:
 - THREAD_ANY: tells thread system to run on ANY thread in the ready queue
 - Run thread at the head
 - FIFP
 - THREAD_SELF
 - Continue the execution of caller
 - Implement as a no-op, may be useful to explicitly switch to current thread for debugging
- Return: identifier of the thread that took control AS A RESULT OF this function call
- Caller cannot see result until it is its turn to run
- Function may fail
 - Caller resumes immediately and indicate reason for failure
 - THREAD_INVALID: tid is not a valid thread
 - THREAD_NONE: there is no more threads to run (if the call is set to THREAD_ANY)

Tid thread_create(void (*fn)(void *), void *arg)

- Creates a thread with starting point at fn
- Arg is a pointer that will be passed to fn when thread starts executing
- Created thread is put on a ready queue, not start execution
- Caller continues to execute after function returns
 - Success: Returns a Tid
 - Fail: return a value:
 - THREAD_NOMORE: alerts the caller the thread package cannot allocate memory to create stack of a desired size

void thread_exit():

- Current thread does not run after this thread
 - Function should never return
- If no other thread are invoking this anymore, program should exit
- Only after the Tid of this thread is destroyed, another thread can use it

Tid thread_kill(Tid tid):

- Kills another thread with tid as identify
 - Killed thread should not run any further
 - calling thread continues to execute
- If success: return tid of thread just killed
- If fail: return THREAD_INVALID

Solution requirements:

- First thread: tid = 0
- Max thread tid = THREAD_MAX_THREADS - 1
- Thread exits: can be reused
- **Library should have a thread control block for each thread running in system**
- Maintain a queue of threads that are ready to run
 - when current thread tueds, next thread in ready queue can run
 - Allows THREAD_MAX_THREADS threads to use, so allocate the structure globally + statically
- Each thread's stack size \geq THREAD_MIN_STACK size
- Must not statically allocate these stacks at initialization time
 - Dynamically allocate using malloc() When a new thread is created
 - Delete when a thread is destroyed
- must use getcontext and setcontext to save and restore thread context state

L1 intro to System Software

What is system software

- Software used by multiple programs
- Software -- close \rightarrow hw and manages hw resources

What does SS concern of

- Performance
- Fairness
- Scalability
- Reliability
- Security
- Fairness
- API abstraction

Some example of SS

- OS
- Web services
- Cloud

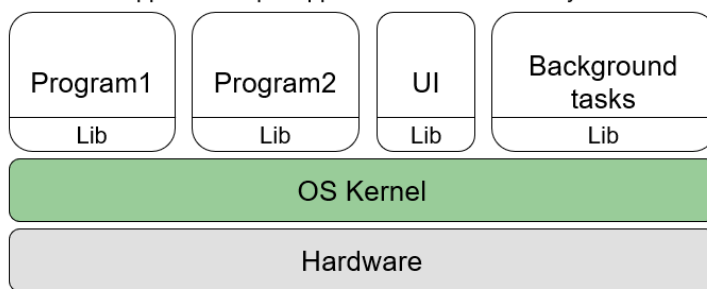
What is an OS?

- A layer of software between hw and application

What does OS serve

- All application
- Supports multiple app simultaneously
- Since each app is written in the belief of "I have entire access to hw", OS needs to go and make sure the apps are managed properly

An overview of OS:



What does OS support? What does OS include

- OS distribution: kernel + library bundled with it
- In this course, simply call kernel OS

What does OS do?

1. Manages hw resources in the way that
 - 1) allowing pro to interact with hw devices
 - Example hardware devices:
 - CPU, memory, disk, graphics card, co-processors, etc., on desktops, phones, routers ...
 - 2) And provides simple interface to devices
 - Access disk as "file"
 - If programs can directly write to disk, then when you change the manufacture of a disk, then your program may not work
 - OS makes it simpler for a program to access hardware. For example, an application that needs to access data from a hard drive can ask the OS to do so, rather than programming the drive hardware directly.
 - 3) Allow multiple programs to
 - run at the same time for them to share CPU, memory...
 - Isolate app from each other
 - so that 1 program does not go and write info of another
 - First, it ensures that multiple programs that are running concurrently (or at the same time) can share hardware resources safely. For example, it ensures that an application, say App1, cannot read or write another application, such as App2's, memory. Otherwise, App1 could steal data or crash App2. Similarly, the OS ensures that applications can access and share devices safely. For example, that when one application is sending a packet over the network interface, then other application cannot overwrite or read that packet.

-
- Isolate OS from other apps
 - So app cannot corrupt kernel. For example, if there is a change in pointer in kernel, the system can be ruined, and OS can no longer insure isolation to applications.

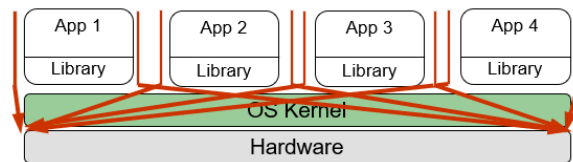
L2

What are the key concepts in the design of OS?

- Virtualization
 - To partition resources
- Abstraction
 - Provided via system call

What does Virtualization do?

- Provides illusion that: "I have entire access to hw", to the apps
- With 1 phy machine, OS gives multiple virtual machines so the pro think they have their own VM



What is the benefit of virtualization?

- Since each program is not aware of each other
 1. Achieve isolation amongst programs
 2. Programs can now be written independently
 3. Don't need to worry about accidental overwrite from one program to another program's memory/files and causing it to crash
 4. Achieve performance isolation so if a pro uses too much memory, it is the only one that has its performance degraded
- Since each program thinks it has a virtual machine
 1. The programs can be written portably regardless of amount of phy resources available
 - a. E.g. writing a program with 100 cores in mind can still be run in a 2 cores system
 - b. Any networking interface will work for the pro as OS provides an ideal and abstract env

How OS implements virtualization?

- OS provides the illusion of ... using ...
 1. Virtualize physical CPU as threads
 - a. Large number, are available to each program
 - b. Programs can use one or more threads to run their code concurrently, without worrying about the actual number of cores on the physical machine
 2. Virtualize physical memory as virtual memory
 - a. pro has access to large amount of contiguous + private memory

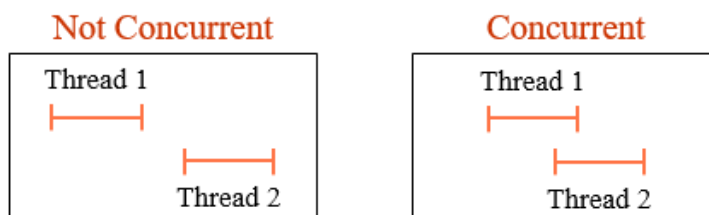
- b. The amount of virtual memory available depends, among other factors, on the number of bits in the processor architecture, e.g., 32 or 64 bit, as we will see later in the course. For example, on a 32 bit architecture, since 2^{32} bytes is 4GB, each program may have 4 GB of contiguous virtual memory available to it, even though the total amount of physical memory available on the system is 1 GB.
 - c. Also, the virtual memory available to each program may overlap. For example, each program may have access to all virtual memory addresses in the 0-4GB range. Thus when a program accesses memory, it doesn't have to worry about which other programs are running or what memory they are accessing, since the program has its own virtual memory.
3. Virtualize physical disk as files
 - a. Large number, are available to each program
 4. Virtualize physical network/interface as socket
 - a. allow programs to communicate remotely, without worrying about other programs that may be communicating remotely at the same time, while hiding the details of the network protocols and layers.

Difference between a PHY machine and a Virtual machine?

- A physical machine consists of:
 - Processor
 - memory (or DRAM),
 - disk and a network card
- A virtual machine:
 - Threads
 - virtual memory
 - Files
 - sockets.
- For each hardware resource, the OS provides a corresponding virtual resource. With the virtual resources, the program has the illusion that it has full access to the corresponding hardware resource.
- With virtualization, you can write a program that uses 10 threads and run it on a machine with 2 cores. You also don't need to worry about what other programs will be running on the machine.

What is difference between concurrency and parallel?

- Concurrent: virtually overlapping in time



- Parallel: physically running at the same time. Different cores run instructions streams in parallel
- Concurrent tasks can only run in parallel when: multi-core processors are available

What can threads do?

- Speed up app tasks by splitting 1 request into multiple threads and run concurrent requests in parallel using multi core
- Consider a web server that serves web requests. Modern web servers running at large websites need to serve 10000s of requests per second. To use all the cores on a machine, the web server can be designed so that each web request is served by a separate thread. These threads can then be run on different cores. This will allow serving concurrent requests in parallel using the multiple cores on a machine, which will speed up the webservice.
- Code Example:
- Run threads-race.c with args 100, 1000, 10000, 100000
- Run threads-sync.c
-
- Notice that the results for threads-race.c seem correct for small value of args. Why do you think that happens? Answer: when the value of args is small, a thread runs to completion before the next thread starts running, so the computation works correctly. When the value of args is large, the OS runs each thread for a short period, and then runs another thread, in a round-robin manner. This switching of threads induces the problem, as we will see later. Alternatively, if both threads run in parallel on two different cores, the same problem can occur.

What does concurrent threads require?

- Synchronization. Otherwise may generate error or crash.
- Synchronization uses a coordination mechanism, which is a mech that allows only 1 thread to access 1 queue at a time

What needs Synchronization?

1. Concurrent threads
2. OS because it runs programs concur/in parallel
 - a. For example, two programs may be trying to access the network card at the same time and so these programs need to be synchronized to avoid corrupted network packets.

How does OS provide abstraction?

- Through system calls

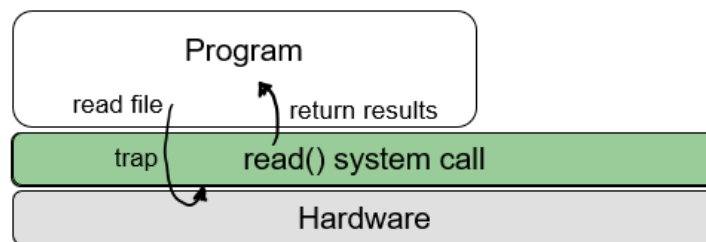
What are system calls?

- An APP request service provided by OS
- OS functions that provide portable access to hw and other OS functions. How do programs use system calls?
- A set of system calls: OS application programming interface, API. enables app to access all OS functions
- Example of OS API:
- creating and destroying threads
- allocating and deallocating memory from the system
- reading and writing files from disk.

how does program use API

- Program Invoke system call when they need to access the hardware
 - a. Because this makes it easy to write programs
- 1. A program function invokes a system call
- 2. A trap is issued, control is passed from program to OS

3. System call runs an OS function to completion
4. System call returns a result to the caller function
- Uses trap

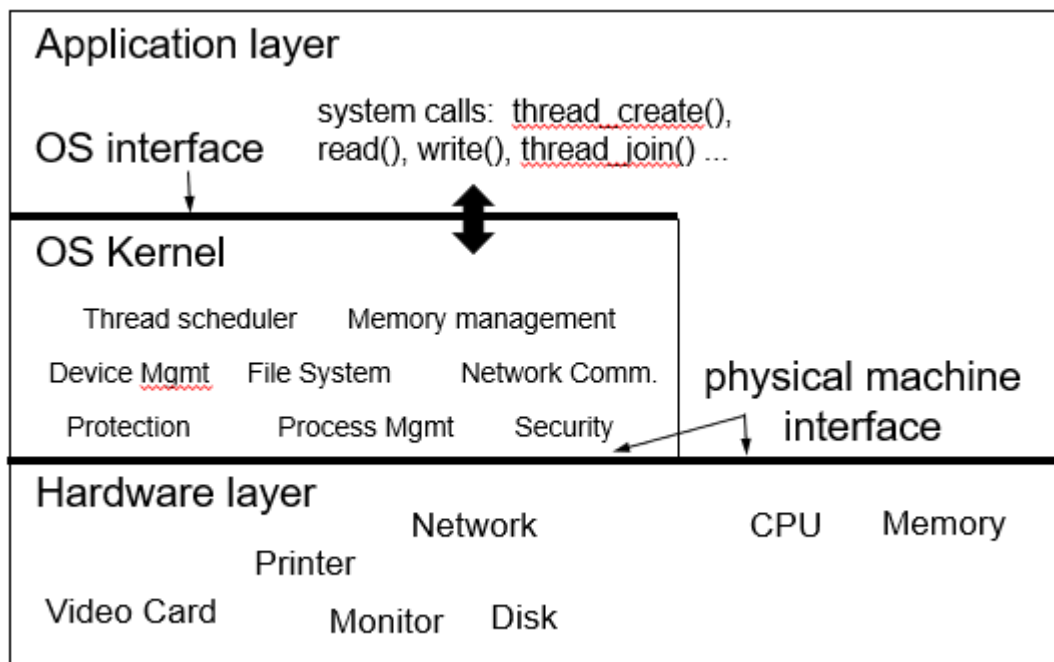


5.
 - Example:
 - Say a program needs to read a file from disk.
 - The program accesses the read() system call in the operating system by invoking a special instruction called a **trap instruction that transfers execution from the program to the operating system.**
 - The operating system then runs the read() system call. The read() system call access the file from disk and then it returns the contents of the file to the program, which then restarts execution.

Benefits of system calls?

- To summarize the benefits of system calls, consider the file read system call. A program can use this system call to read a file, without worrying about the details of the disk hardware. As long as the operating system supports a particular type of disk hardware, the program will be able to read from that disk.

OS interface



- Hardware layer: all hardwares
- Physical machine interface: CPU instruction set that enables assessing the hardware
- OS kernel (OS system):

- Uss PHY machine interface to access hw
- Provides VM abstractions like threads, VM, files, socket to applications
- OS interface: system calls that allows OS abstraction to access HW

L2 Think Time

1. Difference between printf (library call) and write (system call)
 - printf is a part of app
 - write is a part of OS
 - From app point of view, both calls are like function calls and give returns
 - But different: they are invoked differently. System call is invoked using a trap instruction, but printf
 - When printf function is used to write to terminal, when the program calls printf, it directly calls write
2. What is an ideal VM?
 - a. Behaves exactly lie a phy machine
 - b. Program cannot detect another program's existence
 - c. Program cannot access the OS memory
 - But hard to achieve, programs can realize this by seeing: not at full speed, not have access to all memory

L3

What is a processor, also called CPU?

- CPU executes a set of instruction. The instructions are used to Load data from/store data into memory. To perform this, various CPU registers are used
- Instructions can be different for diff CPU arch

What are the CPU regs?

- 8-64 range general purpose reg, temp hold val/var of a program
1. PC: program counter
 - a. Address of next Instruction
 2. IR: instruction register
 - a. Holds Instruction being exe NOW
 3. Reg: General reg
 - a. Holds variables, 8-64
 4. SP: Stack pointer
 - a. Hold address for accessing stats
 - b. The stack has var/data associated with the exe of a function
 5. SR: status registers
 - a. Holds control bits that affect program exe
 - b. status register holds various control bits that affect program execution. For example, it holds the overflow bit that is set when arithmetic operations overflow, and its contents are used to decide whether a conditional jump should be taken. Later, we will see that the status register also contains various bits that are important for implementing operating system functionality.

How is CPU executed?

```
PC = <start address>;  
// fetch-decode-execute loop  
while (halt flag is not set) {  
    IR = memory[PC]; // fetch next instn. from mem  
    PC = PC + 1;  
    execute(IR); // decode & execute instruction  
                // uses registers, stack pointer,  
                // status register, etc.  
}
```

-
- 1. Computer boots, it sets a PC (pre-det start add)
- 2. Then IR is assigned as the instruction stored at add PC
- 3. PC increment to access next instruct
- 4. CPU execute current IR instruction
 - a. SP, SR, Reg are used in here

How does memory(DRAM) provide storage for programs?

- Think of it as arr of bytes, each **byte** as a unique address

7	12	byte
6	150	
5	34	
4	0	
3	6	
2	4	
1	3	
0	5	

ddress value

What is address width?

- # of bits used to represent a memory's ADDRESS
- 32 bits: 2^{32} bytes, 64 bits...

How to access memory?

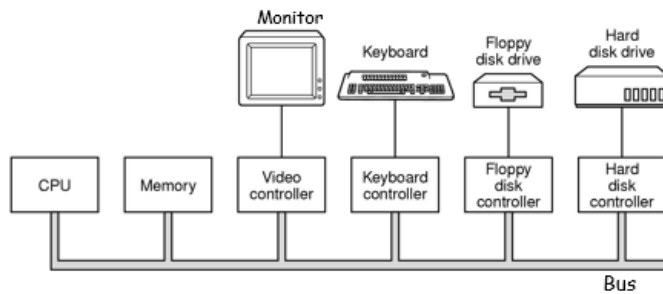
- Using a read/write abstraction
- *write(address,value)* write a value to a given address
- *read(address)* reads the value that was LAST WRITTEN at a given address using a load instruction

How are computer connected to IO devices?

- Via device specific controllers

How is CPU connected to memory and device specific controllers, and how do they communicate?

- Thru buses



-
- Each controller: certain range of bus address
- CPU send message to shared bus with an address, and message will sent to device with the shared address range
- Device sleep and wait until they receive messages

2 ways for OS to access devices

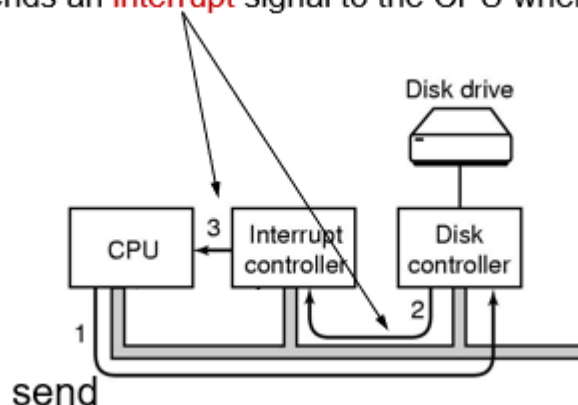
1. CPU give IO instruction: read and write from device reg in device controller to OS
2. CPU support memory mapped IO, so device reg are mapped to a memory location, OS can load/store instructions to these mem location, then these are routed by hw to read and write directly from device reg
 - a. This can lead to holes in phy mem, because there need to be add to store devices

Example of CPU, OS communication

- OS wants hard drive controller, it wants to read a file

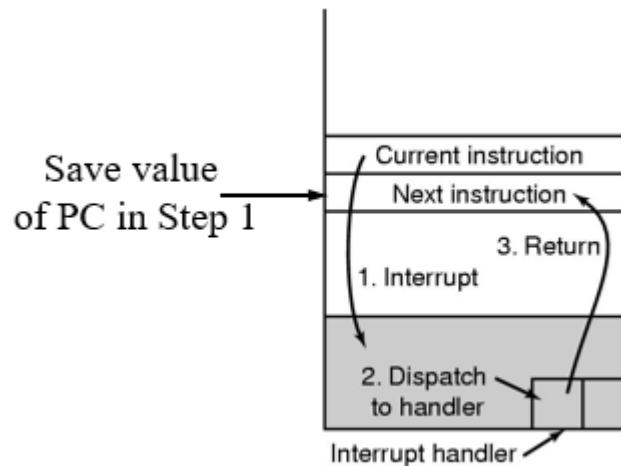
 1. OS use IO instructions (W1) or memory-mapped IO (W2) to write an add associated with the hard drive controller (with val of file location)
 2. Driver controller gets this message, start reading from file
 3. OS reads file data that is now in driver controller.
 - a. Issue IO instructions (W1) or memory-mapped IO (W2) to read from driver controller and check if the file is ready (check in loop, called polling)
 - b. Once available, issue read
 - c. But polling is inefficient!
 4. Instead of 3, use interrupts:
 - a. CPU uses a flag, called "interrupt request". Can be set by devices
 - b. Once the device is ready (driver has all data)
 - c. Set interrupt request flag on CPU, indicating ready!
 - d. CPU reads file from controller, so CPU and device can operate concurrently

e.



How processor execute with interrupt?

1. CPU saves current PC
2. CPU PC points to the interrupt handler function in the OS
 - a. Interrupt handler code communicates with device,
 - b. Get data
 - c. Return to the application
3. Program counter is set to the previous PC again



i.

Where is interrupt generated?

- By hardware external to CPU, thus the interrupt function can occur when the CPU is running other app

What is special about IH?

- Program is unaware that an interrupt has occurred/run

Processor exe with interrupt code

Hardware	<pre> PC = <start address>; while (halt flag is not set) { IR = memory[PC]; // fetch next <u>instn.</u> from mem PC = PC + 1; execute(IR); if (<u>InterruptRequest</u>) { hardware saves PC, SP, SR; PC = 0xIH; // h/w sets PC to address 0xIH, // contains code of intr. handler } } </pre>	} Step 1
OS Software	<pre> <u>Interrupt handler()</u> { <u>save_processor_state()</u>; // saves most CPU registers <u>handle_interrupt()</u>; <u>restore_processor_state()</u>; // restores registers <u>return from interrupt;</u> // restores saved PC, SP, SR } </pre>	} Step 2 } Step 3

-
- 1. CPU after one execution, check if the interrupt request flag is up
 - If yes, save pointer, set PC to add of IH
- 2. CPU next execution is actually executing the interrupt

- a. Store, handle, restore
- Return from interrupt
 - Restore PC, SP, SR, so next CPU iter will run on the previous next instruction

Why is save and restore needed in Interrupt handler?

- Because the handle_interrupt function can use and rewrite the reg values

L3 Think time

1. What program variables are stored on the stack?
 - a. Local variable
 - b. function arguments and return values
 - Global variables and heap var are not stored on stack
2. What is memory-mapped IO?
 - a. A location in memory that can be loaded and stored by OS to communicate with device
3. Is the device communication model similar to the memory abstraction?
 - a. The device communication model involves reading and writing to device
 - i. Done with help of memory mapped IO
 - ii. Much more applicated
 - iii. Many bugs in OS are caused by bugs in device controller/driver
 1. Device driver is the code in OS that handles interaction with devices and their controller
 - b. Memory abstraction is reading and writing to memory using API
 - i. Such prob don't happen to memory
 - ii. A load is guaranteed to return last value stored at that address
4. Should an OS ever use polling instead of interrupts?
 - a. Interrupts enable concurrent CPU and device running
 - b. When to use polling? With high speed devices
 - i. Because interrupt handling is expensive
 - ii. If high speed device, we need lots of interrupt
 - c. Polling a device tends to be cheaper in this case, and so it may make sense for the OS to poll the device whenever convenient. For example, the OS may poll the device at a fixed frequency. The tradeoff is that higher frequency polling is more expensive, but reduces the latency for handling the device event and is less likely to lose data.
5. Can you think of a real world analogy when human use polling versus interrupts?
6. In the processor execution code with interrupts, why are some registers saved by hardware, while others are saved by software?
 - PC, SP, SR are saved by hardware
 - Other are saved by OS, and if no need to reload, OS can just overwrite them
 - a. the code shows that the PC, SP and the SR are saved by hardware, while the rest of the registers are saved by the interrupt handler in save processor state.
 - b. The reason is that the hardware needs to save the PC, SP, SR to ensure that the interrupt handler code in software can start running correctly and it can restore these values when it returns.

- c. The rest of the registers do not need to be saved by hardware. For example, the hardware can run a null (empty) interrupt handler function in the OS by saving these three register values.
- d. The reason that the interrupt handler saves the rest of the registers is that it may overwrite them when it runs the handle_interrupt function. However, if the OS knows that the handle_interrupt function doesn't use of these registers, then it doesn't need to save and restore them, which makes the interrupt handling more efficient.
- e. For example, the Linux OS does not use floating point code, and so it does not save and restore the floating point registers in its interrupt handling code.

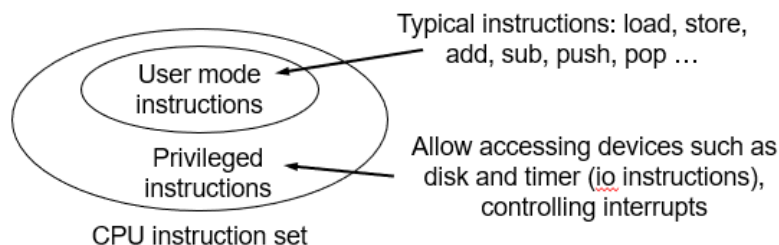
L4

What are the three hardware features used by OS?

- CPU mode (vir)
- Memory management unit (vir)
- Trap instruction (abs)

What two modes does the CPU have?

1. Kernel mode
 - a. User mode + control to devices and interrupts
2. User mode
 - a. Its instruction only has affect on itself
 - b. Cannot access hardware which allows protection to the hardware

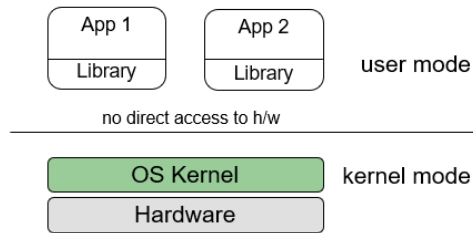


What are Instructions can only run in kernel mode not in user mode called?

- Privilege instructions : control to devices and interrupts
 - Have impact on the entire machine
 - For example, these instructions include IO instructions that allow accessing devices, such as the hard drive, network card, or the timer. Thus changing the time or the clock on the machine, which impacts all programs, can only be done via a privileged instruction. Privileged instructions also allow controlling interrupt behavior, such as disabling or enabling interrupts.

Which mode does OS run in?

- Kernel mode
- It has full privileges, access to all memory and devices



How does OS operation work?

1. Hardware boots up, it starts running in a bootstrapping program bios (a tiny OS) in **kernel mode**
 2. Bios loads main OS executable (kernel image) from hard-drive → memory, then transfer control to OS and execute in **kernel mode**
 3. When OS tries to start an app, it **switches to user mode**, run application mode
 - a. Applications can ONLY be run in user mode
 - b. No direct access to devices, limited access to memory
 4. When a program needs to access a device, it call OS, then access device in **kernel mode**
- Note: OS codes in program library runs in **user mode**
 - It only switches to kernel mode when there is a contact switch

Why can't programs be run in kernel mode?

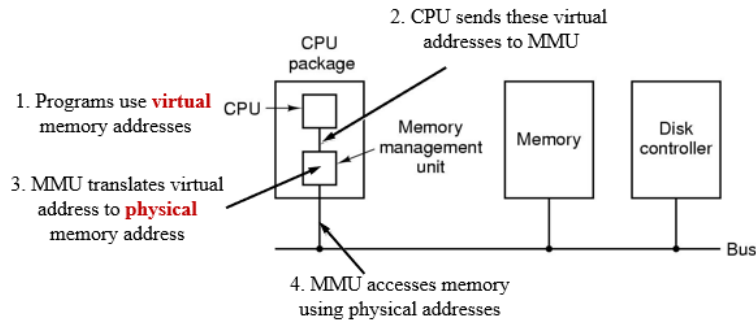
- Since the OS runs in kernel mode and has access to all CPU instructions, including privileged instructions, the OS is a privileged, trusted program.
- Thus it cannot have bugs or vulnerabilities, or else an application may be able to compromise the OS, run with kernel privileges, and bypass the hardware virtualization and abstraction provided by the OS.
- Another way to understand why the OS is trusted is that correct system operation depends on correct OS design and implementation, but not on correct user programs. The OS ensures that a buggy or vulnerable application will only affect itself, but not other applications, or the OS.

What is a Memory management unit MMU?

- A feature used by the OS to provide the illusion that each program is running on a virtual machine with lot of contiguous memory, and memory ranges of different programs running @ same time can be overlapping
 - 2 program can get diff val on virtual mem at the same time
- MMU is part of a CPU package
- When debug, print memory address in program, seeing virtual memory add

Does CPU instruction on a program execute on PHY or virtual memory location?

- Virtual
- These memory address are sent to MMU
- MMU then translate them to PHY memory address

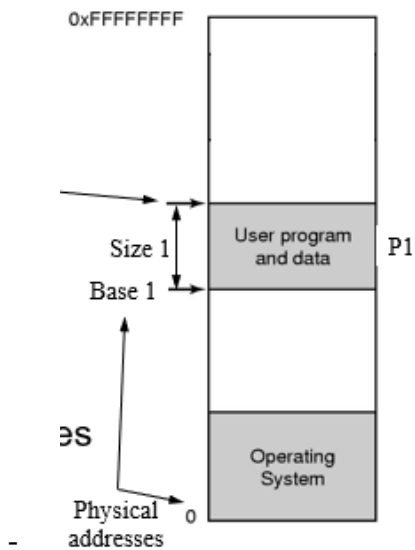


What does MMU act like

- A mapping between virt mem to phy mem
- And this allows OS to give each application its own private memory. MMU will translate them to different PHY memory

How MMU maps add?

- Input, for a program PA
- Base register = physical address of base A
- limit register = physical address of base A + physical size A
 - Ensures the memory is only within this range
- Virtual memory x in A = physical memory base A + x
 - Last byte in program = physical address of base A + physical size A - 1



What about 2 pro running concurrently?

1. For program A, set limit A and base A ^
2. When program B STARTS, a **different, non-overlapping** range of phy mem add is loaded as base B and limit B
3. Any time program wants to access virtual address x, the MMU reg val are changed to the coor base and limit val

How does OS + MMU help with memory isolation?

- OS:
 - ensures phy mem of which program are loaded and disjoint

- Ensure base B does not overlap with limit A
- MMU:
 - Ensure A cannot access memory above limit A
 - B cannot get lower than base B
- OS also make sure itself is loaded at a memory with no overlap of location with other

L4 think time

What if a program tries to cheat?

1. Why can't a program access a device directly?
 - a. Because it is run in the user mode and does not have the privilege IO instruction.
 - b. OS also ensure device registers are mapped to its memory that is not visible for other programs
 - c. Recall that programs access devices using IO instructions, or by using memory-mapped IO. IO instructions are privileged and the OS runs programs in user mode, so programs cannot use IO instructions to access devices. For memory-mapped IO, the OS ensures that device registers are mapped to OS memory. Since the OS uses the MMU to provide each program its own private memory, programs cannot access any OS memory, including the memory regions that map to the device registers. Thus, with the two hardware features, CPU modes and memory-mapped IO, the OS sandboxes programs, ensuring that they simply cannot access devices directly. This property that programs cannot access hardware directly ensure that the OS implements virtualization correctly.
2. For a base-limit MMU, what happens if a program accesses memory outside its base-limit range?
 - a. The MMU will check if the memory is outside of the range. If yes, it will generate an exception
3. What stops a program from modifying the OS so that the OS runs user code in kernel mode?
 - a. For example, Disabling interrupts is a privileged instruction. For example, a program can't disable the timer interrupt.
4. What stops a program from changing the MMU registers?
 - a. Program does not have privilege to set the MMU, that is something OS only
5. What happens if a program sends a bogus or malicious argument to a system call?
 - a. OS uses MMU to make sure programs cannot see and modify OS code
6. Operating systems provide administrator/root accounts that have full machine access. Do programs running with root privileges run in kernel mode?
 - a. root privileges in kernel mode: no, all programs, even root owned programs, run in user mode. these programs have higher privileges, such as being able to access the files or other users because the OS implements this behavior as part of the system call code, as we will see later.
7. How does virtual to physical address translation work when the OS runs two programs in parallel on two cores?
 - a. Each CPU core has its own MMU
 - b. MMUs perform translation independently

- c. OS sets up MMU reg for program running on that core
- 8. If programs can only access virtual memory then how does the OS access all physical memory?
 - a. Set its virtual memory = range of phy memory
 - b. os access all physical memory: this depends on the CPU and MMU. However, consider the base and limit register MMU. With that MMU, the OS can setup the base register value to 0 and the limit register value to the maximum physical memory size whenever it runs (on an interrupt or on a system call). In that case, the virtual address the OS generates will be the SAME as the physical memory address (size base = 0), and the OS will be able to access all physical memory while using virtual addresses!
- 9. Would it be possible to implement OS functionality, i.e., virtualization and abstraction, without hardware support?
 - a. implement OS functionality without h/w support: one could have an interpreter/simulator that interprets every CPU instruction and provides h/w virtualization and abstraction, but this would be very slow since each CPU instruction would need to be interpreted in software. this would have similarities with the way valgrind works.

L5

How can a program access a device?

- Run OS code in **kernel mode** to access devices

Is OS visible for programs?

- No. OS code is not visible to the program bc of MMU, so programs cannot be called by a user, and user cannot switch MMU reg to kernel mode because that is a privileged instruction

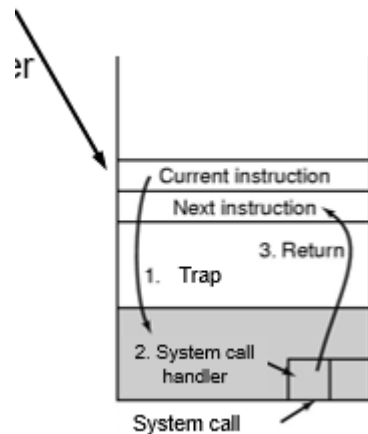
Why is trap instruction useful

- Due to previous two issues, trap instruction allows user-mode programs to switch to kernel mode and call a pre-def OS function
- Trap is generated **by the CPU** when **program issues a trap/syscall instruction**
- You can think of the trap instruction as a door that allows entering the OS room. but through a well-defined entrance.

How does trap instruction work?

- Similar to interrupt
- 1. Program issues a trap instruction
- 2. CPU save program counter, status reg
- 3. CPU **switches to kernel mode**
- 4. CPU runs system call handler code at a well-def location (determined by the CPU)
- 5. System call handler function in the software:
 - a. Save the rest of CPU reg and processor status
 - i. Do not want to overwrite these
 - b. Run a system call function
 - i. Such function provides a OS service
 - ii. Specified by users to access hardware
 - c. Restore the CPU registers

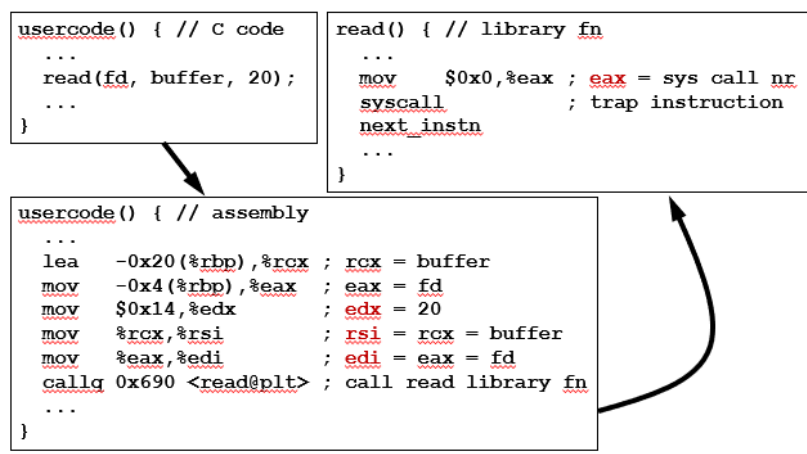
6. the system call handler function uses a special instruction to restore the saved status register and program counter values. This results in switching to user mode and returning control to the user code by executing the next instruction of the user program.



System calls relationship with trap?

- Trap are used by the programs to invoke the OS system calls
- OS functions provide an abstraction of hardware
- The operating system provides many system calls and the entire set of these calls is called the OS API since these are the only methods by which a program can access OS services. We have previously seen that examples of system calls include creating and destroying threads, allocating and deallocating memory from the system and reading and writing files from disk.

How are system calls invoked



1. User code function calls a library function `read()`
2. Library function will call a `read()` function in OS
 - a. `read(fd,buffer,20)` system call
 - i. `Fd`: file description, uniquely identifies a file. If this is `<0`, then an error occurs
 - ii. `Buffer`: of a particular size, usually 20, data read from file will be placed here
 - iii. These arguments are set up by usercode and read library functions

- Assembly part
 - Next, we show the assembly code that is generated for the usercode function. Normally, the OS chooses the calling convention, i.e., the registers that need to be loaded to pass arguments to the system call function in the kernel. In the case of the Linux operating system, the first parameter, fd, is passed in the edi register, the second parameter, buffer, is passed in the rsi register, and the third parameter, 20, is passed in the edx register. Finally, the usercode function calls the read library function.
 -
 - The read library function sets the eax register with the value 0, which is the system call number of the read system call. Then it issues the trap instruction on x86, which is called the syscall instruction. This instruction will invoke the kernel's system call handler. When the kernel finishes running the system call, the next instruction after the syscall instruction will run. Next, we will see what the kernel does in response to the syscall instruction.

OS code for system call:

```
OS_system_call_handler () { // OS code
...
save_processor_state();    // saves most CPU registers
system_call_table[eax](edi, rsi, edx);
restore_processor_state(); // restores registers
return from interrupt;    // restores saved PC, SP, SR
...
};
```

`system_call_table[0] = sys_read;`

Invokes: `sys_read(fd, buffer, 20)`

1. saves and restores CPU registers.
2. the `system_call_table[]` is an array of function pointers. Each function pointer points to a system call function, and the `eax` register value is used to decide which system call function to invoke
 - read library function had set `eax` to 0
 - Thus the first element of the system call table is a pointer to the read system call function.
3. The system call handler uses the `edi` (fn), `rsi` (buffer), `edx`(20) registers that had been setup by the **user mode** code to pass arguments to the read system call. Thus the handler eventually calls `sys_read(fd, buffer, 20)` to read the file into the buffer in **kernel mode**.
- Summarize:
 - When `read()` is called in the library, it sets the system call # to 0, this number is then used as the index for `system_call_table[]` and the three parameters are passed in which this function. Then, the read system call is invoked and reads file to buffer

What are the similarities and differences between traps, interrupts and exceptions?

	Interrupt	Trap	Exception
Cause	H/W external to CPU	Explicit instruction (trap instruction)	Instruction failure, e.g., divide by zero, bad memory access
Effect	None, program is unaware that interrupt, or interrupt handling occurred	Appears like program invoked (system call) function, and function returns data	Abnormal control flow
Timeliness	OS needs to respond to external event quickly	OS can take time to respond, since program is suspended	OS can take time to respond

Cause:

1. Differences:
 - Interrupt is generated by a hw device external to CPU
 - Trap is generated **by the CPU** when **program issues a trap/syscall instruction**
 - Exception is generated when the instruction fails to exe correctly, cn be generated by many instructions
 - For example, a divide instruction will generate an exception when dividing by zero. Similarly, the simple base-limit MMU will generate an exception when a program tries to access memory beyond the limit register.
2. similarities
 - For all three cases, when they are generated, CPU switches to **kernel mode**, and start executing corresponding OS system handler code
 - interrupt, system call or exception handler code.

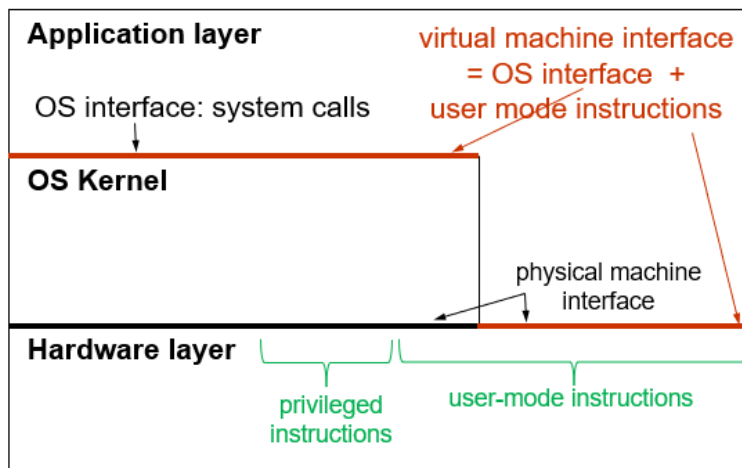
Effect:

1. Differences:
 - Interrupt: is transparent, thus program will not be aware of it
 - Trap: is issued by a program through trap/syscall instruction, it is like a system call **with no arguments**
 - Exception: OS exception handler kills the program that causes exception → crashes

Timeliness:

2. Differences:
 - Interrupt: OS needs to response to external event quickly as data is arriving from an external device. A delay may result in the loss od data
 - Trap: OS takes time to respond
 - For example, if the program is reading a file from disk, the OS can put the program to sleep, and run other programs, while the hard drive controller is reading the file from the drive. When the hard drive controller issues an interrupt indicating that the file is available, then the OS can read the file, wakeup the program and then return from the system call.
 - Exception: OS takes time to respond

A graph on OS interface



- OS interface consists of system calls invoked using trap instruction
- **Trap instruction is a user mode instruction**
 - Program running in user mode needs trap instruction to invoke kernel mode
- VM interface = OS interface + user-mode instructions on a subset of the physical machine interface
- OS uses phy machine interface, programs use VM interface

L5 think time

What if a program tries to cheat?

1. What happens if it invokes a privileged instruction directly?
 - a. The processor will no run instruction
 - b. Instead, an exception will be generated that causes CPU to switch to OS model call OS exception handler to kill the program
2. What if a running program doesn't make a system call to the OS and hence hogs the CPU?
 - a. thread doesn't make system call: OS uses a timer device to register a future timer interrupt before it hands control of the CPU to a user program. When the timer interrupt goes off, the OS gets control. Thus even if the program doesn't make a system call, the OS will get control periodically.
3. What stops the running program from disabling an interrupt?
 - a. Running program does not have privilege instructions, thus it cannot modify the interrupt handler or disable it
4. What happens if a program sends a bogus or malicious argument to a system call?
 - a. bogus argument to system call: it is possible that a program will send a bad argument to a system call.
 - b. For example, a program run by one user may try to read a file owned by another user or read a non-existing file.
 - c. Each system call checks each of its arguments for such cases and returns an error if the argument is problematic. similarly, the system call handler checks that the system call number lies within the system call table array.
5. How does the trap instruction provide a "secure" way to enter the kernel?
 - a. Trap jumps to a well defined location, this is the only entrance to the kerbal

- b. Thus at this entrance, kernel can check the system call number and argument to ensure only things are correct enter the kernel
- 6. Why is the OS not a normal program?
 - a. It runs with full privilege and can execute all instructions in the machine
 - b. Has access to the entire physical memory
 - c. OS only runs when a system call/interrupt occurs. One can enter OS through system call or interrupt in response to external events
 - d. Never terminates
 - e. OS has no thread of control, it can be invoked simultaneously
 - i. (e.g. two different system calls, or a system call & an interrupt).
- 7. How does the OS solve:
 - a. Time sharing the CPU among programs?
 - i. Timer interrupt to make sure all programs get to run
 - b. Space sharing memory among programs?
 - i. MMU to setup memory and make sure they are disjoint
- 8. Does library code (executing in user mode) provide isolation and abstraction?
 - a. It provides abstraction, but not isolation
 - b. Programs can jump to any instruction in the library code and rewriting them, they can bypass any isolation library code tries to provide
- 9. Does a virtual machine monitor (VMM) such as VMware provide isolation and abstraction?
 - a. Provides abstraction, and isolation
 - b. a VMM is a system program, similar to an OS, that allows running multiple operating systems simultaneously on a single physical machine. It provides isolation/virtualization, similar to an OS, but the same abstraction as a physical machine (each OS thinks it is running on physical hardware), thus providing no additional abstraction than the physical machine.
- 10. Put on your security hat: why can't user code execute some arbitrary code of its choosing in kernel mode?
 - a. Write instructions into kernel image - can't do that due to memory protection.
 - b. Transfer control to arbitrary places in kernel image to skip checks - can't do this due to memory protection, and control can only be transferred via TRAP to well known kernel entry locations.
 - c. Execute privileged instructions - can't do this in user mode.
- 11. The trap instruction switches to kernel mode and runs OS handler code atomically, i.e., both occur together. Why is that required?
 - a. Since no other user codes are allowed to run in the kernel mode in between
 - b. If handler code is run and then the kernel mode is switched, then the handler code cannot run bc it is in the user mode
- 12. What is the minimum number of privileged instructions that h/w must implement so that the OS can work correctly?
 - a. with memory mapped IO, you can hide all device accesses with memory protection. So 1) programming the MMU (i.e., modify MMU registers) should be privileged. Also, 2) returning from a trap (e.g., iret instruction) should ensure that we cannot somehow switch to running in kernel mode and run arbitrary kernel code. For example, on x86, a return from trap is guaranteed to execute code with the same or lower privilege level.

L6

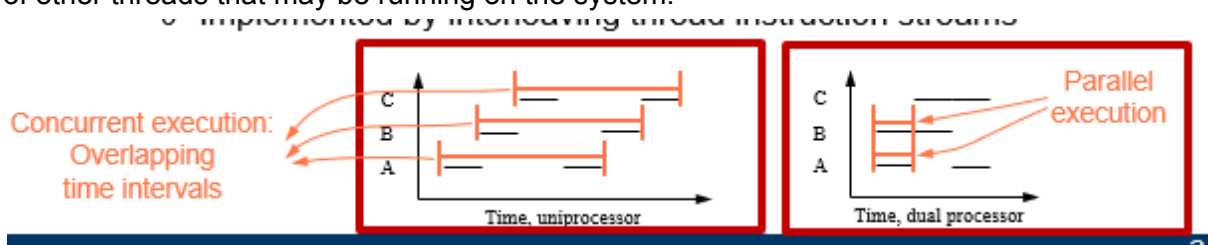
What is a thread?

- An OS abstraction for virtualizing the CPU
 - Provides illusion that there are arb number of CPU for each program



What does thread do?

- executes a stream of instructions.
- Each thread thinks that it has its own set of CPU registers, and it runs independently of other threads that may be running on the system.



How can OS run an arbitrarily large number of threads on a fixed number of CPUs?

1. multiplexing/interleaving
 - The OS implements threads by multiplexing/interleaving the instruction streams of threads on the available CPUs.
 - thread A runs for some time on the processor, then thread B runs, and then thread C runs. After that the three threads run again in the same order. Thus the execution of the three threads is interleaved in time on a single processor, and the three threads could continue running in this round-robin order. Notice that each thread is not aware that other threads are also running.
 - We say that the three threads run **concurrently** because they run in **overlapping** time intervals.
2. running in parallel
 - The figure on the bottom right shows a dual processor system.
 - In this case, threads A and B are running in parallel on the two CPUs for some time. After that A is stopped on one CPU and C is run on that CPU. Later, B is stopped on the other CPU, and A runs on that CPU. In this case, three threads are running concurrently but two threads run in parallel on the two CPUs at any given time.

How does the OS interleave the execution of threads on 1 CPU?

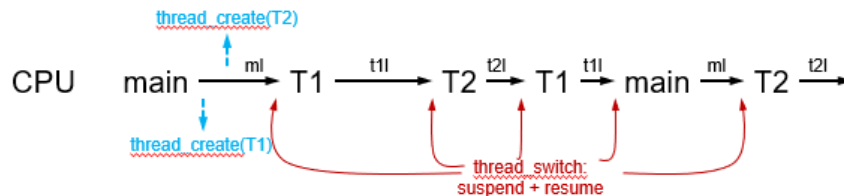
```

main() {
    thread_create(T1);
    thread_create(T2);
    main_loop();
}

T1() {
    t1_loop();
}

T2() {
    t2_loop();
}

```



-
- Main function has a single thread
- Creates additional threads using `thread_create()`
 - Thus in this case, total of 3 threads are created
 - Each having its own running loop
- The main thread first creates thread T1 and then it creates thread T2, and then runs its main loop, shown as ml. → After sometime, the OS decides to stop running the main thread on the CPU and start running thread T1. T1 starts running and runs its t1_loop. → Next, the OS decides to stop running thread T1 on the CPU and start running thread T2, which then runs its t2_loop. → Later, the OS decides to stop running T2 and restart running the T1 thread that it had stopped running previously. → Similarly, the OS later decides to resume main instead of T1 and then resume T2 instead of main on the CPU. Thus the OS runs the three threads concurrently by interleaving their execution on the CPU.
- Note that none of the threads are aware that other threads are running, and each thread runs its own set of functions independently of the rest of the threads.
 - For example, each loop function can call its own set of functions, without worrying about what functions another thread might be running.

How does OS implement the threads abstraction?

- by **switching threads**
- At each thread switch, the OS **suspends** the thread that is currently running on the CPU and **resumes** another thread on that CPU. Thus each thread runs for some time and then is suspended while other threads are running on the same processor. The OS can switch threads on a CPU at any time, without a thread being aware that it has stopped running or has restarted running.

Why doesn't the user realize that the program has stopped making progress?

- The operating system switches threads frequently, much faster than human perception or response time, and so even though on a CPU, the threads are being interleaved, we think they are running simultaneously.

How does the OS interleave the execution of threads run on 1+ CPU?

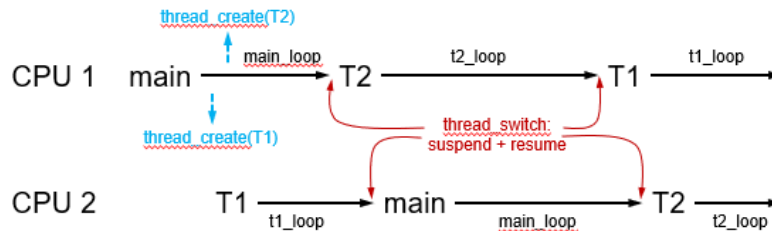
```

main() {
    thread create(T1);
    thread create(T2);
    main_loop();
}

T1() {
    t1_loop();
}

T2() {
    t2_loop();
}

```



-
- After the main thread starts running on CPU1, it creates the T1 and T2 threads and runs its main_loop.
- After the T1 thread is created, the OS can run the T1 thread on CPU2, which starts running its t1_loop computation in parallel with the main_loop (this starts a little behind CPU 1 because it takes time for T1 to be created)
- After sometime, the OS decides to stop running the main thread on CPU1 and start running thread T2. T2 then runs its t2_loop. At this point, T1 and T2 are running in parallel, while the main thread is stopped.
- Next, the OS decides to stop running thread T1 on CPU2 and restart running the main thread on CPU2, so now the main thread and T2 are running in parallel.
- Similarly, the OS later decides to run T1 instead of T2 on CPU1 and T2 instead of main on CPU2.
- In this way, the OS interleaves the execution of the three threads, allowing threads to run in parallel on the 2 CPUs.

Can a thread run on any CPU?

- yes, but it only runs on one CPU at a time.

Why can we say OS virtualizes CPU with thread abstraction?

- With the threads abstraction, the threads do not need to be aware of the number of physical CPUs, whether one or two or more, on the system.
- Each thread always thinks it is running on its own CPU

What is the relationship between multiplexing, concurrent and parallel?

- On any core, at most one thread can run at any one time. To give the impression that multiple threads are running on the same core, then the threads are multiplexed onto the core, by, for example, running thread t1 for a short while, then t2 for a short while, then t3 for a short while, then t1 again for a short time, and so on. → concurrent happens within each thread
- When you have multiple cores available, then typically such multiplexing occurs on each of the cores, but with say 4 cores, at any given time you have four threads running at the same time "in parallel".

L6 think time

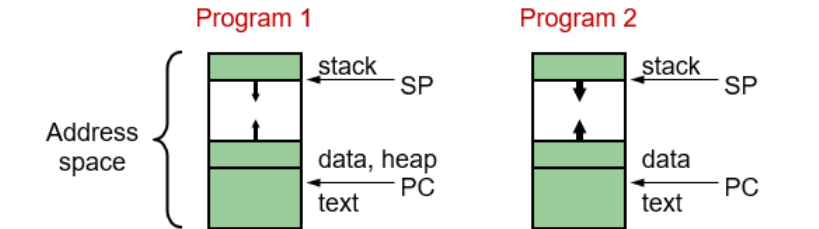
1. What is the difference between a thread and a function
 - a. A thread is a virtualized CPU, it is used to execute instructions in a function
 - b. It runs one or more functions on a single stack
 - c. When functions are run on the same thread, they cannot have time overlaps, meaning F2 can only run after F1 has completed.
 - d. Threads are independent streams of execution. They are NOT aware of each other
 - e. Thus the threads does not have to finish running before another starts
 - f. . Each thread calls its own set of functions and has its own stack.
2. The OS switches threads frequently on a CPU and so user think that all threads are running. Can you think of other ways that users are “fooled” by timing-based optical illusions in real life?
 - an optical illusion is a wheel that is spinning fast -- it may appear to spin in the opposite direction

L7

What is an address space?

- A set of **virtual** memory addresses that a program can access is called the address space of a program.

What are the regions in an address space?



1. Text region
 - a. Programs code that will be executed
 - b. A read only region
 - c. A PC points in this region because it contains the address of the next instruction that should be executed
2. Data region
 - a. Static and global program variables
 - b. Has a fixed size
3. Heap region
 - a. Contains dynamic variables, variables allocated at runtime
 - i. MALLOC
4. Stack region
 - a. For function calls
 - b. Stores function arg, local var, return val
 - i. Grows when function invoked, shrink when return
 - c. SP is on the top, and stack grows downward

Is address space shared among address space?

- No , Each running program has its own, private address space, and so different programs accessing the same virtual address will access their own code or data.
- For example, Program1's and Program2's text region could start at the same virtual address, but they would be accessing different instructions.
- This is true even if the same program is run multiple times. Each running instance of the program has its own address space. Thus the address space is the OS abstraction for virtualizing memory. We will discuss how the OS implements memory virtualization and address spaces in much more detail later in the course.

What is an OS processes?

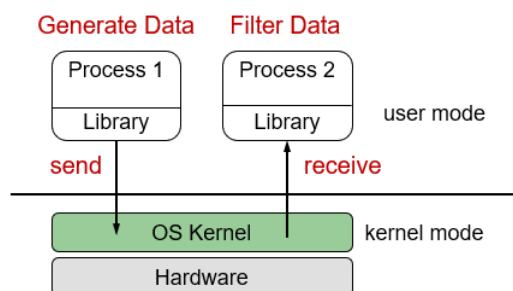
1. The lifecycle of a process starts from a program.
 - a. A program is an executable file on the hard drive
 - b. Such file has instructions and data that define how a computation should be performed.
 2. OS runs the program, it creates an address space into which it loads the instructions and the data from the program file.
 3. OS Then it creates a thread and associates it with the address space so that the thread can run instructions and access data from the address space.
- When 2 and 3 are both performed, we say that the OS has created a **process**.

What is a Traditional process?

- = an address space + one thread associated with it.
- The address space provides memory protection because each process has its own address space and thus two processes cannot access each other's memory.
- A thread enables running processes concurrently and in parallel on CPUs

How does a process-based communication work?

- When processes need to communicate with each other to perform some tasks together
 - For example, one process could be generating some data, while another process can be filtering this generated data.
- Two processes do NOT share memory, they CANNOT communicate with each other by reading and writing memory
 - Because address spaces are separated for each program
- They communicate through OS system calls



- For example, the figure shows two processes, one that generates data, and the other that filters this data. Since this data is not shared between the processes, Process1 needs to send this data to Process2.

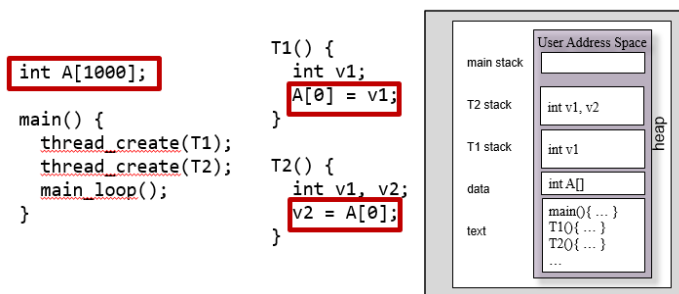
- A process sends data to another process by invoking a system call.
- The OS temporarily stores this data. (copy x1)
- The recipient process then invokes a system call to read this data from the operating system. (copy x2)
- This communication is similar to message-based communication used by clients and servers running on different machines to communicate with each other.

Problem and limit with system-call based communication with traditional processes?

- expensive because it requires two system calls and making two copies of data, once on a send, and once on a receive.
- If a program wants to speed up operation with 2 threads, then traditionally, 2 processes should be created, and the threads are communicated through expensive system calls

What is a Modern process?

- 1 process = 1 address space + 1 or more threads.
- Why is modern better than trad?
 - Since the threads within a process share the address space and thus memory, these threads can communicate with each other by reading and writing the shared memory, which is much more efficient than communicating with system calls.
- How is protection separated from concurrency with modern processes?
 - Previously, in order to get concurrency, one needed to have a separate **protection domain (separate address space)**.
 - With the new definition of a process, the process provides protection, but enables concurrency using threads within the protection domain.
- An example:



- The code on the left shows a program that will create one process with two threads T1 and T2.
- The figure on the right shows the address space when this program is run.
- The two threads:
 1. share all memory, including all code in the text region and the global array A in the data region.
 - a. the two threads can communicate with each other using a shared variable, which in this case is the global array A
 - b. T1 writes to an array element and T2 can read this array element
 2. However, each thread has its own private **stack** region for running its own functions.
 3. Thus T1 stores its local variable v1 in its stack region.

4. and similarly, T2 stores its local variables v1 and v2 in its stack region.

Why is modern process better?

- Unlike with processes, system calls and data copying are not required for threads to communicate with each other.

How to speed up vector operation?

```
For (k = 0; k < n; k++)  
    a[k] = b[k] * c[k] + d[k] * e[k];  
           t1      t2
```

1. Use 2 processes
 - 1) Process1 stores the arrays b and the c, multiplies them, and stores them in a temporary array, say t1.
 - 2) Process2 stores the arrays d and e, multiplies them, and stores them in a temporary array, say t2.
 - 3) Since t1 and t2 are on diff process, if storing array is on t1, then process 2 needs to send all data in t2 to t1 through system calls
 2. Use 2 threads in 1 process
 - 1) thread1 can compute the array t1
 - 2) thread2 can computer the array t2
 - 3) thread1 can add the two temporary arrays and assign them to the array a.
- This is possible because both threads share memory and thus they can both access the arrays a, t1 and t2.

Will #2 speed up the process when there is 1 CPU?

- No.
- In this case, both threads would be consuming the CPU (100%) and so speedup requires parallelism, which doesn't exist on a single processor, and so there would be no speedup.
- In fact, there would be a slight slowdown, because the OS will be frequently switching the two threads, which adds some overhead.

Will #2 speed up the process when there is 2 CPU?

- Yes
- in this case, the two threads will be running in parallel when computing the arrays t1 and t2, reducing the time for the vector operation compared to running the operation on a single CPU.

Summary:

- If only 1 CPU is available, separating into 2 thread that are both computational intensive will only slow down the program because switching takes time

How to speed up web server program

Run in loop:

1. get network message (URL) from client
2. get URL data from disk, cache in memory
3. compose response containing webpage
4. send response

- A webserver serves each client request in 4 steps:

1. Server receives a network message from the client containing the URL of the web page
2. Server uses the URL to retrieve the web page from disk and caches it in memory
3. Server composes a response containing the webpage
4. Server sends this response to the client

a. A single threaded web server:

- i. will run these 4 steps in a **loop**, once iteration for each request.
- ii. steps 1 and 4 access the network, and step 2 accesses the disk.
- iii. These are IO operations, and while they occur, the CPU is not used.

#1 Will using 2 processes speed up the program?

- We can run the same loop in each process.
- When a request arrives, we can send the request to one of the processes.

Problem with using 2 processes speed up the program?

- this approach is that the processes do not share memory and so they cannot share the cached web pages.
- For example, suppose a web page is served by process1, which then caches this page. Now, if another request arrives for the same web page, and process2 serves this second request, it will need to read the page from disk even though process1 has it cached in memory. After that, both processes will cache the same webpage, which also wastes memory.

#2 Will using 2 threads, 1 process speedup this program?

- run the same loop in each thread
- Good thing: this time the web page cache can be shared by the two threads.

Will #2 speed up the process when there is 1 CPU?

- Yes
- while one thread is waiting on IO in steps 1, 2 or 4, the other thread can run step 3 on the CPU.
- Thus, speedup is possible with a single CPU because threads allow exploiting concurrency between the CPU and IO devices.
- This is unlike the vector operation, in which both threads were only using the CPU, and so more than one CPU is required to get speedup.

Will #2 speed up the process when there is 1+ CPU?

- Yes
- in this case, the two CPUs will run the two threads in parallel, and so even if the two threads were both composing a response, they can be run in parallel.
- **Example: GOOGLE**
- Google had to decide how to handle that separation of tasks. They chose to run each browser window in Chrome as a separate process rather than a thread or many threads, as is common with other browsers. Doing that brought Google a number of benefits. Running each window as a process protects the overall application from bugs and glitches in the rendering engine and restricts access from each rendering engine process to others and to the rest of the system. Isolating a JavaScript program in a process prevents it from running away with too much CPU time and memory and making the entire browser non-responsive.

- Google made a calculated trade-off with the multi-processing design. Starting a new process for each browser window has a higher fixed cost in memory and resources than using threads. They were betting that their approach would end up with less memory bloat overall.
- Using processes instead of threads also provides better memory usage when memory gets low. An inactive window is treated as a lower priority by the operating system and becomes eligible to be swapped to disk when memory is needed for other processes. That helps keep the user-visible windows more responsive. If the windows were threaded, it would be more difficult to separate the used and unused memory as cleanly, wasting both memory and performance.

Difference between thread and process?

	Threads	Processes
Memory needed	Shared, less	Not shared, more
Communication & Synchronization	Via shared variables, faster	Via system calls, slower
Switching	Faster	Slower
Robustness	Memory sharing can cause hard-to-detect bugs	All communication is explicit, more robust program design

-
- In the first three cases, memory needed, communication and switching, threads have benefits over processes because they share memory.
- However, since threads share memory, a bug in one thread can easily effect another thread. For example, if one thread corrupts a shared variable, such as the global array shown in previous examples, other threads will see this corruption when they access this variable. With processes, memory is not shared, and so all communication between processes is via messages.
- A process can check for errors in other processes by carefully checking just the messages it receives, which leads to more robust programs.
- This is the reason that each tab in the Chrome browser is a separate process, as we mentioned in the first lecture of the course. This may make the Chrome browser slightly slower than a threads-based browser. However, in Chrome, if there is a bug or a vulnerability in a tab, it can only affect that tab, but not other tabs, making Chrome a more robust browser.

What are the 3 OS level states?

- Similar to struct in C
- What is a process control block PCB/ process state?
 - A state created by OS after it creates a process. It is used to maintains information in its memory for the various abstractions it provides to the process when a program is run
- 1. Thread state
 - a. for each thread associated with the process (for virtualizing the CPU)
 - b. Has thread ID to uniquely identify each thread
 - c. CPU registers/CPU state

- i. PC, SP, SR, a copy of the values of the CPU registers when a thread is suspended
 - d. Thread parameter
 - i. scheduling
- 2. Address space state (virtualizing memory)
 - a. MMU state
 - i. MMU register
 - ii. Also maintains the location of the address space regions, such as the text, data, heap, stack regions.
- 3. Device related state (virtualizing devices)
 - a. information related to files and network connections that are in use by a process.
 - i. Is a file opened
 - ii. Is something allowed to open the file

L7 think time

1. What is the difference between a program and a process?
 - a. Program = a set of instruction and data contained in a file
 - b. Process = program loaded in memory + executing
2. What is the difference between a thread and a process?
 - a. Process = address space + thread
3. What is the difference between an address space and a process?
 - a. Process = address space + thread
4. We saw that the web server threads running concurrently on a single CPU help hide IO latency. Have we seen this idea elsewhere?
 - a. Interrupt
 - b. CPU continues executing while a device operation is in process, so interrupt also helps hide IO latency
 - c. Difference: Interrupt is only for OS, threads are available for user-level programs
5. Suppose you wanted to write a single-threaded web server (that runs on a single CPU). Can you think of a way to hide IO latency?
 - a. single-threaded web server, hide IO latency: Without threads, the problem is that while a file is being read from disk (which is slow), or another response is being sent (which is also slow), another request that has arrived cannot be processed. One way to deal with this is to issue the file request or the network send/receive operation and then operate on other requests. Then, you can periodically check whether the pending requests have completed. This behavior is similar to using polling, instead of interrupts. This kind of programming is hard to get right, because one needs to build a state machine, that tracks the status of each request. Threads simplify all of this, because each thread implicitly maintains the state of a request.
6. Do you know how the Linux OS allows a program to create multiple processes?
 - a. when a program starts, the OS creates one process for the program.
 - b. After that, in Linux, a process can create additional processes using the fork() system call.
7. Does the OS code run in a separate process?

- No
 - OS code runs when a process makes a system call or when an interrupt occurs.
 - The OS generally runs in the thread and address space context of the user process that was executing when the system call or the interrupt occurred, and so it is not a separate process.
8. We saw that the heap grows up and the stack grows down in the process's address space. What if they collide?
 - a. If they collide, the program will run out of memory and crash. But will not affect any other programs
 9. Why do threads have their own, private stack regions?
 - a. To store local variables, call functions, return values as threads are independent from each other
 10. Can threads access the stack regions of other threads?
 - a. **Yes**
 - b. Because they share the same address space, so 1 thread can access another thread's address space, but usually this is avoided
 11. Does the OS have an address space?
 - a. Recall that all programs, including the OS, see virtual addresses.
 - b. So the OS also has a virtual address space.
 - c. The OS typically maps the entire physical memory to its virtual address space so that it can access physical memory, including the memory of programs. This is how it can read system call parameters that are passed by reference. For example, the buffer variable in the read system call is passed via a **pointer**. The OS needs to access this buffer to write to it (the data that is read from the file). By mapping the entire physical memory to its address space, the OS can access this buffer, wherever it is in physical memory.
 12. The OS maintains device state, such as open files, network connections, for each process. What other device state must the OS keep?
 - a. device state such as terminal state, display/graphics card state, essentially state for each physical device on the system that is being used by the process
 13. Is there any other state the OS needs to maintain for a process?
 - a. it will keep any data that a process has sent to another process that the other process hasn't received yet (see send/receive system calls).
 - b. the OS also keeps state related to pending signals, timers, swap, etc.
 14. Can a process modify the per-process state maintained by the OS?
 - a. No
 - b. Because if this happens, the user can modify the OS, and thus isolation will not happen anymore because users can access info of other users in this way
 15. Does the OS need a process structure?
 - a. NO
 - b. OS does not run as an independent process. It runs in thread and address space of user process when system call or interrupt occurs

L8

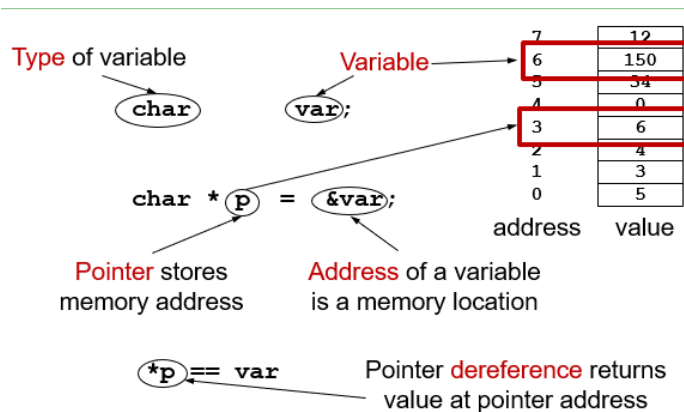
What is an address space of a program?

- the set of virtual memory addresses that a program can access
- each running program has its own, private address space, and so different programs accessing the same virtual address will access their own code or data.

What does an address space of a program contain?

- program code and data
- Divided into:
 - Text
 - contains the program code
 - Data
 - Global or static variables
 - Heap
 - Dynamic memory
 - Stack
 - Function calls, local var

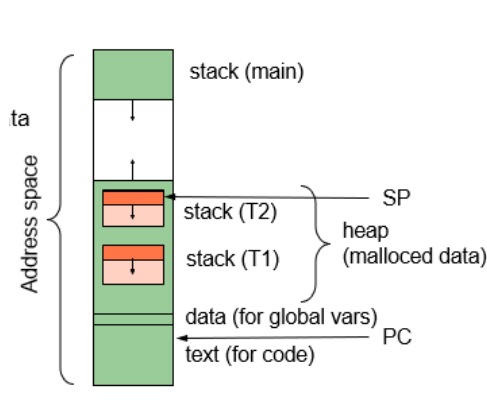
What is program data in data and stack region



1. Variable
 - a. All programming languages provide the notion of a variable.
 - b. is a symbolic name for some data stored at a memory location.
2. memory Address
 - a. The memory location of that variable
 - b. the address of a variable is obtained by using the ampersand operator. In this case, &var is the memory address at which variable var is located, and its value is 6.
3. Type
 - a. determines the size and alignment of the variable in memory
 - For example, an integer can be 4 or 8 bytes in size. In this case, a character variable is 1 byte long.
4. A pointer
 - a. a variable that stores a memory address
 - b. the pointer p is located at, say, memory address 3, and it stores the value 6, which is the location of the variable var.
 - c. When the pointer p is **dereferenced**, using *p:
 - i. it returns the value at the address stored in the pointer p

- ii. In this case, the pointer `p` stores the address of `var`, and so the pointer dereference returns the value of the variable `var`, which is 150.
- iii. `*p` as another name for the variable `var`.

Where is the data of a thread?

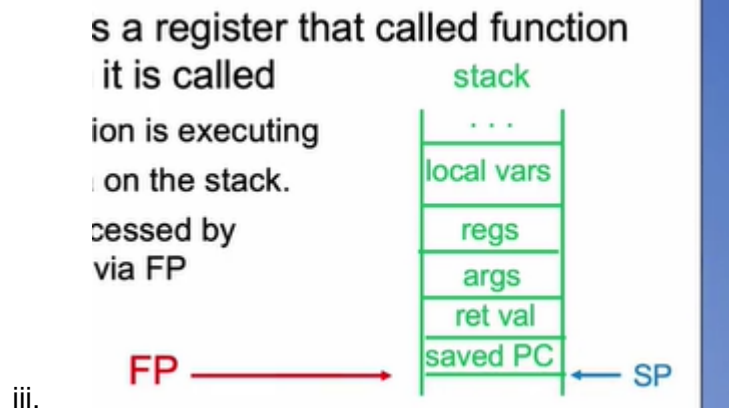


1. Program starts with the stack of the main stack
2. For each thread created, we allocate stack for it in the heap
 - a. In all cases, stacks grow down
- Data regions
 - global variables
 - Can be accessed by any thread
 - But this is not a good idea for any thread to access places other than its own stack
 - Because getting a pointer and want to refer to stacks that change dynamically may just make this pointer invalid
 - Read only
- Heap regions
- Stacks:
 - main stack and dynamic malloced stacks are used to save:
 - arguments to the functions
 - Function return values
 - Local vars
 - Temporary data
- Register
 - Can be full of data the thread need to execute
 - Ensure each thread is executed correctly

What happens when 1 function calls another function?

1. Saves some register, pushes them to the stack
 - a. Convention: callee will save all the reg
 - b. 3 options:
 - i. Caller saves all reg so callee can use them
 - ii. Callee save register
 - iii. Some saved by caller, rest save by callee
2. Places argument o registers, and push extra argument onto stack
3. Make space on stack for return
4. Calls function with processor call instruction

- a. On the call instruction
 - i. Pushes current PC onto stack
 1. To make sure the call instruction is executed, the code can continue to run
 - ii. Sets PC to address of first instruction of function to be called, so the next instruction to execute is the first instruction of the function



What does callee do when they get call?

1. Set FP onto stack
2. Let FP = SP
3. Saves some registers by pushing them on stack
4. Make space for local var
5. Execute functions

What is the use of a stack function:

- Constantly moving up and down on the function
 - Thus it is hard to refer to data on the stack
- Frame pointer FP/BP
 - Set = SP when the function is called
 - Making it easier to access data on the stack

L8 think time

1. Why do we need two pointers, the frame pointer and the stack pointer, to locate data on the stack?
 - a. the frame pointer register is used to track the data in the current activation frame, and to link the activation frames of the active functions. the stack pointer register is used to track the top of the stack. typically, all parameters, local variables and return values are accessed using the frame pointer. This pointer is only updated on function call and return. the stack pointer can be updated at any time within a function. for example, when a new local variable is allocated within a new scope, the stack pointer is updated. This pointer is also updated by the call and the return instructions.
2. You see the following instructions at the beginning of an x86 function: `push %rbp;` `mov %rsp, %rbp`. Similarly, you see the following instructions at the end of an x86 instruction: `mov %rbp, %rsp;` `pop %rbp`. Can you explain how these instructions:

3. Modify the two pointers above?
4. How they modify the activation frame?
- 5.
- 6.

L9 threads API

What are the two things required to implement threads?

1. Thread scheduling
 - a. Choosing which thread to run next, and when to run it
2. Thread switching
 - a. Suspend a running thread
 - b. Resume the next chosen thread

How is the order of running thread chosen?

- In orders based on scheduling policy
 - c. round-robin scheduling, in which every thread runs for a short period of time, in a round-robin order, ensuring fairness
 - d. priority-based scheduling, in which some threads run with higher priority than other threads.

What is a thread API?

- A thread scheduler implements a set of interface functions that define the threads API.
- thread_create, thread_exit, thread_yield, thread_sleep, and thread_wakeup
- Programs call these thread scheduling functions, using similar named system calls, when they need to use threads.

How does a thread scheduler implement its scheduling functions?

- By keeping track of the thread status
- It transfers threads from one status to another

What are the 4 values of thread status

- Ready
 - Can run, not running yet
 - Can have 1+ threads in this status
- Running
 - Using the cpu
 - the current thread.
- Blocked
 - No need to run any longer, but is not destroyed yet
- Exit
 - A thread may be blocked/waiting for other events
 - such as an input from the keyboard, a packet to arrive, etc.

Difference between a thread status and a thread state?

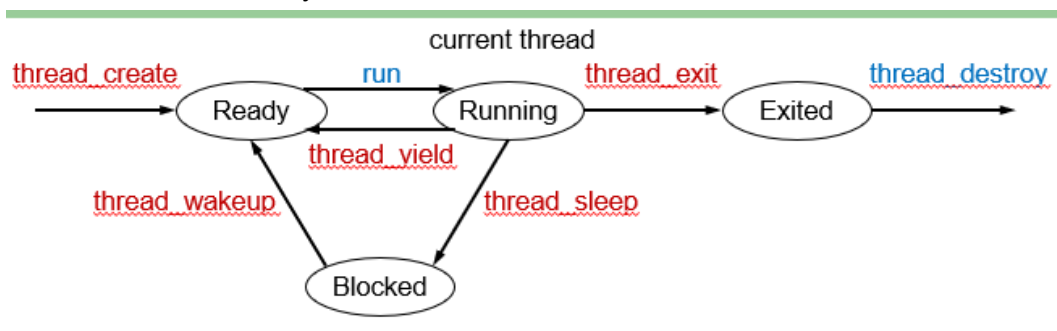
- Thread state:
 - for each thread associated with the process (for virtualizing the CPU)
 - Has thread ID to uniquely identify each thread
 - CPU registers/CPU state

- PC, SP, SR, a copy of the values of the CPU registers when a thread is suspended
- Thread parameter
 - scheduling
- Thread status:
 - 4 values keeping track of the current “mode” of the thread

What is difference between thread API and thread scheduler?

- That the scheduler runs another thread when the current thread calls one of the thread API functions.
- The scheduler functions:
 - transition threads from one status to another
 - Run
 - At some point, the scheduler chooses to run one of these READY threads, based on its scheduling policy.
 - Thread_destroy
 - The scheduler will finally destroy all state associated with the thread by invoking the thread_destroy function. Note that this function is invoked by the scheduler and not by threads directly.
 - functions that are invoked by threads (threads API functions)
 - Thread_create
 - create a new thread, new thread is created with the READY status.
 - thread_exit
 - After the current thread has finished its operation, it can invoke this function itself to tell the thread scheduler that it does not need to run any longer.
 - Thread_yield
 - yield or give up the CPU to other threads
 - Scheduler Put this thread into READY and picks another thread to run
 - thread_sleep
 - To block the current thread and picks another thread to run
 - thread_wakeup
 - A RUNNING thread Invokes a thread in BLOCKED and put it into READY
 - The invoked thread may not run immediately

What is the whole life cycle of a thread?



- A thread invoke thread_create to create a new thread (in READY)

2. At some point, the scheduler chooses to *run* one of these READY threads, based on its scheduling policy.
3. The chosen thread is current thread. (in RUNNING)
4. After the current thread has finished its operation, it can invoke the `thread_exit` function to tell the thread scheduler that it does not need to run any longer (Exited)
5. the scheduler chooses another READY thread to run, ensuring that the CPU remains busy.
6. The scheduler will finally destroy all state associated with the thread by invoking the `thread_destroy` function. Note that this function is invoked by the scheduler and not by threads directly.
7. The current thread can call the `thread_yield` function to yield or give up the CPU to other threads.
8. The scheduler changes the status of the current thread to **READY** (yield puts and then chooses another READY thread to run).

→ FROM 3

9. The current thread can call the `thread_yield` function to yield or give up the CPU to other threads.
 - a. current thread in READY
 - b. and then chooses another READY thread to RUNNING.

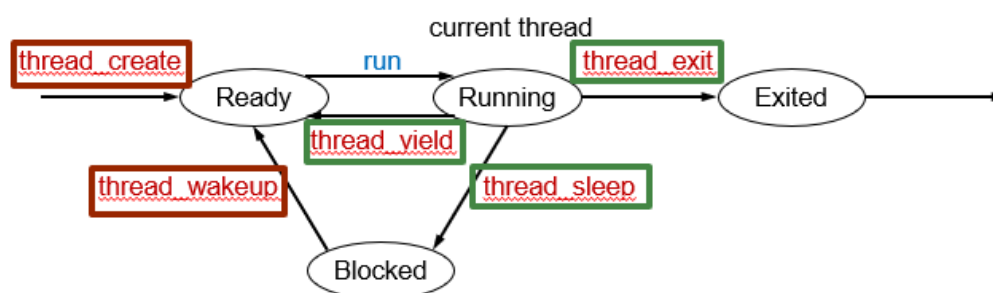
→ FROM 3

10. The current thread can also invoke the `thread_sleep` function to block on some event.
 - a. For example, a thread that tries to read data from the keyboard will sleep if no characters are available.
 - b. In this case, the scheduler changes the status of the current thread to BLOCKED,
 - c. and then chooses another READY thread to run.

→ FROM 10

11. current thread (in RUNNING) invokes the `thread_wakeup` to wakeup *another* thread that is blocked on a specific type of event.
 - a. For example, when the user types a key, the keyboard interrupt handler function in the kernel will wakeup a thread that is waiting to read characters from the keyboard.
 - b. In this case, the scheduler changes the status of the thread that is BLOCKED to READY.
 - c. However, this thread is not necessarily run immediately. Instead, the current thread that invoked `thread_wakeup` may continue to run.

When does the scheduler run a thread?



- Whenever the currently running thread invokes `thread_yield`, `_sleep`, `_exit`, scheduler needs to choose a READY thread to run
- If func are create and wakeup, scheduler might choose thread from READY based on priority scheduling
 - another thread has become READY and the scheduler may choose to run this thread or another thread
 - For example, if a high priority thread is woken up, the scheduler may stop the current thread and run the high priority thread.

Programming with threads

```

main() {
1  |   thread_create(workerA);
   |   thread_create(workerB);
   |   while (1) {
4  |       thread_yield();
   |   }
   | }

workerA() {
2  |   do_A();
   |   thread_yield();
5  |   do_more_A();
   |   thread_exit();
   | }

workerB() {
3  |   do_B();
   |   thread_yield();
6  |   do_more_B();
   |   thread_exit();
   | }

```

- Step 1:
 - in the main function, the OS has already created one thread
 - 1.1) `thread_create` function called x2 to create 2 new threads and sets both to READY that will run the workerA and workerB functions (but not yet), Thus the main thread continues running.
 - Main thread: RUNNING
 - A: READY
 - B: READY
 - 1.2) Next, the main function calls the `thread_yield` function in a loop
 - Main thread: READY
 - pick a READY thread (A OR B) to run.
- Step 2:
 - Chooses workerA thread to run.
 - Main thread: READY
 - A: RUNNING
 - B: READY
 - This thread runs `do_A` and then calls `thread_yield`.
 - A: READY
 - pick a READY thread (B or main) to run.
- Step 3:
 - chooses the workerB thread to run this time
 - Main thread: READY

- A: READY
 - B: RUNNING
- This thread runs `do_B` and then calls `thread_yield`. The scheduler
 - B: READY
 - pick a READY thread (A OR main) to run.
- Step 4:
 - The main thread runs the second iteration of the while loop
 - Main thread: RUNNING
 - A: READY
 - B: READY
 - calls `thread_yield` again.
 - A: READY
 - pick a READY thread (A or B) to run.
- Step 5:
 - Choose to go back to A again, from where it had **last stopped** in `thread_yield`, and so it runs `do_more_A`.
 - Main thread: READY
 - A: RUNNING
 - B: READY
 - Next, workerA calls `thread_exit` and stops running.
 - Main thread: READY
 - A: EXITED
 - B: READY
- Step 6:
 - Scheduler chooses another thread to run, say workerB. WorkerB completes `do_more_B`.
 - Main thread: READY
 - A: EXITED
 - B: RUNNING
 - B calls `thread_exit`
 - Main thread: READY
 - A: EXITED
 - B: EXITED
- Step 7:
 - Can only go back to running the main thread, which is the only thread left in the system, runs this time.
 - Main thread: RUNNING
 - A: EXITED
 - B: EXITED

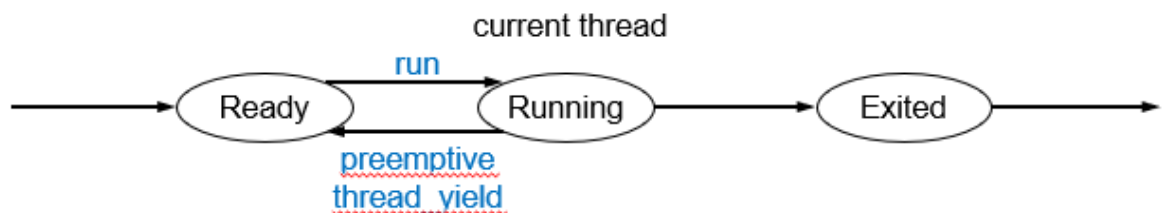
What is a cooperative scheduler?

- Often: thread calls thread API function voluntarily (when it is convenient for the thread), and scheduler will runs another thread

- For example, a thread will call `thread_yield` if it is cooperating with other threads and would like other threads to make progress, as shown in the previous example.
- That is why the scheduler that we have discussed until now is called a cooperative scheduler.

What is a preemptive scheduler?

- When a thread never calls the thread API functions, such as `yield`, `sleep` or `exit` functions
- The scheduler uses a timer **interrupt** to regain control.
 - interrupt suspends the current user program
 - switches to kernel mode
 - and runs the kernel interrupt handler function
 - Then the timer interrupt handler, running on behalf of the current thread, forces a call to `thread_yield`, which stops the current thread and runs another thread.
- A preemptive scheduler can invoke a preemptive `thread_yield` call, as shown in the figure, at any instruction in the thread's instruction stream.
-



When do we say a thread is preempted (or forced to stop),

- When a scheduler forces a thread to stop running, even though it hasn't invoked `thread_yield` in its code,

L9 thinking time

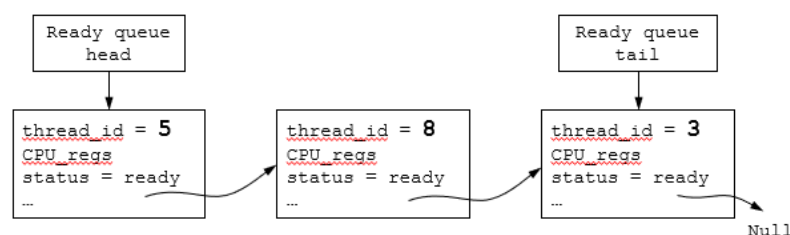
1. What does `thread_yield` do?
 - a. Stop a current running thread, put it to READY state, run another thread in the READY state
2. What are the differences between cooperative and preemptive scheduling?
 - a. cooperative scheduling is used by programs that use user-level threads.
 - i. Switching only happens when the program explicitly invokes `thread_yield`.
 - b. preemptive threading is normally used for kernel-level threads:
 - i. where threads are switched at arbitrary times (from timer interrupts)
 - ii. In this case, different programs are using kernel-level threads, and they cannot be trusted to call `thread_yield` (each program may not even be aware that other threads are running).
3. What are the benefits of cooperative versus preemptive scheduling?
 - a. cooperative scheduling is used by programs:

- i. that use user-level threads.
 - ii. In this case, the different threads of the program are cooperating and so they are expected to yield to other threads.
 - iii. With cooperative threading, switching only happens when the program explicitly invokes `thread_yield`.
 - iv. This is more efficient than
 - b. preemptive threading where threads are switched at arbitrary times (from timer interrupts)
 - i. there is a cost involved with each thread switch.
 - ii. preemptive threading is normally used for kernel-level threads.
 - iii. In this case, different programs are using kernel-level threads, and they cannot be trusted to call `thread_yield` (each program may not even be aware that other threads are running).
 - iv. Thus the OS kernel uses preemptive threading to run programs, so that no program can hog the CPU.
- 4. In the example program using threads, what happens after both workerA and workerB finish executing?
 - main runs the `thread_yield` function in the next iteration of its loop.
 - Since there is only one thread in the system (main thread), this `thread_yield` call does not do anything (it switches back to the same thread),
 - so main keeps running `thread_yield` in a loop.
- 5. what would happen if `thread_exit` is not called?
 - a. the **threading library** will normally call `thread_exit` on behalf of the thread when the thread finishes executing its thread function (the thread function is the first function that started executing when the thread started, this function is passed as an argument to `thread create`).

L10 thread implementation

How does scheduler track threads?

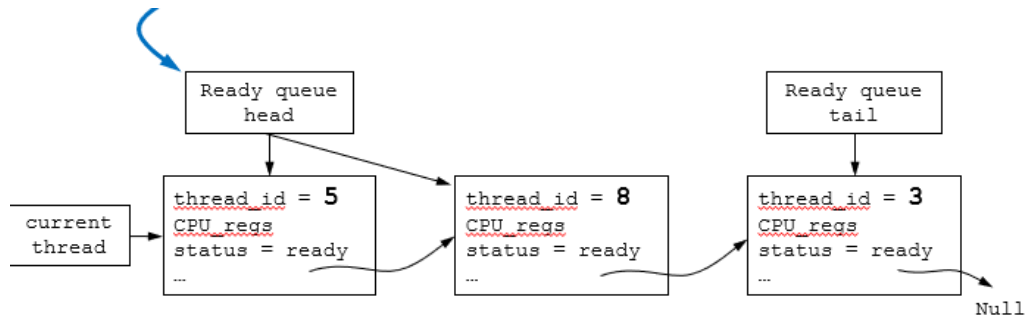
- In queues



-
- Each block represents a thread-related state, with id, CPU reg and thread status

How does the scheduler manage to run a thread?

- By dequeuing a thread from READY queue, have current thread pointer pointing to it, and the `ready_queue_head` will be moved to the second thread state



What happens when `thread_yield` is called

- thread/scheduler preemptively call `thread_yield`
- Scheduler will enqueue the current running thread into the READY queue (at tail)
- Dequeue a thread from the head and run it

What are the three queues used by scheduler?

1. READY queue
 - a. `run()`: dequeue head, run as current thread
 - b. `thread_yield()`: enqueue current to tail, dequeue from head, run
2. WAIT queue
 - a. Threads with BLOCKED status
 - b. Typically, a separated wait queue for each type of event
 - i. a heavily loaded system may have 1000s of wait queues in the system. For example, all threads waiting to read a given file could be enqueued into a separate queue.
 - c. Threads here are not run until they are woken up.
 - i. When the given file becomes available, the storage subsystem of the OS will wakeup these threads by marking them READY and enqueueing them in the READY queue, so that they can be run sometime in the future.
 - ii. Similarly, threads waiting to read from a specific network socket connection could be enqueued in their own wait queue and woken by the networking subsystem of the OS up when a packet arrives on that socket connection.
3. EXITED queue
 - a. Similarly, the scheduler maintains an exited queue for threads with the Exited status. These threads do not need to run any longer and will be destroyed eventually.

Programming with threads + sleep and wakeup version

- With the use of a wait queue `wq`
 - `wq` is a shared variable, which allows threads to share this queue.

1. Step

- a. The main thread starts running and creates two new threads as in the previous example.
 - i. MAIN: RUN
 - ii. A: READY
 - iii. B: READY
 - iv. Wq: []

- b. When it calls `thread_yield`, the scheduler picks another thread to run.
 - i. MAIN: READY
 - ii. A: RUN
 - iii. B: READY
 - iv. Wq: []
- 2. Step
 - a. Say the scheduler chooses the workerA thread to run. This thread runs `do_A` and then invokes `thread_sleep`.
 - i. MAIN: READY
 - ii. A: BLOCKED
 - iii. B: RUN
 - iv. Wq: [A]
- 3. Step
 - a. Say the scheduler chooses the workerB thread to run.
 - b. This thread runs `do_B` and then it invokes *`thread_wakeup`* to wake up any thread that is sleeping on the wq wait queue. This function dequeues the workerA thread from the wq wait queue, marks the thread READY and enqueues it to READY queue. workerB continues running
 - i. MAIN: READY
 - ii. A: READY
 - iii. B: RUN
 - iv. Wq: []
 - c. `do_more_B` and then calls `thread_exit` to stop running.
 - i. MAIN: RUN
 - ii. A: READY
 - iii. B: EXITED
 - iv. Wq: []
- 4. Step
 - a. scheduler chooses another thread to run, say the main thread, in the 2nd iteration of the while loop and calls `thread_yield` again. Only A is ready now
 - b. MAIN: READY
 - c. A: RUN
 - d. B: EXITED
 - e. Wq: []
- 5. Step
 - a. Do more A, and then finish operation, exit, goes back to main again
 - i. MAIN: RUN
 - ii. A: EXITED
 - iii. B: EXITED
 - iv. Wq: []
- 6. The main thread, which is the only thread left in the system, runs again.

How does thread switching work

- thread switching code.
- Recall that the threads abstraction requires
 - switching/suspending the current thread
 - resuming another thread.

Thread switching in `thread_yield`

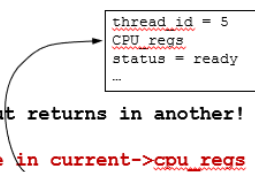
```
// "current" is thread struct for currently running thread
thread_yield()
{
    ...
    // enqueue current thread in the ready queue
    // choose next thread to run, remove it from ready queue
    next = choose_next_thread();
    // switch to next thread
    thread_switch(current, next);
    ...
}
```

1. Enqueue current thread to tail of ready queue
2. uses its scheduling policy, such as round-robin scheduling, to choose the next thread to run, and removes this next thread from the ready queue.
3. Next, the code invokes the `thread_switch` function, which suspends the current thread, and starts running the next thread.

`Thread_switch` function and how it works

```
// call is invoked by one thread but returns in another!
thread_switch(current, next) {
    // save current thread's CPU state in current->cpu_regs
    save_processor_state(current->cpu_regs);
    ...

    // restore next thread's CPU state from next->cpu_regs
    restore_processor_state(next->cpu_regs);
}
```




-
- `Thread_switch` is invoked by the current thread, but it returns in the next thread!
- 1. `save_processor_state` function:
 - a. save all the CPU registers values of the current thread (the current PC) in the `cpu_regs` field in the thread structure (of current thread)
 - i. When this thread is run the next time, these saved values in the `cpu_regs` field will be used to set the CPU registers so that this thread can resume execution. (so the thread restart from where it left behind)
- 2. `restore_processor_state` function:
 - a. Takes the `CPU_regs` values of the next thread (stored in its thread states)
 - b. Sets the current running CPU register values with these new `CPU_reg` from next thread, thus resuming execution of the next thread. (start from where it was left over)

An example of thread switching (this implementation is wrong, needs to add stuffs)

```

// current thread id is 5, next thread id is 6
thread_switch(5, 6) {
    // Step 1: save PC in Thread 5 structure
    thread5->cpu_regs.PC = PC;
    next_instruction:
    ...
    // Step 2: restore PC from Thread 6 structure
    PC = thread6->cpu_regs.PC;
}

```



1. Current PC points to the next instruction after thread_switch
2. We save thread 5's reg PC with the current PC so we can restore from where we left behind next time we run the thread
3. Then set current PC to cpu_reg from the state in thread 6

What is wrong with this implementation?

- At 5, we are not loading the next instruction of where thread 5 leaves at, but the next instruction after loading thread 5 PC with current PC
- Then when we set current PC as thread 6 CPU_reg, and exit from running all things in thread 6, it will save the same PC value in step 1 (to restore thread 5), but this PC value is pointing to the next_instruction, and there will be a loop

How does Process switching work?

- Switch thread + switch address space as well
- So the resumed thread can access its own address space → called context switch
 - Address space is switched by updating the state of MMU

How is thread_create() used to create thread?

1. Thread scheduler starts and create a thread
2. Then, threads invokes thread_create()
3. *thread_create(thread_fn)* function:
 - a. allocating a thread structure
 - b. a new stack for the new
 - c. Initialize the thread's cpu reg pc to thread_fn
 - i. Which is the start instruction of thread function
 - ii. thread function will run after the thread starts exe
 - d. Initialize cpu reg stack to the stack we created
 - i. So thread function can use
 - e. Set threads to READY state, enqueue to the READY queue

How does thread terminate and what happens after that?

- A thread that does not need to run any longer terminates by calling thread_exit
- The scheduler suspends the thread by setting its status to EXITED,
- It then runs another thread, similar to thread_yield and thread_sleep.

Why don't scheduler call thread_destro immediately and destroy all the state associated with the thread, such as the thread structure and the thread stack?

- The reason is that the thread is using this stack when thread_exit is invoked and so destroying it could cause serious corruption.

- After the scheduler switches to running another thread, the stack of the previously exited thread is no longer in use. At this point, the scheduler can destroy any state associated with exited threads.

L10 think time

1. What is the difference between a thread switch and a mode switch?
 - a. Thread switch changes the current executing thread, a kernel switches threads when a thread should stop running and another should start
 - b. Mode switch switches CPU mode between user and kernel modes. This provides a different range of usable instructions to the program, happens when interrupt occurs
 - c. they are unrelated, thread switch switches threads while a mode switch changes the CPU mode. The kernel switches threads when it needs to stop running a thread and resume running another thread. A mode switch occurs when a system call or an interrupt occurs, in which case, the CPU switches from user mode to kernel mode, and starts running kernel code.
2. What is the difference between a thread switch and a context switch?
 - a. Context switch = thread switch + MMU switch
 - b. More expensive
3. What is the difference between a process switch and a context switch?
 - a. Process switch = context switch
4. What are the main steps in the thread_yield implementation?
 - a. Save current thread CPU register to its thread state
 - b. Change thread state from RUNNING to READY
 - c. Add to the READY queue
 - d. Remove another thread from the READY queue
 - e. Mark it as running
 - f. Restore the register state of the next thread to start running this thread
5. The scheduler maintains READY, BLOCKED and EXITED threads in queues. What about RUNNING threads – are they maintained in queues?
 - a. 1 CPU: only 1 running thread, thus can simply put this at the head of READY, or use a global var
 - b. 1+ CPU: put in each CPU's ready queue, or put in a per-CPU variable for tracking the running thread on each CPU maintained by the kernel
6. How does the scheduler implement the first thread?
 - a. When the kernel scheduler starts, it "custom" creates the first thread from the current stream of instructions that the scheduler is already executing.
 - b. To do so, the scheduler creates a thread structure for this stream so that it can save the state of this thread in this structure.
 - c. This thread creation is **different** from a thread_create call, because thread_create runs a function specified in the thread_create call, and it runs this function in the future, while the instructions in the first thread are already running when this thread is created!
7. thread_create allocates a stack for a new thread and sets the stack pointer to point to the new stack. At what precise location does the stack pointer point in the stack?

- a. the stack pointer points to the top of the memory allocated to the stack. This is because the stack grows down and it is initially empty (no functions have been called). → in MMU
8. The scheduler implements `thread_sleep` and `thread_wakeup` to synchronize threads on an event, such as a packet arrival. Can you think of why program generally do not use these calls directly?
 - a. As we will see later, `thread_sleep` and `thread_wakeup` are generally not used directly by programs because they are **race prone**. For example, suppose a receiving thread 1) checks if a packet has arrived or not, and then 2) sleeps if the packet has not arrived. Now suppose the sending thread wakes up the receiving thread when it delivers a packet. The problem is that the sending thread may deliver the packet between steps 1 and 2 above. When it tries to wakeup any thread, no thread is sleeping yet, and so it will do nothing. Then the receiving thread runs step 2 and sleeps forever, even though a packet has arrived. We will discuss this synchronization problem and its solution in great detail later in the course.

L11 kernel threads and user threads

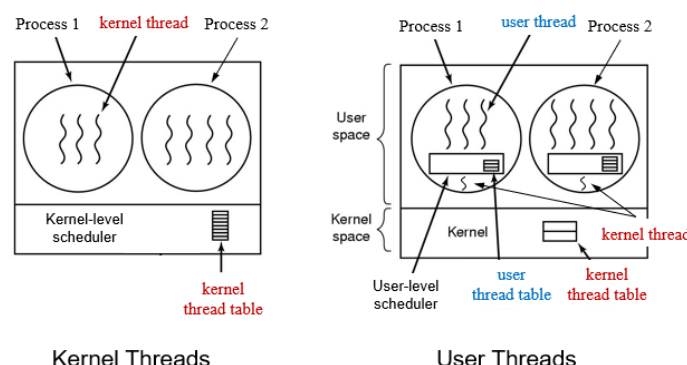
What is a kernel thread?

- When the thread scheduler functions are implemented in the kernel
- Kernel threads virtualize the CPU, allowing an arbitrary number of kernel threads to run on one or more CPUs. In this case, the OS provide system calls to invoke the scheduler's API functions.
 - For example, in Linux, the `pthread_create()` is a system call invokes the kernel's thread create function.

What is a user thread?

- When the thread scheduler functions are implemented in a user program
- User threads virtualize a kernel thread, allowing **multiple user threads** to utilize kernel threads. **These threads run when a kernel thread is running.**

Kernel v.s. User threads



-
- Left: shows a kernel-level thread scheduler running 7 kernel threads. The kernel knows about these 7 threads, as shown by the 7 slots in the kernel thread table. Two processes are running, one with 3 kernel threads, and the other with 4 kernel threads.

- Right: shows how processes can use their own user-level thread schedulers. In this case, both the processes are using one kernel thread each. Thus the kernel knows about two kernel threads, as shown by the 2 slots in the kernel's thread table. Each process implements its own user-level thread scheduler. Process1 has three user-level threads that run when the kernel thread associated with Process1 is run by the kernel. Similarly, Process2 has four user-level threads that run when the kernel thread associated with Process2 is run by the kernel. Notice that kernel scheduler does not know about the user threads. Note also how the user-level thread scheduler is stacked above the kernel-level thread scheduler.

	Kernel Threads	User Threads
Switching cost	Kernel switches threads, requiring running kernel code, more expensive	Program switches threads, time closer to procedure call, less expensive
Scheduling policy	System has fixed policies	User can define custom policy
Blocking system calls	When system call blocks (in kernel), kernel switches to another thread, so overlap of IO and computable is possible	When system call blocks, all user threads (associated with the corresponding kernel thread) block, so overlap of I/O and computation is not possible
Multiprocessors	Different kernel threads can use multiple CPUs in parallel	Different user threads (associated with a given kernel thread) cannot use multiple CPUs concurrently

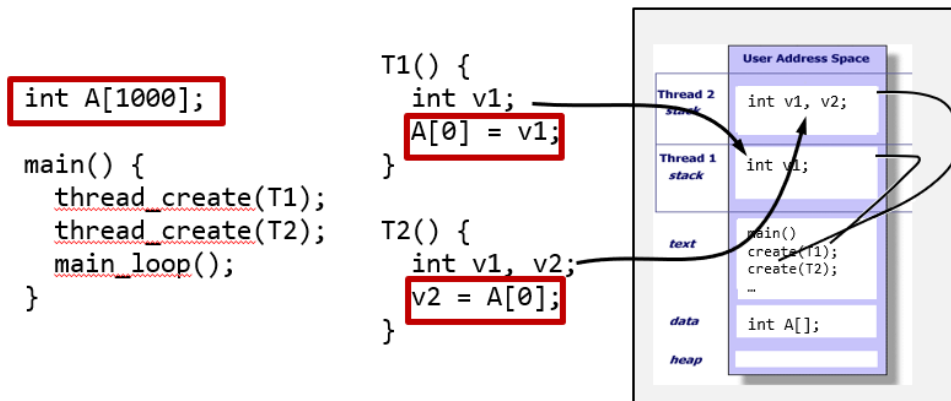
L11 think time

1. We have described the threads API. Do programs invoke this API by using system calls or function calls?
 - a. Kernel threads: system call invokes kernel scheduler's function
 - b. User threads: function calls invoke program's scheduler's function
2. Why is preferable to use kernel threads instead of user threads in more programs?
 - a. Because different kernel threads can run on multiple CPU in parallel, thus more efficient
3. When would you definitely consider using user threads?
 - a. When the program is dealing with a lot of threads, and the program needs precise control over which thread runs next, and which thread runs on which core, and when IO is performed, then user threads are beneficial.
 - b. In a sense, the program is really implementing OS functionality, and so needs to schedule threads itself. Typical examples of such programs are large, complex programs such as web browsers, in-memory databases, that need to handle highly concurrent operations, etc.

L12 mutual exclusion

What is concurrent programming?

- Programming with 2 or 2+ threads cooperating to perform a common task
- Threads cooperate by accessing shared data



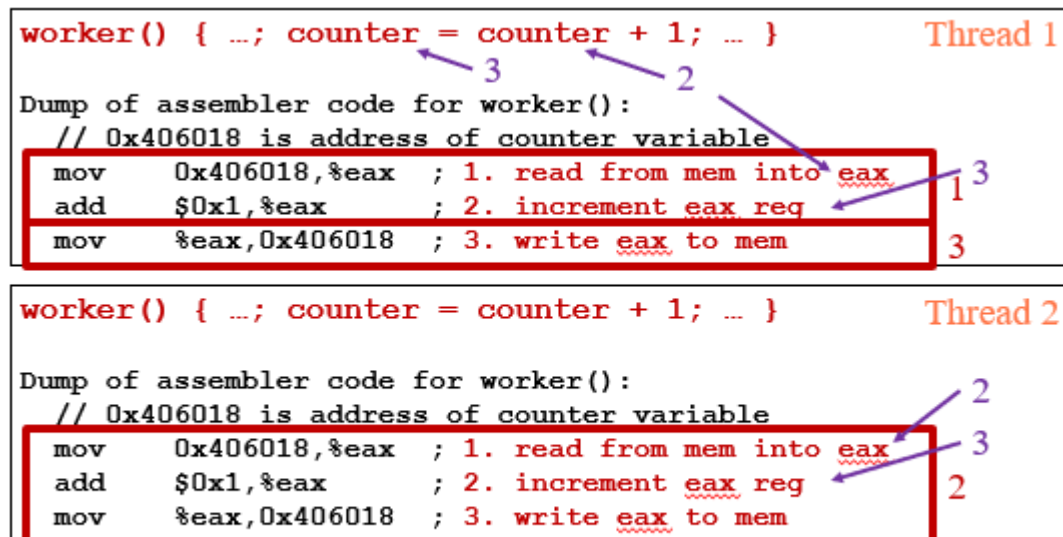
How can threads cooperate with each other?

- Recall this example in which the main thread creates two other threads T1 and T2. Each thread has its own private stack, in which it stores its local variables.
- Thus T1 stores v1 in its stack, and T2 stores v1 and v2 in its stack.
- The global array A is shared by the two threads. Thus T1 and T2 can cooperate by accessing this shared variable. For example, T1 writes to the first element of this array and T2 reads from this element.

Two potential issues in concurrent programming?

- Due to overlapping CPU and IO operations
1. Race condition
 - a. occur when threads access shared data concurrently
 - b. certain interleaving or ordering of thread execution causes incorrect program behavior
 - i. For example, if two threads T1 and T2 both read and update a variable, races can occur, causing incorrect execution, as we will see in the next slide.
 2. Synchronization
 - a. Needed to ensure performance are done in order
 - i. For example, in the previous slide, T1 initializes the first element of the array A, while T2 reads from this initialized variable, and so we need to synchronize these operations to ensure that T1's initialization is performed before T2's read, or else T2 could read a garbage value.

Example of race condition occurring:



□ Is there a thread interleaving that causes problems?

- Step 1 reads the value of the counter variable from memory into the eax register,
- Step 2 increments the value of the eax register
- Step 3 writes the updated eax register value to the counter variable in memory.
- When interleaving happens:
 1. Read val of counter 1 to eax reg → 1
 2. Increment the counter to eax → 1+1=2
 3. Thread 2 starts running, Read val of counter 2 to eax reg → 1
 4. Increment the counter to eax → 1+1=2
 5. Mem is updated to 2
 6. Goes back to thread 1, the mem is 2
- Instead of 3, the value is 2

When does racing occur?

- when certain thread execution interleavings cause unexpected behavior.
- These races may or may not occur depending on many reasons such as how threads are scheduled by the operating system, how many programs or threads are running on the system, and how fast the threads execute on a given CPU.

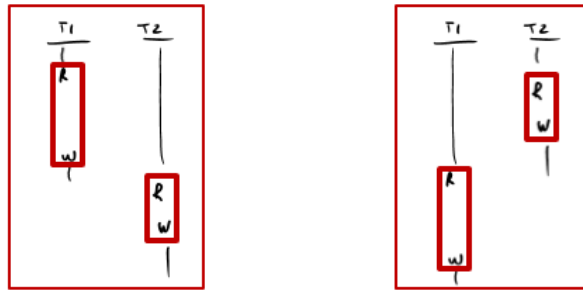
Why is racing hard to debug?

- Races are thus timing-dependent bugs that can lead to random or occasional program failures or crashes

How to avoid races?

- By setting a critical section that disallow the concurrent access, then we can avoid the race. Thus when T1 runs, it will both read and update the counter, before T2 can access the counter, and vice versa.
 - critical section: a region of code that needs to be protected from concurrent access.

- E.g., shared counter updates occur one after the other



How can we enforce critical sections?

- Using mutual exclusion.

What is mutual exclusion?

- Mutual exclusion is a requirement imposed on concurrently executing threads that ensures that at most one thread executes in the critical section at a time.
- If we can ensure mutual exclusion, then we can avoid races.

What is a mutex lock abstraction?

- Something used to implement mutual exclusion
- Threads that access shared data need to use mutex locks, or simply locks, when accessing their critical sections.

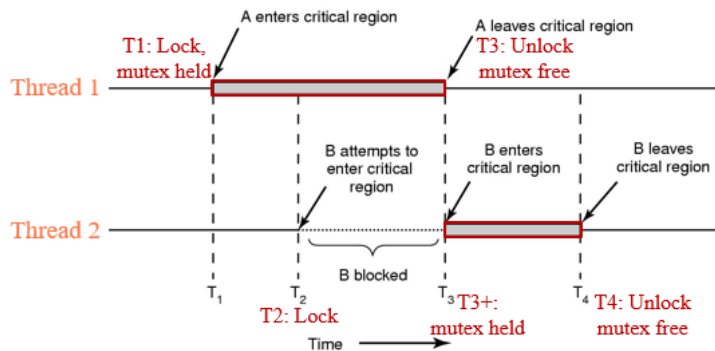
What are the two main parts of mutex lock abstraction?

1. Lock
 2. Unlock
- Both functions operate on a shared variable:
 - Mutex
 - The mutex variable can be in two states, free or held
 - It is a global variable located in the shared address space

How does mutex lock help?

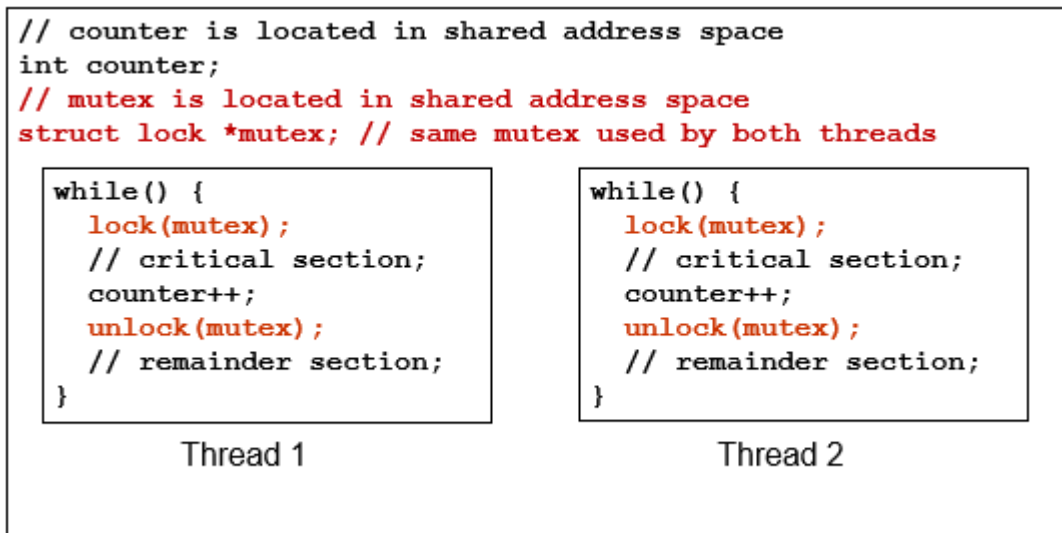
- When a thread needs to access or enter a critical section of code, it first calls the lock function, the lock function checks if the mutex variable is in the free state.
 - Mutex = free: it acquires the lock by marking the lock as held. At this point, the lock function returns and the thread can enter the critical section.
 - Mutex = held: this thread waits until the mutex lock becomes free.
- When a thread that is in the critical section leaves its critical section, it calls the unlock function, the unlock function releases the lock by marking it free.
 - If other threads are waiting to acquire the lock: the unlock function lets them know that the lock is free now, and other threads that are trying to enter their critical section can do so now.

How does mutex locks work with 1 threads



-
- Before Thread1 accesses its critical section, it issues the lock call at time T1.
- mutex lock is free: Thread1 acquires the mutex lock, marks it held, and proceeds to enter the critical section.
- Next, Thread2 attempts to enter its critical section by invoking lock at time T2. Currently, the mutex lock is held by Thread1 and so Thread2 is blocked or it waits until the mutex lock is free.
- When Thread1 leaves its critical section at time T3, it issues the unlock call, which marks the mutex lock as free, and lets threads that are waiting on the mutex lock, like Thread2, know that the mutex lock is free.
- Next, Thread2 can acquire the free mutex lock just after time T3. Thread2 marks the mutex lock as held and then proceeds to enter its critical section.
- Finally, Thread2 leaves its critical section at time T4. It issues the unlock call, which marks the mutex lock as free.

Implementing a mutex in code



What are the conditions a correct locking implementation should SAT:

1. Correctness
 - a. No 2 threads are in the same critical section
 - b. No assumption on the speed of thread execution
2. Performance

- a. No thread running outside of critical section can block the thread running in that section
- b. No thread will wait forever to get in a critical section

L12 think time

1. What is a race condition?
 - a. when certain thread interleavings are possible that lead to incorrect results
2. How can we protect against race conditions?
 - a. Make sure code can access shared variable needs to be executed 1 @ a time
3. What is a critical section?
 - a. A section that no concurrent access should happen
 - b. When code accesses shared variables
4. What is mutual exclusion?
 - a. the requirement that critical sections are executed by at most one thread at a time
5. Are there any drawbacks of long critical sections?
 - drawback of long critical sections: only thread operates in a critical section at a time. that is different threads execute serially in critical sections. thus, long critical sections mean that there is less concurrency or parallelism available, which reduces performance.
 - thus it is important in concurrent programs to lock just the critical sections (where shared variables are accessed) and not the non-critical sections of code.
6. What is a data race?
 - a. we have talked about race conditions in general in the lecture. a special kind of a race condition is a data race: two threads access a shared variable concurrently, at least one access is a write, and the threads do not use any synchronization (e.g., locks) to control access to the variable. the counter example in this lecture is a data race. race conditions can be of other types as well.

L13 Implementing mutex locks

What is the abstraction of mutex lock?

- 1 shared variable: mutex
- 2 states: free and held
- 2 functions:
 - lock(mutex)
 - Acquire lock if free
 - Else wait
 - unlock(mutex)
 - Release lock if is holding lock
 - Tell other threads if threads are waiting

How can mutex implementation go wrong?

#1	<p>Use global variable to track whether a thread is in crit section</p> <div> <pre>lock(mutex) { while (mutex == TRUE) ; // no-op mutex = TRUE; }</pre> <p>CS →</p> </div> <div> <pre>unlock(mutex) { mutex = FALSE; }</pre> </div>	<p>Race condition</p> <p>T1 find lock = false, exit loop CS T2 find lock = false, enter crit sec CS T1 entering crit sec as well Thus, mutex is a shared variable and also needs a crit sec</p>
#2	<p>□ Disable <u>preemption</u> during lock()</p> <div> <pre>lock(mutex) { <u>disable interrupts;</u> while (mutex == TRUE) ; // no-op mutex = TRUE; <u>enable interrupts;</u> }</pre> <p>→</p> </div> <div> <pre>unlock(mutex) { mutex = FALSE; }</pre> </div>	<p>Deadlock</p> <p>No CS allowed in the while loop → mutex will never be changed to false, meaning never giving T1 a chance to unlock T2 spinning in while loop forever</p>

What are the 4 types of lock that can be implemented?

1	Interrupt disabling lock	Single CPU Uses Uses interrupt	<div> <pre>lock() { <u>disable interrupts;</u> }</pre> </div> <div> <pre>unlock() { <u>enable interrupts;</u> }</pre> </div> <p>Interrupt is disabled once the lock is acquired == disable preemption for the entire crit sec (problem) only works on a single CPU(uniprocessor), does not work on multiprocessor/multicore system Why?</p> <ul style="list-style-type: none"> - Interrupt only disabled on local CPU - Multicore, threads can run in parallel - Mutual exclusion is still not guaranteed
2	Spin locks with tset	Multiprocessor Uses polling	<pre>int mutex = 0; // lock is free</pre> <div> <pre>lock(int *mutex) { while (tset(mutex)) ; // no-op }</pre> </div> <div> <pre>unlock(int *mutex) { *mutex = 0; }</pre> </div> <div> <pre>int tset(int *mutex) { int old = *mutex; *mutex = 1; return old; }</pre> </div> <p>If tset return value is 1, then someone else has lock, try again</p> <p>If tset return value is 0, lock is acquired</p> <p>↑ spin lock</p> <p>Why does this work while imp#2 does not?</p> <ul style="list-style-type: none"> - There is not crit sec in read and write in imp#2 - Atomic instruction ensures one instruction runs

			to completion before the other Problem: thread perform no useful work while spinning in a loop
3	Yielding locks	Multiprocessor Uses polling	<div> <pre>lock(int *mutex) { while (tset(mutex)) thread_yield(); }</pre> </div> <div> <pre>unlock_s(int *mutex) { *mutex = 0; }</pre> </div> <p>Thread yields CPU voluntarily while waiting for the lock: CPU performs useful task Later this thread will run tset again to check whether it can acquire lock</p> <ul style="list-style-type: none"> - Need to choose the right polling frequency <p>(Problem:) Scheduler determines when thread_yield return. When lock is available but scheduler does not have the thread running</p>
4	Blocking locks	Multiprocessor and single CPU Uses thread_sleep() and thread_wakeup()	<ul style="list-style-type: none"> - Current thread is blocked by being forced to sleep (thread_sleep) <ul style="list-style-type: none"> - Thread moved from READY to BLOCKED queue - Unlock wake up threads waiting to acquire lock through thread_wakeup <ul style="list-style-type: none"> - Move from BLOCKEDQ to READYQ - Implemented as part of the thread scheduler (problem) <ul style="list-style-type: none"> - Thread_sleep and thread_wakeup access READY and BLOCKED queues == share data - This require a crit sec to avoid raced between changing queues <ul style="list-style-type: none"> - E.g. enqueue and dequeue must be crit sec

What is an atomic instruction?

- The form of support provided by hardware to help locking on multi-processors
 - Operates on a memory word
 - Perform multiple operation while having a crit sec on it

Give 3 examples of atomic instruction

- Atomic increment
- Atomic test and set
 - Tset instruction is hardware instruction operates on an integer
 1. Passed in the address of a variable
 2. Reads variable and store to temp location called old
 3. Write 1 to the variable
 - a. So without changing value, this will always be locked
 4. Return old value
 - Hardware makes sure these performance is done in a crit sec
- Compare and swap

How does multi-processor hardware execute an atomic instruction?

- When a program running on a CPU issues this instruction, this CPU reads and writes the variable, without other CPUs being able to access this variable.
- The CPU requests the memory controller to lock the memory location associated with the variable, so the read and write operations can be performed to completion, before other CPUs can access this memory location.
- In essence, an atomic instruction, such as tset, provides a hardware lock on a word of memory, allowing only one CPU to access this word of memory at a time, in a critical section. This multiprocessor hardware lock will allow us to implement locking in software for multiprocessors.
- This instruction locks the memory location associated with variable, then allow read and write to complete without allowing any other CPU to allow this word of memory

L13 think time

1. Can locks be implemented by reading and writing to a binary variable?
 - No because race condition will happen if these are not performing in a critical section
2. In the interrupt disabling lock, what might be a problem if locks are nested as follows: lock; critical section1; lock; critical section2; unlock; unlock;
 - Since unlock will restore the interrupt back to state when lock is called
 - lock; -- interrupt off
 - critical section1;
 - lock; -- interrupt off
 - critical section2;
 - unlock; -- restore to state when first lock is called, so interrupt on
 - Unlock; -- restore to state when first lock is called, so interrupt off
3. When you would use spin locks versus blocking locks?
 - Use spin when the crit sec are short (with a few 10-100s instructions)
 - Use block when crit sec is long
4. Spin locks and yielding locks use polling. Can you think of similarities between blocking locks and interrupts?
 - a. Blocking locks let the thread sleep and allow other threads use CPU. then wakes up threads when the lock is available
 - b. Similar to using interrupt to let CPU know data is available, so no polling is required
5. Would it make sense to build a blocking lock from a yielding lock?
 - a. a yielding lock runs thread_yield() when a lock is held and cannot be acquired
 - b. a blocking lock runs thread_sleep() when a lock is held and cannot be acquired.
 - c. building a blocking lock using a yielding lock wouldn't be beneficial because the blocking lock implementation has short critical sections (to lock ready queue/wait queue) for which disabling interrupts
 - d. using a spinlock will be more efficient than a yielding lock that invokes the scheduler (by calling thread_yield).
 - e. note that the scheduler for a multiprocessor needs to use a spin lock, not a yielding lock, because a yielding lock uses thread_yield, which the scheduler implements – i.e., the implementation of thread_yield can't use a yielding lock

- f. Summarize:
- g. reason 1: not sufficient enough. Blocking lock has short critical sections (needed to prevent race condition when interacting with data on queues). Using interrupt (uniprocessor) or spin lock (multiprocessor) will be more efficient.
- h. reason 2: both yielding and blocking locks depend on the scheduler. You cannot lock a scheduler using a scheduler function. Blocking locks move threads between a ready/wait queue and operations on these shared data structures require mutual exclusion - you achieved this in lab 3 using interrupt disabling. If you were to try and use yielding locks instead, the lock would call thread_yield, which would operate on a ready queue, which would require mutual exclusion - how would you achieve that? The way out of the chicken-and-egg problem is to use a lower-level lock which doesn't depend on the scheduler.
- i. Technically, you could implement a blocking lock using a yielding lock and then implement the yielding lock using a spin lock, but that brings us back to reason #1.

L14 Using Mutex locks

Which mutex lock to use?

- Locking solutions depend on lower-level locking
- Lower level, more efficient

level	Uniprocessor	Multiprocessor	
high	Blocking lock	Blocking lock <ul style="list-style-type: none"> - USE: Long crit sec, has blocking calls in the middle (thread_sleep) 	
lower	Interrupt disabling block <ul style="list-style-type: none"> - USE: when crit sec are short, 10-100s instructions, no blocking call in the instructions 	Spin lock <ul style="list-style-type: none"> - USE: when crit sec are short, 10-100s instructions, no blocking call in the instructions 	Yielding lock <ul style="list-style-type: none"> - USE: critical sections are long, has blocking calls in the middle (thread_yield) -
Lowest, provided by hardware	Interrupt disabling	Atomic instruction <ul style="list-style-type: none"> - More efficient, this is only a single instruction - USE: when crit section only performs 1 operation e.g. increment a counter atomically 	

How to use mutex locks correctly?

- Some not working code

#1	<pre>// counter is located in shared address space int counter; // mutex1 and mutex2 are located in shared address space struct lock *mutex1, *mutex2; while() { lock(mutex1); // critical section; counter++; unlock(mutex1); // remainder section; } while() { lock(mutex2); // critical section; counter++; unlock(mutex2); // remainder section; }</pre> <p>Thread 1 Thread 2</p>	<p>Will not work</p> <p>Mutex1 and mutex 2 does not post constraints on each other, even if t1 is in crit sec, t2 can still get in because mutex2 is not held by t1</p>
2	<pre>// counter is located in shared address space int counter; // mutex1 and mutex2 are located in shared address space struct lock *mutex1, *mutex2; while() { lock(mutex1); // critical section; counter++; unlock(mutex1); // remainder section; } while() { lock(mutex2); // critical section; counter--; unlock(mutex2); // remainder section; }</pre> <p>Thread 1 Thread 2</p>	<p>Same problem as #1</p> <p>No matter how op in crit sec changes, race cond will occur</p>
3	<pre>// counter is located in shared address space int counter; // struct lock *mutex; // commented out while() { struct lock *mutex; lock(mutex); // critical section; counter++; unlock(mutex); // remainder section; } while() { struct lock *mutex; lock(mutex); // critical section; counter++; unlock(mutex); // remainder section; }</pre> <p>Thread 1 Thread 2</p>	<p>Mutex is a local variable, updated separately on two different stack, each belonging to 1 thread</p> <p>Thus cannot ensure mutex</p>
4	<pre>int counter; update_counter() { struct lock *mutex; lock(mutex); // critical section; counter++; unlock(mutex); } while() { update_counter(); // remainder section } while() { update_counter(); // remainder section }</pre> <p>Thread 1 Thread 2</p>	<p>Mutex is a variable local to the function. Two calls to the function from different thread will create two copies of the mutex variable</p>
5	<pre>// counter is located in shared address space int counter; // mutex is located in shared address space struct lock *mutex; // same mutex used by both threads while() { lock(mutex); // critical section; counter++; unlock(mutex); // remainder section; } while() { lock(mutex); // critical section; if (counter <= 0) continue; counter--; unlock(mutex); }</pre> <p>Thread 1 Thread 2</p>	<p>Will not work because once "continue" is executed, will break out of the loop leaving the lock being held by the thread forever</p>

How to use mutex with data structure?

- Say, 3 threads access a linked list
 - Thread 1 adds elements to the list
 - Thread 2 deletes elements from the list
 - Thread 3 reads an element from the list

- Questions:
 - Should Threads 1 and 2 use the same lock or different locks?
 - Use same lock, otherwise will have race condition
 - Should Thread 3 use a lock at all?
 - Yes, make sure nothing is changing while reading
 - Or can read stale values, or crash due to inconsistent list
 - Should we create one lock for the entire list, or one lock per list node?
 - Depends
 - Single: simplified code
 - 1 per node: more concurrency

L14 think time

1. When and why is a blocking lock better than interrupt disabling or using spin locks?
 - a. Interrupt disabling & spin holds for the entire crit sec
 - b. Spin stays in a while loop for long time
 - c. Blocking only need crit sec at lock and unlock sections, put thread to sleep and allow other threads to use the CPU
2. Is the blocking lock always better?
 - a. Nope. blocking has overhead because it asks the scheduler to perform thread switch. Better to use interrupt disabling (single) or spin (muti)
3. Does it make sense to use a spin lock, rather than a blocking lock on a single CPU?
 - a. No it does not. Using spin lock on single CPU will make the CPU continue spinning in the while loop and the lock will never be available because no other threads can make any progress
 - b. Eventually time interrupt will fire and pre-emp will come in, run thread holding the lock and allow progress to continue
 - c. But if use blocking, this will not happen because the thread that wants to acquire the lock will sleep right away and allow the thread holding the lock to run
4. Why would a thread block within a critical section? Shouldn't a critical section run fast, and so threads shouldn't block within a critical section?
 - a. Synchronization!

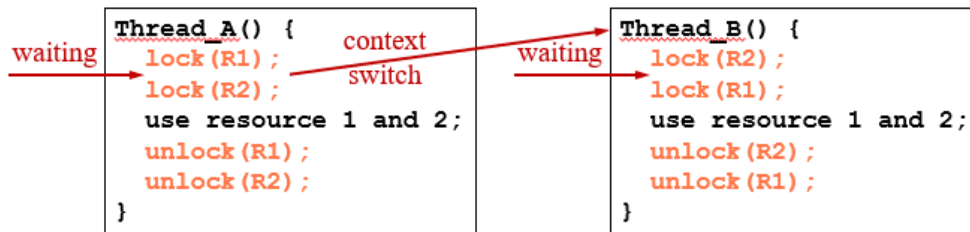
L15 misusing mutex lock

What are the three mutex lock problems?

- Starvation
- Deadlock
- Livelock

What causes deadlock?

- Each thread in the set holds a lock and is waiting for a lock that some other thread in the set holds so no thread can run



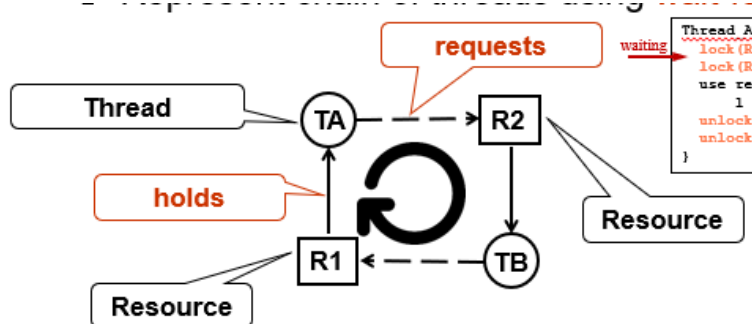
-
- T1 locks R1 → cs → T2 locks R2 → since R1 is locked, T2 waiting → cs → since R2 is locked, T1 waiting
- None of the thread can make progress
- No thread can make progress, will wait forever, need to kill thread to break

Conditions that deadlock occurs:

1. Mutex: resources cannot be shared between threads
 - a. Thread cannot acquire resources held by other thread
2. Hold and wait
 - a. A thread has to be waiting for a resource while holding another one
3. No preemption
 - a. Never force a thread to release resource
4. Circular wait
 - a. Each thread waiting for a resource from next thread in the chain

How to Detect deadlocks?

- Use circular wait condition
- Represent chain of threads using wait-for graph



-
- If we see a cycle in this graph, there is a deadlock

How to avoid deadlock?

1. Avoid hold and wait
 - a. Can only reacquire after release previously acquired locks
 - b. Problem:
 - i. Thread can be waiting to acquire locks again and again, no progress made
 - ii. Data structure changes in previous lock should be undone or rolled back so the data structure is consistent
2. Prevent circular wait
 - a. Number all resources, make sure lower number resources are acquired before the higher one

b. Problem

- i. Hard to number
- ii. Code needs to be written so resources acquired in correct order

What is starvation?

- Thread performs no work because resources it need is held by others constantly
 - E.g. low priority thread not acquire a lock because we always have high priority threads
- Temp:
 - Will run once all high priority threads are done

What is a livelock?

- Thread continues to run but makes no progress
 - Releases and continues to reacquire job
 - Always running, but makes no progress
- Temporary
 - Once thread gets all threads, it will make progress

Benefit and bad thing of fine-grained locking:

- Good: fine-grained locking allows different data structures to be accessed concurrently, so better performance
- Bad: complicated code, locking problems, degradation

L15 think time

1. Are there systems that can handle deadlocks automatically?
 - a. yes, databases often handle deadlocks automatically. databases acquire locks for applications automatically and the locking order may cause cycles, which lead to deadlocks. databases use a heavy weight machinery called transactions to abort and rollback transactions when a deadlock is detected. This abort and rollback prevents deadlocks by avoiding the no-preemption condition required for deadlocks.
2. How can one avoid starvation?
 - a. High priority threads should not be allowed to run for a long time
 - b. For threads with same priority, use FIFO
3. How can one avoid livelock?
 - a. Ensure threads run for a while before switching to running other threads
 - b. Make sure thread can make progress before switching to running another thread
4. Look up interrupt storm on wikipedia. Does this event cause a deadlock, starvation or livelock? How can be solved?
 - a. Interrupt storm can cause a livelock. the solution to an interrupt storm is to disable interrupts and switch to polling when interrupts are arriving too often. This is similar to receiving too many popups (interrupts) for email messages, which leads to no work being done (livelock). At some point, the user may disable the popups, and then periodically check their mail inbox (polling).

L16 intro to synchronization

Why is synchronization needed?

- To ensure operations performed in specific order
 - No reading variable before writing

What is a producer-consumer problem

- A OS synchronization problem
- Bounded buffer, a shared buffer of a fixed size is used to allow threads communication

What are the synchronization conditions?

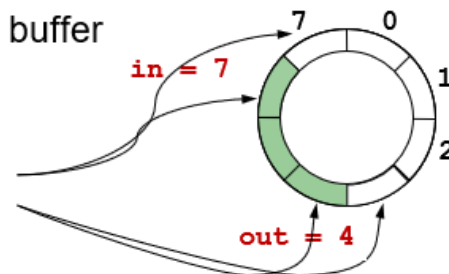
- Producers: fill buffer, wait if buffer full
- Consumers: empty buffer, wait if buffer empty
- (bc buffer is off bounded size)

How to implement bounded buffer

- Implement using a circular buffer

□ Implementation uses a circular buffer

```
shared variables:
char buf[8]; // 7 usable slots
int in;      // place to write
int out;     // place to read
```



-

1. Declare 3 global variables
 - a. Buf[n], then there will only be n-1 usable slots
 - b. Int in: place to write, the first empty place
 - c. Int out: place the read, the last non-empty place
2. If $n = 8$, number of elements in buffer = $(in - out + n) \% n$
 - a. $In = 7, out = 4$, total element = $(7-4+8)\%8 = 3$
 - b. Why + n? Sometimes get negative, so make it positive
3. Buffer = full when count = $n - 1$
 - a. Why? If we use up all n spaces, will have $in = out$, count = 0, then cannot know whether it is full or empty
4. Buffer = empty when count = 0

How to correctly implement producer-consumer?

1	<div data-bbox="272 1637 799 1704">Try 1: Single Producer-Consumer</div> <div data-bbox="272 1715 799 1850"><pre>void send(char elem) { int count = (in - out + n) % n; while (count == n - 1) { } // full buf[in] = elem; in = (in + 1) % n; } char receive() { while (in == out) { } // empty elem = buf[out]; out = (out + 1) % n; return elem; }</pre></div>	<p>Main problem: Count is calculated outside of the while loop, so even if in and out are changed, this value is not updated</p>
---	--	--

2	<div data-bbox="272 210 855 264" data-label="Section-Header"> <h3>Try 2: Single Producer-Consumer</h3> </div> <div data-bbox="272 286 855 416" data-label="Text"> <pre> void send(char elem) { while ((in-out+n)%n == n - 1) { } // full buf[in] = elem; <u>CS</u> in = (in + 1) % n; } char receive() { while (in == out) { } // empty elem = buf[out]; out = (out + 1) % n; return elem; } </pre> </div>	<p>Correct if using a single producer, single consumer. IN OUT variables are read by both threads, but IN is only modified by producer while OUT is only changed by consumer, so no problem</p> <p>If use on multiple producer & consumers? Nope cuz race condition</p> <p>When count != n-1, meaning we can start sending element → CS → another thread sees and adds 1 element, now it is full again. Before it can update in → CS → adds another element in again</p>
3	<div data-bbox="272 772 855 837" data-label="Section-Header"> <h3>Try 3: Use Locking</h3> </div> <div data-bbox="272 860 855 1025" data-label="Text"> <pre> void send(char elem) { while ((in-out+n)%n == n - 1) { } // full lock(1); buf[in] = elem; <u>CS</u> in = (in + 1) % n; unlock(1); } char receive() { while (in == out) { } // empty lock(1); elem = buf[out]; out = (out + 1) % n; unlock(1); return elem; } </pre> </div>	<p>Will not work</p> <p>When not full, t1 goes into the crit sec, t2 will wait outside</p> <p>When t1 leaves, the buffer is full, but t2 will still go in and add element</p>
4	<div data-bbox="272 1070 855 1135" data-label="Section-Header"> <h3>Try 4: Use Locking for While Loop as well</h3> </div> <div data-bbox="272 1158 855 1323" data-label="Text"> <pre> void send(char elem) { lock(1); while ((in-out+n)%n == n - 1) { } // full buf[in] = elem; in = (in + 1) % n; unlock(1); } char receive() { lock(1); while (in == out) { } // empty elem = buf[out]; out = (out + 1) % n; unlock(1); return elem; } </pre> </div>	<p>Will never leave a crit section when it is full because consumer cannot consume the element when lock is acquired by producer</p>
5	<div data-bbox="272 1368 855 1433" data-label="Section-Header"> <h3>Try 5: Release Locks Before Spinning</h3> </div> <div data-bbox="272 1444 855 1646" data-label="Text"> <pre> void send(char elem) { lock(1); while ((in-out+n)%n == n - 1) { unlock(1); lock(1); } // full buf[in] = elem; in = (in + 1) % n; unlock(1); } char receive() { lock(1); while (in == out) { unlock(1); lock(1); } // empty elem = buf[out]; out = (out + 1) % n; unlock(1); return elem; } </pre> </div>	<p>Correct!</p> <p>When full, producer temp release lock so consumer can empty it by acquiring the lock</p> <p>Has no deadlock because the hold and wait condition is avoided</p> <p>Problem:</p> <ul style="list-style-type: none"> - Inefficient - Constantly spinning in loop while not full, waiting - Similar to polling for a lock release in a spin lock

6	<div data-bbox="277 210 847 264" data-label="Section-Header"> <h3>Try 6: Sleep After Unlocking</h3> </div> <div data-bbox="277 271 847 555" data-label="Text"> <pre>// shared variable wait_queue full, empty; void send(char elem) { lock(l); while ((in-out+n)%n == n - 1) { unlock(l); thread_sleep(full); ← CS lock(l); } // full buf[in] = elem; in = (in + 1) % n; thread_wakeup(empty); unlock(l); } char receive() { lock(l); while (in == out) { unlock(l); thread_sleep(empty); lock(l); } // empty elem = buf[out]; out = (out + 1) % n; thread_wakeup(full); unlock(l); return elem; }</pre> </div>	<p>Race condition between thread_sleep and thread_wakeup</p> <p>If CS happens before thread goes into sleep, then in receiver, thread_wakeup is called once, then CS back to thread_sleep. This thread will sleep even the buffer is no longer full.</p> <p>Suffers from lost wakeup problem</p>
7	<div data-bbox="277 589 847 642" data-label="Section-Header"> <h3>Try 7: Sleep Before Unlocking</h3> </div> <div data-bbox="277 649 847 934" data-label="Text"> <pre>// shared variable wait_queue full, empty; void send(char elem) { lock(l); while ((in-out+n)%n == n - 1) { thread_sleep(full); unlock(l); lock(l); } // full buf[in] = elem; in = (in + 1) % n; thread_wakeup(empty); unlock(l); } char receive() { lock(l); while (in == out) { thread_sleep(empty); unlock(l); lock(l); } // empty elem = buf[out]; out = (out + 1) % n; thread_wakeup(full); unlock(l); return elem; }</pre> </div>	<p>Thread_sleep before releasing the lock: can cause a deadlock problem</p> <p>A thread is sleeping, so no other threads can run and go in the lock session to wake this thread up</p> <p>Similar to spinning while holding the lock</p>

L16 think time

- What is the difference between mutual exclusion and synchronization?
 - Mutex ensure no 2 threads will be accessing shared data at the same time using critical section
 - Avoid races
 - Synchronization ensures threads perform operations in a particular order
 - Order operations
- In Try 4, suppose send and receive use different locks. Would that work?
 - Yes will work. Because producers and consumers will each have a crit section. Similar to how single producer and single consumer works
 - Not good because will always be spinning
- In Try 6, producer woke up consumer before releasing the lock. Wouldn't that cause a problem?
 - Will not. Since consumer cannot run, will switch back to the producer and unlock
- In Try 6, suppose the producer woke up consumer BEFORE incrementing the IN variable. Would that cause a problem?
 - No. consumer needs a lock to wake up. So will eventually go back to producer and finish running the thread until unlock
- In Try 6, suppose the producer woke up consumer AFTER releasing the lock. Would that cause a problem?
 - Cs happens before the thread_wakeup, then consumers will be sleeping forever

L17 monitors

What do monitors do?

- Provides systematic solution for both mutex and synchronization

What are the two requirements for mutex?

1. Any data is accessed with methods of data structure
 - a. Within a shared queue data structure
 - b. All accesses to queue is performed in enqueue and dequeue methods
2. Every method should have lock at beginning and unlock at the end

```
Enqueue(queue) {  
    lock(1);  
    ...  
    unlock(1);  
}
```

```
Dequeue(queue) {  
    lock(1);  
    ...  
    unlock(1);  
}
```

-
- This ensures all data are accessed in critical section

How does monitor realize synchronization?

- Use 1/1+ condition variables within methods
 - Allow threads to
 - Wait
 - Wakeup another waiting thread

2 conditional variables

- wait(cv,lock)
 - Wait on condition until another thread signals it
 - Release lock when waiting
 - When thread is signaled, first reacquire lock
 - Return

```
void  
cv_wait(struct cv *cv, struct lock *lock)  
{  
    int enabled = interrupts_set(0);  
    // disable interrupt  
    assert(cv != NULL);  
    assert(lock != NULL);  
    if(lock->is_held)  
    {  
        // check that the calling thread had acquired lock  
        when this call is made.  
        // will be checked in lock_release  
        // release the lock before waiting  
        lock_release(lock);  
        // Suspend the calling thread on the condition  
        variable cv  
        thread_sleep(cv->cv_wait_queue);  
    }  
}
```

```
        // reacquire lock before returning from this
        function.
```

```
    - Sleep on wait queue
```

```
        lock_acquire(lock);
```

```
    }
```

```
    interrupts_set(enabled);
```

```
    - ENABLE interrupt
```

```
    return;
```

```
}
```

- signal(cv,lock)
 - Wakeup 1 thread waiting on condition
 - If signal occurs before a wait, then does nothing

```
void
```

```
cv_signal(struct cv *cv, struct lock *lock)
```

```
{
```

```
    int enabled = interrupts_set(0);
```

```
    assert(cv != NULL);
```

```
    assert(lock != NULL);
```

```
    // check that the calling thread had acquired lock
```

```
    if(lock->is_held)
```

```
    {
```

```
        // Wake up one thread that is waiting on the
        condition variable cv.
```

```
        thread_wakeup(cv->cv_wait_queue,0);
```

```
    - Wake up threads sleeping on wait queue
```

```
    }
```

```
    interrupts_set(enabled);
```

```
    return;
```

```
}
```

1	<pre> char *v = NULL; lock l = FALSE; cv init_cv; // called by Thread T1 Init() { lock(l); v = malloc(...); // signal that v // is non-NULL signal(init_cv, 1); ... unlock(l); } // called by Thread T2 Read() { lock(l); // wait until v is non-NULL wait(init_cv, 1); assert(v); // read *v ... unlock(l); } </pre> <p>Sync</p> <p>□ Is this code correct?</p>	<p>Will have problem If signal is called before a wait happens, then when wait is called, the thread will never get a signal to wake up</p> <p>Race condition between signal and wait</p>
2	<pre> char *v = NULL; lock l = FALSE; cv init_cv; // called by Thread T1 Init() { lock(l); ? v = malloc(...); // signal that v // is non-NULL signal(init_cv, 1); ... unlock(l); ? } // called by Thread T2 Read() { lock(l); ? if (v == NULL) { CS // wait until v is non-NULL wait(init_cv, 1); } assert(v); // read *v ... unlock(l); ? } </pre> <p>CS</p> <p>□ Wait/signal within lock</p>	<p>In V==NULL, get a CS out, V is updated and set to not null, CS back will go to wait again because no checking before entering wait</p> <p>Race condition So signal and wait should be called between lock</p>
3	<p>Global variables: buf[n], in, out; lock l = FALSE; cv full; cv empty;</p> <p>Why use "while", instead of "if"?</p> <pre> void send(char elem) { lock(l); while ((in-out+n)%n == n-1) { wait(full, 1); } // full buf[in] = elem; in = (in + 1) % n; signal(empty, 1); unlock(l); } char receive() { lock(l); while (in == out) { wait(empty, 1); } // empty elem = buf[out]; out = (out + 1) % n; signal(full, 1); unlock(l); return elem; } </pre> <p>Why use while instead of if? If we use if, then a CS happening in wait(), 1 thread come, already pass full condition, will write to buffer, CS back to previous thread, will continue, since no more checking, will also write to buffer</p> <p>** note, wait and signal passes in cv and lock</p>	<p>WORKS!</p> <p>1 shared lock variable 2 condition variables, full/empty (producer and consumers waiting on different buffer conditions, but the same buffer)</p> <ul style="list-style-type: none"> - In send() - When count not full, fill buffer + signal threads waiting on empty condition to wake up - If no consumer is sleeping on empty condition, do nothing - When count == n-1, Producer wait on full condition

Monitor v.s. Try 6 (not working) Producer consumer

Global variables:
 buf[n], in, out;
 lock 1 = FALSE;
 cv full;
 cv empty;

Why use "while",
 instead of "if"?

```
void send(char elem) {
    lock(1);
    while((in-out+n)%n == n-1) {
        wait(full, 1);
    } // full
    buf[in] = elem;
    in = (in + 1) % n;
    signal(empty, 1);
    unlock(1);
}

char receive() {
    lock(1);
    while(in == out) {
        wait(empty, 1);
    } // empty
    elem = buf[out];
    out = (out + 1) % n;
    signal(full, 1);
    unlock(1);
    return elem;
}
```

Try 6: Sleep After Unlocking

```
// shared variable
wait_queue full, empty;

void send(char elem) {
    lock(1);
    while((in-out+n)%n == n-1) {
        unlock(1);
        thread_sleep(full); ← CS
        lock(1);
    } // full
    buf[in] = elem;
    in = (in + 1) % n;
    thread_wakeup(empty);
    unlock(1);
}

char receive() {
    lock(1);
    while(in == out) {
        unlock(1);
        thread_sleep(empty);
        lock(1);
    } // empty
    elem = buf[out];
    out = (out + 1) % n;
    thread_wakeup(full);
    unlock(1);
    return elem;
}
```

- Combine unlock, thread_sleep, lock → wait
- Thread_wakeup = signal
- Difference: monitor ensures [unlock, thread_sleep, lock] are performed in a critical section. So no context switch can happen in this section i.e. running sleep before wakeup

L17 think time

1. Why are locks, by themselves, not sufficient for solving synchronization problems?
 - a. Locks cannot ensure the order
 - b. Synchronization requires sleep and wakeup which are more general primitive
2. Why is a lock passed to wait and signal in a monitor?
 - a. To ensure lock, sleep, and unlock are in 1 critical section
 - b. Wait: release lock before sleeping and reacquires lock after waking up
 - c. Signal: no lock is used, but must be called when current thread holds the lock, so if there is a bug no signal can report it
3. What is the broadcast operation on a condition variable?
 - a. To wake up ALL threads sleeping on this condition variable

L18 semaphores

What do semaphores do?

- Converts mutex and synchronization to a resource management problem
 - Acquire: resources when needed. If not available,
 - waiting
 - Release: when don't need resource,
 - Waking up threads waiting for this

What is a semaphore

- A variable that tracks number of available resources
- 2 operations are used:
 1. Down/probeer
 - a. Acquire a resource
 - b. Call it down because now available resource becomes 1 less
 2. Up/verhoog
 - a. Release a resource
 - b. Up bc one more resource is available now

How is semaphore implemented

<p>1. Basic idea</p>	<pre>semaphore s = INIT_NR_RESOURCES;</pre> <pre>down(semaphore s) { while (s <= 0) { // wait until resource is available } s = s - 1; // acquire a resource // after down(), s >= 0 }</pre> <pre>up(semaphore s) { s = s + 1; // make a resource available }</pre> <ul style="list-style-type: none"> - First declare a semaphore variable == number of resources initially available - If in down: <ul style="list-style-type: none"> - wait until resource is available (if $s \leq 0$, stay in while loop) - If resource available, acquire 1 resource, decrement s - If in up: <ul style="list-style-type: none"> - Simply increase s by 1, so make +1 resource available 	<p>But this code is wrong!</p> <p>Accessing same semaphore variable, so locks are needed</p> <p>Spinning in a while loop. So very inefficient</p>
<p>2. Blocking semaphores</p>	<p>Implementing Blocking Semaphores</p> <pre>struct semaphore { int count; wait_queue_wg; ... };</pre> <pre>down(semaphore *sem) { while (sem->count <= 0) { thread_sleep(sem->wg); } sem->count--; }</pre> <pre>up(semaphore *sem) { sem->count++; thread_wakeup(sem->wg); }</pre> <p>Implemented using condition variables and blocking locks (==monitor) In down, in the while loop, as semaphore count ≤ 0, call thread sleep Then in up, once counter++, more resource is available, thread_wakeup informs the waiting thread</p>	<p>No problem for spinning in loop But still need mutex</p>
<p>3. Blocking semaphores on single CPU</p>	<p>Blocking Semaphores on Single CPU</p> <pre>struct semaphore { int count; wait_queue_wg; ... };</pre> <pre>down(semaphore *sem) { disable interrupts; while (sem->count <= 0) { thread_sleep(sem->wg); } sem->count--; enable interrupts; }</pre> <pre>up(semaphore *sem) { disable interrupts; sem->count++; thread_wakeup(sem->wg); enable interrupts; }</pre> <p>Disables interrupt when in down, and enable when getting out Why disable interrupt before thread_sleep?</p>	<p>Is it a problem to sleep with interrupts disabled?</p> <ul style="list-style-type: none"> - After sleep() called, either run thread_yield or thread_sleep - They both access shared run queue, so disabled interrupt is needed in both cases - But in yield/sleep, interrupts will be

	<ol style="list-style-type: none"> 1. Avoids a lost wakeup notification (before sleep, after while) 2. Thread_sleep and thread_wakeup accesses shared data (run queue), so disable interrupts for mutex <p>-</p>	<p>enabled later on, which means when next thread resumes execution in yield/sleep, enables interrupt</p>
--	--	---

How can semaphore help with synchronization?

1. Variable initialization

```
char *V = NULL;
semaphore init_sem = 0;
```

```
// called by Thread T1
Init() {

    V = malloc(...);
    // signal that V
    // is initialized
    up(init_sem);
    ...
}
```

```
// called by Thread T2
Use() {

    // wait until V is initialized
    down(init_sem);

    assert(V);
    // read V
    ...
}
```

Sync

- Up is called after V is initialized
- Since S no longer ≤ 0
 - Down is called, decrement s by 1
 - Then read
- Why work?
 - CS before calling up after allocating V: use() will not proceed because down will not allow it to pass

2. Producer-consumer with semaphores

Producer-Consumer with Semaphores

Global variables:

```
buf[n], in, out;
sem full = 0; // no full slots
sem empty = n; // all slots are empty
lock l = 0;
```

- Why does the code not check the full and empty buffer conditions?

- Does code work for multiple producers/consumers?
- Can we switch down(), lock()?

```
void send(char elem) {
    down(empty);
    lock(l);
    buf[in] = elem;
    in = (in + 1) % n;
    unlock(l);
    up(full);
}
```

```
char receive() {
    down(full);
    lock(l);
    elem = buf[out];
    out = (out + 1) % n;
    unlock(l);
    up(empty);
    return elem;
}
```

- Declare 2 variables, full and empty

- Full = 0
- Empty = n
 - Originally there are n lots empty
 - Total initialize the array to have n+1 size
 - Every Time increment, in = (in+1)%n
- In send()
 - Down on empty
 - Wait here until empty is not <= 0
 - Which means there is empty slot available
 - Up on full
 - +1 to full
- In receive()
 - Down on full
 - Wait until full is no longer <= 0
 - So resource is available
 - Up on empty
 - +1 empty spot
- CANNOT switch lock and down()
 - Then, **deadlock** may occur
 - If buffer is full, then send function will wait in down
 - Since t1 is holding lock while waiting, t2 cannot go in receiver() crit sec to read element.
- But order of unlock and up can be switched
 - I increment but don't update will just cause me to only be able to run after i come back and update

Why does the semaphore implementation not check the full and empty buffer conditions, but monitor does?

- Monitor: need to check whether buffer is empty (in == out) for consumer, if is full ((in - out + n)%n == n-1) for producer
- Semaphore: don't need to count this for buffer. Only need to check semaphore variable is <= 0 or not
- Reason: semaphore tracks the # of resources currently available. Check is done in the down() operation already, so operation only waits when resource not available
- Condition variable wait() occurs every time, so need to check condition before invoking wait

Semaphores VS monitors

Semaphores versus Monitors	
<pre>Init() { lock(l); v = malloc(...); signal(&init_cv, 1); unlock(l); }</pre> <p>Variable initialization using monitor</p>	<pre>Use() { lock(l); if (v == NULL) { wait(&init_cv, 1); } assert(v); // read v unlock(l); }</pre>
<pre>Init() { v = malloc(...); up(&init_sem); }</pre> <p>Variable initialization using semaphore</p>	<pre>Use() { down(&init_sem); assert(v); // read v }</pre>

1. Semaphore does not require lock
2. Semaphore combines if(V==NULL) and wait together to down()

- No need to check for the shared variable, because it has semaphores that store number of resources available
- Condition variables do NOT store any state.
- So with wait(), the thread will always be locked v.s. With down(), thread is only blocked when resource is not available (done in a crit section in down implementation)

Similarities between interrupts and semaphores?

- Disabling interrupt == acquiring CPU resource
- Enabling interrupt == releasing CPU resources

Difference between semaphores and locks?

- Similarity:
 - When s = 1 (initialize)
 - Down == lock (set 1 to 0)
 - Up == unlock (set 0 to 1)
- Differences
 - Semaphores is for synchronization
 - Locks are for mutex
 - down() and up() can be called by different threads
 - up() (resource added to buffer before reader, called by consumer, "bank") can be called before down()
 - Unlock is never called before lock

L18 think time

- Why are locks not sufficient for solving synchronization? How are locks different from semaphores?
 - Locks cannot help with maintaining orders.
 - Locks for mutex, semaphore for synchronization
 - Lock and unlock can only be called by same thread, up() down() can be called by different threads
 - up() can happen before down(), where a thread banks resource. But unlock never called before up
- What are the P and V operations on a semaphore?
 - P == down
 - V == up
- How would you solve the producer-consumer problem using interrupt disabling on a single CPU?

Global variables:

```
buf[n], in, out;
lock 1 = FALSE;
cv full;
cv empty;
```

```
void send(char elem) {
    lock(1);
    while ((in-out+n)%n == n-1) {
        wait(full, 1);
    } // full
    buf[in] = elem;
    in = (in + 1) % n;
    signal(empty, 1);
    unlock(1);
}
```

Why use "while",
instead of "if"?

```
char receive() {
    lock(1);
    while (in == out) {
        wait(empty, 1);
    } // empty
    elem = buf[out];
    out = (out + 1) % n;
    signal(full, 1);
    unlock(1);
    return elem;
}
```

Producer-Consumer with Semaphores

Global variables:

```
buf[n], in, out;
sem full = 0; // no full slots
sem empty = n; // all slots are empty
lock 1 = 0;
```

```
void send(char elem) {
    down(empty);
    lock(1);
    buf[in] = elem;
    in = (in + 1) % n;
    unlock(1);
    up(full);
}
```

- Why does the code not check the full and empty buffer conditions?

- Does code work for multiple producers/consumers?
- Can we switch down(), lock()?

```
char receive() {
    down(full);
    lock(1);
    elem = buf[out];
    out = (out + 1) % n;
    unlock(1);
    up(empty);
    return elem;
}
```

- How would you implement a blocking semaphore on a multi-processor?

- a. blocking semaphore for multiprocessors:
- b. In addition to disabling interrupts (so there is no concurrency with the local CPU), a **spinlock** on a **wait queue** associated with the sem semaphore needs to be acquired (to avoid lost wakeup).
- c. This spinlock must be released in the scheduler because threads can't sleep while holding the spinlock (or a deadlock will happen).
- d. To ensure mutual exclusion and avoid a **wakeup** being lost, the scheduler's **thread sleep** function must first **acquire a ready queue spinlock** (this could be per-processor), add the thread to the ready queue, and only then **release the wait queue spin lock before calling thread switch.**
- e. When the thread returns from **thread_switch**, **thread_sleep** should reacquire the **wait queue spin lock**, and then **release the ready queue spin lock.**
- f. Similarly, **thread wakeup** must be called after acquiring the **wait queue spin lock**, and it should **acquire the ready queue spinlock** when the **thread being woken up is added to the ready queue.** These steps will ensure that there are no deadlocks and there is no race between the up and down operations (i.e., a wakeup is not lost). The reason is as follows:
 - g. down:
 - h. **lock(wait_queue): check condition for sleeping:**
 - i. **sleep: lock(ready_queue):**
 - i. **add to ready queue:**
 - ii. **unlock(wait_queue):**
 - iii. **thread switch; ...**
 - j. **up:**
 - k. **lock(wait_queue):**
 - l. **wakeup:**
 - i. **lock(ready_queue):**
 - ii. **remove from ready queue; ...**
 - m. **No deadlock: sleep unlocks wait queue before thread switch, which will avoid deadlock**
 - n. **no race: since down and up acquire both the wait_queue and the ready_queue locks, a lost wakeup is not possible. By the time, down() checks the condition and is ready to sleep, it has **acquired the wait_queue lock**, and so up() cannot race with it. Similarly, by the time down() releases the wait_queue lock (in sleep), it has acquired the ready_queue lock, so, and so again up() cannot race with it!**
5. How are monitors, semaphores different?
 - a. Monitors use condition variables
 - i. Has no state
 - ii. Each cv associates with a queue of that state
 - iii. Everytime needs to go back and check the state of the buffer
 - b. Semaphores use semaphore
 - i. Stores number of resources available
6. Why do semaphores requires initialization but condition variables don't require initialization?

- a. Semaphores need to keep track of number of resources
 - b. Condition variables do not hold any state
- 7. What are the differences between wait() and down()?
 - a. wait() will always call thread_sleep and block the current thread
 - i. Releases lock and re-acquire lock later
 - b. down() will first check if the resource is available. If yes, then no block occurs
 - i. No notion of an associated lock
- 8. What are the differences between signal() and up()?
 - i. Signal can be lost if no one is waiting
 - 1. Needs lock with condition variables (verify if the lock is being held by this variable)
 - ii. Always increase resource available
- 9. Why might you prefer monitors or semaphores?
 - a. semaphores: variable init and consumer-producer problem are easily solved with semaphores
 - b. Monitor: some problem needs mutex, and wants arbitrary condition for waiting

L19 classic synchronization problems

3. Readers and Writers problem

- Multiple readers and writers want to access shared data
- What are the synchronization requirements?
 - 1. Multiple readers CAN read concurrently
 - 2. When 1 writer is accessing data, no other reader/writer can access
- Goal: max concurrency, prevent starvation

```

Int rc = 0;
// 1. number of readers reading the resource
Mutex lock = unlocked;
// 2. use lock to protect rc
Semaphore available = 1;
// 3. Count resource for synchronization
writer()
{
    down(available);
    // only write data if available
    // since semaphore is initialized to 1, down behaves like lock acquire

    // write shared data
    up(available);
    // make this available again since
    // behaves like releasing lock
}

reader()
{

```

```

lock(lock)
If (rc == 0)
{
    down(available);
    //when the first reader starts reading the data, no writer can write
    // but other readers can still read
}
Rc = rc + 1
unlock(lock)

// start reading

// once reading finishes
lock(lock)
Rc = rc - 1
If (rc == 0)
{
    up(available);
    //when the last reader finishes reading the data, release the resource
    // writer can start writing after
}
unlock(lock)
}

```

Readers/Writers – Synchronization

```

int rc = 0;
Mutex lock = UNLOCKED;
Semaphore available = 1;

```

```

Writer () {
    // non-critical section
    down(available);
    // Write shared data
    up(available);
}

```

```

Reader () {
    lock(lock);
    if (rc == 0)
        down(available);
    rc = rc + 1;
    unlock(lock);
    // Read shared data
    lock(lock);
    rc = rc - 1;
    if (rc == 0)
        up(available);
    unlock(lock);
    // non-critical section
}

```

- Problem?
 - Can lead to **starvation**
 - If many readers arrive one after another to read data, writer will only make progress after ALL readers have finished
- Solution?
 - Queue the readers and writers, ensuring FIFO order

4. Sleeping barber

- Logic:
 - Barber:
 - Sleep when no customer
 - Wake when customer comes, cut hair
 - When done, move to next one

- If no customer, go back to sleep
- Customer:
 - If barber sleep. Wake barber up
 - If barber working, has seat, go sit and wait
 - If no seat, leave
- Model this problem:
 - B, C are all threads
 - 1 state variable: # of customers
 - 1 mutex lock to protect state variable
 - 2 semaphores:
 - Barber semaphores: is barber available
 - Customer semaphores: # customers

Const int CHAIRS = 5

Int nrwaiting = 0;

// shared variable

Mutex lock = UNLOCKED;

// protect nrwaiting

Semaphore customer = 0;

Semaphore barber = 0;

B_thread()

```
{
    while(1)
    {
        down(customer); ** note this is outside of the lock
        lock(lock);
        nrwaiting --;
        unlock(lock);
        up(barber);
        // becomes available
        cut_hair();
    }
}
```

}

C_thread()

```
{
    lock(lock); *** needs to be outside of if, otherwise will risk updating nrwaiting multiple
times if CS happens in the middle
    if(nrwaiting < CHAIRS)
    {
        nrwaiting ++;
        unlock(lock);
        up(customer);
        down(barber);
        // acts like a lock for barber
        get_hair_cut();
    }
    Else
    {
        // leave
        unlock(lock); ***remember to unlock here otherwise will get deadlock
    }
}
```

```

}

}

```

```

const int CHAIRS = 5; int nr_waiting = 0;
Mutex lock = UNLOCKED; // protect nr_waiting
Semaphore customers = 0; Semaphore barber = 0;

Barber_Thread() {
    while (1) {
        down(customers);
        lock(lock);
        nr_waiting--;
        unlock(lock);
        up(barber);
        cut_hair();
    }
}

Customer_Thread() {
    lock(lock);
    if (nr_waiting < CHAIRS) {
        nr_waiting++;
        unlock(lock);
        up(customers);
        down(barber);
        get_hair_cut();
    } else { // give up
        unlock(lock);
    }
}

```

-
- Why need the number of customers waiting variable in solution if we have semaphore?
 - Semaphore ONLY allows up and down operations
 - Does NOT allow reading value of semaphore variables because this read can RACE with up and down op
 - If allow up after read, then any value base on the previous read will not be valid

L19 think time

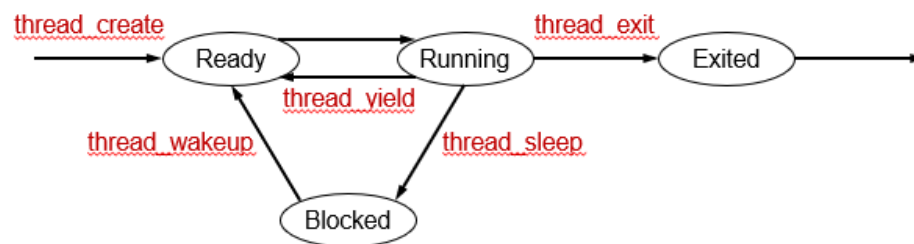
1. A variant of the reader-writer problem is the reader-writer lock problem. Find out how a reader-writer lock works, and then design a solution for reader-writer locks using the solution we have provided for the reader-writer problem.
 - reader-writer lock problem:
 - writer lock: down(available)
 - writer unlock: up(available)
 - reader lock: the first part of the reader() function we have shown:
 - lock(lock);
 - if (rc == 0)
 - down(available);
 - rc = rc + 1;
 - unlock(lock);
 - reader unlock: the second part of the reader() function:
 - lock(lock);
 - rc = rc - 1;
 - if (rc == 0)
 - up(available);
 - unlock(lock);
2. The reader-writer problem suffers from starvation. Design a solution for avoiding starvation.
 - a. Have a queue that stores reader and writer threads, run them in FIFO order
 - b. Access to queue should be protected by lock

- c. Add synchronization to make sure reader and writer enqueue and dequeue in FIFO order
3. Does the sleeping barber problem suffer from starvation?
 - a. If barber never finishes for one thread, then threads will be coming and leaving (nope)
 - b. customers wait on barber by using down(barber). If the implementations of the down and up operations use queues, then there will be no starvation. Otherwise, starvation is possible.
4. What are the similarities and differences between the sleeping barber and the producer-consumer problems?
 - a. Sleeping barber initializes both semaphore to 0
 - b. producer-consumer has full = 0 and empty = n
 - c. Only 1 consumer in sleeping barber question

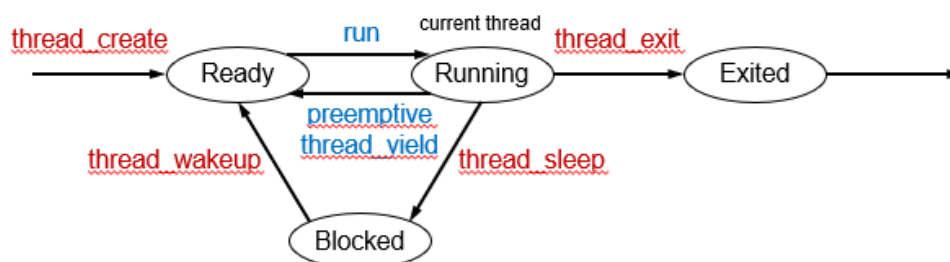
L20 scheduling policies

Overview on thread scheduler

- Implements threads abstraction
- Allows running threads concurrently & in parallel
- With no pre-emption



- With preemption



- OS uses preemptive scheduling
- Scheduler forces current thread to yield under certain condition e.g. thread has been running for sometime
- Good about preemption:
 - 1 thread cannot stop another thread when running
- Bad about preemption:
 - Adding overheads with frequency thread switching

What is the scheduling mechanism (only 1)?

- Thread switching

- Suspend current thread
- Run another thread by resuming the thread

Why does scheduling policy do?

- Choose next thread to run
- Choose when to run
- OS only has 1 scheduling mechanism, but multiple scheduling policy

What are the two types of programs?

- CPU-bound program
 - Frequent CPU bursts
 - E.g. matrix calc
- IO-bound program
 - Frequent IO bursts
 - E.g. web server

Why need to distinguish between these two programs?

- When programs waiting for IO to finish, it is not using CPU
- If scheduler can run CPU, and keep it busy, will help with CPU utilization

What are the two computer systems that are commonly used?

1. Batch system
 - a. Long running
 - b. CPU-bounded jobs
 - c. No interactive users
 - i. No user waiting to get immediate response
 - d. Run program with minimal timing constraints (i.e. don't have a lot of timing constraints)
 - e. E.g. fb analyze user preference through analysing the large amount of data
 - Scheduling goal:
 - CPU utilization, max % of time CPU is busy
 - Max throughput, max number of jobs completed per unit time
 - Min turnaround time: min time needed from start to finish job
 - Turnaround time = processing time (running) + waiting time (not running)
 - Turnaround time = finish time (last) - arrive time
 - Waiting time = start running time - arrive at time (if non-preemptive)
 - Waiting time = turnaround - running time (preemptive)
2. Interactive (general purpose) systems
 - a. Short running
 - b. IO-bounded
 - c. User interactive
 - d. Wait for immediate response
 - e. E.g. typing by pressing keystrokes, move the mouse
 - Scheduling goal:
 - Response time: min time between receiving request and producing response
 - Response time = arrive - start running for first time
 - Good throughput

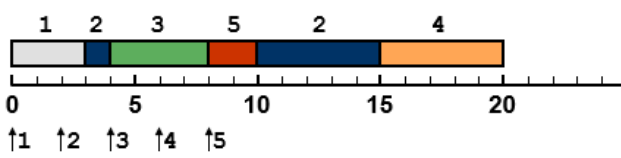
L20 think time

1. Is there a difference between throughput and CPU utilization? If so, why?
 - a. Max Throughput means in a given period of time, max number of job completed. But this can only be using 1% of CPU time
 - b. difference between throughput and cpu utilization: throughput is the number of jobs that complete over time, while CPU utilization is the % of time that the CPU is busy.
 - c. The two are correlated for CPU bound jobs since higher CPU utilization will generally increase throughput. However, this may not always be the case. Consider two schedulers that have 100% CPU utilization, i.e., the CPU is never idle. If one of the schedulers is less efficient than the other, or it performs too many pre-emptions, then its throughput will be lower, since more CPU time is being spent on running the scheduler code (either the scheduler algorithm or the thread switching code).
2. Why does improving (reducing) the response time reduce throughput as well?
 - a. Throughput can be improved by running threads for long period of time → reduce number of preemption & increases cache locality. Low overhead and no need to restore memory back and forth so higher throughput
 - b. Running thread for long period of time increases response time. Best is to run thread as soon as it can run so don't waste time waking up/blocking threads. But this inc preemption and reduces cache locality. So reduce performance

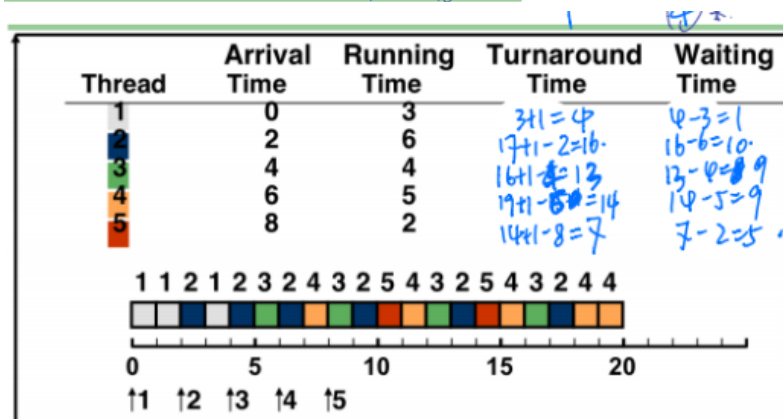
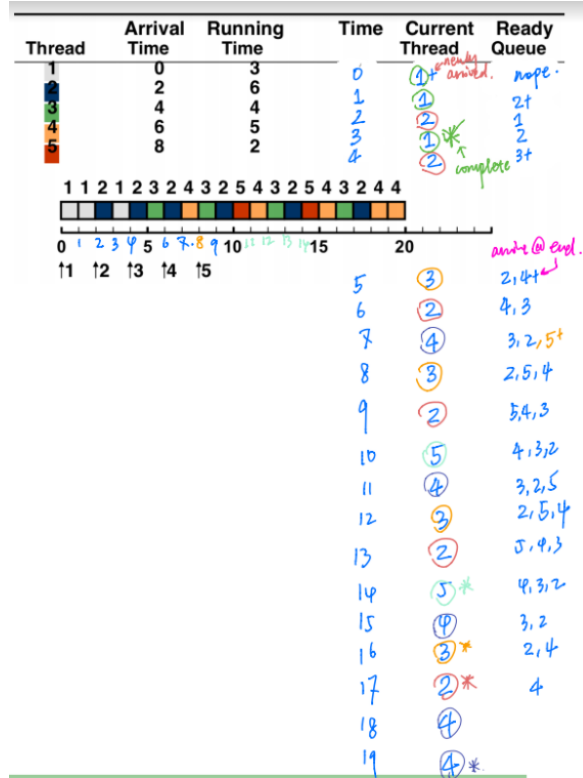
L21, L22, L23 batch and interactive scheduling

	Policy name	Pre-emp?	Know runtime?	Throughput	Wait time	Response time	Fairness
Batch scheduling policies	First-come, first serve FIFO	No	No	highest	longest	Long	yes
	Long response time	Run: select job in order they arrive Stop: run to completion Block: place job at end of ready queue when woken. Each CPU burst served in FIFO order					

	<div>FIFO Metrics</div> <table><thead><tr><th>Job</th><th>Arrival Time</th><th>Running Time</th><th>Turnaround Time</th><th>Waiting Time</th></tr></thead><tbody><tr><td>1</td><td>0</td><td>3</td><td>3</td><td>0</td></tr><tr><td>2</td><td>2</td><td>6</td><td>7</td><td>1</td></tr><tr><td>3</td><td>4</td><td>4</td><td>9</td><td>5</td></tr><tr><td>4</td><td>6</td><td>5</td><td>12</td><td>7</td></tr><tr><td>5</td><td>8</td><td>2</td><td>12</td><td>10</td></tr></tbody></table> <p>□ Average waiting time = $(0 + 1 + 5 + 7 + 10)/5 = 4.6$</p> <p>Turnaround 2 = finish (9) - arrive (2) = 7</p> <p>Waiting 2 = turnaround(7) - running (6) = 1</p>						Job	Arrival Time	Running Time	Turnaround Time	Waiting Time	1	0	3	3	0	2	2	6	7	1	3	4	4	9	5	4	6	5	12	7	5	8	2	12	10
Job	Arrival Time	Running Time	Turnaround Time	Waiting Time																																
1	0	3	3	0																																
2	2	6	7	1																																
3	4	4	9	5																																
4	6	5	12	7																																
5	8	2	12	10																																
Shortest job first SFJ	No	Yes	Highest	Shorter than FIFO	Long	no																														
Starvation Long response time Needs to know time first	<p>Run: after finish one job, select next job with shortest running time</p> <ul style="list-style-type: none">- Order ready queue by shortest running time <p>Stop: run job to completion</p> <p>Block: Put job back to ready queue, sort in correct order (by total running time, shortest first)</p> <div>SJF Metrics</div> <table><thead><tr><th>Job</th><th>Arrival Time</th><th>Running Time</th><th>Turnaround Time</th><th>Waiting Time</th></tr></thead><tbody><tr><td>1</td><td>0</td><td>3</td><td>3</td><td>0</td></tr><tr><td>2</td><td>2</td><td>6</td><td>7</td><td>1</td></tr><tr><td>3</td><td>4</td><td>4</td><td>11</td><td>7</td></tr><tr><td>4</td><td>6</td><td>5</td><td>14</td><td>9</td></tr><tr><td>5</td><td>8</td><td>2</td><td>3</td><td>1</td></tr></tbody></table> <p>□ Average waiting time = $(0 + 1 + 7 + 9 + 1)/5 = 3.6$</p> <p>When T2 finishes, RQ: T5, T3, T4</p> <p>Turnaround 3 = finish (15) - arrive(4) = 11</p> <p>Waittime = turnaround 11 - running 4 = 7</p>						Job	Arrival Time	Running Time	Turnaround Time	Waiting Time	1	0	3	3	0	2	2	6	7	1	3	4	4	11	7	4	6	5	14	9	5	8	2	3	1
Job	Arrival Time	Running Time	Turnaround Time	Waiting Time																																
1	0	3	3	0																																
2	2	6	7	1																																
3	4	4	11	7																																
4	6	5	14	9																																
5	8	2	3	1																																
Shortest remaining time SRT	Yes	Yes	Low bc preempt	Lowest (lower than FIFP	Long	no																														

					and SJF)																																
	Starvation Needs to know time first	Run: select job with shortest remaining time to finish																																			
		- Ready queue in the order of shortest remaining time																																			
		Stop: preempt job if another job with shorter remaining time arrives																																			
		Block: put job back to ready queue, sort by order of remaining time																																			
		<div>SRT Metrics</div>																																			
		<table><thead><tr><th>Job</th><th>Arrival Time</th><th>Running Time</th><th>Turnaround Time</th><th>Waiting Time</th></tr></thead><tbody><tr><td>1</td><td>0</td><td>3</td><td>3</td><td>0</td></tr><tr><td>2</td><td>2</td><td>6</td><td>13</td><td>7</td></tr><tr><td>3</td><td>4</td><td>4</td><td>4</td><td>0</td></tr><tr><td>4</td><td>6</td><td>5</td><td>14</td><td>9</td></tr><tr><td>5</td><td>8</td><td>2</td><td>2</td><td>0</td></tr></tbody></table> <div></div>						Job	Arrival Time	Running Time	Turnaround Time	Waiting Time	1	0	3	3	0	2	2	6	13	7	3	4	4	4	0	4	6	5	14	9	5	8	2	2	0
Job	Arrival Time	Running Time	Turnaround Time	Waiting Time																																	
1	0	3	3	0																																	
2	2	6	13	7																																	
3	4	4	4	0																																	
4	6	5	14	9																																	
5	8	2	2	0																																	
		<div><div>□ Average waiting time = $(0 + 7 + 0 + 9 + 0)/5 = 3.2$</div><div>□ Provably optimal <u>w.r.t.</u> average wait time</div><div>□ Number of preemptions = 1</div></div> <div>When T3 arrives, its remaining time = 4, current thread T2 remaining time = 5, so preempt once</div> <div>At time=9, running 5, RQ: 2,4</div> <div>Turnaround 2 = finish time 15 - arrive time 2 = 13</div> <div>Waiting 2 = turnaround 13 - running 6 = 7</div>																																			
		Problem with batch policies																																			
		<div>1. Long response time</div> <div>a. SJF, FIFO runs to completion</div> <div>2. Needs estimate of processing time</div> <div>a. SJF and SRT needs this</div> <div>b. Impossible for IO devices</div> <div>3. Starve long running jobs</div> <div>a. SJR and SRT do not prefer these, may be waiting forever</div>																																			
Interactive scheduling policies	Round robinhood RR	Yes	No	Lowest, lower than SRT	Longest, longer than FIFO	short	Yes																														
	Long waiting time	Run: select threads in FIFO order																																			
		Stop: preempt thread when time slice expires																																			
		<div>- Time interrupt after each slice</div> <div>- Each thread runs for at most 1 slice before another thread runs</div> <div>- Context switch here and thus run next thread at front of</div>																																			

- ready queue
- How to choose time slice?
- Response time of a thread just come in = $TS * \# \text{ ready threads}$
 - Shorter TS better
 - Context switch overhead = $CS / (CS + TS)$
 - Larger TS, smaller overhead, better
 - Usually: $CS = 10\mu s$, $TS = 1ms$, overhead = 1%



Total 13 preemptions

Turnaround time 2 = last run finish time 18 - first arrive 2 = 16

Wait 2 = 16 - 6 = 10

Avg weight time = 6.8, higher than FIFO

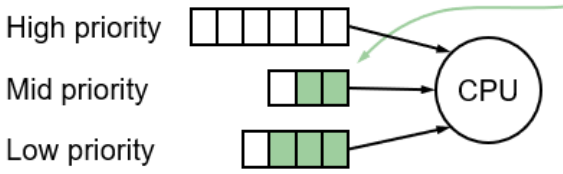
- Could make system unresponsive

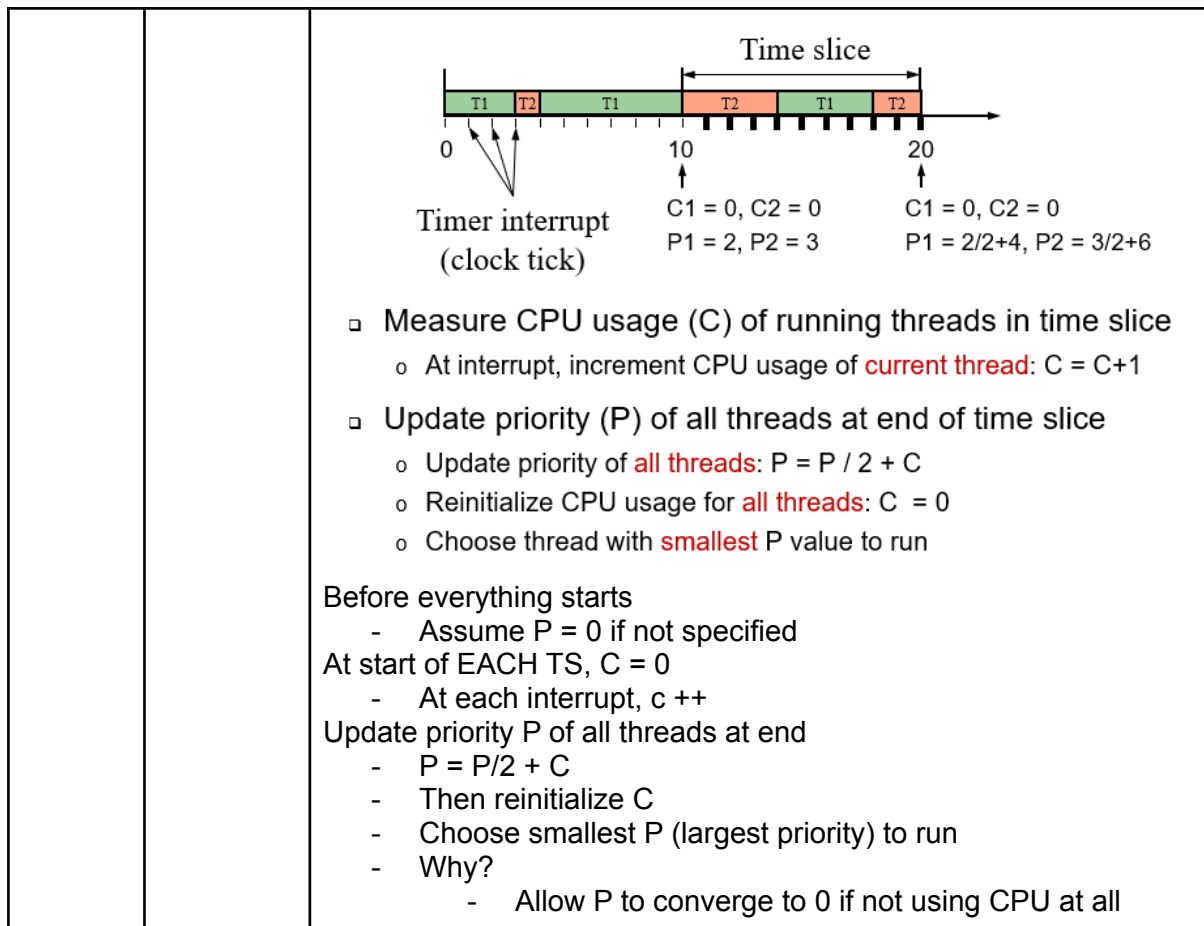
Static
priority

starvation

When started: each thread assigned with a priority

Run: select in priority order

	n	Stop: preempt thread if another thread with higher priority arrives (similar to SRT)				
	Multi-level scheduling	Static priority scheduling + round robinhood <ul style="list-style-type: none"> - Maintains multiple ready queues, 1/priority - Queues ordered from highest priority to lowest - RR scheduling within each queue <div style="text-align: center;">  </div> <p>IO usually has higher priority than CPU</p> <ul style="list-style-type: none"> - Gives good response time to IO bounded threads 				
	Dynamic priority (unix feedback) scheduling	Pre- Emp? Yes	Know runtime? No	Through put Good	Wait time Small for IO	Response time Small for IO
		<p>At start: no priority specification needed Dynamically does 3 things:</p> <ol style="list-style-type: none"> 1. Measure CPU usage over time 2. Raise priority of threads that do not use a lot of CPU <ol style="list-style-type: none"> a. Good response time to IO threads 3. Lower priority of threads if use CPU, and give them long time slices <ol style="list-style-type: none"> a. Avoids starvation b. Good throughput to CPU threads <p>Run: run thread in priority order</p> <ul style="list-style-type: none"> - Multiple threads running within a time slice: <ul style="list-style-type: none"> - A thread with higher priority wakes up, immediately preempt current and run that - Otherwise current thread run for entire time slice and wait for update to see which runs next - CPU bounds will run in round-robin order <p>Stop: preempt thread if another thread with higher priority arrives After each TS: update thread priority</p> <ol style="list-style-type: none"> 1. Measure CPU usage within a time slice 2. Update priority of ALL threads at the end of time slice 				



Does Round-robin fix batch problems?

- Long response time?
 - Preempts threads after time slice and run in round-robin order, ensure all jobs make progress, so short response time
- Starvation?
 - Has fairness
 - No starvation
- Estimation of processing time?
 - No need

What are problems with round-robin scheduling?

- Its response time depends on TS
 - Small slice large overhead
 - Large slice long response time
- Not provide good response time to IO and good throughput to CPU bounded threads
 - Because it treats all threads in the same way, despite the fact that IO and CPU bounded program have different requirements
 - Not providing priority to any thread

Problem with static priority scheduling

- Hard to use
 - Priority is often specified by user/programmer
 - Hard to determine a priority value because it dep on other programs running in the system as well
- May cause **starvation**

- a. Incorrect choice → run program within loop for only high priority programs, leading to **starvation**
- b. Starvation occurs when CPU-bounded thread is run at high priority }+ scheduler chooses not to run threads with low priority

How does feedback scheduler avoid starvation?

- Threads with lot of CPU work have lower order in time, other threads can run

Wh is feedback efficient?

- Long time slice for CPU bound program, so low switch overhead, thus shorter response time!

L21 think time

1. What happens when a job blocks under the SJF and SRT policies?
 - a. SJF: Put job back to ready queue, sort in correct order (by total running time, shortest first)
 - b. SRT: put job back to ready queue, sort by order of remaining time
2. The SJF and SRT schedulers require an estimate of the job's running time. What if the estimates of these running times are wrong, either
 - Too long:
 - CPU not running for a long time, job delay unnecessarily
 - Too short?
 - Program being cut off while not finish its task
 - Scheduler preempt it and arbitrarily assigns it a long runtime to delay it so jobs do not try to cheat

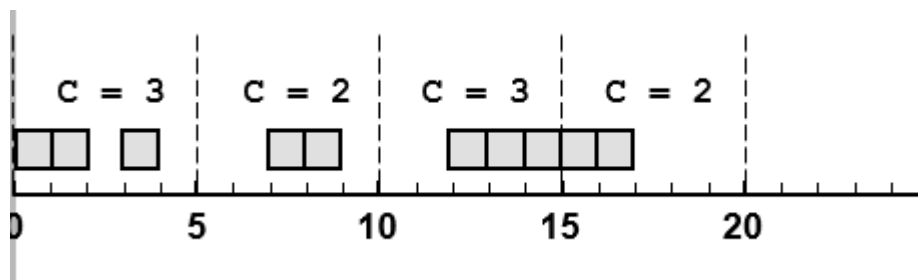
L22 think time

1. Use the "man sched" command to find out the default time slice period of the round-robin scheduler in Linux
 - 100ms (interrupt period = 4ms)

L23 think time

1. Static priority schedulers can suffer from priority inversion. What is priority inversion?
 - priority inversion: say we have three thread with high priority (H), middle priority (M), and low priority (L). Say L acquires lock(a), and then H tries to acquire lock(a). H will block, and so M gets to run, since it has higher priority than L. However, L will not get to run, and so it can't release its lock, and so H doesn't get to run. In essence, a lower priority thread (M) is running, while preventing a higher priority thread (H) from running, which is why it is called a priority inversion.
 -
 - H: lock(a)
 - M: running, so L doesn't get to run, which blocks H.
 - L: lock(a)
2. Feedback scheduling require a timer interrupt for estimating CPU usage. What is the timer interrupt period in Linux?
 - a. 4ms

3. Run the “top” program. On the top right, you will see three numbers for the load average. How do you think the OS calculates these values?
 - a.
4. In Unix, users can change thread priorities by using the “nice” value. How does this nice value work?
 - a.
5. Say 10 timer interrupts occur in a time slice, and a thread takes 30% of the CPU in each time slice. With the Unix feedback scheduler, what will its priority value be over time?
 - a. $P = 0, P = 3, P = 4.5, P = 5.25$
 - b. Stabilize to 6
6. Say time slice of the scheduler is 5 timer interrupt units
7. A thread runs as follows:



- 8.
- 9.
10. Assume the initial priority value of this thread is 0
11. The priority of the thread P is updated as: $P = P/2 + C$
12. Calculate its priority at time 20
13. $P_1 = C_1$
14. $P_2 = C_1/2 + C_2$
15. $P_3 = (C_1/2 + C_2)/2 + C_3 = C_1/4 + C_2/2 + C_3$
16. $P_4 = (C_1/4 + C_2/2 + C_3)/2 + C_4 = C_1/8 + C_2/4 + C_3/2 + C_4 = 3/8 + 2/4 + 3/2 + 2$
- 17.
18. so the pattern is $P_4 = C_4 + C_3/2 + C_2/4 + C_1/8$ (each preceding time slice's CPU usage is progressively halved, so the CPU usage of a long time back does not affect P .)

L24 Unix Process

fork()

- Create children that has exact same everything as parent
- Each modify data in their own address space

How to distinguish parents from children?

- Parent $ret \neq 0$, child $ret = 0$

execve(“program”)

- Starts running “program”

Why is code after execve never run?

- New program will overwrite the current program

exit(ret_val)

- Ret_val is made available to parent processes

wait(child_pid)

- Parent wait for a child process to exit

What if child exits before parent issue wait?

- Child holds on to ret_value until wait is issued by parent
- Parent wait, then child exit, parent continue, child exit done WECD
- If child exits happens first, it is put into a zombie state until parent issues wait, parent continues, and then child exit done EWCD

kill(pid, sign_number)

- Acts like an interrupt
- Interrupt execution
- Used to send signal to another/itself

sigaction()

- Signal handler
- If no handler set, then will force process receiving a signal to exit

L24 Think time

1. What happens when the various Unix system calls we have discussed fail (due to some error)?
 - Return value is -1
 - Variable errno indicating the cause of error
2. Unix provides the getppid() call to get the parent's process id. Why is this call required?
 - Fork ret=0 for child
3. Why does Unix not provide a getcpid() call to get the child's process id?
 - A process can have multiple children so this is ambiguous
 - Child's PID is found on a fork where the parent ret = child's PID
4. Why does fork have the weird semantics of creating a replica process?
 - a. Parent can setup any state needed to be shared with child before the call
 - b. After the call, p and c can modify variable separately
5. Why do we need a separate fork and execve – aren't they always needed together to run a new program?
 - a. A program can have multiple processes, so when fork we can create difference processes but do not need to run another executable file
6. A Unix program typically returns 0 when it is successful, and a non-zero value when it fails for some reason. Why does this choice make sense?
 - a. Diff ret value for diff failure
7. Say a program leaks memory. When does the OS get this memory back?
 - a. When program exits, program address space is destroyed
8. When a program exits, when is its thread state destroyed?
 - a. When exit is called, OS is still executing context of current thread, so thread state is not destroyed
 - b. Is destroyed when next thread starts running
 - c. May be kept even longer, until parent issues wait
9. A parent calls wait(child_pid) to wait for a child process to exit. How does parent know child_pid?
 - a. Ret from fork
10. Can a Unix process call wait on a process that is not its child?

- a. No. should be descendant process
- 11. In Lab 3, we allow any thread to invoke wait on any other thread. Could this approach cause a problem?
 - a. Yes deadlock
 - b. A and B wait for each other
- 12. We discussed how the parent can issue wait either before or after the child's exit. What happens if the parent never issues wait?
 - a. If child already exit, child is doing child_wait synchronization so when parent exit the child will know parent is done
 - b. Parent exited before child exits, then no synch will needed, child will be done immediately
- 13. What are the steps involved in sending and receiving signals?
 - Sender -- kill system call → another process
 - Receiver registers a signal handler using sigaction()
 - When signal is received, signal handler invoked
- 14. When a signal is received and the recipient process has not registered a signal handler, the recipient process is killed. Is the recipient process terminated immediately?
 - a. No
 - b. It exits when it runs next
 - c. Because: race problem on locks
 - d. Is the recipient process terminated immediately? The recipient process is not terminated immediately. Instead, it exits when it runs next. The reason is as follows: a process exits by calling thread_exit on itself, so thread_exit assumes that the calling thread is exiting. So a thread cannot directly call thread_exit to force another thread to exit. One reason that thread_exit is not invoked on other threads directly is that a thread R (receiver) might be holding a lock, and on exit, it will release the lock. however, unlock may assume that the thread that acquired the lock (thread R) will release it. If thread S (sender) called thread_exit on thread R, then the unlock would be performed by thread S, so it will appear that thread R acquired the lock, while thread S is releasing it. Similarly, on a multi-processor, thread R might be running when thread S calls exit on it from another processor. Unless proper synchronization is done, this could cause race problems. The moral is that it is best to die by oneself!

L25 POSIX threads

POSIX thread: standard thread API

pthread_create(thread, attr, **start_routine**, arg)

- Returns new thread ID in thread
- Executes function specified by **start_routine**(arg)

pthread_exit(status)

- Terminates current thread, returns **status** to a joining thread
- Joining thread is the tread that is waiting for current thread to exit

pthread_join(**thread_id**, **status**)

- Blocks thread until thread specified by **thread_id** terminates
- Return value from pthread_exit is passed in **status**

pthread_yield()

- Thread gives up processor

POSIX threads mutex API

```
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_mutex_lock(&mut);
```

```
pthread_mutex_unlock(&mut);
```

- Locks are in OS kernel not user
- Good: when lock is called, will be blocked, no spinning occurs

POSIX threads synchronization API

- Semaphore

- sem_t sem_name;

- sem_init(&sem_name, 0, 0); /* 2nd arg is flag, 3rd arg is init value */

- sem_wait(&sem_name); /* down operation */

- sem_post(&sem); /* up operations */

- Condition variables

- pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

- pthread_cond_wait(&cond, &mut); /* mut is a lock */

- pthread_cond_signal(&cond);

POSIX monitor

- Say a thread wishes to wait until $x > y$

```
int x,y; // shared variables
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&mut);
while (x <= y) {
    pthread_cond_wait(&cond, &mut);
}
/* operate on x and y */
pthread_mutex_unlock(&mut);
```

- Another thread signals when $x > y$

```
pthread_mutex_lock(&mut);
/* modify x and y */
if (x > y) pthread_cond_signal(&cond);
pthread_mutex_unlock(&mut);
```

L26 Memory Management Overview

3 requirements:

- Isolation
- Sharing
- Abstraction

2 goals:

- Min memory overhead
 - Mem used to manage memory
- Min performance overhead

- CPU used to manage memory

2 techniques used to manage memory

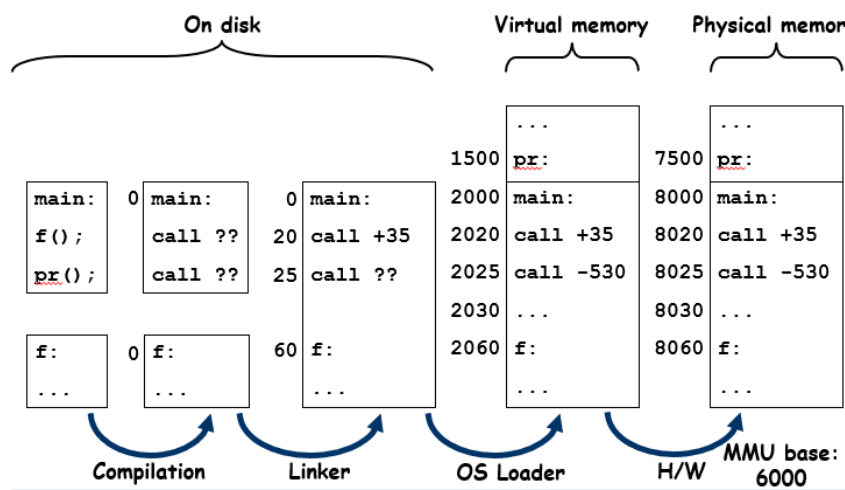
- Bitmaps
 - A long string, 1 bit to keep track of each chunk of memory
 - 1 = in use, 0 = free
 - Size of chunk \rightarrow size of bitmap
 - 32 bits \rightarrow overhead = $1/(32+1)$
 - 4KB = $4*1024$ bytes = $4*1024*8$ bits $\rightarrow 1/(4*1024*8+1)$
 - Larger chunk:
 - Smaller overhead, waste more memory, has internal fragmentation
 - Internal frag: allocated region of memory is not fully used
 - Operations
 - mem = allocate(K)
 - Search bitmap for K consecutive 0 bits
 - Set the K bits to 1
 - Cost: linear search
 - free(mem, K)
 - Determine starting bit in bitmap based on memory address
 - Set next K bits to 0
 - Problem: hard to find K consecutive bits \rightarrow external fragmentation
 - External frag: holes between allocated regions are fragmented, so large memory allocations are not possible, without compacting allocated regions
- Linked lists
 - mem = allocate(K)
 - Search linked list for a hole(free region) with size $\geq K$
 - Size $> K$, break hole to allocated region and turn into smaller hole
 - free(mem, K)
 - Determine region based on memory address
 - Convert allocated region into hole
 - If has whole around, merge with them
 - Searching through linked list:
 - First Fit: Start searching at the beginning of the list
 - Best Fit: Find the smallest hole that will work
 - Tends to create lots of little holes
 - Quick Fit: Keep separate lists for common sizes
 - Efficient but more complicated implementation
 - Problem: hard to find hole size $> K \rightarrow$ external fragmentation

Simple memory management

- Require continuous region in physical memory
 - Region size == estimate of program size
 - OS takes up a region
- Problems:
 1. Internal fragment: program not using entire region allocated
 2. External fragment: region not allocated for programs may not be large enough
 3. Hard to grow: need to copy entire library, expensive

Mapping from program variable to memory address:

- Compiler
 - Converts program source file to object file
 - Create RELOCATABLE virtual memory address
- Linker
 - Links multiple object related to each other (to a single program) on disk
 - Generate ABSOLUTE virtual memory address
- OS
 - Loads program and dynamic libraries to physical memory
 - Can reallocate address of some programs
 - Links program codes with dynamic lib code
 - A separated library that all programs can use
 - When loaded by OS, OS does patching for each single program to access the library
- Hardware support
 - MMU translating Vaddr (from CPU) to Paddr in REAL time



- "Call" + 35 means next execution will happen at 25 (location of exe right after this call) + 35
- Compilation: no address is known
- Linker: relative address of each internal variable is known, external var addr NA
- OS loader: internal addr in OS is adjusted, but still virtual address; external var addr is known, linked to shared library addr in OS, also virtual
- MMU base: translates to phy addr for both internal and external var

L26 Think time

- What is the difference between a virtual and a physical memory address?
 - virtual : addr seen by program, CPU
 - physical : addr seen by memory
- Why is hardware support required for address translation?
 - Software: too slow
- How does the OS manage memory using bitmaps? using lists? Under which conditions is each approach preferable?
 -
- What is internal and external fragmentation?

- Linked list:
- pros: Small space requirements, as each block can store a pointer to the next free block.
- cons: To traverse the list, you need to read each block! Also, it is costly to maintain the list in a "contiguous" manner, in order to avoid fragmentation (think about the cost of updating the list in a smarter way than just appending each new free block at the end).
- Bitmap:
- pros: Random allocation: checking if a block is free only requires reading the corresponding bit; also, checking for a large continuous section is relatively fast (as you can check a word size of bits from the bitmap in a single read). Fast deletion: you can just flip a bit to "free" a block, without overwriting the data.
- cons: Higher memory requirements, as you need one bit per block (~128MB for a 1TB disk with 1KB blocks).
- If disk is almost full: use linked list, if need faster traverse: use bitmap

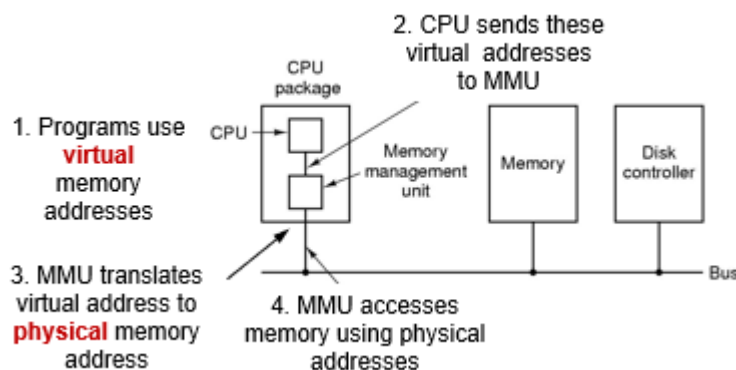
L27 Virtual Memory Hardware

Non-contiguous memory

- Can solve issues in simple memory management scheme
- Should hide from programs

Memory management Hardware MMU

- Translate Vaddr to Paddr so
- Programs use contiguous
- Physical memory is allocated non-continuously



2 base-register MMU

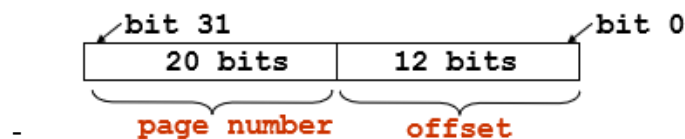
- Base register (Paddr)
- Limit register (length is the same for P and V)
- Given $0 \leq V < L1$: $P = V + B1$
- Given $L1 \leq V < L2$: $P = V - L1 + B2$
- Problem:
 - Only support 2 phy mem region
 - Translation is slow, N reg, $O(N)$ time

Paging MMU

- Supports non-contiguous phy mem regions that are large
- Partitions virtual address space to chunks "page"
- Partitions physical address space to chunks "frame"
 - Page size = 2^n bytes, usually 2^{12} bytes = 4KB
 - Frame size == page size
 - Frame = map(page)

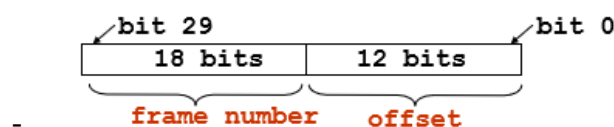
Virtual address

- $V_{addr} = (\text{page number} \mid \text{offset})$
- Page size = 2^{12} , offset == 12bits
- In 32 bit machine, page number = 32 - offset = 20
- Number of pages = 2^{20}



Physical address

- $P_{addr} = (\text{page number} \mid \text{offset})$
- Frame size = page size = 2^{12} , offset == 12bits
- In 30 bit machine, page number = 30 - offset = 18
 - Address space size can be diff from V_{addr}
- Number of frames = 2^{18}

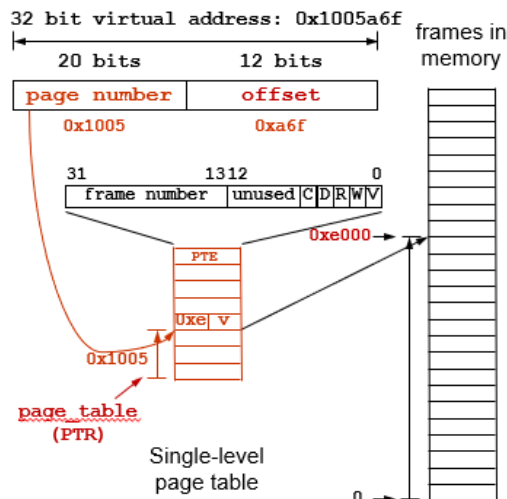


Benefits of paging:

Contiguous Memory Allocation	Paging
Growing a program requires copying entire program	Growing a program requires allocating a page at a time
Wastes memory due to internal and external fragmentation	No external fragmentation, internal fragmentation is $\frac{1}{2}$ page per region
Running a program requires loading entire program in memory	As program runs, pages can be loaded in memory (we will see this later)
Maximum program size is limited by memory size	Maximum program size is limited by disk size (we will see this later)

Page table

- Used to maintain mapping information of Paging MMU
- **1 page table has 1 page table entries PTE**, maps 1 page to 1 frame
 - Each entry = 1 word, 32 bits architecture, each entry = 32 bits = 4 bytes
- Each entry has frame number and bits \rightarrow valid, writable dirty...



- When given a Vaddr, 32 bits, page number | offset
- Find the page number and offset in the PT. In this entry of PT, it stores a Paddr, 32 bits (maybe), frame number | V. Links Vaddr to Paddr by:
 - vaddr = 0x01005a6f
 - offset = vaddr & 0xfff = 0xa6f (offset is the same for page and frame)
 - page = vaddr >> 12 = 0x1005 (get the corresponding page number)
 - fr = page_table[page].frame = 0xe (get the frame number)
 - paddr = (fr << 12) | offset = 0xea6f (offset is the same, so concat back)
- V has bits indicating valid, writable, dirty...

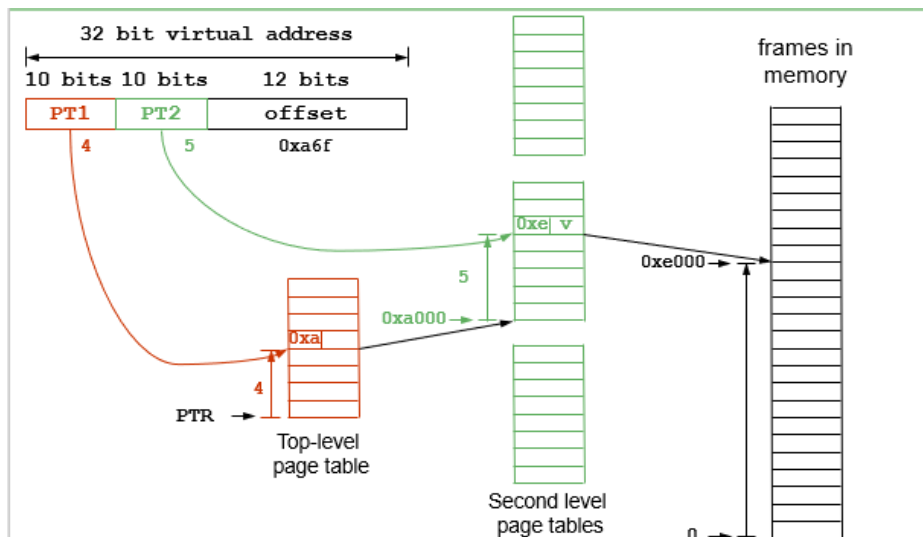
How does MMU find page tables

- MMU knows the location of page table in physical memory using page table register PTR
 - Stores location of start of **page table**
- 1 PT per process
- Context switch = thread switch + address space switch (done by changing PTR)

Calculate size of page table

- Number of PTE == Number of pages = Vaddr space size / page size = $2^{32}/2^{12} = 2^{20}$
- PTE size = 1 word size = 32 bits = 4 bytes
- Page table size = PTE size * #PTE = 4 bytes * $2^{20} = 4\text{MB}$
- Larger page size → smaller number of PTE, more internal fragmentation

Multi-level page table



- 1 top level page table, can have 0 or more second level page tables
- 2^{10} first level, each can point to 2^{10} second level table. Total $2^{10} + 2^{10} \times 2^{10}$
- A second level PT is not allocated when empty, so no need to take up space (a NULL pointer pointing toward it)
- Translation

<code>vaddr = 0x1005a6f</code>	
<code>offset = vaddr & 0xfff = 0xa6f</code>	get 12 low bits of <code>vaddr</code>
<code>pg2 = (vaddr >> 12) & 0x3ff = 0x5</code>	get next 10 bits of <code>vaddr</code>
<code>pg1 = vaddr >> (12+10) = 0x4</code>	get top 10 bits of <code>vaddr</code>
<code>pt_1 = page_table_register</code>	base of top page table
<code>pt_2 = pt_1[pg1].frame << 12 = 0xa000</code>	look up top page table to find physical memory location of second level page table. <code>pt_2</code> is stored page aligned
<code>fr = pt_2[pg2].frame = 0xe</code>	look up second page table
<code>paddr = (fr << 12) offset = 0xea6f</code>	generate physical address

Single vs multiple level:

Single level is faster

Multiple level requires 2 look ups for 2 levels, but saves space

Problem with page table:

- Large and takes up space
- MMU needs to access page table on every instruction, so 2 access in total
- Slow!
- Solution: cache PTE in MMU with TLB, handle misses

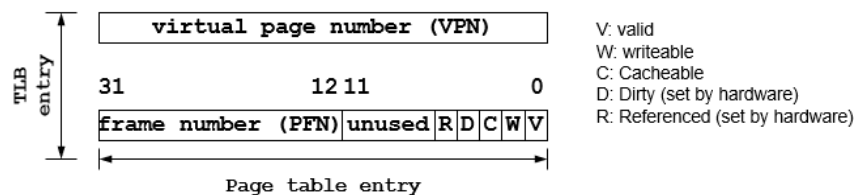
L27 think time

1. What is the purpose of a page table?
 - a. Store Vadd to Padd mapping
2. Where is the page table located?
 - a. In physical memory
3. How does the h/w locate the page table?
 - a. With PTR pointing to the start (bottom) of a PT. PTR holds physical address of PT
4. How many address bits are used for page offset when page size is 2KB?
 - a. Page size = 2KB $\rightarrow 2 \times 2^{10} = 2^{11}$, so 11 bits in offset
5. Discuss advantages and disadvantages of linear and multi-level page tables

L28 Translation Lookaside Buffer TLB

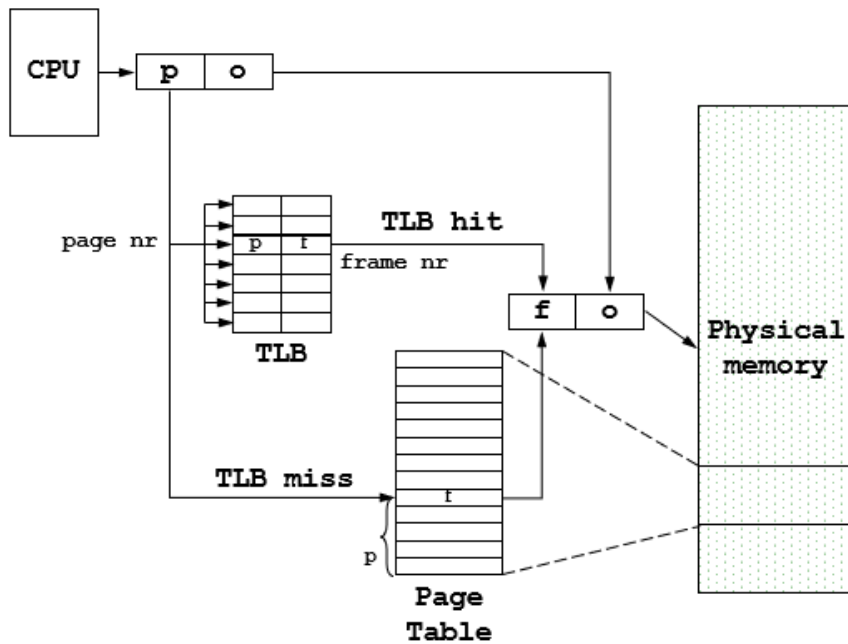
Each TLB entry:

- Key: page number
- Data: page table entry



-
- VPN: 32 bits, for look up to see if an address is here (Page number | Offset)
- + PTE
- In PTE: (PFN: frame number)|(V)
- In V: R: reference (accessed recently), D: dirty, C: cacheable (if not, processor should bypass the cache when accessing, for memory-mapped device register), W: writable (if not, then read only), V: valid

TLB look up



TLB cache miss handling

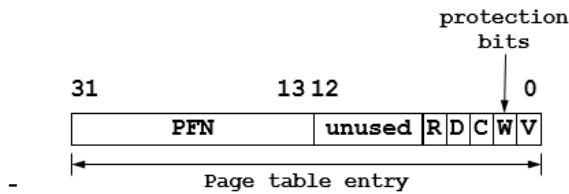
- Happens when TLB lookup fails == VPN not found in TLB cache ++ TLB cache is filled with this new VPN
 - By hw or OS
 - Hw: use PTR to locate PT
 - OS:
 1. HW generates trap called TLB miss fault
 2. OS handles TLB miss similar to handling exception
 - a. Find correct PTE, add to TLB (CPU instruction)
 3. Restart the execution
 - HW has no PTR, replacement policy done in software
 - If TLB is full, choose which to fill using TLB replacement policy

TLB cache invalidate

- TLB == cache of PTE, so need to be consistent with PT
- When OS modifies a PTE, it should invalidate TLB entry
- When a context switch is done, also must invalidate TLB to prevent use of mapping from last address space
 - Adds cost because all instructions coming in now will get a miss
- solution:
 1. Clear TLB
 - a. Empty TLB by clearing the valid bit
 - b. Start caching all entries for next thread
 2. Tagged TLB
 - a. Hardware maintain current process ID in a register
 - i. When context switch happens, OS updates the reg
 - b. Each TLB entry has an ID tag
 - i. On TLB fill, assign current ID tag
 - ii. TLB lookup: only hits when ID tag matches
 - c. Enables space multiplexing + reduces need for invalidation

TLB also helps with memory protection

- Different region, different protection
- MMU is used to implement page-level protection with protection bits on PTE
- Protection fault generated when read-only page is written



- TLB also caches page-level protection bits, generate protection fault when access inconsistent with protection bit
- TLB entry should be invalidate when page protection is changed

TLB faults

1. TLB miss fault
 - No matching **page number** found
 - a. Read fault/write: want to read/write but entry not matched
2. TLB protection fault
 - a. Read-only fault: want to write but write-bit is not set
 - b. No-execute fault: execution of an instruction in a page was attempted but execute-bit not set

L28 think time

1. What is the purpose of a TLB?
 - a. Speed up address translation by keeping cache of V->P mapping
2. What are the benefits of using a hardware managed TLB?
 - a. TLB miss handling is fast
3. What are the benefits of a software managed TLB?
 - a. More flexible, can have different TLB replacement policy

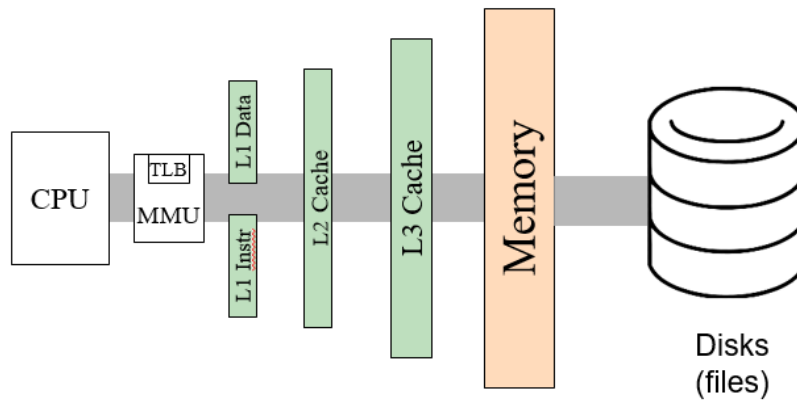
L29 Demand Paging

Problem with paging

- Allows access to non-contiguous memory, but entire process needs to be in memory
 1. Number of program
 2. Max size of program limited by PHYSICAL memory

Demand paging's benefit

- Can run programs that are only partially in physical memory



-
- Memory becomes a cache for data on disk.
 - Cache hit: program accesses page in memory
 - Cache miss: program accesses page NOT in memory, so pages are loaded from DISK to memory on demand
 - Cache eviction: pages in memory infrequently used are moved to disk in a region called swap

How to detect a cache miss == know a page needs to be loaded from disk to memory?

1. PTE valid bit == **invalid**
 - a. Means this page is NOT associated with a frame
2. A cache miss occurs when try to access this page
 - a. MMU generates **page fault exception**, OS page fault controller starts handling
3. OS page fault controller performs **miss handling**
 - a. Here, a free frame taken, allocate frame to this page, update PTE with frame number, change valid bit to valid, restart instruction

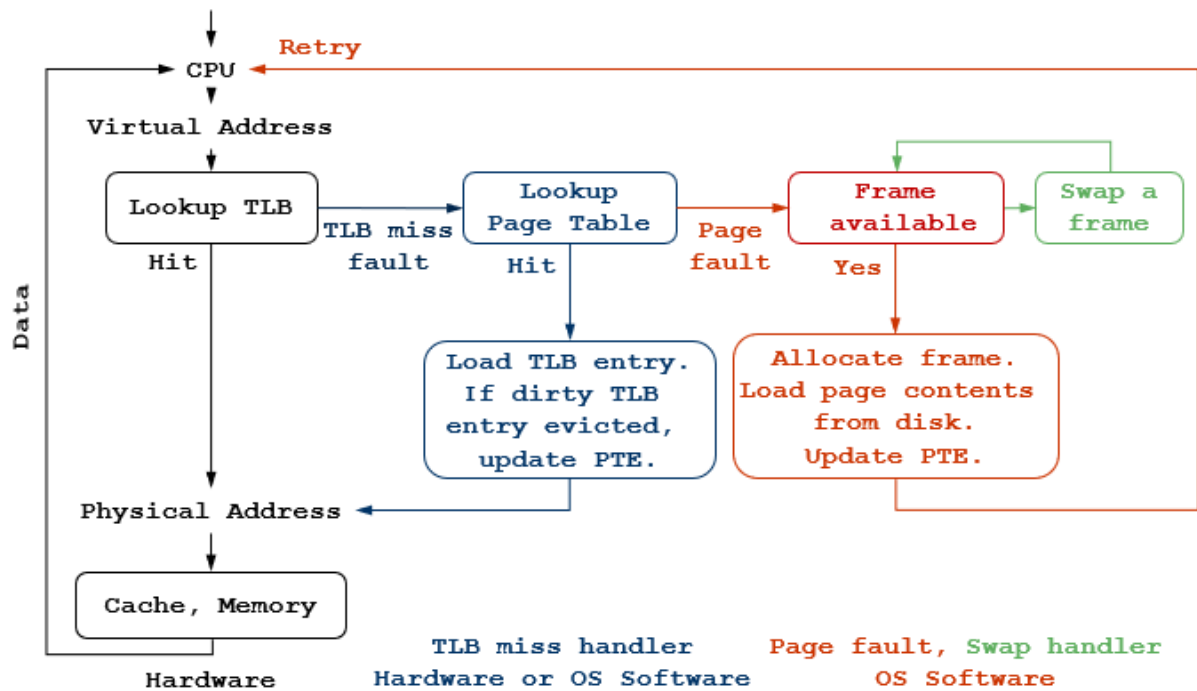
Properties of demand paging:

1. Transparent to application, similar to interrupt handling
2. Works efficiently with locality

Benefits of demand paging:

1. Program with total mem > available mem can be run
2. Faster startup
3. Uses less memory

Virtual memory hierarchy



- Thread generates Vaddr
- 1. Lookup TLB with Vaddr
 - a. TLB Hit: find Paddr directly, get data, go back
 - b. TLB Miss fault (TLB miss handler, HW/OS):
- 2. Lookup Page table:
 - a. Page hit (PTE valid): load data using page number and update TLB entry, if TLB full, choose one to evict. If evicted is dirty, update PTE
 - b. Page fault (PTE invalid) (page fault handler, OS)
- 3. Check if frame is available
 - a. Available: allocate frame, load page content from disk, update PTE to match frame to page and make valid, get data, ask CPU to retry the whole process
 - b. No: swap a frame (swap handler)
- 4. Swap handler evict a page to disk and return a freed frame
 - Which page? Depends on replacement algorithm
 - If the evicted page is dirty, write it to a free location on swap
 - When freeing a frame:
 - Find all pages mapped to the frame
 - Change PTE to invalid
 - Track where frame is in swap, record on PTE

L29 think time

1. Why and when does hardware access page tables?
 - a. If hw is managing TLB, hw access table on TLB miss. This is faster
2. Why and when does OS software access page tables?
 - a. If software managing TLB, access when TLB miss fault, reads page table
 - b. On page fault, allocate frames and update page table with new frame

- c. Current address space modified (frame reallocated, context switch), needs to update PT
- 3. Describe what a page fault handler does
 - a. Checks for errors: seg fault, protection fault
 - b. Allocates frame
 - c. Loads data from disk to frame
 - d. Maps page to frame by updating PTE
- 4. How does virtual memory allow running programs larger than physical memory?
 - a. Some portion of program can be put in disk.

L30 virtual memory implementation

3 abstractions for managing hw resources (paging hardware MMU, physical memory, disk):

- 1. Per-process virtual address space management
 - a. 1 process / address space
 - b. Address space has:
 - i. Partitioned into pages
 - ii. Contiguous memory
 - iii. multiple regions, each has different info
 - 1. contiguous
 - 2. No overlaps
 - 3. Each has a protection bit
 - 4. Require different page fault handling
 - c. Different page fault handling
 - i. Execution instruction
 - 1. Code not in phy memory
 - 2. OS goes to exe file, copy it to a new frame, restart
 - ii. Heap:
 - 1. Address is on heap, but invalid
 - 2. S goes to disk, find free frame, update PTE, restart
 - iii. Outside of stack
 - 1. Automatic stack growth
 - iv. Outside of heap
 - 1. Get segmentation fault
 - 2. If malloc runs out of memory, send sys call to OS, explicitly get mem
 - d. Address space API
 - id = as_create()
 - Create empty address space, allocate new page table associated with id, with no pages mapped to frames
 - as_destroy(id)
 - Destroy address space id
 - Free page table associated with id
 - as_define_region(id, ...)
 - Add a new region to an address space
 - as_modify_region(id, ...)
 - Change the size of a region

- This will be needed to implement heap and stack regions
- as_find_region(id, vaddr)
 - Find a region, given a virtual address
- as_load_page(id, ...)
 - Load a page in memory from file (demand paging)
 - (if get fault here, means data is on disk)
- new_id = as_copy(id)
 - Create a full copy of the address space and the page table with same mappings
 - (all paging mapped to the same frame)
- as_switch(id)
 - Switch the address space to id. Mappings in the TLB from the previous address space must be removed.

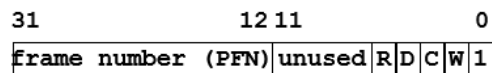
2. Physical memory management

- a. Coremap used to track usage of frames
 - i. Array of structure, 1 struct / frame
 1. Each struct:
 - a. Is a frame free?
 - b. Track pages (all) mapped to the frame
 - i. Helps with swapping
- b. Coremap API
 - frame = allocate_frame(n)
 - Allocate n free contiguous memory frames, returns physical address of first frame
 - free_frame(frame)
 - Free the block of frames, starting at frame
 - map(id, page_nr, frame)
 - Maps frame to virtual address (page_nr) in address space id
 - Allocate and mark page table entry as valid
 - Note that a frame may be mapped to multiple pages in the same or in different address spaces (these are called shared pages)
 - unmap(id, page_nr)
 - Removes frame corresponding to page_nr from address space
 - evict(frame)
 - Used by swap handler to unmap **all pages** associated with frame

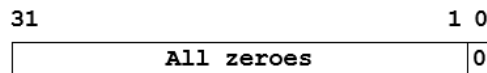
3. Swap management

- Manages swap area
- Dirty pages need to be placed in swap area
- Techniques:
 - bitmap/linked list, keep track of swap frames
- When evicting a page to swap
 1. Find a free swap frame
 2. Write the page to this location on disk
 3. Set PTE to invalid on location of swap frame so handler can use PTE to locate the frame in swap area

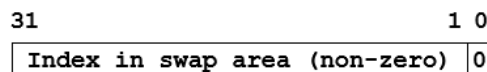
- a. Ensure to update ALL pages associated with this frame
4. Next time when the page is swapped out, it may be written elsewhere



Page Table Entry (PTE) when virtual page is in memory



Invalid PTE (never allocated)



PTE when virtual page is on swap

-
- Use PTE to record the index of page in swap area

L30 think time

1. How is managing swap similar to managing memory
 - a. Both uses linked list/bitmap
 - b. Both needs to keep track of used and free frames

L31 process and virtual memory

OS invokes virtual memory system when address space of the current process changes with command:

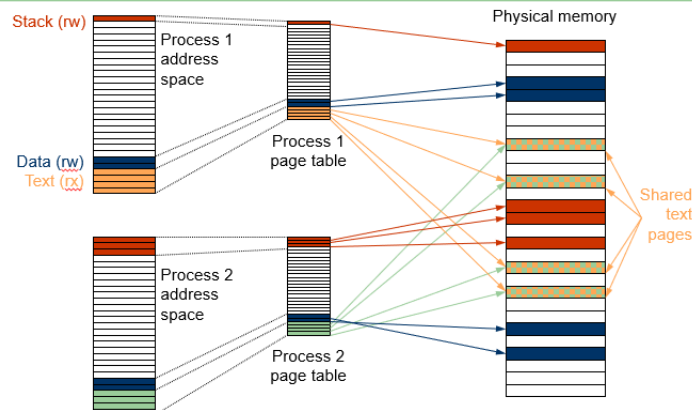
1. Process creation (fork)
 - a. Copies parent address space structure
 - i. Everything is the same, including the V address space, size, permission
 - b. Creates a new page table
 - i. Allocate a copy of memory frames for all regions in address space
 - ii. Create valid PTE for each allocated frames
 - iii. If parent's page is swapped, also make a copy of swapped page
2. Process execution (execv)
 - a. Destroy old address space structure
 - i. Free all pages in PT&swap region
 - ii. Free space used by address space, PT
 - iii. Check reference counts on coremap
 1. Frames can only be freed if all pages mapped to it == reference count == 0
 - b. Create new address space
 - i. Size text and code region based on exe
 - ii. Size of heap and stack by defaults

- iii. Initialize new page tables, all entries are invalid now because they map to no frame
- 3. Process termination (exit)
 - a. Destroy old address space structure
 - i. Free all pages in PT&swap region
 - ii. Free space used by address space, PT
 - iii. Check reference counts on coremap
 - 1. Frames can only be freed if all pages mapped to it == reference count == 0
- 4. Context switch
 - a. Change currently active page table
 - b. Hardware managed TLB (x86)
 - i. Change the page table register, flush TLB
 - c. Software managed TLB (MIPS, Sparc, etc.)
 - i. Flush TLB, TLB misses are handled in s/w
- 5. Memory allocation or deallocation (sbrk, stack)
 - a. When want to grow stack/heap regions
 - b. OS allocates a new page, thread continues
 - c. Grow heap region:
 - i. User-level malloc request allocation using a free pool
 - ii. If pool has no memory, malloc request more heap memory with sbrk()
 - 1. Use of sys call allows error detection
 - iii. sbrk(increment)
 - This system call increase the heap size by increment bytes
 - It increases the heap region associated with address space
 - Initializes appropriate PTEs
 - Returns the old size of the heap
 - d. Grow stack region
 - i. CANNOT use system call to grow stack
 - ii. When faulting address is close to stack, extend the stack region automatically with standard page fault handler code

Page sharing

- No memory is shared between processes
 - Page sharing allows memory shared on page granularity
 - Multiple pages mapped to the same frame
 - Good: fast communication via shared memory
 - Saves memory
 - Good isolation
1. Sharing text regions
 - a. programs/lib share ALL pages in text region
 - b. Update all pages when fram is evicted

Sharing Text Regions



c.

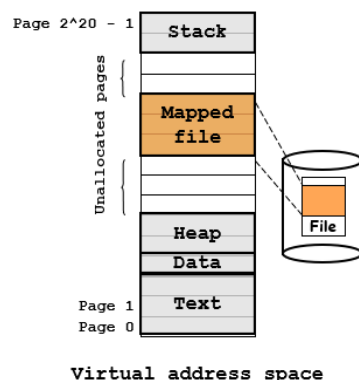
2. Copy-on-write pages

- a. Child shares frames with parents until pages are modified
- b. When modified, change all WR to RO on page table entry
 - i. COW pages
- c. When process modifies a COW page, cause a TLB read-only protection to fail, then can make a copy of a page then
 - i. Allocate a new frame
 - ii. Copy original frame to new frame
 - iii. Remap page from old to new frame
 - iv. RO changed back to RW
 - v. Resume exe
- d. If evicting shared frame, must update ALL shared pages

3. Memory-mapped files (mmap)

- a. Access and share a file with memory interface
- b. Thread read and write using memory load/store instructions
- c. Maps a file given an offset contiguously with an address space
`mmap(addr, length, prot, flags, fd, offset)`
 - addr: virtual address of mapped region
 - length: length of mapped region
 - prot: protection flags (writeable, readable, executable)
 - fd: descriptor of file
 - offset: offset in file

After mmap, accessing `addr + N` refers to `offset + N` in file `fd`



Operation:

- File data is loaded in memory on page fault (demand paging)
- When dirty page is evicted, page frame is written to file
- Essentially, file is used for backing store instead of swap area

Shared memory:

- Process map same file region: share file data
- Table entries can have different protections
- Memory can be mapped at same or different Vaddr in each process

L31 think time

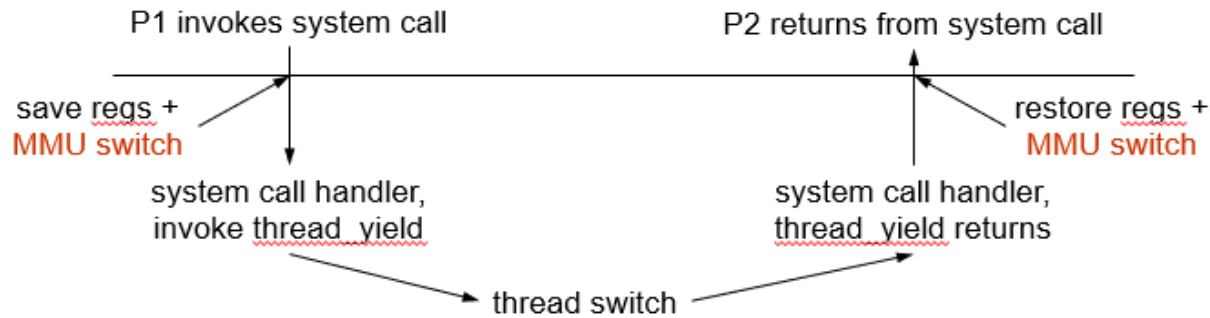
1. Since processes can share memory with mmap(), what is the difference between processes and threads?
 - a. mmap allows fine-grained sharing of parts of the processes address space
2. Does the OS use page tables for its memory?
 - a. Normally, OS accesses its memory using a simple translation mechanism that doesn't require page tables. However, the OS uses user page tables to copy in or out the data from a user process when a system call is made.
3. Do the different cores in an SMP use a single page table or different page tables?
4. Why is it hard to program with shared data structures that are mapped at different virtual addresses in two processes?
 - a. Data structures often contain pointers. The pointer values are virtual addresses. The pointer value will not make sense if the data structure is mapped at a different virtual address in another process.
5. Why is it more efficient to access a memory-mapped file than using a read or write system call?
 - a. a read or a write system call makes a copy of data from kernel to user (read) or user to kernel (write). a memory mapped file access does not require this copy – on a page fault, data is copied from disk to the user's program's memory directly, without requiring a file buffer in the kernel.

L32 Kernel Address Space

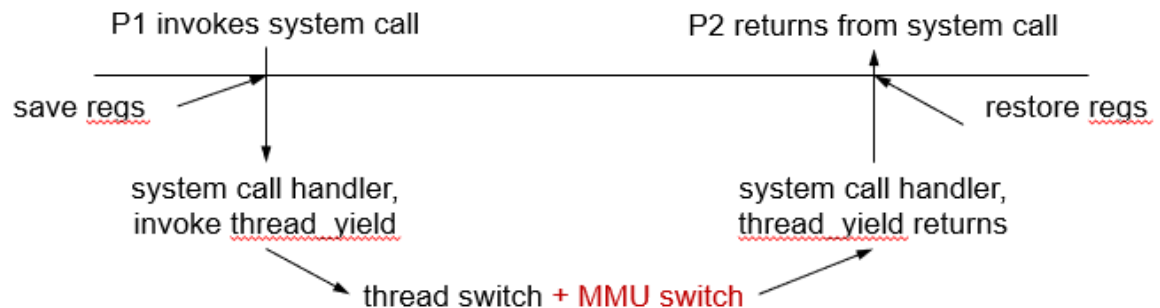
3 options available:

1. OS runs in physical memory
 - a. Address translation on MMU is turned off when running in kernel mode
 - b. OS accesses physical memory directly
 - c. Pro
 - i. OS can access phy memory directly + page table
 - ii. Does not use TLB, so no need to flush unless there is a context switch
 - d. Cons
 - i. When data is copied in and out of user space, OS needs to simulate paging hardware
 1. Address translation
 2. Dealing with dirty bit
 - ii. Uses up lots of address space → 32 bit processor uses 4GB
2. OS uses separated Virtual address space
 - a. OS has its own page table

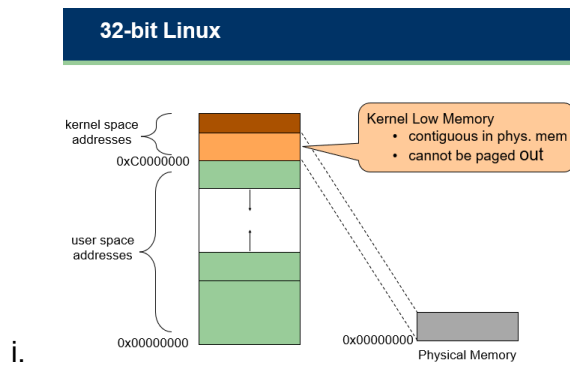
- b. Pro:
 - i. Clean design, each process has the entire address space
- c. Con:
 - i. Require MMU switch on system entry/exit
 - 1. Require TLB flush which is expensive
 - ii. Traverse page tables in software + map frames to kernel address space to copy in and out parameter



- d.
- e. When thread1 invokes system call [save user level state in regs] + [MMU switch x 1] → system call handler invoke thread_yield [thread switch x1] → system call handler takes place as return from thread_yield → [restore reg values] + [MMU switch x1] returns from system call, get to P2
- 3. OS mapped to address space for each process
 - a. OS uses page table of current process to access the process's memory
 - b. Uses current process address space
 - c. Mapped to high addresses in Virtual address space
 - d. When pages are accessed in kernel mode, the PTE bit protects OS code/data from being accessed
 - e. Pros:
 - i. No MMU switch required on system call entry/exit, cuz OS is already in the address space
 - ii. Copy in and out of system call can reuse the paging softwares
 - f. Cons:
 - i. Address space of the process is reduced
 - ii. Page table of each process needs to be setup to access OS code
 - 1. Or require hardware that allows this usage



- g.
- h. MMU switch only happens together with thread_switch when thread_yield is called in kernel mode. If return to same process, then no MMU switch required



L33 Page Replacement

Objective: minimize number of page misses

Why is a good paging algo required?

- A page miss == a TLB miss / a cache miss
- A miss required going to the disk, and is expensive + gives latency

What does paging effectiveness rely on?

- Memory access locality
- What are the two localities of reference?
 1. Spatial locality
 - a. Program access a small fraction of memory near memories just accessed
 2. Temporal locality
 - a. Program uses same memory over short period of time
 - b. And may access the memory again
- These help reduce the cost of paging

VM mechanisms

- Page table, MMU, TLB handling

VM policy

- Page replacement

Several Algorithms:

1. Optimal Algorithm
 - Select page that will not be needed for the longest time

Time		0	1	2	3	4	5	6	7	8	9	10
<u>Requests</u>			c	a	d	b	e	b	a	b	c	d
Page 0	a		a	a	a	a	a	a	a	a	a	
Frames 1	b		b	b	b	b	b	b	b	b	b	
2	c		c	c	c	c	c	c	c	c	c	
3	d		d	d	d	d	e	e	e	e	e	

Page faults X X

- Most optimal but not realistic bc don't know the future of a program
- But used to evaluate other algo

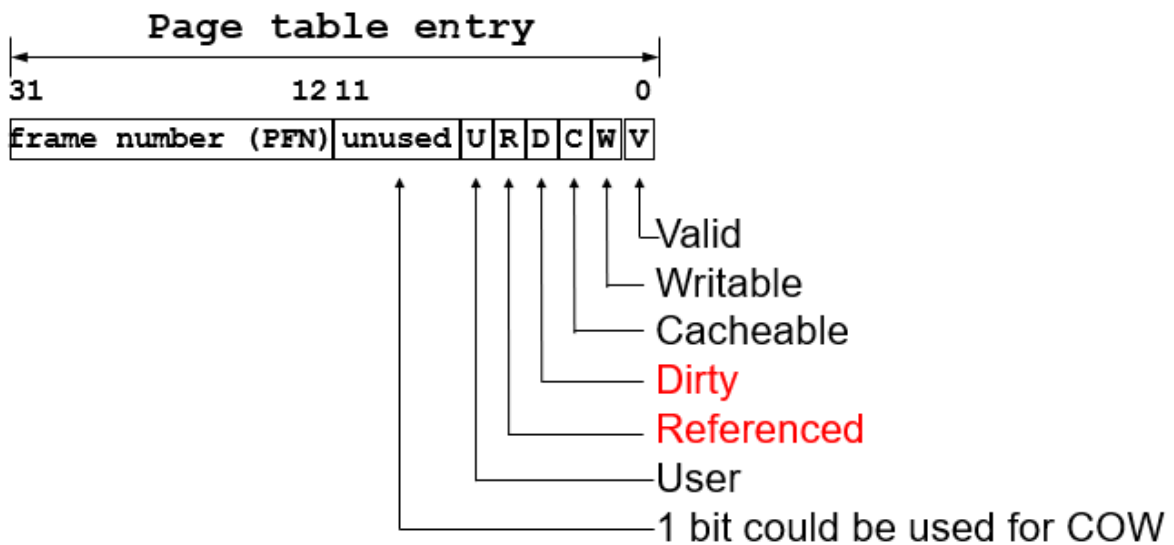
2. First In First Out (FIFO)

- Replace page that has been in memory for the longest time (first arrival)

Time		0	1	2	3	4	5	6	7	8	9	10
<u>Requests</u>			c	a	d	b	e	b	a	b	c	a
Page 0	a			a	a	a	a	a	a	a	c	c
Frames 1	b					b	b	b	b	b	b	b
2	c		c	c	c	c	e	e	e	e	e	e
3	d				d	d	d	d	d	d	d	a

Page faults X X X

- A list of all page frames in memory is kept (linked list)
 - When allocate the frame, move frame to front
 - On a page fault, take frame from end
- Problem:
 - Oldest frame might be needed soon
 - As more memory given, more faults → Belady's Anomaly



- Reference bit and dirty bit can be used to track the paging process

- Reference R:
 - Set by processor when a page is currently read/written
 - Cleared by OS (NOT HARDWARE)
- Dirty
 - When a page is written
 - Cleared by OS (NOT HARDWARE)
- Synchronizing TLB and PTE
 - Hardware:
 - A write-through cache
 - update PTE R and D when TLB R and D are updated
 - A write-back cache
 - update PTE R and D when TLB entry is invalidated
 - Software
 - When TLB read fault occurs
 - Set R, make page READ-ONLY
 - When TLB read-only/write protection fails
 - Set R and D, make page writable
 - OS is used to synchronize TLB R/D and PTE R/D

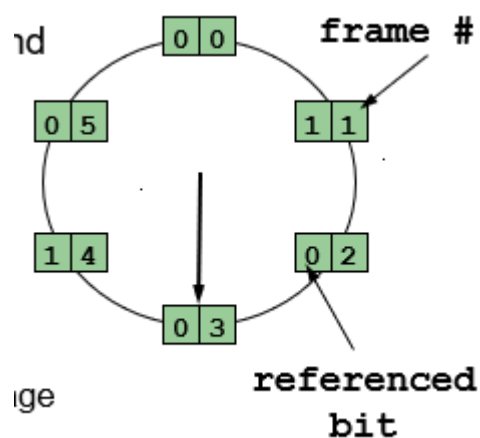
3. Least Recently Used (LRU)

- a. Replace pages that are used least recently using a list of pages kept sorted in LRU order

		<div><div>c</div><div>a</div><div>b</div><div>d</div></div>	<div><div>a</div><div>c</div><div>b</div><div>d</div></div>	<div><div>d</div><div>a</div><div>c</div><div>b</div></div>	<div><div>b</div><div>d</div><div>a</div><div>c</div></div>	<div><div>e</div><div>b</div><div>d</div><div>a</div></div>	<div><div>b</div><div>e</div><div>d</div><div>a</div></div>	<div><div>a</div><div>b</div><div>e</div><div>d</div></div>	<div><div>b</div><div>a</div><div>e</div><div>d</div></div>	<div><div>c</div><div>b</div><div>a</div><div>e</div></div>	<div><div>d</div><div>c</div><div>b</div><div>a</div></div>
Time	0	1	2	3	4	5	6	7	8	9	10
<u>Requests</u>		c	a	d	b	e	b	a	b	c	d
Page 0	a	a	a	a	a	a	a	a	a	a	a
Frames 1	b	b	b	b	b	b	b	b	b	b	b
2	c	c	c	c	e	e	e	e	e	e	d
3	d	d	d	d	d	d	d	d	d	c	c

- b. Page faults x x x
- c. Problem:
 - i. Updating LRU list is expensive
 - ii. Requires a counter from MMU to increment at each clock cycle BUT MMU DOES NOT HAVE THIS
 1. Ideal scenario:
 - a. MMUS writes counter values to PTE
 - b. Timestamp recorded as time of last used
 - c. When page fault occurs
 - d. OS look for page with oldest timestamp
- d. Solution: LRU approximation
 - i. OS has a counter in software
 - ii. Periodically (with time interrupt):

1. Increment counter on each page when a R bit is set
 2. Write counter value
 3. Clear R bit once finished
 - a. So if R is not ref after this increment, it will not be counted again
 4. On a page fault
 - a. Software looks through PT
 - b. Identify page with oldest timestep
 - c. Break tie randomly
- iii. This method approximates LRU because of the granularity of counter is the timer interrupt
4. Clock algorithm
- a. FIFO + give 2nd chance to referenced pages
 - b. Keep a circular list of page frames in memory

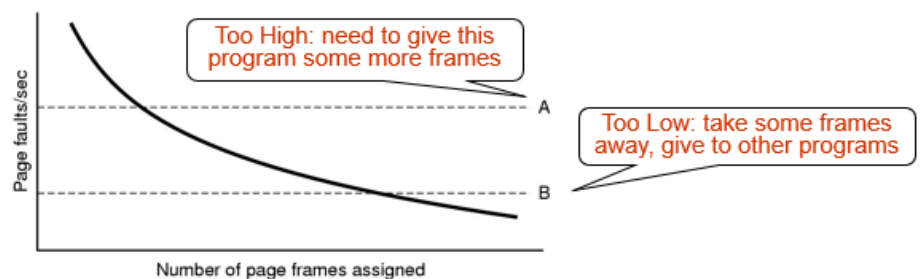


- c.
 - d. Have a clock hand to keep track of which frame will be evicted next
 - e. Steps:
 - i. Add new frame to the block behind the block hand
 - ii. On page fault, start looking at the page pointed by clock hand
 1. If R = set
 - a. Unset R
 - b. Go to next without evicting current page
 2. If R = unset
 - a. If page is dirty
 - i. Schedule page write, go to next page and see if can evict that
 - b. If page is clean
 - i. Select this for replacement
- What is a working set?
- A set of pages a program currently needs
 - Look at the last time interval T
 - $WS(T) = \{\text{pages accessed in interval (now, now - T)}\}$
5. Working Set Clock (WSClock)
- a. Keep working set in memory so few locality reference page faults
 - b. Each entry contains time of last use rather than a reference bit
 - c. Steps:
 - i. Add a new frame w/ frame # and time last visited

- ii. On a page fault
 - 1. If R = set
 - a. Update time-of-last-used field to current Virtual time
 - b. Unset R, continue without evicting
 - 2. If R not set
 - a. Compute age
 - b. If age < T
 - i. Continue without evicting
 - c. Else if age > T
 - i. If D = dirty
 - 1. Schedule for page write, continue without evicting
 - ii. Else
 - 1. Select, evict

6. Page Fault Frequency (PFF)

- a. Better than WSClock because
 - i. Does not need to know working set interval T for each process
 - ii. Can give estimate of working set needed

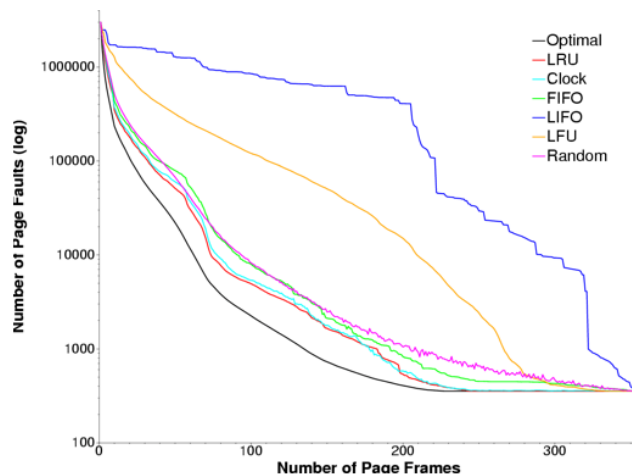


- 1.
- 2. If page frame fault high and #page frames assigned low → give program more frames in working set
- 3. If page frame fault low and #page frames assigned high → give program fewer frames in working set
- iii. Measuring page faulty frequency
 - 1. For each thread
 - a. On each fault, count fault (f)
 - b. $f = f + 1$
 - Every second, update faults/second (fe) via aging for all threads
 - c. $fe = (1 - a) * fe + a * f, f = 0$
 - d. $0 < a < 1$, when $a \rightarrow 1$, history is ignored, a is weighting factor, usually chosen for 0.99
 - b. Goal: allocate frames so PFF is equal for ALL program
 - 2 step process:
 - 1. Choose victim process with lowest PFF
 - 2. Within the victim process, use WS/Clock or LRU to evict a page

How to know which algo is the best?

- 1. Look at which virtual memory addresses are accessed
- 2. Pages accessed:
 - a. 0000001222333300114444001123444

3. Eliminate duplicates
 - a. 012301401234, Why? Because duplicate does not cause replacement for any algo
4. Defines the reference string
 - a. Use same reference string for all algo
5. Count # page faults



- Summary of paging algorithm

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
FIFO	Might throw out important pages
Clock	Realistic
LRU	Excellent, but difficult to implement efficiently
Working Set Clock	Efficient working set algorithm
Page Fault Frequency	Fairness in working set allocation

Paging issues:

1. Paging and IO
 - a. EX: T1 calls read(), suspend during I/O, T2 runs, get a page fault
 - i. A frame needs to be evicted
 - ii. Frame selected to be evicted == frame involved in read from T1
 - iii. Then this frame is allocated for page needed in T2
 - iv. When IO returns, OS will copy data to the new page, and now data is on T2 page
 - b. Solution:
 - i. Each frame has a "DO NOT EVICT ME" flag
 - ii. Aka "pinned page"
 1. Pin during I/O, unpin after I/O
2. Paging performance
 - a. Paging works best when there are a lot of free frames
 - i. So no need to evict
 - b. If paging is full of dirty pages
 - i. 2 disk operations needed for each page
 1. Swap out dirty page

2. Read in new page
- c. Methods to improve performance:
 - i. Paging daemon → swap OUT in advanced
 1. Maintain a pool of free frames
 2. 2 watermarks:
 - a. Low watermark
 - i. Starts running when this is reached
 - ii. Writes out dirty pages, mark the pages as free
 - b. High watermark
 - i. When this is reached, stop freeing pages
 3. Frames in pool still hold the previous content (daemon does NOT delete the content) → rescue!
 4. Problem:
 - a. If a page will be written again: swapping causes the page to be written twice
 - ii. Prefetching → swap IN in advanced
 1. Page fault only process 1 page/time
 2. Predict future page usage at current fault, prefetch other pages
 3. Problem:
 - a. Wasting disk accesses and memory

What is thrashing?

- A situation where OS spends most time paging from disk and making no progress
 - == livelock
- System is over-committed because:
 1. page replacement algo is NOT working
 2. System does NOT have enough memory to hold working set of ALL currently running program
- Solution:
 - Suspend some programs, buy more memory

L33 think time

1. What the optimal page replacement policy?
 - Is it achievable in practice? Why is it used?
 - To evict program following the future pattern
 - Not achievable in practice
 - Used to compare with other policies and see if they are good
2. What is the problem with FIFO page replacement?
 - a. Oldest page may be needed soon
 - b. More faults as more memory given
3. What is the assumption used by most replacement policies to improve on FIFO?
 - a. Assuming we are able to keep track of the access time/order of pages
4. What is the working set of a process?
 - a. A set of pages needed recently
5. Which of the policies described in these slides take working set into account?
 - a. Working set clock
6. What happens when the working set does not fit in memory?
 - a. thrashing!

7. How does virtual memory interact with I/O?
 - a. pages involved in IO cannot be evicted
 - b. paging daemon: starts when too few clean or free frames are available (low threshold), stops when a sufficient number of clean or free frames are available (high threshold)
8. When should a paging daemon start operation and stop operation?
 - a. Starts when low watermark is reached and stops when high watermark is reached
9. Describe some situations when prefetching of pages will work well
 - a. When it is easy to know exactly what pages will be needed soon
 - b. when a sequence of virtual pages are read (say page 8, 9, 10), then keep reading the pages in the sequence (e.g., 11, 12). This will work with a large array, and for files that are read sequentially.

L34 Disk and Raid

- Disk: multiple platters
 - data written on platter through modifying the magnetic fields on platter
 - Read by disk head → 1 and 0
 - Can move itself to read data
 - Each platter has multiple tracks
 - Each track has multiple sectors
- Sectors:
 - A header
 - 512 bytes of data
 - Modern: larger ~4k
 - 16 bytes of Error correcting code ECC
- Cylinder with same track across different platters
 - Platter arms move together
 - Different heads access data in parallel

What are the three delays that determine the time needed to access the disk sector?

1. Seek time
 - a. Disk head moves over to the correct track
2. Rotational delay
 - a. Wait until disk is rotated to the correct location
3. Transfer time
 - a. Time to read/write the bits of sector

What are the trends of disk performance improvement?

- Capacity
 - Capacity doubles every year
- Transfer rate BW
 - 2X every year
 - 50-200 MB/s
 - 100-200 ms sector transfer time
- Seek time and rotation time
 - Becomes ½ every 10 years
 - Pure mechanical → hard to make improvement

- Rotational delay as well

Relationship between BW and access type

1. Highest bandwidth for **sequential access**
 - a. No seek time
 - b. No head movement involved
 - c. No transfer time
2. Lowest BW for **random access**
 - a. Seeking data that randomly locate on the disk
 - b. Head movement, rotational delay, transfer time

What is the goal of disk scheduling?

- Minimize seek time

Addressing disks

- Modern: complicated, not all sectors have the same size
- Higher-level interface
 - Disk exports data as a logical array of sector
 - Maps logical sectors to its surface
 - Simplifies OS code
 - Disk parameters are hidden
 - Each disk has a separated OS system on the disk not visible to the OS on laptop

Disk errors

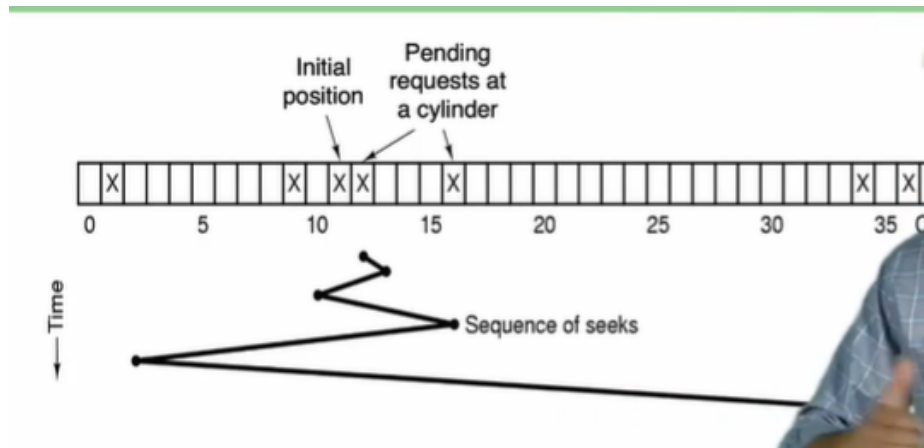
- When trying to change/read magnetic field, may get error
- Latent sector errors, mis-directed writes
- Errors could be:
 - Transient errors
 - Hard errors
- Some errors are masked by ECC
 - When disk controller notices that a sector is not operating correctly, it maps data on that sector to an available sector

Goal of disk scheduling algorithms

- To improve disk performance
1. Reduce seek time
 2. Read several sectors of data at once

4 scheduling algorithms:

1. First come. First served FCFS
 - a. Slow, simple
 - b. If requests have no locality → can take a long time
2. Shortest seek time first SSF
 - a. Look at all cylinders on disk, if currently serving a request
 - b. Next request will be the closest to the current request



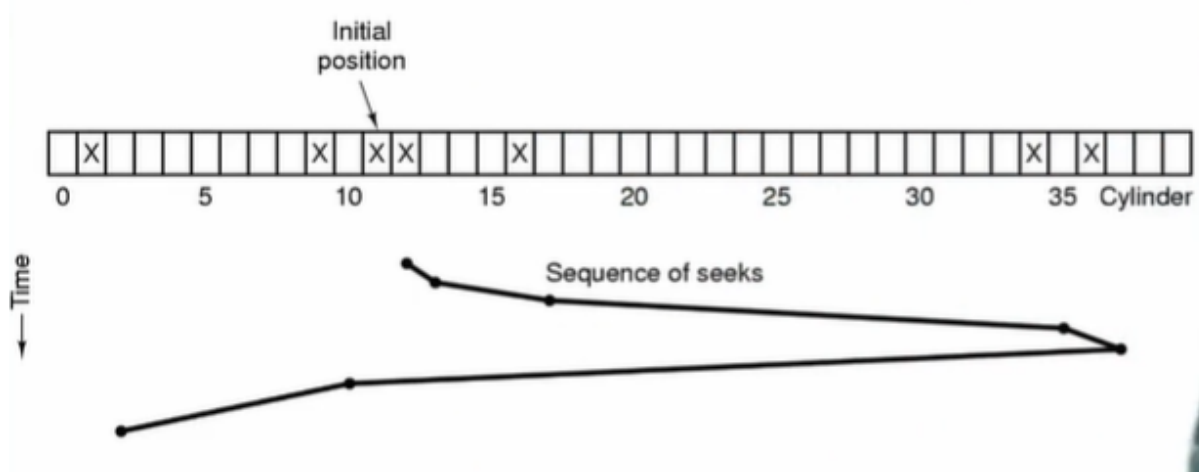
c.

d. Problem: starvation

- i. If new requests keep coming to the location close to the current request, requests farther away might starve

3. SCAN (Elevator)

- a. Move along in one direction, go all the way to the end
- b. Then change direction



c.

d. Implementation:

- i. Use a bit to track the direction (inward/outward)
- ii. Serve the next pending request in same direction
- iii. Change bit value when no more request is in the current direction

e. Advantage:

- i. Increases seek time
- ii. Prevents starvation

f. Disadvantage:

- i. Unfair: requests in the middle get serviced twice, requests at the end only once

4. C-SCAN

a. Solves the above problem:

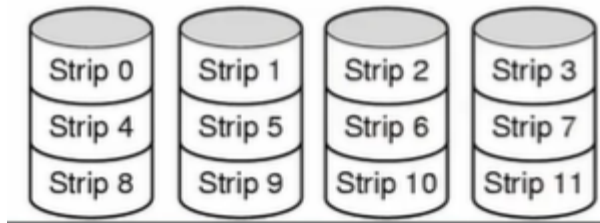
- i. Only go in one direction

b. Advantage: does not discriminate against requests at the two ends

Redundancy in storage system

- Redundant Array of Inexpensive Disks (RAID)

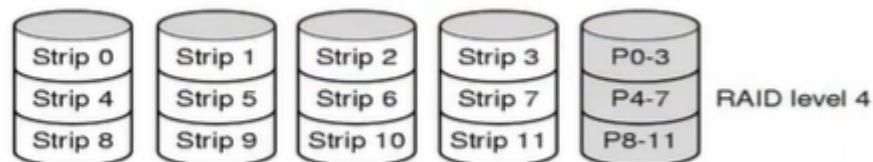
- Use many disks in parallel
- Chock disks putting as packages



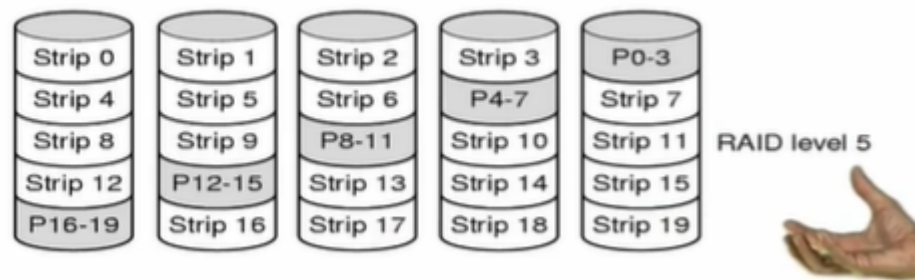
- RAID organization
 - A chunk: a unit of read or write on a disk
 - Consists of one or more sectors
 - Consecutive chunks placed on different disks

3 levels of RAID:

1. RAID level 0: disk striping
 - a. Distributes data across several disks for speed
 - i. Can read in parallel on different disk
 - b. Disadv: no redundancy, if one disk fails, then all fail
2. RAID level 1: mirroring
 - a. Backup solution:
 - i. When write a disk in strip 0, also write one to 0'
 - ii. Each chunk goes to two different disks
 - iii. When read, can go to either disk → higher throughput
 - iv. Far more read than writes, read one of the two data
 - b. Downside: doubling number of disks
3. RAID level 4:



- a.
 - b. Add 1 parity disk that contains parity information
 - i. Calculates XOR value of chunks and store on parity disk
 - ii. $P03 = S1 \text{ XOR } S2 \text{ XOR } S3$
 - iii. Assume writing to S0, updating S0_new, it has S0_old before
 - iv. $P03_{\text{new}} = P03_{\text{old}} \text{ XOR } S03_{\text{old}} \text{ XOR } S0_{\text{new}}$
 - c. Ensures: if one disk fails, the information can be reconstructed with info on the other 2 disks (P03, S1, S2 -- fix → fail in S0)
 - d. Disadv:
 - i. Parity disk becomes a bottle-neck
 - ii. Needs to be updated every time
4. RAID level 5: striping with distributed parity



- a.
- b. Parity info distributed to all arrays
 - i. Avoids bottleneck for parity disk

L34 think time

1. What are the main delays associated with disks?
 - a. Seek time
 - b. Rotational delay
 - c. Transfer time
2. Disk scheduling algorithms optimize for which of these delays? Why?
 - a. For the seek time delay
 - b. Because rotational and transfer time delays are related to the hardware and mechanics so hard to improve
 - i. Exact location of sectors and current position of head is unknown
 - c. Seek time delay can be optimized directly because they are expensive

L35 file system introduction

What is a file system?

- User interacts with shell, shell interacts with file system
- File system provides abstraction for
 1. Storing
 2. Organizing
 3. Accessing persistent data
 - a. Data survives after each create process has terminated
 - b. Data can survive after machine crashes, reboot...
- Data stored on disks, tapes, solid-state drives SSD as they remains unchanged when power is turn off
 - SSD
 - Stores data persistently with electronic circuits
 - Adv:
 - Much faster, no need for waiting disk heads
 - portable
 - Disadv:
 - each block can only be written for a finite amount of time. If a spot has reached its lifetime, move it ot
 - Expensive
 - Old data: rotating disk New data: SSD

-

What is file-system data organized as

- As objects called files (just a sequence of bytes)
- Is a virtualization of disk
 - Seek, read, write → operations can be done on file and disk
- To name files and locate files
 - Names organized into directories

How are files accessed?

- Through system calls
- Files can be accessed concurrently by different processes
 - OS needs structure to keep track of this to ensure the concurrency

What is a buffer cache

- A part of file system
- A cache in memory of disk blocks
- Needed because:
 - Open a file, read three files, need to go to file disk, read, and return
 - Instead of read disk every time a data is required, cache the entire block to buffer cache, so when data around that variable is being accessed, it's faster

What are the key resources that a file system has to manage?

1. Storage media
 - a. Disk
2. Files
3. Directories
4. Buffer cache
5. Operating system also needs to keep track of open files

What is a disk block?

- Disks are accessed at granularity of sectors
- File system allocates data in chunks called block
 - File -- triggers → array, ask for data array of blocks
 - 1 block = 2^n contiguous sectors
 - 4K block, need 8 sectors, each sector has 512 bytes
 - $4K/512 = 8$
- Why organize in blocks?
 - Access files in terms of bytes, so why in block?
 - Cuz this will increase overhead
 - In blocks, reduce overhead
 - Larger block size, better throughput, less overhead, MORE internal fragmentation

What is the primary role of managing block?

- Free block management
 - Keep track of free blocks
 - Bitmap
 - 1 block/bit, 1 if allocated
 - Linked list
 - Link free blocks
 - B-trees

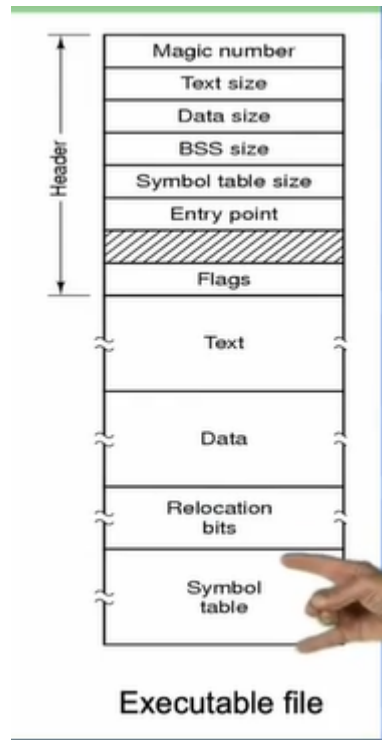
- Each node has multiple links going out
- Designed for disks, 1 node/block
- Allocate blocks to a file
- Manage free blocks
- Similar to memory space

Bitmap used for free block management:

- 1 bitmap in a separate area on disk
 - WHY has to be on disk?
 - In case the system crashes, still need to know which block is that
- Block size = 4KB, disk size = 1TB
 - # of blocks: $1\text{TB}/4\text{KB} = 244\text{M}$
 - Bitmap: 244MB, stored on disk, takes 30MB disk space == 7500 disk blocks for bitmaps
- Adv:
 - Simple usage
 - Allocate contiguous blocks to a file easily
 - Can do file fetching in a quicker manage, only accessing 1 bitmap / time
- Disadv:
 - Not optimal for allocating large contiguous set of blocks
 - 100GB large file, want blocks of this file to be contiguous
 - Hard to find all bits == 0 that are consecutive in bitmap

What is a file?

- A sequence of bytes
- OS does not know what is in file, just a sequence of file, up to the application to type of numbers, structures
- Most of the time, internal file type does not matter and OS does not care
 - Files given name with extension
 - .PDF → OS does not care, no requirement
 - For conventions
- In case OS needs to be able to read a file → executable file



-
- Executable and linkable file ELF
- Has size, and can specify which OS can work with the file
 - The symbol table contains debugging information that tells a debugger what memory locations correspond to which symbols (like function names and variable names) in the original source code file.
- “A.out → assembler output”

What is the metadata in file?

- Various attributes → “**metadata**”
 - Name, owner, creation time, access permission
- System manage metadata in per-file data structure on disk

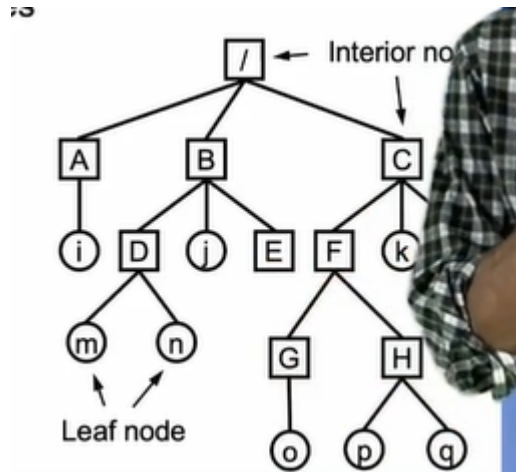
What are the file related system calls?

1. Open
 - a. Start using file, set offset of reading pointer positions to the beginning of file
2. Read, write
 - a. N bytes from/to current position
 - b. Read 3 bytes, offset position moved 3 bytes
 - c. Update positions
3. Seek
 - a. Move current offset position to byte location specified
 - b. Seek at beginning/end...
 - c. Allows seeking at random place
 - d. Create a random file, seek 1GB, start writing 1 byte to that, the file size will be 1GB+1, that 1GB space will be kept each with 0 indicating free, so can use as a hash table
4. Create, rename, delete, get/set, change permissions...
 - a. Done in shell

Abstraction helps name, organize, and locate files? → directories

- Stores a list of directory entries

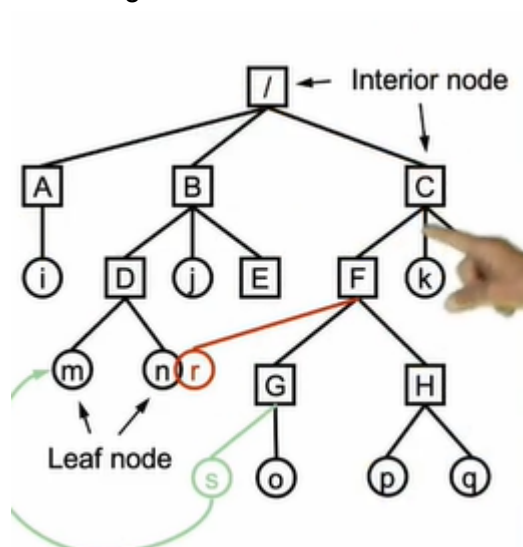
- Each pointing to a file
- Each directory contains reference to the subnode directory/file



- Files are identified with pathnames
 - Root director has an empty string, root directory does NOT have a name, ALL other directories have NAME
 - 1. Absolute pathname
 - /B/D/n
 - Dash: going from root
 - B: to be, /D to D, /n, to reference n
 - 2. Relative pathname
 - a. Current working directory PWD: the directory operating in
 - b. Programs inherit the directory, thus file names can be specified relative to the current directories
 - c. Cd /B (currently in B), cat /D/n
- Directory also has metadata, similar to files

Example of a Unix directories: Links

- A directory can be assigned for more than one name with LINKS
- Hard links
 - 2 names referring to same file



- /C/F/r == same file
- Disadvantage:

- What are the 2 restrictions?
 1. Can only refer to files, not directories
 2. Can only refer to files in the same file system
 - a. OS may be mounting files from different file systems
 - i. Network, local file system ...
- Adv: Hard link has "reference count" keeps track of how many hard links are referring to the same file, will decrement counters and delete file when counter = 0 → safer
- Symbolic links
 - Aka shortcut on windows
 - A regular file s contains the name of another file
 - /C/F/G/s, s is a file created after creating the symbolic link
 - the content of file s contains link to /B/D/m
 - Advantage:
 - Can refer to files in other file systems
 - Disadv:
 - A symbolic link pointing to a file in another file system
 - That file got removed in the other system, symbolic link does not know

What are the directory related calls?

1. Open
2. Readdir
3. Seekdir
4. Close
5. Create, rename, delete...
6. Link/Unlink
7. *** NO WRITE DIRECTORY
 - a. May be creating a directory, someone else overwrite the directory, so all data lost

L35 think time

1. What is the purpose of directories in a file system?
 - a. To allow naming for files
 - b. Provide sub directories
2. What operations update directories?
 - a. Modification/creation/deletion/rename on file → update directory timestamp
 - b. Changing the attribute of the directory
3. In Unix, the directory hierarchy forms an acyclic graph. Explain how.
 - a. Acyclic graphs are caused by hard links on files
4. How are cycles not allowed in the graphs
 - a. If the graph is cyclic, then every graph will be a sub-directory of each other
 - i. Every directory needs to be empty before they can be removed, thus a directory in a cycle will never be emptied or removed
5. benefits/drawback of inodes:
 - a. Benefit:
 - i. Scalable
 - ii. Allows hard links

- b. Drawback
 - i. Less efficient because path lookup requires reading two block per component
 - 1. Data block
 - 2. Inode block
 - ii. Without inode, the lookup will only require 1 reading of block
- 6. difference between hard and symbolic links
 - a. Hard link:
 - i. Allows multiple directory entries to point to the same inode
 - ii. But can only link files that are in the same file system
 - b. Symbolic link
 - i. Is a directory entry that points to a special file which contains the path of another file
 - ii. Allows linking files in different file system
 - iii. Faces the risk of having dangling pointers

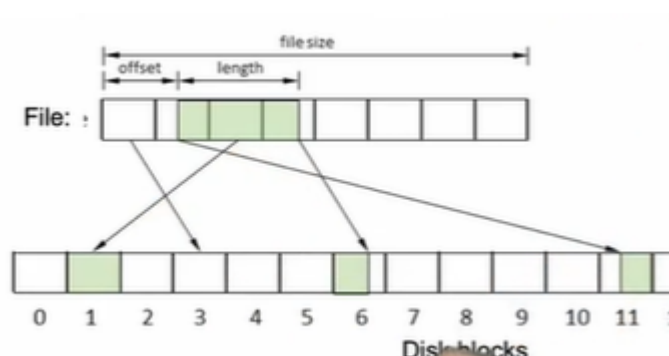
L36 file system design

What does it mean “Need to store file durably”

- Data in file durably
- Information of file, directory is durable
 - File is in the same place after system crashes
 - Should be able to recover data and bring file system back to a consistent state

Key resources for files

- 1. Storage media: blocks
 - a. Bitmap
- 2. Files
 - a. Sequence of bytes, disk blocks used to store them
 - b. Each file is partitioned into a sequence of blocks, each mapping to a different disk block



- c.
- d. To read data:
 - i. Seek to the given location
 - ii. Identify the blocks needed to be read
 - iii. Find the blocks on disk blocks mapped to the file
 - iv. Return back to app

e. 4 options for allocation of file blocks on list:

i. Contiguous allocation

1. All blocks in a disk are contiguous on disk

- Adv:

- Sequential access
- Smaller seek time → good performance
- Easy to identify all blocks for the same file -- increment

- Disadv:

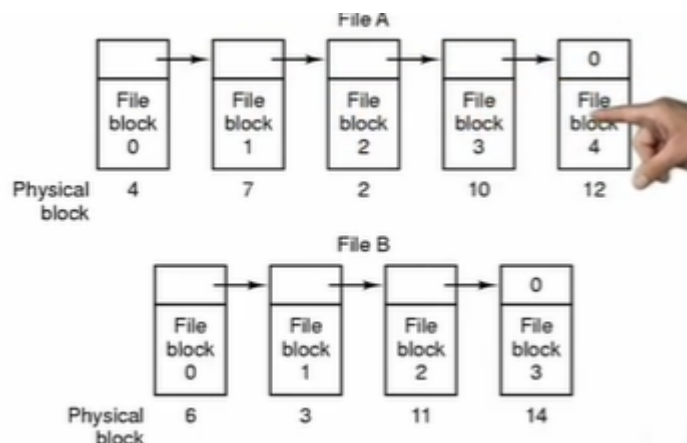
- cannot grow size of file on disk, need to find a larger space and copy whole file, delete, reallocate
- Create fragmentation, having multiple holes in disk
 - Requires periodic compaction
- CD-ROMs still using this bc file size is pre-defined and never changed

ii. Linked list allocation

1. Each file is a linked list of block

2. Each block links to the number of next block, and keep track of the physical block location that block is in

- Disadv:



- Random access to file data are slow
- Location of next block takes 4byte → takes up memory
- Solution?
 - Keep a linked list in memory -- FAT

iii. File allocation table FAT

1. Keep a linked list information in memory

2. The index table with 1 entry per disk block is stored on DISK



3.

a. E.g. file

4. For a file, file A starts with physical block 4, the next block after 4 is 7, next block after 7 is 2 -- 10 -- 12... until reach -1: end of file

5. Update the table everytime the block is changed

- Adv:

a. Random access is in memory so quick

- Disadv

b. Entire table stored in memory

i. Takes up a large memory

c. Does NOT scale with large file system

i. 1TB -- 1GB

iv. Inode Based allocation

1. Problem with previous two implementation:

a. Linked list on disk spreads index information on disk, so slowing down access

b. FAT keeps linked-list in memory, but the size of file system is then limited

2. Core idea:

a. Store index info of file--block close on disk

b. Cache this index info in memory when file is open

3. What if a file size increases, the index may take up more memory as file number increases?

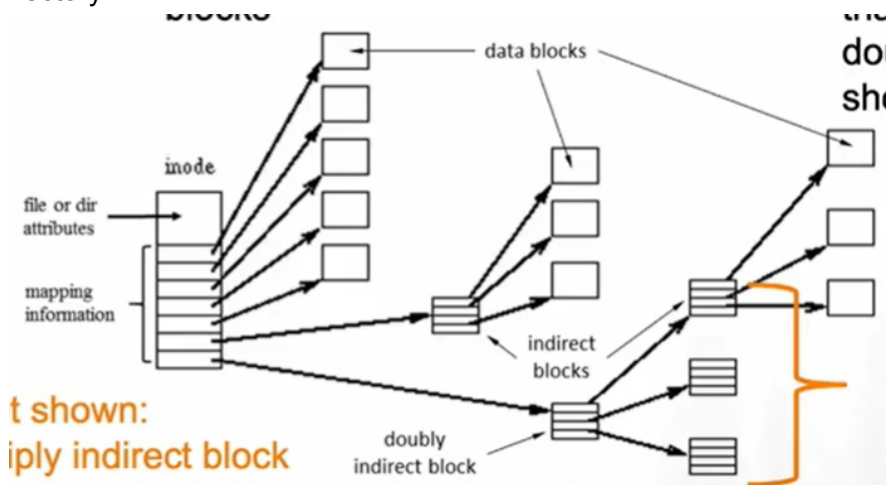
a. Tree structure to store index information related to block

i. Allows the growth of index

ii. Without spreading info to much, so easier search

b. Inode: root of the tree

- i. Has a fixed size
 - ii. Organized in an array on disk
 - iii. "I-node" Index into the array used to identify the inode
 - 1. A low-level name of file
 - 2. 1,2,3,... → file name
 - c. Inode and all information are stored on disk, but when accessing a file, the information in that block is copied to memory
 - d. 1 inode / file or directory
4. Inode directory



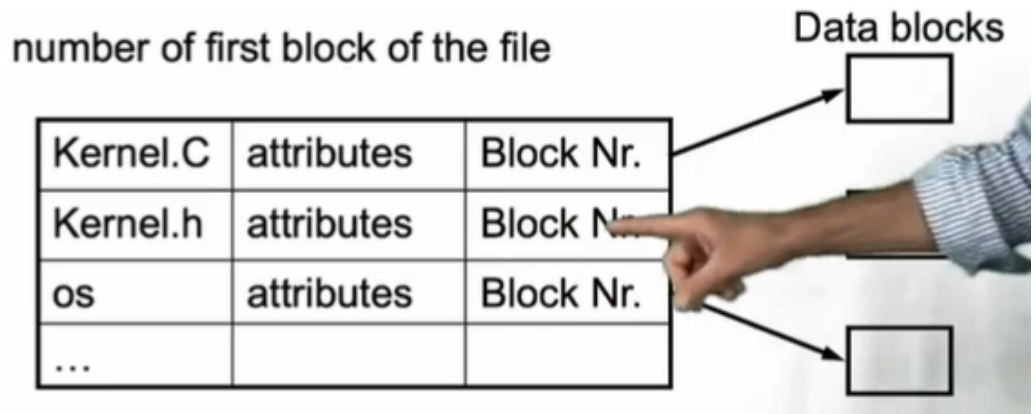
- a.
- b. 3 types used here:
 - i. Inode
 - ii. Data block
 - iii. Indirect block
- c. In inode, 2 types
 - i. Beginning: all file/dir attributes
 - ii. 12 pointers/reference (direct block): mapping information
 - 1. Each pointer refer to a data block
 - iii. 13th pointer: points to an indirect block (1 level of indirection)
 - 1. Each reference in the indirect block, points to a data block
 - iv. 14th pointer: points to an indirect block. Each pointer in the indirect block points to another indirect block (2 level of indirection), then each pointer points to a data block
- d. Indirect block disk size: 4K
 - i. capable of pointing to $4K/4bytes = 1K$ indirect block in the next level
 - ii. Also capable of pointing to 1K data blocks
 - iii. 2 levels of indirect blocks:
 - 1. Can refer to $1K \times 1K$ bytes of data blocks

5. Adv:

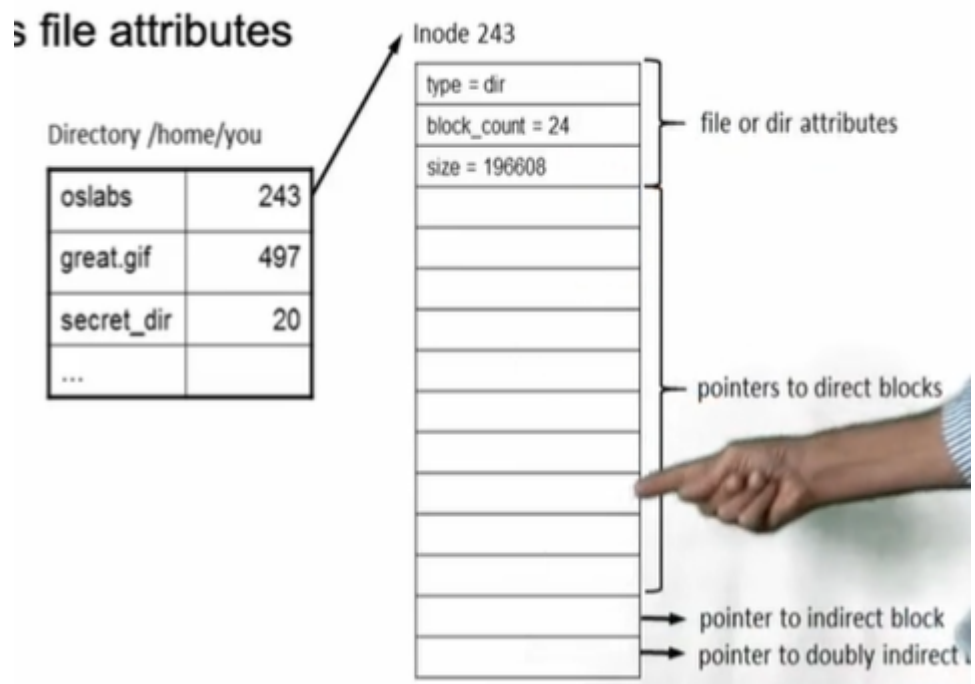
- a. Data are relatively small, so usually 1 inode is enough
- b. Medium size file can use 1 level of indirect node
- 6. Each block size 4K, each pointer size = 4bytes
 - a. Each block can hold 1024 block pointers
 - b. Max number of blocks in a file:
 - i. 12 direct blocks
 - ii. 1K in 13th
 - iii. $1K * 1k$ in 14th
 - iv. $1K^3$ in the triple indirect block
 - c. Max file size: 4TB

3. Directories

- a. Contains 0/0+ entries
 - i. 1 entry/file in the directory
- b. Directory data block
 - i. Same as file block, but marked as directory so cannot ruin that file
- c. Entry entry maps file name to location of the starting block/first block in FAT/ inode

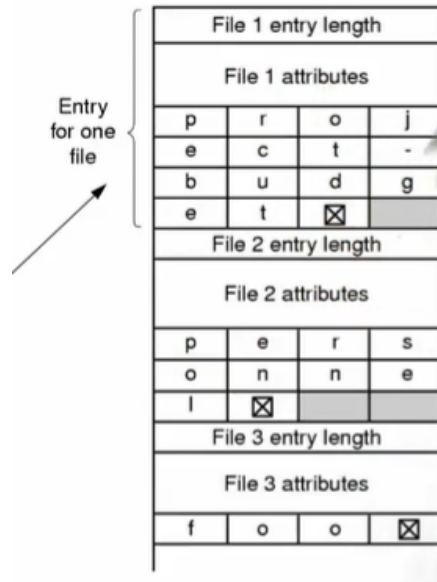


- i.
- d. Unix directories



- i.

- e. File names:
- Variable length
 - At most 4K in Unix
 - File entry length, file attribute, name



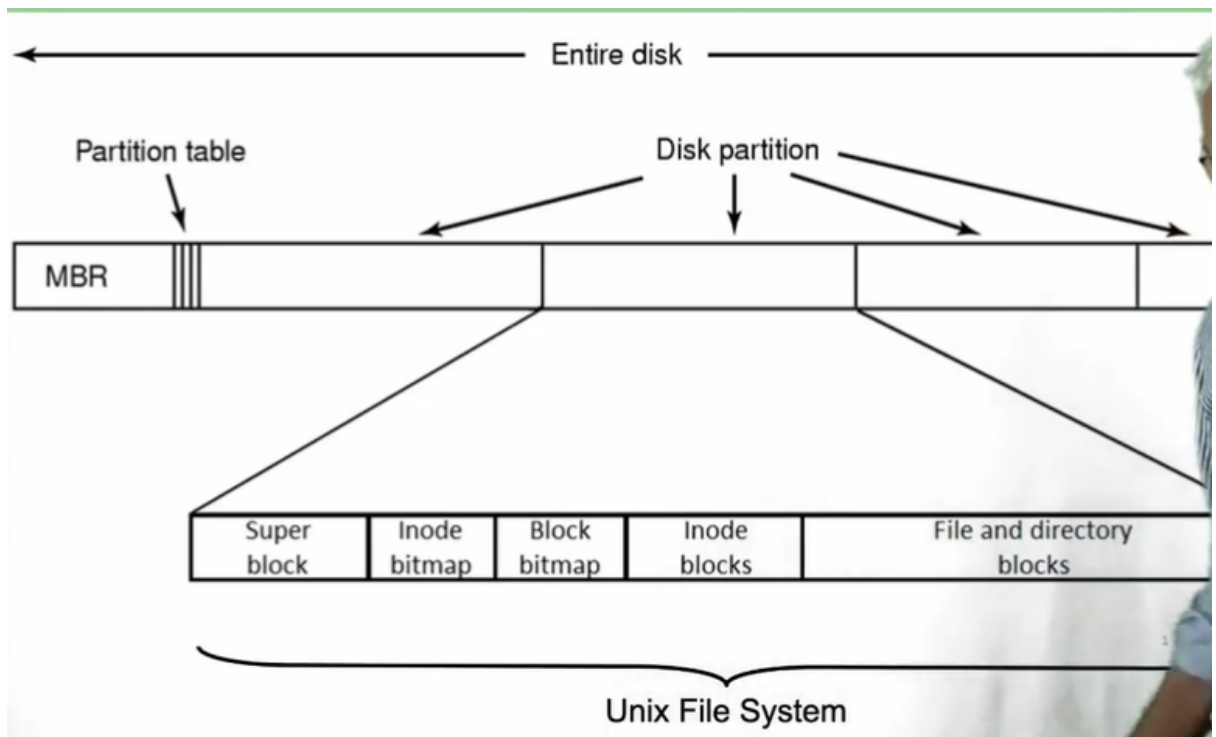
- iv.
- v. Path lookup in Unix
- Ex: F in /D1/D2, total 8 disk accesses
 - Read root directory inode
 - get inode for that Data Block with D1's inode
 - Read D1 inode
 - Read Data Block with D1's inode refers to inode of D2
 - Read D2 inode
 - Read Data Block with D2's inode refers to inode of F
 - Read F inode
 - Read data blocks of F file
 - To solve this, cache!

4. Buffer cache
- Going to disk every time slows down the process
 - File operations access same disk block again and again
 - E.g. root directory will be accessed every time
 - Locality
 - Tend to read files sequentially
 - If read 1 byte from a file, probably will read another byte from the same file
 - Cache operations
 - Block lookup
 - If block is in memory, return data from buffer
 - Block miss
 - If block not in memory
 - Allocate a block in the buffer cache

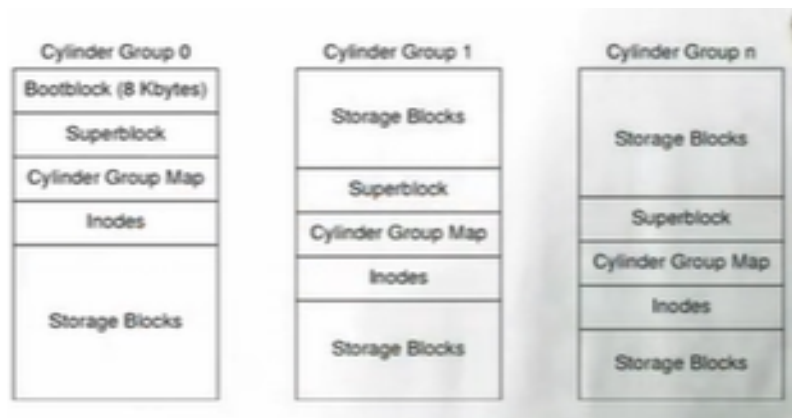
3. Load in block from disk, store in that memory
- iii. Block flush
 1. If a write causes buffer to be modified
 2. Need to write it back to disk
- e. Many blocks can be cached in memory
 - i. 16GB machine, 8GB for buffer cache
 - ii. Block size = 4K, number of blocks cached = 2M
- f. Trade-off:
 - i. Increase physical memory for buffer cache → VM memory decreases
 - ii. Decrease buffer cache → bad IO performance
- g. Implementation:
 - i. Hash table
 1. Able to lookup whether block is in memory efficiently
 2. Each block number is a key
 3. Disk blocks in memory is stored in a linked list
 - ii. When should disk blocks be updated? Want to make sure content on cache page and disk page are the same
 1. Immediately (synchronously)
 - a. Write-through cache
 - b. Disadv: too slow, too much overhead
 2. Later (asynchronously)
 - a. Write-back cache
 - b. Adv: fast
 - c. Disadv: what if the system crashes
 - i. Some blocks modified in memory are not on disk
 - ii. File system becomes inconsistent
 - iii. Buffer cache issue?
 1. Has limited size, need replacement algorithm to enable page evicting
 - a. LRU
 2. Unified memory cache for buffer cache and VM memory pages
 - a. Can be used to solve the tradeoff between VM and buffer cache size
 - b. Problem: if a program reads a large file, can affect programs that are not accessing files much (a lot of pages are evicted from VM system)
- h. Biggest adv of buffer cache:
 - i. Read ahead
 - ii. File system can predict the process will request a file block after the one that is requesting
 - iii. File system prefetches next block from disk == Read ahead
 1. Decreases wait time for IO

5. Open file structures

Unix file system layout



-
- 1 file system / disk partition
 - Can determine which file system (mac/windows) to boot
- Partition table
 - Where each partition located
 - How large
- Master Boot Record MBR
 - Disk info
 - Code to start boot process during system boot out
- In each file system:
 - File and directory blocks
 - Inode blocks
 - Bitmap describing which block in file/directory blocks is free
 - Bitmap describing which block in inode blocks is free
 - Superblock: info of each of the previous section
 - Block placement:
 - A policy → block allocation
 - 2 placement problems:
 1. If need more space for file directory, will get space from unused inode bitmap → blocks of new files eventually scatter across disk
 2. Inodes are far from blocks, so Many back and forth → long seek time problem
 - Solution: BSD fast file system
 - New file system called FFS fast file system
 - Partition disks in to columns of cylinders, each having different group
 - In each group:



-
- Placement policy:
- Ensure storage blocks and inodes are stored in the same cylinder space
- If cylinder is full, try to place in a group that is close to this one

L36 think time

1. What are the benefits/drawbacks of using inodes in a Unix file system vs. the FAT file system?
 - a. FAT:
 - i. Adv:
 1. Random access lookup is quick because the entire lookup table is stored in memory
 2. Efficient for smaller file system
 3. Lookup only requires 1 block per path component
 - ii. Disadv:
 1. Stored in memory so takes up space
 2. Not scalable to larger program
 - b. Inode
 - i. Adv:
 1. Does not take up a lot of space in memory because most information is stored on disk
 - a. Inode stored in memory and only be cached to the cache buffer when read
 2. Scalable to large file system
 3. Allows hard links
 - ii. Disadv:
 1. Path lookup requires reading two blocks per component:
 - a. Inode block
 - b. Directory data block

2. What were the problems in the Unix file system that led to the FFS design?
 - a. Files become fragmented when they are too spread out on disk
 - b. Inodes and file data were far apart

L37 Unix file operations

Inode recap

- Map disk to file
- Locate disk
- Contains metadata

ref count	time	uid
mode (permissions)	ctime	gid
size (in bytes)	mtime	blockcount

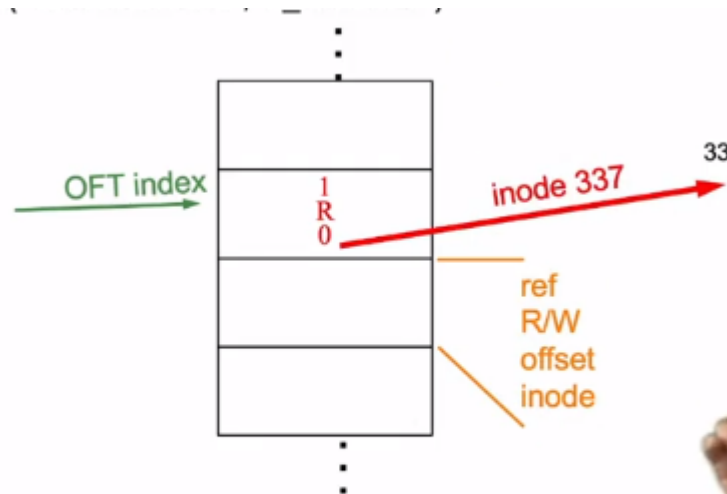
-
- Ref count: How many directories are referring to this file
- Time: last access, ctime: create time, mtime: modified time
- Owner of the file: uid
- Group of the file belonging to: gid
- Blockcount: #blocks allocated to this file

File operations:

1. Open

```
fd ← open( "/usr/nerd/hello", O_RDONLY )
```

- a.
- b. O_RDONLY, specify how we want to access this file
 - i. Get to the file
 1. Directory lookup (starting at the root directory)
 2. Read inode if usr
 3. In usr directory, read in an inod of nerd directory
 4. Read nerd directory inode, find inode of file
 - ii. Cache inode for this file in buffer cache
 - iii. Check permissions
 - iv. Record the file as an open file, set up entry in open file table OFT
 1. Open File Table

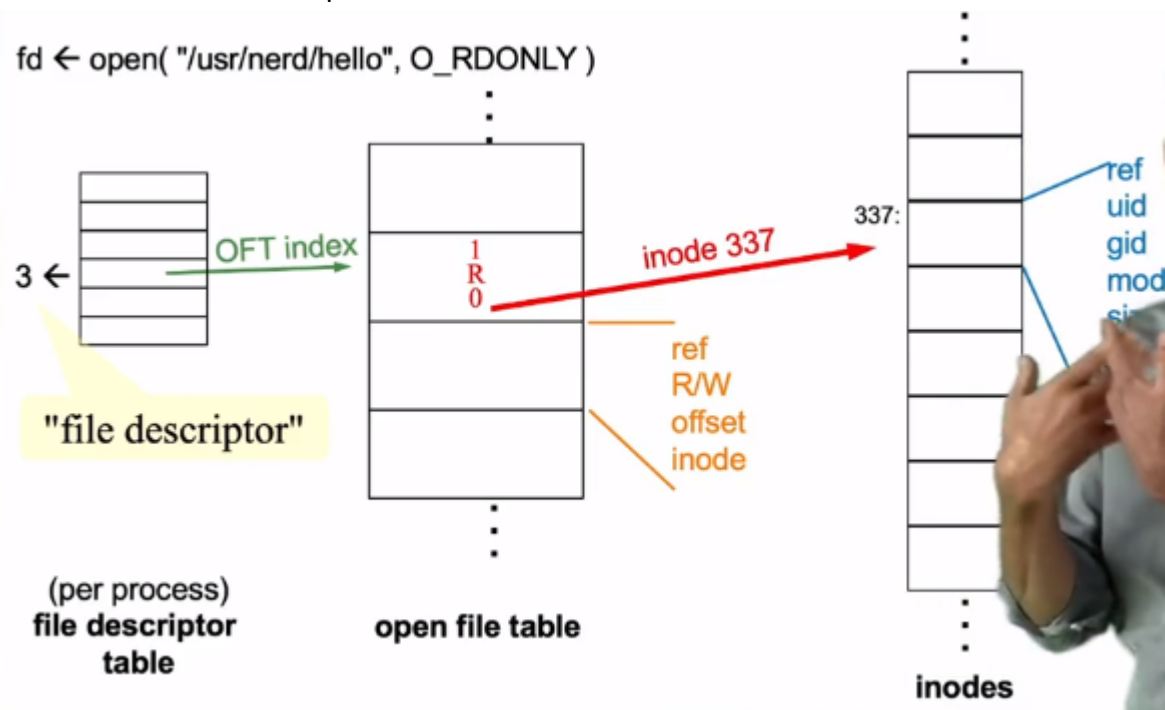


- 2.
3. An array, each entry representing an opened file

- v. Allocate a free entry in array, record attribute related to the open file
 - 1. Ref → how many processes related to this open file = 1
 - 2. R/W → how access = R
 - 3. Offset → where start reading from = 0 (start reading from 0 byte)
 - 4. Inode: which file this entry refers to = inode 337
 - In order to prevent returning the entire open file table to the processor, because don't want process to access other files
- vi. Setup entry in file descriptor table
 - 1. Go through FDT, find first entry empty
 - 2. Populate with the index on the open file table
 - 3. The index of the entry in FDT is returned to processor/application



- 4.
- vii. Return: "file descriptor" == fd



- Array of inodes: on disk/cached in memory
 - Stores info of map(file, disk)
- OFT: in memory → do not need to persist
 - Store info of opened file (inode, processors referring to this file...)

- FDT: one file per processor that is RUNNING
 - Store index to the open file table on an entry on this array
 - Return the index to the entry on this array to ensure isolation
- File descriptors
 - Non-negative integer
 - When application make system calls → kernel → file descriptors
 - Returned by kernel on all open + file creations
 - Can be used to identify, lseek, write...
 - Number of fd / processor is limited
 - Fd value: 0,1,2,...19
 - 3 predefined file descriptor
 - Stdin: standard input, 0
 - Keyboard typing in
 - Stdout: standard output, 1
 - Output to screen
 - Stderr: 2 standard error
 - Output to screen
 - Parent process forks a child
 - Child inherits ALL fd from parent

2. Read operation

- a. **byte_count = read(fd, buffer, buffer_size)**
 - Fd: which file reading from
 - Data read in to a buffer
 - Buffer_size == number of bytes we want to read
- b. Steps to perform:
 - i. Given fd, fd index into per process file descriptor table FDT → locate file table entry
 - ii. Check if the operation is allowed → consistent to the way the file is opened
 1. Read if open("dir",RDONLY)
 - iii. Locate inode from open file table OFT
 - iv. Based on info in inode, figure out the blocks need to be accessed
 - v. Check if the blocks are in buffer cache
 1. Yes: use
 2. No: buffer cache miss, need to read data to buffer cache
 - a. Slow, may have context switch
 - vi. Copy data from buffer cache → buffer specified by system call
 1. The buffer passed in is in the user address space
 2. OS accessing user address space, it can access buffer with Virtual Address
 - vii. Update file offset based on bytes read
 - viii. Update inode → change access time
 - ix. Return number of bytes read

3. Seek

- In rotating disk: head seek looking for data spot on platter
- This: set offset to a position in the file

offset = lseek(fd, offset, whence)

- - Fd: used to locating the FDT entry
 - Update file offset stored on the FDT

4. Close

RC = close(fd)

-
- Fd: used to index into file descriptor table to locate OFT
- Decrement reference count.
 - If reference count == 0:
 - Free up the entry

5. Delete file

RC= unlink(pathname)

- Can delete file without it being opened
- Steps:
 1. Locate directory entry using pathname
 2. Locate inode wrt to the path directory
 3. Decrement reference count
 - a. If ref count == 0
 - i. Free up inode by setting free bit in each inode
 - ii. Set bitmap in disk to free to remove directory entry
- What if a file is open but someone is trying to unlink a file?
 - Another counter used to keep track of the ref count for processes that have the file open
 - If a process is using the file, the deletion will happen after it closes the file

Redirection

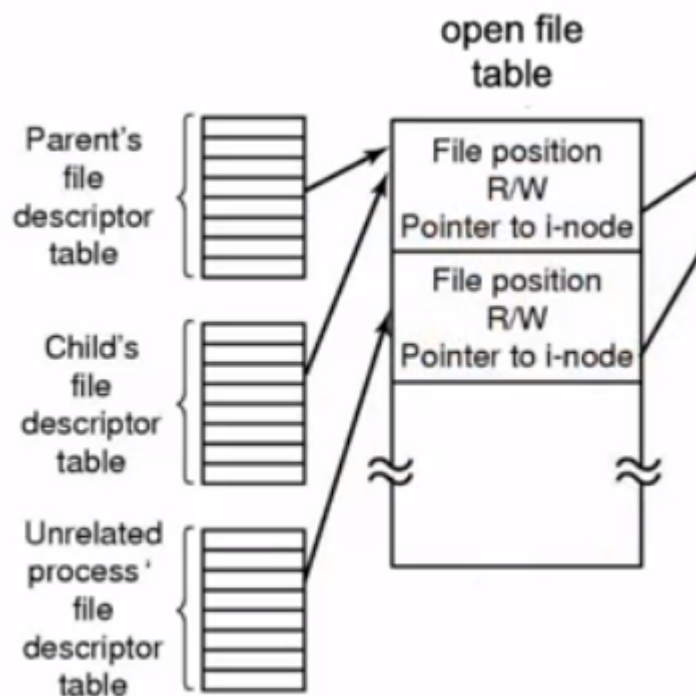
in Shell: /bin/cat < file1 > file2

- - Cat: read from stdin, write to stdout
 - Redirect input: < file1:
 - Cat is reading from file1 instead of keyboard
 - Redirect output > file2:
 - Output will be printed to file2 instead of screen
- When executing command when shell runs:
 - A child will be fork() to execute /bin/cat
 - Steps:
 1. Shell fork() create command
 2. Parent waits until child terminates
 3. After child is created:
 - a. close(0)
 - i. Close stdin in child so things typed in from keyboard are not tracked
 - b. open("file1",O_RDONLY)

- i. Locate inode for file1
- ii. Create entry in OFT
- iii. Search for an empty spot in DFT (starting from beginning, 0)
 1. Since 0 is just closed, the newly available one is 0
 2. Thus fd = 0
 3. So when cat reads from stdin, instead of keyboard input that has been closed, will be reading from file1
- c. close(1)
 1. Closes stdout default
- d. open("file2", O_WRONLY|O_RDONLY)
 - i. If it does not exist, allocate a new inode.
 - ii. Search through DFT, first available = 1
 - iii. Thus fd = 1
 - iv. Now stdout is going to file2
- e. execve("bin/cat...")

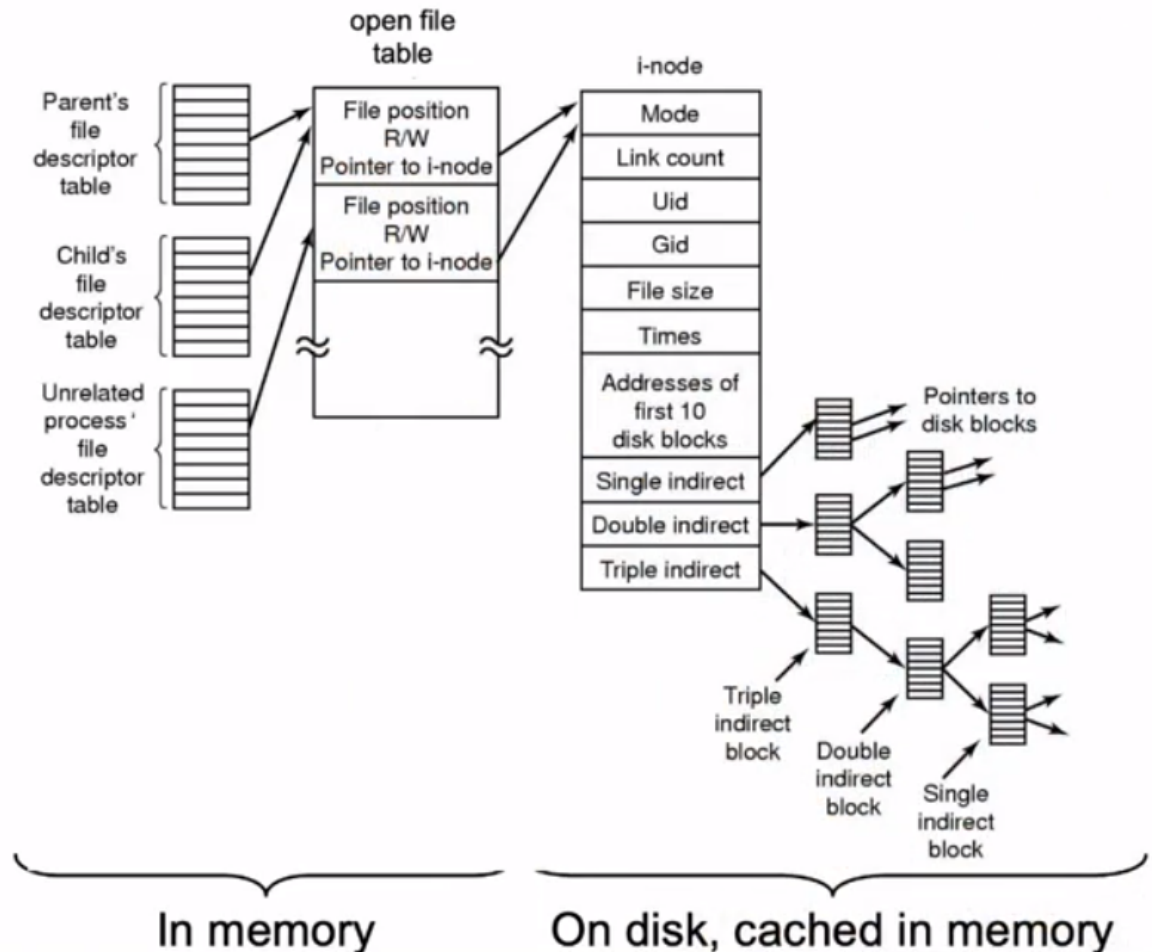
Sharing

- When fork(), fd is inherit to child from parent
- Reference count



-
- Child and parent both refer to the same entry in a OFT
 - Reference count = 2, both refer to the same open file
 - Offset is shared between the children and parent processes
 - When offset marked as 100, parent reads 100, offset on OFT = 200

- When child starts reading file, starts at 200
- Another process come and open same file
 - A NEW OFT will be created for that , pointer to inode may be the same
 - Offset is independent of the offset in nother OFT entries, even if they are referring to the same file
- 2 OFT can be pointing to the same inode



- Multiple processes can access same file, read and write concurrently

Concurrent access

- OS ensures strong consistency aka linearizable
 - 1 process issues a write, returned
 - Any read happening after that write should read the new written info as well
- What if offset is being updated by processes at the same time
 - Critical section, requires lock
- 2 independent processes, write() to different OFT but modifying same block data
 - Locks!
 - Tradeoff between concurrency and the locking structure

Consistency and crash recovery

- problem:
 1. Modifying so many structure

- Inode, OFT, bitmap for inode, bitmap for disk...
- 2. Write-back caching to improve performance of file writes
 - dirty blocks in memory not saved to disk on a crash → lost of data
- 3. Inconsistent file system on disk
- What is file system inconsistency?
 - There are several block write operations:
 - E.g. delete a Unix file with method A:
 1. Remove file directory entry in directory data block
 2. Mark inode as free in inode bitmap
 3. Mark file block as free in block bitmap
 4. Mark indirect blocks as free in block bitmap
 5. Update metadata in inode of the directory → timestamp change because things in this directory has been changed
 - If the system crashes after step 1?
 - Free data block is not updated → these blocks will still be marked as occupied → storage leak!
- E.g. delete a Unix file with method B:
 1. Mark inode as free in bitmap
 2. Mark file block as free in block bitmap
 3. Mark indirect blocks related to this file as free in block bitmap
 4. Update metadata in inode of the directory → change timestamp
 5. Remove file directory entry in directory data block
- System crashes after step 2?
 - Dangling pointer!
 - The FDT entry is still recorded as before, so have a pointer there
 - Pointing to empty
 - Pointing to new inode
- Method A > method B:
 - Try to mitigate through:
 1. Write metadata block synchronously to disk
 - a. Things related to structure of file system
 - b. Minimizes windows of time that crash can happen
 2. Write data asynchronously
 - a. Other data blocks are stilled written out asynchronously → help system performance
 - b. If these blocks are lost, does not affect the file system

Crash recovery

- Metadata of last file system operation may not have reached disk → inconsistent
- To solve:
 - When reboot, restore file system consistency
 - “Crash recovery”
 - A FFCK system gives a full scan of the file system
 1. Check bitmap indicating which is free/occupied
 - a. Make sure free → free block
 - b. Go through all items make sure item → not free bit

2. All directories
 - a. Check inode referring to, make sure reference count is consistent
3. Block inode id
 - Focuses on the system file, data block not so much
 - Takes a long time → bc disk capacities increase > disk throughput

L37 think time

1. Describe the operations needed to write the string "xyz" to an existing file "/a/b" in Unix
 - a. Locate the inode for root directory, read directory block of /
 - b. Locate the a directory inode, read directory block of a
 - c. Locate the b directory inode, read directory block of b
 - d. If the first block of b exists
 - i. Cache the block to the buffer cache
 - e. Else
 - i. Allocate a block on disk
 - ii. Allocate a block in buffer cache for this empty disk
 - f. Update block in buffer with the string "xyz"
 - g. Update file position
 - h. Update inode of b with new timestamp, file size
 - i. Schedule writing of file block, inode, block bitmap to disk