Intro, Threads & Concurrency

- What are the three views of an operating system? Describe them.

Application View - provides an execution environment for running programs
System View - manages the hardware resources of a computer system
Implementation View - OS is a concurrent, real-time program

- Define the kernel of the OS. What things are considered part of the OS but not the kernel?

Kernel: The OS kernel is the part of the OS that responds to system calls, interrupts and exceptions.
OS: The OS as a whole includes the kernel, and may include other related programs that provide services for applications. e.g. utility programs, command interpreters, programming libraries.

- What are some advantages of allowing more than one thread to exists simultaneously?

1. programs can run faster
2. if one thread has to block, another may be able to run

- What is contained in a thread context?

Each thread has its own private register contents and stack, and they share access to the program's global variables and heap.

- Describe what happens during a context switch? What is dispatching?

  During a context switch:
1. decide which thread will run next(scheduling)
2. save register contents of current thread
3. load register contents of next thread

- What is the difference between thread_yield() and thread_sleep()?

thread_yield: running thread voluntarily allows other thread to run
thread_sleep: the running thread blocks,waits for some particular event to wake it up
thread preempted: running thread involuntarily stops running

- How is an involuntary context switch accomplished? Define quantum.

Preemption means forcing a running thread to stop running so that another thread can have a chance. This is normally accomplished using interrupt.
Quantum: an upper bound on the amount of time that a thread can run.

- Describe what happens when an interrupt occurs. Where do interrupts come from?

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

When an interrupt occurs, the hardware automatically transfers control to a fixed location in memory, and call interrupt handler:

1. create trap frame to record thread context at the time of the interrupt
2. determines which device caused the interrupt and performs device-specific processing
3. restores the save thread context from the trap frame and resumes execution of the thread

Key point:
- The preempted thread changes the state from running to ready, and it is placed on the ready queue.

Thread_switch (high-level context switching code):
1. Finds the next thread to run
2. Calls switchframe_switch to perform the low-level context switch

- Thread_switch(caller) save the t-registers prior to calling switchframe_switch
- switchframe_switch(callee, generate swtichframe, which should be drawn on stack)
1.save s-registers which data is from OLD thread, and
2.load NEW thread data into s-registers
3.pop the switchframe stack from off of the NEW THREAD's stack.
4.jump to NEW thread thread_switch return address

Context Switch can be caused by: Thread_yield, Thread_exit, Wchan_sleep, Preemption.

Synchronization

Keyword *volatile*:
- Value in memory is NOT accurate
- Compiler does NOT optimize value into register
- *volatile* forces the compiler to load and store the value on every use
- Does NOT provide synchronization; ONLY turns off that optimization to prevent these optimizations from causing race conditions

- Multi-threaded code requires **sequential consistency** for shared variables
○ I.e., load/store order is same for optimized and non-optimized code

- To prevent race conditions, we can enforce **mutual exclusion** on critical sections in the code.
- any primitive type value insides mutual exclusion should be volatile to prevent race condition!

- Hardware-Specific Synchronization:
Test and Set Examples:
x86: xchg src, addr
SPARC cas: cas addr, R1,R2
MIPS: load-linked and store-conditional
LL returns the current value of a memory location, where subsequent SC to the same memory location will store a new value only if no updates have occurred to that location since the load-linked

- **Locks**:
Spinlocks **spin**(busy waits), Locks **block**.
When a thread blocks, it stops running:
1. the scheduler chooses a new thread to run
2. a context switch from the blocking thread to the new thread occurs
3. the blocking thread is queued in a wait queue(NOT ready list)
Eventually, a blocked thread is signaled and awakened by another thread

- **Wait Channels**
Wchan_sleep: * blocks calling thread on wait channel
* causes a **context switch**, like Thread_yield
Wchan_wakeall: unblock **all** threads sleeping on wait channel
Wchan_wakeone: unblock **one** thread sleeping on wait channel
Wchan_lock: prevent operations on wait channel - No wakeup on wait channel before Wchan_sleep get called

- **Semaphores**
    - What is a semaphore? Which two operations does it support? Are they atomic?

A semaphore is a synchronization primitive that can be used to enforce mutual exclusion requirement.
A semaphore is an object that has an integer value(let's call it sem), and that supports two operations:
- **P**: if sem > 0, decrement the value. Otherwise, wait until sem > 0
- **V**: increment sem

P and V are *atomic*

- What is the primary difference between a lock and a binary semaphore?

- A **binary** semaphore is just a semaphore with **one** resource, it protects a critical section much like a lock but with NO concept of ownership - which means the resource can be released by anybody.
- A normal semaphore lets a number of threads in, it's for resource counting

- **Condition Variables**

    - What is a condition variable? How are they used in conjunction with locks?

CV is a synchronization primitive. Each CV is intended to work together with a lock.
CVs are **only** used from within the critical section that is protected by the lock
- Note: signalling\broadcasting to a CV with no waiters has NO effect
The signal should happen inside of the critical section **protected by the lock**.
Hence, the signalling thread should be the **owner** of the lock passed in.

- Deadlock:
A deadlock is a situation wherein two or more competing actions are each waiting for the other to finish.

    - Can deadlock occur with only one thread?

No, def is above.

Two techniques for Deadlock Prevention:

1. No Hold and Wait: A thread must make a single request for all of the resources.

   Advantage:

   ● Threads **never** own locks while they are blocked waiting to acquire another thread

   Disadvantage:

   ● Threads spin while trying to repeatedly acquire a set of locks

2. Resource Ordering: Order the resource types, and require that each thread acquire resources in increasing resource type order.

Processes and the Kernel

    - What is a Process?

A process is an environment in which an application program runs.

    - What information is contained within a process?

- one or more threads

- virtual memory, used for the program's code and data

- other resources, e.g. file and socket descriptors

* Processes are created and managed by the kernel.

* Each program's process **isolates** it from others programs in other processes

  - Can threads from one process access memory from another process?

No, see above.

Fork:

- *fork* creates a new process(the child) that is a **clone** of the original(the parent)
- after *fork*, both parent and child are executing copies of the same program
- virtual memory of parent and child are identical at the time of the fork, but may diverge afterwards
- *fork* is called by parent, but returns in both the parent and the child

  * parent and child see different return values from *fork* (parent: get child pid,

    child: get 0)

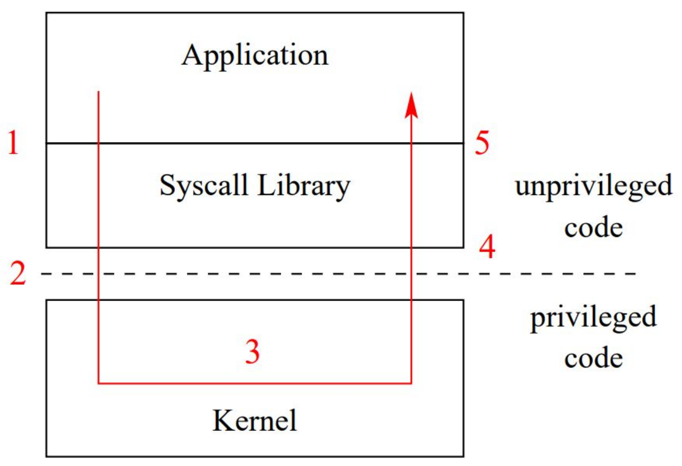
- _*exit* terminates the process that calls it
- process can supply an exit status code when it exits.
- kernel **records** the exit status code in case another process asks for it(via *waitpid*)
- Parent CAN DIE before Child
- *waitpid* let a process **wait** for another to terminate, and **retrieve** its exit status code
- Only PARENT can call *waitpid* on CHILD(NO grandchild)
- *execv* changes the program that a process is running
- the calling process's current virtual memory is destroyed
- The process gets a **new** virtual memory, initialized with the code and the data of the new program to run
- After *execv*, the **new** program **starts** executing

System Calls:

- System calls are the interface between processes and the kernel.

- A process uses system calls to request operating system services.

System Call Software Stack (again)



Kernel code runs at a **higher** level of ***execution privilege*** than application code

- privilege levels are implemented by the CPU

Application programs CANNOT directly call kernel functions or access kernel data structures

- What is an exception? What is the difference between an exception and a system call? And exception and a interrupt?
- How does a program make a system call?
- Only two things that make kernel code run

1. **Interrupts**

   * interrupts are generated by devices(hardware)

2. **Exceptions**

   * exceptions are caused by instruction exception(software)

- What can the kernel do in privileged execution mode that user programs can not directly accomplish?
- Handle interrupts and exceptions

- Interrupt handlers are part of the kernel

- Exception handler is part of the kernel

- Exceptions are detected by the CPU during instruction execution

- When an interrupt/exception occurs, the processor switches to privileged execution mode

E.g.: system call: fork, exception handler in kernel: sys_fork

How System Calls Work?

- Application calls library wrapper function(syscall) for desired syscall

- The kernel's exception handler checks the syscall codes to determine which syscall has been requested.

- Syscall codes must be known by both the kernel(check which syscall should be requested and application(store into specific register).

- System Call Parameters: On the MIPS:

1. parameters go in registers a0-a3
2. result success/fail code is in a3 on return
3. return value or error code is in v0 on return

a3: 0(success) => v0: return value

a3: 1(fail) => v0: error code

User and Kernel Stacks

- Every OS/161 process thread has two stacks, **User Stack and Kernel Stack**, although it only uses one at a time
- **User(Application) Stack:** used why application code is executing

  * the stack is located in the application's virtual memory

  * the **kernel** creates this stack when it sets up the virtual address memory from the process

- **Kernel Stack:** used while the thread is executing kernel code, after an exception or interrupt

  * this stack is kernel structure

  * this stack also holds **trap frames** and **switch frames**(cuz the kernel creates trap frames and switch frames)

Exception Handling in OS/161

* *mips_trap* gets called after *trap frame* was created

- The mips trap function is called to handle system calls, exceptions, and interrupts. How does it differentiate between these three cases?
- *mips_trap* determines what **type** of exception this is by looking at the exception code(interrupt? system call? address translation exception? something else?)

- When returning from a system call, the program counter is not automatically advanced by the hardware. Why is this?
- The program counter for the user process is on the trapframe, and it was saved at the time of an exception/interruption. The kernel that handles the syscall doesn't use the same PC

Virtual Memory: Pre-Midterm:

Physical Addresses:

- A system uses physical address and virtual addresses. How many physical address spaces are there in a computer?

  If physical addresses have P bits, the max amount of addressable physical memory is 2^P bytes.(Virtual address spaces is similar)

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

Virtual Addressees:

- The kernel provides a separate, private virtual memory for each process

- The VM of a process holds the code, data, and stack for the program that is running in that process

- How many virtual address spaces?

  If virtual addresses are V bits, the max size of a VM is 2^V bytes

- Running applications see only VM

- Each VM is mapped to a different part of physical memory

Address Translation:

- Address translation is performed in hardware, using information provided by the kernel

Address Translation fro Dynamic Relocation:

- CPU includes a **memory management unit(MMU)**, with a **relocation register** and a **limit register**

- relocation register holds the physical offset(R) for the running process' virtual memory
- limit register holds the size L of the running process' virtual memory

- Translation is done in hardware by MMU

- In dynamic relocation, when does the value of the relocation register in the MMU change?
- The kernel maintains a separate R and L for each process, and changes the values in the MMU registers when there is a context switch between processes

- Each virtual address space corresponds to a ***contiguous range of physical addresses***

- The kernel is responsible for deciding *where* each virtual address space should map in physical memory

- The OS must **track** which parts of PM are in use, and which parts are free
- OS must allocate/deallocate variable-sized chunks of PM
- this creates potential for ***fragmentation*** of physical memory

- What is the difference between external fragmentation and internal fragmentation?

Internal fragmentation is when there is wasted space within the allocated memory. For example a fixed size of memory is allocated to a process. but the amount allocated is bigger than needed. External fragmentation is wasted space between allocated memory, this is seen in the dynamic relocation example. The total free physical addresses space is greater than needed, but there is no contiguous range of physical addresses satisfies needed addresses.

Remember: 2^{12} bytes = 4KB

Paging: Physical Memory

Physical memory is divided into fixed-size chunks called **frames** or physical pages.

Paging: Virtual Memory

Virtual memories are divided into fixed-size chunks called **pages**. Page size is equal to frame size.

Paging: Address Translation

Each page maps to a different frame.

Address Translation in the MMU, Page Tables:

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

- Look up the PTE in the current process page table, using page number. If the PTE is NOT valid, raise an **exception**

- PTE may contain other fields(write protection bit used for read-only page, reference bit, dirty bit)

- A page table has one PTE for each page in the virtual memory
  - page table size = #pages * size of PTE
  - #pages = VMsize / PGsize

Page Tables are kernel data structure.

Paging reduces external fragmentation.

Kernel:

1. Manage MMU registers on address space switches(Context switch)

2. Creates & manage page tables

3. allocate/deallocate PM

3. Handle exceptions raised by the MMU

MMU(hardware):

1. Translate virtual addresses to physical addresses

2. Check for and raise exceptions when necessary

TLBs(Translation Lookaside Buffer):

Hardware-Managed TLBs:

- TLB is a small, fast, dedicated cache of address translations, in the MMU
- Each TLB entry stores a (page# -> frame#) mapping

● If the MMU cannot distinguish TLB entries from different address spaces, then the kernel must clear or invalidate the TLB on each context swtich from one process to another

● the MMU handles TLB misses, and must understand kernel's page table format

Software-Managed TLBs:

MIPS has a Software-Managed TLBs.

In case of a TLB miss, MIPS raise exception, the kernel must handle the exception and the instruction that caused the exception is re-tried.

- What is the difference between a hardware-controlled TLB and a software-controlled TLB? Which is the MIPS TLB?

Generally speaking, such hardware implementations are faster. However, the downside is that hardware dictates the form of the PTE. Which restricts what we can do in software.

The software managed TLB handles TLB misses by throwing exceptions and letting the kernel of the OS handle things. This gives us more **flexibility** in implementation, but, is generally slower.

Segmentation:

● Instead of mapping the entire virtual memory to physical, we can provide a separate mapping for **each segment of the virtual memory** that the application uses

● With K bits for the segment ID, we can have up to:

- 2^K segments
- 2^{V-K} bytes per segment

- Fragmentation of physical memory is **possible**
- Translating Segmented virtual addresses approach:

  1. MMU, similar as dynamic relocation

  2. segment table

Two-Level Paging:

Directory table and page table