OS Midterm Review

By Ethan Richards

Review these at your own risk! This is just what I found from the studying materials, my notes, and the slides. It may not be comprehensive. Hit me up over text (7202407369) or email (erichards@mines.edu) for any questions/comments/concerns/additions :)

Chapter 1:
Computer Systems

Registers

Data registers: Registers that store data

Address registers: Registers that store memory addresses

Instruction registers: Registers that store words/instructions

Program Counter (PC): Stores the address of the instruction being executed (increased/decreased after instruction is done)

Takeaway: Registers are temporary memory for the CPU.

Interrupts (1/2)

Interrupts are mechanism which allow other processes/modules to interrupt the normal sequential execution of the processor.

Why? We want to improve processor utilization and reduce waiting time. I/O devices are also slower than the processor, so sometimes we need to pause to wait for the device.

Interrupts (2/2)

Properties:

- The interrupt handler is a part of the OS code.
- There can be an interrupt inside of an interrupt; a priority scheme decides which should be processed first.
- **Interrupts are checked at the end of the execution cycle (after the execute stage)**

Types of Interrupts

Program: Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, etc.

Timer/clock: Generated by a timer within the processor. (clock)

I/O: Generated by an I/O controller

Hardware: Power failure/memory parity error

Memory Storage/Hierarchy

Why? We have design constraints on memory in a computer: how much, how fast, how expensive? If capacity is provided, applications will want to use it.

Faster access time = greater cost per bit

Greater capacity = smaller cost per bit

Greater capacity = slower access speed

See Chapter 1 Slide #37 for the full pyramid image and description.

Cache/Cache Memory

A block of “temporary” memory used because we want to access memory more often and more quickly.

- Cache is invisible to the operating system.
- Has algorithms like Least Recently Used (LRU) to shuffle old things out of the cache.
- Exploits temporal & spatial locality.

Locality (1/2)

Temporal locality: Limited range of memory addresses are requested repeatedly over a period of time; memory is served faster because the data being accessed stays in the cache during this time period.

Locality (2/2)

Spatial locality: Memory addresses are requested sequentially and are served faster because the cache block is being fetched already with adjacent addresses.

We get locality "for free" because memory references by the processor tend to cluster.

Effective Access Time (EAT)

$$T_S = H \times T_1 + (1 - H) \times (T_1 + T_2)$$

T_S = average system access time

T_1 = access time of M1 (cache, disk cache)

T_2 = access time of M2 (main memory, disk)

H = hit ratio (fraction of time reference is found in M1)

Chapter 2:
Operating Systems

Operating Systems

"A program that controls the execution of application programs"

"An interface between applications and hardware"

Goals of an OS: convenience, efficiency, ability to evolve & security.

The key responsibility of an OS is to be a resource manager!

OS Interfaces

- **Application Programming Interface (API)**: Standard libraries, useful because they provide abstractions for system calls.
- **Application Binary Interface (ABI):** Methodology of how the binary should work
- **Instruction Set Architecture (ISA)**: RISC-V, MIPS, etc.

Kernel Mode

Why? We need to access protected areas of memory or execute potentially dangerous instructions.

Programs execute in user mode but we can use **system calls** as an abstraction to doing things in kernel mode.

- Kernel mode modifies OS structure, files, memory, etc.
  User mode **doesn't (and can't)**.

Multiprogramming

Multiprogramming is like (but not exactly) multitasking: we switch between programs to **increase utilization and decrease response time**, for instance, while we wait for I/O or another event.

Why? Because we want to be able to listen to music while we code! (e.g., Belviranli's Spotify + Visual Studio Code example)

Processes (1/4)

A process is…

- A program in execution
- An instance of a running program
- A unit of activity characterized by a single thread of execution

Processes (2/4)

Process Management: We need to keep track of processes and their state; we can't deal with something that isn't named!

Identifying features of a process are passed through by **context**; the entire state of the process at any instance is contained in its context.

Processes (3/4)

Processes contain three components:

- An executable program
- The associated data needed by the program
- The **execution context** of the program

Processes (4/4)

Execution context (process state) of a program:

- Holds internal data which allows the OS to supervise and control
- Includes contents of registers
- Includes priority of the process
- Includes whether or not it’s waiting for I/O or interrupts

Threads & Multithreaded Processes

Why? Switching between threads is quicker than processes and because we want to exercise multiprogramming.

Threads are a unit of execution that you can assign to different cores.

Multithreading: The ability of an OS to support multiple, concurrent paths of execution within a single process.

Chapter 3:
Process Description & Control

Swapping

What if we run out of process memory?

We swap things! Swapping is the act of moving part or all of a process from main memory to disk. (The OS manages this too!)

The swapper (Process 0) is part of the kernel and is known as the system_process.

Process Switching

Why? We don't want to waste time and resources!

- When a job is waiting for I/O, the processor can **switch** to the other job and run it until I/O completes, which increases utilization.

The **dispatcher** is a small program that handles this; it switches the processor between processes.

Process Image

The contents of a process! Containing..

- The program (aka the instructions, aka text)
- Data (string literals and such, think back to comporg)
- Context: other metadata from the system (PC)

Basically, enough to resume a program in a timesharing system.

Process Control Block (PCB)

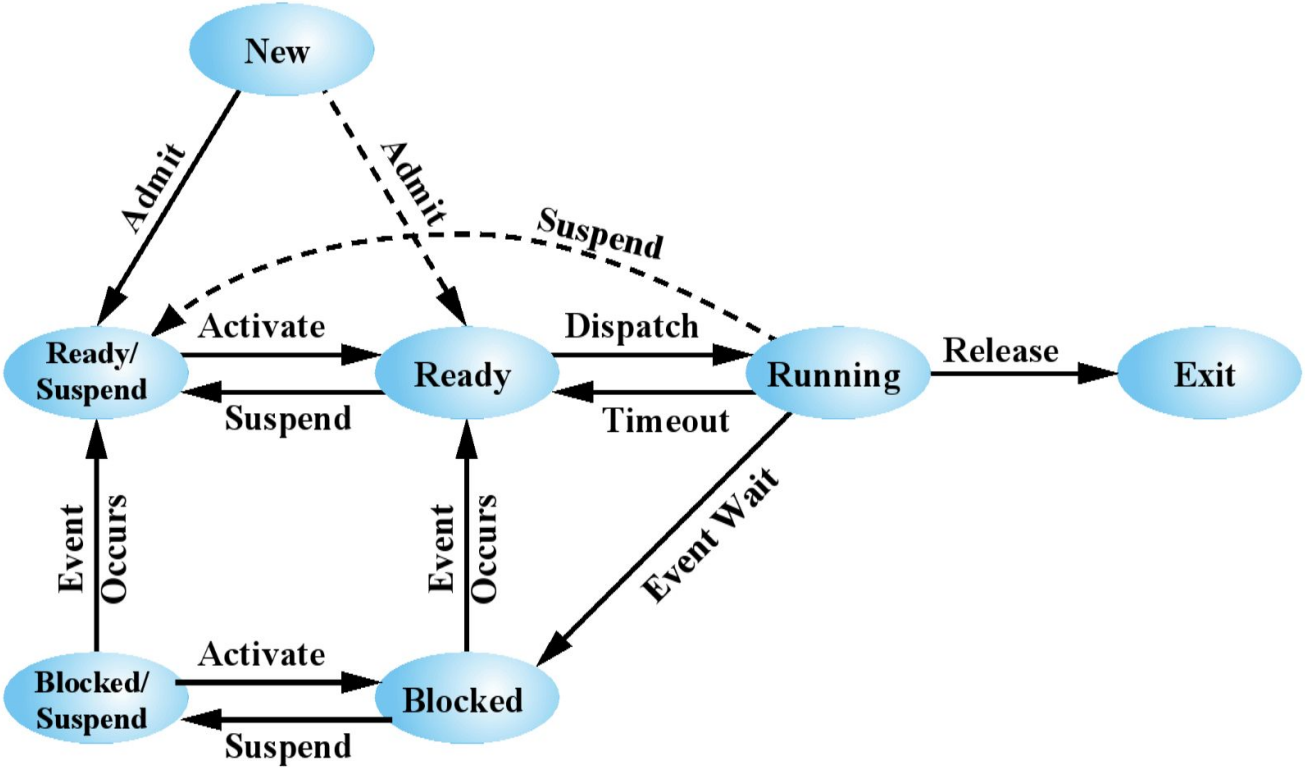
- aka the **process context**; Contains all of the information about a process that is needed by the OS
- Blocks are read and/or modified by virtually every module in the OS
- Defines the state of the OS
- **Only created and managed by the OS so that the OS can manage your processes.**

Ready & Event Queues

A queue of processes which are ready to be run. Queues are handled in virtual memory, so they can’t run out of space.

When an event comes in, processes in the event queue can transition to ready state (to switch to running eventually).

Five-ish State Process Model



Transitions

States: New, Ready, Running, Blocked, Exit (+ Ready/suspend, Blocked/suspend)

- **Admit**: New -> Ready
- **Event Occurs**: Blocked -> Ready
- **Dispatch**: Ready -> Running
- **Release**: Running -> Exit
- **Timeout**: Running -> Ready

Transition Handlers

Dispatcher: Ready -> Running (aka dispatch/timeout)

Interrupt Handler: Blocked -> Ready (aka an event occurred)

Swapper: Suspended -> activate (aka activate/suspend)

Chapter 4:
Threads

Why threads?

- Processes are more like a container of data, whereas threads are the actual unit of execution
- All threads are spawned from the same process and share the same data

Big Idea: Processes are meant for isolation; threads are meant for sharing.

Threads vs. Processes

- **Creating a new thread is much easier than forking and executing a new process!**
- Terminating threads also takes less time than terminating a process
- Switching between two threads is easier than switching between two processes
- Better efficiency in communication between programs

Thread Switching

Why do threads switch faster than processes?

Threads switch faster because threads are all spawned from the same process and share the same data; the thread control block (TCB) is also smaller than a process control block (PCB).

Note: Switching between threads from two different processes is the same as switching between two processes.

Properties of Threads

- **A thread can't exist without a process!**
- **Threads inherit the state of their parent**
  - You can only suspend processes (not threads); suspending a process involves suspending all threads of the process.
- Thread states: Running, ready, blocked
- Thread change of states: Spawn, block, unblock, finish

Thread Control Block (TCB)

- Execution state (running, ready, etc.)
- Saved thread context when not running
- An execution stack
- Some per-thread static storage for local variables
- Access to the memory and resources of its process

Each thread has its own user stack and kernel stack.

The Main Thread

- The main thread is spawned by whatever function starts the program (aka the **main()** function in most languages).
- Threads can run in parallel to the main thread, but there is only one main thread.
- All threads are spawned from the same process and share the same data!

Multithreading

- **The ability of an OS to support multiple, concurrent paths of execution within a single process**
- Threads run concurrently, which saves time.
- This gets into the idea of **concurrency**, which deals with threads accessing the same data at the same time.

User Level Threads (ULTs)

Thread management is done by the application; the kernel/system is not aware of the existence of threads. This is good because we don't require kernel mode privileges.

- A naive implementation.. this is overwhelmingly bad mainly because of security and that we can't do multi core executions (which ruins the point of threading). We want to use **KLTs** (or a hybrid of the two) instead.

Kernel Level Threads (KLTs)

Why? ULTs kinda suck, so thread management is done fully by the kernel in this case. (Windows does this).

This is better because the operating system sees the threads *and* the process. If one thread is blocked, more can continue (which was not the case with ULTs).

Amdahl's Law

- ..tells us the theoretical speedup of the execution of the whole task
- **S_latency(s) = 1 /[(1 - p) + (p / s)]**
- Where s is the speedup part of the task, p is the proportion of execution time
- **Can't speed up an entire program; whatever you speed up has to affect the entire program.**

Parallelism

Parallelism: benefit from using multiple threads on multiple cores (aka servers/machines) simultaneously to speed up execution.

Note: Parallelism speedup is limited, some tasks are serial (like initialization and storing of data) and can't benefit from parallelism; the application is the limiting factor.

Chapter 5:
Concurrency

Race Conditions

A **race condition** occurs when multiple processes or threads read and write **the same** data items.

Why is this bad? Because of memory consistency problems! Two threads might try to modify the same data in an uncontrolled fashion.

Critical Section

The **critical section** is where we're trying to do something to the data with multiple processes/threads at the same time.

Mutual Exclusion

The idea that processes are blocked from reading/writing at the same time, which solves memory consistency problems.

Links heavily into the idea of the **critical section** of memory; **critical sections for the same resource should not (and cannot) execute at the same time!**

Mutual Exclusion Requirements

- Must be enforced (i.e., with **semaphores**!)
- A process that halts must do so without interfering with other processes
- A process must not be denied access to a critical section when there is no other process using it

Mutual Exclusion Requirements

- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only.
  - **If it was infinite, nobody could access the resource because its critical section is being used!**

Semaphores (1/2)

A variable with a special integer value representing three operations:

- Initialization: nonnegative integer value (1 by default)
- semWait decrements the value
- semSignal increments the value

semSignal and semWait are abstractions of the OS calling the compare & swap instruction.

Semaphores (2/2)

If the value is negative, that's how many people are waiting in the queue. If the value is 1, nobody is in the queue. If the value is 0, people are leaving the queue.

Semaphores ensure that nobody enters the critical section at the same time!

Projects & C

fork

- Allocates a slot in the process table for the new process
- Assigns a unique ID number to the child process
- **Makes a logical copy of the context of the parent process (shell)**
- **Returns the child process ID to the parent process, and 0 to the child process**

Child-parent processes

The return value from fork() differs between child and parent processes; this is necessary to identify the difference between the two processes.

Parent process knows the process ID, child process doesn't necessarily need to know (it will be 0 so they can identify themselves)

execve

exec() morphs the process: it replaces the current process with a brand new program.

(*execve* is just exec under some PATH environment)

pipe

A way to transfer data between parent and child processes.

The pipe function creates a new pipe and populates the passed in array's filedescriptors. fd[0] = read, fd[1] = write

dup

Creates a new file descriptor in the file descriptor table (in the PCB); allows for pointing of stdin/stdout to the file descriptors (which is what we did with the shell project).