Os midterm review

Chapters 1-6, 13

Lecture slides 1-11

Does not include memory management

Use slides and notes and the assignments (written) to study

**Practice Exam**

1. D
2. D
3. D
4. D
5. B
6. C
7. C (7 number 2) E
8. B
9. A
10. A

**Short Answers**

1. Schedule function operates the ready queue and run queue (ready waiting to be run) (run is running). Two states a process can be in, ready and run. Schedule will update which processes are running and happens at some scheduled interrupt (hardware clock). Switch\_to saves the current state of the program (register, stack pointer) and switches to a different program with a new state.
2. DMA. Copy all the memory and then interrupt at the end. Interrupt driven I/O. Saves from the CPU having to wait for the I/O to return data. Instead of the CPU asking the devices If they need something (polling) it does stuff until a device sends an interrupt.
3. Post diagnostic, check hardware, BIOS, find boot device, find bootstrap program, boostrap finds the kernel, kernel loads, OS takes over
4. P2 can call wakeup() before P1 calls sleep(), then P1 will never terminate
5. I/O intensive is a lot of hardware operations, like disk accesses, CPU intensive is stuff like large calculations. From a scheduler, and I/O process is going to block a lot. CPU intensive just doesn’t block. I/O bleeds time, CPU takes all of it. You can run a CPU intensive task and an I/O intensive task at the same time. I/O intensive will be called first, so we can free time for CPU.

**Lecture Slide 1**

* An operating system is a layer of software between applications and hardware that provides useful services to applications
* Provides users with an easy way to use the computer
* Efficient use of resources
* Batch Processing:
  + Execute a pre-defined collection of programs called a batch
  + No human interaction
  + All programs execute in order, and are compute bound, cpu is locked on them
  + CPU does not switch
* Multiprogramming
  + When CPU is idle, start another program (idle from I/O block)
  + Normally when an I/O blocks the CPU is idle, a scheduler switches CPU
* Context Switch
  + Name for switching CPU from one program to another
  + Save state
  + Load new state
  + Overhead is present
* With multiprogramming a CPU bound program may delay the execution of other programs
* Multitasking
  + CPU rapidly switches between programs
  + Efficent CPU usage, no idle time if blocked
  + An inifinite loop can’t hog CPU
  + Lots of context switch overhead
* Cooperative VS Preemptive
  + Cooperative: programs quickly and voluntarily yield CPU (poor isolation)
  + Preemptive: OS forces programs to give up CPU (on time slice)
  + All modern OS are preemptive

**Lecture Slide 2**

* Three design issues, System Boot, Protecting OS from apps, System Call API
* System Boot
  + Bootstrap program:
    - Locates kernel, loads it into main memory, starts execution
    - 2 step process
    - Runs diagnostics to determine state of machine
    - Initialize registers, main memory, controllers
    - Starts OS
  + When CPU receives reset
    - IR is loaded with predefined memory location that contains the bootstrap program
    - In ROM: no initialization cannot be infected easily
* System Boot (large system)

1. Power on self test (POST) from rom
   1. Check hardware, cpu and memory to make sure OK
2. BIOS looks for a device to boot from
   1. May be prioritized to look for a USB or CD before disk
   2. Can boot from network
3. BIOS finds a disk to boot from
   1. Looks at Master Boot Record (MBR) in sector 0 of disk
   2. Containts primitive code for later loading and partition table listing the partition or bootloader location (512 bytes intel)
4. Primitive loader loads the secondary stage bootloader
   1. Can select among multiple OS’s
   2. Selects OS bootloader finds partition and loads the kernel

* Protecting OS from apps
  + Processors have a hardware mode bit that identifies if we are in user or kernel mode
    - Supervisor Mode (mode bit = 0): processor can do everything
    - User mode (mode bit = 1) processor can execute a subset
  + There are privileged instructions that can only happen in supervisor mode, these include I/O and privileged load and store instructions
  + Two memory spaces, User and System
  + Some OS support more modes
    - 0 = supervisor, 3 = lowest, 1 and 2 unused
* Trap Instruction
  + Used to switch from user to supervisor mode, entering OS
    - Trap sets mode to 0, also called syscall, mode is changed back to 1 on return
  + Anything that invokes a trap is a system call
  + Trap indexes into a trap table (kernel) and runs the function pointed to from the table
    - Process of indexing is called dispatching
    - Jump table or branch table synonyms
    - A trap is a software interrupt
    - Preforms specific system call
* System calls need to pass paramaters
  + Pass them in registers
  + Stored in block in memory and block address passed in register (limited)
  + Parameters pushed onto stack and popped off by OS
  + Using block / stack unlimited parameters
* Trap includes
  + Process control
  + File management
  + Device management
  + Info management
  + Communications

**Lecture Slide 3**

* Device Manager
  + Controls Operation of I/O devices
  + Issues commands, catches interrupts, handles errors, easy to use interface
* Everything Supports
  + Open, close, read, write, set, stop, etc.
* Blocking vs. Non-blocking
  + Blocking system call: process put on wait queue until I/O completes
  + Non-blocking system call: returns immediately with partial number of bytes transferred, e.g. keyboard, mouse, network
* Synchronous versus asynchronous
  + Asynchronous returns immediately, but later, the full number of bytes requested is transferred
* Ioctl and fcntl
  + Allows user-space to configure stuff on I/O, set speed eject, ect.
  + Int ioctl(int fd, int cmd,…); Invokes system call to execute device specific cmd on I/O device fd
  + Used for stuff that can’t be done with normal syscalls, direct to correct driver
  + Avoids new syscalls for each device
* Device Driver
  + Support device system call interface
  + An I/O syscall traps to kernel, invokes trap handler, indexes trap table, runs correct device driver
    - BUSY = 0, DONE = 0 > Idle
    - BUSY = 1, DONE = 0 > Working
    - BUSY = 0, DONE = 1 > Finished
    - BUSY = 1, DONE = 1 > Undefined
    - 2 bits for all this
* Polling I/O Busy waiting
  + Os spins in loop twice
  + Check for idle
  + Check for finish
  + Wastes CPU cycles
* We want to overlap CPU and I/O
* 3 Types
  + Direct I/O with polling
    - OS device manager busy-waits
  + Direct I/O with interrupts
    - More efficient than busy waiting
    - CPU incorporates hardware interrupt flag
    - Whenever a device is finished with a read/write, it communicates to the CPU and raises the flag
    - Upon interrupt, CPU interrupts execution and invokes handler
      * Save processor state
      * Find device causing interrupt
      * Jump to device hangler
      * Reenable interrupts
    - CPU checks interrupt flag every fetch/execute cycle
  + DMA with interrupts
    - Solves I/O problem of too many interrupts
    - Bypass CPU for large copies and raise interrupt at end of data transfer
    - Uses special processor, Direct-Memory-Access (DMA) controller
    - Operates memory bus directly, without help of CPU
* DMA and CPU need to play nice and share memory
  + Burst mode: while DMA is transferring CPU is bocked from memory
  + Interleaved Mode: DMA transfers one word to/from then CPU accesses, then DMA, ect
  + Transparent mode: DMA only transfers when CPU is not using memory (efficient but difficult)
* Memory Mapped I/O
  + Non mapped needs to be accessed with special instructions and dealt with
    - Write and read from registers
  + Port-mapped
    - Can only store / load
    - Don’t have full range of operations
    - Have to copy register into memory, add one, copy back
  + Memory mapped
    - Mapped to system address space
    - Normal instructions to load in and out of memory
    - Memory management unit maps values to/from device registers
  + Typically devices mapped into lower memory
* Device independent calls
  + Set of syscalls that an application can use to invoke I/O opetations
  + Certain devices have certain calls
  + Uses trap table to switch to specific device function
* Adding new device
  + Write funtions
  + Add new case clause to switch statement
  + Compile the kernel and drivers
* PROBLEM: Need to recompile kernel everytime

**Lecture Slide 4**

* Kernel has no libc, none of the libraries, and no memory protection, you can mess shit up
* One big namespace
* Always multi-threaded, module must be thread safe
* /proc directory
  + proc file system is the same thing
  + Mechanism for kernel and modules to send info to processes
  + Allows user level to read kernel stuff
  + Listed as 0 bytes but contains a lot of info
* Procfs
  + Special file systems
  + Open/Read/Write requests get passed to the procfs file system which knows about all these files and directories
  + You can view virtual files with cat, more, less within proc
* Changing virtual files
  + Most virtual files with /proc/ are read only
  + Some can change settings
  + /proc/cmdline: shows the parameters passed to the kernel at the time it started
  + /cpuinfo/: indentifies the type of processor used by your system
  + More stuff
* Printk()
  + Prints messages to the kernel log
  + Printf for kernel programming
  + Printk() can be called anywhere in the kernel at any time
* Dmesg
  + Display message or driver message, prints the message buffer
  + Allows the review of kernel boot process
* Loadable Kernel modules (LKMs)
  + Contains code to extend a running kernel
  + Makes it so you don’t have to recompile, can load and unload lkms
  + Written in C
  + .ko
  + MODULE\_AUTHOR(“your name”)
  + MODULE\_LICENSE(“GPL”), must be open source
* - in init\_module(void): called when you load the module
  + initialize
  + return 0
* void cleanup\_module(void)
  + called when unload
  + free resources
* Insmod installs, rmmod removes
  + Modprobe automatically manages dependent modules
* Dependent modules are automatically loaded/unloaded
* Reconfigurable device drivers
  + Allows system admins to add device without recompile
  + Stored as .ko
  + Register\_chrdev, register the device
    - Device independent function calls

**Lecture Slide 5**

* Loading a program into memory
  + OS loader, invoked by typing a program name in shell, or double click
  + Copies p1 from disk to RAM
  + When a program is loaded into RAM, it becomes an actively executing application
  + OS allocates a stack and heap to the app in addition to code and global data
* Running executable object files
  + Stack contains local variables
  + Stack dynamically expands and contracts as program runs and different levels of nested functions are called
  + Heap contains run-time variables and buffers, obtained from malloc, program should free the malloc’d memory
  + Heap expands and contracts
* A program consist of a sequence of code instructions and data stored on disk
  + Program is passive entity
* A process is a program actively executing from main memory within its own address space
  + Has a program counter and execution state
  + Owns its own address space

Code

Data

Heap

Stack

* Run-time memory image, essentially code, data, stack, and heap
* Code and data loaded from executable file
* OS provides the illusion that the process has its own subset of RAM, its own address space, on its own subset of the CPU
* Process Manager
  + Creation/deletion of processes /threads
  + Syncronization of processes / threads
  + Managing process state (PC, stack ptr, resources, memory limits)
  + Scheduling processes
  + Monitoring processes
* State of a process
  + Memory, code, data, heap, stack, state (ready/running/waiting), accounting (PID), program counter, CPU registers, CPU scheduling (priority), memory management info, I/O status info
* Each process is represented in OS by a process control block (PCB), contains complete info of a process
* OS maintains a PCB table for each process
* PCB table is typically of fixed size, this size determines max number OS can have
* Context Switch
  + Running > Ready
  + Running > Blocked
  + Switching the CPU from currently running process to another
  + Save state in PCB, load saved state of new from PCB, MINIMIZE context switch time
* Creating a process
  + In windows CreateProcess()
  + Use fork() in unix
    - Returns an int, 0 if child
    - Both parent and child execute the same code from same place
* Loading Process
  + Exec() system call loads program code into the calling process’s memory (same address space) clears the stack and executes
  + Using fork() and exec() to create new process in new address space
* Wait() system call is used by a parent process to be informed of when a child has completed IE call exit()
* Waitpid() to wait on a particular child to finish
* Accessing Process state
  + One way is through syscalls
  + Another way is through proc file system
* Using /proc
  + Can read and write status variables
  + Many system utilities like ps (process status) and top are simply calls to files in the /proc directory
* Linux allocates two stacks for each process: a user stack that resides in the user address space and a kernel stack that resides in the kernel
  + Kernel stack when process is in the kernel
  + Kernel stack is needed for security
  + 8kb for each kernel stack, PCB is stored at one end of this space, stack starts other end
* Context switch example
  + Suppose P1 and P2 alternate their execution on CPU
  + Lets assume P1 is running and a time interrupt occurs to preempt this process
    - Switch\_to()
    - Schedule()
    - Timer interrupt service routing
    - Foo() info (called from main())
    - Main() function info

**Lecture Slide 6**

* Threads
* Reduced context switch overhead vs multiple processes
  + Faster than switching contexts
  + Shared resources means less memory consumption
  + Don’t duplicate code, data, or heap or have multiple PCBs as for multiple processes
  + Scalable
* A piece of code is thread safe if it functions correctly during simulataneous or concurrent execution
* If two threads share and execute the same code, then unprotected use of shared
  + Global variables is not safe
  + Static variables is not safe
  + Heap variables is not safe
* Some tasks are not suited for threads
* No fault isolation between threads
  + Crashes can crash others
* Thread safe code is difficult to write
* User space threads are usually cooperatively multitasked, user threads within a process voluntarily give up the CPU to each other
  + OS is not aware of user space threads, only sees process
* Kernel threads
  + Supported by OS
  + Kernel sees threads and schedules

**Lecture Slide 7**

* Communicating between processes : interprocess communication (IPC)
* Two types
  + Shared Memory
    - Os provides mechanisms for creation of shared memory between processes
    - Applies to processes on same machine
    - Problem, shared access complexity
    - Shmid = shmget(key name, size, flags) creates shared memory
    - All processes sharing memory agree on key in advance, create new shared segment if no such memory with name exists
  + Message passing
    - OS has mechanisms for communication via buffers
    - Send() and receive() are basics
    - Via syscalls, slowed than shared memory
    - Can send directly to other process buffer
    - Indirect sends to OS by calling send() OS is the mailbox
    - Doesn’t need synchronization, but is slow
* IPC via pipes
  + P1 writes into one end of pipe, P2 reads from other end
* Signals
  + Used to inform processes of unexpected external events, time out forced termination
  + Allows one process to interrupt another process
  + Without signals, low-level hardware exceptions are processed by the kernel only, not visible to user process
  + 30 types in linux

**Lecture Slide 8**

* Concurrency
  + Multiple processes/threads doing stuff same time and accessing shared resource (file)
  + Speed
  + Few adapted concurrent languages
* Producer consumer problem
  + Two processes share a fixed size buffer
  + Producer puts stuff in, consumer takes stuff out
  + Shared resources pose a problem with same access
* Race condition
  + When a result is dependent on a shared resource
  + Called critical section
* Mutual exclusion
  + If a process is in the critical section no one else can be in there
* Solution 1
  + Disable interrupts before critical section
  + Enable upon leaving
* Atomic Test and Set
  + TS, all modern have this instruction set
  + This is an automic instruction enforced by hardware
  + Boolean TS(Boolean \* target)
    - Boolean rv = \*target;
    - \*target = TRUE;
    - return rv;}
* Sleep() and wakeup()
  + Sleep causes process to block
  + Wakeup causes process to ready (no effect if already not sleeping)
* Semaphore
  + Wait() and signal()
  + Wait decrements the value of S
  + Signal increments the value of S
  + Can be used for mutual exclusion but can have the problem of deadlock
* Pthreads Synchronization and mutex locks
  + Pthread\_mutex\_t m; declare
  + Pthread\_mutex\_init (&m); initialize
  + Pthread\_mutex\_lock(&m);
  + Pthread\_mutex\_unlock(&m);
  + Like semaphores, but only the thread that locks can unlock,
  + Only used for exclusion, not synchronization
* Non-preemtive kernel does not allow a process to be preempted while running in kernel mode
  + Race conditions cannot occur

**Lecture Slide 9**

* Reader Writers problem
  + Database is accessed by two types of process, reader and writer
  + Readers only read, writers modify
  + Readers can access concurrently, writers must be exclusive

Solution:

Reader()

Wait(mutex)  
 if (readers == 0) wait (database)

Readers ++

Signal (mutex)

Read the database here

Wait (mutex);

Readers - -

If (readers == 0) signal (database)

Signal(mutex)

Writer()

Wait(database)

Write database

Signal(database)

The semaphore mutex can be used to provide exclusion for editing the readers variable, but allow multiple readers into the db, writer uses exclusion on the DB.

* If a reader keeps coming then the writer starves
* Dining philosopher problem, FAMOUS
* Condition Variables
  + Provides support for sync between two or more threads
  + Used with a pthread\_mutex
  + Wait() always blocks, releases mutex before blocking, re-acquire mutext before returning
  + Signal() wakes at least one thread blocked on variable
  + Broadcast() wakes up all threads blocked on variable
* Monitors
  + Abstract data type
  + Collection of functions, variables, data structs
  + Process only has access to monitor variables when calling functions in the monitor
  + Only one process at a time

**Lecture Slide 10**

* Scheduling
* Metrics
  + Execution time E(Pi) = the time on the CPU required to fully execute process i
  + Wait time W(Pi) = sum of the times process I spends in the ready state
    - Calculate by counting how much time the process spent not doing anything
  + Turnaround time T(Pi) = the time from 1st entry of process I into the system to its final exit from the system (exits last run state)
    - Calculate by finding time where the process ended
  + Response time R(Pi) = the time from 1st entry of process I into the ready queue to its 1st scheduling on the CPU (1st run state)
  + CPU utilization: Percentage of time the CPU is busy
  + Throughput: number of processes completed per time unit
* Preemptive vs non-preemtive scheduling
  + Preemptive: Running process may be forced to give up CPU, relies on timer interrupts, can result in race conditions
  + Non-Preemptive: Running process keeps CPU until terminating or switching to wait, long CPU processes can prevent other from getting CPU
* FCFS
  + First come first serve
  + Non preemptive
  + If a process arrived in order P1, P2, P3 before time 0, then it goes in order
  + Lot of variation in wait and turnaround time
* SJF
  + Shortest job first
  + Choose process with the lowest execution time, priority is given to short stuff
  + Minimizes the average wait time
  + Must know the times in advance
* Deadline scheduling
  + Schedule all according to deadline times
  + Cannot miss a deadline
  + Sometimes impossible to meet a deadline
* Round Robin
  + Rotate among processes and work on them for a given time slice
  + Simple and fair, wait times can be really long
  + If something finishes before time slice is up just go to next process
  + Can do weighted round robin where you assign time slices based on a weight

**Lecture Slide 11**

* Priority Scheduling
  + Assign each task a priority and schedule higher priority tasks first
* Multi-level queue scheduling
  + Use priorities to partition the ready queue into several separate queues
  + Queues can be organized by priority, or each given a percentage of CPU, or a hybrid combo
* Preemptive priorities can starve low priority processes
  + Higher priority process always gets served ahead of a lower priority, which never actually sees CPU time
* Multi-level feedback queues solve this problem
  + Number of queues
  + Scheduling algorithms for each queue
  + Method used to determine when to upgrade a process
  + Method used to determine when to demote a process
  + Method used to determine which queue a process will enter when that process needs service
* Criteria for process movement
  + Age of a process: old processes move to higher priority queues, or high priority are demoted
  + Behavior of a process (I/O vs CPU bound), higher priority to I/O, allows parallelism between CPU and I/O
* O(N) scheduler
  + CPU time is divided into epochs
    - Each epoch every process can execute up to its time slice, if a task does not use all of its slice, scheduler adds half of the remaining time to allow it to execute longer in next epoch
  + If an interactive process yields before time slice is over, priority is higher next time
  + Keep list of this goodness
  + Scheduler chooses task with highest priority from active runqueue
  + When all tasks exhast time slice, two runqueues are exchanged, expired runque becomes active runqueue
  + When a task is move from active to expired, calculate new priorit
* CFS
  + Complete fair scheduler
  + Doesn’t use priorites but instead uses them as a decay factor for time a task is permitted
  + Uses red-black tree
  + Lower runtime are on left of tree
  + Scheduler picks the left most node to schedule ne

When the result depends exactly on when the program runs with multiple threads it is called race conditions

2