CSCI 4730 Review

Tyler Bugbee

December 6, 2017

4-5 multiple choice questions, 2-3 long answer questions

1 Overview

OS Structure (Ch. 2)

- OS Interface, System calls
- Multi-tasking OS - Foreground and background processes
- Microkernel and loadable Kernel modules

Processes and Thread (Ch. 3 and 4)

- Program vs. process
- Memory structure of the process (stack, heap, data, text)
- Context switch and Process Control Block (PCB)
- Linux Process Management (fork, exec) and interprocess communication (IPC)
- Memory structure of the multi-threaded process
- Multi-process vs. multi-threaded system
- Thread local storage (TLS)

Synchronization and Deadlock (Ch. 5 and 7)

- Race condition, mutual exclusion and liveness
- Mutex Lock, Semaphore
- Atomic instructions: Test-and-set, compare-and-swap
- Producer-consumer, readers-writers and dining-philosophers problems
- Deadlock conditions (mutual exclusion, hold and wait, no preemption and circular wait)
- Deadlock avoidance: resource-allocation graph, Banker's algorithm
- Deadlock detection algorithm: variation of Banker's algorithm

2 Chapter 2

Operating System Structures

2.1 Graphical User Interface

User-friendly desktop metaphor interface

- Usually mouse, keyboard, and monitor
- Icons represent files, programs, etc
- Directories are folders, hardware inputs provide info

Many systems now provide CLI and GUI interfaces

- Windows is a GUI with CLI "command" shell
- OSX is "Aqua" GUI interface with UNIX kernel underneath and available shells
- Unix and Linux have CLI with optional GUI interfaces

2.2 Command Line Interface

CLI or command interpreter allows direct command entry

- Sometimes implemented in kernel, sometimes by systems program
- Multiple flavors or shells
- Sometimes commands built-in, sometimes just names of programs (new features don't require shell modification)

2.3 System Calls

Programming interface to the services provided by the OS, typically written in a high-level language (C or C++)

2.4 System Call Implementation

Typically, a number is associated with each system call

- System-call interface maintains a table indexed according to these numbers

The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
The caller doesn't need to know anything about how the system call is implemented, just obey API and understand what OS will do as a result call

2.5 System Call Parameter Passing

Often more information is required than the identity of the system call
3 general methods used to pass parameters to the OS

- Simplest: pass the parameters in registers
- Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
- Parameters pushed onto the stack by the program and popped off the stack by the OS
- Block and stack methods do not limit the number or length of parameters being passed

2.6 Types of System Calls

Process control

- create process, terminate process
- end, abort
- load, execute
- get process attributes, set process attributes
- wait for time
- wait event, signal event
- allocate and free memory
- dump memory if error
- debugger for determining bugs, single step execution
- Locks for managing access to shared data between process

File management

- create file, delete file
- open, close file
- read, write, reposition
- get and set file attributes

Device management

- request device, release device
- read, write, reposition
- get device attributes, set device attributes
- logically attach and detach devices

Info maintenance

- get time or date, set time or date
- get system data, set system data
- get and set process, file, or device attributes

Communications

- create, delete communication connection
- message passing model send, receive messages to host name or process name
- shared-memory model create and gain access to memory regions
- transfer status information
- attach and detach remote devices

Protection

- control access to resources
- get and set permissions
- Allow and deny user access

2.7 System Calls and Standard Library

Mostly accessed by programs via a high-level API rather than direct system call use
Three most common APIs are Win32 API, POSIX API (Linux, Unix, OSX), and Java API

System Programs (Utilities)

System programs provide a convenient environment for program development/execution. They can be divided into:

- File manipulation
- Status info sometimes stored in a file modification
- Programming language support
- Program loading and execution
- Communications
- Background services
- Application programs

System Programs Continued

Some are simply UIs to system calls, others are more complex
File Management - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
Status Information

- Some ask the system for info - date, time, amount of available memory, disk space, number of users
- Others provide detailed performance, logging, and debugging info
- Typically, these programs format and print the output to the terminal or other output devices
- Some systems implement a registry - used to store and retrieve config info

File modification

- Text editors to create and modify files
- Special commands to search contents of files or perform transformations of the text

Programming language support - compilers, assemblers, debuggers, and interpreters sometimes provided
Program loading and execution - Absolute loaders, relocatable loaders, linkage editors, and overlay loaders, debugging systems for higher-level and machine language
Communications - Provide the mechanism for creating virtual connections among processes, users, and computer systems

- Allow users to send messages to one another's screens, browse web pages, send email, log in remotely, ftp

Background Services

- Launch at boot time
- Provide facilities like disk checking, process scheduling, error logging, printing

- Run in user context not kernel context
- Known as services, subsystems, daemons

Application programs

- Run by users
- Not typically considered part of OS
- Launched by command line, mouse click, finger poke

Shell

A shell is a job control system

- Allows programmers to create and manage a set of programs to do some task

2.8 Managing Processes

Foreground processes are interactive

- Processes require you to wait until they are done before using the prompt again

Background processes are non-interactive

- Processes run in the background, while you do other things; user generally cannot send information to the process directly

2.9 Background Processing

By using an & at the end of a command, the commands in the line will be executed in the background, returning the user immediately to the prompt
All output to stdout and stderr will be displayed normally, possibly interleaved with output from other processes
Any requests from the background from stdin will be blocked from reading and placed in a stopped state
It's important to structure commands to be run in the background to avoid these limitations and ensure the proper execution of unattended commands

Operating System Design and Implementation

Internal structure of different OS can vary widely
Affected by choice of hardware, type of system
User goals and System goals

- User goals - operating system should be convenient to use and learn, reliable, safe, fast
- System goals - OS should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Implementation

Much variation

- Early OS in assembly
- Later, system programming languages like Algol, PL/1
- Now C, C++

Really a mix of languages

- Lowest levels in assembly
- Main body in C
- Systems programs in C, C++, scripting languages like PERL, Python, shell scripts

More high-level language easier to port to other hardware
Emulation can allow an OS to run on non-native hardware

Operating System Structure

General-purpose OS is very large program
Various ways to structure

- Simple structure - MS-DOS
- More complex - UNIX
- Layered - an abstraction
- Microkernel - Mach

2.10 Microkernel System Structure

Moves as much from the kernel into user space
Mach example of microkernel

- Mac OSX kernel (Darwin) partly based on Mach

Communication takes place between user modules using message passing
Benefits:

- Easier to extend a microkernel
- Easier to port the OS to new architectures
- More reliable (less code is running in kernel mode)
- More secure

Detriments:

- Performance overhead of user space to kernel space communication

2.11 Modules

Many modern OS implement loadable kernel modules

- Uses object-oriented approach
- Each core component is separate
- Each talks to the other over known interfaces
- Each is loadable as needed within the kernel

Overall, similar to layers but more flexible

- Examples are iOS, Android

3 Chapter 3

Processes: What is the difference between a "program" and a "process"?

- Programs are passive entities stored on a disk (executable file)
- Processes are active entities: program + execution state
- Programs become processes when the executable file is loaded into memory

One program can be several processes

Intro

On modern operating systems, processes provide 2 virtualization: a virtualized processor and virtual memory

- The virtual processor gives the process the illusion that it alone monopolizes the system
- Virtual memory lets the process allocate and manage memory as if it alone owned all the memory in the system
- 2 or more processes can exist that are executing the same program
- In fact, 2 or more processes can exist that share various resources, such as open files or an address space

3.1 Process Concept

The program code, also called text section
Stack containing temp data

- Function parameters, return addresses, local variables

Data section containing global variables
Heap containing memory dynamically allocated during run time

3.2 Instruction Processing

Fetch instruction from memory → decode instruction → Evaluate address → Fetch operands from memory → Execute operation → Show result → (repeat)

Instruction

The instruction is the fundamental unit of work
It specifies 2 things:

- Opcode (operation to be performed)
- Operands (data/locations) to be used for operation

An instruction is encoded as a sequence of bits

- Instruction Set Architecture (ISA)

Process State

As a process executes, it changes state

- new: the process is being created
- running: instructions are being executed
- waiting: the process is waiting for some event to occur
- ready: the process is waiting to be assigned to a processor
- terminated: the process has finished execution

There is at most one running process per CPU or core

3.3 Process Control Block (PCB)

Information associated with each process (also called task control block)

- Process state - running, waiting, etc
- Program counter - location of instruction to next execute
- CPU registers - contents of all process-centric registers
- CPU scheduling information - priorities, scheduling queue pointers
- Memory-management info - memory allocated to the process
- Accounting information - CPU used, clock time elapsed since start, time limits
- I/O status info - I/O devices allocated to process, list of open files

Process Information

For each process, there is a corresponding directory "/proc/¡pid¿" to store the process information
"ps" and "top" commands show the process information

Process Scheduling

When more than one process is runnable, the OS must decide which one first
Maximize CPU use, quickly switch processes onto CPU for time sharing
Process Scheduler selects among available processes for next execution on CPU
Maintains scheduling queues of processes

- Job queue - set of all processes in the system
- Ready queue - set of all processes residing in the main memory, ready and waiting to execute
- Device queues - set of processes waiting for an I/O device
- Processes migrate among the various queues

Schedulers

Short-term scheduler (or CPU scheduler) - selects which process should be executed next and allocates CPU

- Sometimes the only scheduler in a system
- Short-term scheduler is invoked frequently (ms) $\rightarrow$ must be fast

Long-term scheduler (or job scheduler) - selects which processes should be brought into the ready queue

- Long-term scheduler is invoked infrequently (seconds, minutes) $\rightarrow$ may be slow
- The long-term scheduler controls the degree of multiprogramming

Processes can be described as either:

- I/O-bound process - spends more time doing I/O than computations, many short CPU bursts
- CPU-bound process - spends more time doing computations; very few long CPU bursts

Long-term scheduler strives for good process mix
Medium-term scheduler can be added if degree of multiple programming needs to decrease
Remove process from memory, store on disk, bring back in from disk to continue execution: swaping

Multitasking in Mobile Systems

Some mobile systems allow only one process to run, others suspended (ex: early iOS)
Due to screen real estate, user interface limits iOS provides for a

- Single foreground process - controlled via UI
- Multiple background processes - in memory, running, but not on the display, and with limits
- Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback

Android runs foreground and background, with fewer limits

- Background process uses a service to perform tasks
- Service can keep running even if background process is suspended
- Service has no UI, small memory use

3.4 Context Switch

When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
Context of a process represented in the PCB
Context-switch time is overhead; the system does no useful work while switching

- The more complex the OS and the PCB → the longer the context switch

Time dependent on hardware support

- Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

3.5 Linux Process Management

Process creation

- clone() - system call to create a new process
- fork() - library call to create a copy of the current process, and start it running

Process execution

- exec() - system call to change the program being run by the current process

Process control

- wait() - system call to wait for a child process to finish
- signal() system call to send a notification to another process

Process Creation

Parent process creates children processes, which, in turn create other processes, forming a tree of processes
Generally, processes are identified and managed via a process identifier(pid)
Resource sharing options

- Parent and children share all resources
- Children share subset of parent's resources
- Parent and child share no resources

3.6 Interprocess Communication

Cooperating processes need interprocess communication (IPC)
2 models of IPC: Shared memory and Message passing

3.7 IPC - Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the OS
- Major issues are to provide a mechanism that will allow the user processes to synchronize their actions when they access shared memory

3.8 IPC - Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system - processes communicate with each other without resorting to shared variables
- IPC facility provides 2 operations:
  - send(message)
  - receive(message)
- The message size is either fixed or variable
- If processes P and Q wish to communicate, they need to:
  - Establish a communication link between them
  - Exchange messages via send/receive

Synchronization

Message passing may be either blocking or non-blocking
Blocking is considered synchronous

- Blocking send - the sender is blocked until the message is received
- Blocking receive - the receiver is blocked until a message is available

Non-blocking is considered asynchronous

- Non-blocking send - the sender sends the message and continues
- Non-blocking receive - the receiver receives a valid message or a null message

Different combinations possible

- If both send and receive are blocking, we have a rendezvous

4 Chapter 4

Process Overheads

A full process includes numerous things:

- an address space (defining all the code and data pages)
- OS resources and accounting info
- process control block (PCB)
  - Where the process is currently executing
  - The status of the process

Creating a new process is costly

- all of the structures (e.g., page tables) that must be allocated

Context switching is costly

- implicit and explicit costs

IPC is costly

4.1 Threads and Processes

Most OS support 2 entities:

- the process, which defines the address space and general process attributes
- the thread, which defines a sequential execution stream within a process

A thread is bound to a single process

- For each process, there may be many threads

Threads are the unit of scheduling
Processes are containers in which threads execute

4.2 Threads vs. Processes

- A thread has no data segment or heap
- A thread cannot live on its own, it must live within a process
- Inexpensive creation
- Inexpensive context switching
- If a thread dies, its stack is reclaimed

- A process has code/data/heap & other segments
- There must be at least one thread in a process
- Expensive creation
- Expensive context switching
- If a process dies, its resources are reclaimed and all threads die

4.3 Multithreaded Applications

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - update display
  - fetch data
  - spell checking
  - answer a network request
- Thread creation is lightweight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

4.3.1 Benefits

- Responsiveness - may allow for continued execution if part of process is blocked
- Resource Sharing - threads share resources of processes, easier than shared memory or message passing
- Economy - cheaper than process creation, thread switching lower overhead than context switching
- Scalability - process can take advantage of multiprocessor architectures

Signal Handling

Signals are used in UNIX systems to notify a process that a particular event has occurred
A signal handler is used to process signals

1. Signal is generated by particular event
2. Signal is delivered to a process
3. Signal is handled by one of two signal handlers:
    - default
    - user-defined

Every signal has a default handler that kernel runs when handling a signal

- User-defined signal handler can override default
- For single-threaded, signal delivered to process

When should a signal be delivered for multi-threaded?

- Deliver the signal to the thread to which the signal applies
- Deliver the signal to every thread in the process
- Deliver the signal to certain threads in the process
- Assign a specific thread to receive all signals for the process

4.4 Thread-Local Storage

TLS allows each thread to have its own copy of data
Different from local variables

- Local variables visible only during single function invocation
- TLS visible across function invocations

Similar to static data

- TLS is unique to each thread

Linux Threads

Linux refers to them as tasks rather than threads
Thread creation is done through clone() system call
`clone()` allows a child task to share the address space of the parent (process)

- Flags control behavior

`struct task_struct` points to process data structures (shared or unique)

5 Chapter 5

5.1 Synchronization Motivation

When threads concurrently read/write shared memory, program behavior is undefined

- 2 threads write to the same variable; which one should win?

Thread schedule is non-deterministic
Compiler/hardware instruction reordering
Multi-word operations are not atomic

5.2 Definitions

Race condition: output of a concurrent program depends on the order of operations between threads
Mutual exclusion: only one thread does a particular thing at a time

- Critical section: piece of code that only one thread can execute at once

Liveness(program): If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

5.3 Mutex Locks

- OS designers build software tools to solve critical section problem
- The simplest is mutex lock
- Protect a critical section by first `acquire()` a lock then `release()` the lock
  - Boolean variable indicating if lock is available or not
- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented by hardware atomic instructions
- But this solution requires busy waiting
  - This lock therefore called a spinlock

1. At most one lock holder at a time (safety)
2. If no one holding, acquire gets lock (progress)
3. If all lock holders finish and no higher priority waiters, waiter eventually gets lock (progress)

5.4 Rules for Using Locks

Lock is initially free
Always acquire before accessing shared data structure

- beginning of procedure

Always release after finishing with shared data

- end of procedure
- only the lock holder can release
- DO NOT throw lock for someone else to release

Never access shared data without lock - dangerous

5.5 Locks

Deadlock - two or more process are waiting indefinitely for an event that can be cause by only one of the waiting processes

Synchronization Hardware

Many systems provide hardware support for implementing the critical section code
Uniprocessors

- Disable interrupts
- Currently running code would execute without preemption
- Generally too inefficient on multiprocessor systems

Modern machines provide special atomic hardware instructions

- Atomic = non-interruptible
- Either test memory word and set value
- Or swap contents of two memory words

5.6 Hardware Mutex Support

Test and Set

- Read boolean, set it true, and set condition codes
- All in one indivisible operation

Compare and Swap

- Read word, compare to register, store other register into word
- Again, indivisible
- Generalization of Test & Set

Test and Set Instruction

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to "TRUE"

- Very fast entry to unlocked region
- Easy to implement
- Guarantees safety & progress
- Wastes CPU when waiting (spin lock/busy wait)
- Doesn't make it easy for other threads to run
- Prone to errors

- Prone to starvation

Compare and Swap Instruction

1. Executed atomically
2. Returns the original value of passed parameter "value"
3. Set the variable "value" the value of the passed parameter "new_value" but only if "value" == "expected". That is, the swap takes place only under this condition

5.7 Semaphores

Synchronization tool that provides more sophisticated ways (than mutex locks) for process to synchronize their activities

- P(sem) or wait(sem) decrements and possibly waits
- V(sem) or signal(sem) increments and lets let somebody else in

Usually implemented by operating system

- Allows scheduler to run different thread while waiting
- OS can guarantee fairness and no starvation
- More flexibility for user

Still error-prone

- P and V must be matched
- Single extra V blows mutual exclusion entirely (compare Test & set)

5.8 Semaphore Usage

- Counting semaphore integer values can range over an unrestricted domain
- Binary semaphore - integer value can range only between 0 and 1, same as mutex lock
- Can solve various synchronization problems
- Can implement a counting semaphore S as a binary semaphore.

5.9 Producer-Consumer Problem

Competition between the two processes to access the shared buffer
Cooperation of the two processes in order to exchange data through the buffer

5.10 Review: Semaphores

Definition: a Semaphore has a non-negative integer value and supports the following 2 operations:

- `Wait()`: an atomic operation that waits for semaphore to become positive, then decrements it by 1
- `Signal()`: an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
- Only time can set integer directly is at initialization time

5.11 Producer-Consumer Problem

n buffers, each can hold one item
Semaphore `mutex` initialized to the value 1
Semaphore `full` initialized tot he value 0, number of full buffers
Semaphore `empty` initialized to the value n, number of empty buffers

5.12 Readers-Writers Problem

A data set is shared among a number of concurrent processes

- Readers - only read the data set; they do not perform any updates
- Writers - can both read and write

Problem - allow multiple readers to read at the same time

- Only one single writer can access the shared data at the same time

5.13 Dining-Philosophers Problem

Philosophers spend their lives alternating thinking and eating
Don't interact with their neighbors, occasionally try to pick up 2 chopsticks to eat from a bowl

- Need both to eat, then release both when done

In the case of 5 philosophers:

- Shared data: bowl of rice (data set) and semaphore (chopstick[5]) initialized to 1

Deadlock handling:

- Allow at most 4 philosophers to be sitting simultaneously at the table
- Allow a philosopher to pick up the forks inly if both are available (picking done in critical section)
- Use an asymmetric solution - an odd-numbered philosopher picks up the left chopstick first and then the right chopstick. Even-numbered philosopher picks up the first right chopstick and then the left chopstick

5.14 Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously:

1. Mutual exclusion
   - At least one resource must be held in non-shareable mode
2. Hold and wait
   - A process holding at least one resource is waiting to acquire additional resources held by other processes
3. No preemption
   - Only the process can release its held resource
4. Circular wait
   - $\{P_0, P_1, \ldots, P_n\}$
   - P_i is waiting for the resource held by P_{i+1}
   - P_n is waiting for the resource held by P_0

5.15 Resource Allocation Graph

A set of processes $\{P_0, P_1, \ldots, P_n\}$
A set of resource types $\{R_1, R_2, \ldots, R_n\}$, together with instances of those types
If the graph has no cycles, then there are no deadlock processes
If there is a cycle, then there may be a deadlock

5.16 Dealing with Deadlocks

Deadlock prevention

- disallow at least one of the necessary conditions
  - Mutual exclusion - shared resources can be read-only mode
  - Hold and wait - each process holds all of its resources before it begins executing, or only allow a process to request resources when it currently has none
  - No preemption - make a process automatically release its current resources if it cannot obtain all the ones it wants, restart the process when it can obtain everything
  - Circular wait - Impose a total ordering on all the resource types and force each process to request resources in increasing order

Deadlock avoidance

- see a deadlock coming and alter the process/resource allocation strategy

Deadlock detection and recovery
Ignore the problem

- most OS, including UNIX
- cheap solution
- infrequent, manual reboots may be acceptable

5.17 Avoidance Algorithms

Single instance of a resource type, use a resource-allocation graph
Multiple instances of a resource type, use the banker's algorithm

5.18 Banker's Algorithm

Assume that:

- a resource can have multiple instances
- the OS has N processes, M resource types

Initially, each process must declare the maximum number of resources it will need
Calculate a safe sequence if possible

5.19 Deadlock Detection

If there are no prevention or avoidance mechanisms in place, then deadlock may occur
Deadlock detection should return enough info so the OS can recover

5.20 Detection Algorithm

1. Vector Copy: Work ≔ Available; Finish ≔ false
2. Find i such that P_i hasn't finished but could:
   Finish[i] == false
   Request[i] ¡= Work
   If no suitable i, go to step 4
3. Assume P_i completes:
   Work ≔ Work + Allocation[i]
   Finish[i] ≔ true
   Go to step 2
4. If Finish[i] == false then P_i is deadlocked