CS 4348 — Operating Systems Concepts, Final Review

https://iniyanjoseph.com | iniyanijoseph@gmail.com

Process Creation

1. Processes are arranged in a tree, with distinct address spaces
2. The process image starts as a replica of another process
    1. fork() system call makes a new "child" process
        1. 0 to the child process
            - This makes sense because the child process can always get getpid() to get the ID
        2. PID to the parent process, since there is no other way to get the child process
        3. -1 on Failure
        4. They begin from the next instruction after the fork
3. exec() system call replaces the current process image with a new process image and begins running it from the beginning

Process Synchronization

1. Sharing data between processes is very useful
2. But its difficult because of read/write
3. Processes cannot read/write concurrently
    1. What if the data is only half-written to a file, then a context switch occurs? This means that the data written to the file is not useful.
    2. What if two processes want to write data at the same time? Even if both finish writing, the data will be garbled
4. So we create some way of "mutual exclusion"
    1. Note that we could create mutual exclusion by stopping multiprogramming (at least partially), but this is of course not ideal
5. Concurrent programs have problems
    1. Race conditions
    2. Deadlocks
    3. Livelocks
6. Mutual Exclusion Problem
    1. Locks
        1. The process can "acquire" the lock
            1. The context switch cannot occur while the lock is acquired
        2. The process can "release" the lock
            1. The process switch can occur while it is released
        3. The code between the `acquire` and `release` is now seen as being atomic, and so is called a "critical section"
        4. Easy to use
        5. Deadlocks may occur
        6.
    2. Semaphore
        1. Integer/counter value which is arbitrarily initialized
        2. It wan be decreased (wait) or increased (signal)
        3. wait()
            1. Decrease integer if its value is > 0
            2. Wait if value of semaphore is $= 0$
        4. signal()
            1. integer value $+= 1$
        5. If a process is waiting for the value of a semaphore to become > 0, then increase will allow some other process's wait() call to occur and continue in the code

6. wait and signal must be atomic, for example if wait() is called 2 times and context switches between, allowing the integer value to decrease below 0 then future signal() operations may not work
7. We can also think of a Lock as being a binary semaphore

3. Monitor
    1. Semaphores are unreliable
        1. They work, but only when used correctly
    2. Monitors are simply a generalization of OOP which encapsulates data and operations within a class
    3. Objects of the class can be allocated and the methods can be invoked on the fields of the object
    4. Only one process can execute an operation on an object at any one time
        1. All the fields are private
    5. Encapsulated semaphore

Deadlocks

1. One of the problems with process synchronization is that of deadlocks
2. A process has to wait for another to release a resource
3. If a wait is circular, then no process is able to execute
    1. This is a deadlock
4. A process requesting a process in a the set of processes involved in the circular wait is also blocked forever
5. Chained waiting
6. A resource must have mutual exclusion (i.e. only so many copies of the resource exist)
7. Every process in the circular wait must be holding a resource and waiting for another
8. No preemption
9. Circular wait
10. If many copies of the resource exist, you can use the bankers algorithm
11. Deadlock prevention
    1. Make resources shareable
        1. Not all types of resources can be shared
    2. Hold and wait
        1. Each process can request all of its resources to perform its task at once
        2. This is wasteful
    3. Preemption
        1. You can interrupt processes to take away their resources
        2. This may require reverting a victim process, which is not always possible
    4. Acquire resources in increasing order of rank
        1. Give all resources a rank
        2. Assign resources to processes in order of rank
        3. Requires knowing all available resources beforehand
12. Deadlock avoidance
    1. Claim edges
        1. A process may or may not use a process in the future
        2. Do not grant a request if it will create a cycle
    2. Bankers algorithm
        1. Allocate resources such that the state is always safe
        2. Safe means that resources can be fully allocated to a process and a deadlock will not be created
13. Deadlock detection and management
    1. Detect by running bankers algorithm
    2. If found
        1. Terminate a process
        2. Preempt a resource

Mass Storage

1. Disk, Tape, SSD
2. We focus on disk
    1. How do we schedule access on a disk (i.e. given a set of addresses, which order should we look up adresses?)
        1. First Come First Serve
            1. Naive and easy to implement
            2. You have to cover alot of ground multiple times
        2. Shortest Seek Time First
            1. Greedily go to shortest seek time
            2. Not amazing in practice
        3. SCAN
            1. Bounce back and forth from beginning to end
        4. Circular SCAN
            1. Keep looping around the
        5. LOOK
            1. Bounce back and forth until you get from first request to last request
        6. Circular LOOK
            1. Bounce back and forth until you get from first request to last request
3. RAID
    1. Redundant storage on inexpensive disks
    2.