CSCC69 Lecture Notes

Joshua Concon

Summer 2018

Textbook: Andrew Tannenbaum: Modern Operating Systems. Prentice Hall (2001 or 2007).

Contents

1 Tuesday, May 8, 2018

1.1 What is an operating system?

To turn "ugly" hardware into "beautiful" abstractions, which essentially means that it makes it easier to use and code on a computer, rather than just using machine code. An operating system also servers as a resource manager and a control program, in the sense that it allocates resources and prevents errors and improper use of the computer.

User Mode This mode gets requests from applications and the UI and hands it off to the Kernel

Kernel Processes requests, manages order of requests and implements the requests.
We can imagine the applications running on an OS that connects the running software to the hardware.

1.2 Storage Hierarchy

The CPU has cache, which is the smallest and fastest memory in the computer due to it's proximity to the CPU. From there, memory gets larger, slower, cheaper, and further away from the CPU (Think, L2, then L2/L3, then RAM, then HDD,...).

1.3 Storage Structure

Main memory (DRAM) stores programs and data during program execution. DRAM cannot store these permanently because it is too small and it is a volatile storage device. Forms a large array of bytes of memory, each with its own addresses. We say that main memory is byte-addressable.

1.4 Caching

When the processor accesses info at some level of the storage hierarchy, that info may be copied to a cache memory closer to the processor, on a temporary basis. Since caches have limited sizes, cache management is an important design problem.

1.5 Concurrency

Every modern computer is a multiprocessor, so the CPU and device controllers can execute different things, concurrently. A memory controller synchronizes access to a shared memory. **Interrupts** allow device controllers to signal the CPU that some event has occurred, they are also used to signal errors or requests

for OS service from a user program (a **system call**). These types of interrupts are called **traps** or **exceptions**.
Note that an operating system is an event-driven program

1.6 C Programming and Memory

A variable in a C program is a symbolic name for a data item, stored in memory. The type of the variable indicates how much **storage** (how many bytes) it needs. Type also determines **alignment** requirements. The **address** of the variable is an index into the big array of memory words where the data item is stored. The **value** of the variable is the actual contents of memory at that location. A **pointer** type variable is just a data item whose contents are a memory location (usually the address of another var).
The **char** data type is 1 byte in size, the **int** data type is 1 word in size (32 bits for most current architectures), and occupies 4 bytes, should be word-aligned. Pointer types are also 1 word in size.
We'll need to know pointers and the size of types for this course.

1.7 The Process Concept

A **process** can be thought of a program in execution, an instance of a running program. It contains all data of the program, and has a **PCB (Process Control Block)** and is named by it's **process ID (PID)**. The PCB contains all of the data of the process, a **Text** of all the code of the program, a **Stack** that keeps track of variables and function calls, and a **Heap** that stores all dynamic memory allocation.
If the stack gets too big and uses more that the space it has, that is called a **stack overflow**

1.8 Process Data Structures

At any time, there are many processes in the system, each in its own PCB. The PCB is also where the OS keeps all of a process' hardware execution state when the process is not running. This state is everything that is needed to restore the hardware to the same configuration it was in when the process was switched out of the hardware.

1.9 Process Control Block

It also includes

Process state (ready, running, blocked,...)

Program counter which is the address of the next instruction

CPU Registers their state must be saved at an interrupt

CPU scheduling information Process priority

Memory Management Info Page tables

I/O status information list of open files

1.10 Process state and state changes

The OS manages processes by keeping tracking of their state. Different events cause changes to a process state, which the OS must record/implement.
We need this because one process can't use the CPU forever, we need to switch processes (called Context Switching). So we time-out the process and we save the memory to the PCB for that process when it's time to switch.
Blocked processes can't go to the running state right away, it must wait in the ready state.

1.11 State Queues

This is the ready queue.
The OS maintains a collection of queues that represent the state of all processes in the system. Typically, the OS has one queue for each state, each PCB is queued on a state queue according to its current state. As a process changes state, its PCB is unlinked from one queue and linked to another.

1.12 PCBs and State Queues

PCBs are data structures dynamically allocated in OS memory. When a process is created, the OS allocates a PCB for it, initializes, and places it on the Ready queue. As the process computes, its PCB moves from one queue to another. When the process terminates, its PCB is de-allocated.

1.13 Context Switch

Context Switch switch the CPU to another process, saving the state of the old process and loading the saved state for the new process.
Context switch time is pure overhead, so some systems offer specific hardware support. It is also an expensive operation, so many context switches will take a long time, and it is also a performance bottleneck, so new structures are being used to avoid it.

1.14 Operations on Processes

Processes execute concurrently and must be created and deleted dynamically.

1.14.1 Process Creation

Who can create processes? Users, system initialization, a running process, and batch jobs can create processes.

1.14.2 Process Termination

Who can kill processes? when a processes finishes executing, when a parent terminates a child, a user can kill a process, and when an error occurs.

1.15 Process Creation

When a process is created by another process, parent is the creator, child is the created. We can use the PPID to differentiate the processes.
In some systems, resources and privileges are inherited from parents to their children. We do this to do concurrency, different processes working on different things.
Parents can either do jobs concurrently with their children, or waits for them.

1.15.1 Unix

In Unix, processes are created using fork. fork creates a new address space for the process, and copies the PCB of the parent, and duplicates it with a different PPID. Fork returns twice, returns the child's PID to the parent, 0 to the child, so we can use and if statement to make these two processes do different things. After this, either the parent or the child could go first, there is no default priority.
Note: a fork duplicating the address space of the original process, so forks are a potentially expensive process.

1.15.2 Why fork?

Very useful when the child is cooperating with the parent or when the child relies on the parent's data to accomplish its task. Also easier to do multiple requests at the same time. For example, this would be useful for a websever in handling multiple client requests concurrently.

1.15.3 Unix Shells

A shell is a process that gets new commands and executes them. To implement one, receive a command, fork, and the parent waits for the child while the child executes the command.

1.16 Example: Concurrent Web Server

If the Text and Data of a PCB are the same, a fork may not be worth the duplication. How can we improve this?
We can improve this by having multiple program counters, split stack resources.

1.17 Parallel Programming

Why don't we put them together and make threads? Have threads in a process. These are all on the same PCB and have separate stacks, so it's still a process, but multiple program counters.
Note that every process has at least one thread.
Now we need synchronization, since they all use the same Heap. This means that cost of creating threads is large as well, just like having multiple processes. There is always a tradeoff.

1.18 Example: Concurrent Servers

Now with threads, you do a thread_fork by passing in a request function and all the other arguments for it.

1.19 Rethinking Processes

What is similar in these cooperating processes? They all share the same code and data (address space), privileges, and resources (files, sockets, etc.)
What don't they share? Each of them has their own execution state.
Key Idea: Why don't we separate the concept of a process from it execution state?
Execution state is also called a **thread of control** or **thread**

1.20 Threads

The thread defines a sequential execution stream within a process, while a process defines the address space and general process attributes (everything but thread of execution)
Processes are the containers in which threads execute. Processes become static, and threads are the dynamic entities.
Consider our example with a forking web server, using fork to create new processes to handle requests is overkill for such a simple task, using threads would be more efficient.

1.21 Thread Interface

Pthread API will create a new thread for you, can specify priority as well.

1.22 Thread Scheduling

The thread scheduler determines when a thread runs, also uses a queue to keep track of what threads are doing. It's just like the OS and processes, but it's implemented at user-level in a library.
How do you implement wait queues? How to wake up a thread?

1.23 Threads Summary

The operating system is a large multithreaded program, with each process executing as a thread within the OS.
Multithreading is also very useful for applications. Efficient multithreading requires fast primitives.

1.24 Cooperating Processes

Threads are in the same process, so they can communicate through the shared memory (Heap), but processes don't have that space predefined, so we need more support for them. We can use message passing or shared memory. In general, we have independent processes that don't depend on other processes, and dependent processes that are the opposite.
A process is **independent** if it cannot affect or be affected by the other processes executing in the system.
No data sharing implies that a process is independent. A process is **cooperating** if it is not independent.

1.25 Interprocess Communication

Cooperating processes need to exchange information, using either shared memory or message passing.

2 Tuesday, May 15, 2018

2.1 Bootstrapping

Bootstrapping is the process of bringing up the operating system when you turn on the machine.
The first program that is loaded is the operating system when you turn on your computer, so when the system is off, the operating system is just a program or code that needs to be run.

BIOS This is the Basic Input Output System, it's job is to just initialize the system, so making sure the hardware is connected and make sure their functioning correctly, scans buses to detect attached decides, determines the boot devices (can access BIOS to change order in which it checks devices, note that people needed to install the operating system separately and had to configure with the BIOS more). This program then reads the first sector of the boot deivce and executes it (this is the bootloader program), then the bootloader reads a partition table, finds the active partition, reads the second boatloader, and then the second bootloader reads the OS into memory and executes it.

2.2 Operating System Startup

The Machine starts in system mode, so kernel code can be executed kernel code can be execute immediately.
Then it creates its internal datastructures, creates the first process, switch mode to allow user input and starts running the first process. Then waits for user input, as an OS is entirely driven by external events.

2.3 Memory Layout

Note that the OS has its own PCB. This is because the OS itself is a process.

2.4 From Program to Process

Consider a program that reboots your computer. We can compile it into an object file and then to create an executable. But that does not mean it can be executed, as it might not have permissions to run.
Note all these files are on the file disk.

2.5 Unix Shells

Lets revisit this, a process that waits for commands, and when it recieves one, forks and the child executes the command and the parents continues to wait.
To do this, we use

```
int exec(char *prog, char *argv[])
```

When exec is executed, exec() stops the current process, loads the program ”prog” into the process’ address space, initializes hardware context and arguments for the new prgram, and places the PCB onto the ready queue

2.6 Requesting OS Services

Operating System and user programs are isolated from each other, but the OS provides service to user programs, so how do they commmunicate?
Any communication between the user and the kernel, is a system call, which are all handled by the OS.

2.7 Boundary Crossings

Here are some examples of getting to the kernel mode from the user mode:

- Boot time (not really a crossing, starts in the kernel)
- Explicit system call – request for service by application
- Hardware interrupt
- Software trap or exception
- Hardware has table of ”Interrupt service routines”

And for Vice Versa, we have

- Jumping to the next application instruction

2.8 System Calls for Process Management

fork Creates a child process identical to the parent

waitpid Waits for a child process to terminate

execve Replace a process’ core image

exit Terminate process execution and return status

2.9 System Calls for File Manamgement

- opening and closing files
- reading from and writing to a buffer
- moving a file pointer with lseek
- getting a file’s status information with stat

2.10 Systems Call Example

```
Read(fd, buffer, nbytes)
```

1. First we push nbytes to the buffer
2. we push the file path name
3. put the code for read in the register
4. we do a trap to the kernel to do a system call
5. (Switched to kernel space) we find the code for the read function
6. it searches for it in a hash table
7. then passes it to a system call handler, which executes the code
8. returns to the caller back in the user space

2.11 System Call Interface

The system call interface is a C library where the C library functions pass arguments to the OS with a system call identifier. The function also executes a special instruction to trap the system mode. The syscall handler figures out which system call is needed and calls a routine for that operation.
This enforcement of separation between the userspace and kernel is for hardware support.

2.12 System Call Operation

There is a fixed number of arguements that can be passed in registers. Often the address of a user buffer containing data (e.g., for `write()`). The result of the system call is returned in the register

2.13 Introduction to Synchronization

2.13.1 Motivating Example

Suppose we write functions to handle withdrawals and deposits to a bank account.
The idea is that we create separate threads for each action, which may run at the bank's central server.
But suppose you share this account with someone and the balance is $1000, you each go to separate ATM machines, you withdraw $100 and your friend deposites $100 at the exact same time. But if your process finishes last, you'll end up with $900, and if their process finishes last, you'll end up with $1100.
We need synchronization for these processes.

2.13.2 Interlevced Schedules

The problems is that the execution of the two processes can be interleaved. One resulting in you ending up with more money and one with you ending up with less money.

2.13.3 What went wrong

Two concurrent threads manipulated a shared resource without any synchronization. The outcome depends on the order in which accesses take place, this is called a **race condition**. We need synchronization to ensure that only one thread at a time can manipulate the shared resource.

2.13.4 Revisiting the Example

T_1, T_2 share X, T_1 increments X by 1 and T_2 decrements X by 1. This is the same problem, since at the machine level, we load it from a register, performing the increment or decrement, and then store it back. This is basically the same bank account problem.

2.13.5 Mutal Exclusion

To solve this problem, we need to define it.
Given a set of n threads $T_0, ..., T_n$, a set of resources shared between threads, and a segment of code which accesses the shared resources, called the **critical section (CS)**.
We want to ensure that only one thread at a time can execute in the critical section, and all other threads are forced to wait on entry, when a thread leaves the CS, another can enter.

2.13.6 What program data is shared between threads?

Local variables are not shared, since each thread has its own stack and local vars are allocated on that.
Global variables and static objects are shared, they are stored in the static data segment, accessible by any thread.
Dynamic objects and other heap objects are shared as well.

2.13.7 The Critical Section Problem

To solve this problem, we need to design a protocol that threads can use to cooperate (code for entering the CS code and code for exiting the CS code).
Each thread must request permission to enter its CS in its **entry** section, the CS may be followed by an **exit** section, the remaining code is the **remainder** section.

2.13.8 Critical Section Requirements

We need to design a protocol that threads can use to cooperate.

1. Mutual Exclusion, If one thread is in the CS then no other is.
2. Progress, If no thread is in the CS and some thread want to enter CS, it should be able to enter in definite time.
3. Bounded waiting (no starvation), If some thread T is waiting on the CS, then there is a limit on the number of times other thread can enter the CS before this thread is granted access.
4. Performace, The overhead of entering and exiting the CS must be small with respect to the work being done on it. This is the most important.

2.13.9 Some Assumptions and Notation

Assume no special hardware instructions, no restrictions on the number of processors.
Assume that basic machine language instructures (like LOAD, STORE, etc.) are atomic, cannot be interrupted.
If there are only two threads, we number them T_0 and T_1

2.13.10 2-Thread Solutions: First Attempt

Let the threads share an integer variable turn that if

```
turn = i
```

then thread T_i is allowed into its CS.
In the code.

```
while (turn != id);
/* this means to keep checking until it is
your id, when it is, continue.*/
turn = 1 - id
```

What is the problem with this? It's bad because there is no prioritizing in it. The example given is that if T_0 is hungry and T_1 is not hungry and it's T_1 turn to access the critical section (which is food in this metaphor), then T_0 has to wait for T_1 to get hungry and eat from the CS.

2.13.11 2-Thread Solutions: Second Attempt

First attempt does not have enough about state of each process, it only remembers which process is allowed to enter its CS.
So in the second implementation, we use 2 booleans, if one boolean is true, the other threads will not enter the CS, and will only enter when the other person's boolean is false.

The problems with this is when both booleans are false, then nothing is done, and when context switch happens before the flag can be declared. This method is not atomic (mutual exclusion is not guaranteed). We can't fix this by changing the order of testing and setting flag variables, this can lead to a deadlock.

2.13.12 2-Thread Solutions: Third Attempt

Combine two key ideas of first two attempts.
The threads share the variables turn and flag, where the flag is an array like before. Check for both the turn variable and the flag boolean to be correct in the while loop.
This means a thread is checking that it wants to enter the CS, and that the other thread is not in the CS.
This is **Peterson's Solution**.

2.13.13 Synchronization Hardware

To build these higher-level abstractions, it is useful to have some help from the hardware.

2.13.14 Atomic Instructions: Test-and-Set Lock (TSL)

No interruptions, but this lock can have multiple instructions.
This uses a lock variable, which is 0 means that nobody is using the lock, and 1 means the lock is in use. Hardware executes this automatically

```
boolean test_and_set(boolean *lock){
  boolean old = *lock;
  *lock = True;
  return old;
}
```

The semantics of this are:

- Record the old value of the variable
- Set the variable to some non-zero value
- Return the old value

This code returns True when in use and False when it is not in use, but then the caller ”acquires” the lock.
So in the implementation, an `acquire()` function checks if the lock is available, and if it returns false, it has been acquired, and a `release()` function sets the lock back.
This is a **spinlock**, uses **busy waiting**, so the thread continually executes the while loop in the acquire function, this consumes CPU cycles, so it is wasting CPU cycles.

3 Tuesday, May 22, 2018

3.1 Producer and Consumer

This is about two processes sharing a limited buffer, the producer puts info into the buffer and the consumer takes info out. It's bad if the consumer consumes nothing and the producer overfilling the buffer so how do we solve this. Our solution is to sleep when we can't perform our action and wake up when we can, so the producer will sleep when the buffer is full and the consumer will sleep when the buffer is empty, and they will both be wake otherwise.
If the buffer has a max size of N, we wake up the producer when it is of size $N - 1$ because they sleep when it reaches N, and the consumder sleeps when it reaches 0, and wakes up when the size is 1.
This is a more relaxed approach, as opposed to always checking.

3.2 Semaphores

These are abstract data types that provide synchronization. They include:

- An integer variable, accessed only through 2 atomic operations.
- The atomic operation **wait** that decrements the variable and block until semaphore is free
- The atomic operation **signal** that decrements the variable and unblocks a waiting thread if there is any
- a queue of waiting threads

3.2.1 Types of Semaphores

- Binary or Mutex
  - Represents a single access to a resource
  - guarantees mutual exclusion to a critical section
- Counting Semaphores
  - Represents a resource with many units available, or a resource allowing certain kinds of unsynchronized concurrent access.
  - Multiple threads can pass the semaphore
  - max number of threads is determined by semaphore's initial value, *count*

This is similar to locks, but semantics are different.
We must ensure that two threads cannot execute wait and signal at the same time though. This is another critical section problem, which we can solve with using lower-level primitives, Uniprocessors to disable interrupts and Multiprocessors, using hardware instructions.

3.3 The reader, writer problem

Consider the following

- An object is shared amount several threads
- Some only read the object, others only write it, we allow multiple concurrent readers, but only one writer. How can we implement this with semaphores.

This is a problem because the writer conflicts with all of the readers. Another critical section.
To solve this, we need 3 variables

One binary semaphore, that is exclusive to reading or writing. This means that either the one writer is writing or ≥ 1 readers are reading. Which thread in the group of readers is in charge of getting and returning the token? Who will return the token? The last one. ”The last to leave the room turns off the light.”

We also need an integer that keeps track of the number of threads reading the object, this is how we detect when a reader is the first or last of a group.

Finally, we need another binary semaphore for control access to readcount.

For implementation, the writer just needs to wait for its global token, once it gets it, writes, and puts the global token back. For the reader, they wait for the read token, then they wait for the global token, once they get it, increment the read count, and then return the token, and when they leave, if read count is 1 (the last reader), they put the global token back.

What if the first reader arrives while the writer is active? What happens? Then they wait for the global token. Then the second reader will wait for the read count token in this scenario if there was a second reader.

3.4 Monitors

It's to easy to make mistakes with counting semaphores. For example, we could accidentally wait for the global token twice or signal twice.
Monitors are an abstract data type with the restriction that only one process at a time can be active within the monitor. This protects local data accessed only by the monitor's procedures, and a process enters the monitor by invoking 1 of its procedures, and other processes that attempt to enter monitor are blocked.

But what if a process in the monitor may need to wait for something to happen? Then that process has to leave and to let another process use the monitor. This provides a special type of variable called a **condition**. The operations on a condition variable are **wait**, to suspend the invoking process, and **signal**, to resume exactly one suspended process.

But what happens if there is one process P in the monitor and it signals the second process Q, waiting in a condition.

- P is already in the monitor, so it does not neat to block.
- Q becomes unlocked by the signal, and want to resume exection in the monitor.
- But both cannot be simultaneously active in the monitor.

There are 3 solutions

- Hoare Monitors
  - `signal()` immediately switches from the caller to a waiting thread
  - Need another queue for the signaler, if signaler was not done using the monitor
- Brinch Hansen Monitors
  - Signaler must exit monitor immediately, so `signal()` is always the last statement in the monitor procedure
- Mesa Monitors
  - `signal()` places a waiter on the ready queue, but signaler continues inside monitor

So how can we implement the readers / writers problem with Monitors?

3.4.1 Bounded Buffer Problem

Consider Producers that want to add to the buffer and Consumers that want to remove from the buffer. So we want to use a Monitor for the buffer, so that only one operation is done at a time, there needs to be one lock, and two conditions, one to make consumers wait, and one to make producers wait.
In terms of implementation, we're going to have an index for where the producers can insert and an index for where consumers can remove. The buffer will be a circular buffer, so adding to the indexes will also be the *mod* of the largest index of the buffer.
If a consumer C enters the monitor and the buffer is empty, the consumer waits on a condition for a producer to put stuff in the buffer and signals the C or another consumer on the enter queue can enter the monitor.

3.5 Process Scheduling

So we have scheduling at different levels of the Operating Systems.
For CPUs, only one process can run at a time on a CPU, the Scheduler decides which process to run and the Goal of CPU scheduling is to give the illusion that processes are running concurrently and maximizing CPU utilization.

3.5.1 What happens on a dispatch /context switch?

When switching the CPU to another process

1. The current process using the CPU is having it's state saved (unless it's going to be exiting)
2. The next process is selected from thr ready queue
3. Then the state of the next process is restored, such as registers.
4. Kernel mode switches to user mode
5. The PC is set to the next instruction in the new process

3.5.2 Process Life Cycle

Processes repeatedly alternate between computation and I/O.

- This is called CPU burts and I/O bursts.
- The last CPU burst ends with a call to terminate the process.

I/O - Bound Processes These processes have short CPU bursts and frequent (long) I/O bursts

CPU - Bound Processes These processes long CPU burts and infrequent I/O bursts
During I/O bursts, CPU is not needed, this is an opportunity to execute another process.

3.5.3 What is processor scheduling?

The allocation of processors to processes over time. This is the key to **multiprogramming**. We want to increase CPU utilization and job throughput by overlapping I/O and computation.

We also have to consider **Policies** in multiprogramming. For example:

- Given more than one runnable process, how do we choose which to run next?
- When do we make this decision?

We schedule these process changes:

- When the running process blocks or exists, such as Operating System Calls (I /O)
- At fixed intervals, such as clock interrupts
- When a process enters the Ready state, such as I /O interrupts, signals, and process creation.

3.5.4 Scheduling Goals

For all systems:

- we want **Fairness**, in that each process recieves a fair share of the CPU
- Avoiding **starvation** (a process waiting too long)
- **Policy Enforcement** – Usage policies should be met
- Balance – That all parts of the system should be busy

.

For batch systems

- Throughput – we want to maximize the jobs completed per hour
- Turn around time – We wanna minimize the time between submission and completion
- CPU Utilization – keep the CPU busy all the time.

For interactive systems

- we want to minimize response time, time between recieving the request and starting to produce output
- Proportionality – ”simple” tasks complete quickly while more ”complicated” ones take longer

For real-time systems

- Deadlines must be met
- Must be predictable

Note that goals sometimes conflict with each other

3.5.5 Types of Scheduling

Non-preemptive scheduling One the CPU has been allocated to a process, it keeps the CPU until it terminates. This is suitable for batch scheduling.

Preemptive Scheduling CPU can be taken from a running process and allocated to another. Needed in interactive or real-time systems.

4 Tuesday, May 29, 2018

4.1 Problem: Dining Philosophers

A philosopher needs two forks to eat, 5 philosophers with 5 meals sharing 5 forks.

One idea of a solution is that if a philosopher gets hungry, grab a left fork and a right fork. But if they all get hungry at the same time, they will all have one fork and can't eat. So we're looking for a better solution.

In the previous non-solution, we only had 2 states, "eating" and "thinking", but in the following solution, we'll have 3 states, "eating", "hungry", and "thinking". One semphore for each philosopher. The Philosopher will think, then call a `take_forks()` function, eat, and then calls a `put_forks()` function. We also need a mutex for entering and exiting the critical section of eating.

What does the `take_forks()` function do? It enter's the critical region with a mutex, since the philosopher and the two philosophers next to that person can change their state. It then calls another function called `test()`, that changes the state to eating if the philosophers next to the one we're currently refering to are not eating, and then exits the critical region. Changing the state to hungry is a critical state since 3 people can change one person's state for all philosophers. The semaphores are used since if `test()` fails since there are not enough forks to eat, then that philosopher will be blocked until a fork is returned, since their semphore cannot be decremented.

After a philosopher finishes eating, he will put down his forks, in teh sense that he will check if the two people sitting next two him want to eat and have access to their other fork.

4.2 Scheduling Algorithms

4.2.1 FCFS

"First come, first serve", all you need is a queue, so it's very easy to implement. However, has no prioritization.

It's problem whoever that it's average waiting time is often quite long and causes the:

Convoy effect All other processes wait for the one big process to release the CPU

4.2.2 SJF

"Shortest Job First", this is better, but we need to know the processing time in advance. It also does not take into account fairness and starvation.

4.2.3 Round Robin

Designed for time-sharing system, is a pre-emptive algorithm. Here the ready queue is circular, each process is allowed to run for time quantum q before being preempted and put back on queue. This is good, but the choice of quantum is critical, the bigger it is, the closer the algorithm is to FCFS, and the smaller it is, the more it becomes like processor sharing, which is inefficient.

However, there is also no prioritization in this solution as well. If we want to enforce prioritization, it would be tricky, like multiple queues. This may cause starvation or priority inversion, as low priority tasks may never run and low priority tasks may prevent high priority tasks from making progress by holding a resource.

4.2.4 Mars Rover Pathfinder Bug Example

The Mars Rover was not making progress being it kept doing context switches, and left all of its work in the shared memory before the context switch. This is priority inversion, as a high priority tasks now has a smaller amount of space on the shared memory to work with

4.3 What do real systems do?

Real systems us a combination of Multilevel queue scheduling, like Round Robin with priorities and Feedback Scheduling, which is adjusting the quantum accordingly by feedback of real usage.

4.3.1 Multi-Level Queue Scheduling

There is going to be two scheduling policies here, the first decides which queue to serve (e.g. based on queue priorities), and second decides which job within a queue to choose.

So in an implementation, a quantum of a lower priority queue would be longer than a quantum of a higher priority queue. So if a highest priority task finishes during quantum, that is a good result, but if not, the task is bumped into second highest priority. This means that the shortest high priority jobs get done first, and that the processes are ordered by how long they take, so we now know if they are short or long jobs.

4.3.2 Feedback Scheduling

We want to give priority to shorter jobs, IO bound jobs, and to interactive jobs, but we don't know beforehand whether a job is short or long and whether it's IO bound or CPU bound.

We can adjust criteria for choosing a particular process based on past history, such as Multi-Level Queues, can prefer processes that do not use full quantum, can change priority of porcesses based on age, can change priority of processes based on CPU consumed so far, can boost priority following a user-input event.

4.3.3 Timesharing in Linux 2.8

Linux always runs the task in the active array with the highest priority, if there are other processes with the same priority, do round robin between then and switch after granularity time units.

If timeslice expires, move to the expired array. This way, tasks are being re arranged and prevents starvation.

But how do we determine priority of a process?

Some processes are more important than others, so static priorities are assigned to processes, and bonuses are added or subtracted depending on if it's an interactive and /or IO bound processes, which can be identified by sleep average (time process has the CPU does not do anything, which is time process is sleeping vs its CPU time.)

How do we determine timeslive and granularity?

Timeslice depends on static priority, which for example, can be anywhere from 5 to 800 milliseconds and Granularity depends on a dynamic priority