Wednesday 6 September 2017

Operating System - Pre-lecture 2 Summary

Processes and Threads

Processes - pseudo-concurrent operation. Very important. When a computer is booted, many processes are started

The Process Model

Sequential process

An instance of an executing program. Conceptually, each process has it's own virtual CPU. A process cannot be programmed with timing in mind , because the CPU may switch to another process for a certain amount of time, ruining the scheduling.

Distinction : Process = Activity of some kind. Has program, input, output, state. Program = Something that can be stored on disk (recipe)

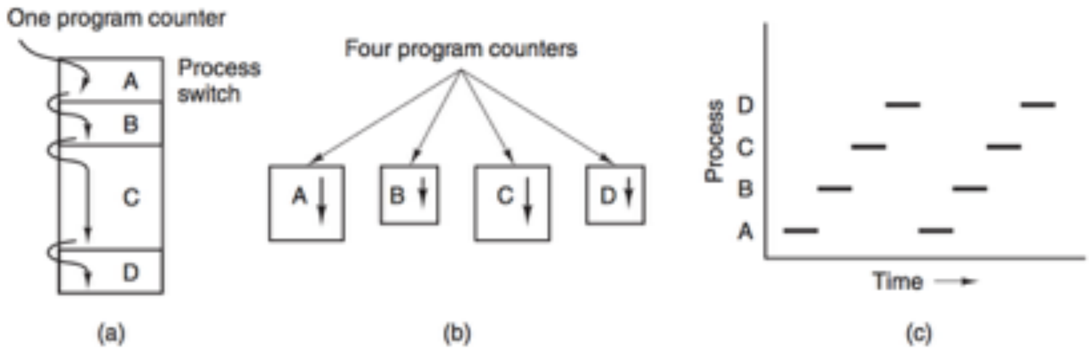


Figure 2-1. (a) Multiprogramming four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

Process creation

System initialization, Execution of process-creation system call, User request to create new process, Initiation of batch job. Fork vs CreateProcess

Daemon: Long-running background processes.

Process Termination

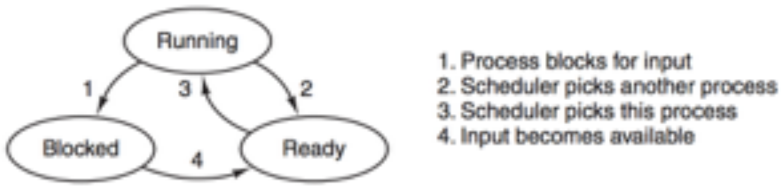
4 conditions Normal exit, Error Exit, Fatal Error, Killed by other process. Of these 4, Normal Exit and Error Exit are voluntary.

Process Hierarchies

When a process creates another process, they become related - One parent, one or more children. These are process groups, and all members of process groups are triggered when an appropriate action happens (example: input on keyboard can trigger many processes). In UNIX, all processes belong to the same tree with *init* as root.

Process States

A process can be Running , Ready (temporarily stopped to let other process run)), or Blocked (unable to run until something happens)



Four transitions possible

1 - OS discovers process cannot continue

2 - Scheduler has decided process has run for long enough

3 - Other processes have run for long enough, this process turn

4 - External event (such as arrival of input) happens

Instead of thinking about interrupts , think about user processes, disk processes, terminal processes, which block when they are waiting for something to happen. When whatever they are supposed to do is finished, they are unblocked.

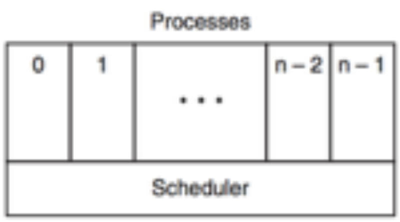


Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

Implementation of Processes

OS implements *process table,* containing important information about each process.

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Figure 2-4. Some of the fields of a typical process-table entry.

This information is necessary when a process changes state, so that it can be restarted as if it hadn't been stopped. Each I/O class also has an *interrupt vector* which is the location in memory where the IRQ-routine is stored.

Interrupt procedure

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly-language procedure saves registers.
4. Assembly-language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly-language procedure starts up new current process.

Figure 2-5. Skeleton of what the lowest level of the operating system does when an interrupt occurs.

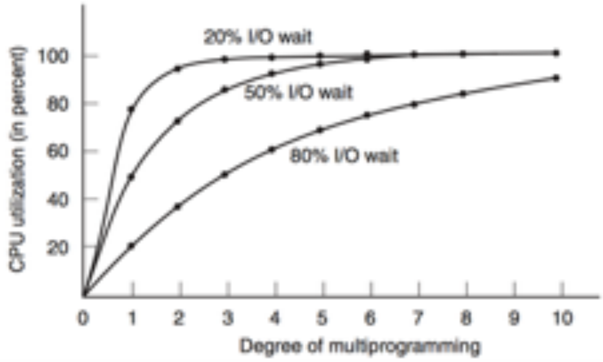
<u>Modeling Multiprogramming</u>

If a process used 20% of CPU when it's in memory, five processes should use 100% right? No, this is optimistic.

Probabilistic formula:

CPU utilization = 1 - p^n, where:

1. p = fraction of time process spends waiting for I/O
2. n = Number of processes



This model is not that accurate because it assume processes are independent and don't have to wait for one another which is not the case in single-core CPU's.