Computer Systems II

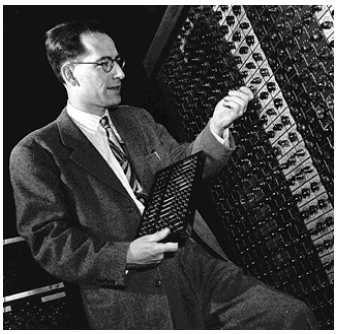
Introduction to Processes

First Two Major Computer System Evolution Steps

Led to the idea of multiprogramming
(multiple concurrent processes)

At First … (1945 – 1955)

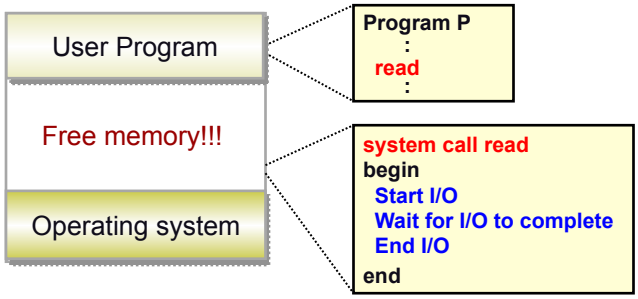
- In the beginning, there really wasn't an OS
  - Program binaries were loaded using switches
  - Interface included blinking lights (cool !)



3

OS Evolution: Step 1

- Very simple OS (1955 – 1965)
  - Library of common subroutines read, write, etc.
  - Single programming (one application at a time)

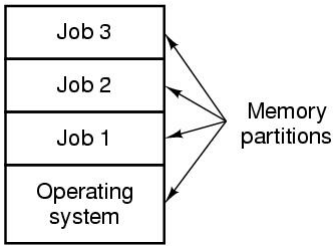


- Problem - system expensive, inefficient
  - Q: What to do while we wait for the I/O device ?
  - A: Run more processes; when one blocks, switch to another

4

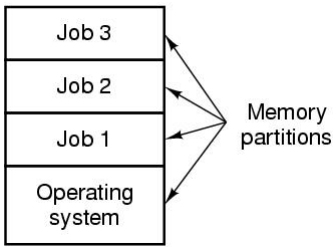
OS Evolution: Step 2

- Multiprogramming (1965 – 1980)
  - Multiple running processes
  - When one process (job) blocks, switch to another



5

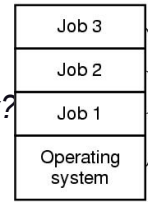
Multiprogramming Issues



- Processes (jobs) share the memory
  - *What if a process starts scribbling in memory?*
- Processes share the CPU
  - *What if a program has infinite loops?*

6

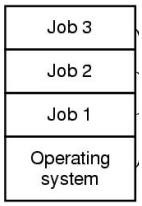
Multiprogramming Issue 1



- Processes (jobs) share the memory
  - *What if a process starts scribbling in memory?*
- *Solution:* Protection
  - A program cannot access memory outside its own space
  - But what about the operating system?
- *Solution*: Two execution modes
  - *user* mode and *system* (*supervisor, monitor*) mode
  - OS executes in system mode, user programs in user mode.
  - Attempts to execute privileged instructions while in user mode results in a trap (to be discussed later).

7

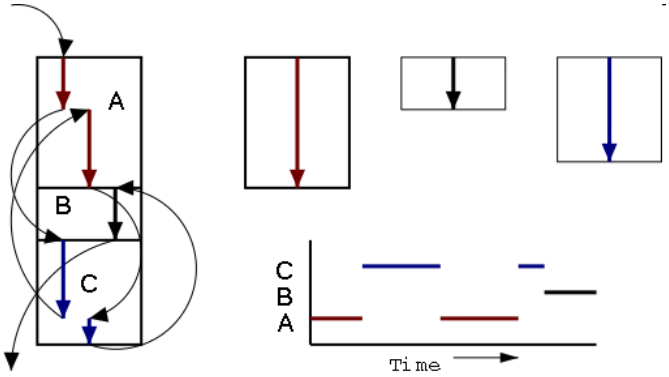
Multiprogramming Issue 2



- Processes (jobs) share the CPU
  - *What if a program has infinite loops?*
- *Solution:* Timer Interrupts
  - The OS sets the internal clock to generate an interrupt at some specified future time.
  - This interrupt time is the process quantum.
  - When the timer interrupt occurs, the CPU switches to another process. This prevents the system being held up by processes in infinite loops.

8

Timer Interrupts



- Multiprogramming is also called ***pseudoparallelism*** since one has the illusion of a parallel processor.
- Different from ***real parallelism*** in which two or more processes are actually running at once on two different processors.

9

Timer Interrupts Example

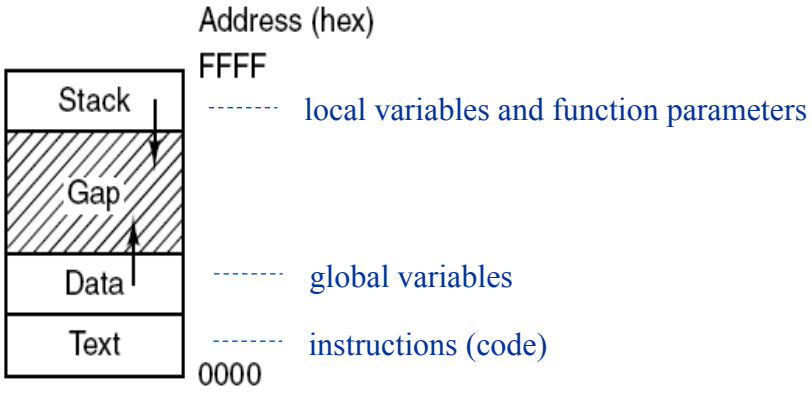
- Suppose P1, P2 and P3 are ready to run (in this order). P1 executes for 3 time units, P2 executes for 4 times units, and P2 executes for 2 time units. Neither process invokes I/O. Assuming that a timer interrupt occurs every time unit, fill in the table below:

Time (in units)	Running	Ready to Run
1		
2		
3		
4		
5		
6		
7		
8		
9		

10

Processes

Recall: Memory Layout of a Process

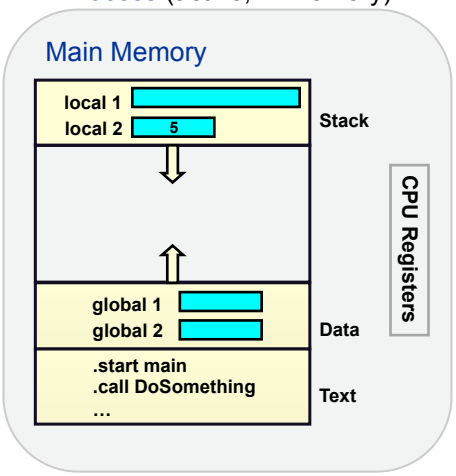


Program vs. Process

Program (passive, on disk)

```
int global1 = 0;
int global2 = 0;

void DoSomething()
{
   int local2 = 5;

   local2 = local2 + 1;
   ...
}

int main()
{
   char local1[10];

   DoSomething();
            ...
}
```

Process (active, in memory)

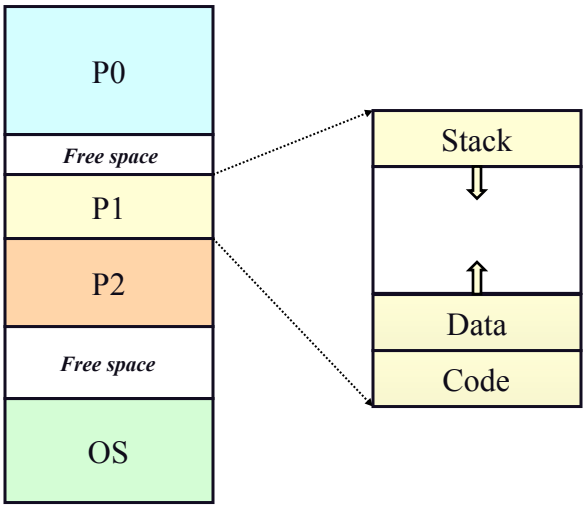


- A process is a program in execution

13

What if More Processes in Memory?

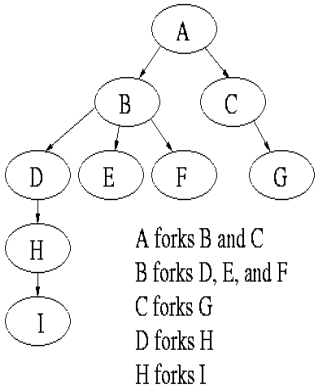
- Each process has its own Code, Data, Heap and Stack



14

Process Creation (1)

- Modern general purpose operating systems permit a user to create and destroy processes.
- In Unix this is done by the **fork** system call, which creates a **child** process, and the **exit** system call, which terminates the current process.

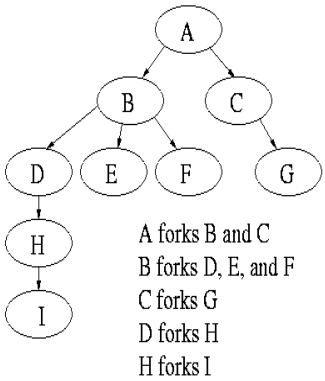


15

Process Creation (2)

- After a **fork**, both parent and child keep running, and each can fork off other processes.
- A process tree results. The root of the tree is a special process created by the OS during startup.

- A process can *choose* to wait for children to terminate. For example, if C issued a **wait**() system call, it would block until G finished.



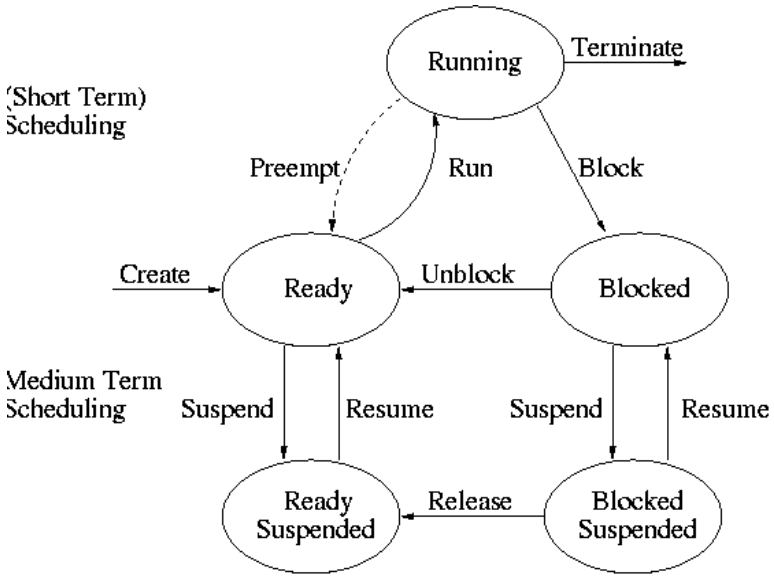
16

Bootstrapping

- When a computer is switched on or reset, there must be an initial program that gets the system running
- This is the bootstrap program
  - Initialize CPU registers, device controllers, memory
  - Load the OS into memory
  - Start the OS running
- OS starts the first process (such as “init”)
- OS waits for some event to occur
  - Hardware interrupts or software interrupts (traps)

17

Process States and Transitions



18

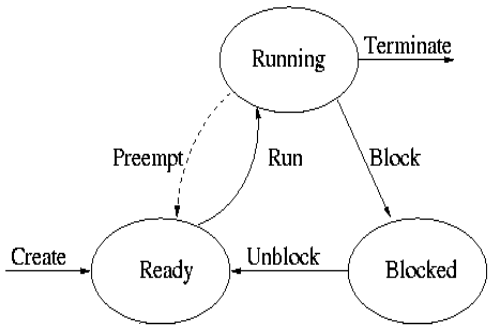
Run, Blocked and Ready States

- Consider a running process P that issues an I/O request
  - The process blocks
  - At some later point, a disk interrupt occurs and the driver detects that P's request is satisfied.
  - P is unblocked, i.e. is moved from blocked to ready
  - At some later time the operating system picks P to run
- Ready:
  - Processes not allocated to the CPU, but ready to run
- Running:
  - Process (one!) allocated to the CPU
- Blocked:
  - Process waiting for event to occur before ready to run

19

Transitions

- Ready – Running:
- Running – Ready:
- Running – Blocked:
- Blocked – Ready:

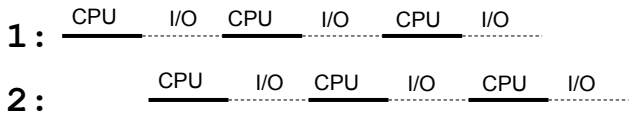


- Suspend and Resume states will be discussed later (In the memory management module.)

20

When to Change the Running Process?

- When a process is stalled waiting for I/O
  - Better utilize the CPU, e.g., while waiting for disk read



- When a process has been running for a while
  - Sharing on a fine time scale to give each process the illusion of running on its own machine
  - Trade-off efficiency for a finer granularity of fairness

21

State Transition Example

- Suppose P2, P3, P4 are in this order in the Ready Queue, and that the following sequence of events occurs:

Time 5: P1 invokes read
Time 15: P2 invokes write
Time 20: Interrupt (P1's read complete)
Time 30: P3 terminates.
Time 35: Interrupt (P2's write complete)
Time 40: P4 terminates
Time 45: P1 terminates

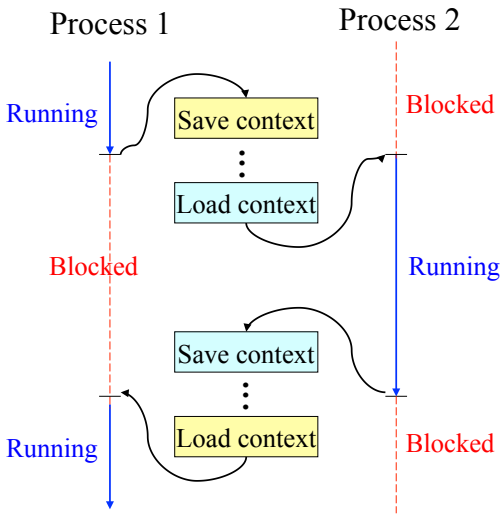
Time	Running	Ready	Blocked
5+ε			
15+ε			
20+ε			
30+ε			
35+ε			
40+ε			
45+ε			

Assume that processes are always appended at the *end* of a queue, and the *first* process in the queue is always scheduled. Fill in the table to show the process states after each event.

22

Switching Between Processes

- Context
  - State the OS needs to restart a preempted process, stored in the PCB
- Context switch
  - Save the context of current process
  - Restore the saved context of some previously preempted process
  - Pass control to this newly restored process



23

Context: What the OS needs to save

Process identifier

Process state

New, ready, blocked, halted

CPU registers

I/O status information

Open files, I/O requests, …

Memory management information

Page tables (to be discussed later)

CPU scheduling information

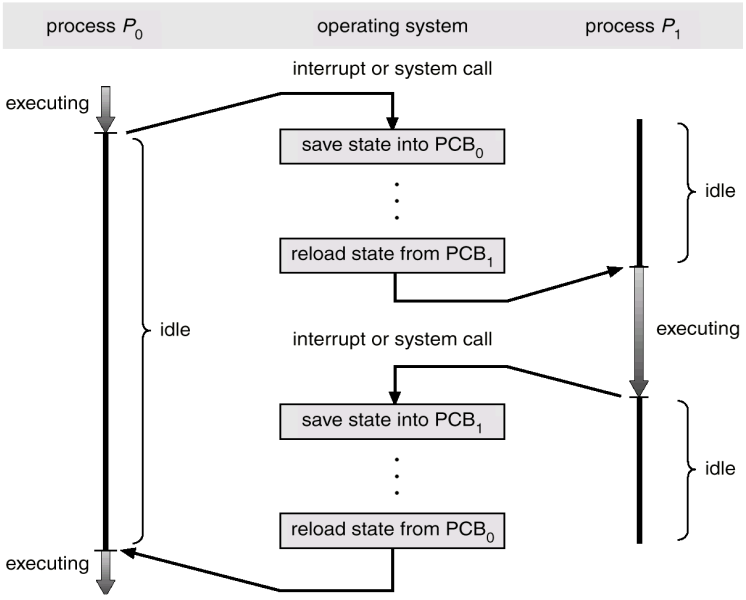
Priority (to be discussed later)

Process Control Block (PCB)

- most important data structure
- read and written by every OS module

24

Context Switching Revisited



25

Summary

- Process
  - Program in execution
- Multiprogramming
  - Processes executing concurrently
  - Take turns using the CPU
- Process States and Transitions
  - Run, Blocked and Ready
  - Suspend (on disk) and Resume
- Process Control Block
  - Information associated to each process
- Context Switching

26