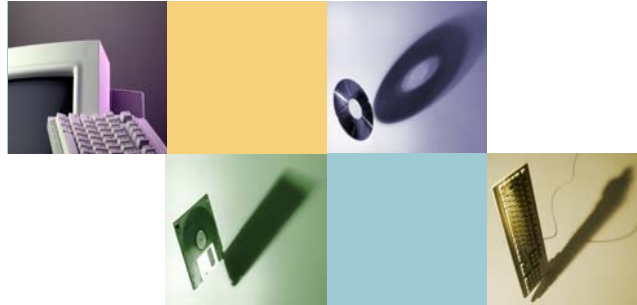


Multi-programming/processing



CSC 469/557

This ppt is based on Silberschatz's Dino OS book

Multi-Processing

- *Running program, job and process* almost interchangeably.
- *Independent* process cannot affect or be affected by the execution of another process.
- *Cooperating* process can affect or be affected by the execution of another process
 - Perhaps exchanging info or control via some mechanisms (comm & sync)
 - Advantages of process cooperation (concurrency)
 - Information sharing
 - Computation speed-up
 - Modularity
 - Convenience



Process Concept

- Process – a program in execution; process execution must progress in sequential fashion.
- A process includes:
 - program counter
 - stack
 - data section
- Support a concept of multiple programming or processing
- Thus process must have state

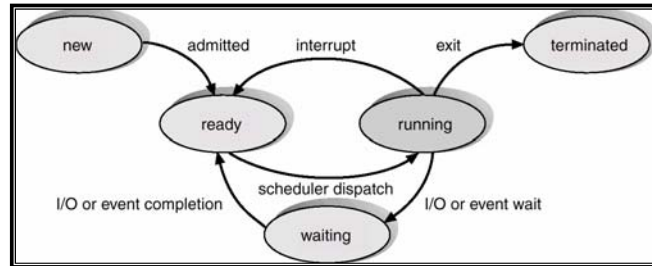


Process State

- 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 process.
 - **terminated**: The process has finished execution.



Diagram of Process State



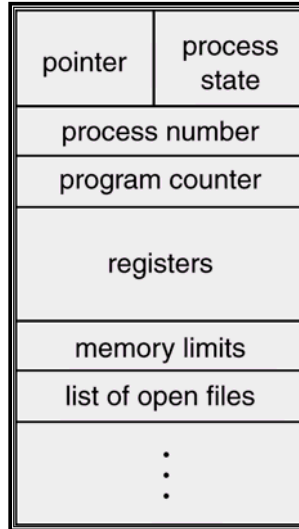
Process Control Block (PCB)

Information associated with each process.

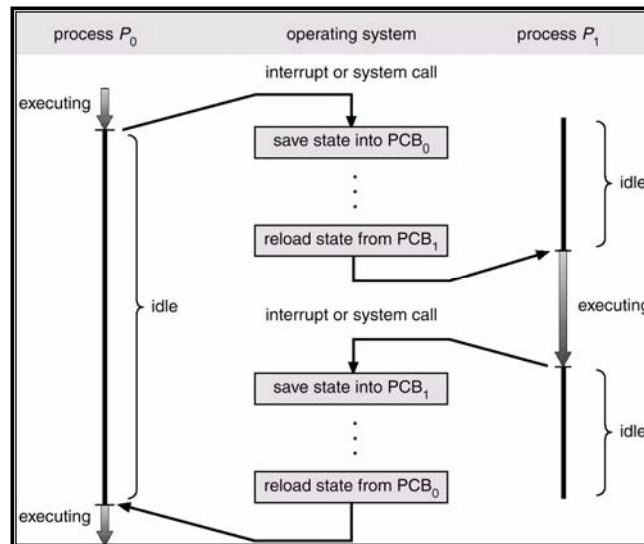
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information



Process Control Block (PCB)



CPU Switch From Process to Process

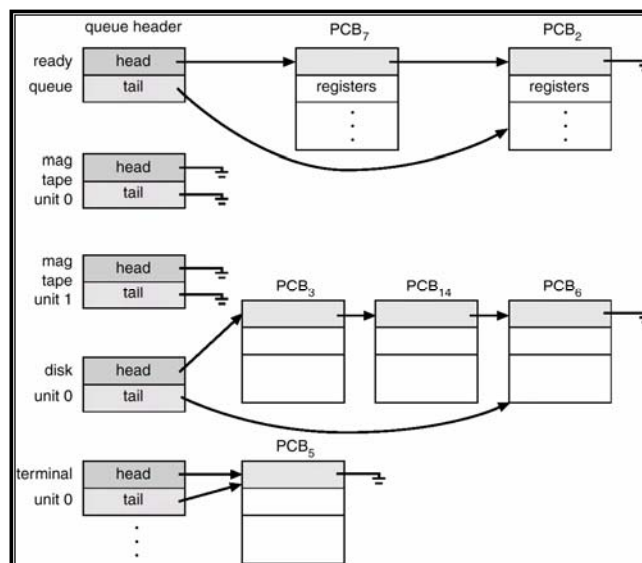


Process Scheduling Queues

- Job queue – set of all processes in the system.
- Ready queue – set of all processes residing in main memory, ready and waiting to execute.
- Device queues – set of processes waiting for an I/O device.
- Process migration between the various queues.



Ready Queue And Various I/O Device Queues



Schedulers

- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue.
- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU.

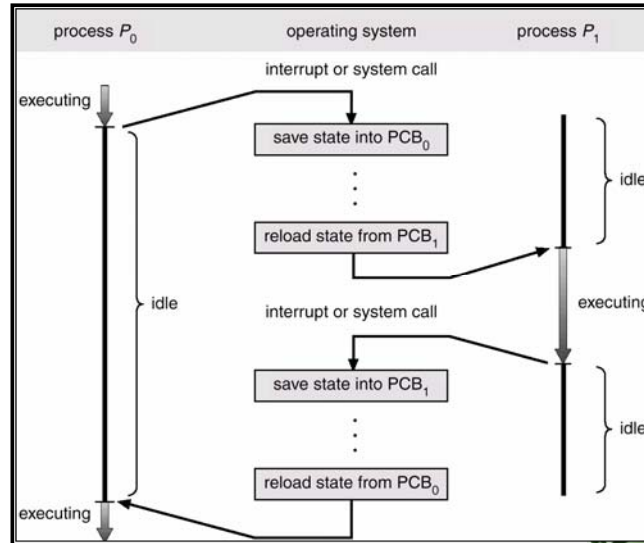


Context Switch

- 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.
- Context-switch time is overhead; the system does no useful work while switching.
- Time dependent on hardware support.



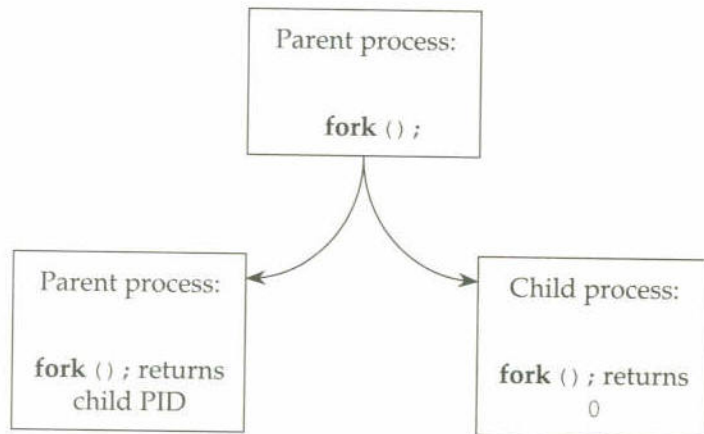
CPU Switch From Process to Process



Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes.
- Resource sharing
 - Parent and children share all resources.
 - Children share subset of parent's resources.
 - Parent and child share no resources.
- Execution
 - Parent and children execute concurrently.
 - Parent waits until children terminate.

Forking a child process



Process Creation & execution (sample code)

```
// forking a child process
if (fork() == 0) { // the child process
    if (execvp (arg1[0], arg1) == -1) {
        cerr << "error executing a given
command" << endl;
    }
}
else { // the parent process
    pid = wait(&status);
    cout << "the child's execution is
completed" << endl;
}
}
```

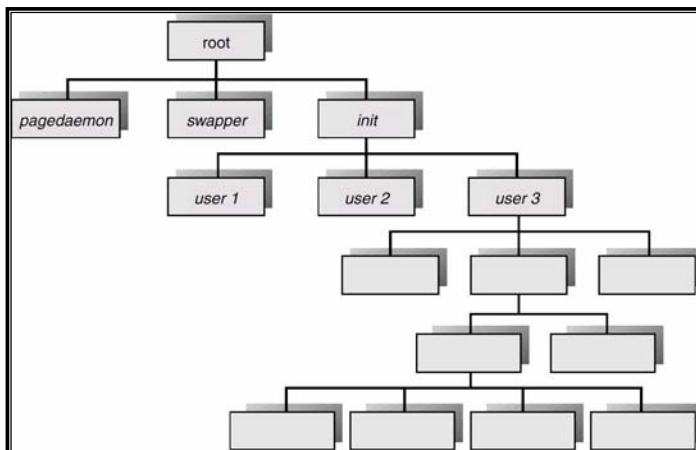


Process Creation (Cont.)

- Address space
 - Child duplicate of parent.
 - Child has a program loaded into it.
- UNIX examples
 - **fork** system call creates new process
 - **exec** system call used after a **fork** to replace the process' memory space with a new program.



Processes Tree on a UNIX System



Process Termination

- Process executes last statement and asks the operating system to decide it (**exit**).
 - Output data from child to parent (via **wait**).
 - Process' resources are deallocated by operating system.
- Parent may terminate execution of children processes (**abort**).
 - Child has exceeded allocated resources.
 - Task assigned to child is no longer required.
 - Parent is exiting.
 - Operating system does not allow child to continue if its parent terminates.
 - Cascading termination.



Interprocess Communication (IPC)

- 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 two operations:
 - **send**(*message*) – message size fixed or variable
 - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
 - establish a *communication link* between them
 - exchange messages via send/receive
- Implementation of communication link
 - physical (e.g., shared memory, hardware bus)
 - logical (e.g., logical properties)



Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?



Direct Communication

- Processes must name each other explicitly:
 - **send** ($P, message$) – send a message to process P
 - **receive**($Q, message$) – receive a message from process Q
- Properties of communication link
 - Links are established automatically.
 - A link is associated with exactly one pair of communicating processes.
 - Between each pair there exists exactly one link.
 - The link may be unidirectional, but is usually bi-directional.



Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports).
 - Each mailbox has a unique id.
 - Processes can communicate only if they share a mailbox.
- Properties of communication link
 - Link established only if processes share a common mailbox
 - A link may be associated with many processes.
 - Each pair of processes may share several communication links.
 - Link may be unidirectional or bi-directional.



Indirect Communication

- Operations
 - create a new mailbox
 - send and receive messages through mailbox
 - destroy a mailbox
- Primitives are defined as:
 - send**(*A, message*) – send a message to mailbox A
 - receive**(*A, message*) – receive a message from mailbox A



Indirect Communication

- Mailbox sharing
 - P_1 , P_2 , and P_3 share mailbox A.
 - P_1 , sends; P_2 and P_3 receive.
 - Who gets the message?
- Solutions
 - Allow a link to be associated with two or more processes.
 - Allow only one process at a time to execute a receive operation.
 - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.



Synchronization

- Message passing may be either blocking or non-blocking.
- **Blocking** is considered **synchronous**
- **Non-blocking** is considered **asynchronous**
- **send** and **receive** primitives may be either blocking or non-blocking.



UNIX/Linux IPC

- Read/Write to a file
- Pipe (at command line, pipe, mknod)
- Message Queue (mailbox concept)
- Semaphore (special
- Shared Memory



Pipe

- unnamed fifo structure for simple process communication
- `Cmd1 | cmd2`
- Normal steps
 - Create a pipe
 - Duplicate a pipe I/O streams to child/parent's ones and close the unused one
- See sample code



Named FIFO (pipe)

- similar to Unix pipe but have a name.
Create
 - “/bin/mknod name p”
 - mknod (2) , e.g.
 - `mknod (“/tmp/mypipe”, S_IFIFO, 0)`
 - simply open a file for read (server) and write (client)
 - `fd = open (“/tmp/mypipe”, 0)`
- Use unlink to clean up

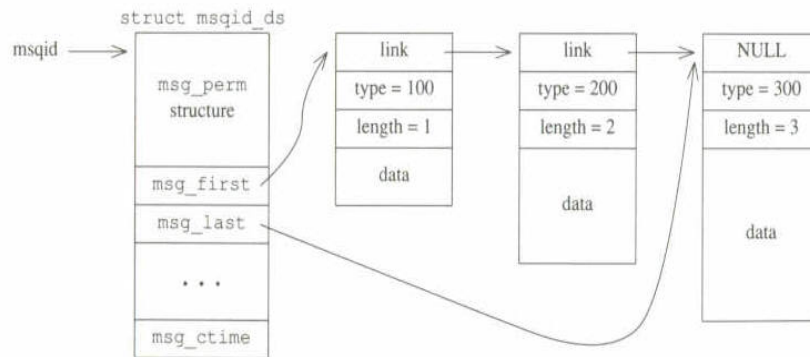


Message Queue

- Similar to mailbox concept in OS book
- In kernel structure
 - Msgqid (key identifies which queue)
 - Each message
 - Type
 - Length
 - Data
 - Create a msg queue with `msgget()`,
 - Send/receive with `msgsnd()`, `msgrcv()`
 - Remove a message queue `msgctl()`



Msg queue structure



Semaphore

- Synchronization primitive (integer value)
- In kernel and use to synchronized the access to shared memory
- Set of semaphore operation guaranteed for atomic
- Should not be used to exchange large amount of data
- Semget(), semop()



Shared memory

- Not in the kernel
- Used with semaphore
- Speed up the access to shared info
- Operations
 - Create : shmget(),
 - Attach to shared memory: shmat()
 - Do something (normal C/C++ operations)
 - Detach : shmdt()
 - Remove: shmctl()



Sample code

- **Server**
 - Create a sh mem
 - Attach to it
 - Create 2 semaphores
 - Client will start first (1)
 - Server initialized to 0
 - Wait to get a request (by reading a filename from a shared mem)
 - Open a file and write content into a shared mem
- **Client**
 - Get a share
 - Attach to it
 - Get a semaphore
 - Write a filename
 - Inform a server to start via server semaphore
 - Wait to get a turn to read a file content



Sample man page

- NAME
- msgget - get a message queue identifier
- SYNOPSIS
- # include <sys/types.h>
- # include <sys/ipc.h>
- # include <sys/msg.h>
- int msgget (key_t key, int msgflg)



Socket Communication

