

Module 4: Processes

Super important and useful

- Process Concept
- Process Scheduling
- Operation on Processes
- Cooperating Processes
- Interprocess Communication

Process Concept

- An operating system executes a variety of programs:
 - Batch system – jobs
 - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably.
- Process – a program in execution; process execution must progress in sequential fashion.
- A process includes:
 - program counter
 - stack
 - data section

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.

→ Gets all resources except CPU *init*

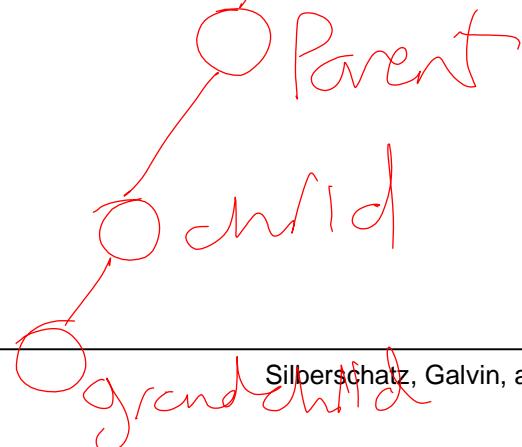
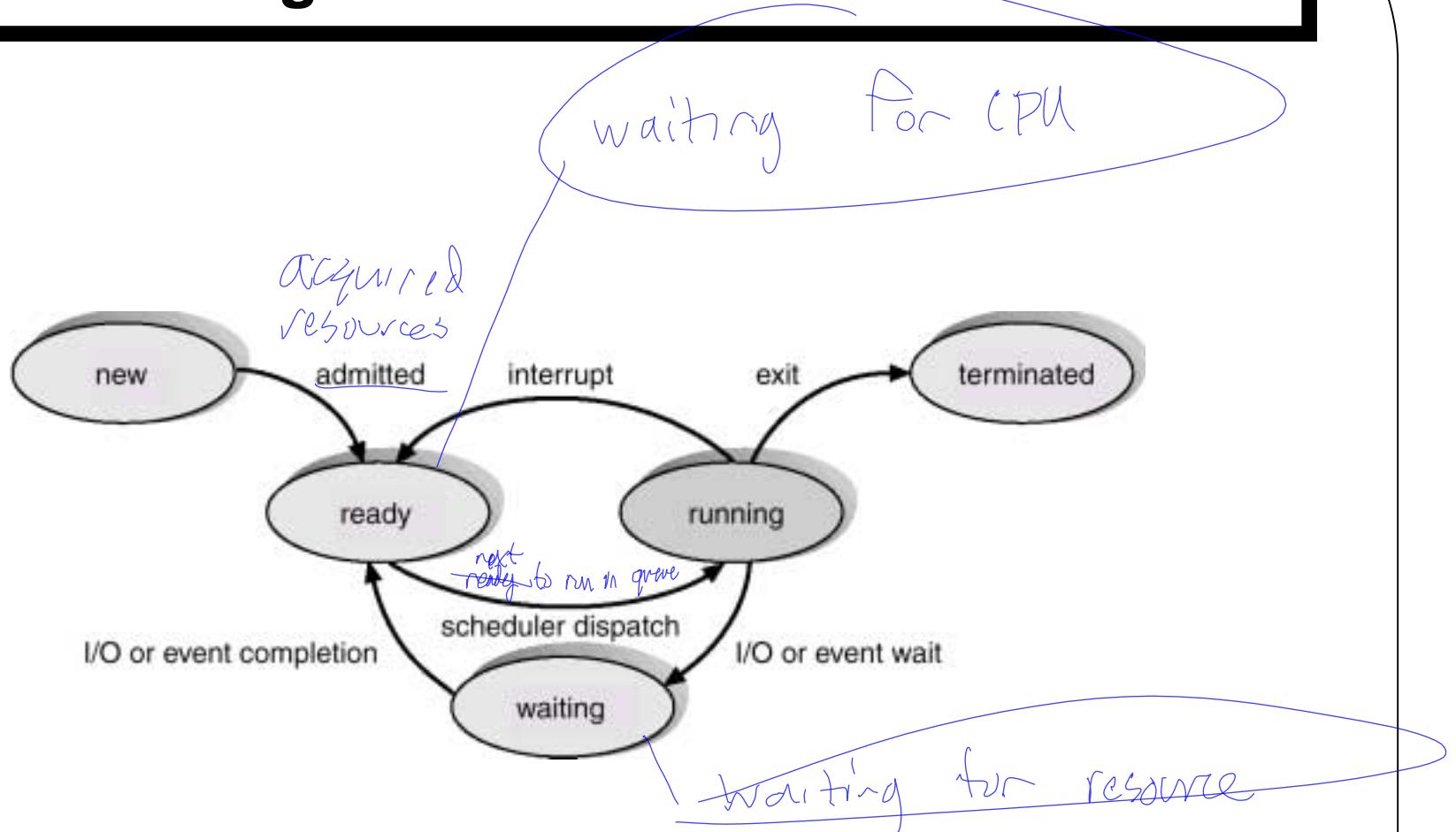


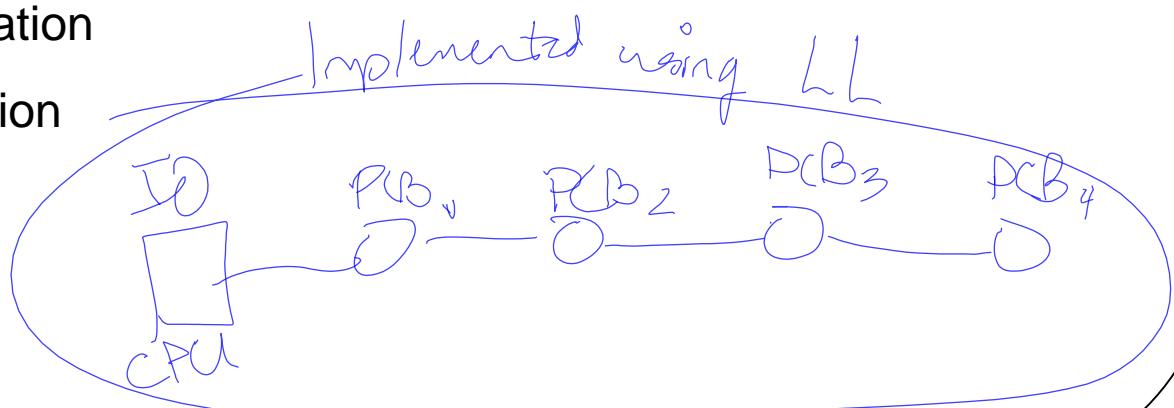
Diagram of Process State



Process Control Block (PCB)

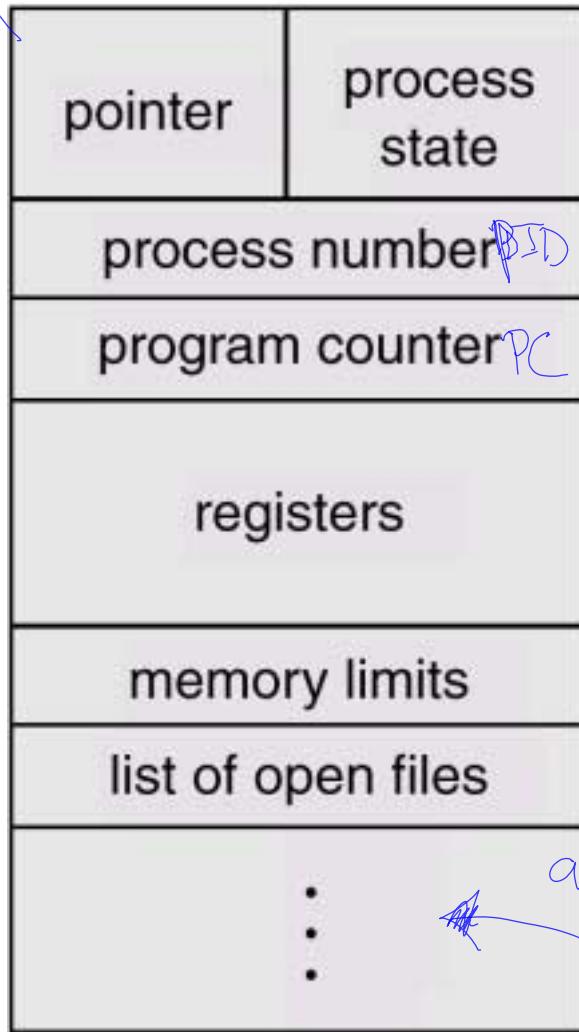
Information associated with each process.

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



Process Control Block (PCB)

Is a queue
like a
linked list



each process

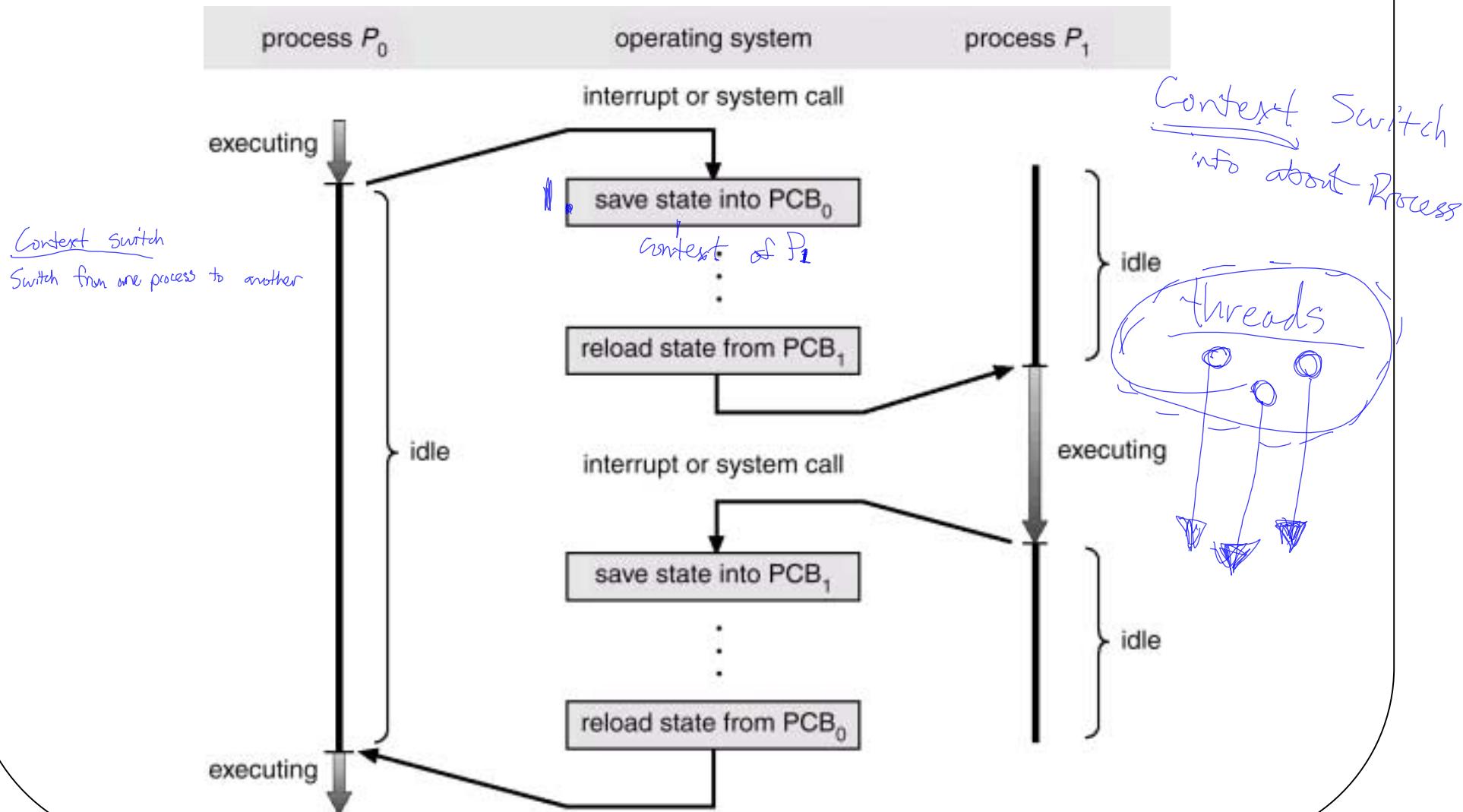
has one

Sometimes used

as a reference

→ SW interrupt

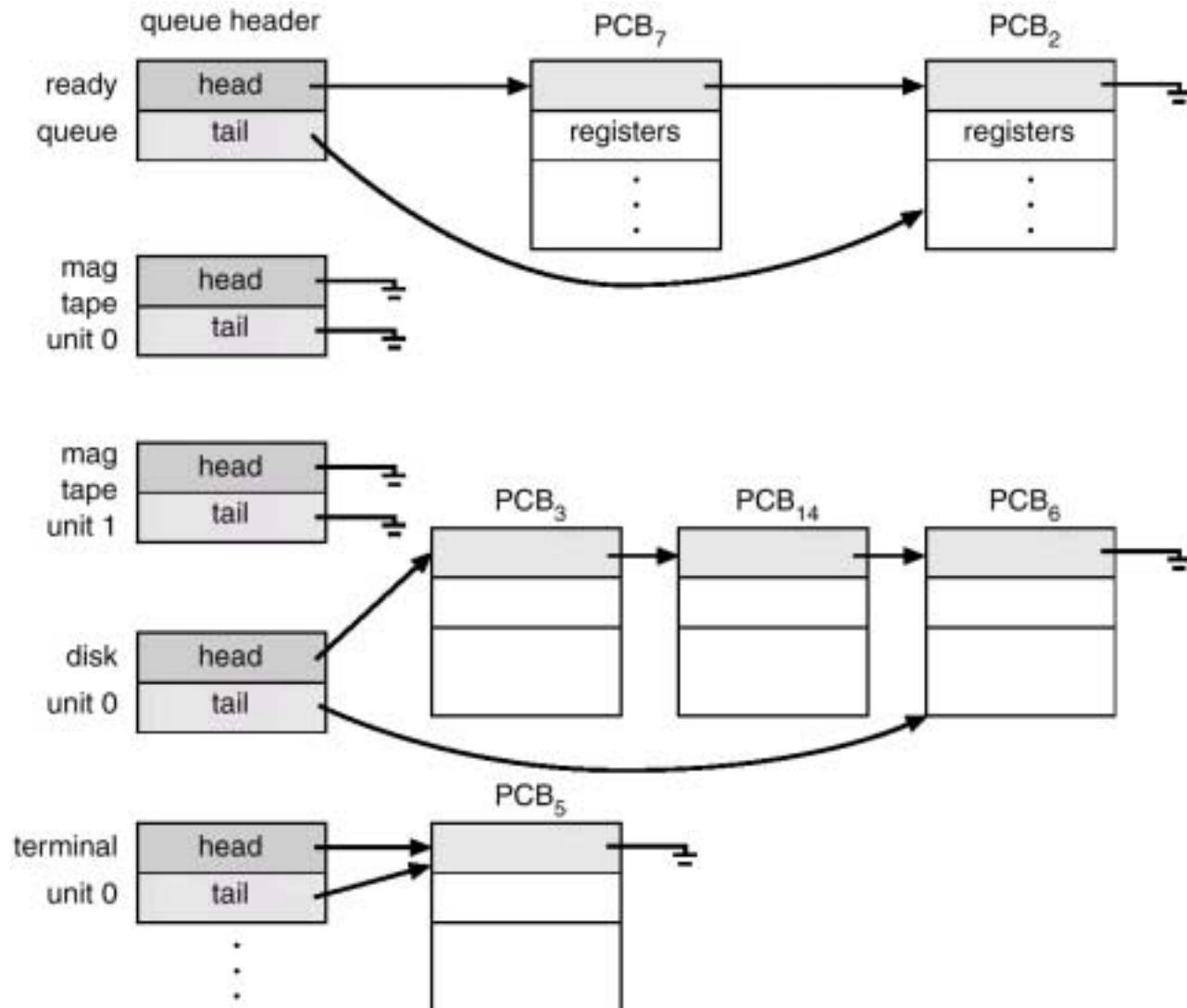
CPU Switch From Process to Process



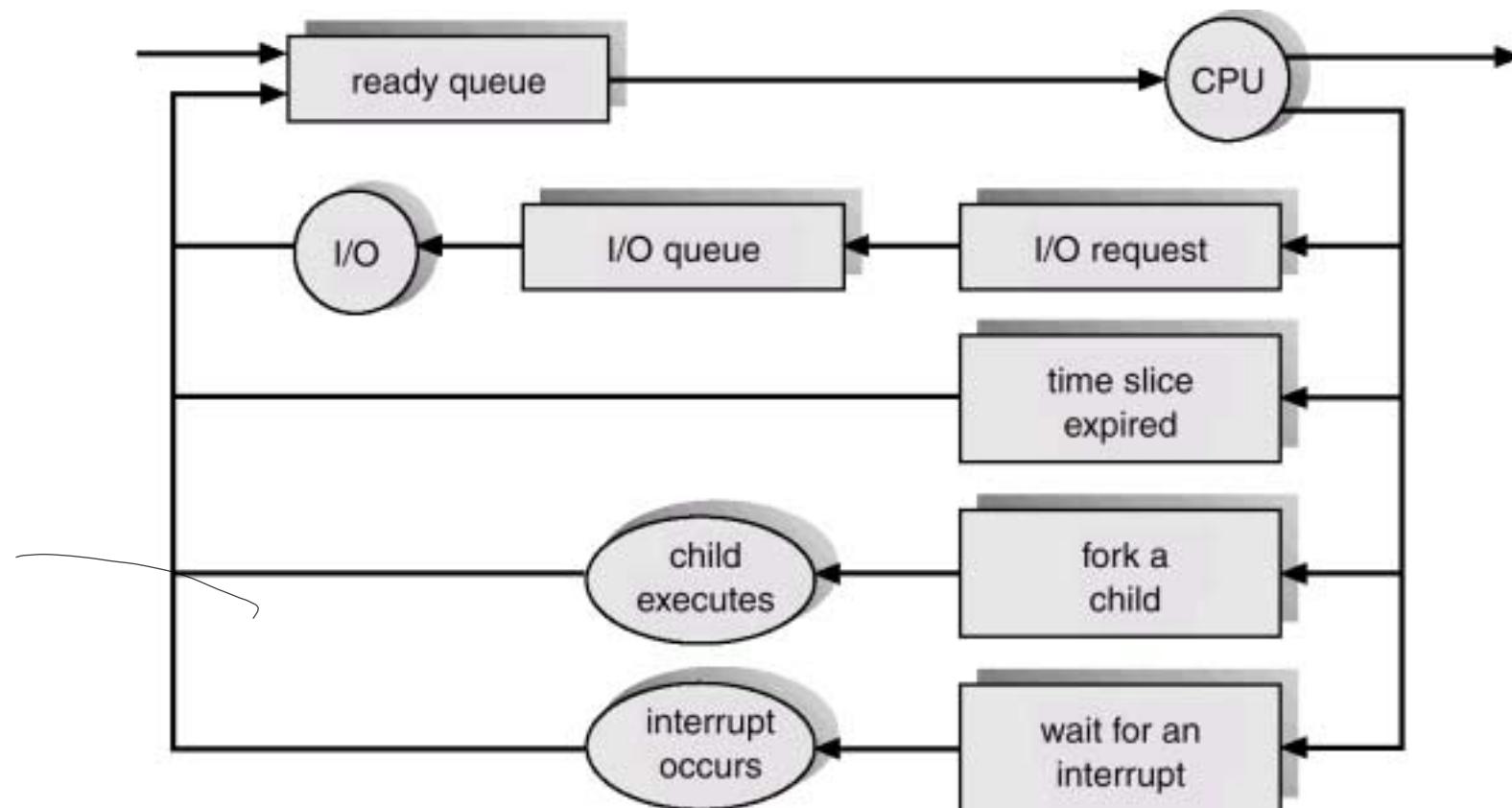
Process Scheduling Queues

- 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



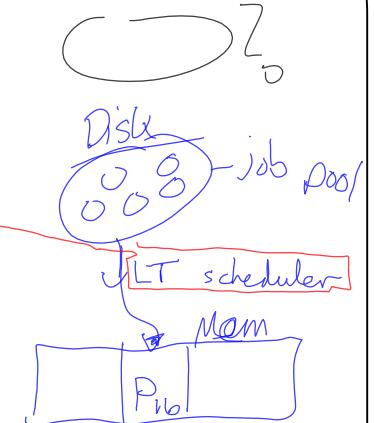
Representation of Process Scheduling



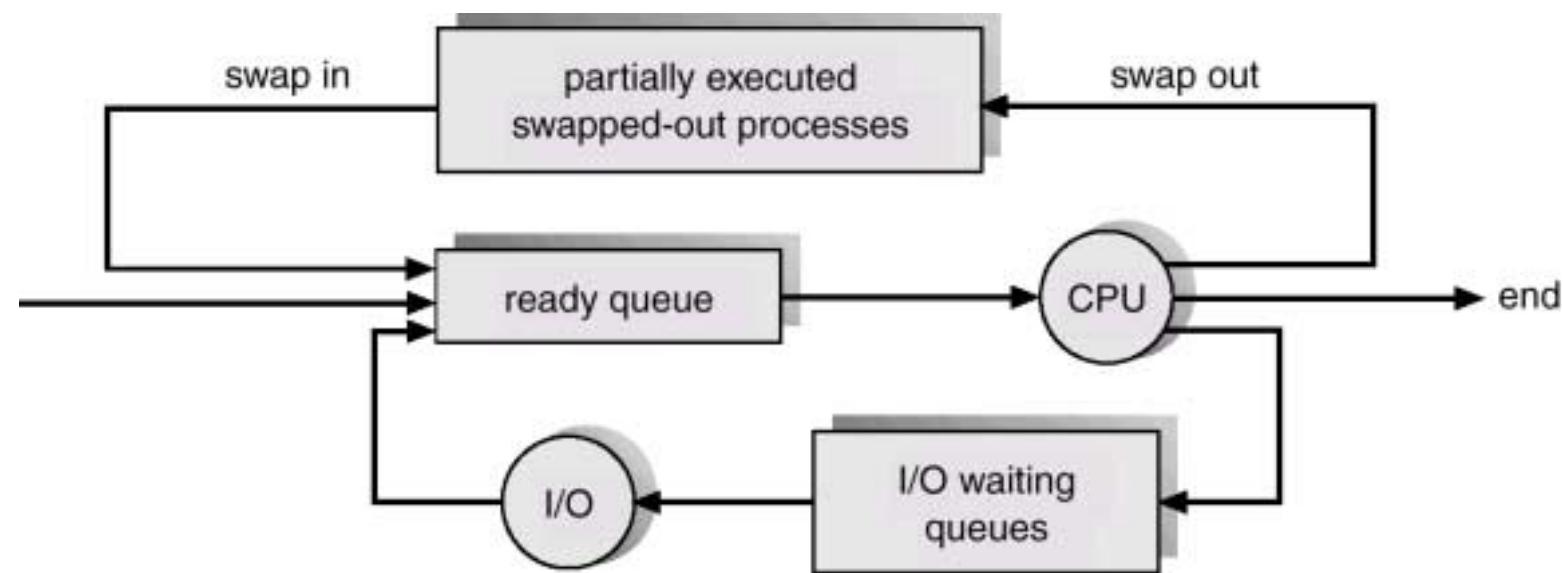
Schedulers

- 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.

Medium Term sch - Swaps in/out partially
executed processes



Addition of Medium Term Scheduling



Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast).
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow).
- The long-term scheduler controls the *degree of multiprogramming*.
- Processes can be described as either:
 - I/O-bound process – spends more time doing I/O than computations, many short CPU bursts.
 - CPU-bound process – spends more time doing computations; few very long CPU bursts.

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.

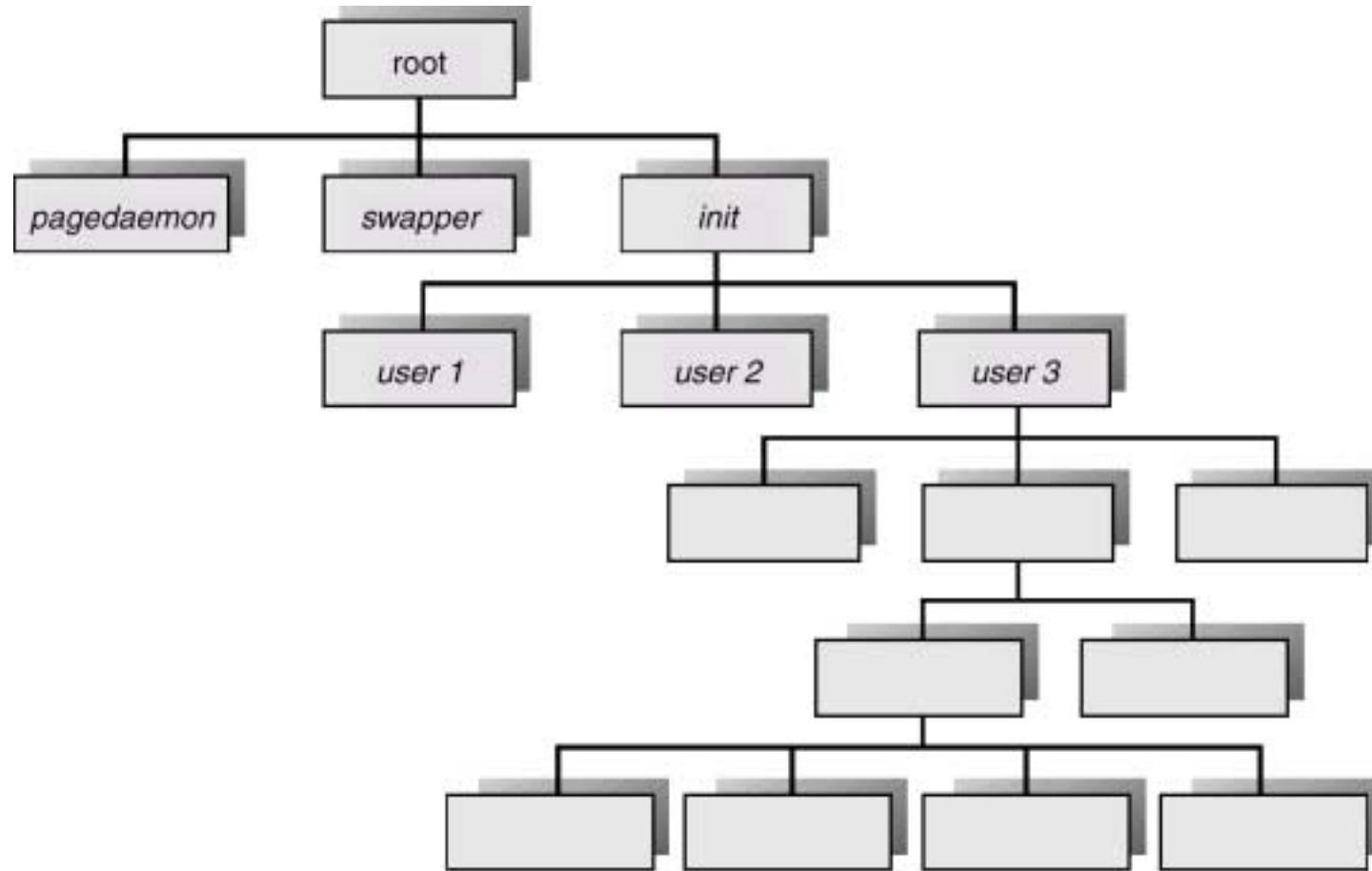
Process Creation

- Parent process creates 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.
- wait (All?)*
- How does process know to continue?*
- OS sends SIGCHLD signal.*

Process Creation (Cont.)

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

A Tree of Processes On A Typical 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). *wait will release PID*
 - 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.

1. child dead & parent alive : Pid taken
S. Kill parent

2. child alive & parent dead : init chd +

Cooperating Processes

- *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
- Advantages of process cooperation

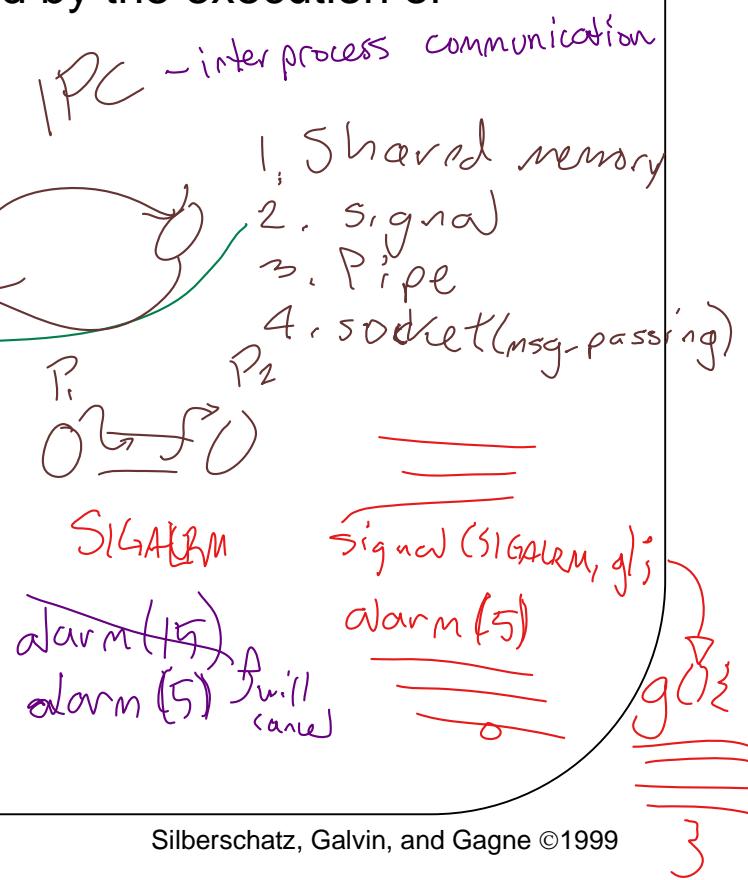
SIGINT: Ctrl+C
SIGALRM
SIGPIPE Broken pipe
SIGCHLD: child finished

\$gcc -o f f.c

./f

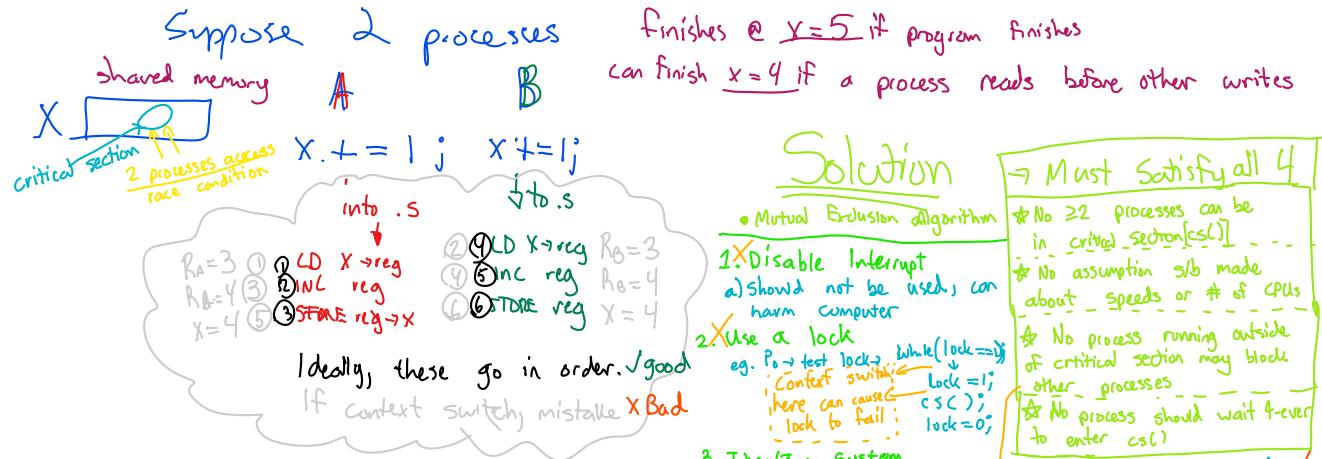


signal
1. Use keyboard
Ctrl+C = SIGINT
Shift+~ = SIGABRT
2. CMD \$KILL -9 5322 pid
3. Syscall kill(pid, SIGSTOP)



Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.
 - unbounded-buffer* places no practical limit on the size of the buffer.
 - bounded-buffer* assumes that there is a fixed buffer size.



Solution

Mutual Exclusion algorithm

1. Disable Interrupt

a) Should not be used; can harm computer

2. Use a lock

e.g. P₀ → test lock₂, while(lock == 0)
Context switch
here can cause lock to fail
lock = 0;

3. Token/Turn System

```
P0 while(true){  
  while(turn != c)  
    csc();  
  turn++;  
  non-csc();  
  process  
  non-csc();  
}  
when(){  
  wh(turn != c)  
  csc();  
  turn=0  
  non-csc();  
}
```

4. Flag system

```
P0 other = 1;  
3 flag[0] = t;  
wh(flag[0] == t){  
  7 csc();  
  9 flag[0] = f;  
}
```

5. Peterson's Alg

```
P0 flag[0] = t;  
turn = 1;  
wh(flag[0] && turn == 1) wh(flag[1] || turn == 0)  
  wait();  
flag[0] = f;  
flag[1] = f;
```

6. Semaphore

```
P(S) {while(ss>0);  
  S--;  
} init  
V(S): stat+;  
or  
  5. P();  
  6. V();  
  2. V();  
all-or-nothing
```

Must satisfy all 4

No 2 processes can be in critical section(cs1)

No assumption s/b made about speeds or # of CPUs

No process running outside of critical section may block other processes

No process should wait forever to enter cs()

other shared
1 other = 1;
2 other = 0;
3 flag[0] = t;
4 flag[1] = t;
5 wh(flag[0] == t){
 7 csc();
 9 flag[0] = f;
 8 csc();
 10 flag[0] = f;

gets stuck (starved)

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
var n;  
type item = ... ;  
var buffer. array [0..n-1] of item;  
in, out: 0..n-1;
```

- Producer process

```
repeat
```

```
...
```

```
produce an item in nextp
```

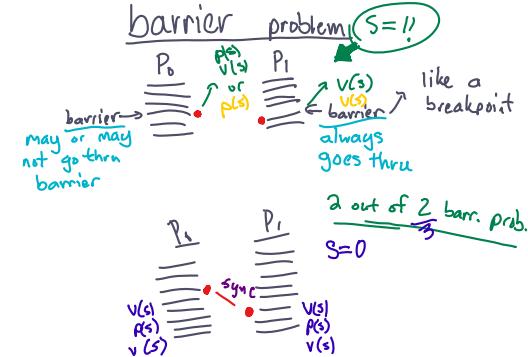
```
...
```

```
while in+1 mod n = out do no-op;
```

```
buffer [in] :=nextp;
```

```
in :=in+1 mod n;
```

```
until false;
```



Bounded-Buffer (Cont.)

- Consumer process

repeat

while *in* = *out* **do** *no-op*;

nextc := *buffer* [*out*];

out := *out*+1 **mod** *n*;

...

consume the item in *nextc*

...

until *false*;

- Solution is correct, but can only fill up $n-1$ buffer.

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.
- Operations
 - create a new mailbox
 - send and receive messages through mailbox
 - destroy a mailbox

Indirect Communication (Continued)

- 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 at most two 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.

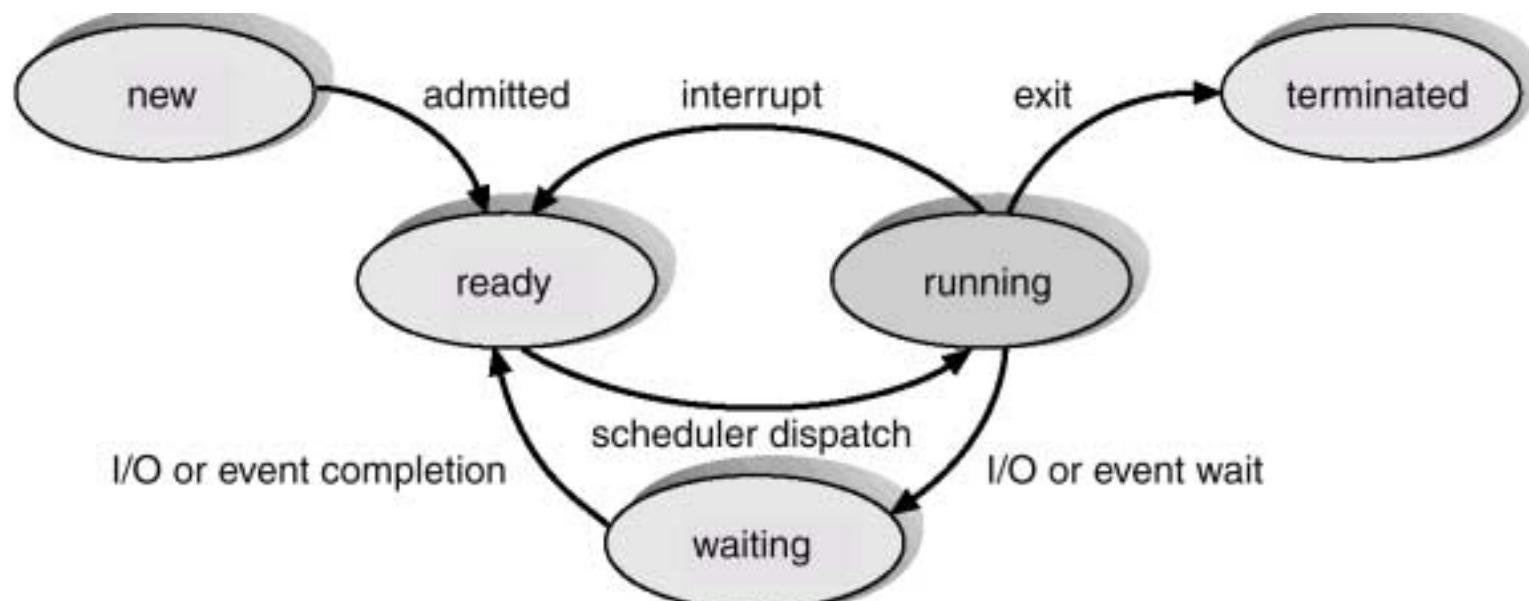
Buffering

- Queue of messages attached to the link; implemented in one of three ways.
 1. Zero capacity – 0 messages
Sender must wait for receiver (rendezvous).
 2. Bounded capacity – finite length of n messages
Sender must wait if link full.
 3. Unbounded capacity – infinite length
Sender never waits.

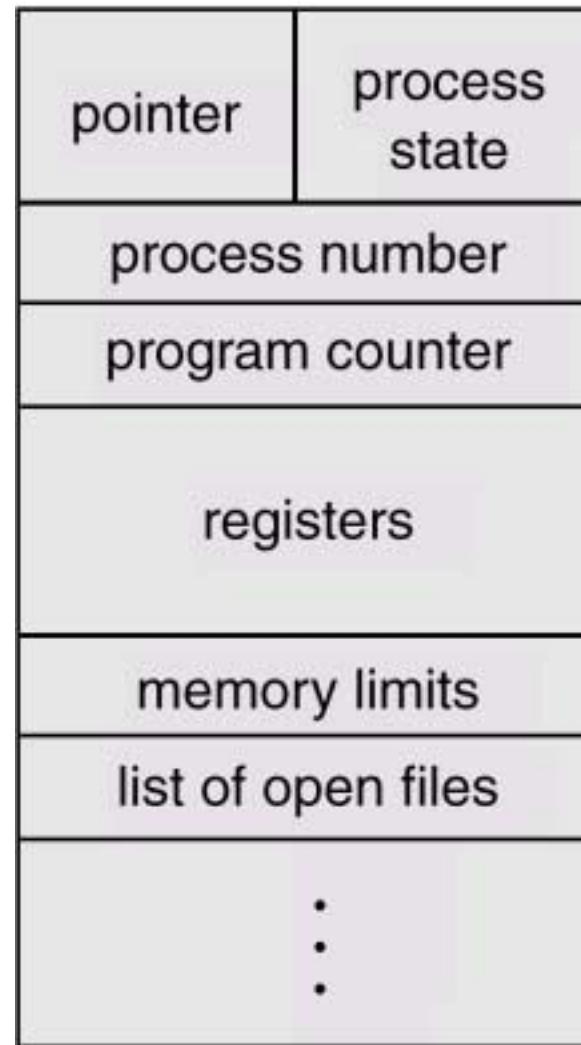
Exception Conditions – Error Recovery

- Process terminates
- Lost messages
- Scrambled Messages

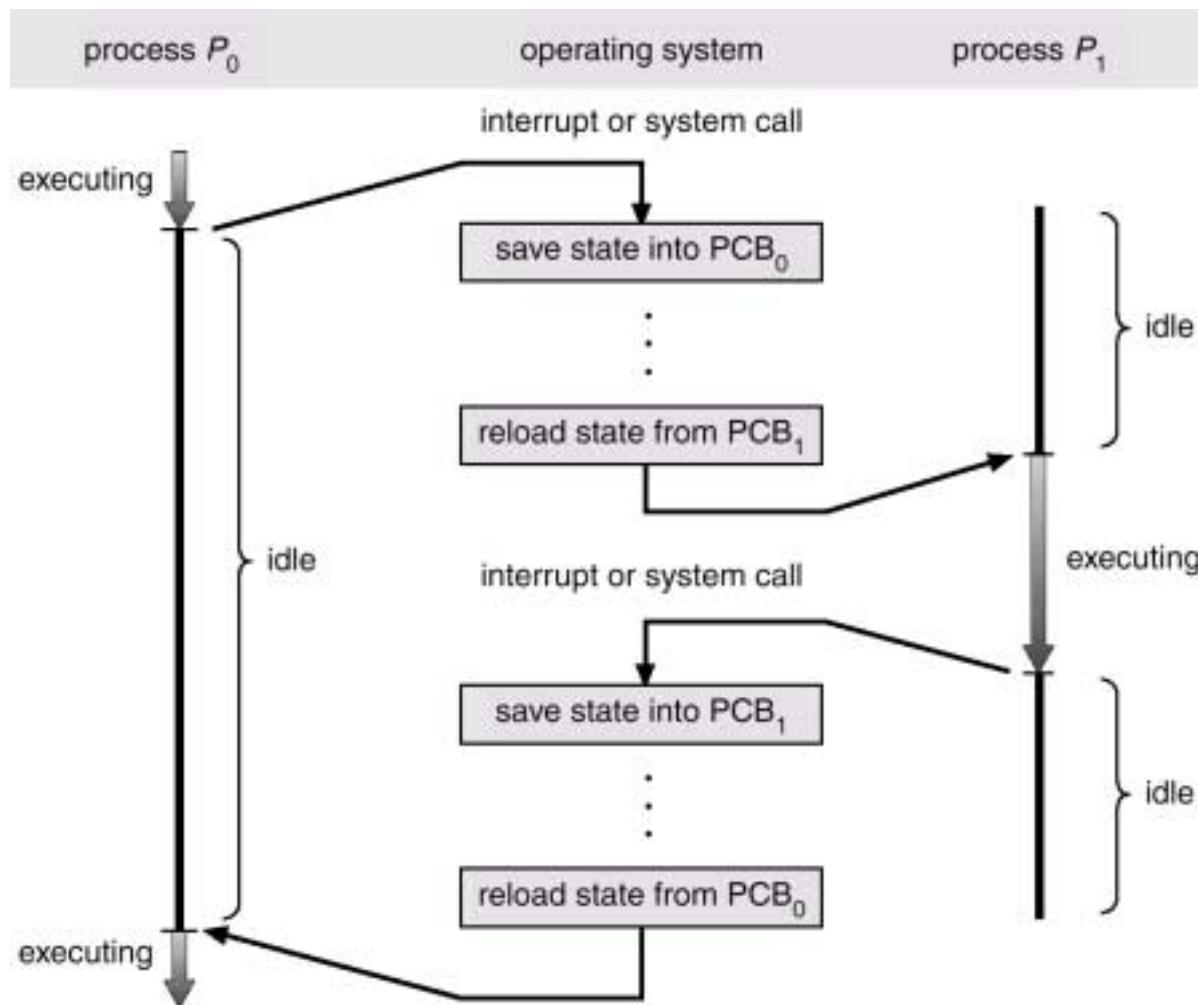
4.01



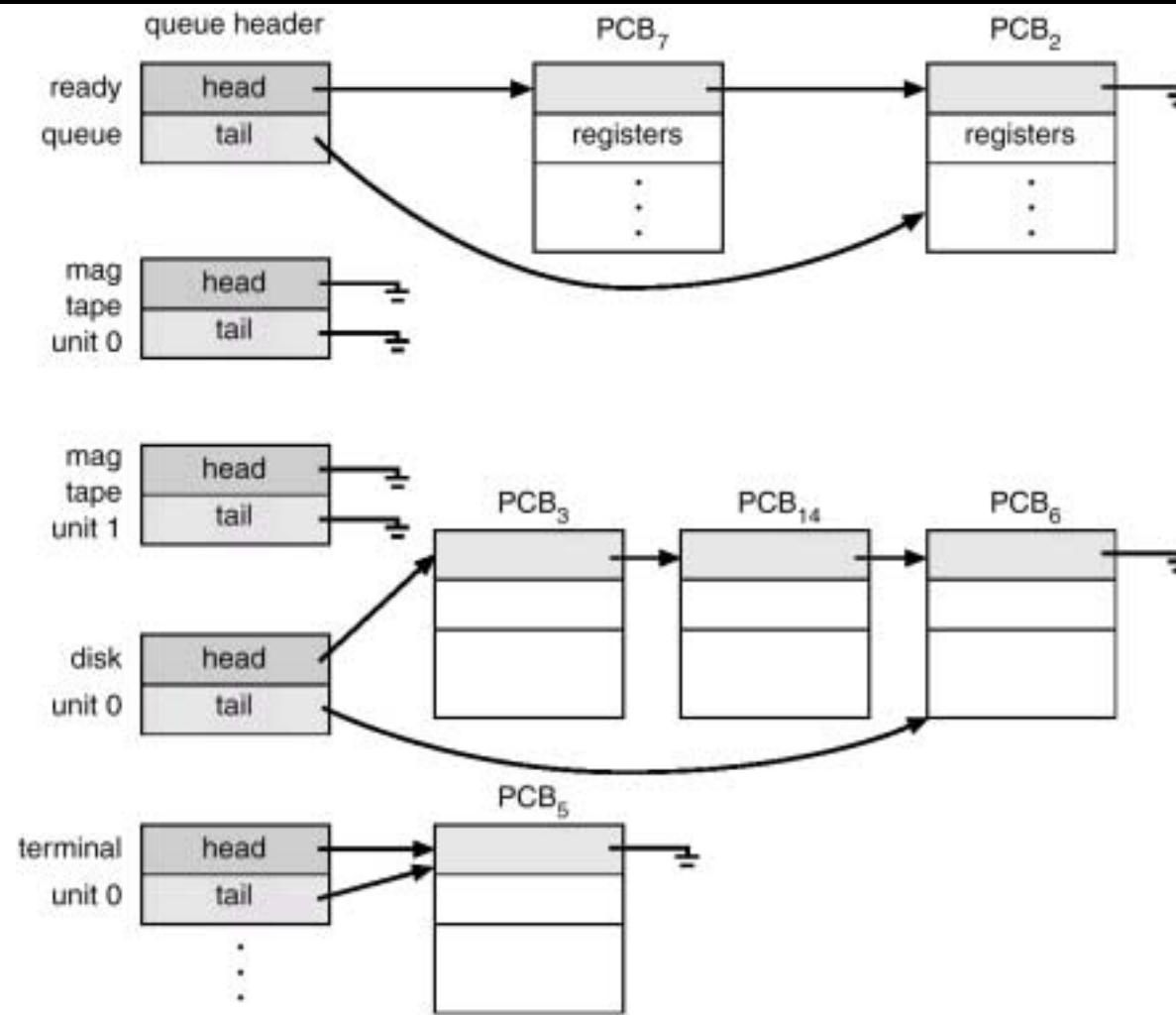
4.02



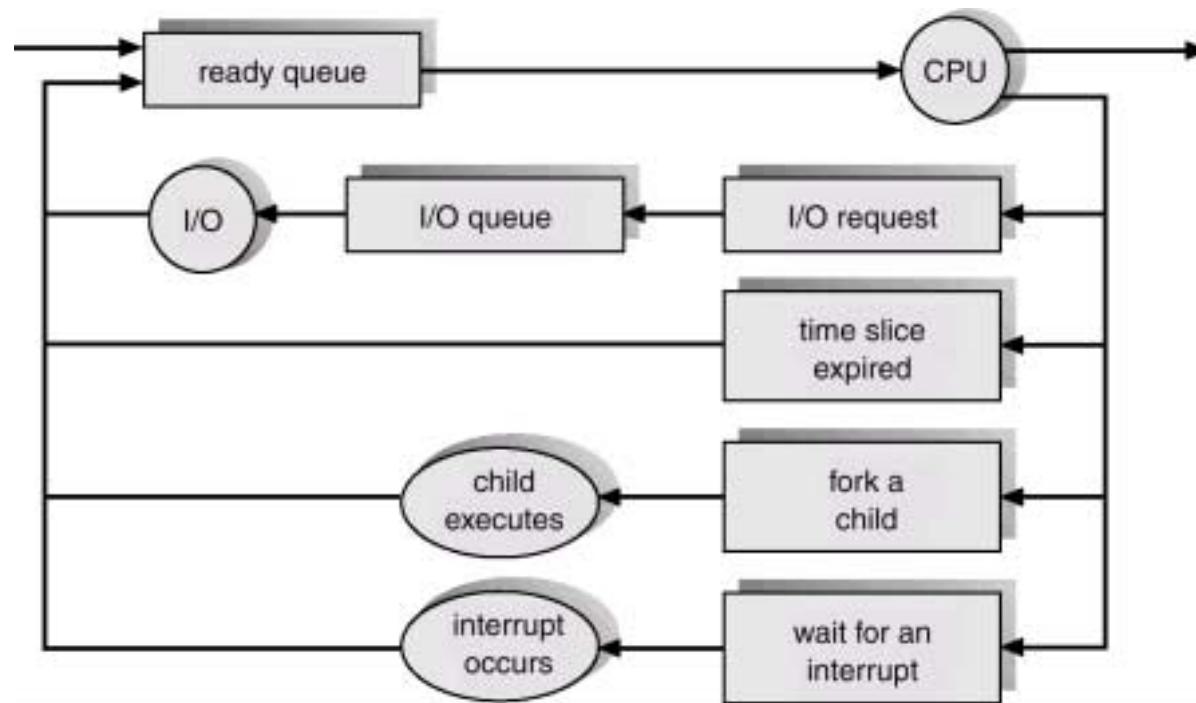
4.03



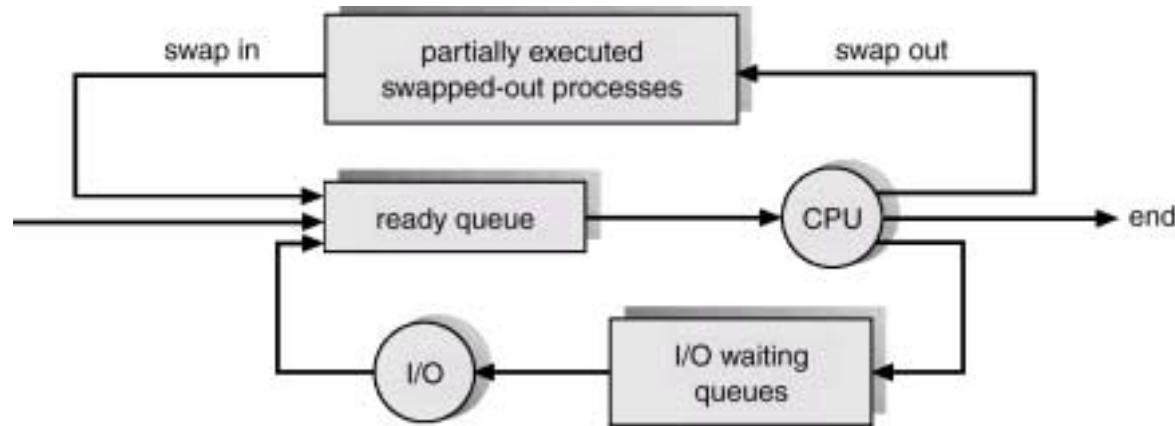
4.04



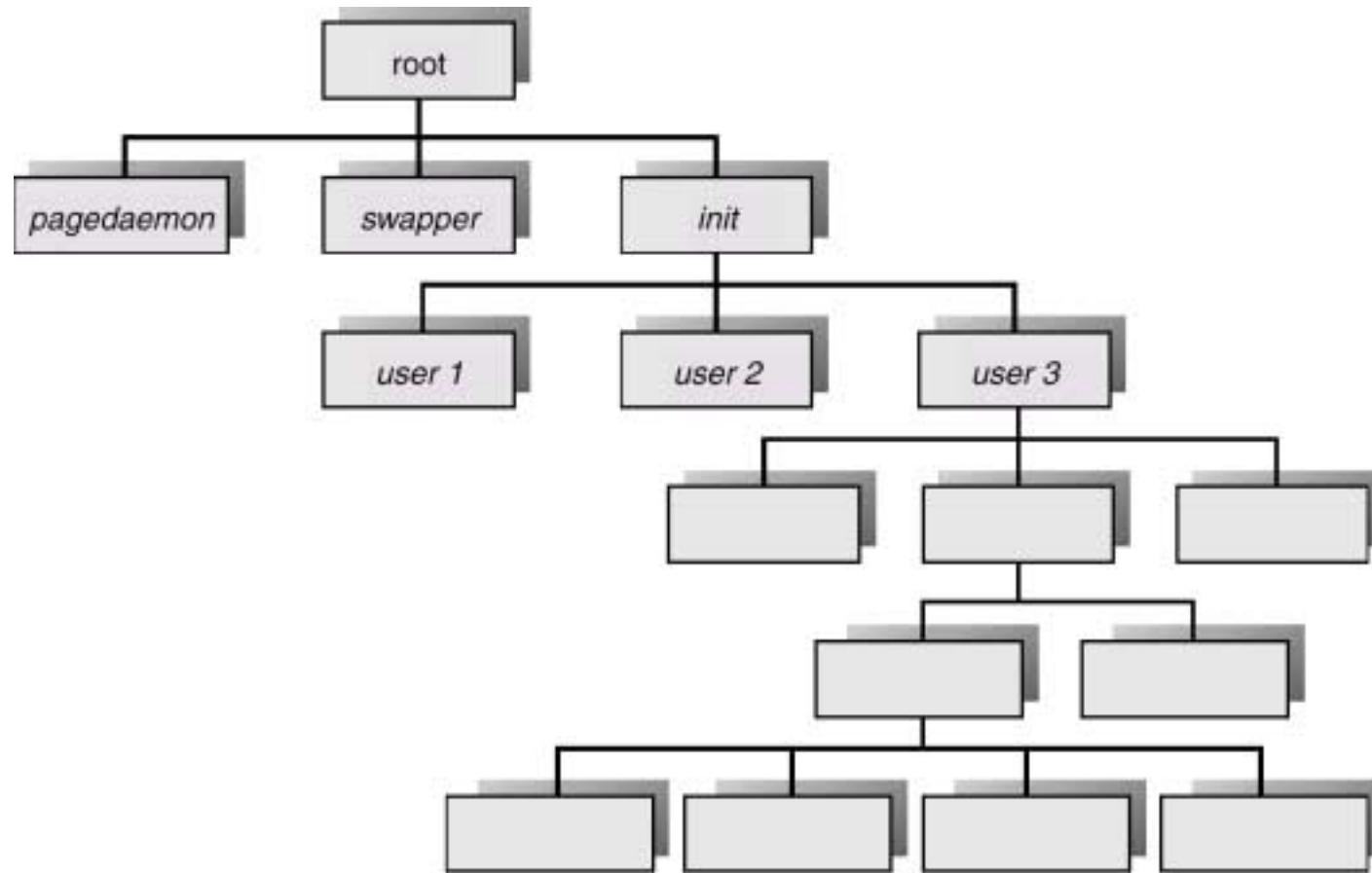
4.05



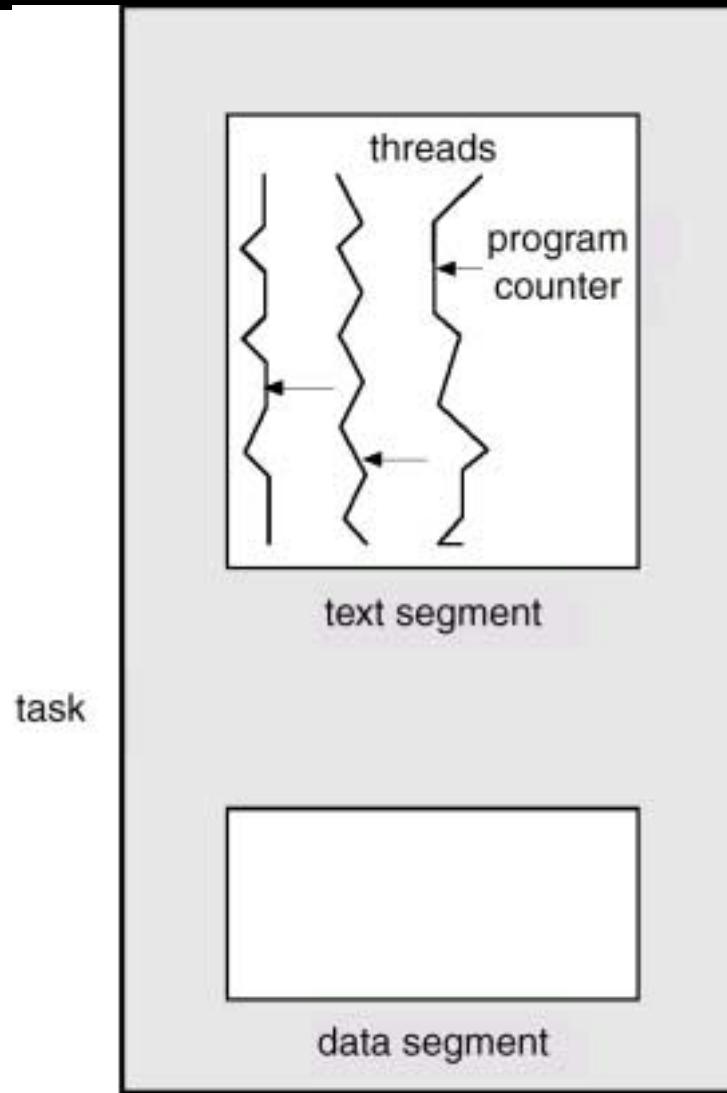
4.06



4.07



4.08



4.09

