Operating Systems

CMP-320

Instructor: Muhammad Adeel Nisar

Lecture-Processes

Acknowledgement

Some slides and pictures are adapted from Lecture slides / Books of

- Dr. Syed Mansoor Sarwar.
- Mr. M. Arif Butt
- Text Book - OS Concepts by Silberschatz, Galvin, and Gagne.

Today's Agenda

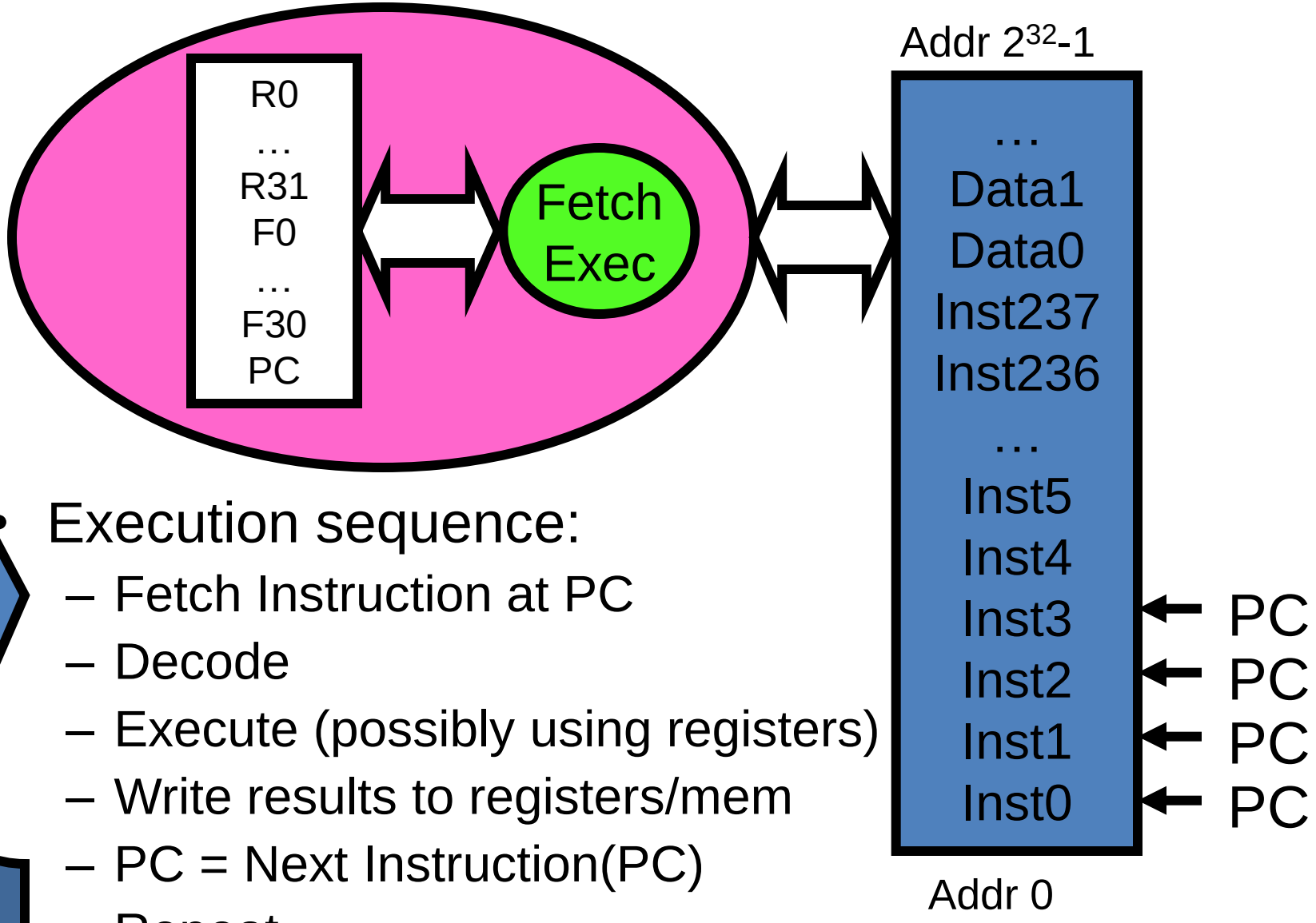
- Review of previous lecture
- Process Concept
- Process States
- Process Scheduling Concepts
- Process Creation and Termination

What is a Process

Process – a program in execution;
process execution must progress in sequential fashion.

The UNIX system creates a process every time you run an **external command**, and the process is removed from the system when the command finishes its execution.

What happens during Program execution?

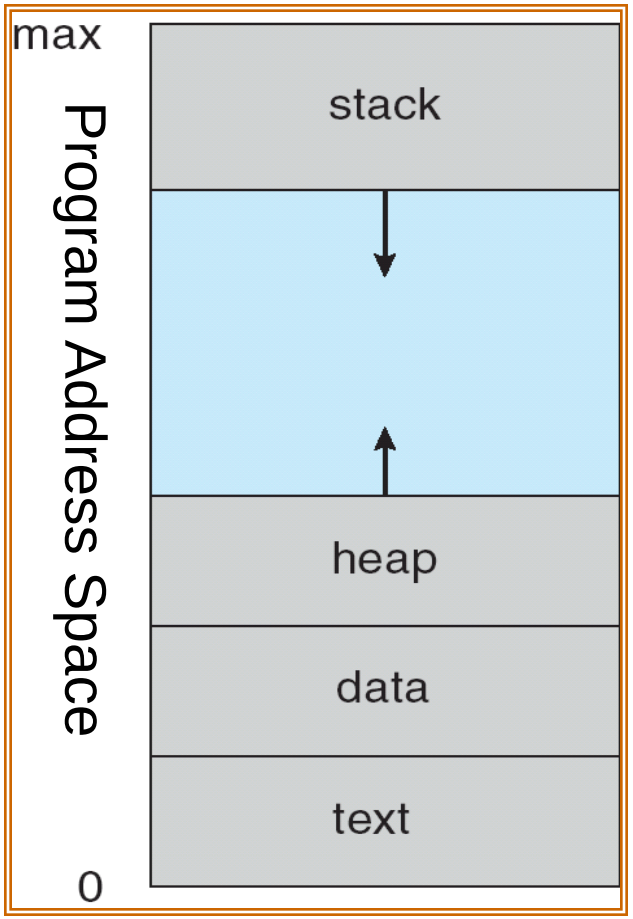


- Execution sequence:
  - Fetch Instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

Components of a Process

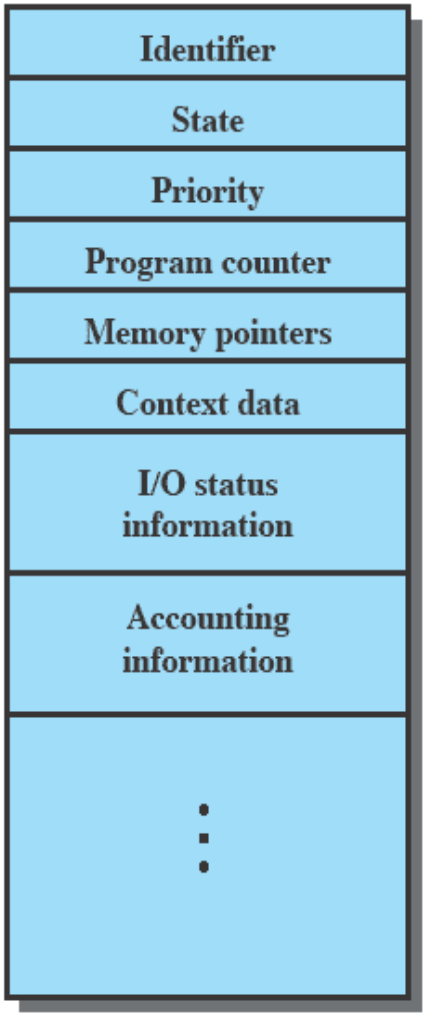
1. **Code/Text section** (does not change ideally)
2. **Data Section**
3. **Heap** (Used to allow a process to request for allocation/ de-allocation of memory at run time)
4. **Stack** (Used in function calls to keep parameters, local variables, return addresses)
5. **Environment** (Command line parameters passed; e.g in C, argc and argv)
6. **CPU state** (Program counter and other registers)
7. **Process Control Block** (A kernel data structure allocated to every process)

A Process Pictorially



Process Control Block

- PCB is the most important and central data structure in an OS maintained by kernel.
- A PCB contains information associated with a specific process that is used by OS to control the process.

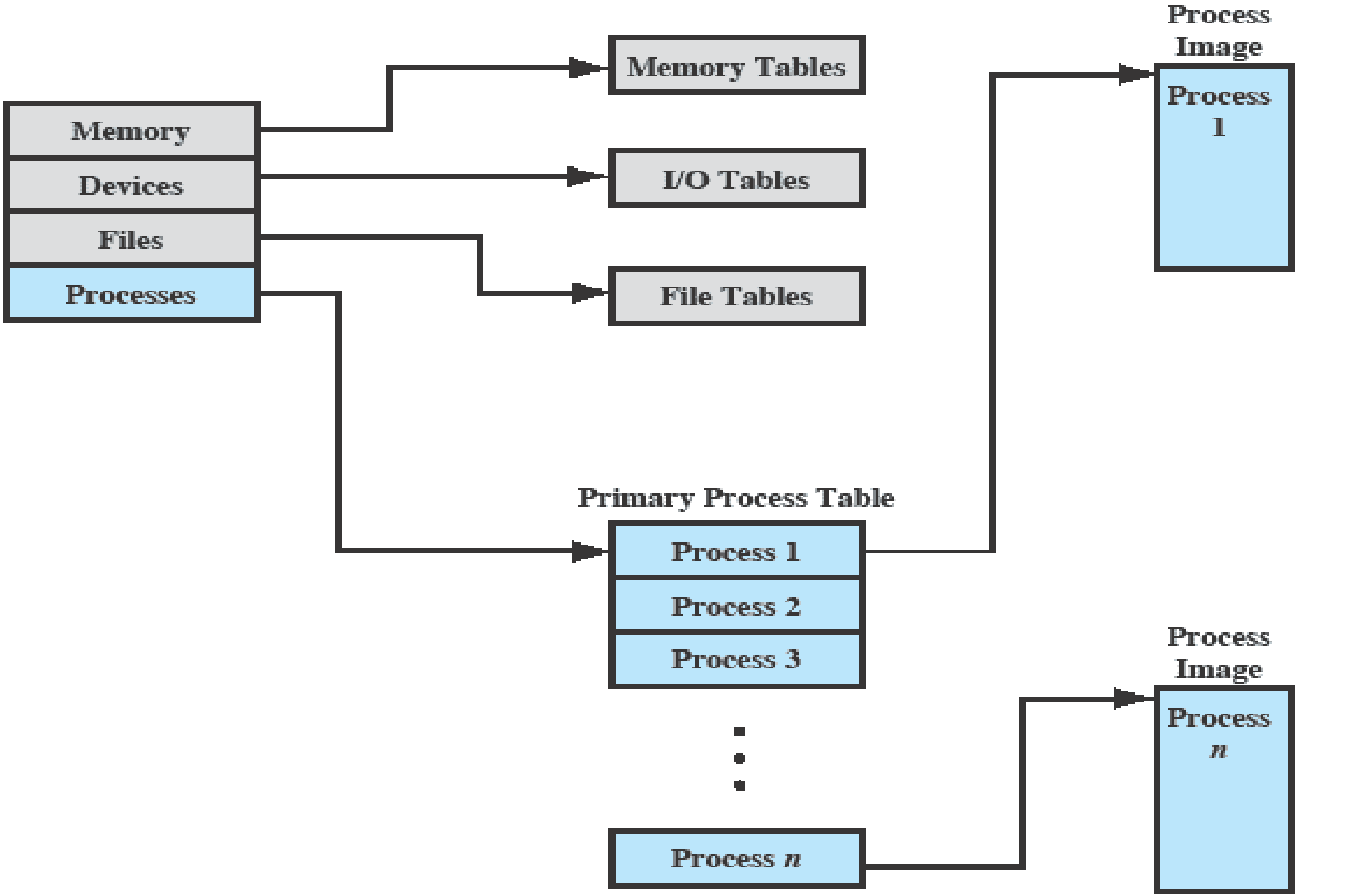


Process Control Block

Information in a PCB can be divided into three general categories:

1. **Process identification**
   - PID, Parent ID, User ID
2. **Processor state information.** This consists of the contents of processor registers.
   - User-visible registers
   - Control and status registers (PC and Condition codes e.g., sign, zero, carry, equal, overflow, and Status information e.g., Interrupt enable/disable, execution mode)
   - Stack pointers
3. **Process control information**
   - Scheduling and State information
   - Data structuring
   - Inter process communication
   - Process privileges
   - Memory management
   - Resource ownership and utilization

Operating System Control Structures



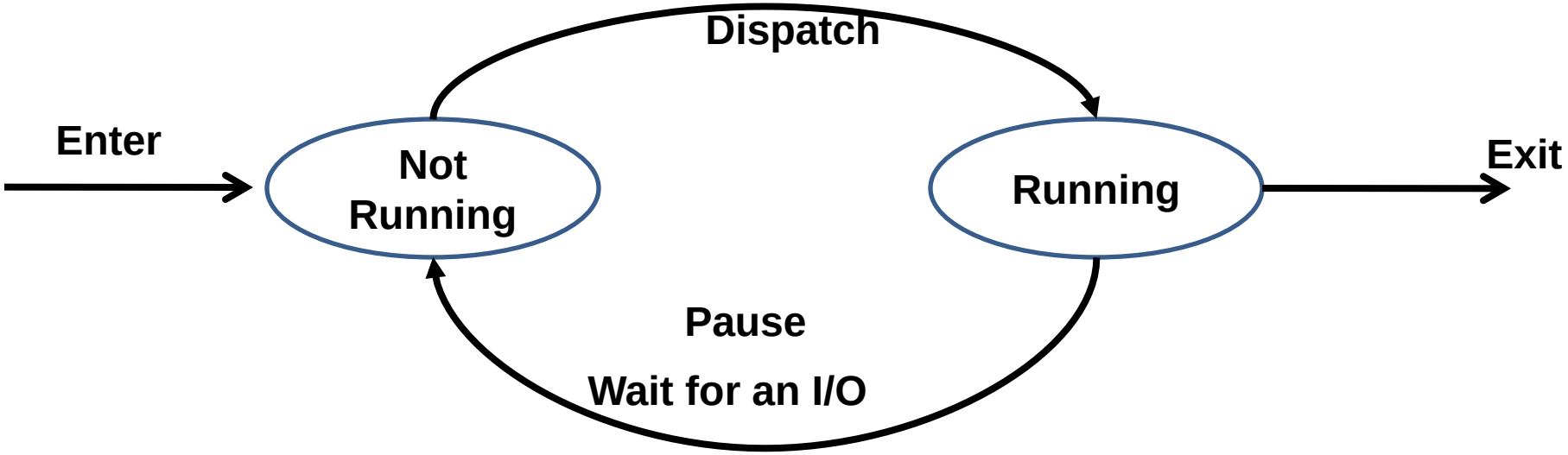
CPU Bound and I/O Bound Processes

- **I/O-*bound process*** – spends more time doing I/O than computations; Many short CPU bursts
- Examples: Word processing, text editors. Billing system of Wapda which involves lot of printing

- ***CPU-bound process*** – spends more time doing computations; Few very long CPU bursts
- Examples: Simulation of NW traffic involving lot of mathematical calculation, scientific applications involving matrix multiplication, DSP applications

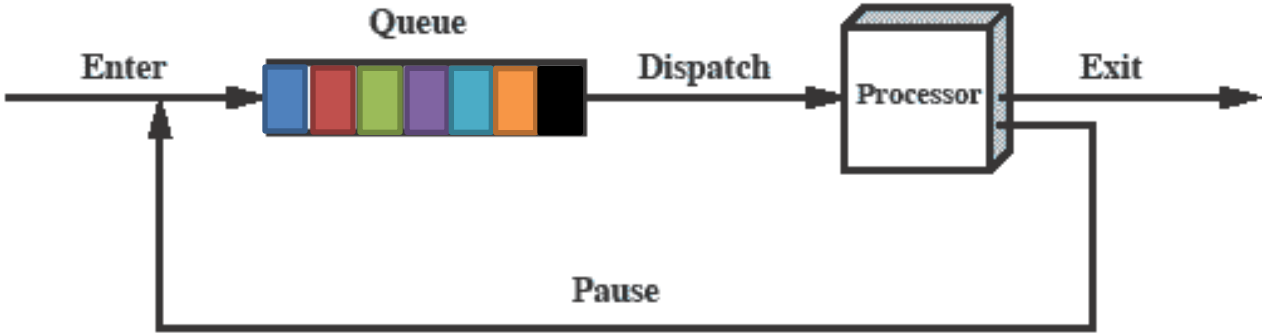
2-State Process Model

- Broadly speaking life of a process consists of CPU bursts and I/O bursts. So simplest possible model can be constructed by observing that at any particular time, a process is either being executed by a processor or is not running or waiting for an I/O
- There may be a number of processes in the “not running” state but only one process will be in “running” state



Queuing Diagram (2-State Process Model)

- Queuing structure for a two state process model is:



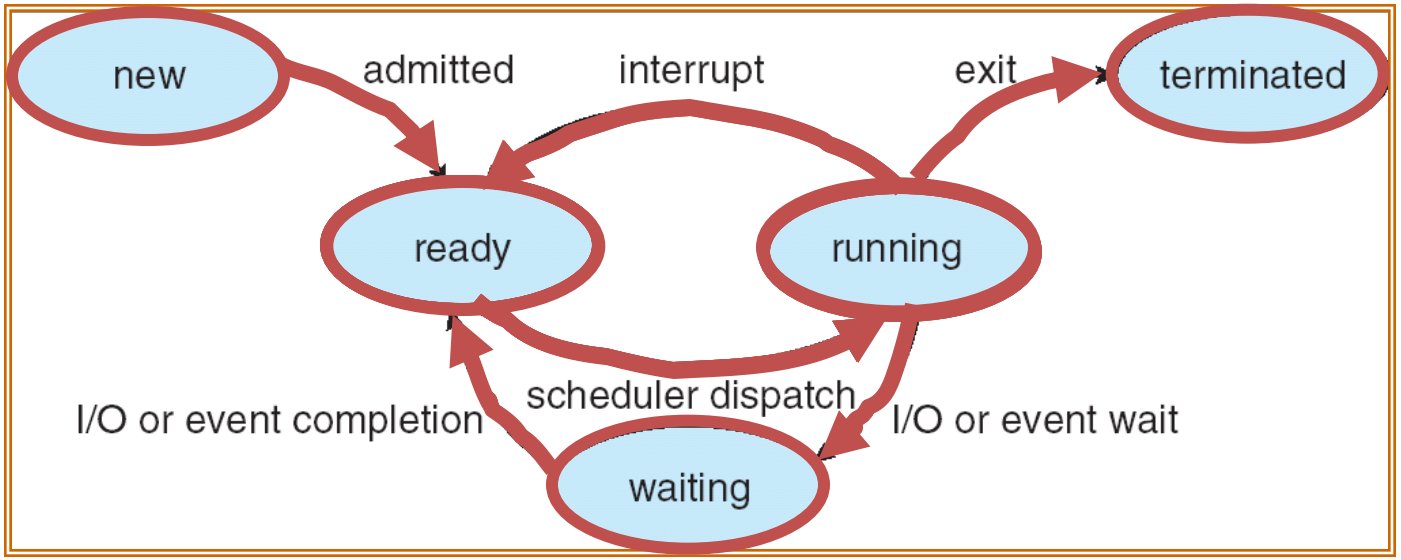
Limitations

- If all processes in the queue are always ready to execute only then the above queuing discipline will work
- But it may also be possible that some processes in the queue are ready to execute, while some are waiting for an I/O operation to complete
- So the **dispatcher** has to scan the list of processes looking for the process that is not blocked and is there in the queue for the longest

5-State Process Model

- Broadly speaking the life of a process consist of CPU burst and I/O burst but in reality
  - A process may be waiting for an event to occur; e.g. a process has created a child process and is waiting for it to return the result
  - A process may be waiting for a resource which is not available at this time
  - Process has gone to sleep for some time
- So generally speaking a Process may be in one of the following five states:
  - **new**: The process is being created ()
  - **ready**: The process is in main memory waiting to be assigned to a processor
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur (I/O completion or reception of a signal)
  - **terminated**: The process has finished execution

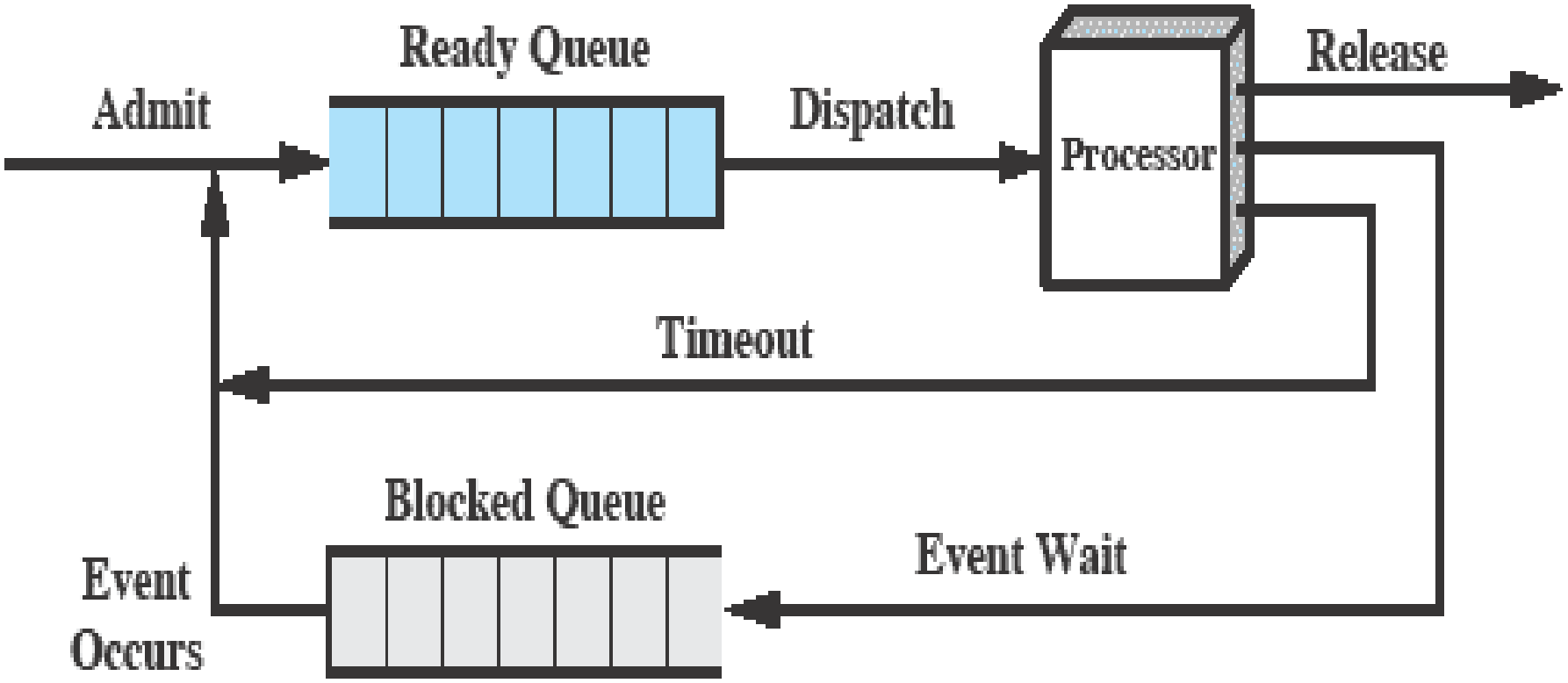
5-State Process Model



As a process executes, it changes state:

- new: The process is being created
- ready: The process is waiting to run
- running: Instructions are being executed
- waiting: Process waiting for some event to occur
- terminated: The process has finished

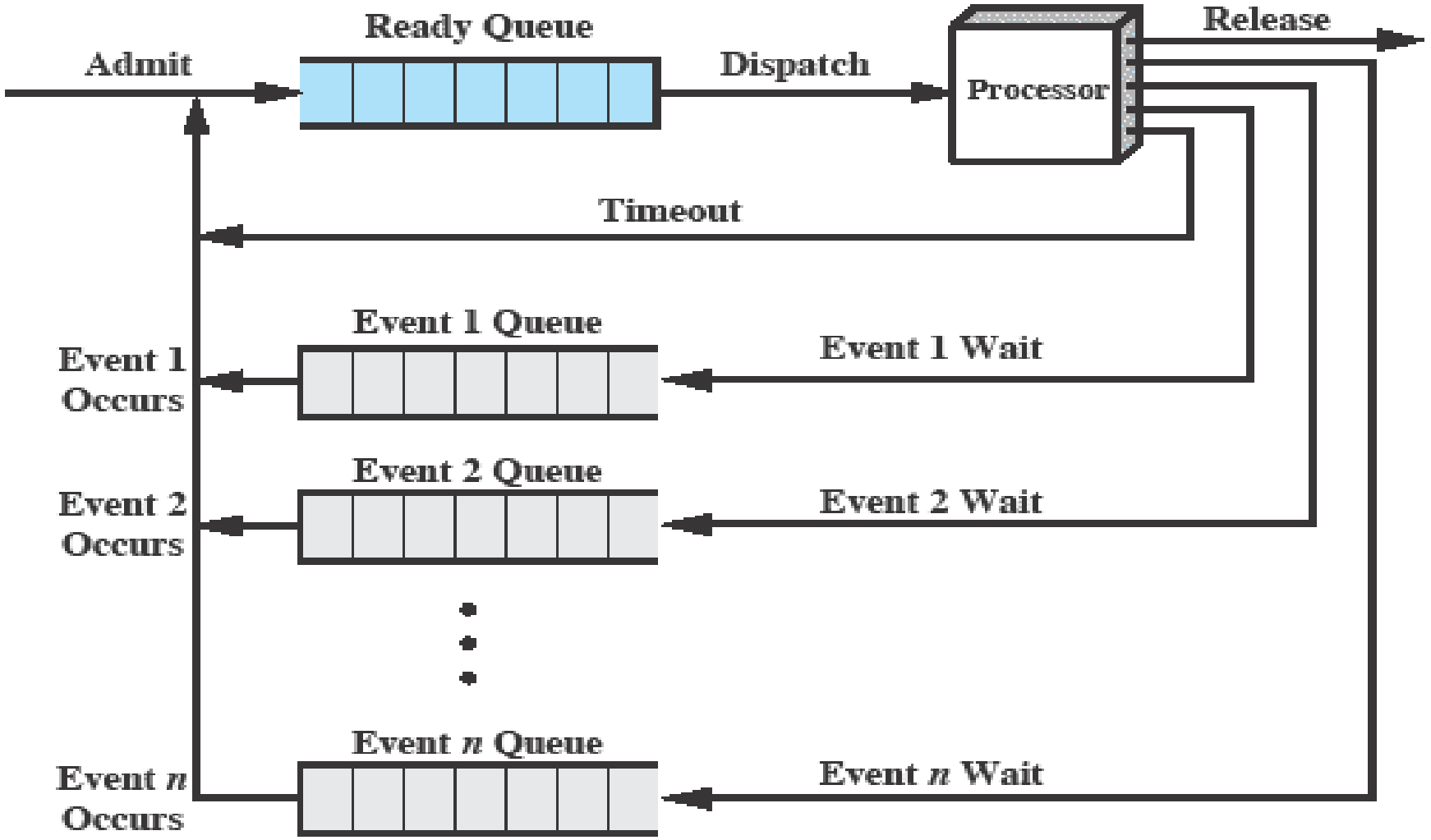
Queuing Diagram (Using 2-Queues



Limitation: When an event occurs the dispatcher would have to cycle through the entire Blocked Queue to see which process is waiting for that event.

This can cause huge overhead when there may be 100's or 1000's of processes

Queuing Diagram (Using Multiple Queues

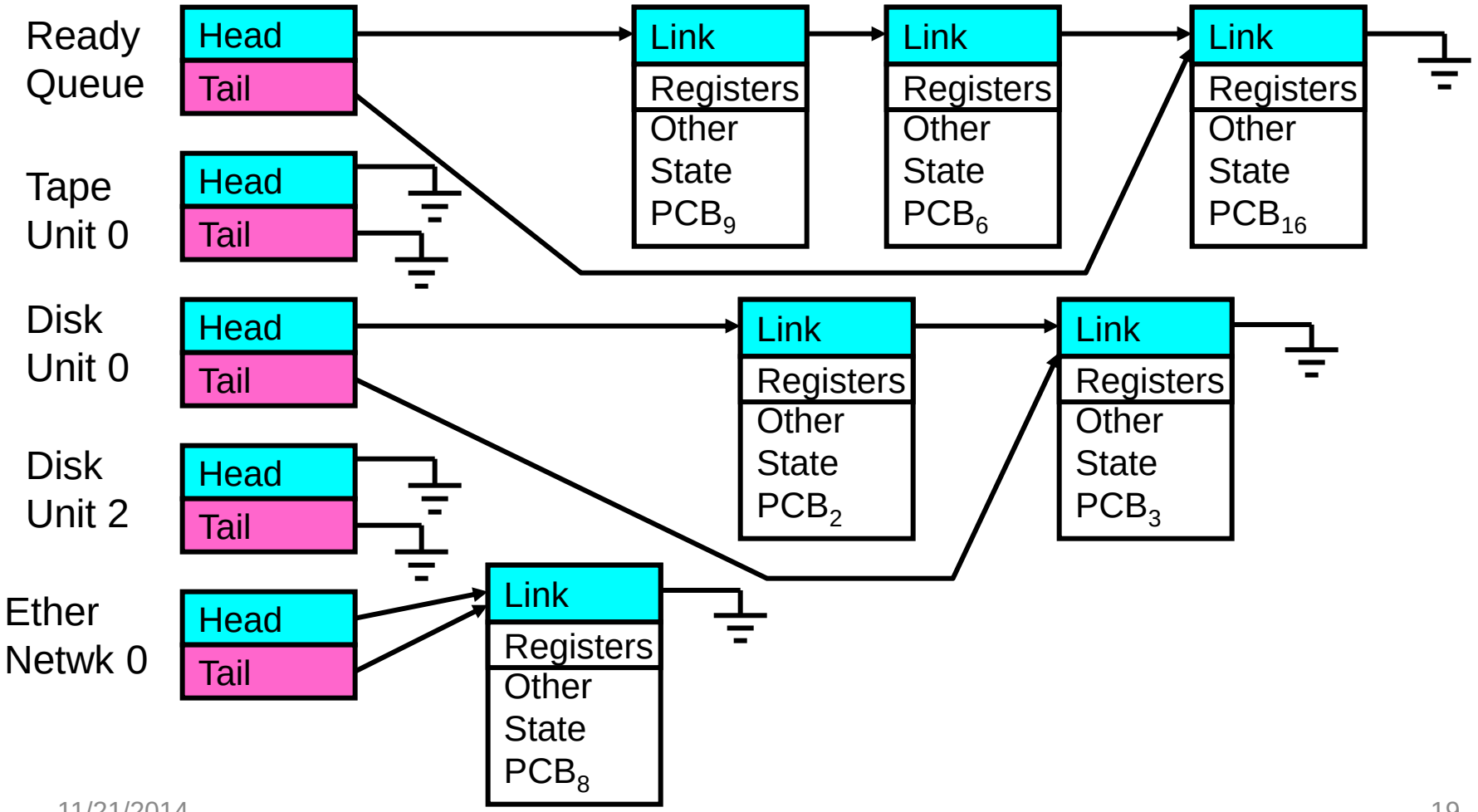


Process Scheduling Queues

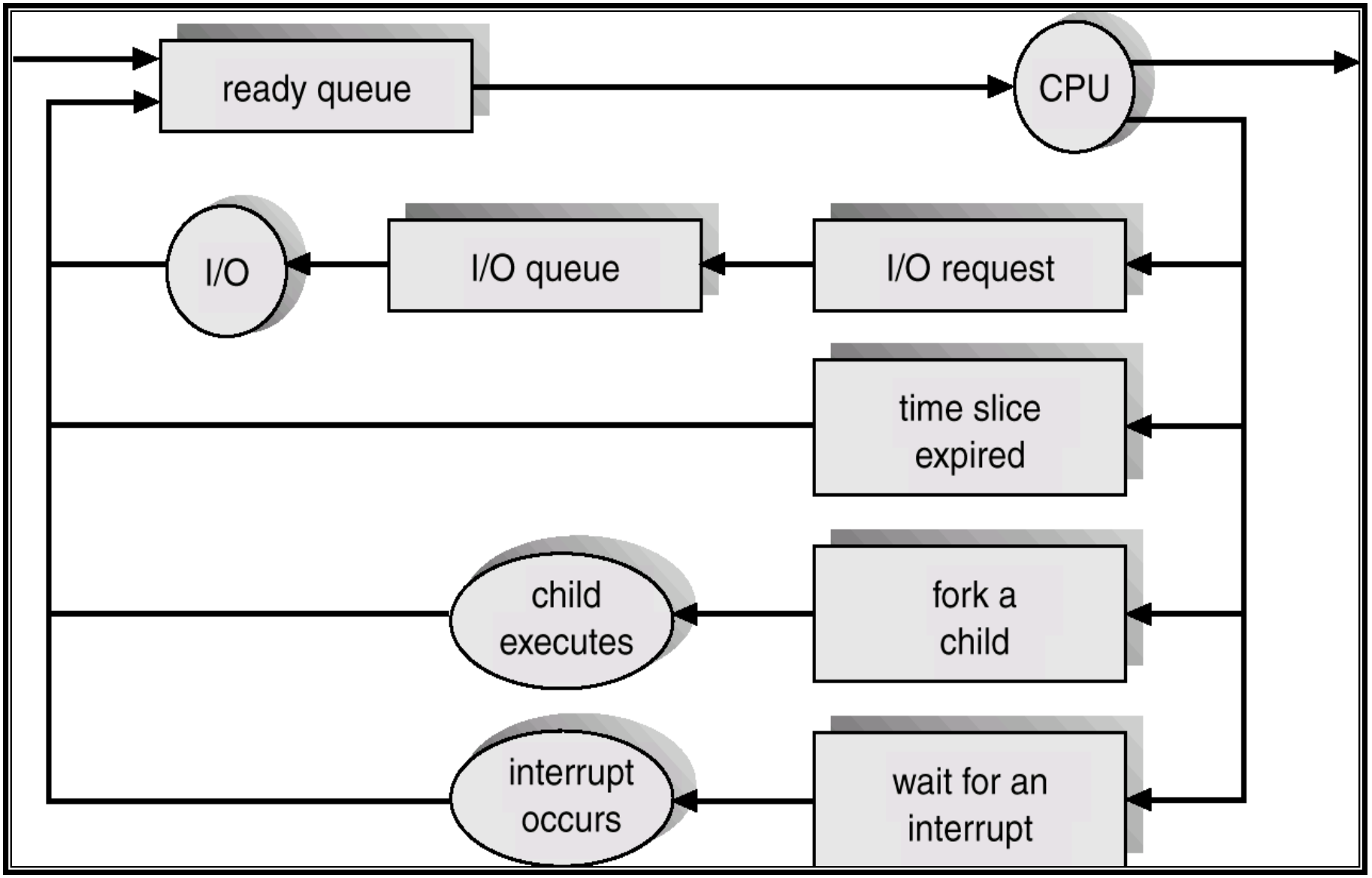
- **Job Queue –** When a process enters the system it is put into a job Queue. This queue consists of all processes in the system
- **Ready Queue –** This queue consists of processes that are residing in main memory and are ready and waiting to execute. It is generally stored as a link list
- **Device Queues –** When the process is allocated the CPU, it executes for a while and eventually quits as it may need an I/O. the list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue
- A process in its life time will be migrating from one **Q** to another **Q**

Ready Queue And Various I/O Device Queues

- Process not running ⇒ PCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have a different scheduler policy



Queuing Diagram



Schedulers

- A process migrates between various scheduling queues throughout its life cycle. OS must select processes from these queues in some predefined fashion. This selection is done by an appropriate scheduler.

“Scheduling is a matter of managing queues to minimize queuing delay and to optimize performance in a queuing environment”.

Long Term Scheduler

- Long-term scheduler (or job scheduler) – selects processes from the job pool (processes spooled on hard disk) to be brought into the ready queue (inside main memory)
- Long-term scheduler is invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
- Must select a good mix of I/O bound and CPU bound processes
  - **If all processes selected by LTS are I/O bound, then Ready Queue will almost always be empty**
  - **If all processes selected by LTS are CPU bound, then I/O waiting queue will almost always be empty**
- The long-term scheduler controls the degree of multiprogramming. More processes, smaller percentage of time each process is executed

Short Term Scheduler

- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates it the CPU through the **dispatcher**
- The OS code that implements the CPU scheduling algorithm is known as CPU scheduler
- Short-term scheduler is invoked very frequently (milliseconds) $\Rightarrow$ (must be fast)
- Invoked when following events occur
    - CPU slice of the current process finishes
    - Current process needs to wait for an event
    - I/O interrupt
    - System call
    - Signal

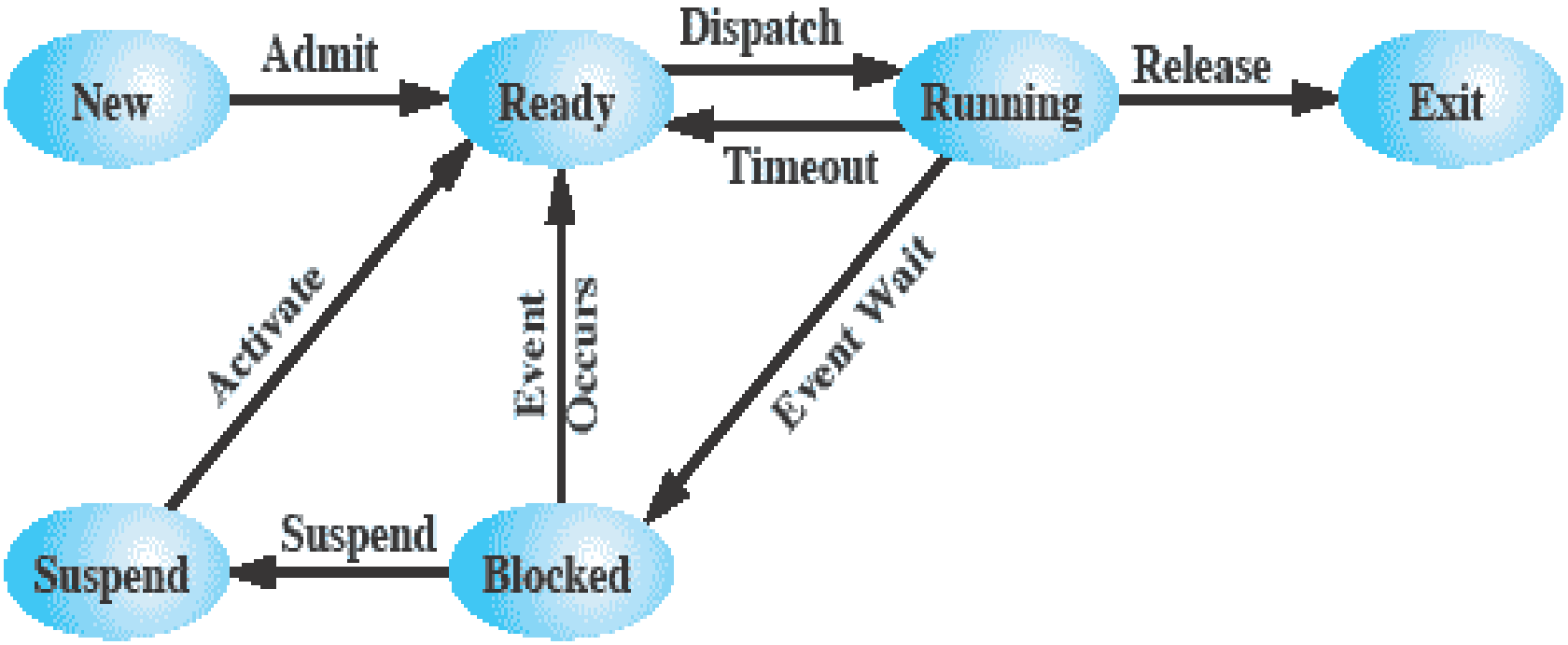
Medium Term Scheduler

- The CPU is so much faster than I/O that it will be common for all of the processes in memory to be waiting for I/O. Thus even with multiprogramming the CPU could be idle most of the time.
- Solution is "**SWAPPING**", which involves moving part or all of the process from main memory to disk
- When all the processes in main memory are in Block state (and some other processes wants to come in, but we don't have enough memory), the OS can suspend one process by putting it in the suspended states and transferring it to disk. This is called a swap-out operation. Now the main memory has space to bring in a new process (swap-in)

Medium Term Scheduler (cont…)

- Also known as **swapper**
- Selects an in-memory process and swaps it out to the disk temporarily
- Swapping decision is based on several factors
  - Arrival of a higher priority process but no memory available
  - To improve process mix.
  - Requiring memory to be freed up because memory requirement of a process cannot be met.
- Blocked state becomes ***suspend*** state when swapped to disk

6-State Process Model (one suspended state)

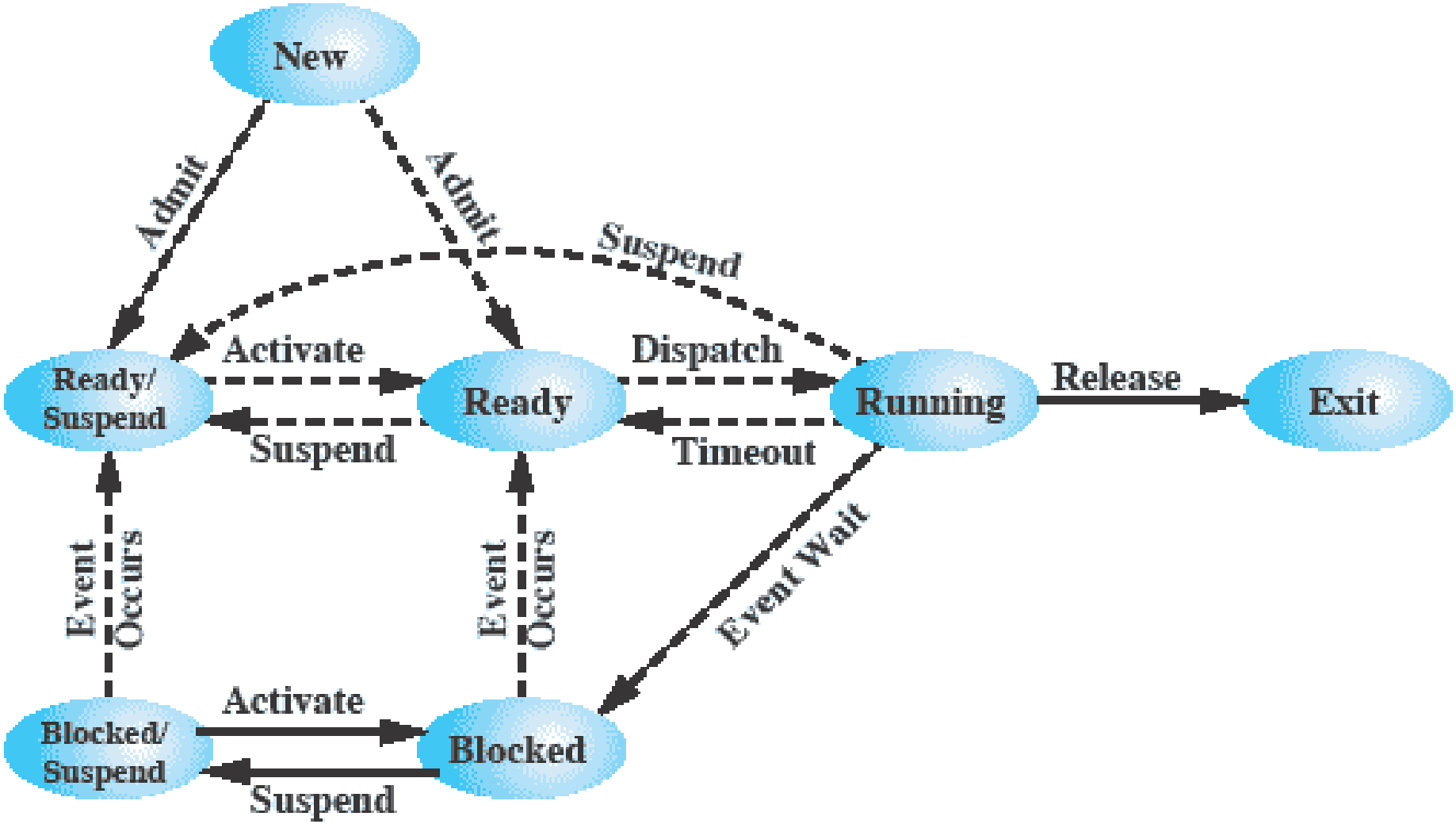


Limitation: This only allows processes which are blocked to be swapped out.

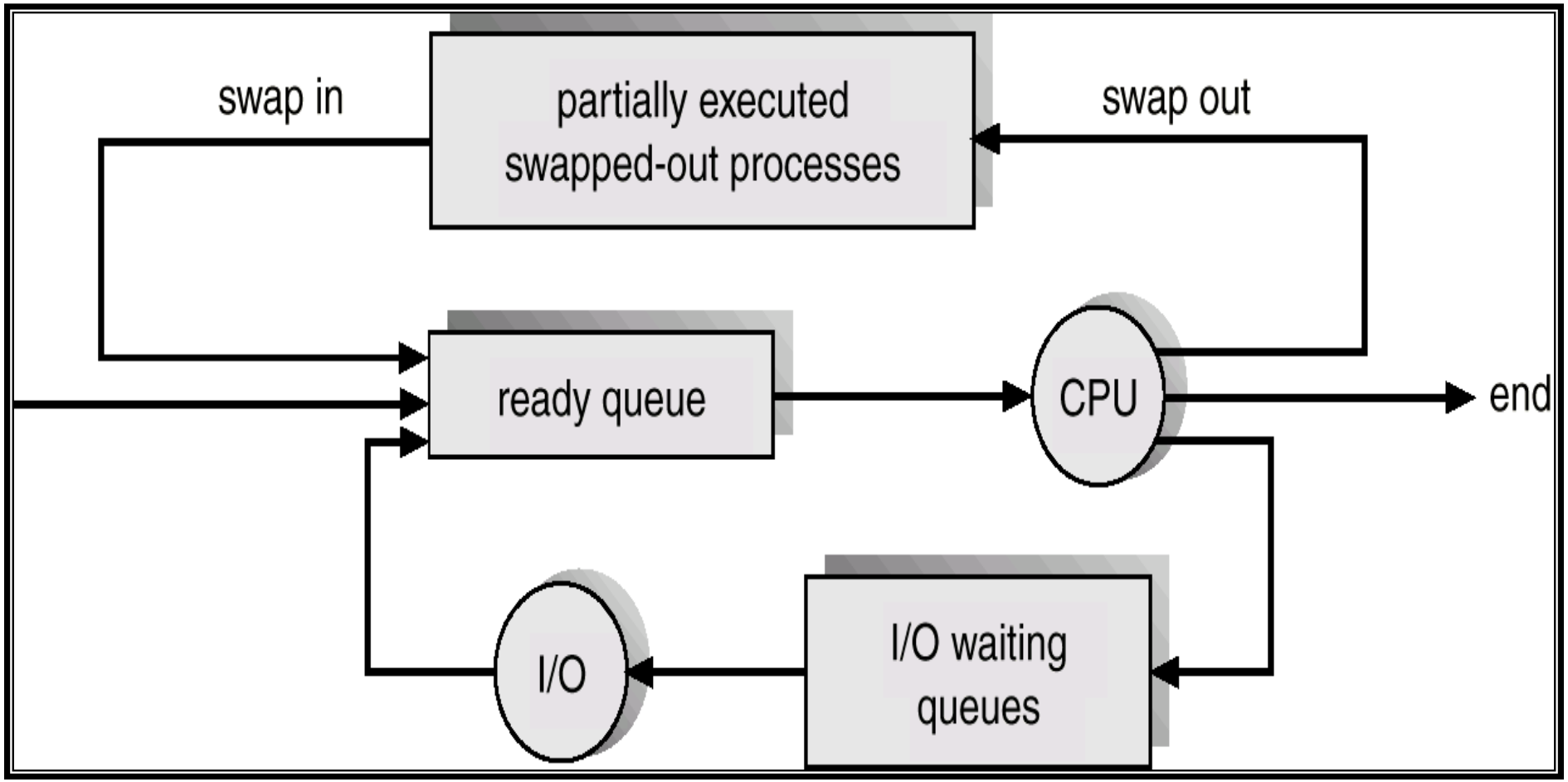
What if there is no blocked process but we still need to free up memory?

Solution: Add another state Ready Suspended and swap out a process from the ready state

7-State Process Model (two suspended state)



Queuing Diagram (cont…)



Context/Process Switch

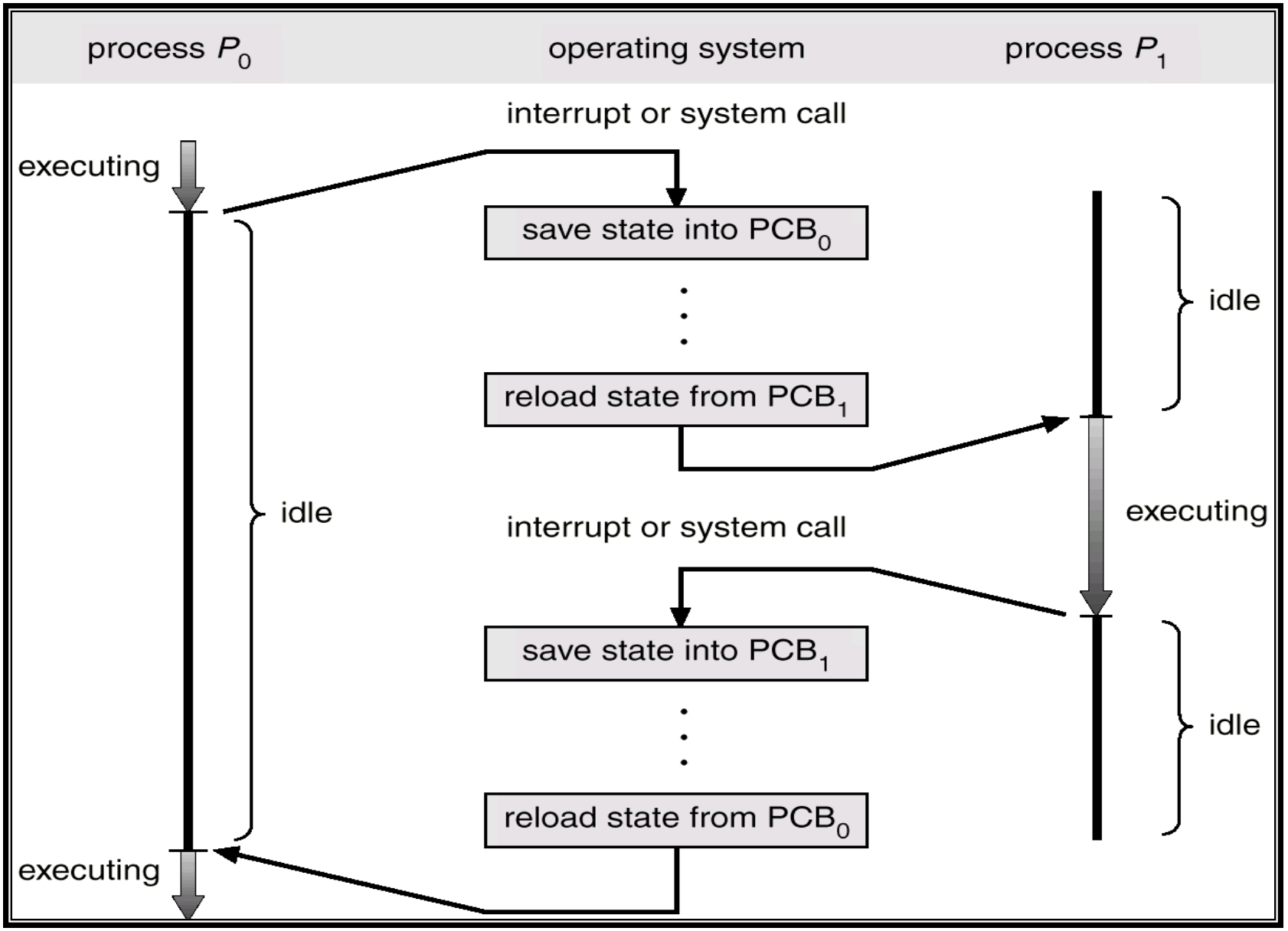
- When CPU switches to another process, the system must save the state (context) of the ‘current’ (old) process and load the saved state for the new process.
- Context-switch time is overhead; the system does no useful work while switching.
- Scheduler has nothing to do with a process switch. It is mainly the job of **Dispatcher**. The time it takes for a dispatcher to stop one process and start another is known as dispatch latency.
- Interrupt, Trap and signal triggers a process switch.

Context Switch (cont…)

Steps involved in a full Process switch are:

- Save context of currently running process (including PC and other registers)
- Move this PCB to an appropriate Queue
- Select another process for execution (Kernel Schedules)
- Update PCB of selected process
- Update memory management data structures
- Restore the context of the process

Context Switch / Process Switch



Process Creation

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes
- Process creation and termination are the only mechanisms used by the UNIX system to execute external commands
- Once the OS decides to create a process it proceeds as follows:
  - Assigns a unique PID to the new process.
  - Allocate space
  - Initialize the PCB for that process.
  - Set appropriate linkages
  - Create or expand other data structures

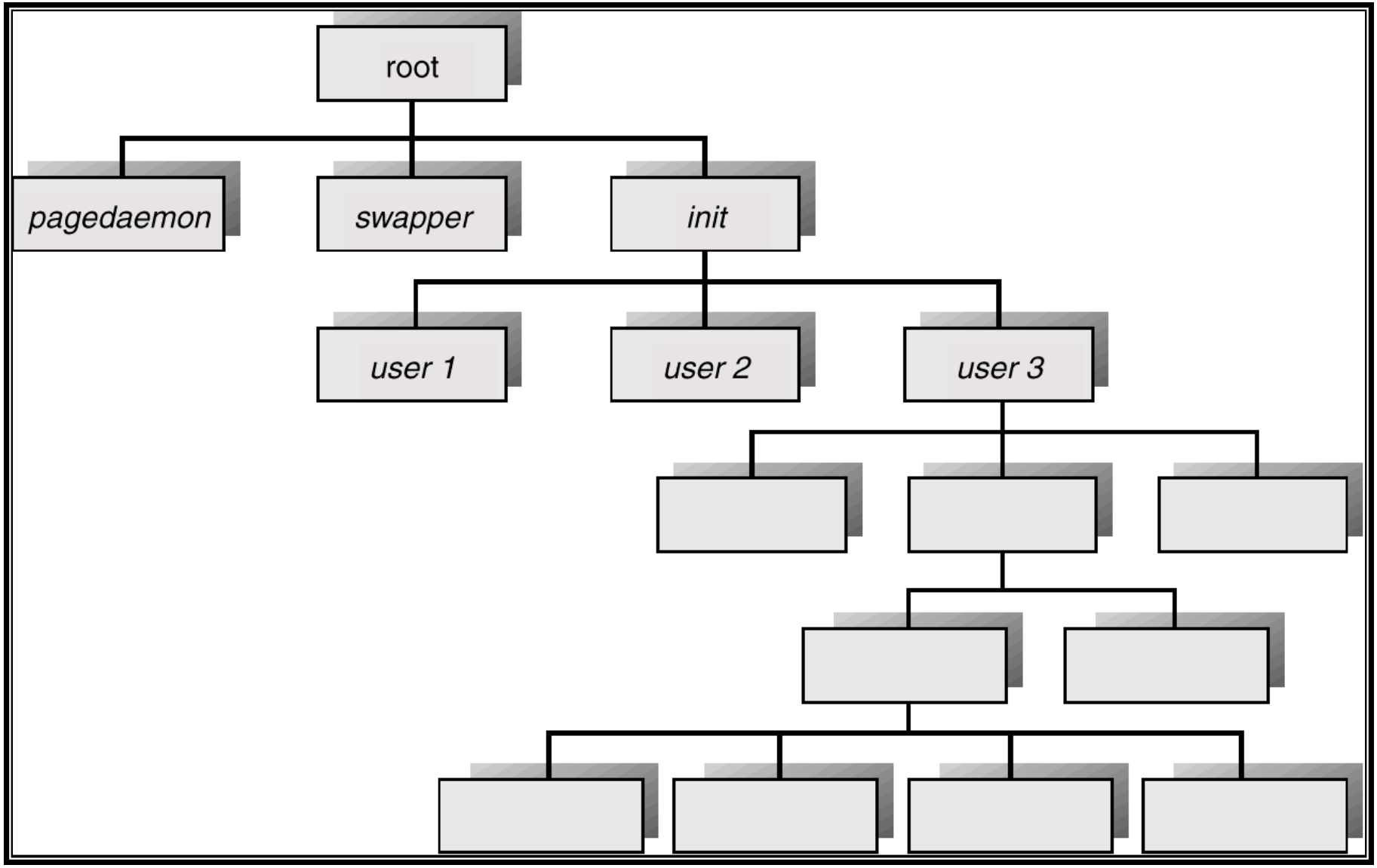
Process Creation (cont…)

- **Resource sharing**
  - Parent and children share all resources
  - Children share a subset of parent's resources (UNIX)
  - Parent and child share no resources
- **Execution**
  - Parent and children execute concurrently
  - Parent waits until children terminate
- **Address Space**
  - Child duplicate of parent process
  - Child has a program loaded onto it

Process Creation (cont…)

- **UNIX examples**
  - **fork()** system call creates a new process
  - **exec()** system call used after a fork to replace the process memory image with a new executable
- **Reasons for Process Creation**
  - In batch environment a process is created in response to the submission of a job
  - In interactive environment a process is created when a new user attempts to log on
  - OS can create a process to perform a function on behalf of a user program. (e.g. printing)
  - Spawning When a process is created by OS at the explicit request of another process.

Processes Tree on a UNIX System



Process Termination

- A process terminates when it finishes executing its last statement and requests the operating system to terminate it using the **exit()** system call
- At this point the process returns data to its parent process
- Process resources are de-allocated by the OS, to be recycled later

Process Termination (cont…)

- A **parent** may terminate execution of one of its children for a variety of reasons such as:
  - Child has exceeded allocated resources (main memory, execution time, etc.)
  - Parent needs to create another child but has reached its maximum children limit
  - Task performed by the child is no longer required
  - Parent exits
    - OS does not allow child to continue if its parent terminates
    - Cascaded termination

Process Termination (cont…)

- A **process** may terminate due to following reasons:
  - Normal completion
  - Memory unavailable
  - Protection error
  - Mathematical error
  - I/O failure
  - Cascading termination (by OS)
  - Operator intervention

Process Related UNIX Commands

Command	Description
# **ps** [-aelu]	Display status of processes
# **top**	Display information about top 20 processes
# **vim** file1.txt **&**	Running a command in back ground
CTRL + Z	Suspend a foreground process to go back to the shell
# **jobs**	Display status of suspended and background processes
# **bg** %job_id	Put a process in background
# **fg** %job_id	Move background process to the foreground
CTRL + C	Kill foreground process
# **kill** [-signal_no] proc_list	Send the signal for signal_no to processes whose PIDs or jobIDs are specified in proc_list. jobIDs must start with %.

Process Management in UNIX/Linux

- Important process-related UNIX/Linux system calls
  - fork()
  - exit()
  - wait()
  - exec()

System Call - fork()

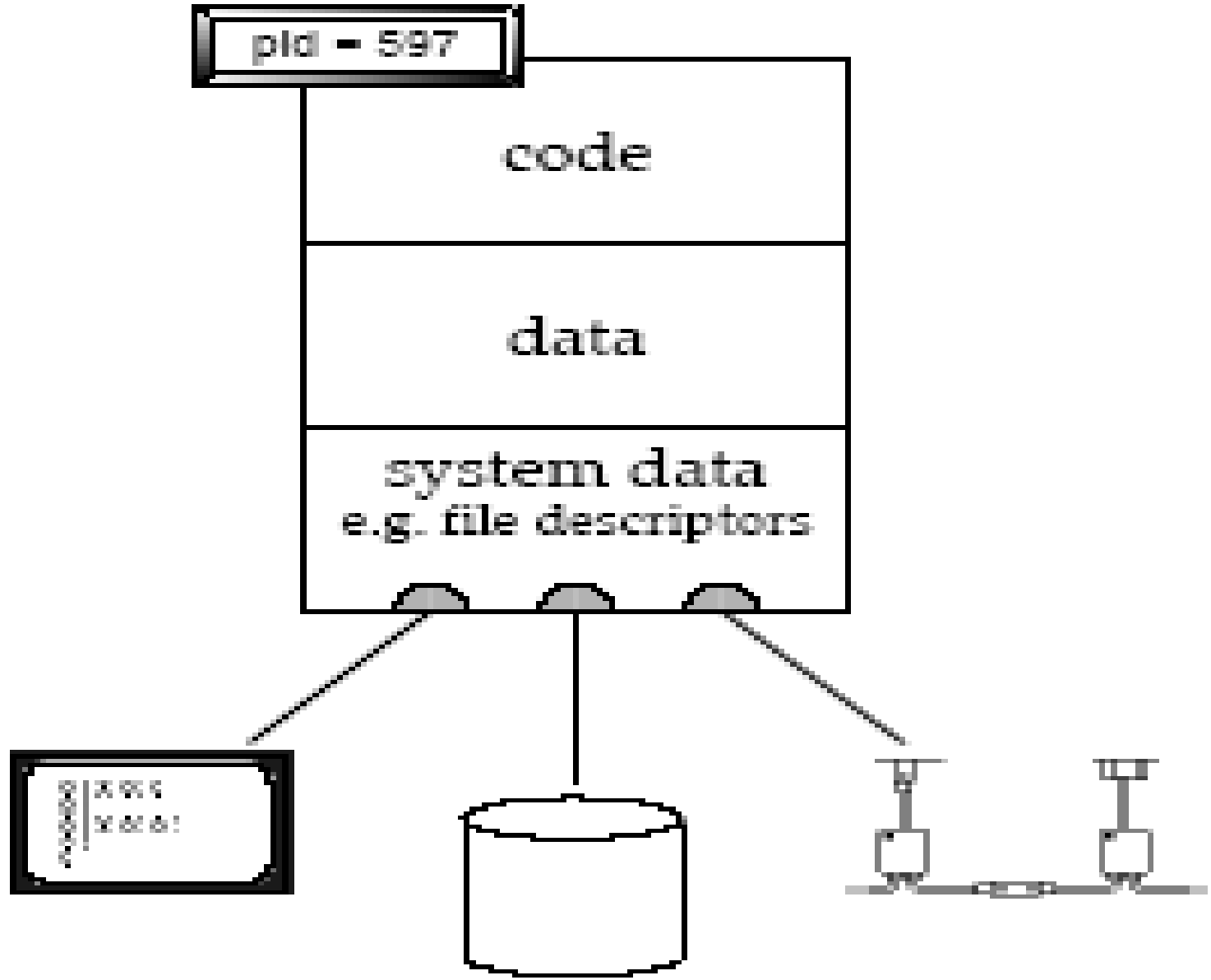
- When the fork system call is executed, a new process is created which consists of a copy of the address space of the parent
- An exact copy of the parent program is created
- This mechanism allows the parent process to communicate easily with the child process

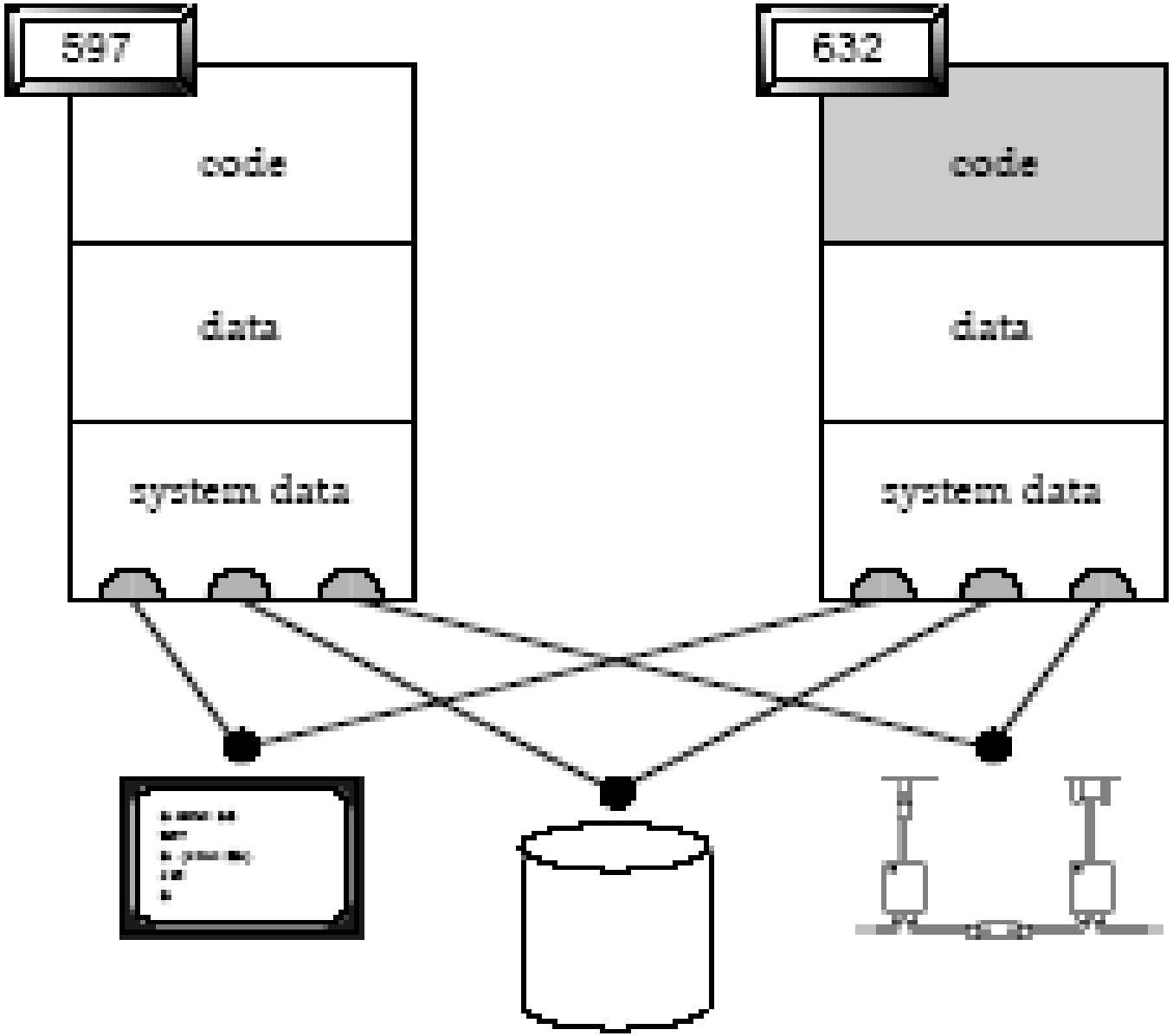
SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

System Call - fork() ...

- **On success:** (Child process is created)
  - The return code for fork is zero for the child process and the child process ID is returned to the parent process
  - Both processes continue execution at the instruction after the fork call
- **On failure:** (No child process is created)
  - A -1 is returned to the parent process
  - Variable **errno** is set appropriately to indicate the reason of failure


pid = 597
code
data
system data
e.g. file descriptors


597
code
data
system data
632
code
data
system data

Using fork() & exit() system call

- **Parent forks**

PID: 597 **Parent**

```
1. //fork1.c
2. int main()
3. {
4.      int i = 54, cpid = -1;
5.      cpid = fork();
6.      if (cpid == -1)
7.      {
8.              printf ("\nFork failed\n");
9.              exit (1);
10. }
11. if (cpid == 0)    //child code
12.    printf ("\n Hello I am child \n");
13. else                  //parent code
14.    printf ("\n Hello I am parent \n");
15. }
```

DATA

i = 54

cpid = -1

Using fork() & exit() system call

PID: 597 **Parent**

```
1. //fork1.c
2. int main()
3. {
4.  int i = 54, cpid = -1;
5.  cpid = fork();
6. if (cpid == -1)
7.  {
8.        printf ("\nFork failed\n");
9.        exit (1);
10. }
11. if (cpid == 0)    //child code
12.       printf ("\n Hello I am child \n");
13. else              //parent code
14.       printf ("\n Hello I am parent \n");
15. }
```

DATA

i = 54
cpid = 632

PID: 632 **Child**

```
1. //fork1.c
2. int main()
3. {
4.  int i = 54, cpid = -1;
5.  cpid = fork();
6. if (cpid == -1)
7.  {
8.        printf ("\nFork failed\n");
9.        exit (1);
10. }
11. if (cpid == 0)    //child code
12.       printf ("\n Hello I am child \n");
13. else              //parent code
14.       printf ("\n Hello I am parent \n");
15. }
```

DATA

i = 54
cpid = 0

- After fork parent and child are identical except for the return value of fork (and of course the PIDs).
- Because data are different therefore program execution differs.
- They are free to execute on their own from now onwards, i.e. after a successful or unsuccessful fork() system call both will start their execution from line#6.

Using fork() & exit() system call

PID: 597 **Parent**

```
1. //fork1.c
2. int main()
3. {
4.  int i = 54, cpid = -1;
5.  cpid = fork();
6.  if (cpid == -1)
7.  {
8.        printf ("\nFork failed\n");
9.        exit (1);
10. }
11. if (cpid == 0)    //child code
12.       printf ("\n Hello I am child \n");
13. else              //parent code
14.       printf ("\n Hello I am parent \n");
15. }
```

DATA

i – 54
cpid = 632

PID: 632 **Child**

```
1. //fork1.c
2. int main()
3. {
4.  int i = 54, cpid = -1;
5.  cpid = fork();
6.  if (cpid == -1)
7.  {
8.        printf ("\nFork failed\n");
9.        exit (1);
10. }
11. if (cpid == 0)    //child code
12.       printf ("\n Hello I am child \n");
13. else              //parent code
14.       printf ("\n Hello I am parent \n");
15. }
```

DATA

i – 54
cpid = 0

- When both will execute line 11, parent will now execute line 14 while child will execute line 12.

Example 1 -fork() & exit()

```
//fork1.c
int main()
{
  int cpid;
  cpid = fork();
  if (cpid == -1)
  {
     printf ("\nFork failed\n");
     exit (1);
  }
  if (cpid == 0)   //child code
     printf ("\n Hello I am child \n");
  else             //parent code
     printf ("\n Hello I am parent \n");
}
```

$100 Question

What will be the output of the code?

Output- 1

Hello I am child

Hello I am parent

OR

Output- 2

Hello I am parent

Hello I am child

- If child process executes first and then CPU executes the parent; we will get **Output-1**
- If parent process executes first, it terminates and the child process become Zombie, process **init** takes over control and execute the child process as its own child and we get output similar to **Output-2**

fork() – Child inherits from the Parent

- The child process inherits the following attributes form the parent:
  - Environment
  - Open File Descriptor table
  - Signal handling settings
  - Nice value
  - Current working directory
  - Root directory
  - File mode creation mask (umask)

fork() – Child Differs from the Parent

- The child process differs from the parent process:
  - Different process ID (PID).
  - Different parent process ID (PPID).

fork() – Reasons for Failure

- Maximum number of processes allowed to execute under one user has exceeded
- Maximum number of processes allowed on the system has exceeded
- Not enough swap space

System Call - wait()

- The wait() system call suspends the calling process until one of its immediate children terminates
- wait() returns prematurely if a signal is received
- See also waitpid() in man pages

```
SYNOPSIS
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
```

System Call - wait()

- If the call is successful, the process ID of the terminating child is returned
- If parent terminates, all its children are assigned process **init** as new parent. Thus the children still have a parent to collect their status and execution statistics
- **Zombie** process—a process that has terminated but whose exit status has not yet been received by its parent process or by **init**. (A process that has died but has yet not been reaped)
- **Orphan** process – a process that is still executing but whose parent has died. They do not become zombie rather are adopted by **init** process

Example 2 - wait() system call

```
//fork2.c
int main()
{
   int cpid, status;
   cpid = fork();
   if (cpid == -1)
   {
       printf ("\nFork failed\n");
       exit (1);
   }
   if (cpid == 0){
       printf ("\n Hello I am child \n");
       exit(0);
       }
   else {
       wait(&status);
       printf ("\n Hello I am parent \n");
}
```

System Call exec()

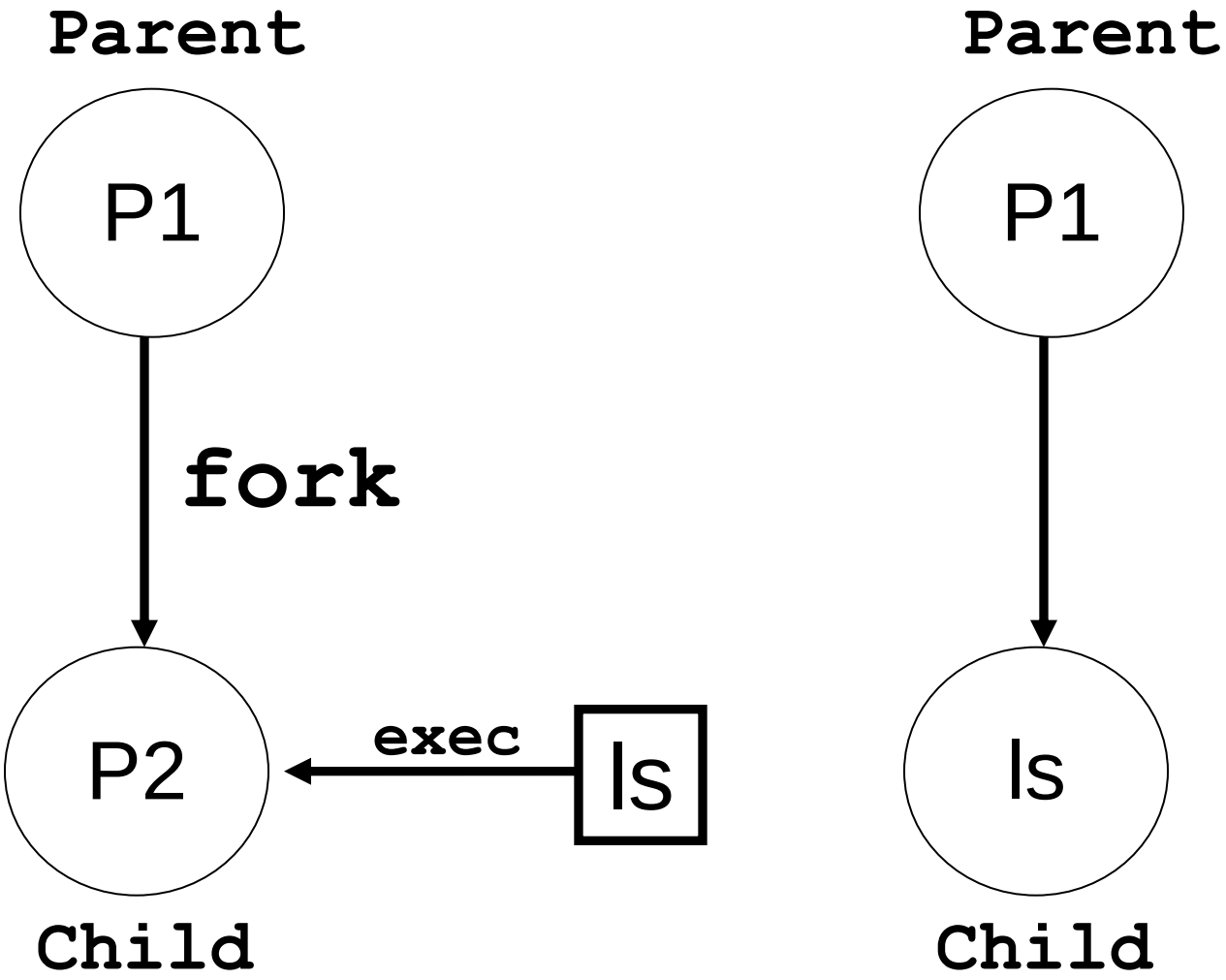
- Typically the exec() system call is used after a fork() system call by one of the two processes to replace the process memory space with a new executable program.
- The new process image is constructed from an ordinary, executable file.
- There can be no return from a successful exec() because the calling process image is overlaid by the new process image.

#include <unistd.h>

int execlp(const char *file,const char *arg0,...,const char *argn,(char *)0);

- `file` **is the path name of executable file which is going to override the caller process**
- **arg0 is the name given to child process.**
- **arg1 may be the options passed to the process.**
- **Last argument is null pointer '\0'**

System Call exec()



Example 3 - execlp() system call

```
//fork3.c
int main()
{
   int cpid, status;
   cpid = fork();
   if (cpid == -1)
   {
        printf ("\nFork failed\n");
        exit (1);
   }
   if (cpid == 0){
        execlp("/bin/ls", "ls", '\0');
        //execlp("/bin/ls", "ls", "-la", '\0');
        //execlp("less", "myless", "/etc/passwd", '\0');
        //execlp("/bin/cal", "cal", '\0');
        exit(0);
   }
   else {
        wait(&status);
        printf ("\n Hello I am parent \n");
    }
}
```

We're done for now, but Todo's for you after this lecture…

- Go through the slides and Book Sections: 3.1 to 3.3
- Type, compile, execute, and understand the programs on the slides and also the programs discussed in class
- Get Ready for Linux Lab # 01

If you have problems visit me in counseling hours. . . .