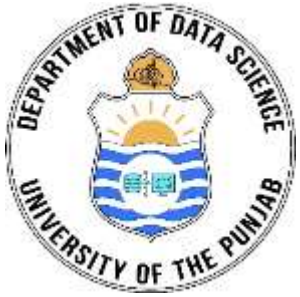


CMP325
Operating Systems
Lecture 04, 05

Process Management

Muhammad Arif Butt, PhD

Note:
Some slides and/or pictures are adapted from course text book and Lecture slides of

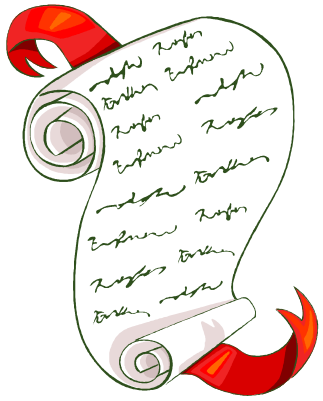
- Dr Syed Mansoor Sarwar
- Dr Kubiatowicz
- Dr P. Bhat
- Dr Hank Levy
- Dr Indranil Gupta

For practical implementation of operating system concepts discussed in these slides, students are advised to watch and practice following video lectures:

OS with Linux:
https://www.youtube.com/playlist?list=PL7B2bn3G_wfBuJ_WtHADcXC44piWLRzr8

System Programming:
https://www.youtube.com/playlist?list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW

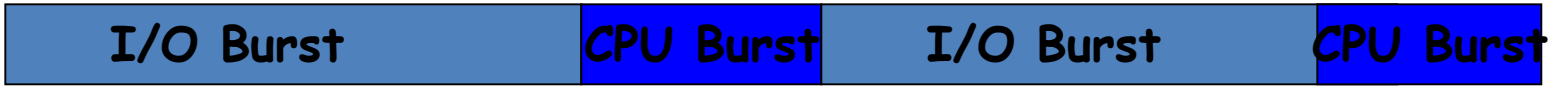
Today's Agenda



- Review of Previous Lecture
- Process States
- Introduction to Queuing Architecture
- Process Schedulers
- Process Creation and Termination
- Process and System Limits

CPU Bound and I/O Bound Processes

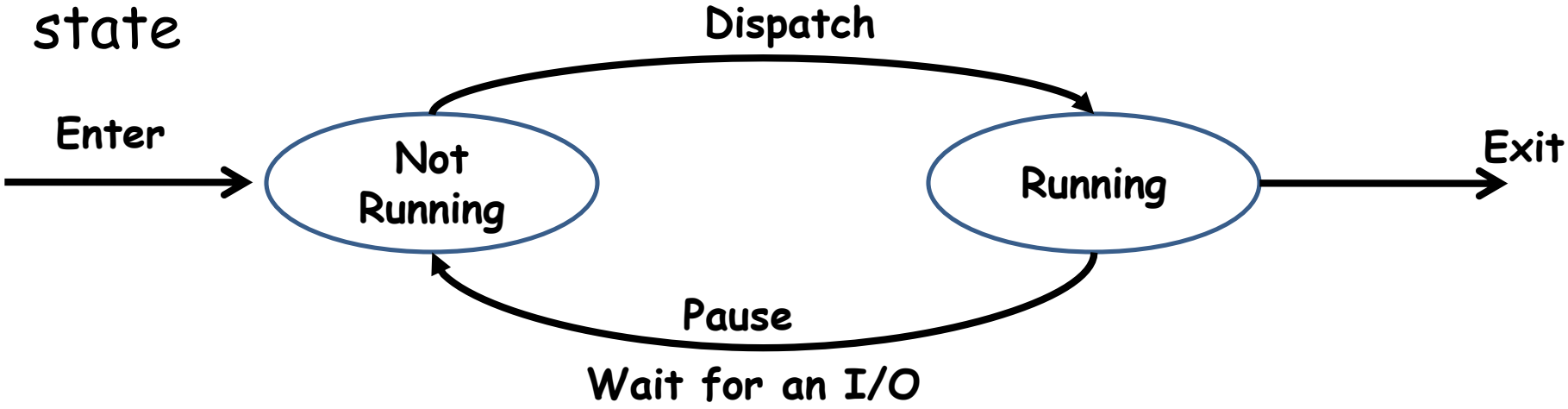
- ***I/O-bound process*** – spends more time doing I/O than computations; Many short CPU bursts
- Examples: Word processing, text editors. Billing system of Wapda which involves lot of printing



- ***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

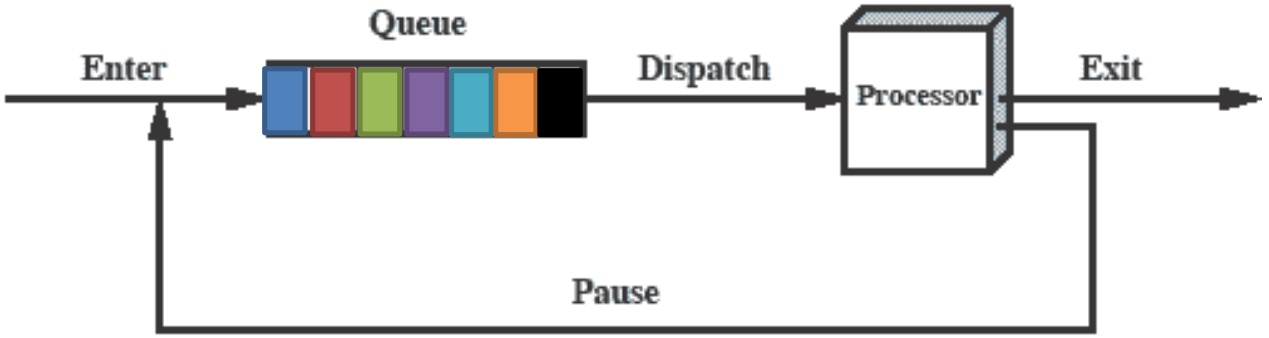
2-State Process Model

- 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:



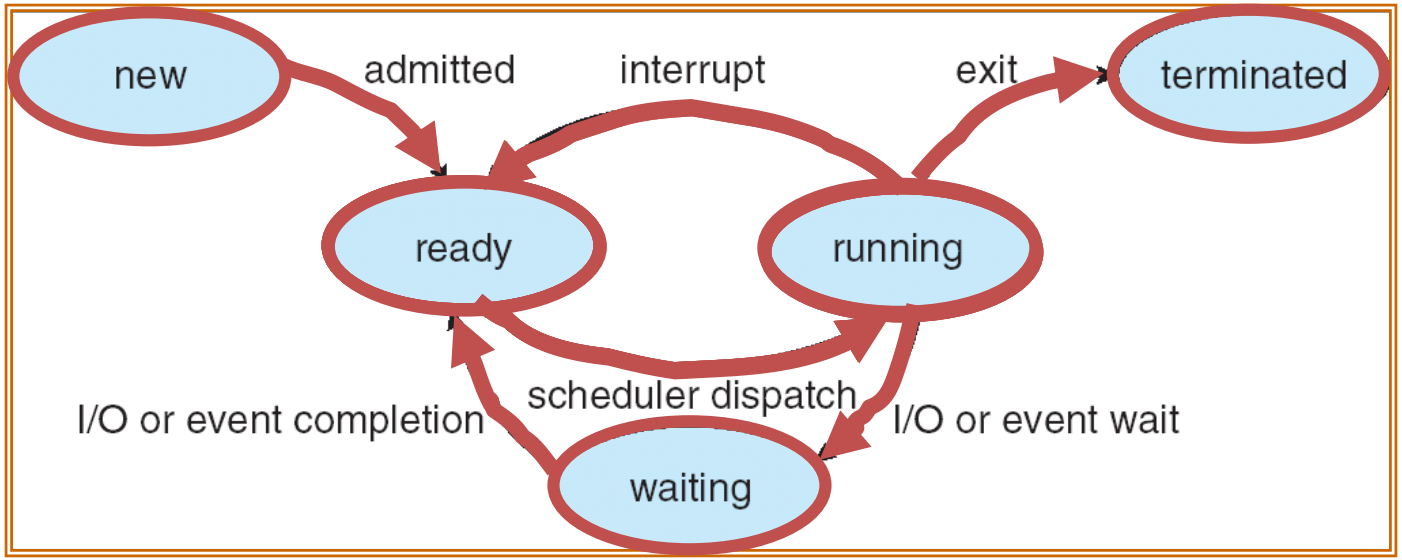
Limitations

- 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 **scheduler/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 (Disk to Memory)
  - **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

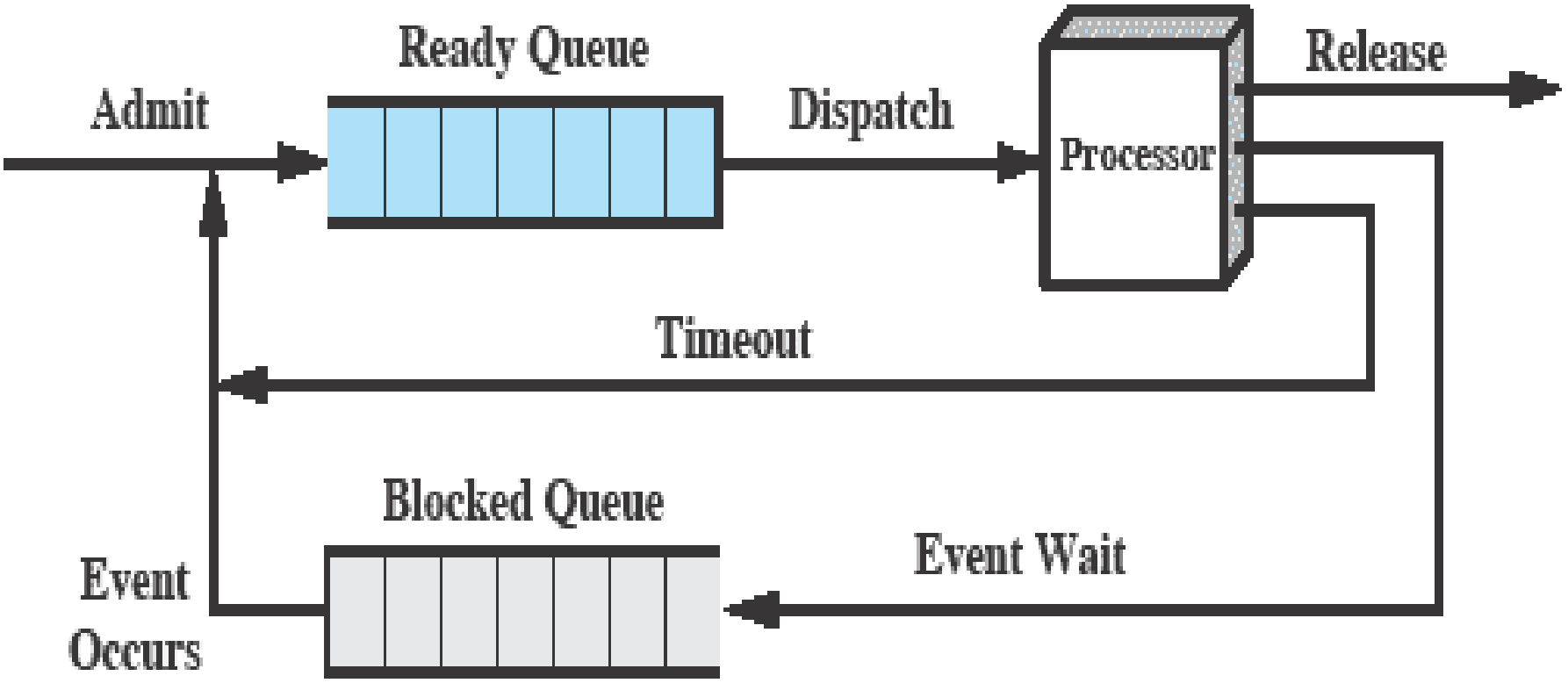
5-State Process Model



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 execution

Queuing Diagram (Using 2-Queues



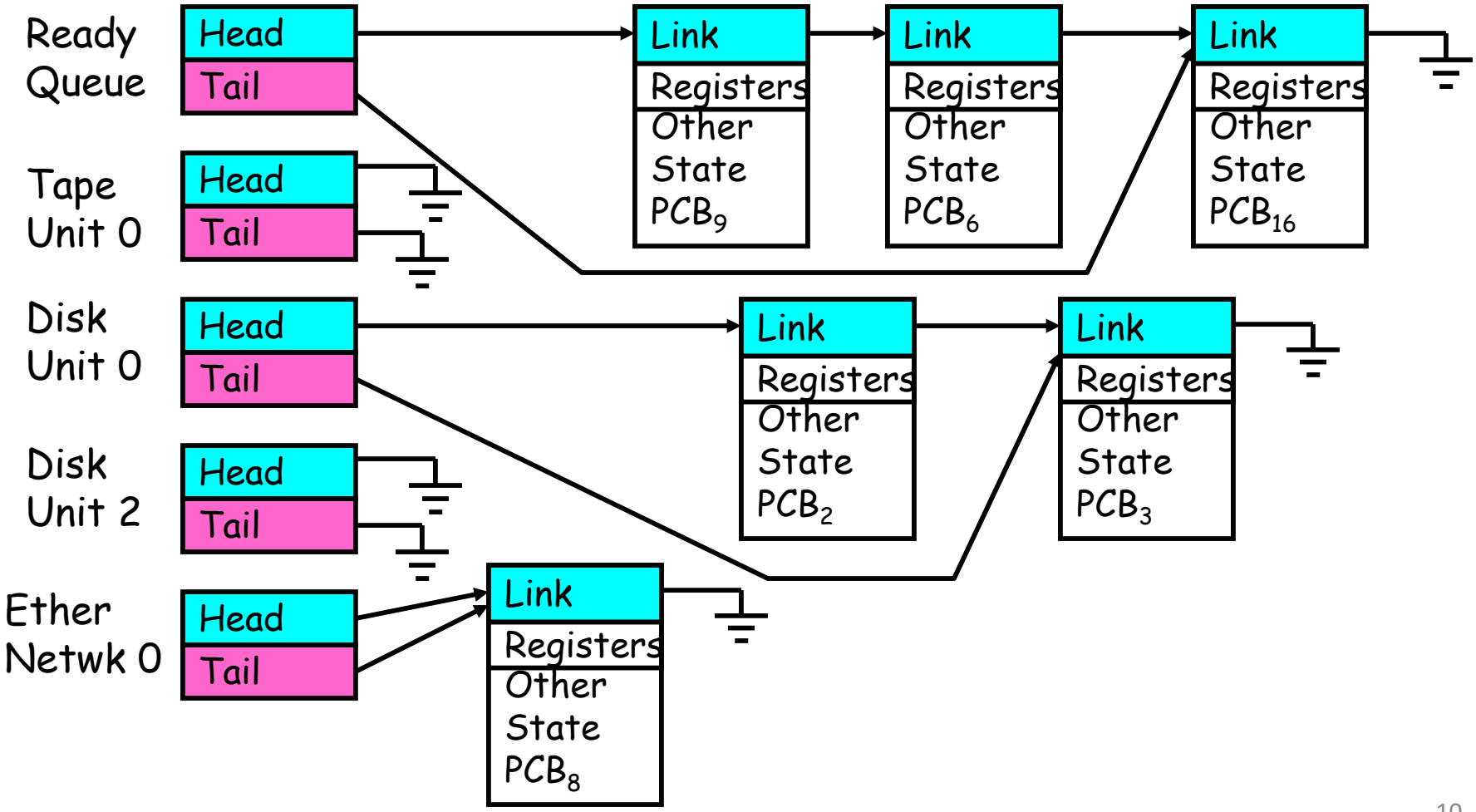
Limitation: When an event occurs the dispatcher would have to cycle through the entire Blocked Queue to see which process is waiting for that event. This can cause huge overhead when there may be 100's or 1000's of processes

Process Scheduling Queues

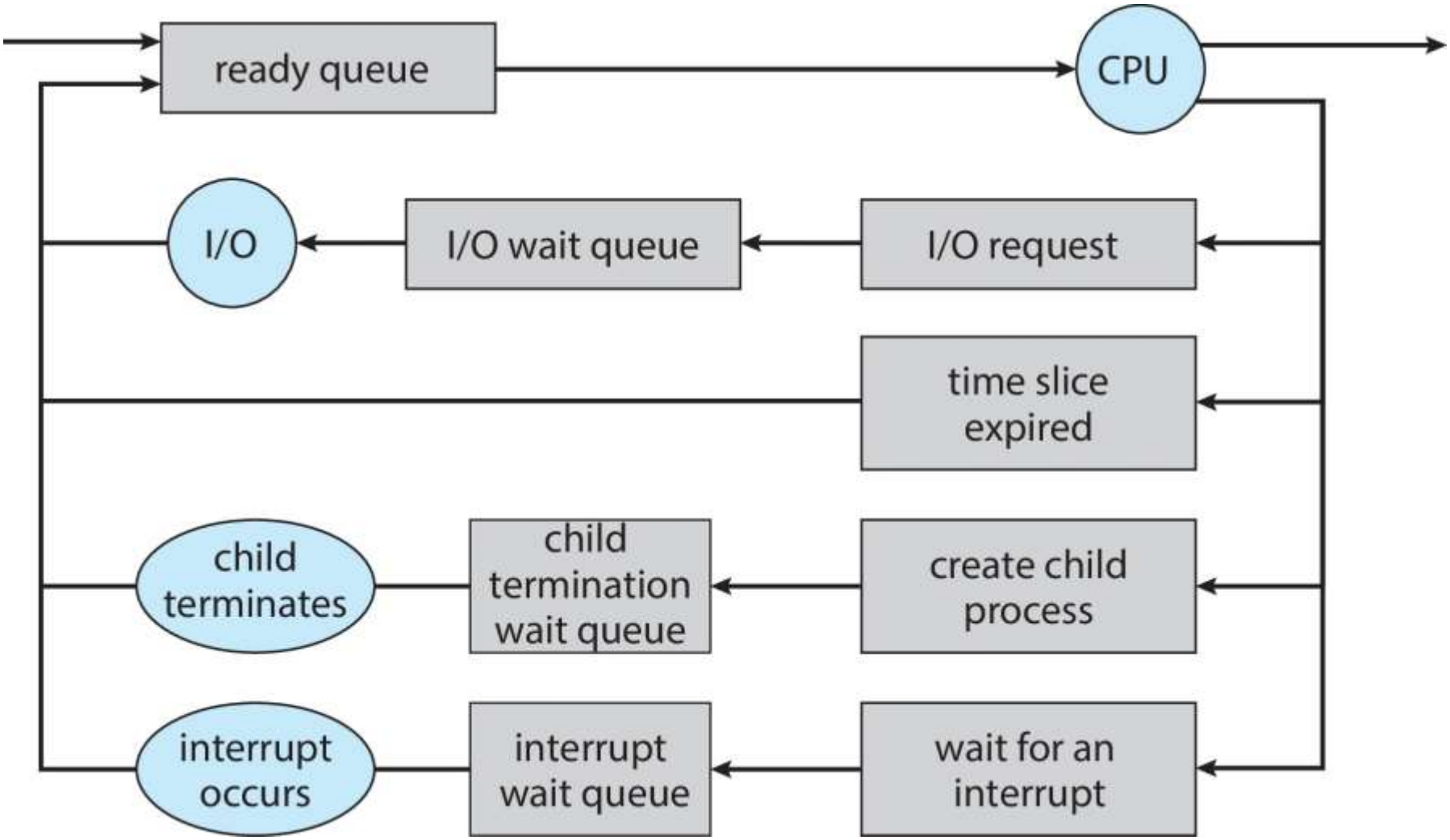
- **Job Queue** – When a process enters the system it is put into a job Queue. This queue consists of all processes in the system
- **Ready Queue** – This queue consists of processes that are residing in main memory and are ready and waiting to execute. It is generally stored as a link list
- **Device 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)
- Used in batch systems, timesharing OS has no long term scheduler
- 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/CPU/process scheduler is a kernel component that decides which process runs, when and for how long
- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- On a context/process switch the system saves the state of the currently running process and loads the saved state of the selected process from the run queue.
- Invoked when following events occur
    - CPU slice of the current process finishes
    - Current process needs to wait for an event
    - Clock interrupt
    - I/O interrupt
    - System call
    - Signal

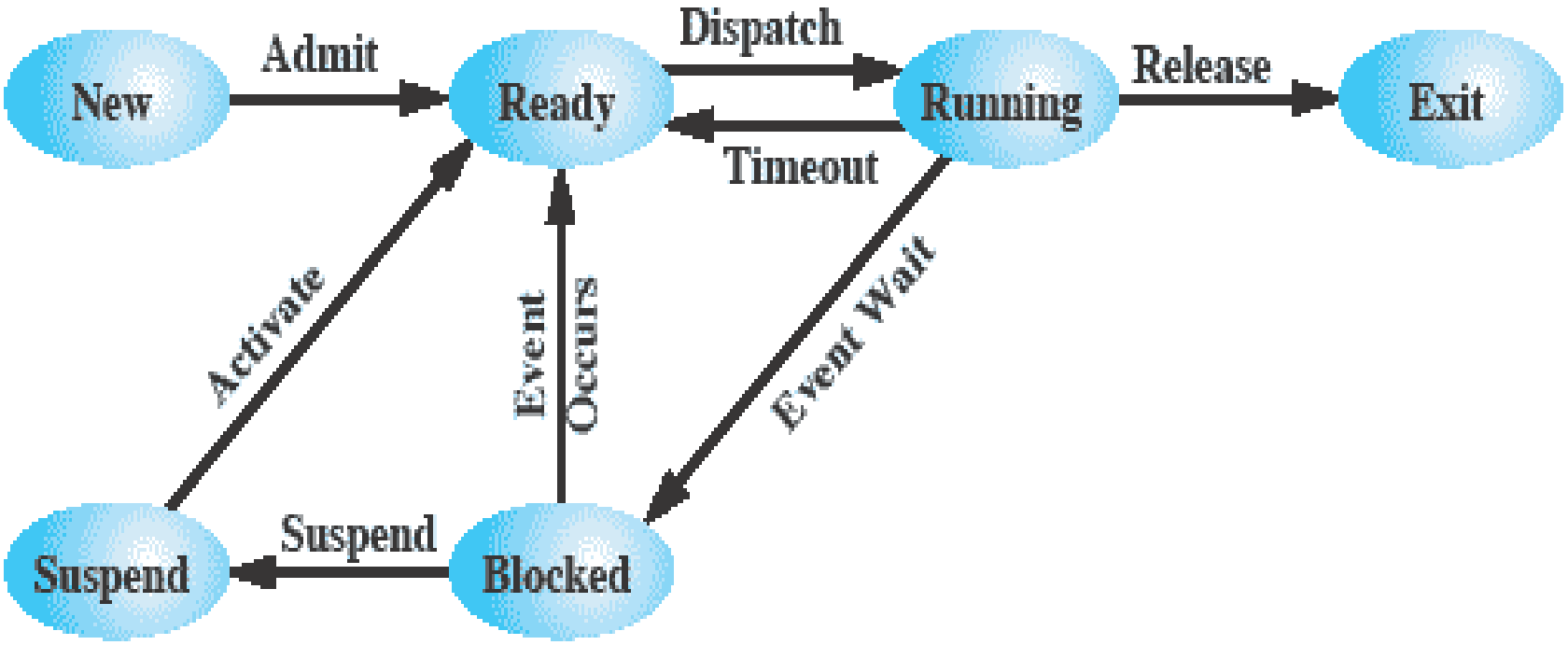
Medium Term Scheduler

- 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)

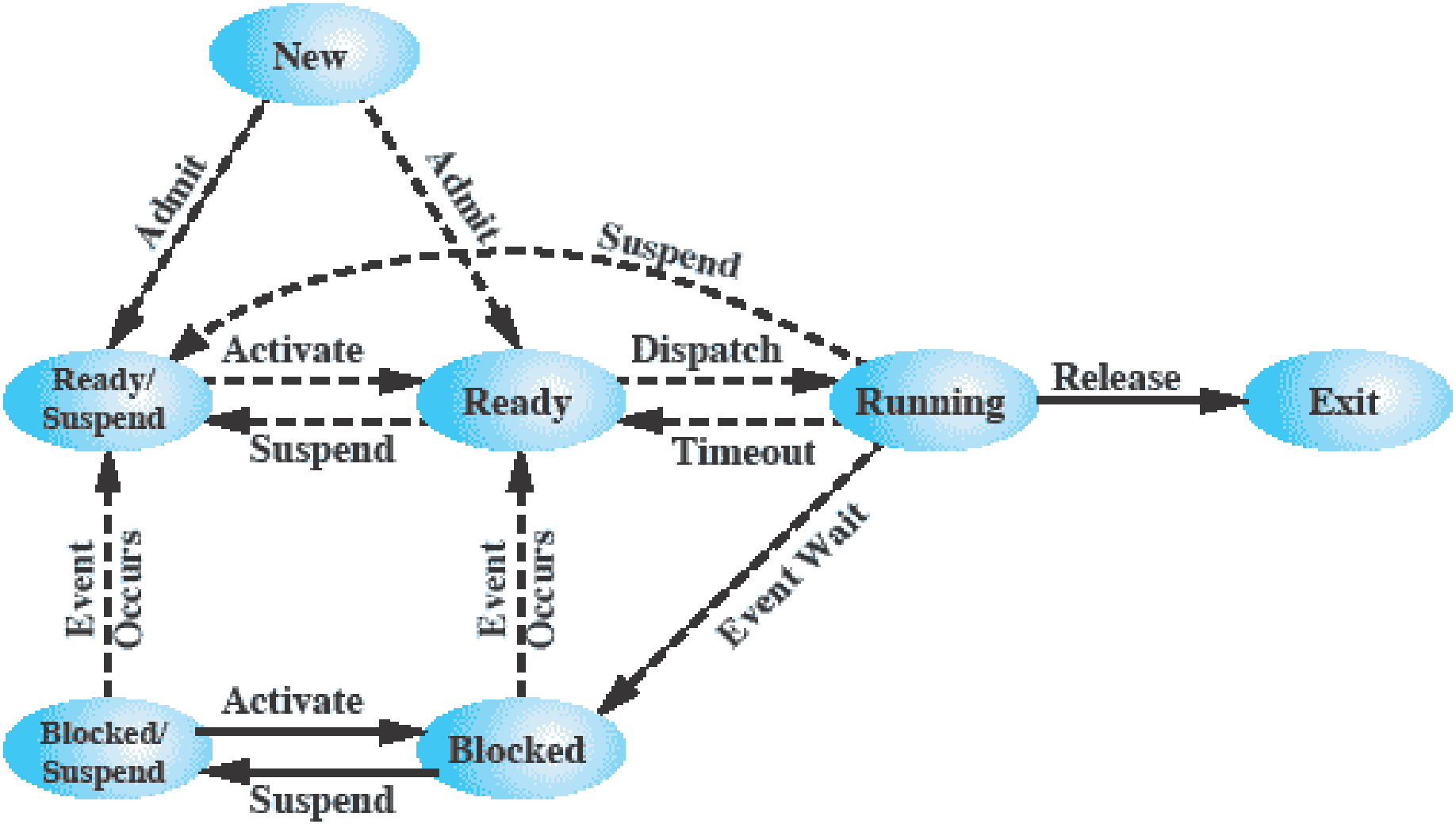


Limitation: This only allows processes which are blocked to be swapped out.

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

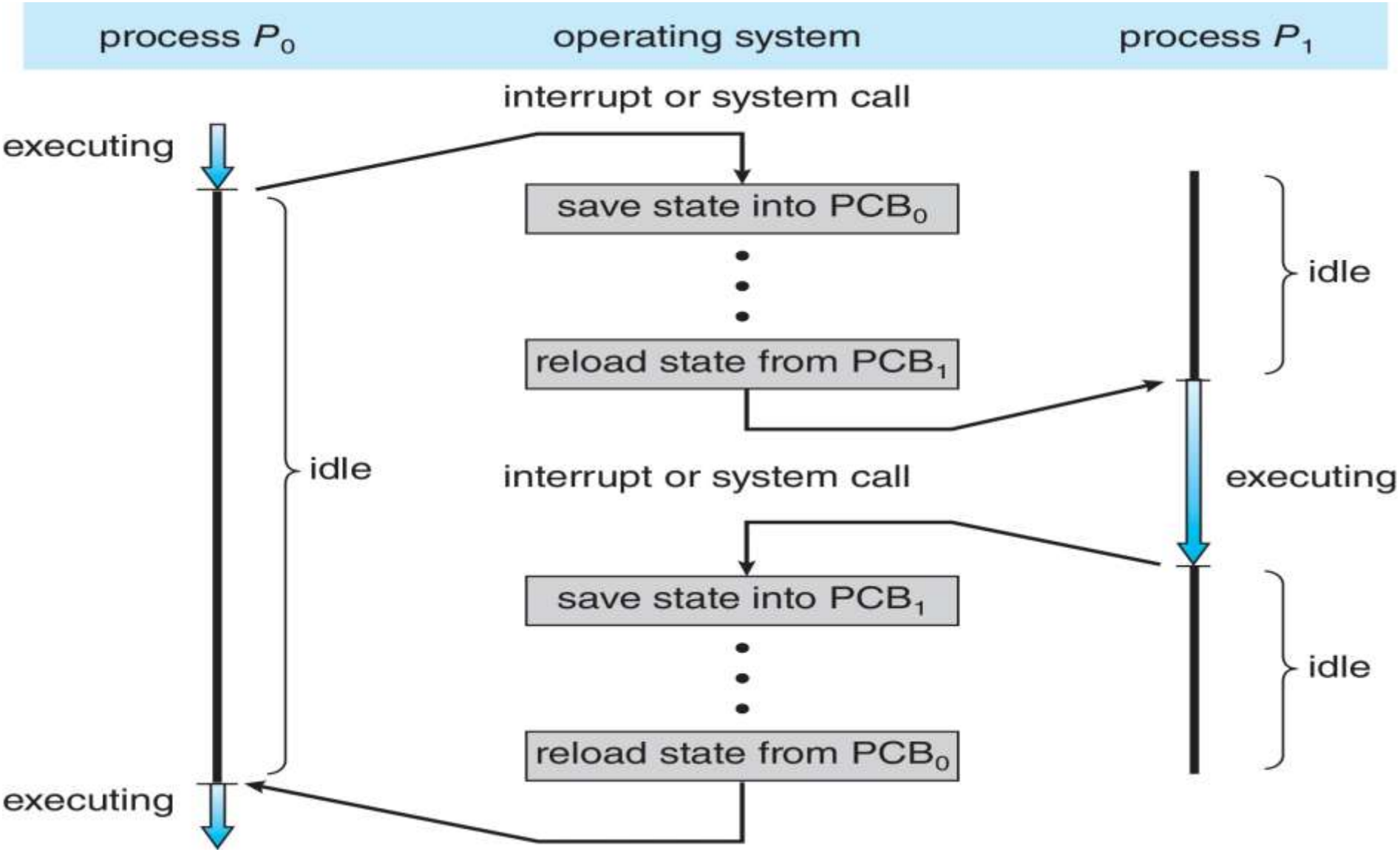
Solution: Add another state Ready Suspended and swap out a process from the ready state

7-State Process Model (two suspended state)



Context Switch / Process Switch

A context switch occurs when the CPU switches from one process to another



Context Switch / 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 / Process Switch

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

Process Creation

- Parent process create 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

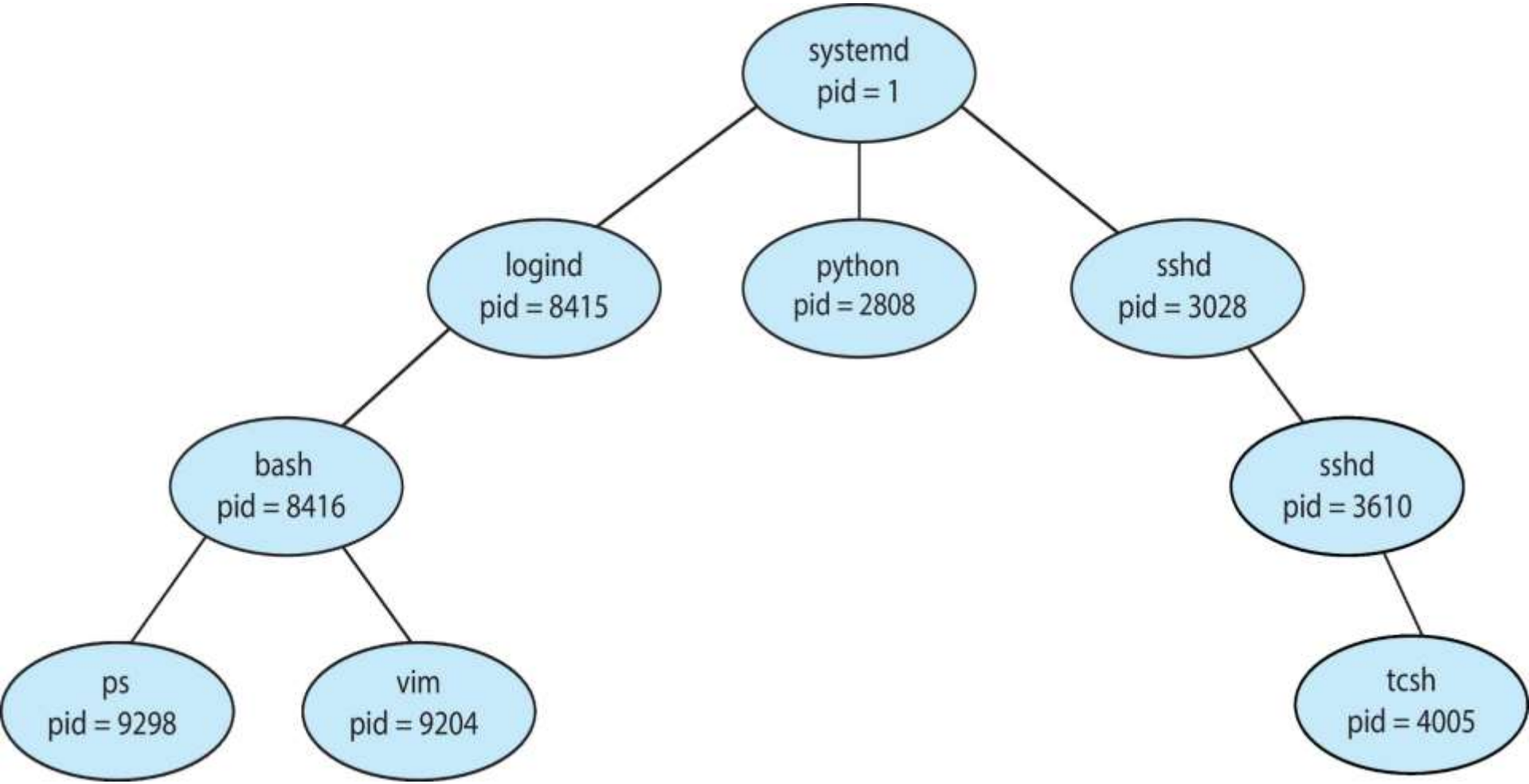
Process Creation (cont…)

- **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 (UNIX)
  - Parent waits until children terminate
- **Address Space**
  - Child duplicate of parent process (UNIX)
  - 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:
  - Task performed by the child is no longer required
  - Child has exceeded allocated resources (main memory, execution time, etc.)
  - Parent needs to create another child but has reached its maximum children limit
  - 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 ID and Parent Process ID

```
pid_t getpid();
pid_t getppid();
```

- Each process has a Process ID (PID), a positive integer that uniquely identifies a process on the system
- Above calls returns PID and PPID of the current process. Never fails
- On a shell you can get the PID of the shell in the environment variable in $$ and the parent ID in environment variable PPID
- The parent of any process can be found by looking at the 4th field of /proc/PID/stat file. Also see the 3rd field which shows the state of the process (RSDZTX). (See man page of proc for details)
- The init, now systemd is a user process having a PID of 1. It is invoked by the kernel at the end of the booting process
- Page daemon now kthreadd is a system process having a PID of 2. It support the paging of virtual memory system

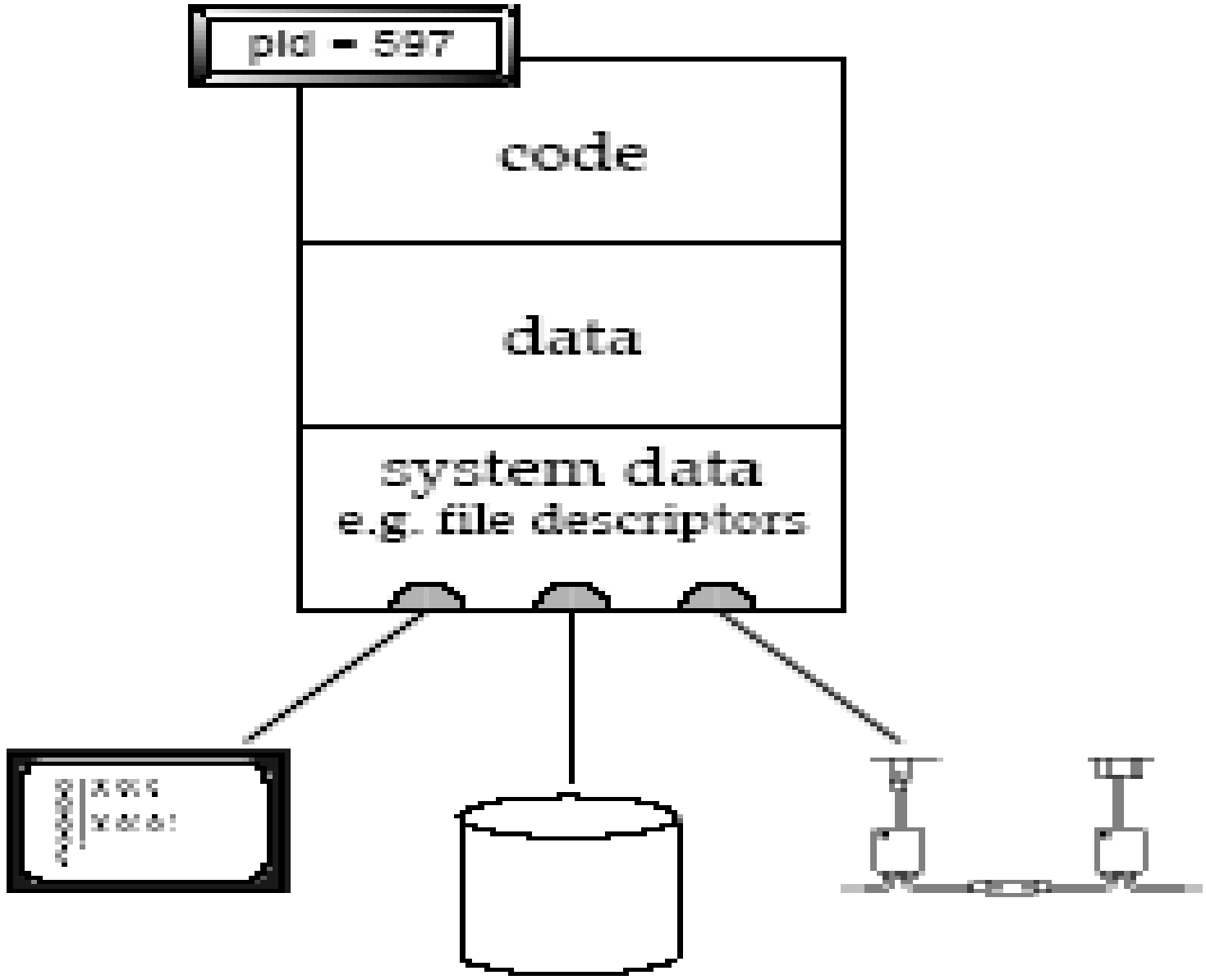
Process Creation using fork()

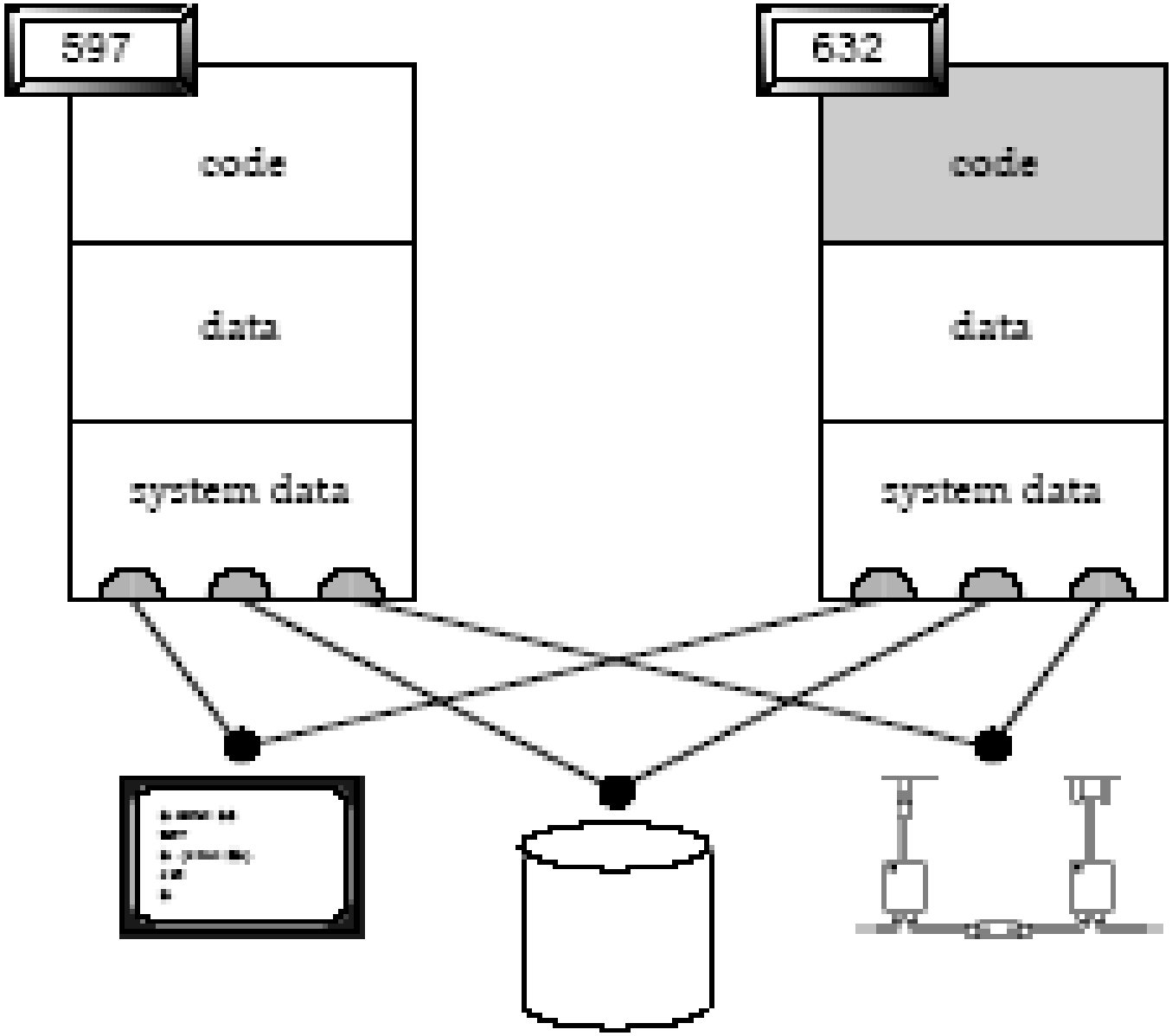
```
pid_t fork();
```

- The fork() system call allows one process, the parent, to create a new process, the child
- It is a system call which is called once but return twice, once in the parent and once in the child. To the parent process it returns PID of child process and to the child process it returns zero
- After the call returns both parent and child processes continues their execution concurrently from the next line of code
- The child process is a clone of the parent and obtains copies of the parent's stack, data, heap,and text segments
- PIDs are allocated sequentially to the new child processes, so effectively unique (but do wrap up after a very long time)
- On success, the return value to the child process is 0 and the return value to the parent process is PID of the child
- On failure, a -1 will be returned in the parent context, no child process created, and errno will be set appropriately

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


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 12 while child will execute line 14.

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 PID and PPID
  - Return value from fork()
  - Child times for CPU usage are reset to zero
  - File locks held by parent are not inherited to child
  - Set of pending alarms in the parent are cleared in the child
  - Set of pending alarms in the parent are cleared in the child

Main uses of fork()

1. When a process wants to duplicate itself, so that parent & child each can execute different sections of code concurrently. Example: Consider a network server; parent waits for a service request from a client. When the request arrives, parent calls fork & let the child handle the request. Parent goes back to listen for the next request
2. When a process wants to execute a different program. This is common for command shells where the child does an exec() right after it returns from the fork

Example 2 -fork() & exit()

```
int main(){
    int cpid = fork();
    if (cpid == 0)
        while(1)
            putchar('x');
    else
        while(1)
            putc('o', stdout);
    return 0;
}
```

Example 3 -fork() & exit()

```
int main(){
    int cpid = fork();
    if (cpid == 0)
       printf("child here.\n");
    else
       printf("parent here.\n");
    printf("who is here.\n");
    return 0;
}
```

Example 4 -fork() & exit()

```
/*Proves that the child inherits the copy of
  variables of the parent*/

int gvar=555;
int main(){
   int lvar = 54;
   int cpid = fork();
   if (cpid == 0)
     printf("Child process PID=%ld, gvar=%d,
     lvar=%d a\n", (long)getpid(),gvar, lvar);
   else
     printf("Parent process PID=%ld,gvar=%d,
     lvar=%d\n", (long)getpid(),gvar,lvar);
   return 0;
}
```

Example 5 -fork() & exit()

```
/* Proves that both parent and child have their
   own copy of variables*/
int main(){
   int i,cpid, ctr=0;
   cpid = fork();
   if (cpid == 0){
      ctr = 100;
      for (i = 0; i< 3; i++)
           printf("Child counter is:%d\n", ctr++);
   }
   else{
      for(i = 0; i< 3; i++)
           printf("Parent counter is:%d\n",ctr++);
   }
   return 0;
}
```

Example 6 -fork() & exit()

```
/* Describes what happens when fork() is called
   multiple times*/
int main(){
   fork();
   fork();
   fork();
   printf("Hello fork...\n");
   return 0;
}
```

Example 7 -fork() & exit()

```
/* Describes what happens when fork() is called
   multiple times*/
int main(int argc, char *argv[]){
   if(argc!=2){
      printf("Must enter one int agrument\n");
      exit(1);
   }
   int n = atoi(argv[1]);
   int i;
   for (i=1;i<=n;i++)
      fork();
   printf("PUCIT\n");
   exit(0);
}
```

Example 8 -fork() & exit()

```
/* Describes what happens when fork() is called
   multiple times*/
int main(int argc, char *argv[]){
   if(argc!=2){
      printf("Must enter one int agrument\n");
      exit(1);
   }
   int n = atoi(argv[1]);
   int i;
   for (i=1;i<=n;i++){
      fork();
   }
   fprintf(stderr, "%s\n","PUCIT");
   exit(0);
}
```

Repeat above program if the fprintf() statement is inside the for loop

Example 9 -fork() & exit()

```
/* Describes what happens when fork() is called
   multiple times*/
int main(){
   fork() && fork();
   fprintf(stderr, "%s\n","PUCIT");
   return 0;
}
```

Orphan Processes

- If a parent has terminated before reaping its child, and the child process is still running, then that child is called orphan
- In UNIX all orphan processes are adopted by init or systemd which do the reaping

```
int main(){
   int cpid = fork();
   if (cpid == 0){
       printf("Running child, PID=%ld PPID=%ld\n",
       (long)getpid(), (long)getppid());
       while(1);
   }
   else
       printf("Terminating parent, PID=%ld PPID=%ld\n",
       (long)getpid(), (long)getppid());
   exit(0);
   return 0;
}
```

Zombie Processes

- Zombie Process is a process that has terminated but its parent has not collected its exit status and has not reaped it. So a parent must reap its children
- When a process terminates but still is holding system resources like PCB and various tables maintained by OS. It is half-alive & half-dead because it is holding resources like memory but it is never scheduled on the CPU
- Zombies can't be killed by a signal, not even with the silver bullet (SIGKILL). The only way to remove them from the system is to kill their parent, at which time they become orphan and adopted by init or systemd

```
int main(){
   int cpid = fork();
   if (cpid == 0){
       printf("Terminating child with PID = %ld\n",
   (long)getpid());
       exit (0);
   }
   else{
       printf("Running parent, PID=%ld\n",(long)getpid());
       while(1);
   }
   return 0;
}
```

Reaping Child Process using wait()

```
pid_t wait(int* status);
```

- The wait() system call is used for reaping and cleaning zombies from system, and serves two purposes
  - Notify parent that a child process finished running
  - Tell the parent how a child process finished
- The parent process calls the wait() system call and gets blocked till any one of its child terminates
- The child process returns its termination status using the exit() call and that integer value is received by the parent inside the status argument
- On the shell, we can check this value in the $? environment variable
- On success, the wait() system call returns PID of the terminated child and in case of error returns a -1

Example -fork() & wait()

```
int main(){
   int cpid = fork();
   if (cpid == 0){
        printf("Hello I m Child.\n");
        sleep(1);
        printf("I m Child again, and my PID is %ld\n",
        getpid());
        sleep(1);
        printf("I m Child again, and I am terminating...\n");
        sleep(1);
        exit(7);
   }else{
        int rv = wait(NULL);
        printf ("Hello I m Parent.\n");
        printf("Return value of wait is %ld, and status is
        unknown\n",rv);
        exit(54);
   }}
```

```
$ ./a.out
Hello I m Child.
I m Child again, and my PID is 6867
I m Child again, and I am terminating...
Hello I m Parent.
Return value of wait is 6867, and status is unknown
$ echo $?
54
```

Executing a different program using exec()

```
int execl(char* pathname, char* arg0, ..., (char*)0);
```

- A process may overwrite itself with another executable image. When a process calls one of the six exec() functions, it is completely replaced by the new program, and the new program starts executing its main function
- There are five library functions of exec family and all are layered on top of the execve() system call. Each of these functions provides a different interface to the same functionality
- There is no return after a successful exec call. The exec() functions return only if an error has occurred. The return value is -1, and errno is set to indicate the error

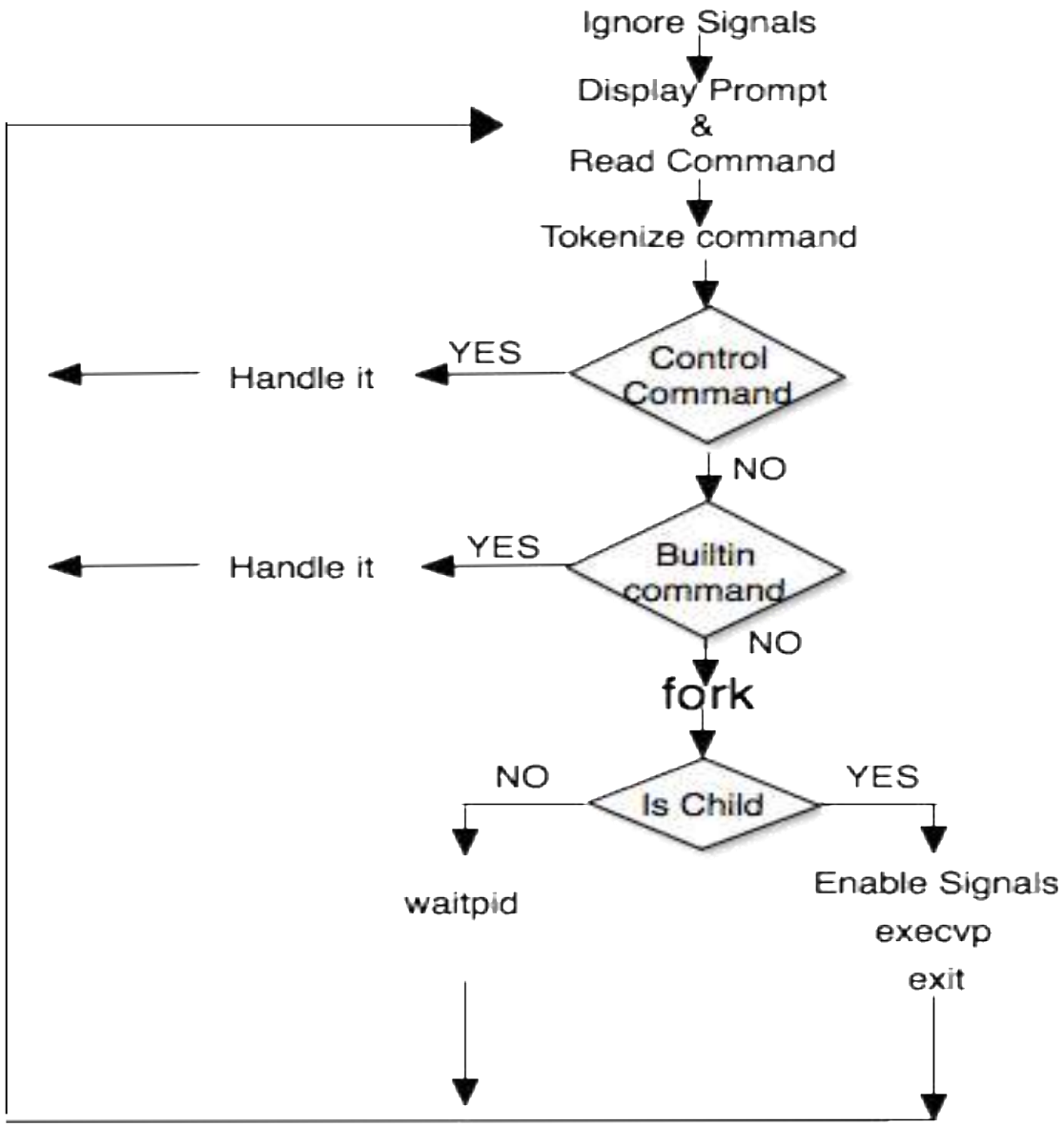
Example 1 -fork() & execl()

```
int main(){
  int cpid = fork();
  if (cpid == 0){
  execl("/bin/ls", "myls", "-l", "/home/", NULL);
  printf("This line will not be printed\n");
  }
  else{
     wait(NULL);
     printf("Hello I m Parent.\n");
  }
  return 0;
}
```

Example 2 -fork() & execl()

```
int main(){
  int cpid = fork();
  if (cpid == 0){
  execl("/usr/bin/gnome-calculator", "mycalc",NULL);
  printf("This line will not be printed\n");
  }
  else{
     wait(NULL);
     printf("Hello I m Parent.\n");
  }
  return 0;
}
```

How a Shell executes Commands



Process Resource Limits

- Every process has as set of resource limits that can be used to restrict the amounts of various systems resources that the process may consume
- We can set the resource limits of the shell using the ulimit built-in command. These limits are inherited by the processes that the shell creates to execute user commands
- In Linux kernel, the /proc/PID/limits file can be used to view all of the resource limits of any process
- The ulimit is an internal command that provides control over the resources available to the shell and to processes started by it. See man pages for details
- To display various limits

```
$ ulimit -a
```

Process Resource Limits

Hard and Soft Limits:

- A hard limit is the upper limit that the user can never, ever go beyond. Say you set a hard limit of 255 processes per user. No one of the users can exceed that limit, ever
- The soft limit, on the other hand, is a "warning" limit. It tells the user and the system admin that you are close to reach the danger level, which is the hard limit. Users are allowed to go over the soft limit, unlike the hard limit
- Regular users can increase their soft limits up to the current hard limit, but can't exceed that. They can decrease their soft limits to zero
- Regular users can also decrease their hard limits to zero, but they can't increase them
- The root is, as always, god of the system, and so can do whatever (s)he likes with both soft and hard limits

Process Resource Limits

- To display stack size of a process

```
$ ulimit -s
8192
$ ulimit -Ss
8192
$ ulimit -Hs
unlimited
```

- To display maximum number of open files that a process can open at any instant of time

```
$ ulimit -n
1024
$ ulimit -Sn
1024
$ ulimit -Hn
4096
```

Accessing System Configurations

- To query different system configuration variables (kernel variables mostly defined in limits.h), we can use the command getconf
- To display all configuration variables and their current value

```
$ getconf -a
```

- To query maximum file name support

```
$ getconf NAME_MAX
```

- To query maximum number of child processes that may be owned by a process simultaneously

```
$ getconf CHILD_MAX
```

- To query min/max ranges of various data types

```
$ getconf UCHAR_MAX
$ getconf CHAR_MAX
$ getconf CHAR_MIN
```

SUMMARY

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
- Go through Unix The Text Book Chapter 13
- Type, compile, execute, and understand the programs on the slides and also the programs discussed in class
- Start making your hands dirty before you get your First Programming Assignment

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