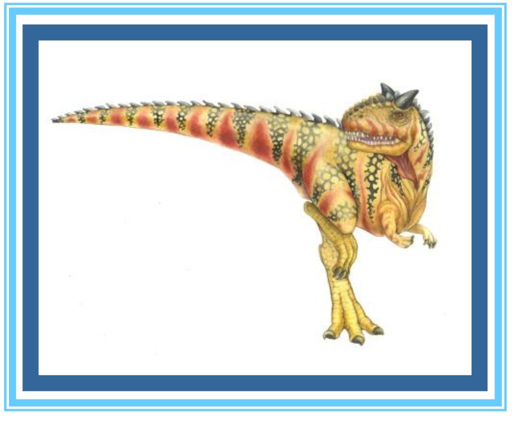
Module 2: Processes



Processes Management

- Process Concept
- Process Scheduling
- Operations on Processes
- Fork (), Exec() , wait ()
- Zombie & Orphan Process

Objectives

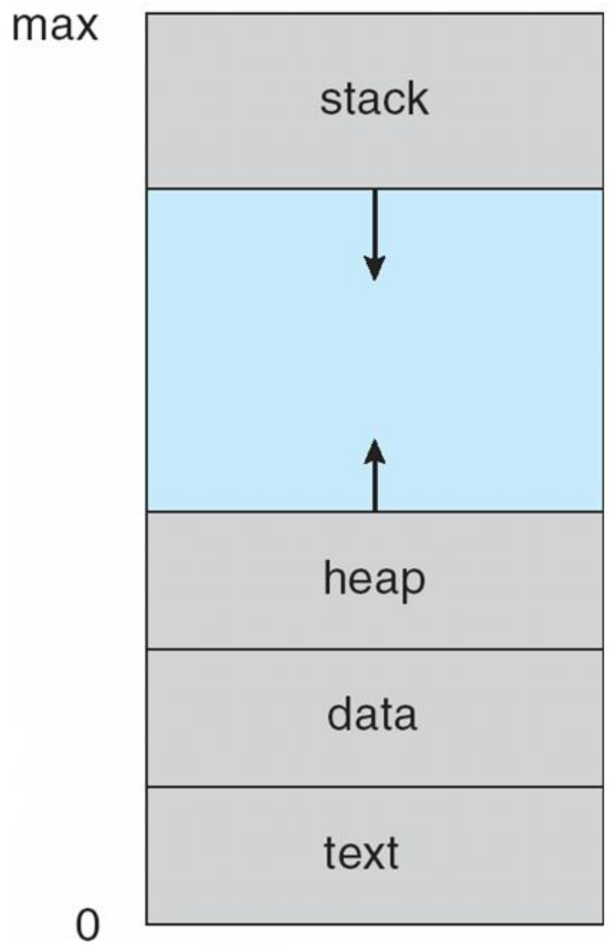
- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication

Process Concept

- **Process** – a program in execution; process execution must progress in sequential fashion
- A process is the unit of work in a modern time-sharing system.
- 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 in Memory

- Multiple parts
- The program code, also called **text** section
- Current activity including **program counter**, processor registers
- **Stack** containing temporary data
  - Function parameters, return addresses, local variables
- **Data section** containing global variables
- **Heap** containing memory dynamically allocated during run time



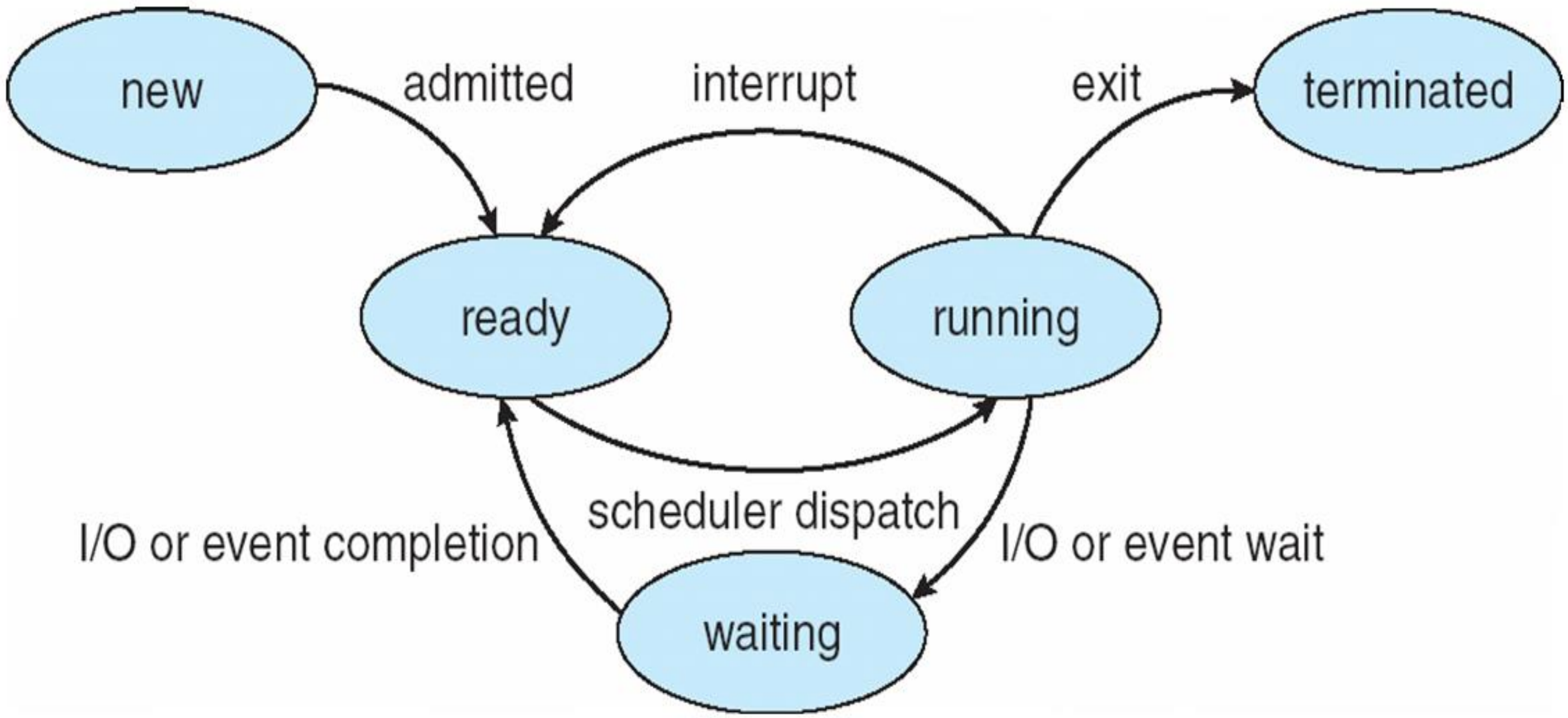
Process Concept (Cont.)

- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes?
  - Consider multiple users executing the same program

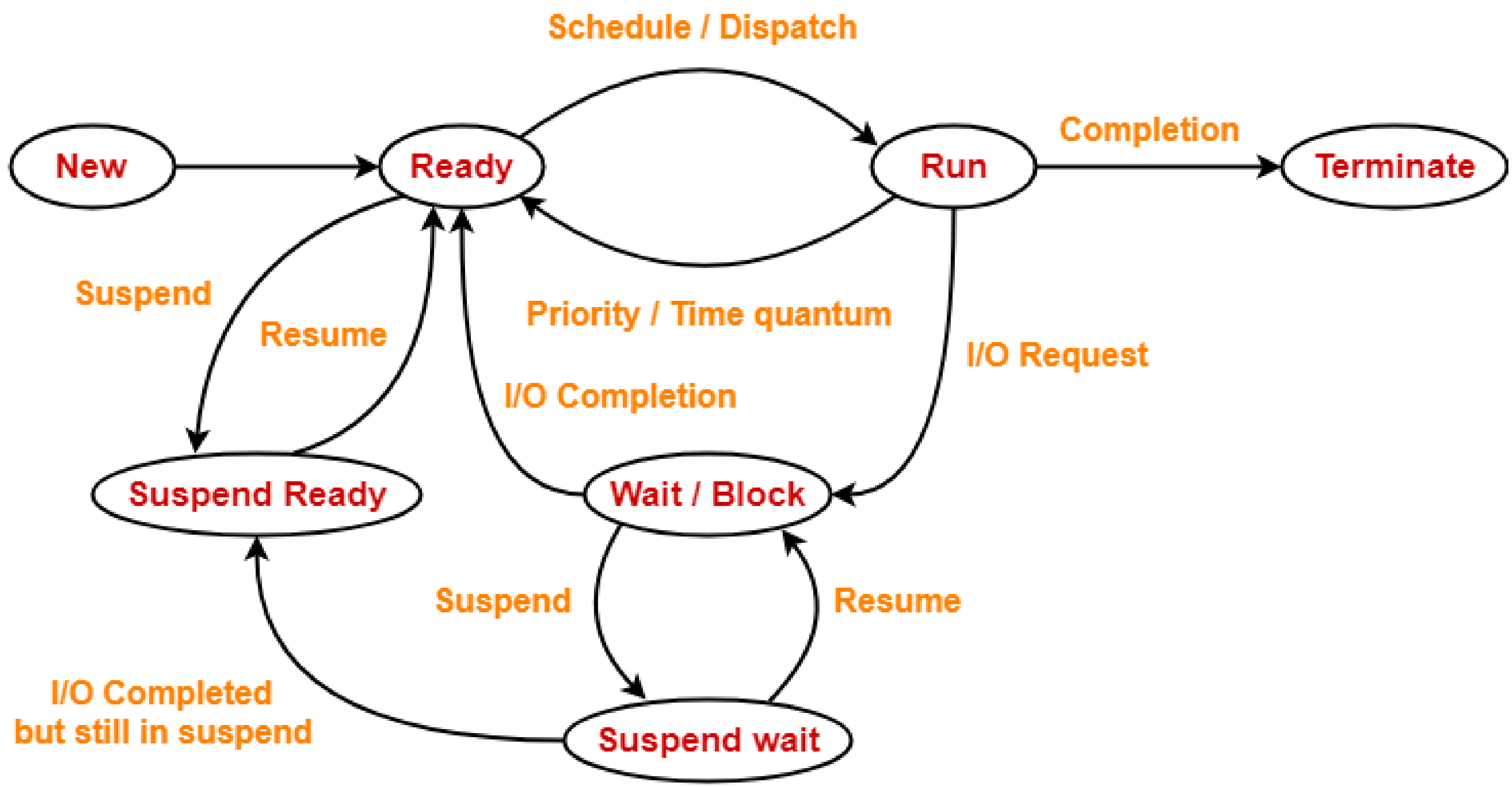
Process State

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

Diagram of Process State



7 process state model



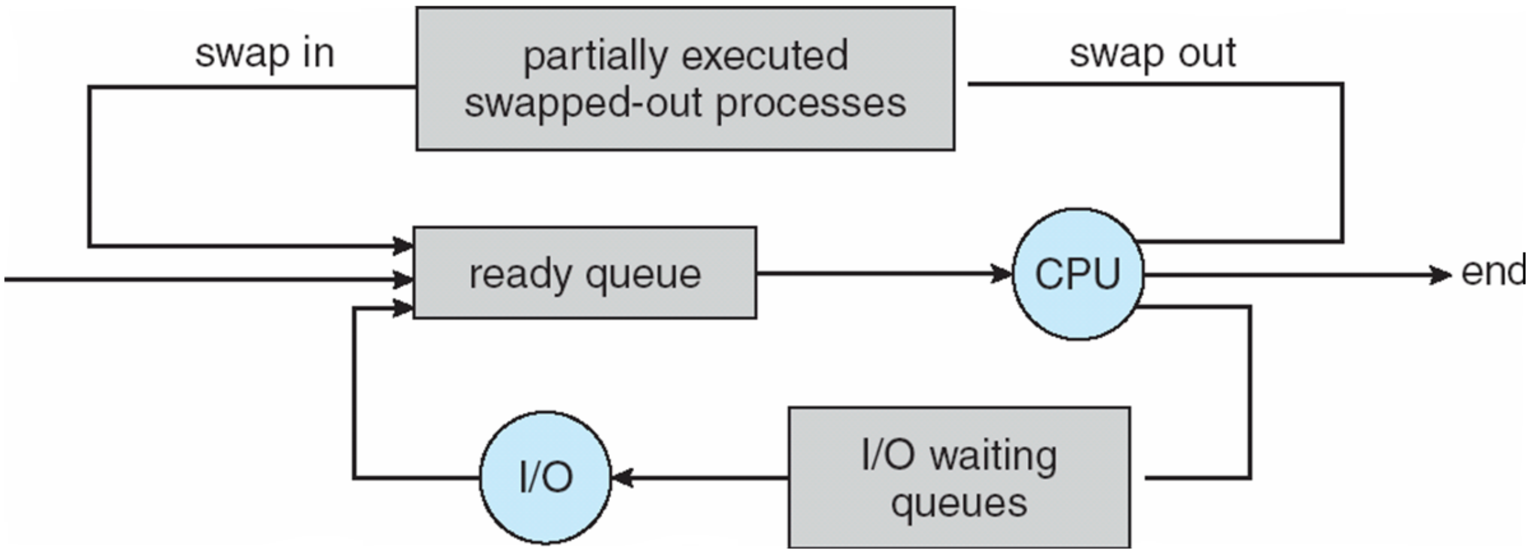
Process State Diagram

Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) $\Rightarrow$ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) $\Rightarrow$ (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
- Long-term scheduler strives for good ***process mix***

Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**

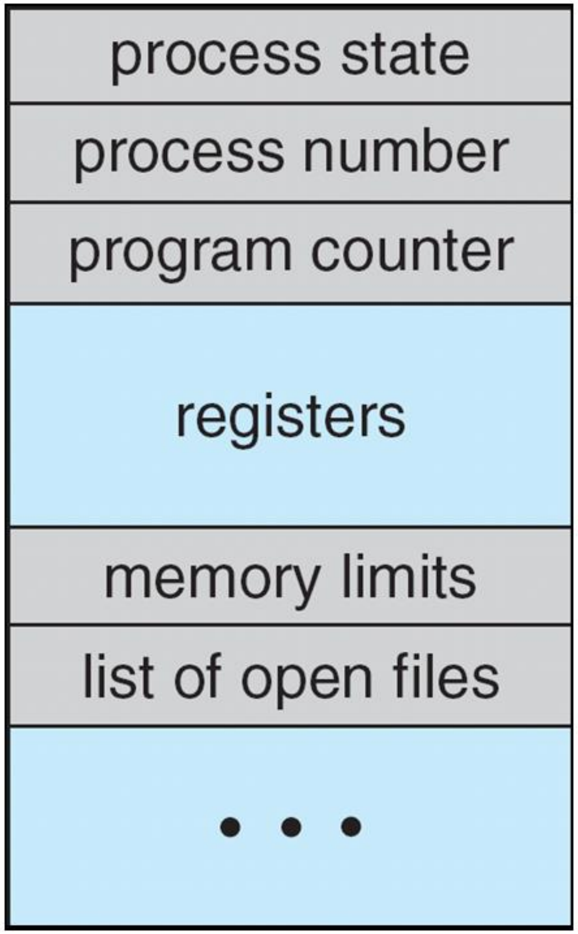


Process Control Block (PCB)

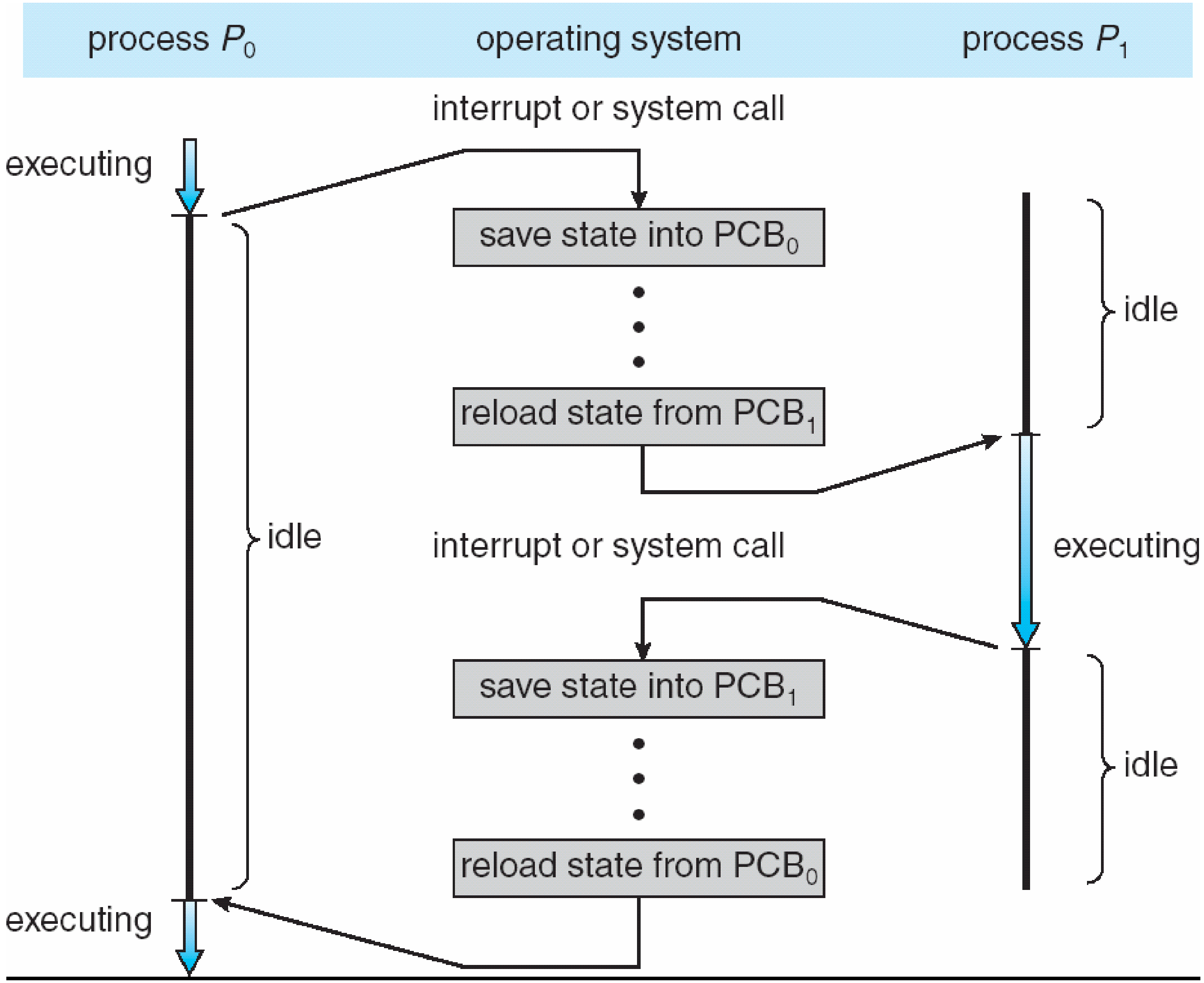
Information associated with each process

(also called **task control block**)

- **Process state** – running, waiting, etc
- **Program counter** – location of instruction to next execute
- **CPU registers** – contents of all process-centric registers-accumulators
- **CPU scheduling information-** priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to the process
- **Accounting information** – CPU used, clock time elapsed since start, time limits
- **I/O status information** – I/O devices allocated to process, list of open files



CPU Switch From Process to Process



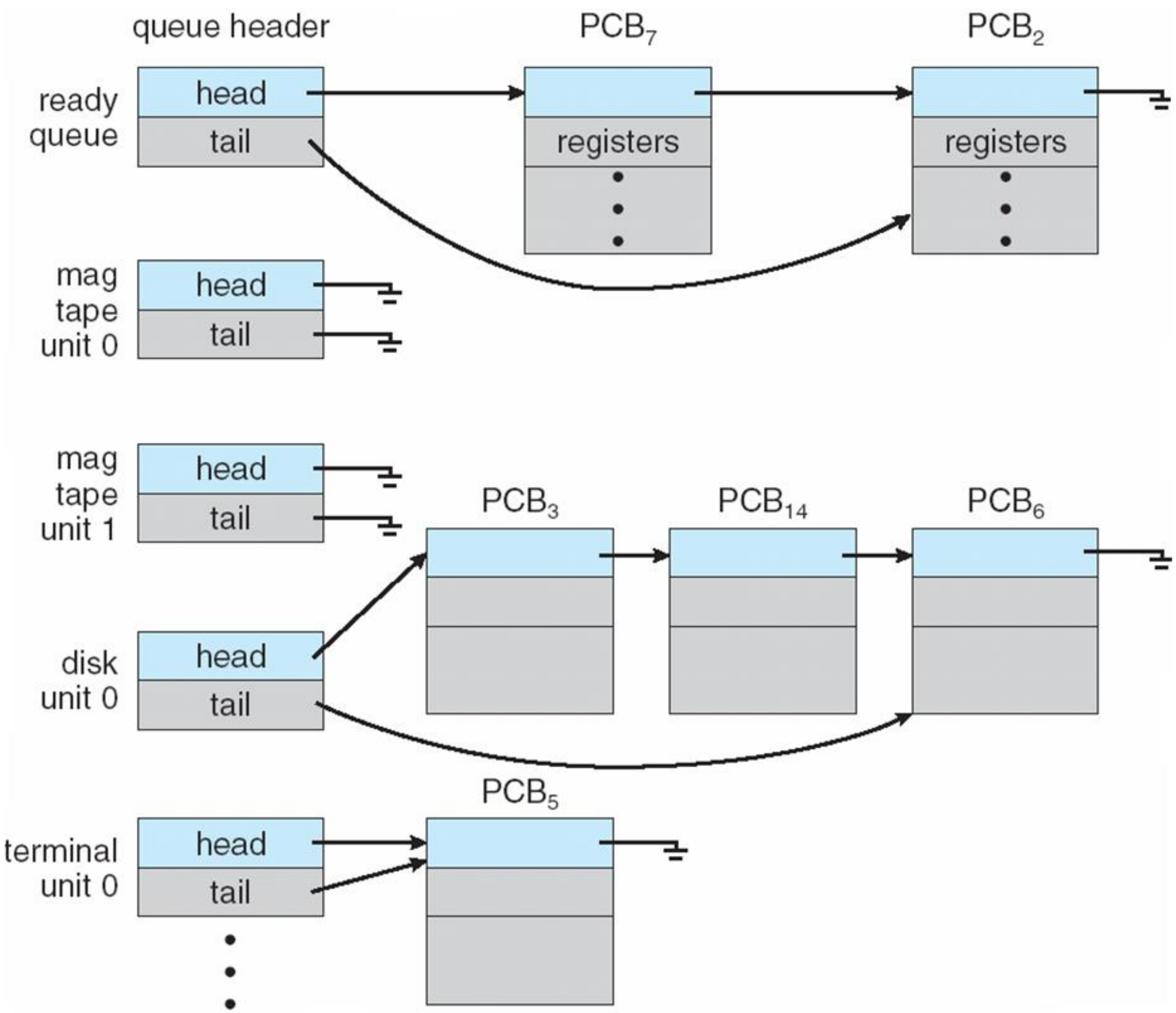
Context Switch

- 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 via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once

Process Scheduling

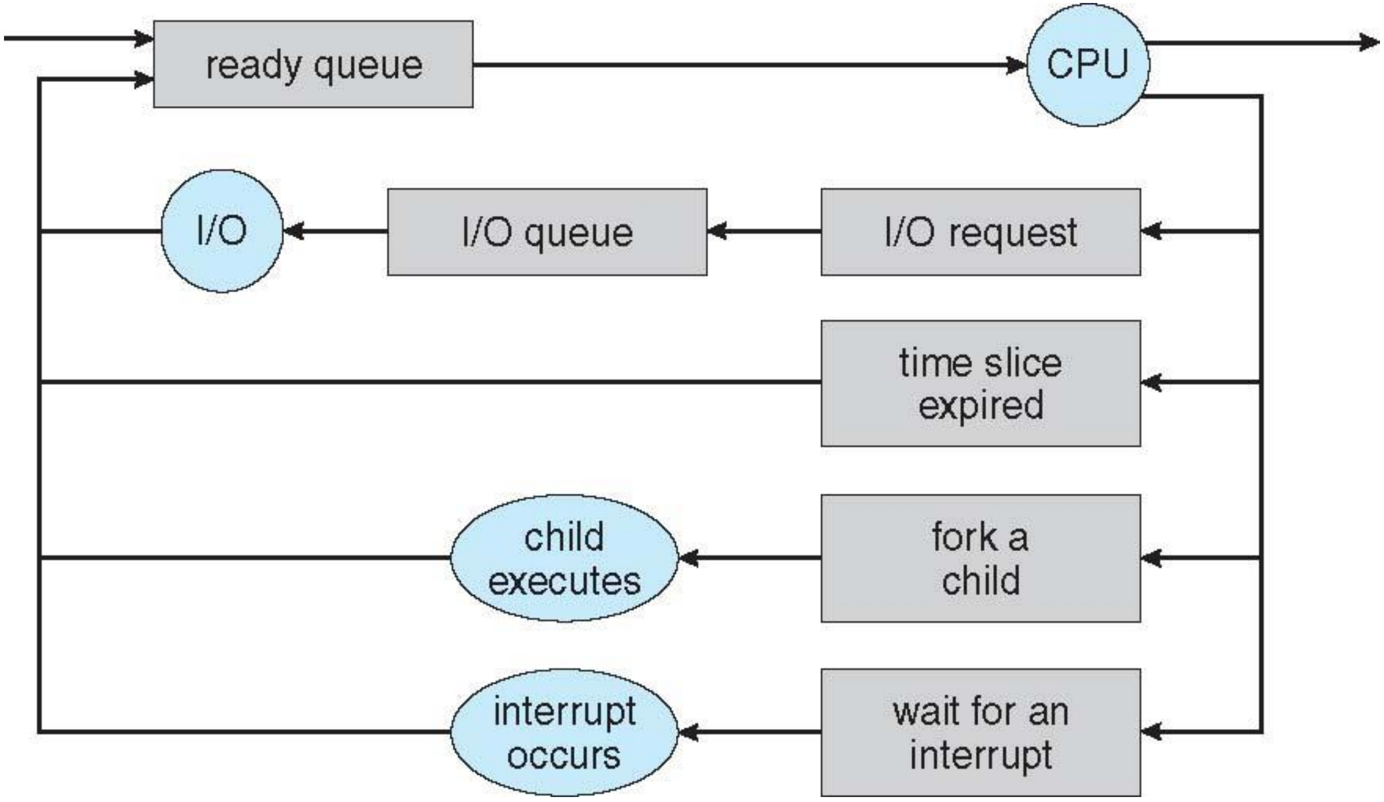
- Multiprogramming: Maximize CPU use,
- Multi-tasking: quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready 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
  - Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

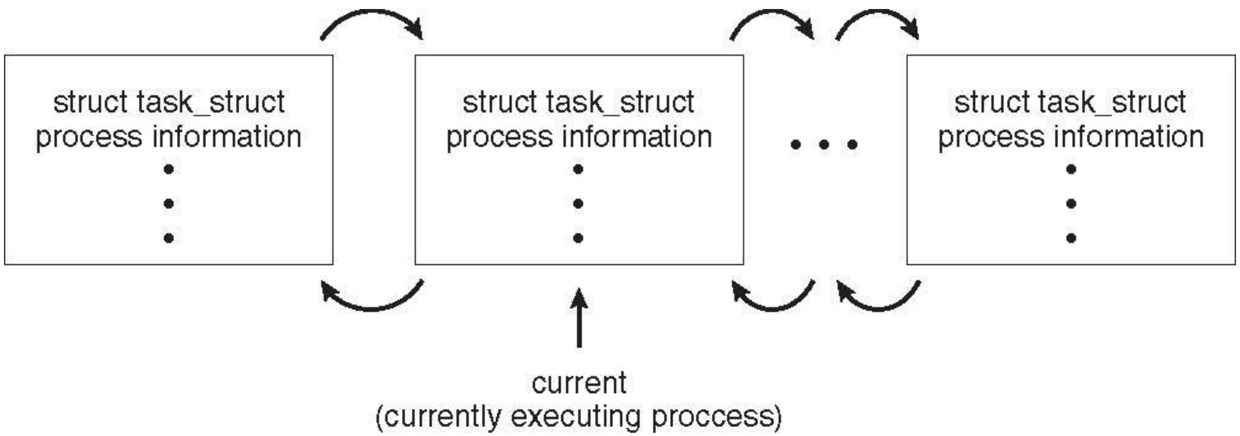
- **Queueing diagram** represents queues, resources, flows



Process Representation in Linux

Represented by the C structure `task_struct`

```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



Operations on Processes

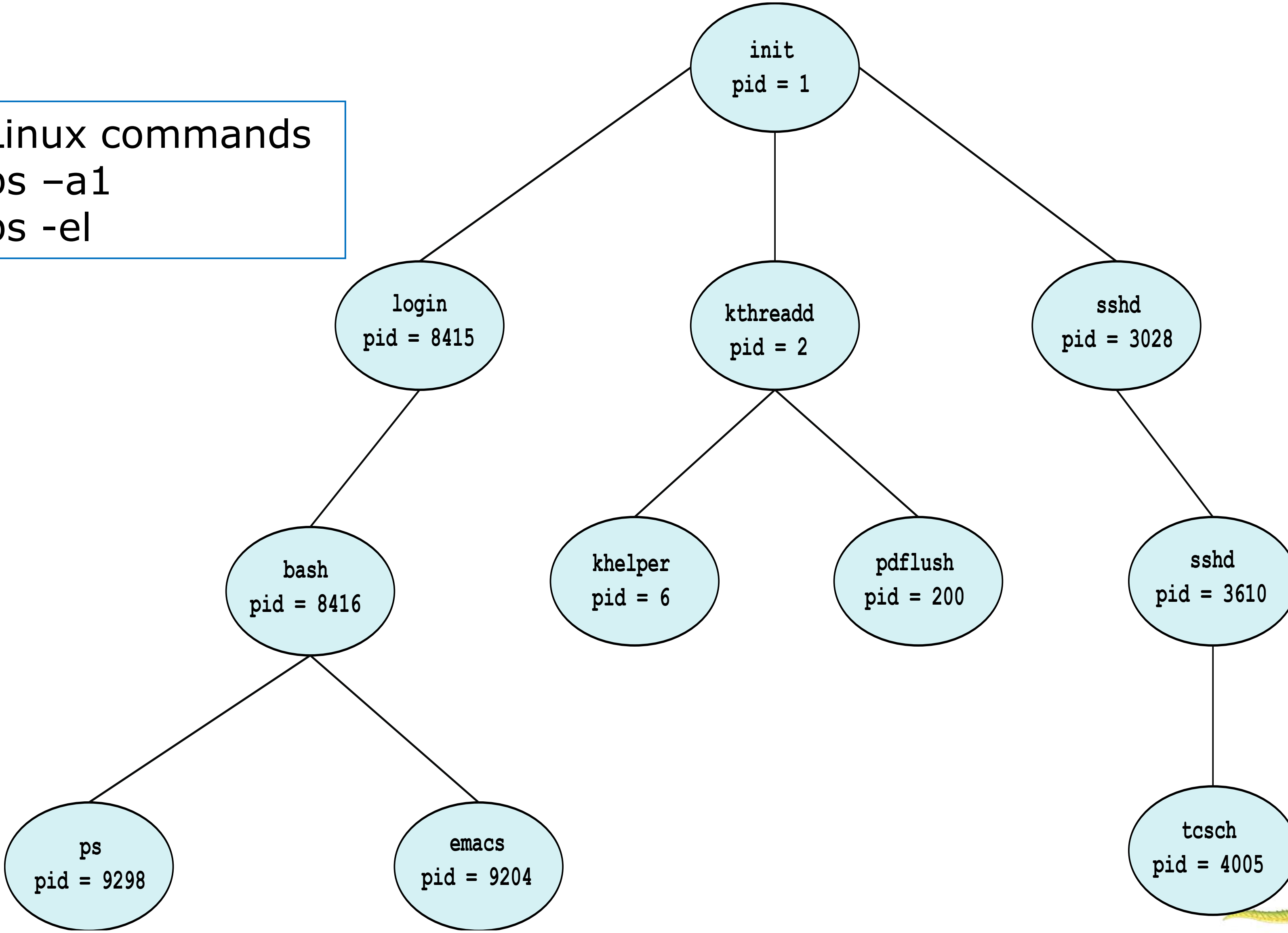
- System must provide mechanisms for:
  - process creation,
  - process termination,
  - and so on as detailed next

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

A Tree of Processes in Linux

Linux commands
ps –a1
ps -el



Process Creation (Cont.)

- Address space
  - Child duplicate of parent (same program and data)
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates new process

  - New process has copy of the address space of the original process.

  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program

  - Loads binary file into memory (destroying the

  memory image of the program containing the exec() system call)

C Program Forking Separate Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
    }
    else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
    }

    return 0;
}
```

The child process overlays its address space with the UNIX command /bin/ls (used to get a directory listing) using the execlp() system call (execlp() is a version of the exec() system call).

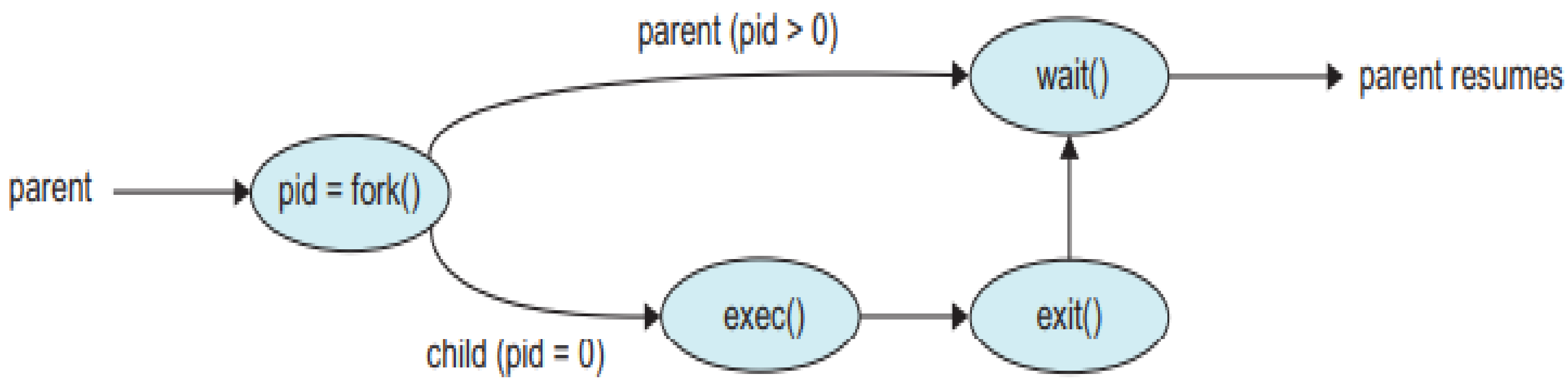


Figure 3.10 Process creation using the `fork()` system call.

Program for fork ()

```c
#include <unistd.h>

int main()
{
    int id;
    printf("Hello, World!\n");

    id = fork();
    if (id > 0) {
        /*parent process*/
        printf("This is parent section [Process id: %d].\n", getpid());
    }
    else if (id == 0) {
        /*child process*/
        printf("fork created [Process id: %d].\n", getpid());
        printf("fork parent process id: %d.\n", getppid());
    }
    else {
        /*fork creation faile*/
        printf("fork creation failed!!!\n");
    }

    return 0;
}
```

Program for fork ( )

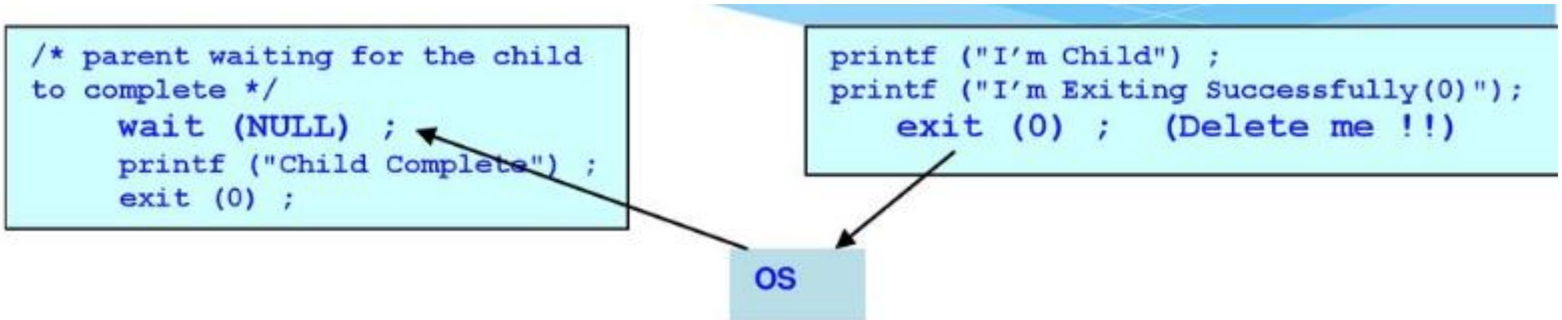
Output:

```
Hello, World!
This is parent section [Process id: 1252].
fork created [Process id: 1253].
fork parent process id: 1252.
```

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data (value) from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination



- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.

  - in Linux and UNIX systems, we can terminate a process by using the exit() system call, providing an exit status as a parameter:

/* exit with status 1 */

exit(1)

Process Termination

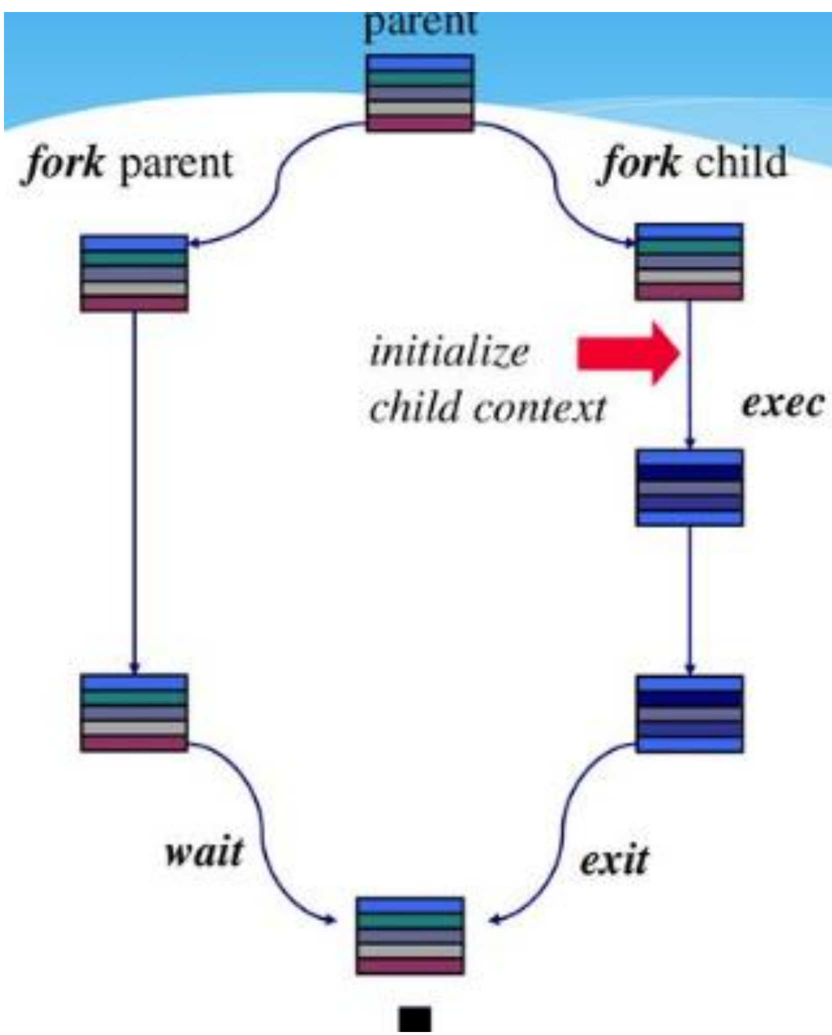
- The parent process may wait for termination of a child process by using the **`wait()`** system call**.**
- The call returns status information and the pid of the terminated process

```
pid_t pid ;
int status;

pid = wait(&status);
```

- Termination = Resource deallocation.
- However entry in the process table must remain there until the parent calls wait(), because process table contains process exit status.

UNIX fork /exec / exit / wait example



int pid = fork();

Create a new process that is a clone of its parent.

exec*("program" *[, argvp, envp]*);

Overlay the calling process virtual memory with a new program, and transfer control to it.

exit(status);

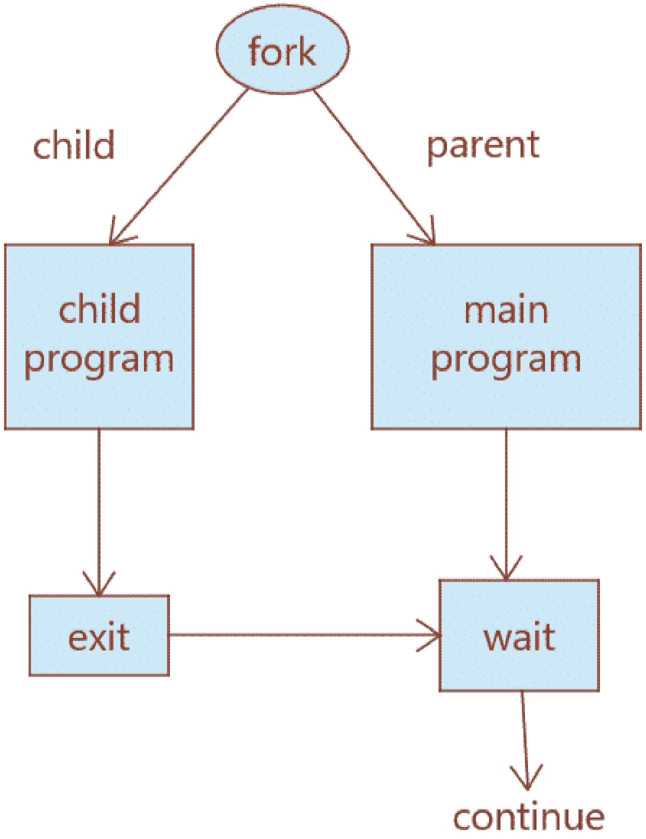
Exit with status, destroying the process

int pid = wait*(&status);

Wait for exit (or other status change) o a child.

Wait () & exit ()

- The fork() system function is defined in the headers **sys/types.h** and **unistd.h**
- In a program where you use fork, you also have to use wait() system call. Wait() system call is used to wait in the parent process for the child process to finish.
- To finish a child process, the exit() system call is used in the child process. The wait() function is defined in the header **sys/wait.h** and the exit() function is defined in the header.



Wait( )

```
int main()
{
        pid_t p;
        printf("before fork\n");
        p=fork();
        if(p==0)//child
        {
                printf("I am child having id %d\n",getpid());
                printf("My parent's id is %d\n",getppid());
        }
        else//parent
        {
                wait(NULL);
                printf("My child's id is %d\n",p);
                printf("I am parent having id %d\n",getpid());
        }
        printf("Common\n"); only after the child has finished.
}
```

Output

Output

```
baljit@baljit:~/cse325/Process$ ./a.out
before fork
I am child having id 458
My parent's id is 457
Common
My child's id is 458
I am parent having id 457
Common
```

Child

Parent

Zombie Process

- When a process terminates, its resources are deallocated by OS. However its entry in the process table must remain there until parent calls wait(), because process table contain's process exit status.
- A child process always first becomes a zombie before being removed from the process table.

```
1061  1060 Ss   -bash
1077     2 S    [kworker/0:1]
1117     2 S    [kworker/0:0]
1119     2 S    [kworker/0:2]
1120  1061 S+   ./a.out
1121  1120 Z+   [a.out] <defunct>
1122  1120 S+   sh -c ps -eo pid,ppid,stat,cmd
1123  1122 R+   ps -eo pid,ppid,stat,cmd
```

Note: Kernel sends a SIGCHLD signal to the parent process to indicate that the child process has ended

Zombie

- A process that has terminated, but whose parent has not yet called wait(), is known as a zombie process.
- A zombie process refers to any process that is essentially removed from the system as ‘defunct’, but still somehow resides in the processor’s memory as a ‘zombie’.

C program for zombie process

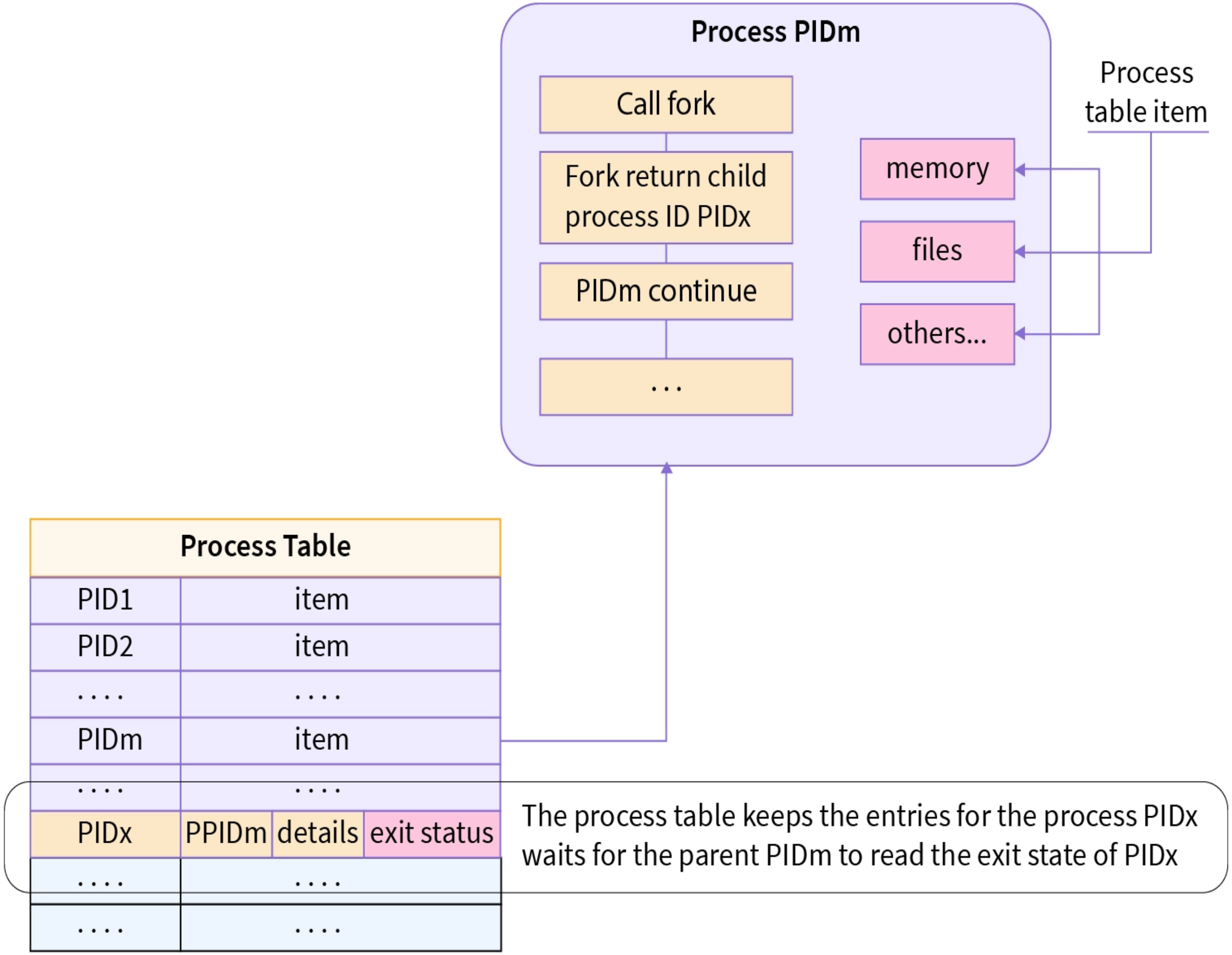
```
int main()
{
   pid_t t;
   t=fork();
   if(t==0)
   {
       printf("Child having id %d\n",getpid());
    }
    else
    {
        printf("Parent having id %d\n",getpid());
        sleep(15); // Parent sleeps. Run the ps command during this time
    }
}
```

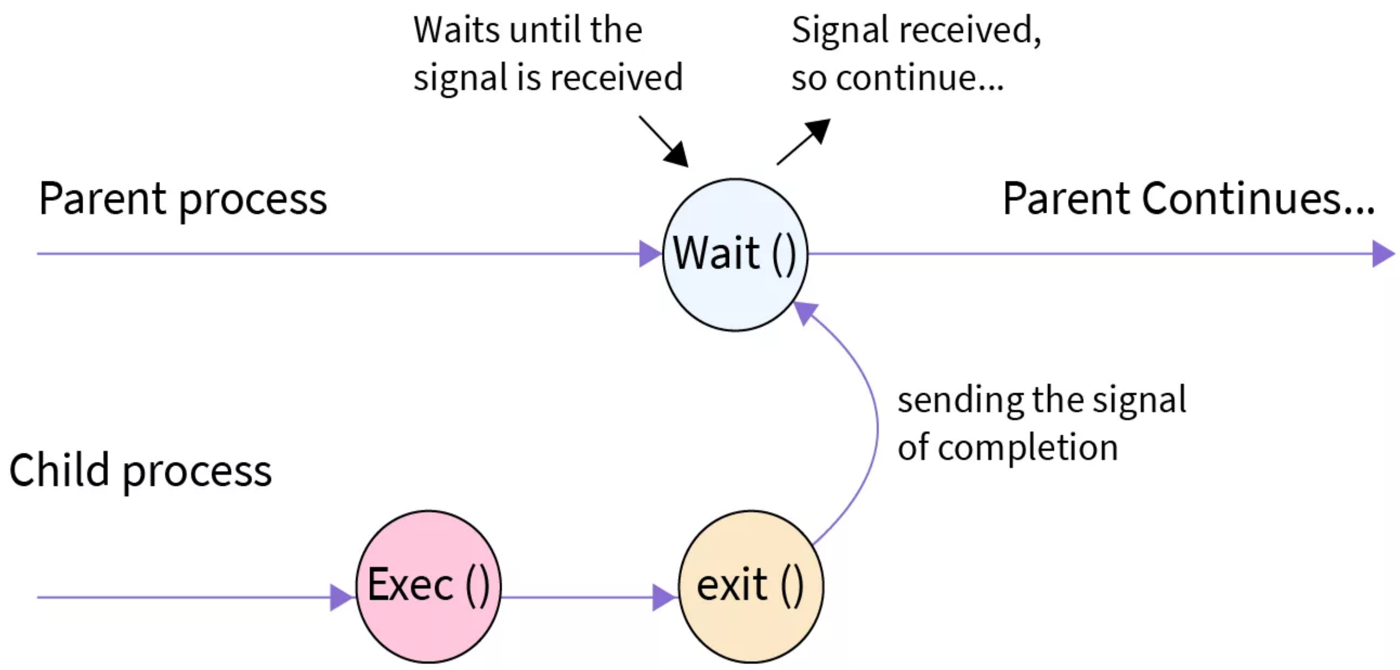
Output

```
user@LPU:~$
 I AM A PARENT having id 3687
I AM A CHILD having id 3688
ps
  PID TTY          TIME CMD
 3033 pts/0    00:00:00 bash
 3687 pts/0    00:00:00 a.out
 3688 pts/0    00:00:00 a.out <defunct>
 3689 pts/0    00:00:00 ps
user@LPU:~$
```

How to prevent the creation of a Zombie Process?

- In order to prevent this use the wait() system call in the parent process.
- The wait() system call makes the parent process wait for the child process to change state.


Process PIDm
Call fork
Fork return child process ID PIDx
PIDm continue
. . .
memory
files
others...
Process table item
Process Table
PID1
item
PID2
item
. . . .
. . . .
PIDm
item
. . . .
. . . .
PIDx
PPIDm
details
exit status
. . . .
. . . .
. . . .
. . . .
The process table keeps the entries for the process PIDx waits for the parent PIDm to read the exit state of PIDx


Waits until the
signal is received
Signal received,
so continue...
Parent process
Wait ()
Parent Continues...
sending the signal
of completion
Child process
Exec ()
exit ()

Before exit() system call

Process table		
Parent PID	→	Parent PCB
...	...	...
Child ID	→	Child PCB
...	...	...

After exit() system call

Process table		
Parent pID	→	Parent PCB
...	...	...
Child pID	→	Child PCB
...	...	...

Zombie process

its indicates that the child process is dome with it’s execution & entered into ‘Zombie State’

After exit() system call

Process table		
Parent PID	→	Parent PCB
...	...	...
~~Child PID~~	→	~~Child PCB~~
...	...	...

Zombie Gone

Once the wait() system call is called by the parent, it reads the exit status of the child process and reaps it from the process table

Orphan Process

- A process whose parent process no more exists i.e. either finished or terminated without waiting for its child process to terminate is called an orphan process.
- However, the orphan process is soon adopted by **init** process, once its parent process dies.

C Program to demonstrate Orphan process

```
int main()
{
        // Create a child process
        int pid = fork();
        if (pid > 0)
                printf("in parent process");

        // Note that pid is 0 in child proce and negative if fork() fails
        else if (pid == 0)
        {
                sleep(30);
                printf("in child process");
        }
        return 0;
}
```

Zombie & Orphan process

- If no parent waiting (did not invoke **`wait()`**) process is a **zombie**
- If parent terminated without invoking **`wait`** , process is an **orphan**

Zombie and Orphan Process

- How can you identify the existence of a zombie process in the system?

- Create a scenario where a parent has two child process C1 and C2 such that C1 becomes a zombie while C2 becomes an orphan process.

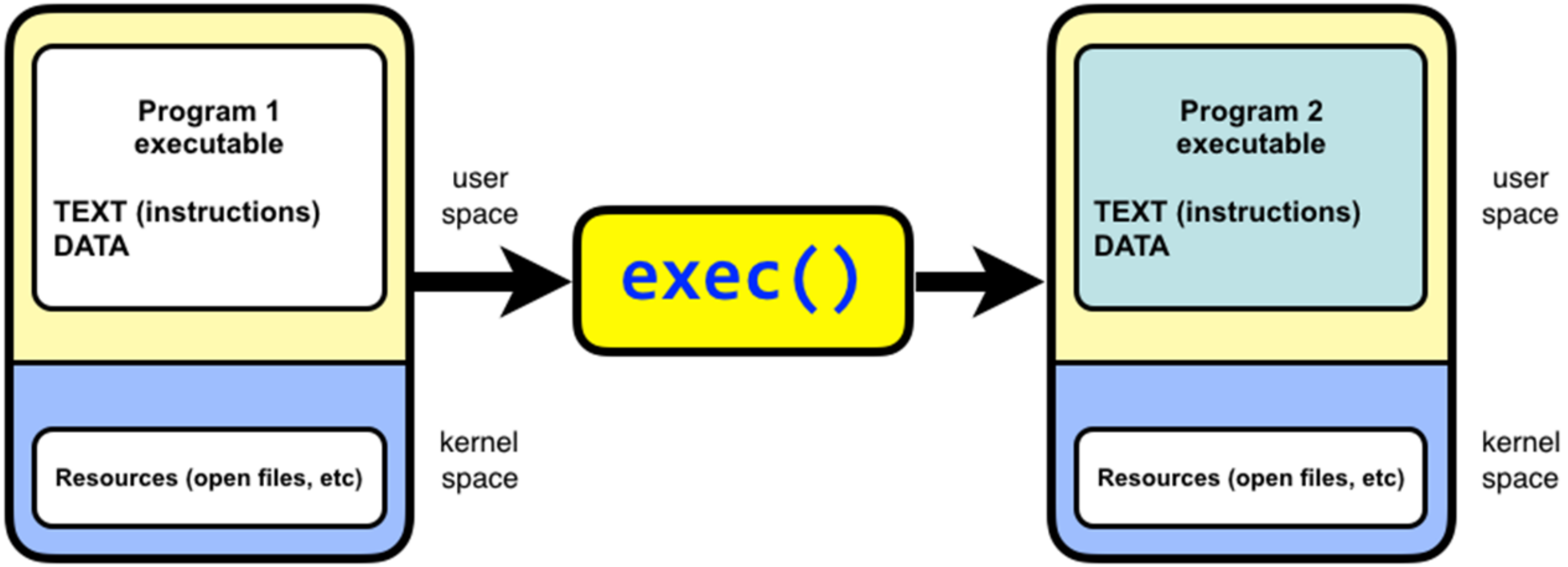
Both Zombie & orphan in a program

```
#include <stdio.h>
#include <unistd.h>
int main() {
   int x = fork(); //create child process
   if (x > 0) //if x is non zero, then it is parent process
      printf("Parent , PID is : %d\n", getpid());
   else if (x == 0) {
      sleep(5); me times
      x = fork();
      if (x > 0) {
         printf("Child- PID :%d and PID of parent : %d\n", getpid(), getppid());
         while(1)
            sleep(1);
         printf(" Child- PID of parent : %d\n", getppid());
      }else if (x == 0)
      printf("grandchild -> PID of parent : %d\n", getppid());
   }
   return 0;
}
```

Linux Exec System Call

- The exec system call is used to execute a file which is residing in an active process.
- Replaces the old file or program from the process with a new file or program.
- When a process calls exec, all code (text) and data in the process is lost and replaced with the executable of the new program.
- PID of the process is not changed but the data, code, stack, heap, etc. of the process are changed and are replaced with those of newly loaded process. The new process is executed from the entry point.

exec ()


Program 1
executable
TEXT (instructions)
DATA
user
space
Resources (open files, etc)
kernel
space
exec()
Program 2
executable
TEXT (instructions)
DATA
user
space
Resources (open files, etc)
kernel
space

Exec ()

- It should be noted here that these functions have the same base *exec* followed by one or more letters.
- **execl**
- **execle**
- **execlp**
- **execv**
- **execve**
- **execvp**

exec()

- **e:** It is an array of pointers that points to environment variables and is passed explicitly to the newly loaded process.
- **l:** l is for the command line arguments passed a list to the function
- **p:** p is the path environment variable which helps to find the file passed as an argument to be loaded into process.
- **v:** v is for the command line arguments. These are passed as an array of pointers to the function.

Difference

Parameters	fork()	exec()
Basics	It is an operation used in the UNIX OS that lets a process create its copies that do not replace the original process.	It is also an operation used in the UNIX OS, but it creates child processes that replace the parent process or the previous process.
Types of Processes	Both parent and child processes exist in the system after making this function call.	After calling the exec(), only the child process exists. No parent process is present since the child process replaces it.
Result and Similarity	The child process created via the fork() system call is always similar to its parent process. They both coexist.	The child process created via the exec() system call replaces its parent process.
Address Space	Since the parent and child process coexist after the fork() system call, they reside in different address spaces.	Since the child process replaces the parent process, it also replaces its address space. Thus, they both have the same address space in the system.

GATE CS 2008

A process executes the following code

```
for (i = 0; i < n; i++) fork();
```

The total number of child processes created is

(A) n

(B) $2^n - 1$

(C) 2^n

(D) $2^{(n+1)} - 1$

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in

End of Chapter