CS 3113 Introduction to Operating Systems

Topic #3. Processes

Outline

- 3.1 Process Concept
- 3.2 Process Scheduling
- 3.3 Operations on Processes
- 3.4 Interprocess Communication

Process Concept

- The term "process" was first used by the designers of the MULTICS in 1960's. Since then, the term process, used somewhat interchangeably with 'task' or 'job'. The process has been given many definitions for instance
  - A program in Execution.
  - An asynchronous activity.
  - The 'animated spirit' of a procedure in execution.
  - The entity to which processors are assigned.
  - The 'dispatchable' unit.

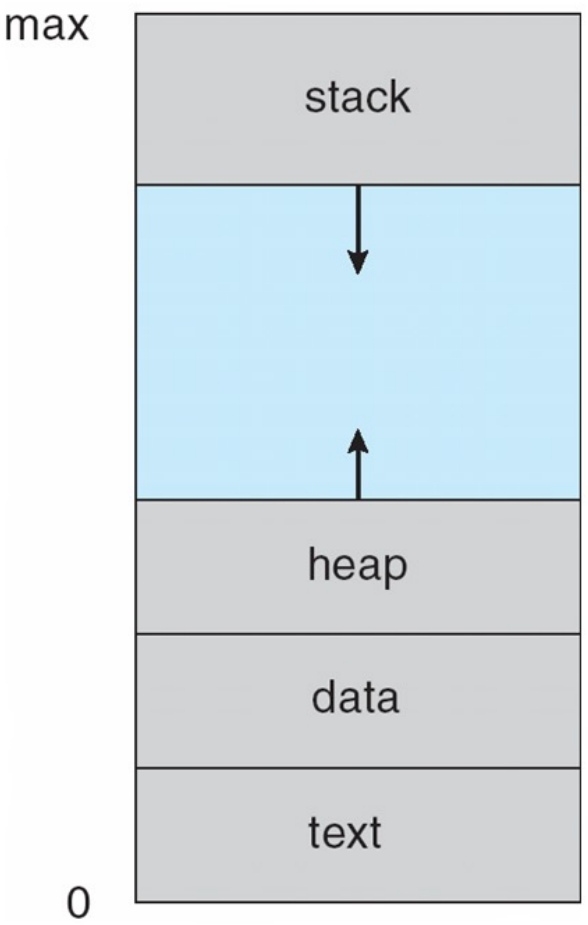
Process VS. Program

- Program is ***passive*** entity stored on disk (***executable file***), process is ***active***
  - A program becomes a process when an executable file is 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

Processes and Memory

On process creation, the process is effectively given its own memory space

- Text: storage of code
- Data: global variables (preallocated space)
- Heap: dynamically allocated space
- Stack: local variable storage



Stack and Heap

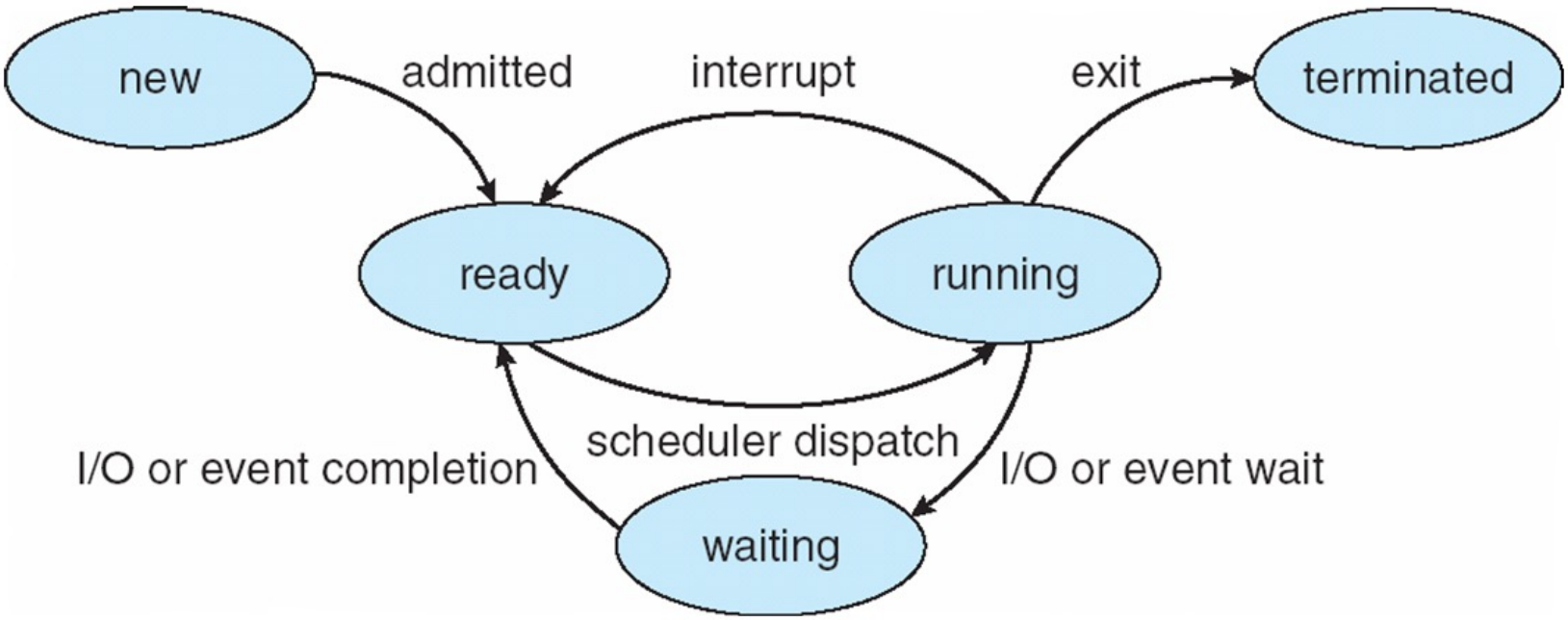
- Stack grows downward with each nested function call
  - Local variables, register state, return memory address
- Heap
  - Storage of dynamically allocated items that must be persistent across function calls (and returns from function calls)
  - OOP languages: object instantiation is done in the heap

Process State

A process is in exactly one state at any instant in time:

- **new**: The process is being created
- **running**: Instructions are being executed by the CPU
- **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

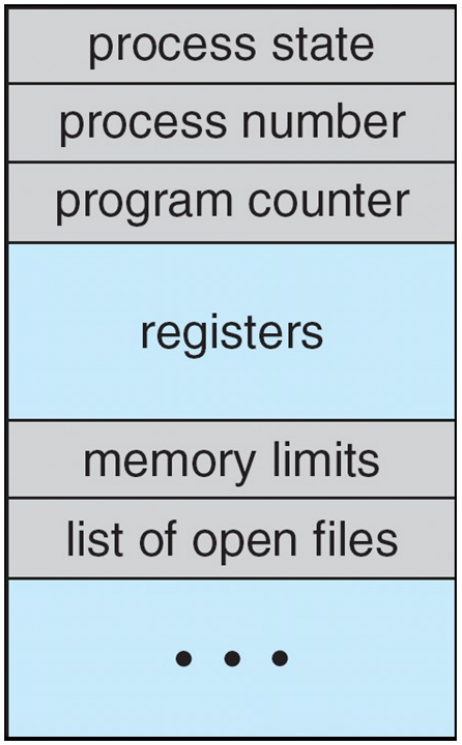
Diagram of Process State



Kernel Data Structure: Process Control Block

Stores information about the running process:

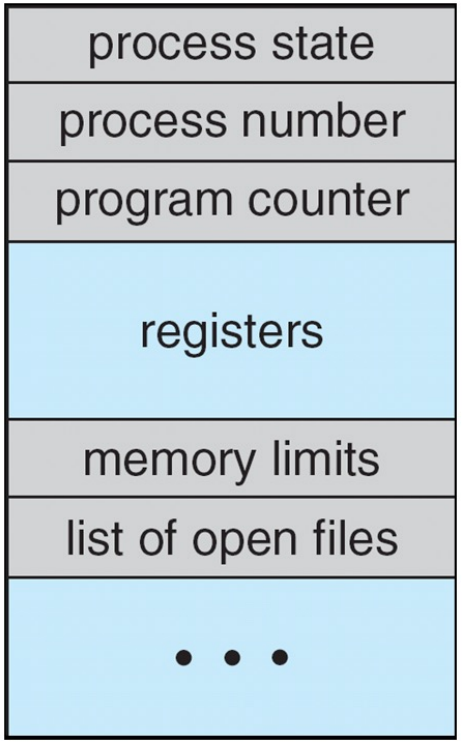
- Process state – running, waiting, etc.
- *Program counter – location of instruction to next execute
- *CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers



Kernel Data Structure: Process Control Block

continued:

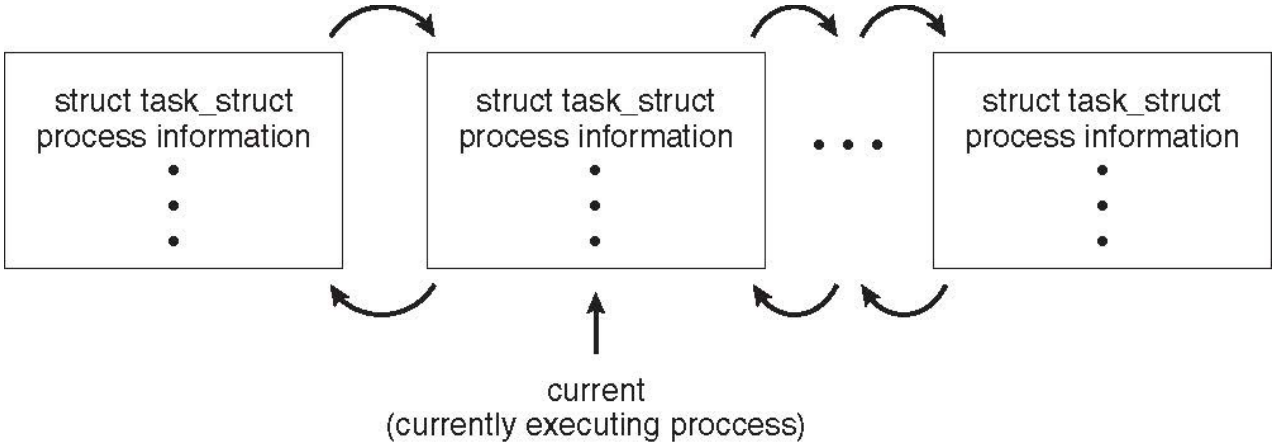
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



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 */
```



Outline

- 3.1 Process Concept
- 3.2 Process Scheduling
- 3.3 Operations on Processes
- 3.4 Interprocess Communication

Process Scheduling

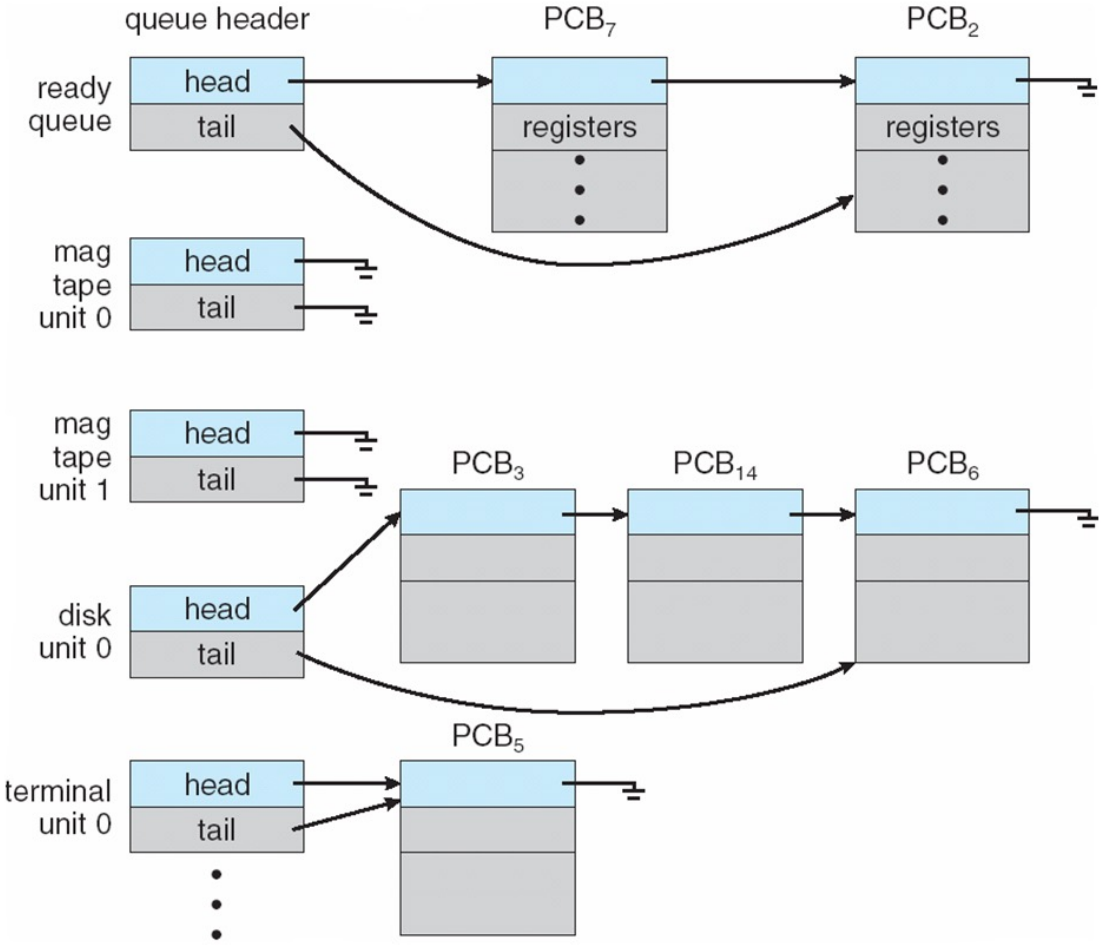
- Our goals are to:
  - Maximize CPU use
  - Give processes the CPU time that they need
- **Process scheduler** selects among available processes for next execution on CPU

Process Scheduler

Maintains **scheduling queues** of processes

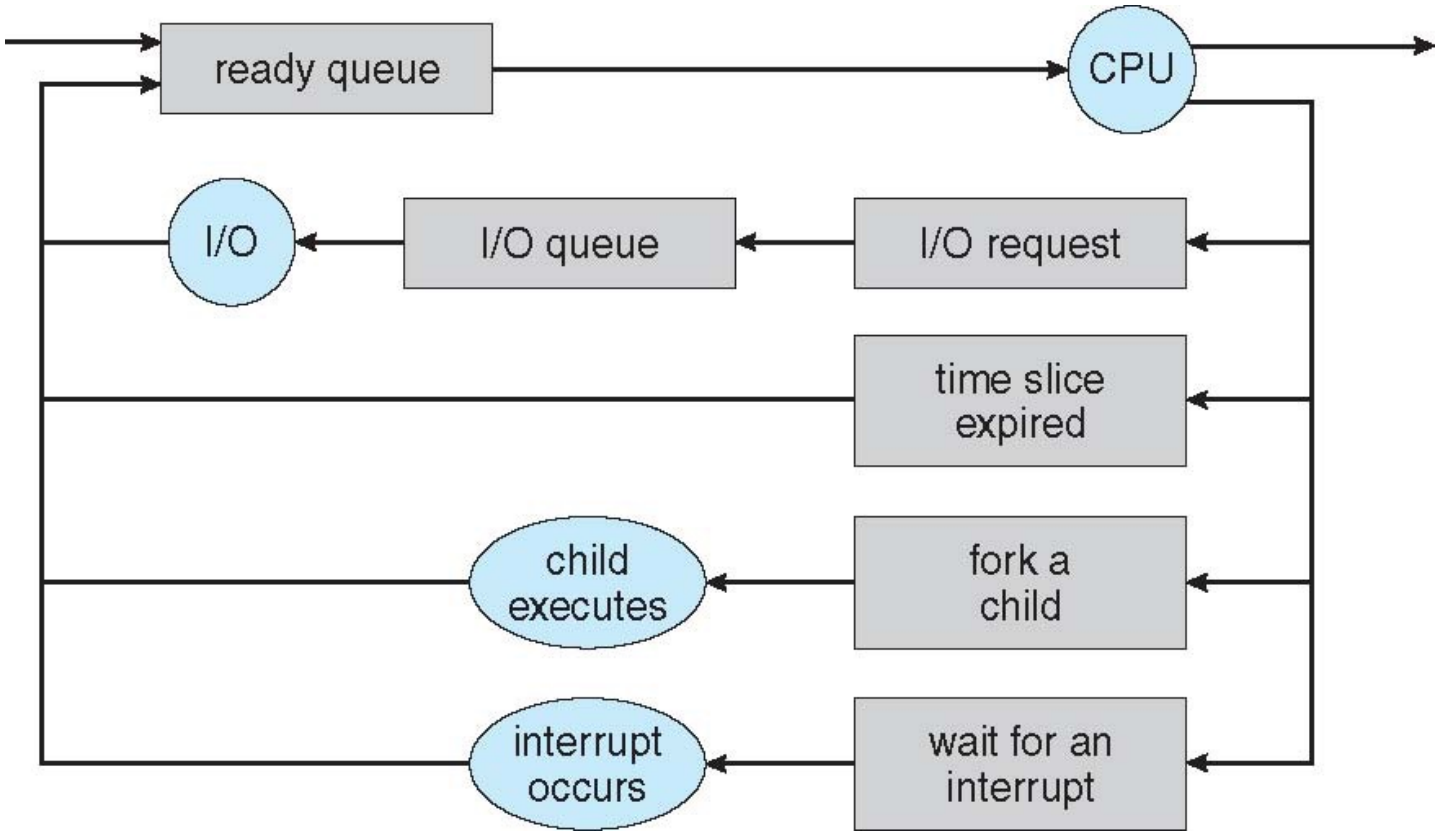
- **Job queue:** set of all processes in the system
- **Ready queue:** set of all processes residing in main memory, ready and waiting to execute
- **Device queues:** set of processes waiting for an I/O device
- Processes migrate among the various queues, depending (in part) on their state

Ready Queue and Various I/O Device Queues



Process Scheduling

Queueing diagram represents queues, resources, flows



Scheduler Components

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), so it must be fast

Scheduler Components

Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue

- Long-term scheduler is invoked infrequently (seconds, minutes), so it may respond slowly
- The long-term scheduler controls the degree of multiprogramming
- Most important in (older) resource-bound systems

Scheduler Components

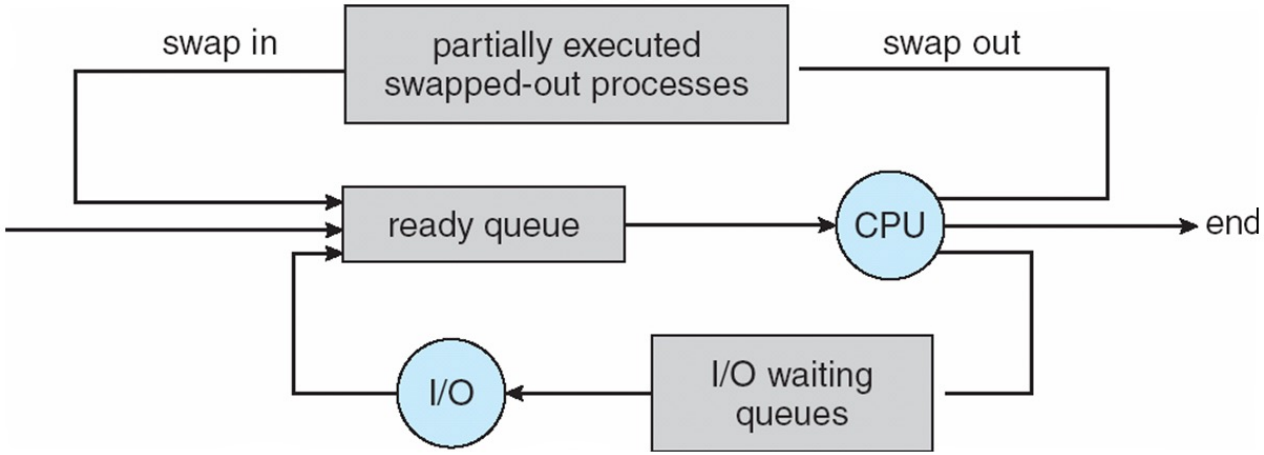
Processes can be described as either:

- **I/O-bound**: spends more time doing I/O than computations, many short CPU bursts
- **CPU-bound**: spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good process mix
  - Goal: keep both I/O and CPU resources as busy as possible

Medium Term Scheduling

Medium-term scheduler can be added if degree of multiple programming needs to decrease

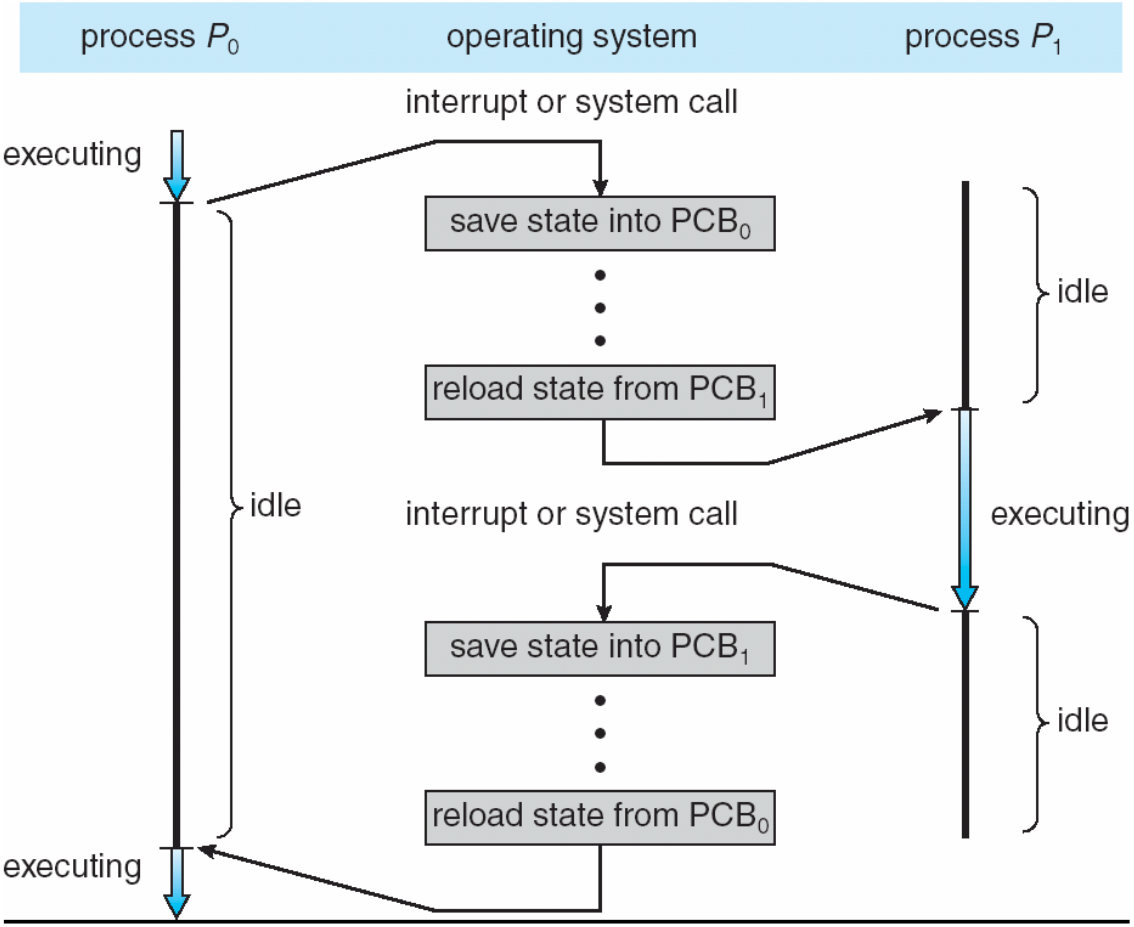
- Remove process temporarily from memory, store on disk, bring back in from disk to continue execution: **swapping**



Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run; others are suspended
- Due to screen real estate and user interface limits, iOS provides for:
  - Single foreground process: controlled via user interface
  - Multiple background processes: in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Split-screen: running two foreground apps at the same time
- Android runs foreground and background processes, with fewer limits

CPU Switching from One Process to Another



Context Switching

- 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 is dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU
    - Allows multiple contexts to be loaded at once

Outline

- 3.1 Process Concept
- 3.2 Process Scheduling
- 3.3 Operations on Processes
- 3.4 Interprocess Communication

Operations on Processes

- System must provide mechanisms for:
  - Process creation
  - Process termination

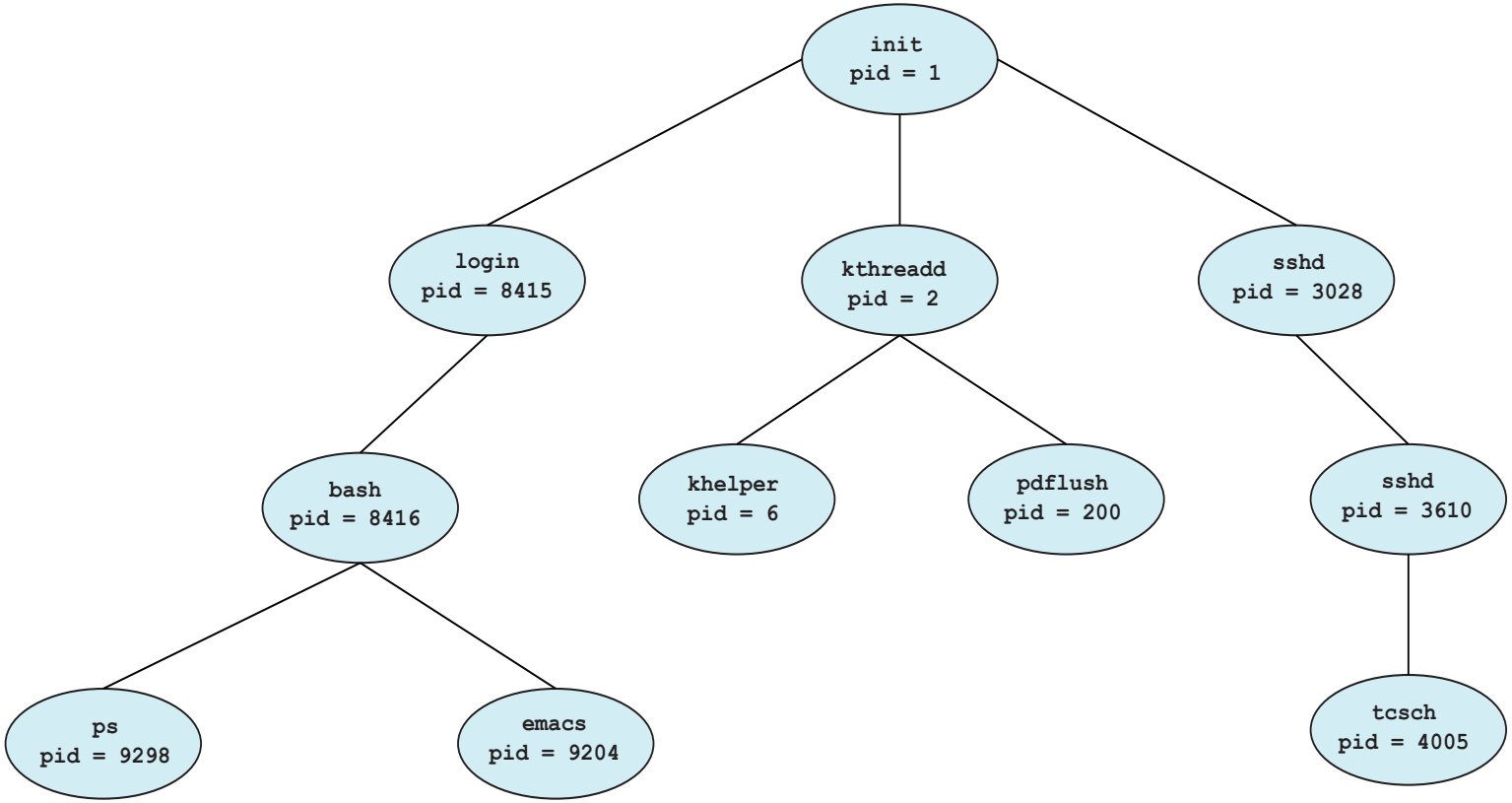
Process Creation

- **Parent** process creates **child** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)

Process Creation

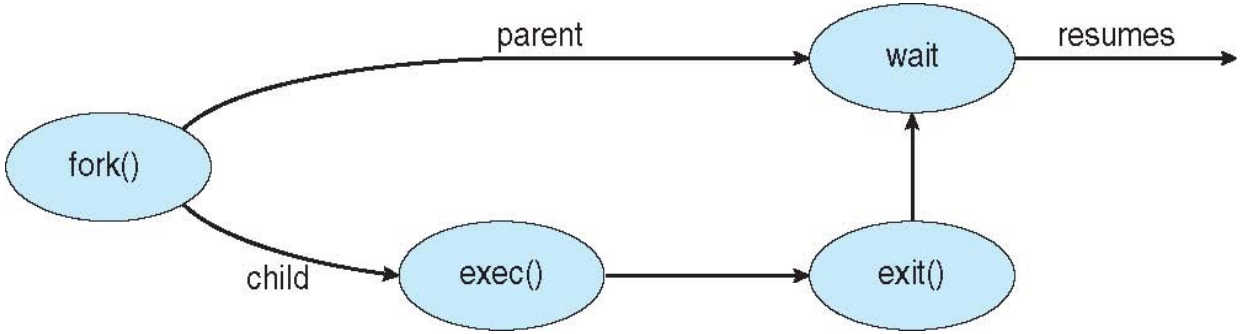
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Tree



Process Creation (Cont'd)

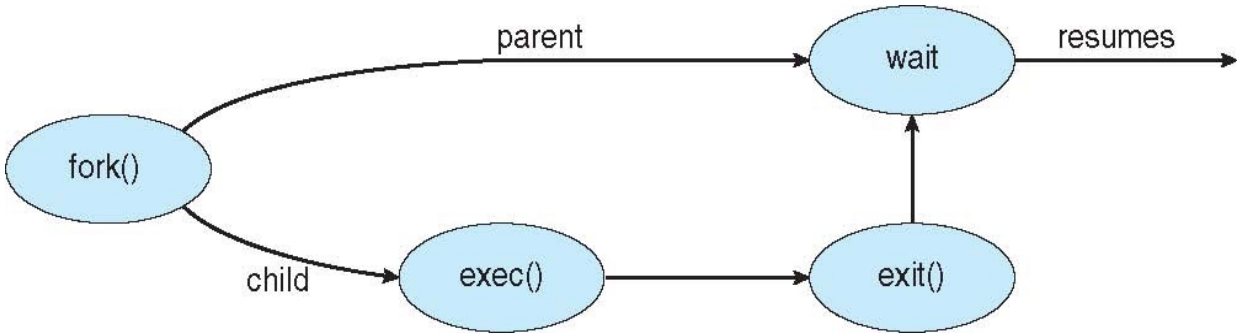
- Address space
  - Child duplicate of parent
  - Child then has a program loaded into it
- UNIX examples
  - fork() system call creates new process
  - exec() system call used after a fork() to replace the process' memory space with a new program



fork()

fork() creates an identical process to the one that called fork()

- Same program
- Same state, including open file descriptors
- Exception: fork() returns 0 to the child; and a positive integer to the parent
- Processes execute in parallel



C Program Forking a Child Process

```c
#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;
}
```

exit.c

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

int main(void)
{
    pid_t pid = fork();
    if (pid != 0) {
        wait(NULL);
    }
    printf("Hello World: %d\n", pid);
    return 0;
}
```

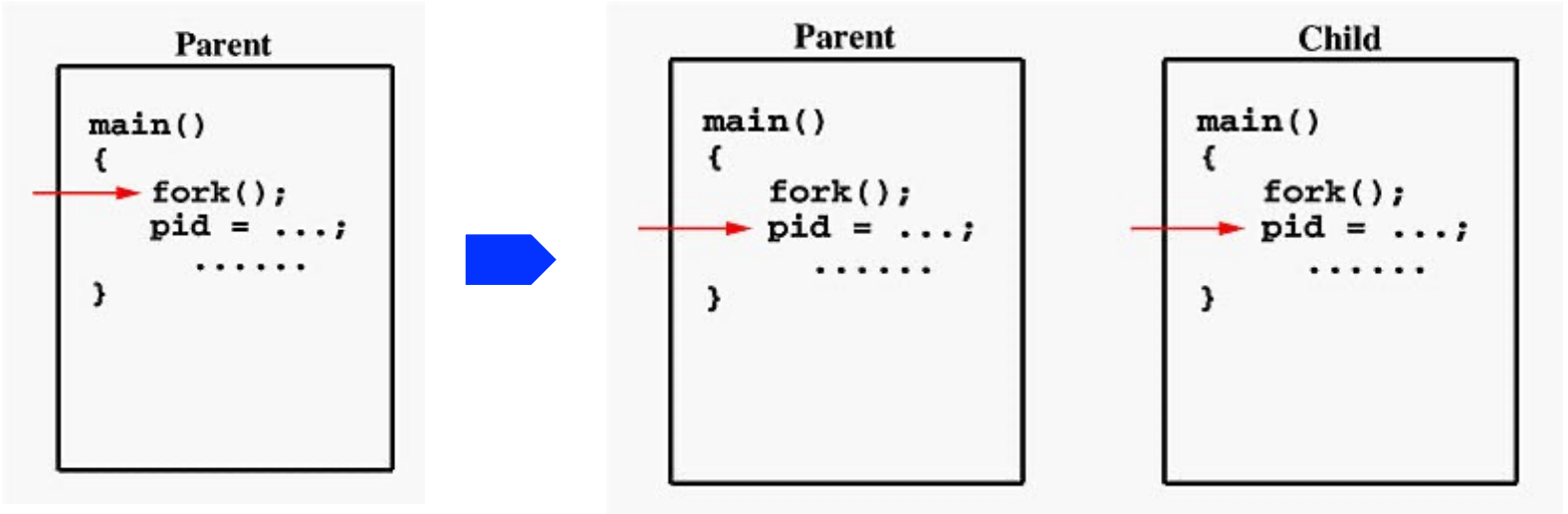
```
Hello World: 0
Hello World: 59510
```

exer1.c

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


int main(void)
{
    int stuff = 5;
    pid_t pid = fork();
    printf("The last digit of pi is %d\n", stuff);
    if (pid == 0)
        stuff = 6;
    return 0;
}
```

```
The last digit of pi is 5
The last digit of pi is 5
```



After **fork(),** Unix will

- make two identical copies of address spaces
- both processes will start their execution at the next statement following the **fork()** call

exer2.c

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

int main() {
      printf("We are good!\n PID=%d\n", getpid());
    fork();
    fork();
    fork();
    printf("Hello, World!\n PID=%d\n", getpid());

    return 0;
}
```

```
We are good!
 PID=59739
Hello, World!
 PID=59739
Hello, World!
Hello, World!
 PID=59742
 PID=59741
Hello, World!
 PID=59740
Hello, World!
 PID=59744
Hello, World!
 PID=59745
Hello, World!
 PID=59743
Hello, World!
 PID=59746
```

execlp

```
execlp(char *command, char *argv0, char *argv1, … NULL)
```

- Replaces the currently executing program with a new program & begins execution
- command is a string that references an executable file
  - Can be absolute or relative path
  - Relative path: use the **path** environment variable to find the executable
- argv0, argv1, … are arguments to be passed to the executable
  - Don't forget the NULL at the end!
- If this function is successful in finding the specified executable, it does not return!

ex1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    printf("Hello, World!\n PID of ex1.c =%d\n", getpid());
    char *args[]={"Hello World", NULL};
    execv("./ex2",args);
    printf("Back to ex1.c\n");

    return 0;
}
```

ex2.c

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

int main() {
    printf("We are in ex2.c \n");
    printf("PID of ex2.c =%d\n", getpid());

    return 0;
}
```

```
% gcc ex1.c -o ex1
% gcc ex2.c -o ex2
% ./ex1

Hello, World!
 PID of ex1.c =59349
We are in ex2.c
PID of ex2.c =59349
```

Process Termination

Process executes last statement and then asks the operating system to delete it using the exit() system call.

- exit(-1): -1 is the exit code returned by the process
- Status data from child can be passed to the parent
  - `pid = wait(&status);`
  - status contains information about the reason for termination
- Process' resources are deallocated by operating system

Process Termination

Some operating systems do not allow child processes to exist if its parent has terminated (including Linux). If a process terminates, then all its children must also be terminated.

- If parent is executing, but not waiting (did not invoke **`wait()`**) and the child process ends, then the child process is a **zombie**
- If parent terminated without invoking **`wait`**(), the child process is an **orphan**
  - Orphans become children of the **init** process

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

int main(void)
{
    int stuff = 5;
    pid_t pid = fork();

    if (pid == 0)
    {
        stuff = 6;
        printf("\n  *** Child: Hello, World! PID of this process =%d\n", getpid());
    }

    printf("The pid value of of this process is %d\n", pid);
    return 0;
}
```

The **return code** for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent

```
The pid value of of this process is 68700

  *** Child: Hello, World! PID of this process =68700
The pid value of of this process is 0
```

Outline

- 3.1 Process Concept
- 3.2 Process Scheduling
- 3.3 Operations on Processes
- 3.4 Interprocess Communication

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
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Multiprocess Architecture – Chrome Browser

Many web browsers used to run as single process (some still do)

- If one web site causes trouble, the entire browser can hang or crash

Multiprocess Architecture – Chrome Browser

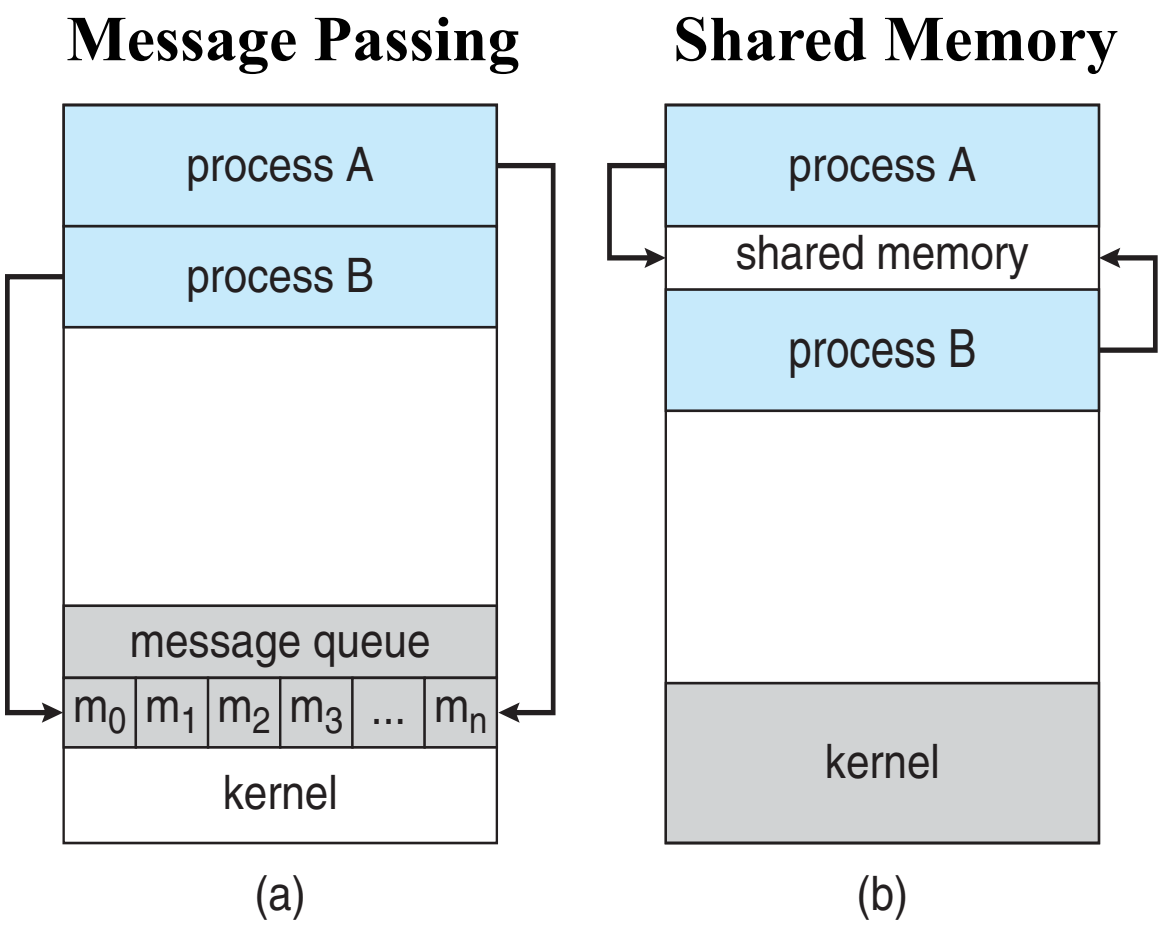
Google Chrome Browser is multiprocess with 3 different types of processes:

- **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

Interprocess Communication

- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communication Models



- Question: Assume the following program contains no syntax errors. As it executes it will create one or more processes. Simulate the execution of this program and show how processes are created.

```
#include<stdio.h>
main()
{
      int p1=1, p2=2, p3=3;
      p1 = fork();
      if(p2>0) p2 = fork();
      if(p1>0) p3 =fork();
      if(p1==0) printf("type 1 \n");
      if(p3!=0) printf("type 2 \n");
      if(p2!=0) printf("type 3 \n");
      if((p1>0)||(p2>0)||(p3>0))
      printf("type 4 \n");
      if((p2==0) && (p3==0)) printf("type
      5 \n");
}
```

Also answer the following question

How many processes are created, including the parent process?_____

How many times will this program print the following?

"Type 1" _____
"Type 2" _____
"Type 3" _____
"Type 4" _____
"Type 5" _____