Operating System
CS-2006
Lecture 4

Mahzaib Younas

Lecturer Department of Computer Science

FAST NUCES CFD

Process

- An operating system executes a variety of programs that run as a process.

- A program in execution
  - process execution must progress in sequential fashion
  - No parallel execution of instructions of a single process

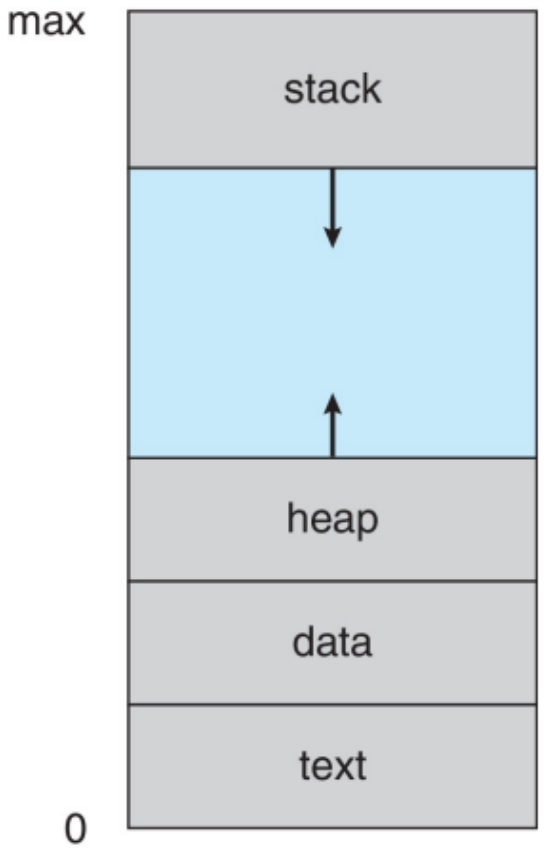
Parts of Process

A process have multiple parts

1.Text Section

- It contain the executable code

- Program Counter
  - The status of the current activity of a process is represente by the value of program counter and the contents of the processor's register
- Stack (containing temporary data)
  - Function parameters, return addresses, local variables
- Data Section
  - Global variables
- Heap
  - Dynamically allocated memory during program runtime

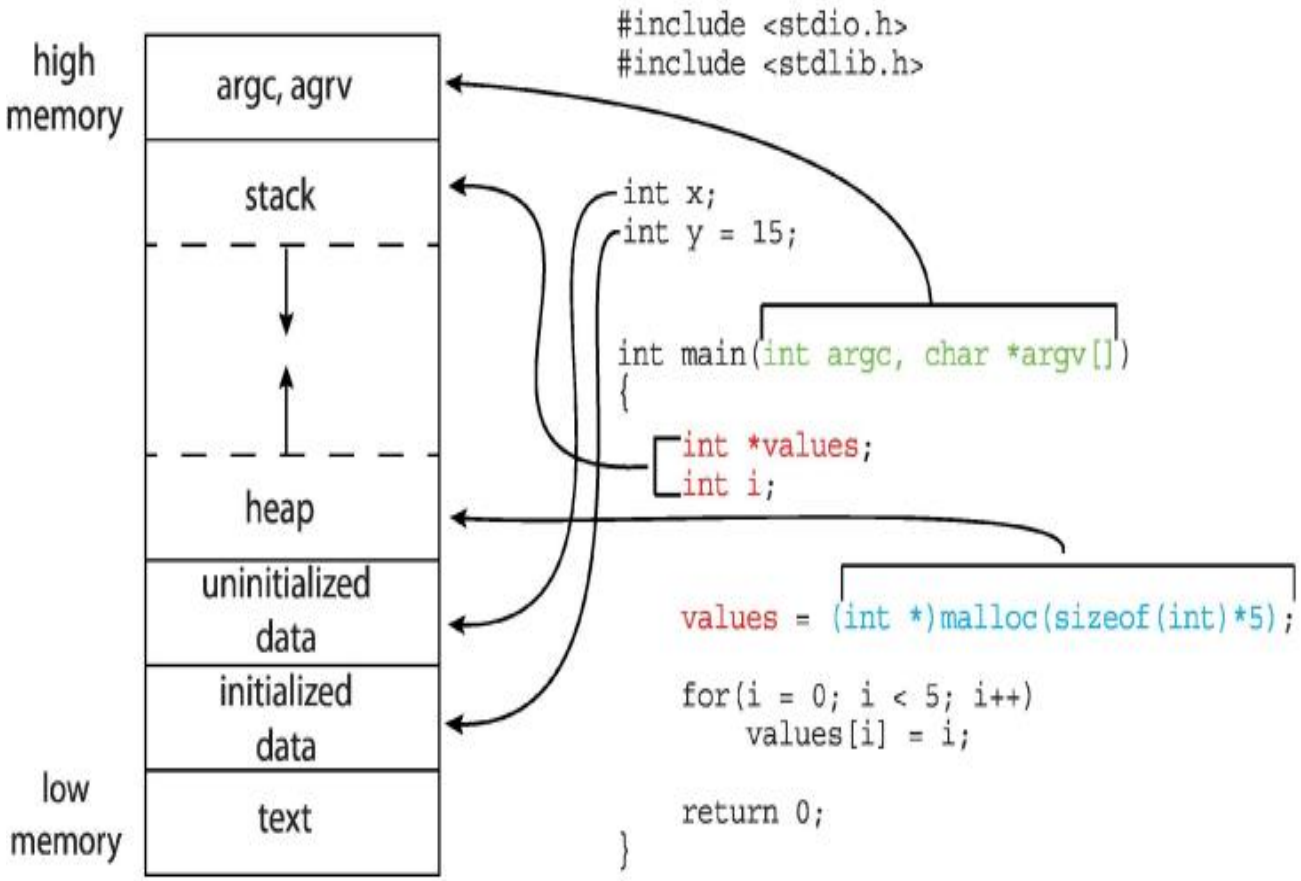


Process Memory

- Program is passive entity stored on disk (executable file); **process is active**
- Program becomes 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

Example:

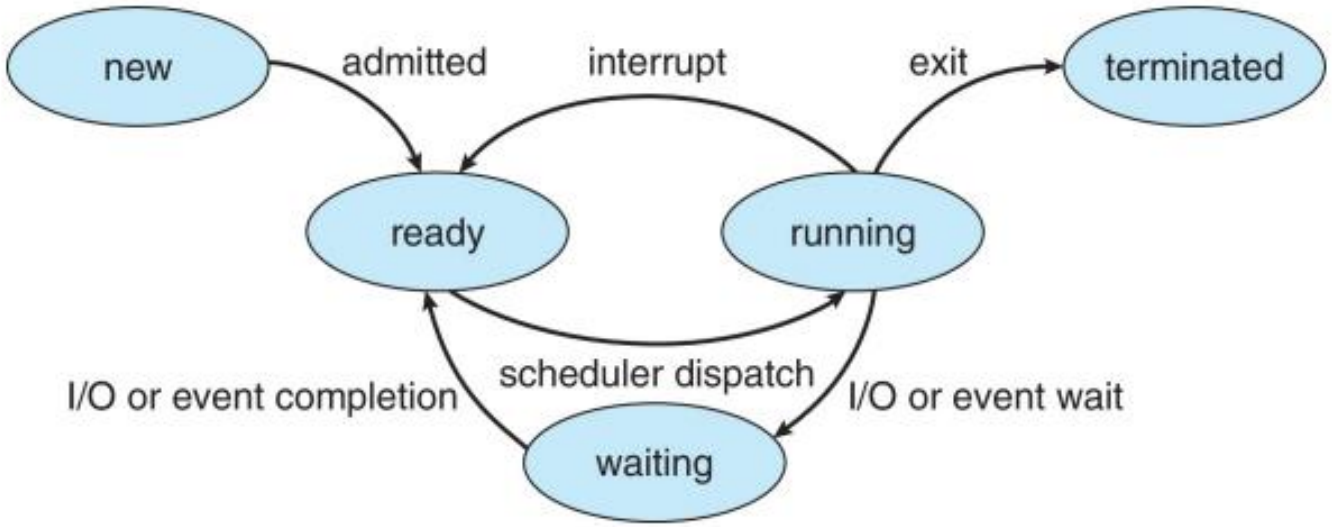
- The global data section is divided into different sections for
  - (a) initialized data
  - (b) uninitialized data
- A separate section is provided for the argc and argv parameters passed to the main() function.



Process State

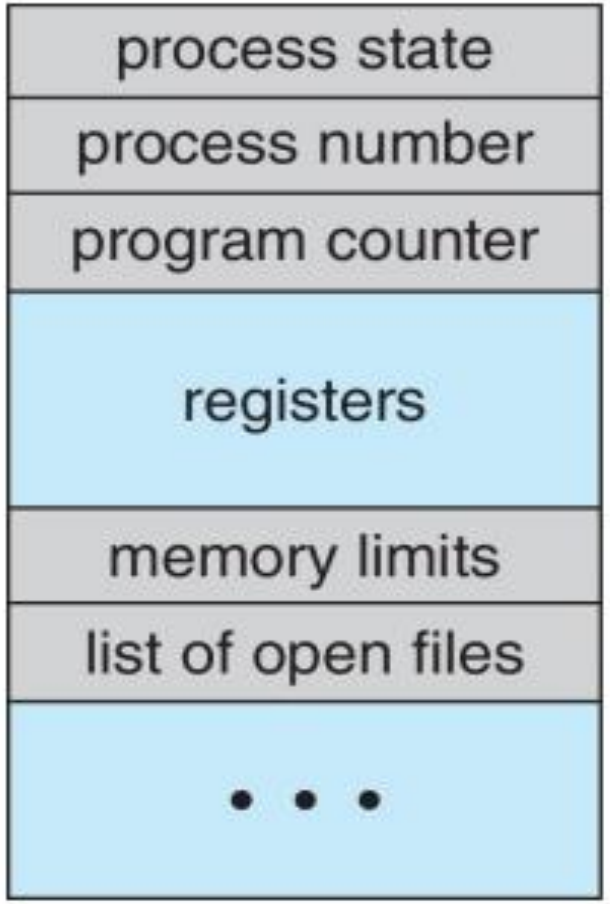
- New
  - The process is being created.
- Running
  - Instructions are being executed.
- Waiting
  - The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- Ready
  - The process is waiting to be assigned to a processor

Diagram:


new
admitted
interrupt
exit
terminated
ready
running
scheduler dispatch
I/O or event completion
I/O or event wait
waiting

Process Control Block (PCB)

- Information associated with each process
- PCB also called task control block



PCB (Cont…)

1. Process State

   Store information about the state of process. It may be new, ready, running, waiting, halted.

2. Program Counter

   The counter indicates the address of the next instruction to be executed for this process.

3. CPU Register

   The register may vary in number and type, depending on the computer architecture.

   contents of all process

PCB (Cont…)

4. CPU scheduling Information
   - priorities, scheduling queue pointers or any other scheduling parameters

5. Memory Management Information
   memory allocated to the process

6. Accounting Information
   CPU used, clock time elapsed since start, time limits

7. I/O Status Information
   I/O devices allocated to process, list of open files

Threads

- process has a single thread of execution
  - Consider having multiple program counters per process
  - Multiple locations can execute at once
- Single Thread
  - This single thread of control allows the process to perform only one task at a time.
- Multiple Thread
  - multiple threads of execution and thus to perform more than one task at a time.
- Example:
  - Word Processor

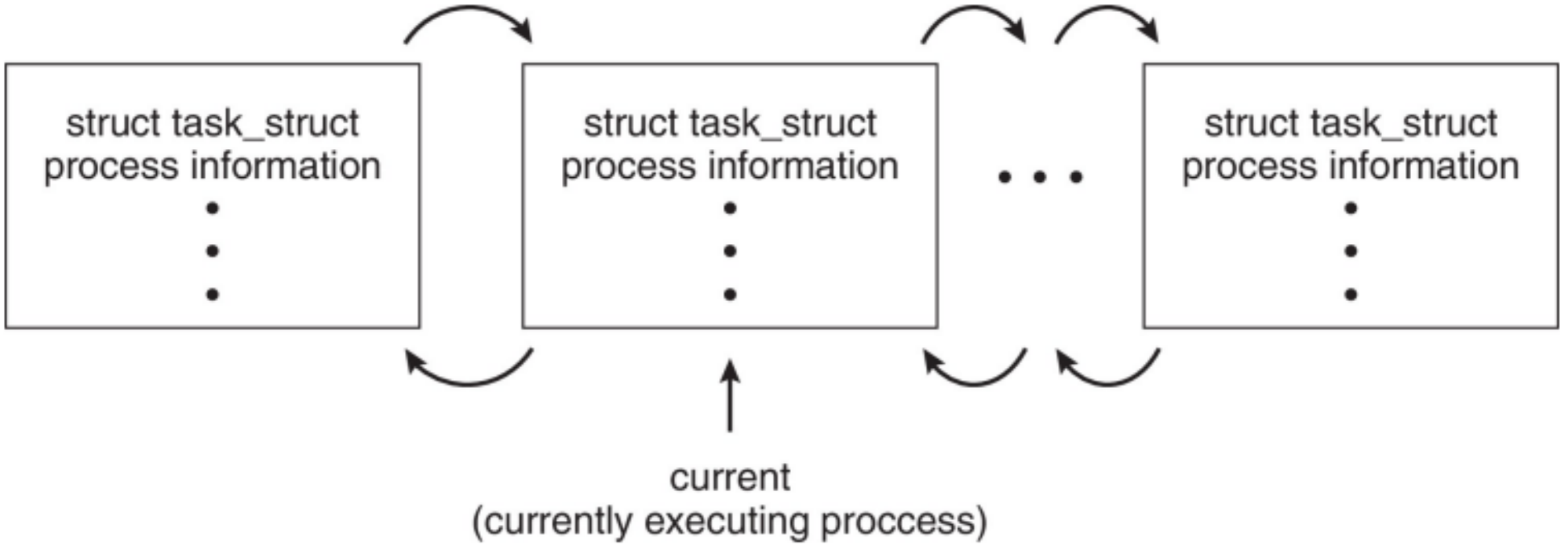
Process Scheduling

- Process scheduler selects among available processes for
  - next execution on CPU core
- Goal
  - Maximize CPU use, quickly switch processes onto CPU core
- Maintains scheduling queues of processes
  - Ready queue – set of all processes residing in main memory, ready and waiting to execute
  - Wait queues – set of processes waiting for an event (i.e., I/O)
- Processes migrate among the various queue

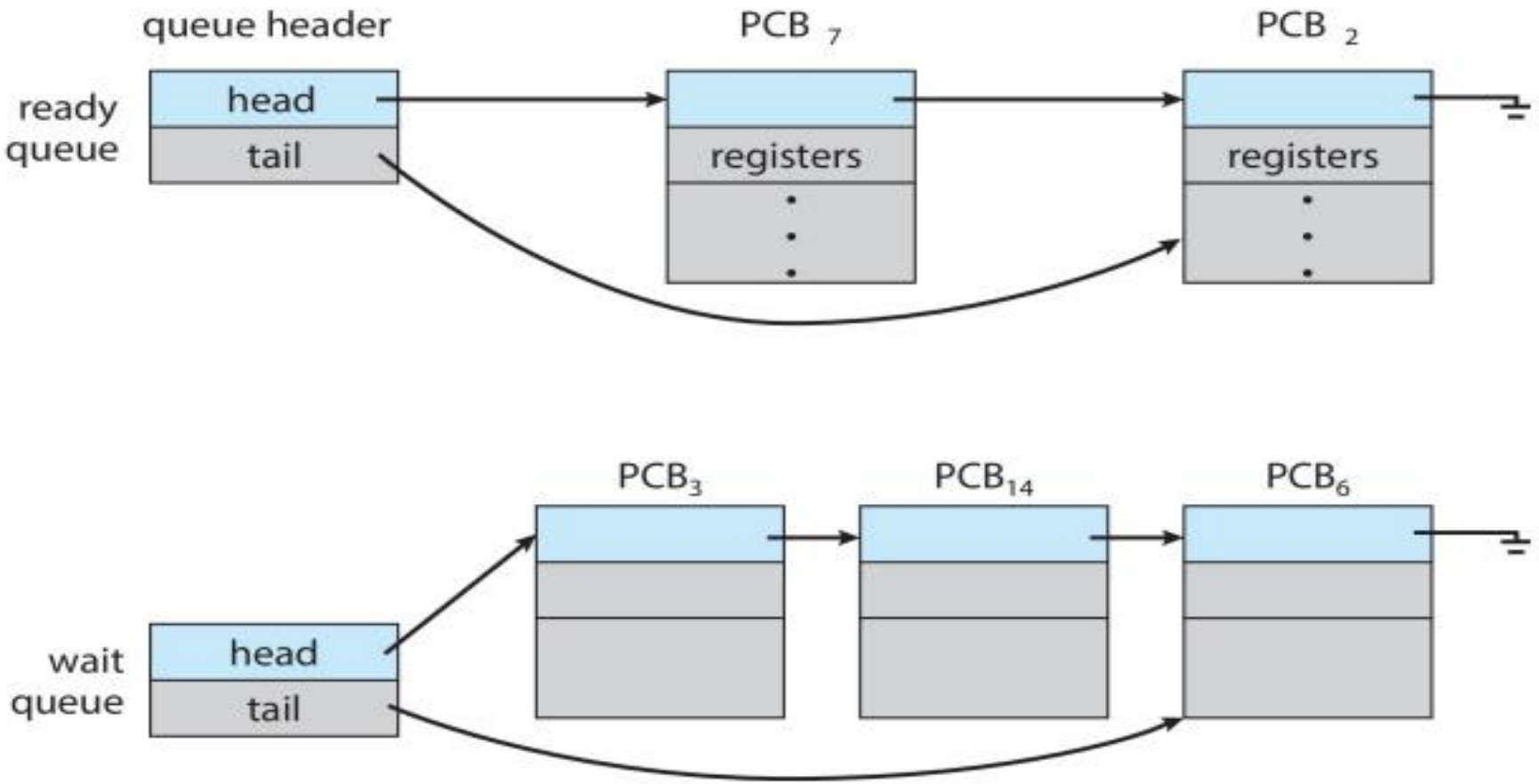
Process Representation in Linux

- 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 *

Process Representation in Linux



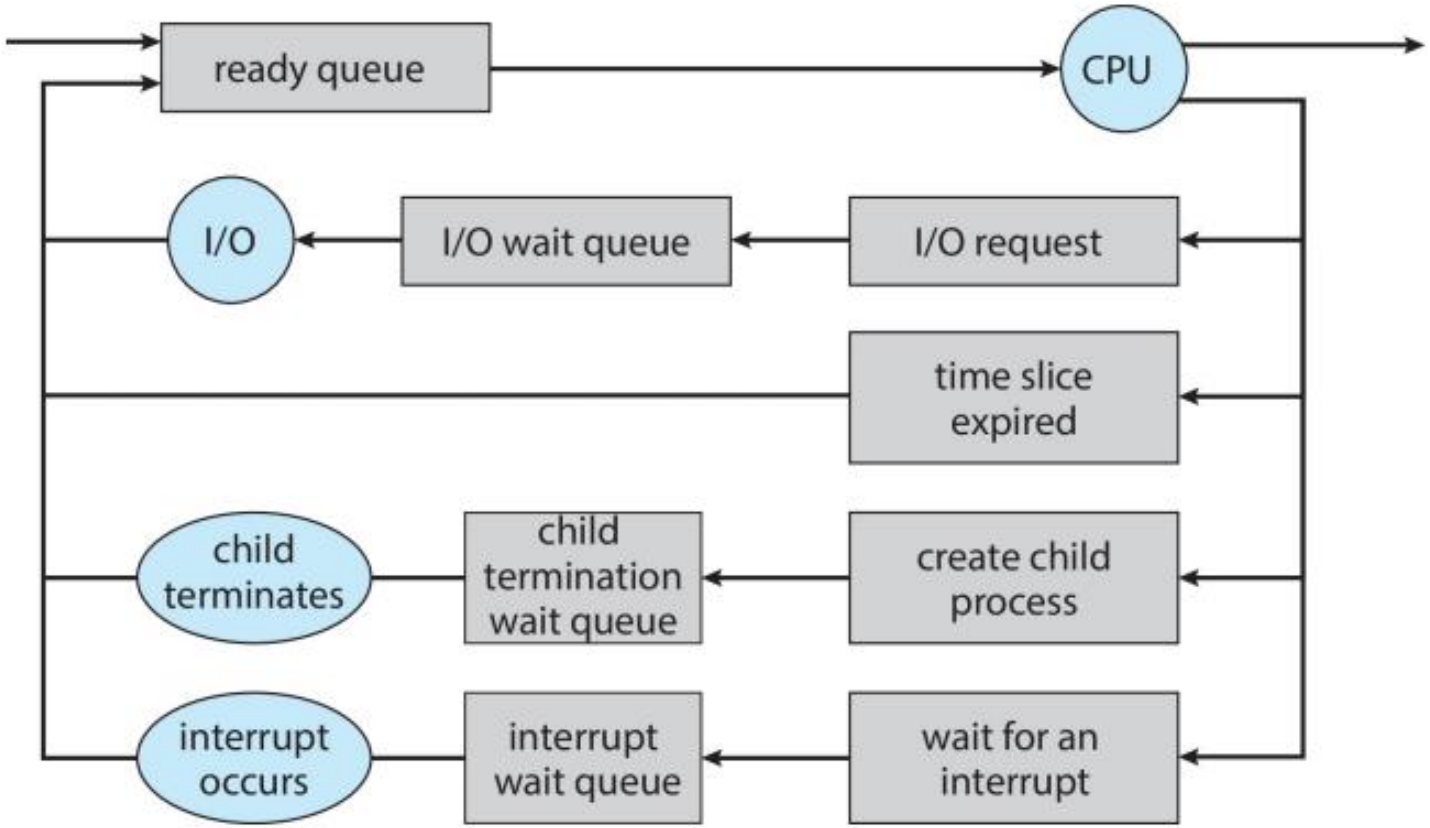
Ready and Wait Queue



Scheduling Queues

- A new process is initially put in the ready queue. It waits there until it is selected for execution, or dispatched. Once the process is allocated a CPU core and is executing, one of several events could occur:
  - The process could issue an I/O request and then be placed in an I/O wait queue.
  - The process could create a new child process and then be placed in a wait queue while it awaits the child's termination.
  - The process could be removed forcibly from the core, as a result of an interrupt or having its time slice expire, and be put back in the ready queue.

Representation of Process Scheduling



CPU Scheduling

- The role of the CPU scheduler is to select from among the processes that are in the ready queue and allocate a CPU core to one of them.
- CPU scheduler must select a new process for the CPU frequently.

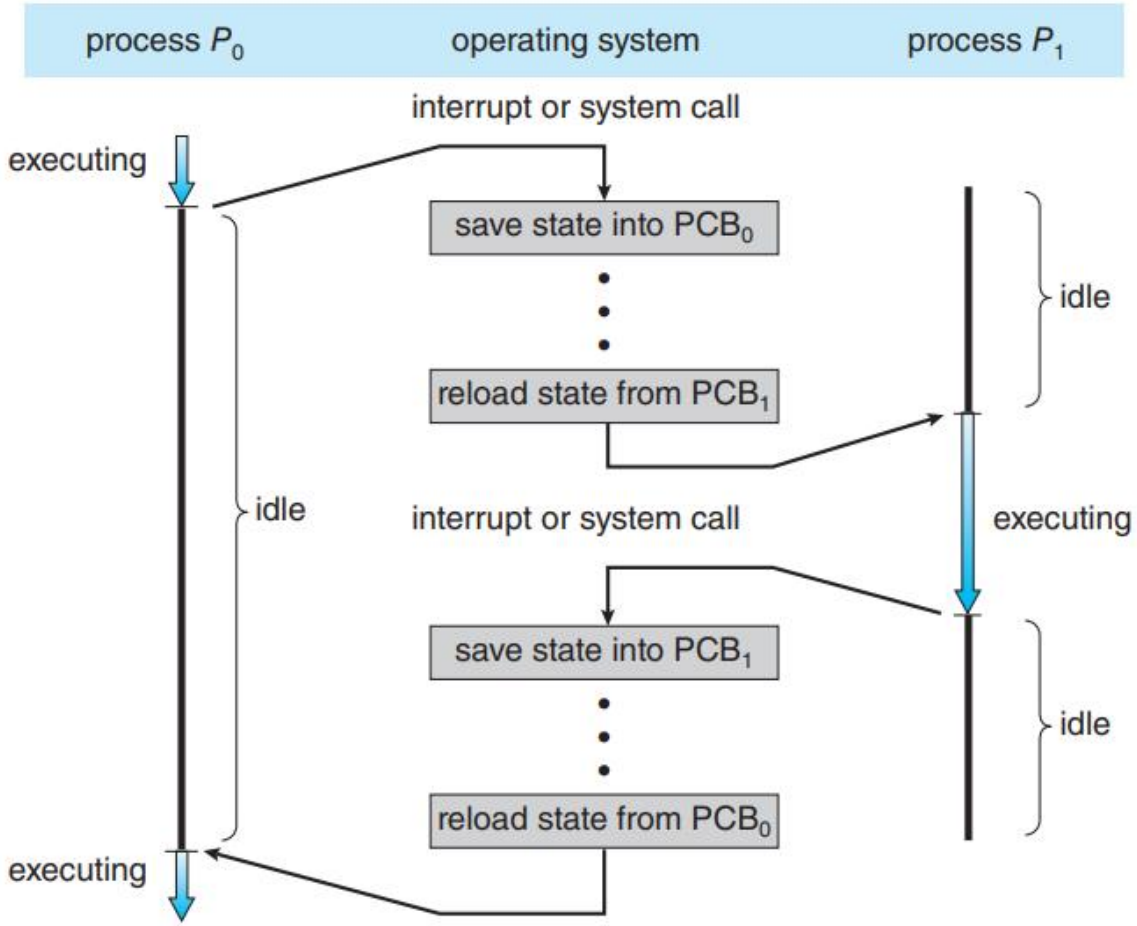
- Swapping:
  - operating systems have an intermediate form of scheduling, known as swapping, whose key idea is that sometimes it can be advantageous to remove a process from memory and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is known as swapping because a process can be "swapped out" from memory to disk, where its current status is saved, and later "swapped in" from disk back to memory, where its status is restored.

Context Switching

Switching the CPU core to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch

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

Context Switching (Cont…)



Context Switching (Cont…)

- 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 pure 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

Multitasking in Mobile System

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - 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
- Android runs foreground and background, with fewer limits
  - Background process uses a service to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

Operations on Process

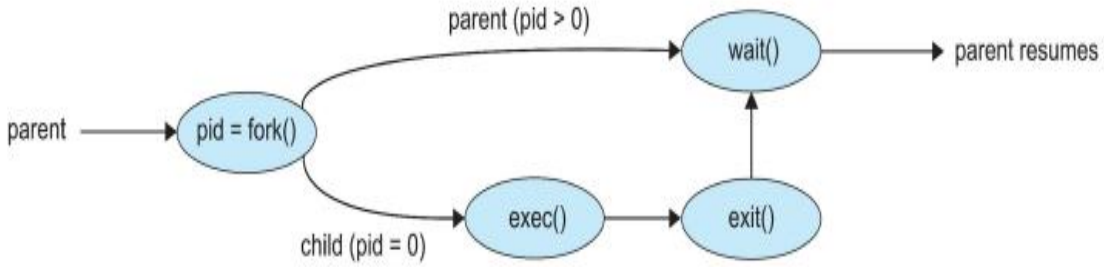
- System must provide mechanisms for:
  - Process creation
  - Process termination

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

Process Creation (Cont…)

- Address space
  - Child duplicate of parent
  - Child 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
  - Parent process calls wait()waiting for the child to terminate

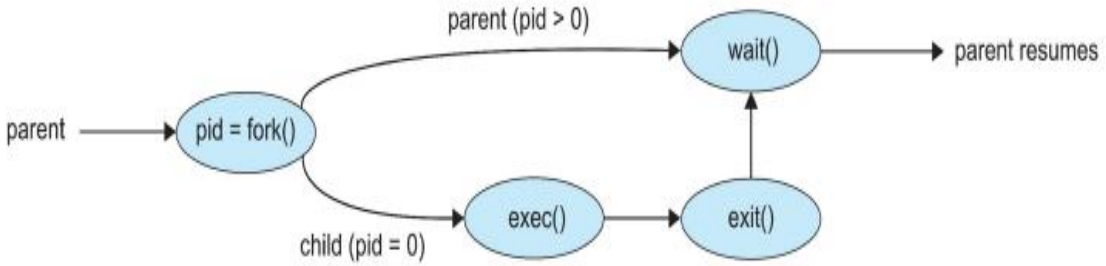


Process Creation (Cont…)

- When a process creates a new process, two possibilities for execution exist:
  - The parent continues to execute concurrently with its children.
  - The parent waits until some or all of its children have terminated.

- There are also two address-space possibilities for the new process
  - The child process is a duplicate of the parent process (it has the same program and data as the parent).
  - The child process has a new program loaded into it.

Process Creation (Cont…)




```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;
}
```

Practice Questions

- What will be the output of following code.

```
int main() {
    pid_t pid;
    pid = fork();
    printf("The process creates child process");
    if (pid = =  0) {
    printf("This is child process")
    pid_t p=fork();
    printf(" Child process has been created");
     if(p==0)
     {
    printf("Grand Child process");
      }
      else
         Printf("Child process);
    }
    Else
    if(pid > 0)
       printf("This is parent process");
    Return 0;
}
```

Practice Questions

- **What will be the output of following code.**

Answer:

The process creates child process.

This is parent process.

This is child process.

Child process has been created.

Grand child process.

```
int main() {
    pid_t pid;
    pid = fork();
    printf("The process creates child process");
    if (pid = =  0)
    {
    printf("This is child process")
    pid_t p=fork();
    printf(" Child process has been created");
     if(p==0)
     {
    printf("Grand Child process");
      }
      else
         Printf("Child process);
    }
    Else
    if(pid > 0)
       printf("This is parent process");
    Return 0;
}
```

Process Termination

- Process executes last statement and then asks the operating system to delete it using **the exit() system call**.
  - Returns status data 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 (Cont…)

- 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.
- 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 = wait(&status);
```

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

Zombie Process

- A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process.
- A child process always first becomes a zombie before being removed from the process table.

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // Fork returns process id  in parent process
    pid_t child_pid = fork();

    // Parent process
    if (child_pid > 0)
        sleep(50);

    // Child process
    else
        exit(0);

    return 0;
```

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.

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

int main()
{
   // Create a child process
   int pid = fork();

   if (pid > 0)
      printf("in parent process");

   // Note that pid is 0 in child process

else if (pid == 0)
   {
      sleep(30);
      printf("in child process");
   }

   return 0;
}
```

Android process Hierarchy

- Mobile operating systems often have to terminate processes to reclaim system resources such as memory. From most to least important:
  - Foreground process
  - Visible process
  - Service process
  - Background process
  - Empty process
- Android will begin terminating processes that are least important.

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
- Plug-in process for each type of plug-in

Inter Process Communication

- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
- Two types of processes
  1. Independent Processes
  2. Cooperating Processes
- Independent Processes
  - A process is independent if it does not share data with any other processes executing in the system.
- Dependent Processes
  - A process is cooperating if it can affect or be affected by the other processes executing in the system.

Inter Process Communication

- There are several reasons for providing an environment that allows process cooperation:
- **Information sharing.**
  - Since several applications may be interested in the same piece of information (for instance, copying and pasting), we must provide an environment to allow concurrent access to such information.
- **Computation speedup**
  - If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Speedup can be achieved only if the computer has multiple processing cores.
- **Modularity**
  - We may want to construct the system in a modular fashion, dividing the system functions into separate processes or thread

Inter Process Communication

Cooperating processes need interprocess communication (IPC)

Two models of IPC

- Shared memory
  - A region of memory that is shared by the cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
- Message passing
  - Communication takes place by means of messages exchanged between the cooperating processes

Inter Process Communication