OPERATING SYSTEMS

Process Management 2

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

OPERATING SYSTEMS

Course Syllabus - Unit 1

UNIT 1: Introduction and Process Management

Operating-System Structure & Operations, Kernel Data Structures,Computing Environments, Operating-System Services, OperatingSystem Design and Implementation. Process concept: Process in memory, Process State, Process Control Block, Process Creation and Termination, CPU Scheduling and Scheduling Algorithms, IPC - Shared Memory & Message Passing, Pipes - Named and Ordinary. Case Study: Linux/Windows Scheduling Policies.

OPERATING SYSTEMS

Course Outline

Class No.	Chapter Title / Reference Literature	Topics to be covered	% of Portions covered	
			Reference chapter	Cumulative
1	1.1-1.2	What Operating Systems Do, Computer-System Organization?	1	21.4
2	1.3,1.4,1.5	Computer-System Architecture, Operating-System Structure & Operations	1	
3	1.10,1.11	Kernel Data Structures, Computing Environments	1	
4	2.1,2.6	Operating-System Services, Operating System Design and Implementation	2	
5	3.1-3.3	Process concept: Process in memory, Process State, Process Control Block, Process Creation and Termination	3	
6	5.1-5.2	CPU Scheduling: Basic Concepts, Scheduling Criteria	5	
7	5.3	Scheduling Algorithms: First-Come, First-Served Scheduling, Shortest-Job-First Scheduling	5	
8	5.3	Scheduling Algorithms: Shortest-Job-First Scheduling (Pre-emptive), Priority Scheduling	5	
9	5.3	Round-Robin Scheduling, Multi-level Queue, Multi-Level Feedback Queue Scheduling	5	
10	5.5,5.6	Multiple-Processor Scheduling, Real-Time CPU Scheduling	5	
11	5.7	Case Study: Linux/Windows Scheduling Policies	5	
12	3.4,3.6.3	IPC – Shared Memory & Message Passing, Pipes – Named and Ordinary	3,6	

Topics Outline

- Process Concept
  - The Process
  - Process State
  - Process Control Block
- Process Scheduling
  - Scheduling Queues
  - Schedulers
  - Context Switch
- Operations on Processes
  - Process Creation
  - Process Termination
- Typical Q and As

Process Scheduling: Schedulers

- A process **migrates** among the various scheduling queues throughout its **lifetime**.
- The operating system must **select**, for scheduling purposes, processes from these queues in some fashion.
- The selection process is carried out by the appropriate **scheduler**
- In a **batch system**, more processes are submitted than those which can be executed immediately.
- **Batch processes** are spooled to a mass-storage device typically a disk, where they are **kept** for later execution.
- The **long-term scheduler, or job scheduler**, selects batch processes from this pool and loads them into memory for execution.

Process Scheduling: Schedulers

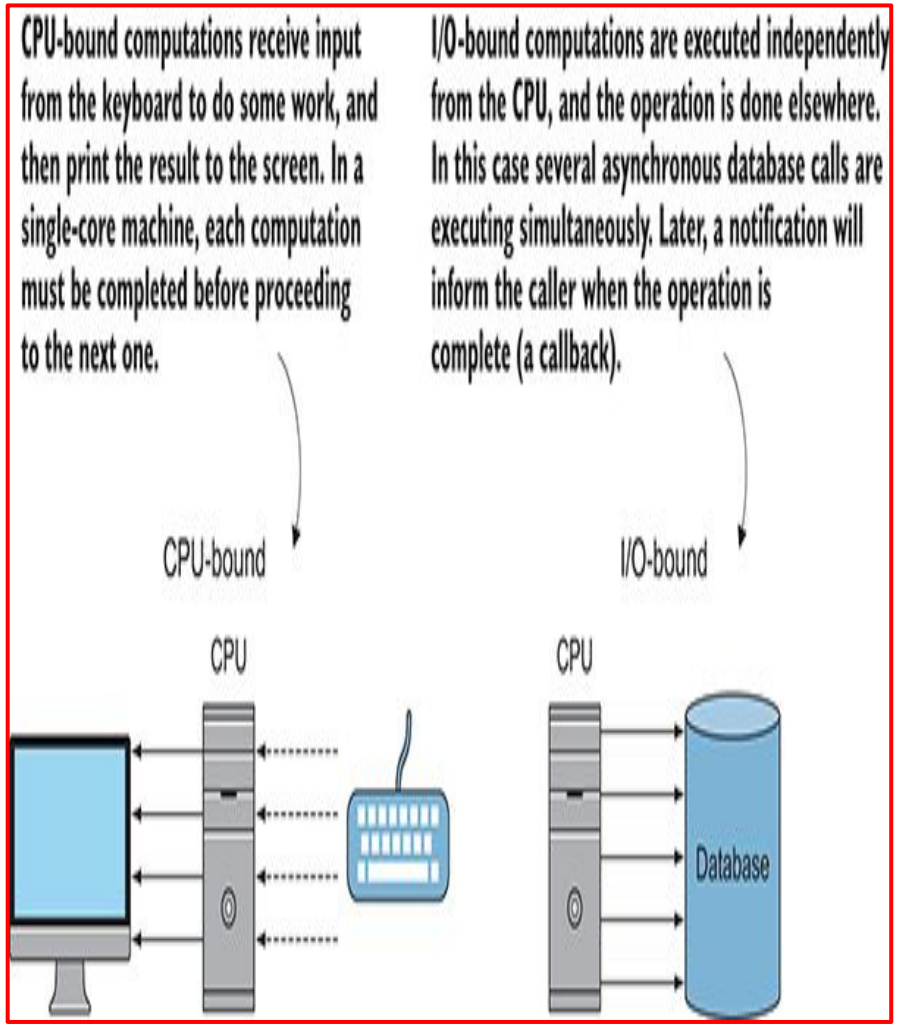
- The **short-term scheduler, or CPU scheduler**, selects from among the processes that are **ready** to **execute** and allocates the CPU to one of them
- The **primary distinction** between these two schedulers lies in **frequency** of execution.
- The **short-term scheduler** must select a new process for the CPU frequently.
- A process may execute for only a few milliseconds before waiting for an I/O request.
- The **short-term scheduler** executes at least once every **100 milliseconds**.
- Due to the short time between executions, the **short-term scheduler** must be **fast**
- If it takes 10 milliseconds to decide to execute a process for 100 milliseconds, then **10**/(100 + 10) = **9 percent** of the CPU is being **used** (wasted) simply for scheduling the work

Process Scheduling: Schedulers

- The **long-term scheduler** executes much less frequently; minutes may separate the creation of one new process and the next.
- The **long-term scheduler** controls the **degree of multiprogramming** which is the **number of processes** in memory.
- If the degree of multiprogramming is **stable**, then the average rate of **process creation** must be equal to the average **departure rate of processes** leaving the system.
- The **long-term scheduler** may need to be invoked only when a process leaves the system.
- Due to the longer interval between executions, **the long-term scheduler** can **afford** to take more time to decide which process should be selected for execution.
- It is important that the long-term scheduler make a careful selection.

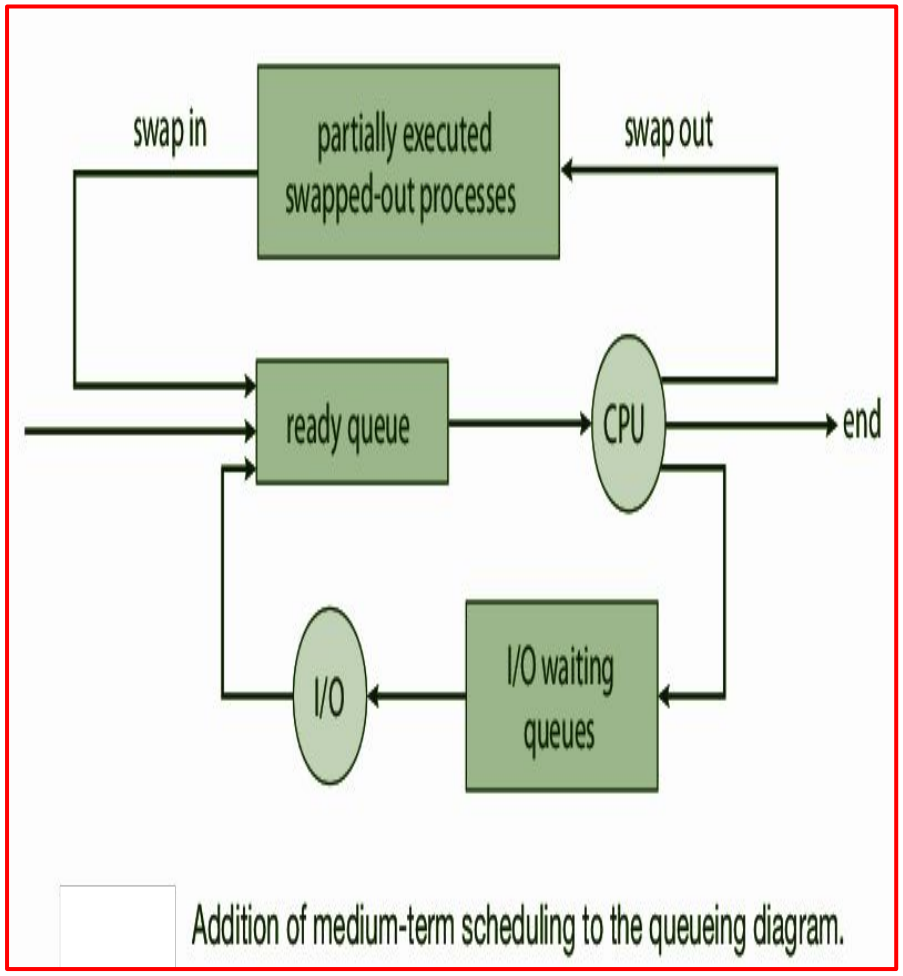
Process Scheduling: Schedulers

- Most processes can be described as either **I/O bound** or **CPU bound**
- An **I/O -bound process** is one that spends **more** of its **time doing I/O** than it spends doing computations.
- A **CPU -bound process**, in contrast, generates I/O requests infrequently, using **more** of its **time doing computations**.
- It is important that the long-term scheduler select a **good process mix** of I/O -bound and CPU bound processes.
- **Unbalanced** selection of Process may result in less work for the **short term scheduler**
- The system with the best performance will thus have a combination of CPU bound and I/O -bound processes.



Process Scheduling: Schedulers

- Some operating systems, such as time-sharing systems, may introduce an **additional**, intermediate level of scheduling.
- **Medium Term Scheduler** is that sometimes it can be advantageous to remove a process from memory and from active contention for the CPU and thus reduce the degree of multiprogramming.
- The process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**.
- The process is **swapped out**, and is later **swapped in**, by the medium-term scheduler.
- Swapping may be necessary to improve the process mix or because a change in memory requirements has **overcommitted** available memory, requiring memory to be freed up.



Addition of medium-term scheduling to the queueing diagram.

Process Scheduling: Context Switch

- Interrupts cause the operating system to **change** a **CPU** from its current task and to run a **kernel routine**.
- Such **operations** happen **frequently** on general-purpose systems.
- When an interrupt occurs, the system needs to **save** the current context of the process running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.
- Switching the CPU 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**.

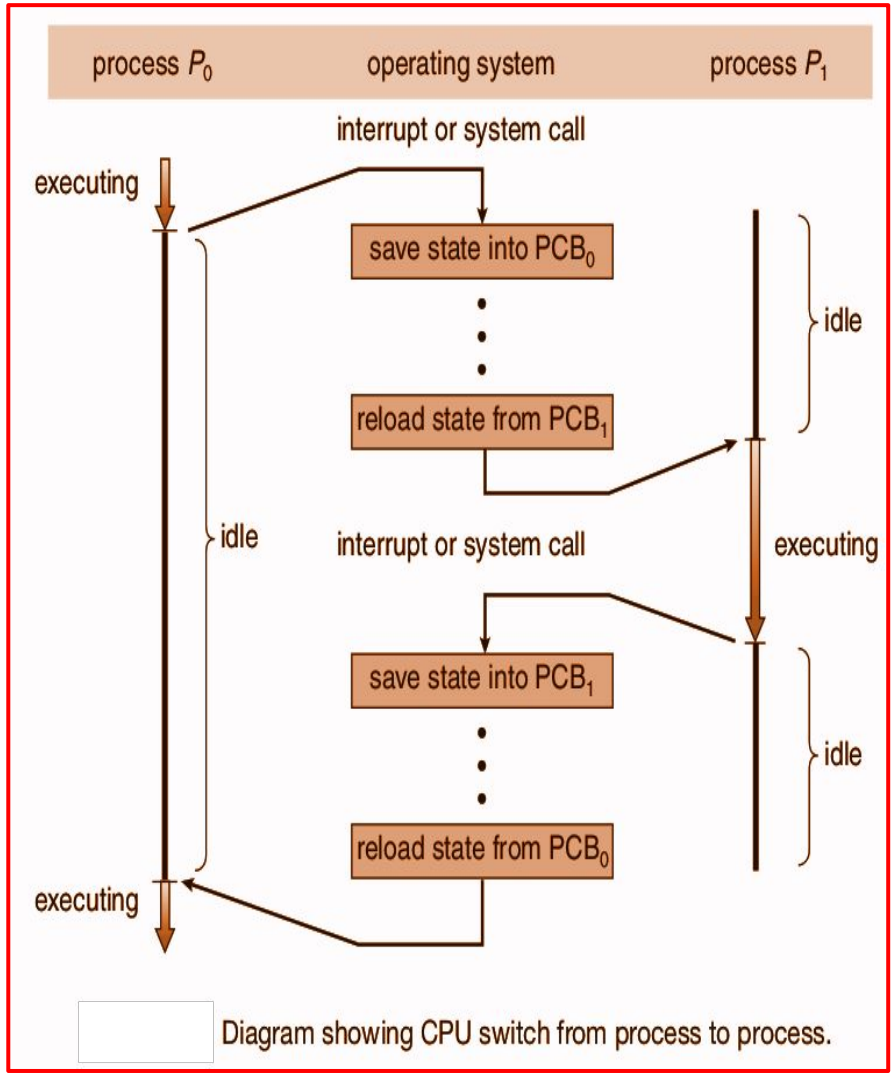


Diagram showing CPU switch from process to process.

Process Scheduling: Context Switch

- Context-switch time is pure **overhead**, because the system does no useful work while switching.
- **Switching speed** which is usually few milliseconds **varies** from **machine to machine**, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions
- Context-switch times are highly dependent on hardware support.

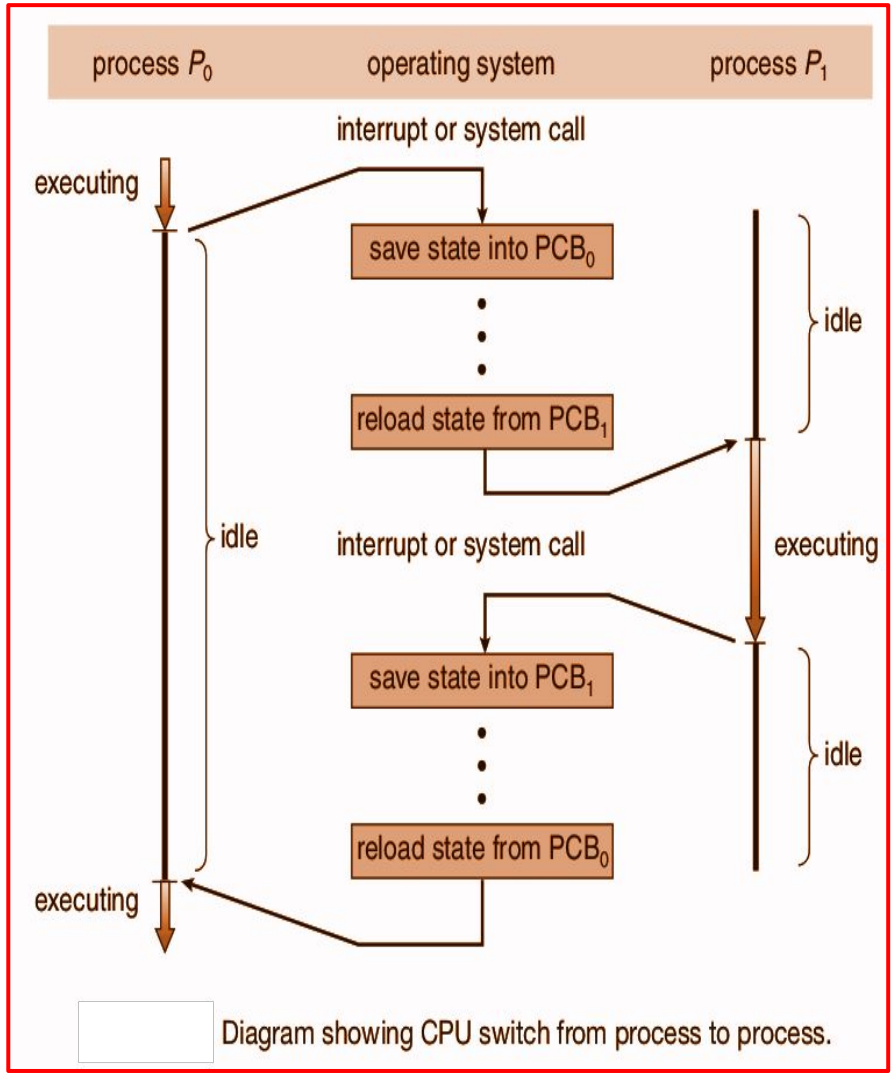


Diagram showing CPU switch from process to process.

Operations on Processes

- The processes in most systems can execute concurrently, and they may be created and deleted dynamically.

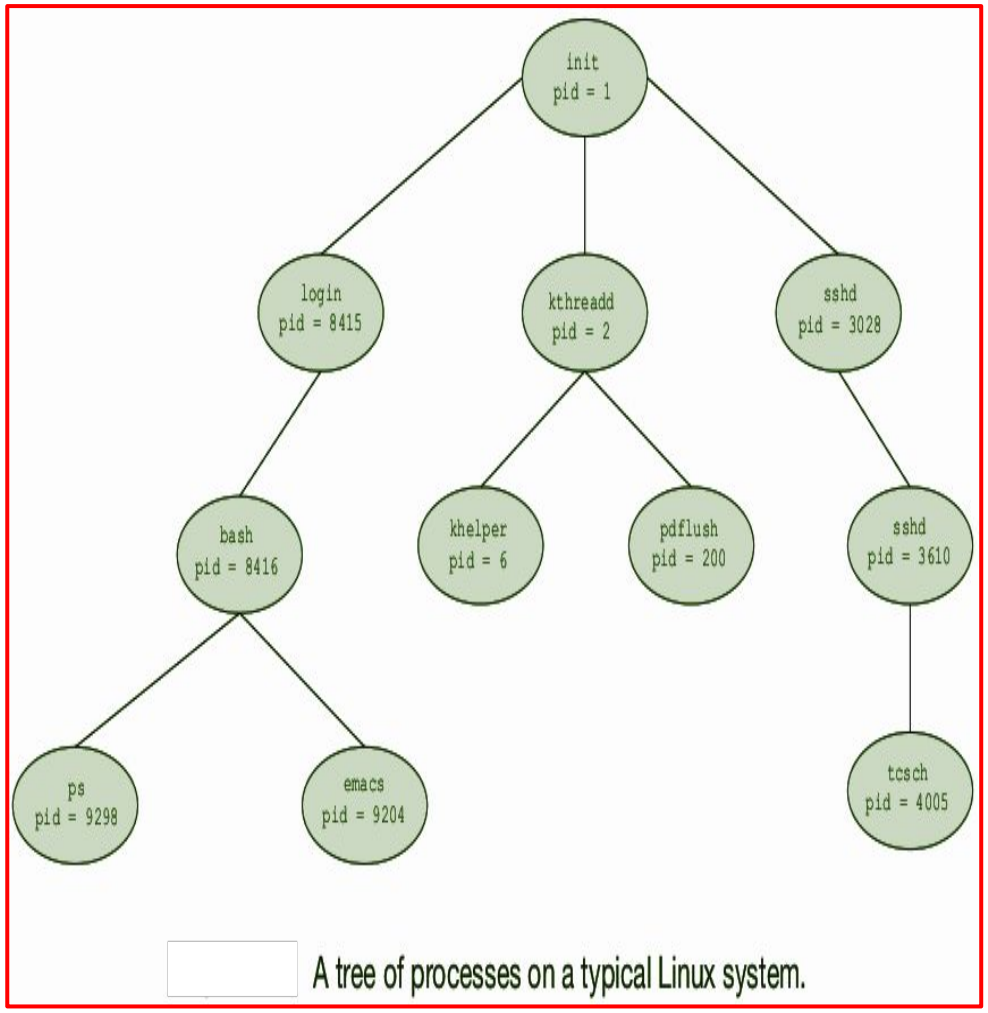
- Such systems must provide a mechanism for **process creation** and **termination**.

Operations on Processes: Process Creation

- During the course of execution, a **process** may **create** several **new processes**.
- The creating process is called a **parent process**, and the new processes are called the **children of that process**.
- Each of these new processes may in turn create other processes, forming a **tree** of processes.
- Most operating systems (including UNIX , Linux, and Windows) identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number.
- The **pid** provides a unique value for each process in the system, and it can used as an index to access various attributes of a process within the kernel

Operations on Processes: Process Creation

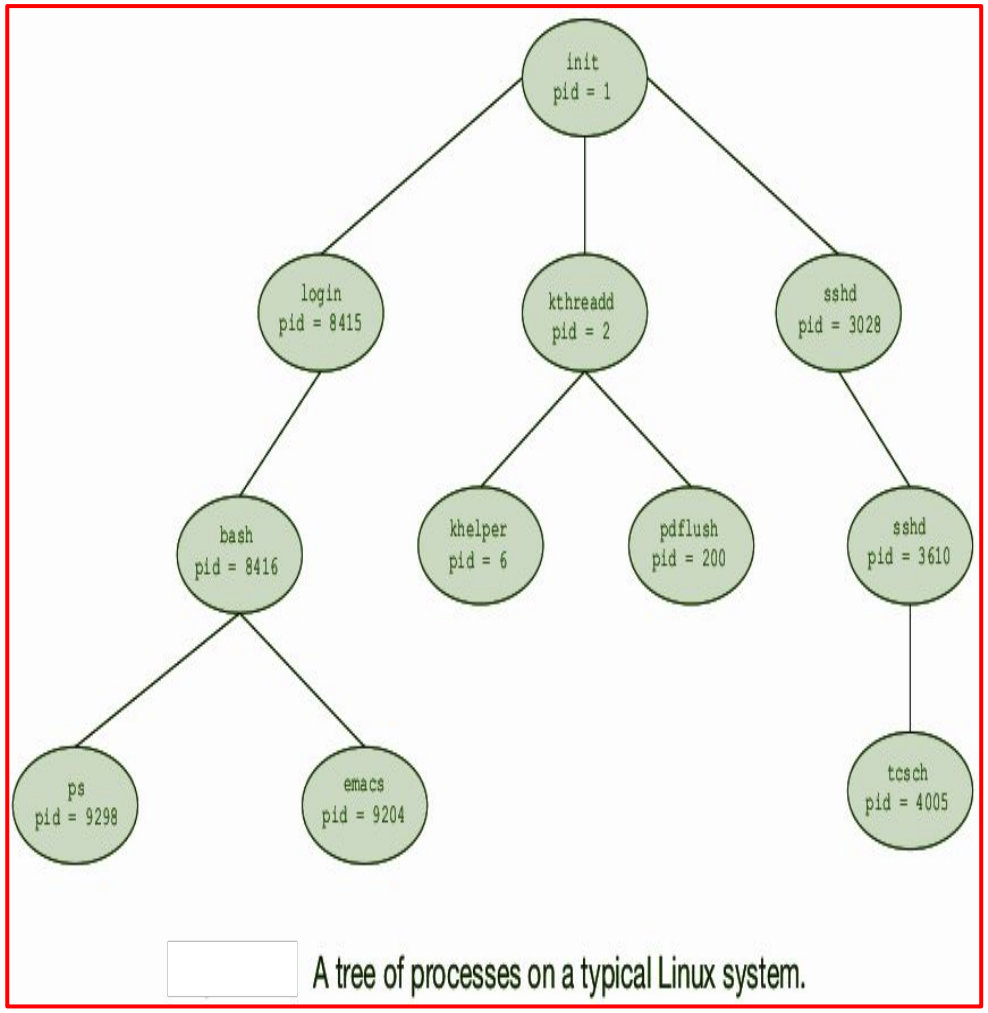
- The **init** process which always has a pid of 1 serves as the **root parent process** for all user processes.
- Once the system has booted, the **init process** can also **create** various user processes
- In Figure on the side, we see two children of init — kthreadd and sshd.
- The **kthreadd** process is responsible for creating additional processes that perform tasks on behalf of the kernel
- The **sshd** process is responsible for managing clients that connect to the system by using ssh which is short for secure shell.



A tree of processes on a typical Linux system.

Operations on Processes: Process Creation

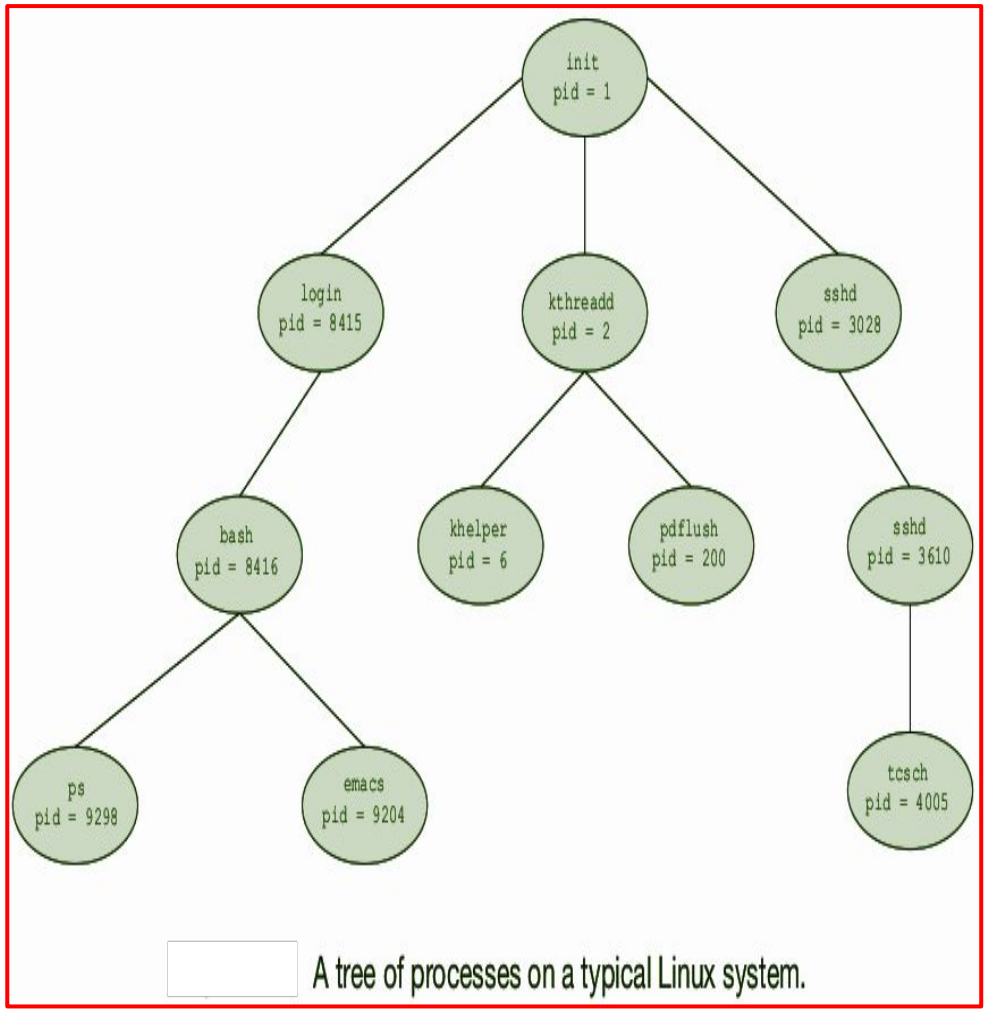
- The **login** process is responsible for managing clients that directly log onto the system.
- On UNIX and Linux systems, we can obtain a listing of processes by using the ps command.
- For example, the command ps -el will list complete information for all processes currently active in the system
- It is easy to **construct** a process tree similar to the one shown in Figure on the side by recursively tracing parent processes all the way to the init process.



A tree of processes on a typical Linux system.

Operations on Processes: Process Creation

- In general, when a process creates a child process, that child process will need certain resources such as CPU time, memory, files, I/O devices to accomplish its task.
- A **child process** may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.
- The parent may have to partition its resources among its children, or it may be able to share some resources such as memory or files among several of its children.
- Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.



A tree of processes on a typical Linux system.

Operations on Processes: Process Creation

- When a process creates a new process, two **possibilities for execution** exist:

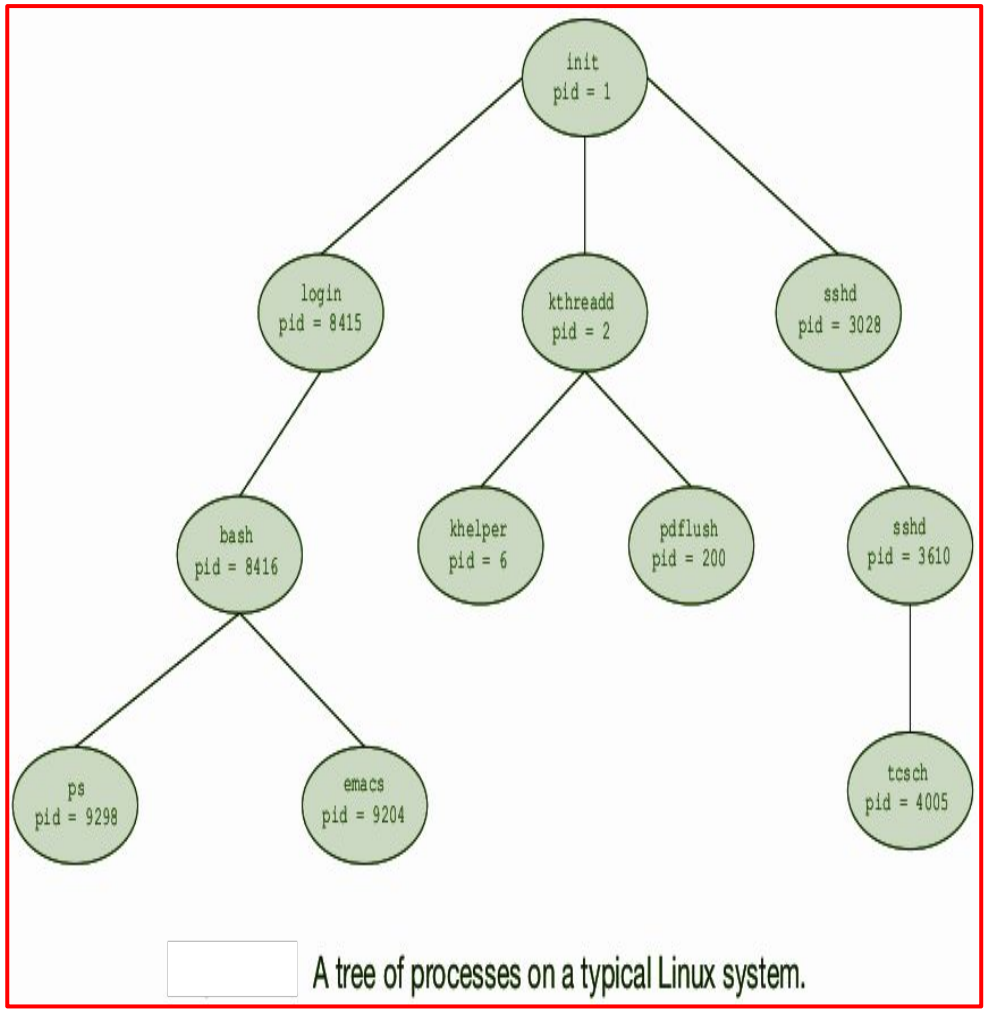
  1. The parent **continues** to execute concurrently with its children.

  2. The parent **waits** until some or all of its children have terminated.

- There are also **two address-space possibilities** for the new process:

  1. The child process is a **duplicate** of the parent process it has the same program and data as the parent.

  2. The child process has a **new** program loaded into it.



A tree of processes on a typical Linux system.

Operations on Processes: Process Creation

- In UNIX ,
- A new process is created by the **fork()** system call.
- The new process consists of a copy of the address space of the original process.
- This mechanism allows the parent process to communicate easily with its child process.
- Both processes the parent and the child continue execution at the instruction after the fork() , with one difference: 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.

```
#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;
}
```

Creating a separate process using the UNIX `fork()` system call.

Operations on Processes: Process Creation

- After a **fork()** system call, one of the two processes typically uses the **exec()** system call to replace the process's memory space with a new program.

- The **exec()** system call loads a binary file into memory destroying the memory image of the program containing the exec() system call and starts its execution.

- **I will create a detailed video offline as learning material to practically demonstrate the working of fork() , exec(), exit()**

```
#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;
}
```

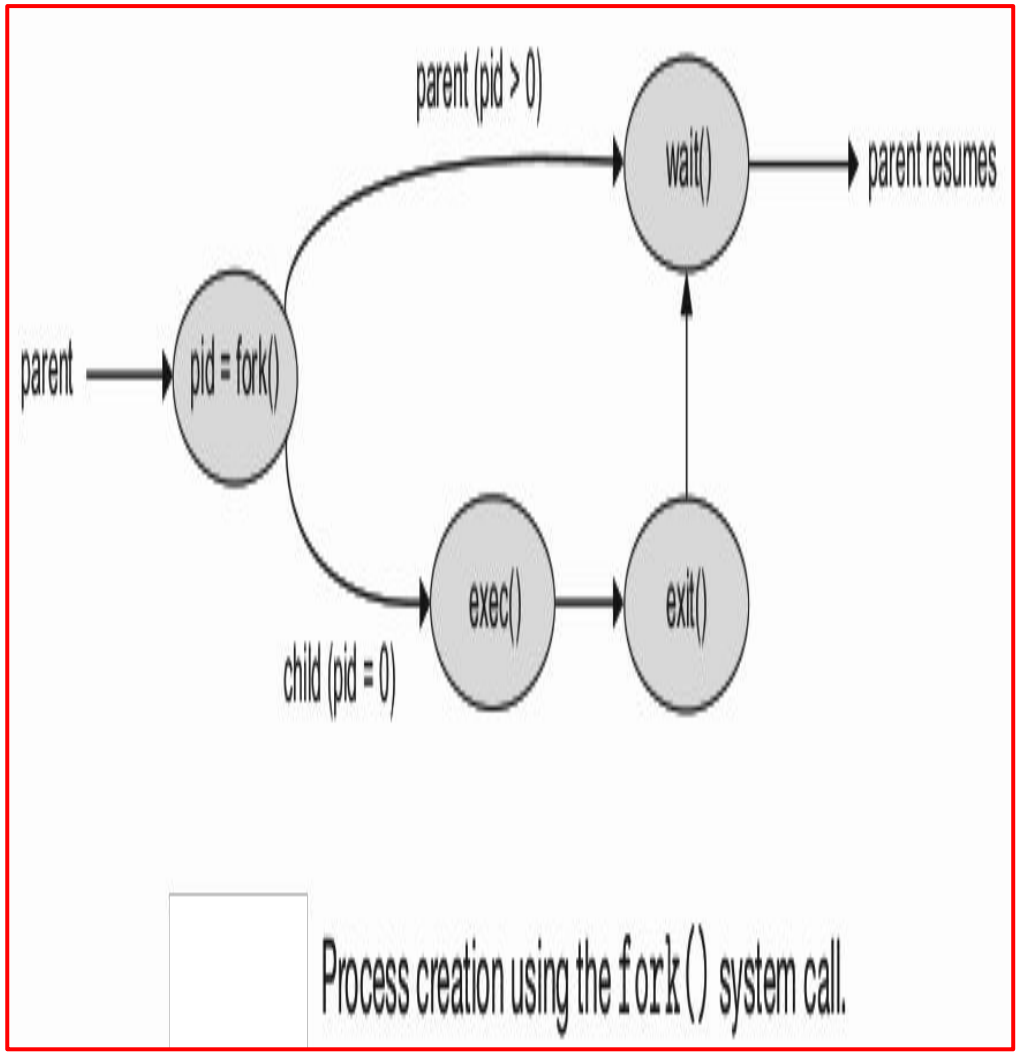
Creating a separate process using the UNIX `fork()` system call.

Operations on Processes: Process Creation

- After a **fork()** system call, one of the two processes typically uses the **exec()** system call to replace the process's memory space with a new program.

- The **exec()** system call loads a binary file into memory destroying the memory image of the program containing the exec() system call and starts its execution.

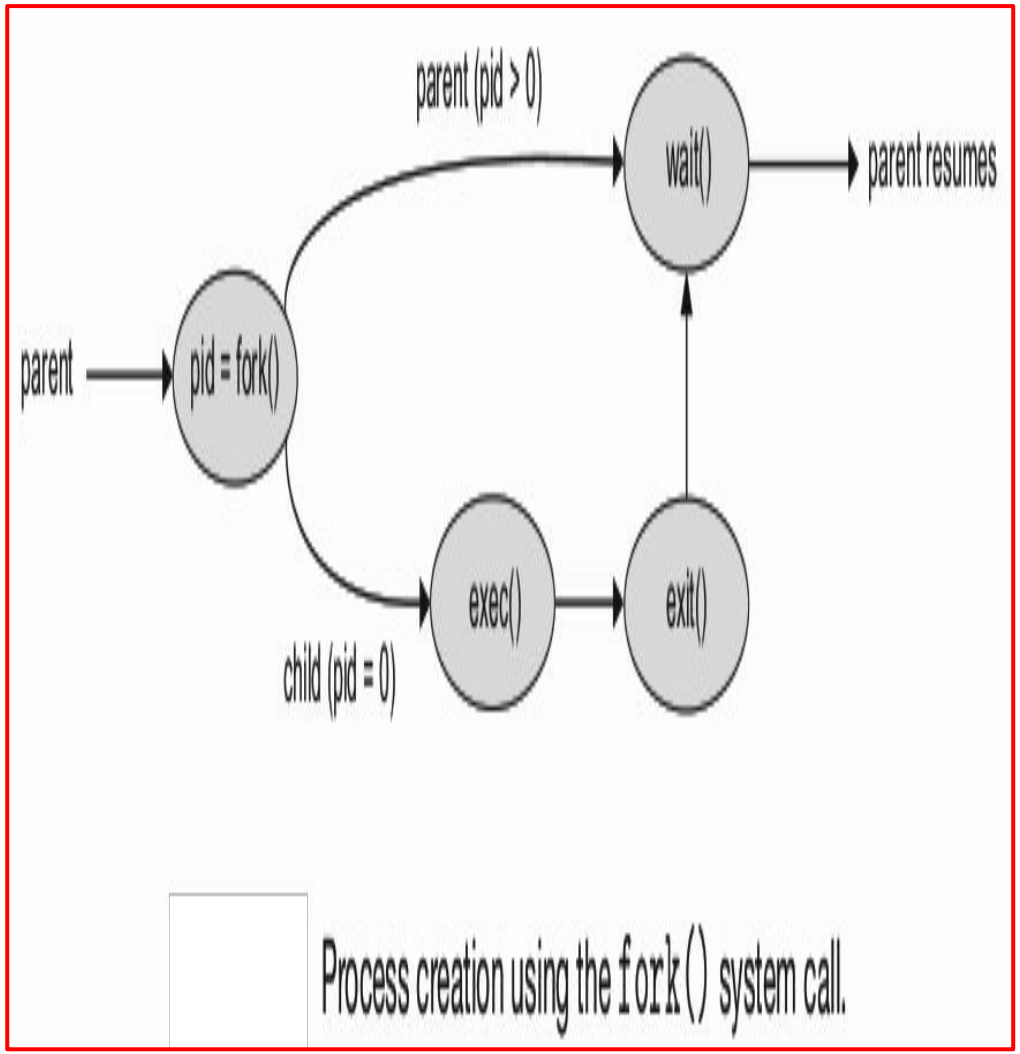
- **I will create a detailed video offline as learning material to practically demonstrate the working of fork() , exec(), exit()**



Process creation using the fork() system call.

Operations on Processes: Process Termination

- A **process terminates** when it finishes executing its final statement and asks the operating system to delete it by using the **exit() system call**.
- At that point, the process may return a status value typically an integer to its parent process via the wait() system call.
- All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.
- **I will create a detailed video offline as learning material to practically demonstrate the working of fork() , exec(), exit()**



Process creation using the fork() system call.

Operations on Processes: Process Termination

- Termination can occur in other circumstances as well.
- A **process** can cause the termination of **another process** via an appropriate system call
- Note that a parent needs to know the identities of its children if it is to terminate them
- A parent may terminate the execution of one of its children for a variety of reasons,
    - The **child has exceeded** its usage of some of the resources that it has been allocated. To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.
    - The **task** assigned to the child is **no longer required**.
    - The parent is **exiting**, and the operating system does not allow a child to continue if its parent **terminates**.
- Some systems do not allow a child to exist if its parent has terminated.
- In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated.
- This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.

Operations on Processes: Process Termination

- A process that has terminated, but whose parent has not yet called wait() , is known as a **zombie** process.
- All processes transition to this state when they terminate, but generally they exist as **zombies** only briefly.
- Once the parent calls wait() , the process identifier of the **zombie** process and its entry in the process table are released.
- Now consider what would happen if a parent did not invoke wait() and instead terminated, thereby leaving its child processes as **orphans**.
- Linux and UNIX address this scenario by assigning the **init process** as the **new parent** to **orphan processes**.
- The init process periodically invokes **wait()** , thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

Typical Q and As

Process	A program in execution
Resources	A process needs several _____ to accomplish its task
Thread	Traditionally, processes contained only a single_____.
Concurrently	Modern computer systems allow multiple programs to be executed _____.
Jobs	A batch system executes _____.
tasks	A time-shared system executes _____.
Text section	The actual program code of a process
Program counter	Represents the current activity of a process
Process stack	Contains temporary data about a process, such as function parameters, return addresses, and local variables.

Typical Q and As

Data section (of a process)	Contains the global variables of a process.
Heap	Memory that is dynamically allocated during process run time.
Executable file	A program stored within a file, waiting to be run.
Double-clicking the program icon Entering program name into command line	2 methods of loading an executable file
State	As a program executes, its _____ changes.

Typical Q and As

Process state	Defined in part by the current activity of that process
New (process state)	The process is being created
Running(process state)	Instructions are being executed
Waiting(process state)	Process is waiting for some event to occur
Ready(process state)	Process is waiting to be assigned to a processor
Terminated(process state)	Program has finished execution
Running	While many processes can be ready and waiting, only one process can be _____ on any processor at any instant.

Typical Q and As

Process control block	The way in which each process is represented in the OS
What does the PCB contain?	Information associated with specific process.
Information in the PCB (7)	Process state, program counter, CPU registers, CPU-scheduling information, Memory information, accounting information, I/O status information
Information contained within the program counter aspect of the PCB	the address of the next instruction to be executed for this process
Information contained within the CPU registers aspect of the PCB	Keeping track of available registers for the process
CPU-scheduling information(PCB)	process priorities, pointers to scheduling queues, any other scheduling parameters
Memory management (PCB)	Controls memory allocated to the process
Accounting information (PCB)	Controls CPU and real time used, time limits, job numbers, etc.
I/O information (PCB)	Controls I/O devices allocated to process

Typical Q and As

Multicore	On _____ systems, multiple threads can run in parallel.
PCB	On a system that supports threads, the _____ is extended to include information for each thread
task_struct	The process control block in Linux
Parent process	A process that creates other processes
Child process	A process created by another process
Sibling processes	Processes with the same parent
Doubly linked list	In Linux, all active processes are represented using a doubly linked list
Objective of time sharing	To switch the CPU among processes so frequently that users can interact with each program while it is running.
Process scheduler	Selects an available process for program execution on the CPU

Typical Q and As

Job queue	As processes enter the system, they are put into a ___ _____.
Ready queue	A list of processes in main memory that are ready and waiting to execute
Ready-queue header	Contains pointers to the first and final PCBs in the list of ready processes
Device queue	List of processes waiting for a particular I/O device
Dispatch	When a process is selected for execution
Deallocated	A cycle of queueing occurs until a process is terminated and its resources get _____.
Scheduler	Selects processes from queues
Long-term scheduler	Selects processes from queues and loads them into memory for execution
Short-term scheduler	Selects from processes that are ready to execute and allocates the CPU to one of them.
Faster	The short-term scheduler is much _____ than the long-term scheduler

Typical Q and As

Degree of multiprogramming	The long term scheduler controls the _____ ___ _____
I/O ; CPU	most processes are either _____ bound or _____ bound
I/O bound process	Spends more of its time doing I/O than doing computations
CPU-bound process	Uses more of its time doing computations
Mix	It is important for the long-term scheduler to select a good _____ of I/O bound and CPU bound processes
Medium-term scheduler	Removing a process from memory and reintroduce later via swapping.
Context switch	Saving the state of the current process(state save) and re-loading the state of a previously interrupted process(state restore).
Context(of a process)	Includes value of registers, process state, and memory management information
Overhead	Context-switch time is _____ because the system does no useful work while switching

Typical Q and As

Foreground application	The application currently on the mobile display
Background applications	The mobile applications that remain in memory but do not currently appear on the display
Service (in Android)	A separate application component that runs on behalf of the background process.
Process identifier (pid)	Provides a unique value for each process that the OS uses to identify the process
ps	The command for displaying a list of processes on UNIX and LINUX systems
init	The process that serves as the root parent process for all user processes in UNIX and LINUX
Registers	If hardware provides multiple sets of _____ per CPU, multiple contexts can be loaded at once
Resource sharing options for processes (3)	-Parent and child share ALL resources -Children share subset of parent resources -Parent and child share NO resources
Execution options for parent-child processes	-Execute concurrently -Parent waits until children terminate

Typical Q and As

Options for address-space of new child processes	-Child is duplicate of parent (same program and data) -Child process has new program loaded into it
fork()	System call for creating a new process in UNIX
Exec()	System call used after fork() to replace the process' memory space with a new program in UNIX
CreateProcess()	The system call used to create a new process in Windows
specified program	Whereas fork() has child inheriting address space of parent, CreateProcess() requires loading a _____ _____ into the address space of the child process.
WaitForSingleObject()	Sysem call that signals for parent process to wait until child terminates in Windows
exit()	Process executes last statement and then asks the OS to delete it.
wait()	Returns status data from child to parent
abort()	Parent terminates execution of children processes

Typical Q and As

Scenarios where parent process would use abort()	-child exceeded allocated resources -Task assigned to child is no longer needed -Parent is exiting and OS does not allow child to terminate if parent terminates
Cascading termination	OS terminates all children, grandchildren, etc. once a parent is terminated.
wait() system call	The parent process may wait for termination of child process by using _____ _____ _____.
pid = wait(&status)	Returns status information and the pid of the terminated process
Orphan process	If parent terminates without invoking wait(), children become _____ _____.
Zombie process	A process that has terminated but whose parent has not yet called wait()
LINUX and UNIX	In these OSes, the init process is assigned as new parent to orphan processes

Topics Outline

- Process Concept
  - The Process
  - Process State
  - Process Control Block
- Process Scheduling
  - Scheduling Queues
  - Schedulers
  - Context Switch
- Operations on Processes
  - Process Creation
  - Process Termination
- Typical Q and As

THANK YOU

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

nitin.pujari@pes.edu

For Course Deliverables by the Anchor Faculty click on www.pesuacademy.com