

Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

Chapter 3: Processes

2	Process Concept and scheduling		8	
	2.1	Process: Concept of a Process, Process States, Process Description, Process Control Block, Operations on Processes. Threads: Definition and Types, Concept of Multithreading,		CO2
	2.2	Multicore processors and threads.		
	2.3	Scheduling: Uniprocessor Scheduling - Types of Scheduling: Preemptive and, Non-preemptive, Scheduling Algorithms: FCFS, SJF, SRTN, Priority based, Round Robin, Multilevel Queue scheduling. Introduction to Thread Scheduling, Linux Scheduling.		

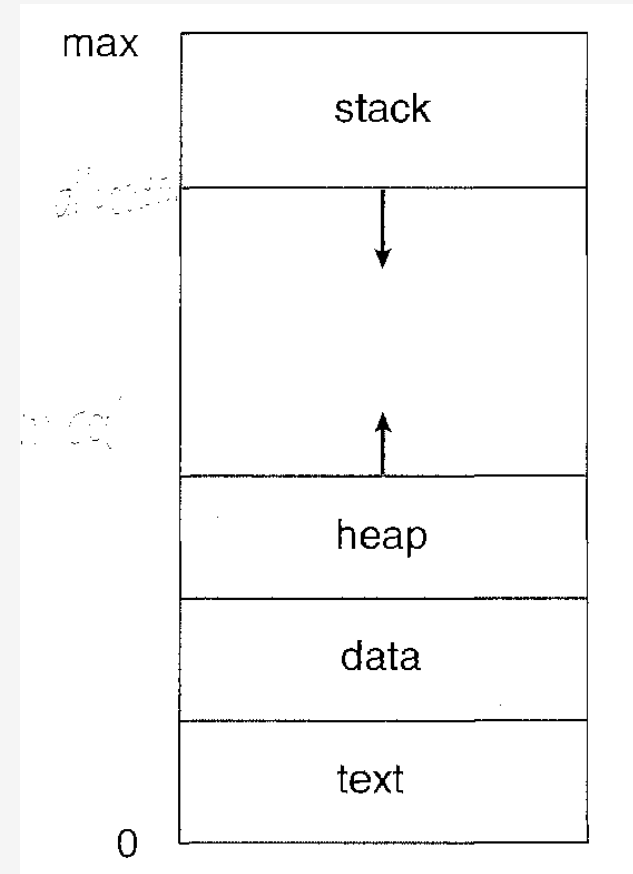
Process

- What ?
 - **A Program in execution**
 - Unit of work in modern time sharing systems
- How?
 - System consists of a collection of processes:
 - **Operating system processes execute system code**
 - **User processes executing user code.**
- Why?
 - All these processes can execute concurrently with the CPU(s) multiplexed among them.
 - **By switching between processes, the OS can make computer more productive.**

Process

- A program in execution
- **A process is more than the program code.**

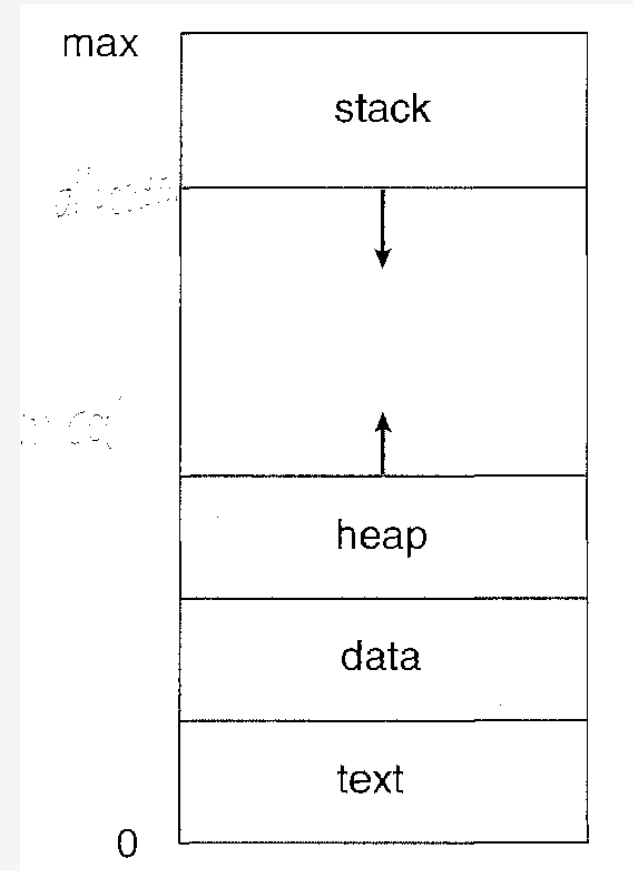
Process Concept



Multiple parts

- **The program code, also called **text section****
- Current activity represented by **value of program counter,**
- Content of processor registers
- **Process Stack** containing temporary data
 - **Function parameters, return addresses, local variables**

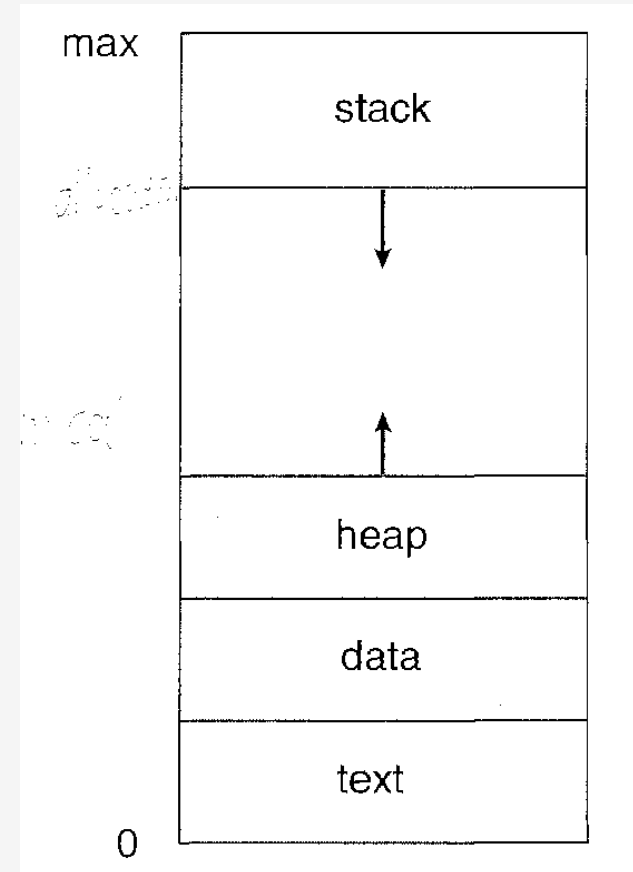
Process Concept



Multiple parts

- **Data section** containing **global variables**
- **Heap** containing **memory dynamically allocated during run time**

Process Concept



Process Concept

- An operating system executes a variety of programs:
 - Batch system – **jobs**
 - Time-shared systems – **user programs** or **tasks**
- The terms **job** and **process** are used almost interchangeably

Process Concept (Cont.)

- A program by itself is not a process.
- Program is **passive entity** stored on disk (**executable file**),
- Eg- A file containing a list of instructions stored on a disk
- Process is **active entity**
- **An Active entity** with a program counter specifying the next instruction to execute and a set of associated resources

When does a Program become a process?

Process Concept (Cont.)

- Program becomes process when executable file is loaded from disk into memory

When is an executable file loaded into memory?

Process Concept (Cont.)

- Two common techniques for loading executable files are :
 - Double-clicking an icon representing the executable file
 - Entering the name of the executable file on the command line

Process Concept (Cont.)

- Two processes may be associated with the same program but are separate execution sequences.
- One program can have several processes

- Consider multiple users executing the same program.
- Eg-several users may be running different copies of the mail program.

- Same user may invoke many copies of the editor program, Each one of them is a separate process.
- The text sections are equivalent, data sections may vary.

- As a process executes, it **changes state**.
- Each process may be in one of the following states:
 - **new**: The process is being created
 - **running**: Instructions are being executed
 - **Blocked/Waiting**: The process is waiting for some event to occur (I/O completion or reception of signal)
 - **ready**: The process is waiting to be assigned to a processor
 - **terminated**: The process has finished execution

New:

- A process that has just been created
- Not yet been admitted to the pool of executable processes by the OS.
- The new process has not yet been loaded into main memory,
- Although its process control block has been created.

For example,

- User attempts to log on to a time-sharing system or a new batch job is submitted for execution,
- What will Happen next?

- The OS can define a new process.
 - 1) An identifier is associated with the process.
 - 2) Any tables that will be needed to manage the process are allocated and built.

- This means that the OS has performed the necessary actions to create the process
- But has not committed itself to the execution of the process.

- While a process is in the new state, information concerning the process that is needed by the OS is maintained in control tables in main memory.
- However, the process itself is not in main memory.**
- Where then?**

- 1) The code of the program to be executed is not in main memory, and
- 2) No space has been allocated for the data associated with that program.
- 3) While the process is in the New state, the program remains in secondary storage, typically disk storage.

Ready:

- A process that is prepared to execute when given the opportunity.

Running:

- The process that is currently being executed

Blocked/Waiting:

- A process that cannot execute until some event occurs,
- Such as the completion of an I/O operation.
- I/O completion or reception of signal

Terminated/Exit:

- A process that has been released from the pool of executable processes by the OS,
- Either because it halted or because it aborted for some reason.

Terminated/Exit:

- A process is terminated –
 - 1) When it reaches a natural completion point,
 - 2) When it aborts due to an unrecoverable error, or
 - 3) When another process with the appropriate authority causes the process to abort

Terminated/Exit:

- Termination moves the process to the exit state.
- At this point, the process is no longer eligible for execution.
- The tables and other information associated with the job are temporarily preserved by the OS, which provides time for auxiliary or support programs to extract any needed information.

Terminated/Exit:

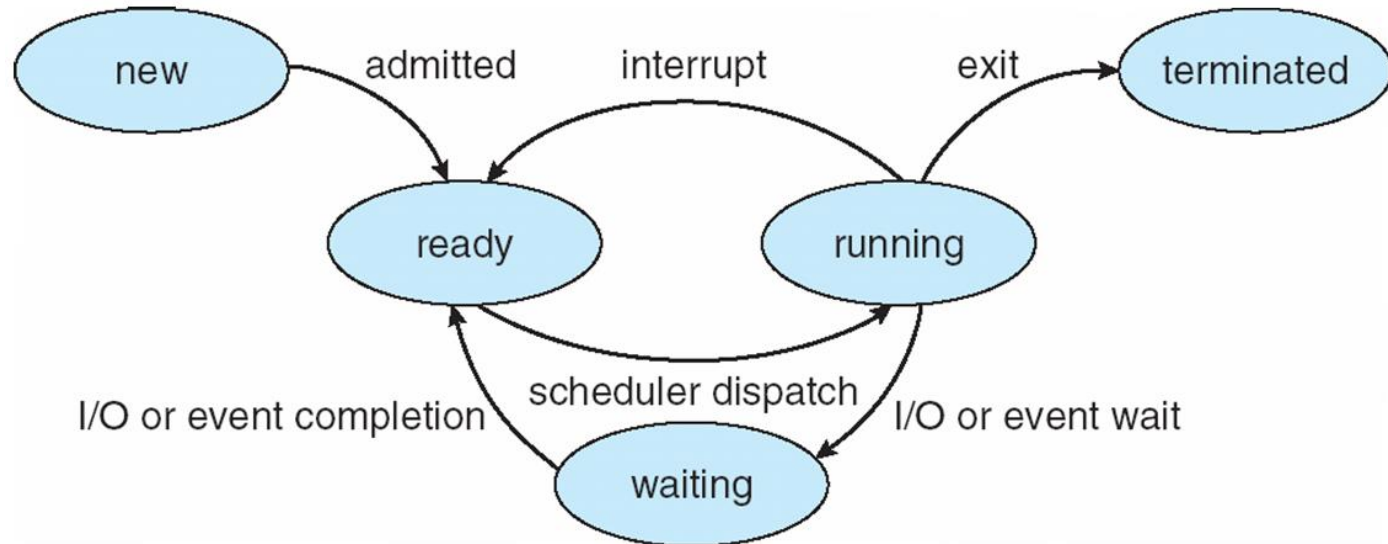
For example,

- An accounting program may need to record the processor time and other resources utilized by the process for billing purposes.
- A utility program may need to extract information about the history of the process for purposes related to performance or utilization analysis.

Terminated/Exit:

- Once these programs have extracted the needed information,
- The OS no longer needs to maintain any data relating to the process and the process is deleted from the system.

Process State Transition Diagram



- Only one process can be running on any processor at any instant, although many processes may be ready and waiting.

Process State Transition

Null to New:

- A new process is created to execute a program

New to Ready:

- The OS will move a process from the New state to the Ready state when it is prepared to take on an additional process.
- Most systems set some limit based on
 - 1) The number of existing processes or
 - 2) The amount of virtual memory committed to existing processes.
- This limit assures that there are not so many active processes as to degrade performance.

Ready to Running:

- When it is time to select a process to run, the OS chooses one of the processes in the Ready state.
- This is the job of the scheduler or dispatcher.

Running to Exit:

- The currently running process is terminated by the OS if the process indicates that it has completed, or if it aborts.

Running to Ready:

- The most common reason for this transition is that
- The running process has reached the maximum allowable time for uninterrupted execution.

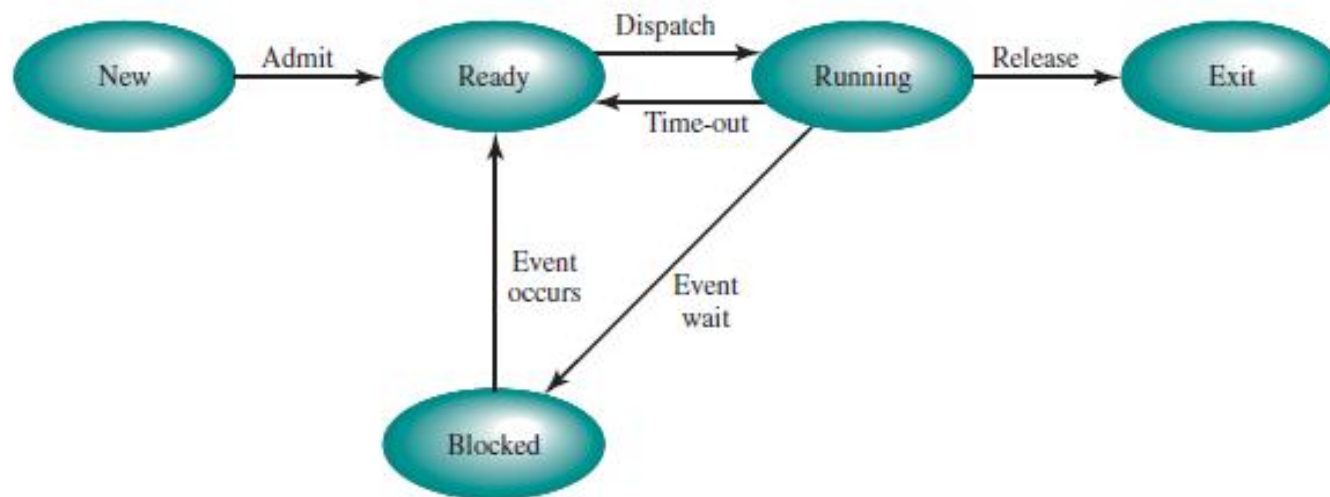


Figure 3.6 Five-State Process Model

Reference Book-William Stallings

Prof. Shweta Dhawan Chachra

Running to Ready:

- There are several other alternative causes for this transition-
- OS assigns different levels of priority to different processes.

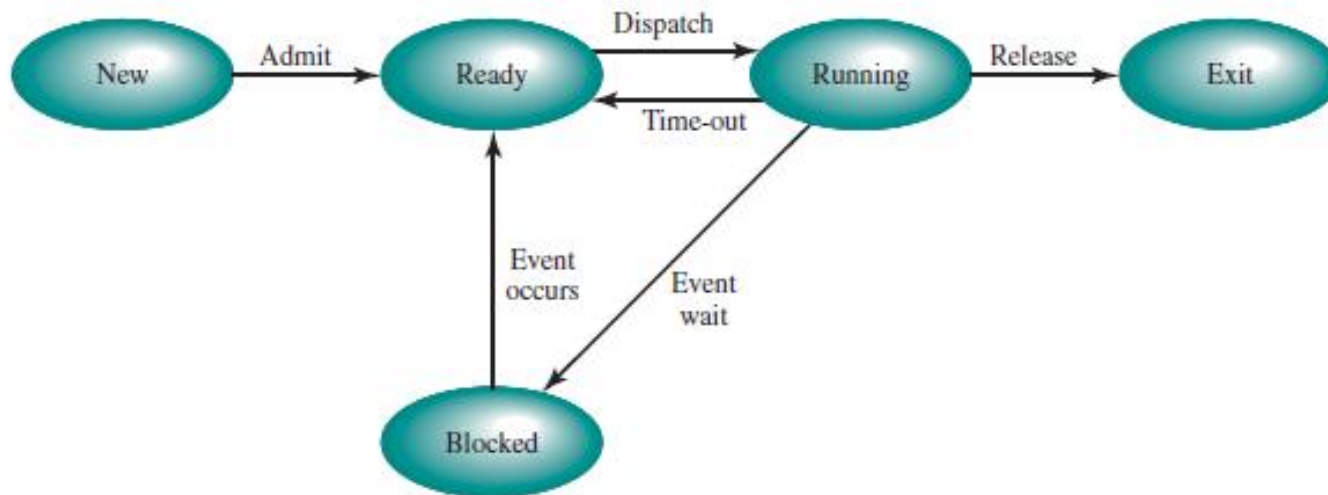


Figure 3.6 Five-State Process Model

Reference Book-William Stallings

Prof. Shweta Dhawan Chachra

Running to Ready:

For Example-

- Process A is running at a given priority level, and process B, at a higher priority level, is blocked.
- If the OS learns that the event upon which process B has been waiting has occurred, moving B to a ready state,
- Then it can interrupt process A and dispatch process B.
- Thus, the OS has pre-empted process A

Process Control Block (PCB)

Information associated
with each process
(also called **task control
block**)

Repository of any
information related to
Process



Process Control Block (PCB)

- **Process state** – running, waiting, etc
- **Program counter** – address of the next instruction to be executed for this process



Process Control Block (PCB)

CPU registers –

- contents of all process-centric registers,
- **Registers may vary in number and type depending on system architecture.**



Process Control Block (PCB)

CPU registers –

- They may include
 - **accumulators,**
 - **index registers ,**
 - **stack pointers and**
 - **general purpose registers**
 - **Any condition code information**
- **When interrupt occurs , then this state information along with program counter must be saved to be continued correctly afterward.**

Prof. Shweta Dhawan Chachra



Process Control Block (PCB)

CPU scheduling information-

- Process priorities,
- scheduling queue pointers,
- any other scheduling parameters.



Process Control Block (PCB)

Memory-management information –

- memory allocated to the process,
- value of the Base and limit registers,
- the page tables or segment tables.

Process Control Block (PCB)

Accounting information –

- CPU used,
- **clock time elapsed since start,**
- **time limits,**



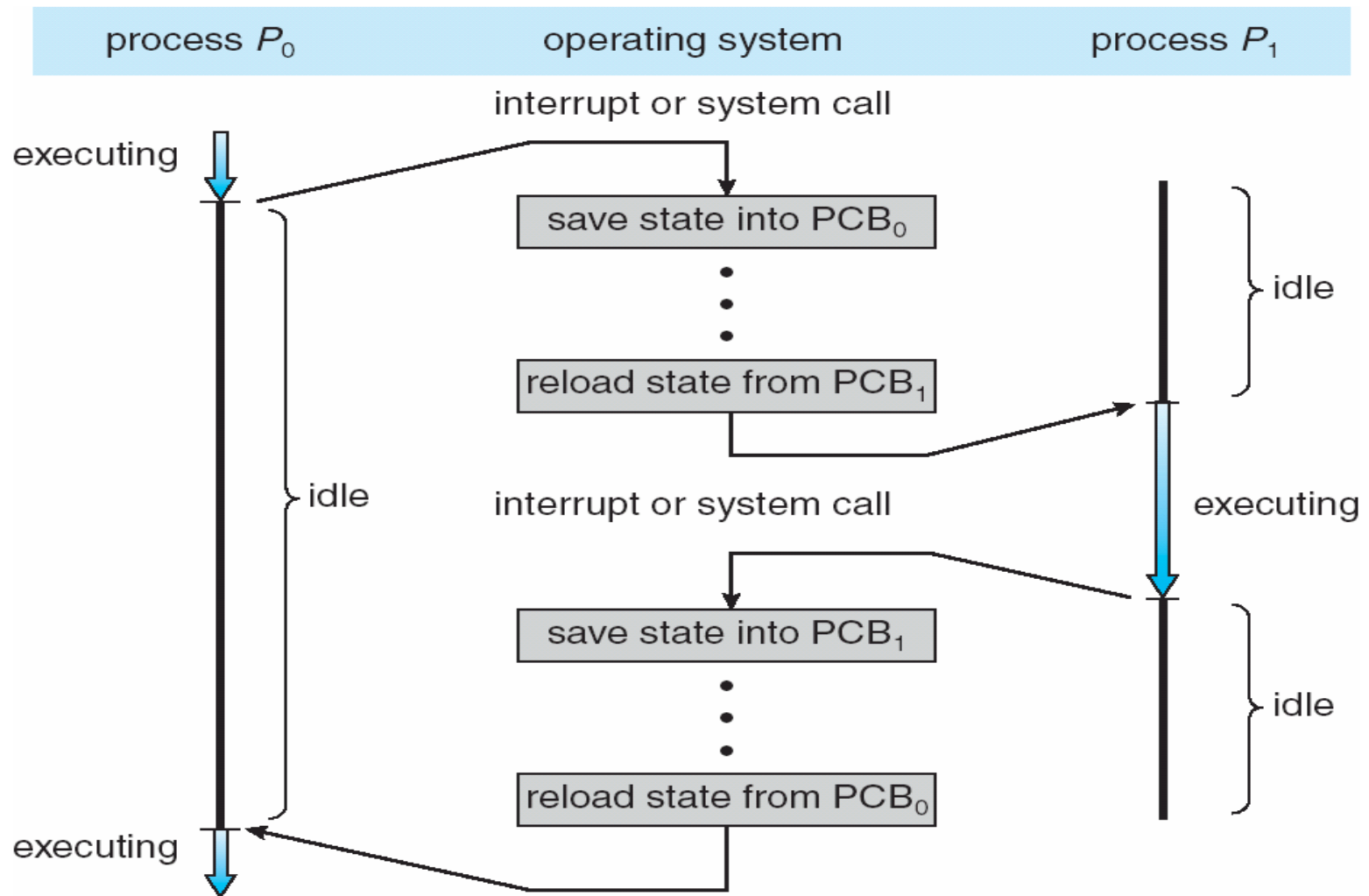
Process Control Block (PCB)

I/O status information –

- list of I/O devices allocated to process,
- **list of open files**



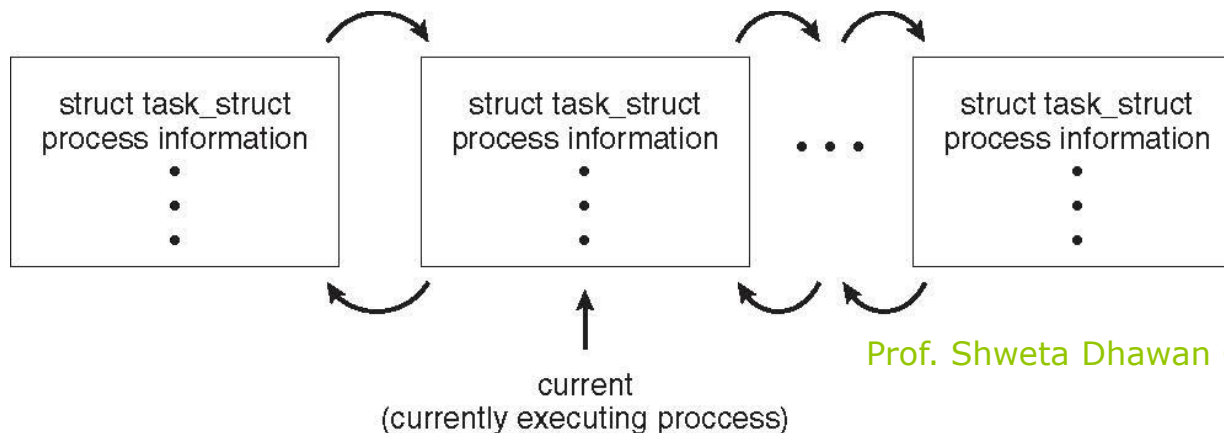
CPU Switch From Process to Process



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 */
```



Prof. Shweta Dhawan Chachra

What is a Thread?

- A thread is a path of execution within a process.
- **A process can contain multiple threads.**

Traditional Heavy Weight process

- Has **a single thread of control**.
- Process performs a single thread of execution.

Traditional Heavy Weight process

For Example,

- If a process is running a word processor program,
- A single thread of instructions is being executed.
- **The user could not simultaneously type in characters and run the spell checker within the same process.**

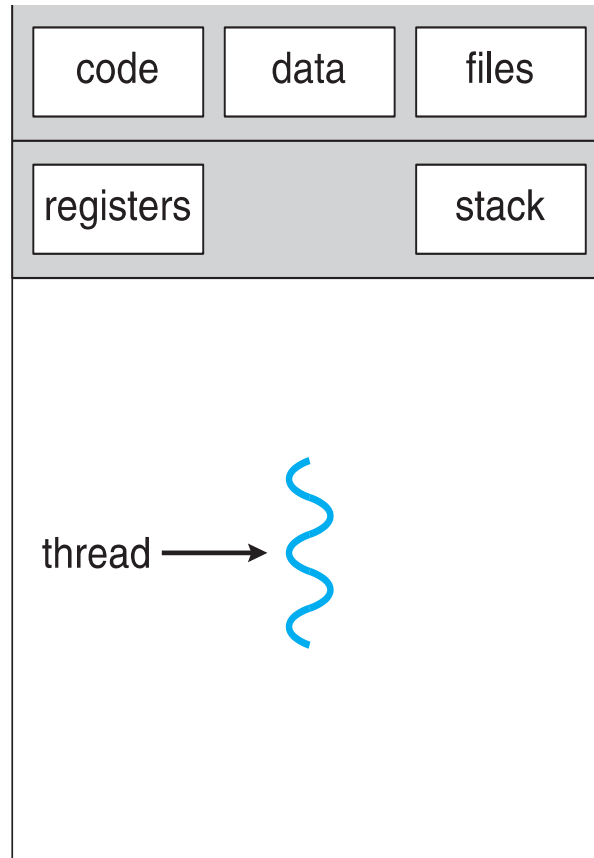
Multithreaded Process

- If a process has **multiple threads of control**,
- It can do **more than one task** at a time.

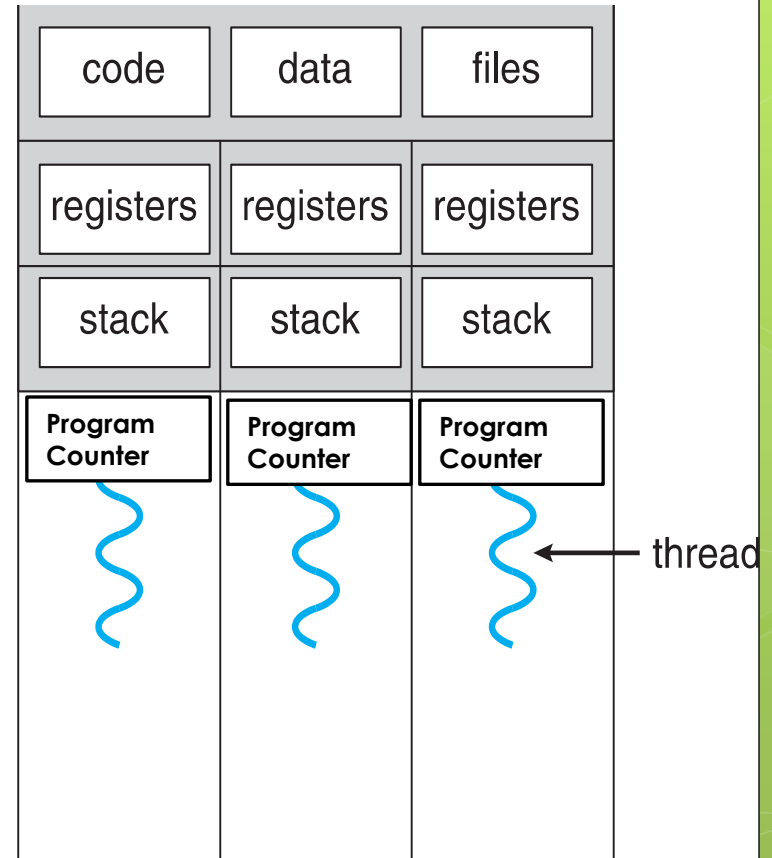
Why Multithreading?

- A thread is also known as lightweight process.
- The idea is to **achieve parallelism by dividing a process into multiple threads.**

Single and Multithreaded Processes

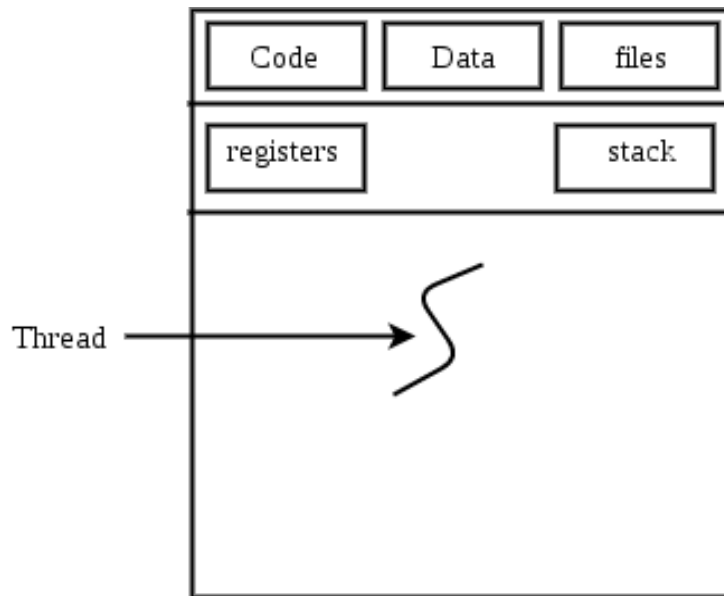


single-threaded process

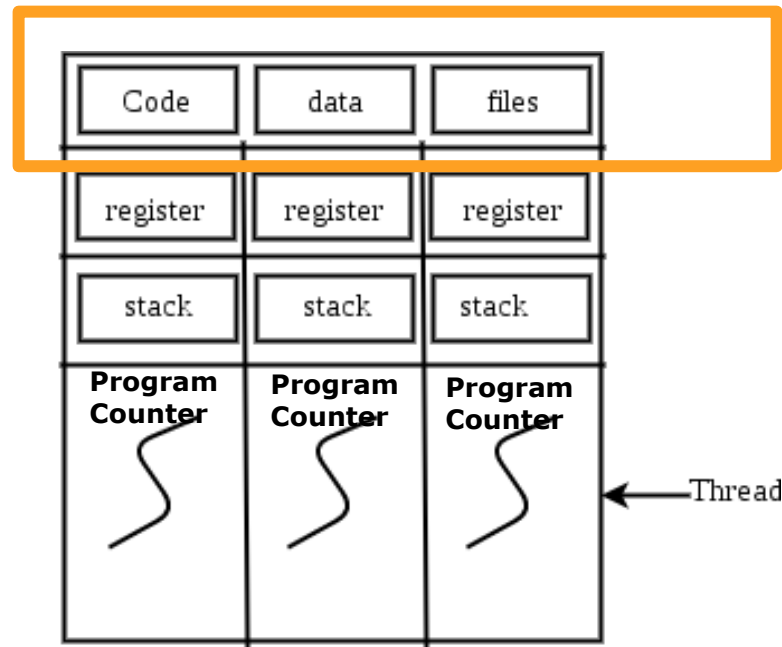


multithreaded process

- Threads are not independent of one another like processes
- **Threads share with other threads their**
 - code section,
 - data section,
 - OS resources (like open files and signals).



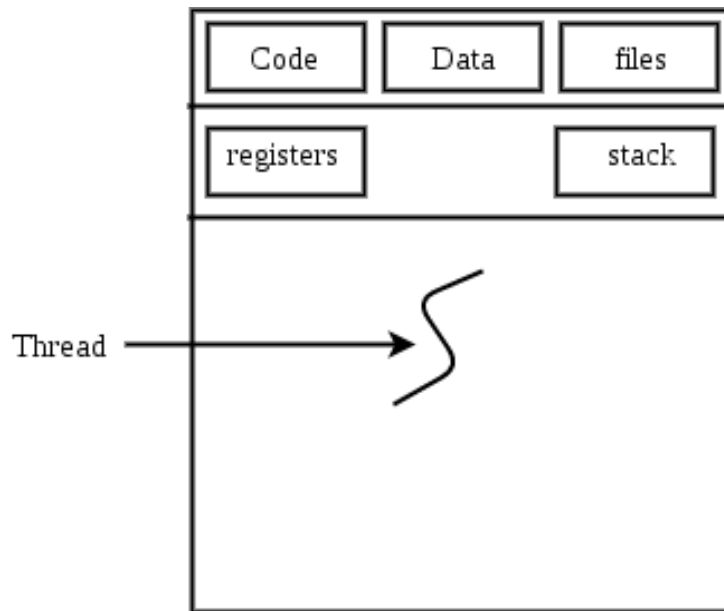
Single Threaded process



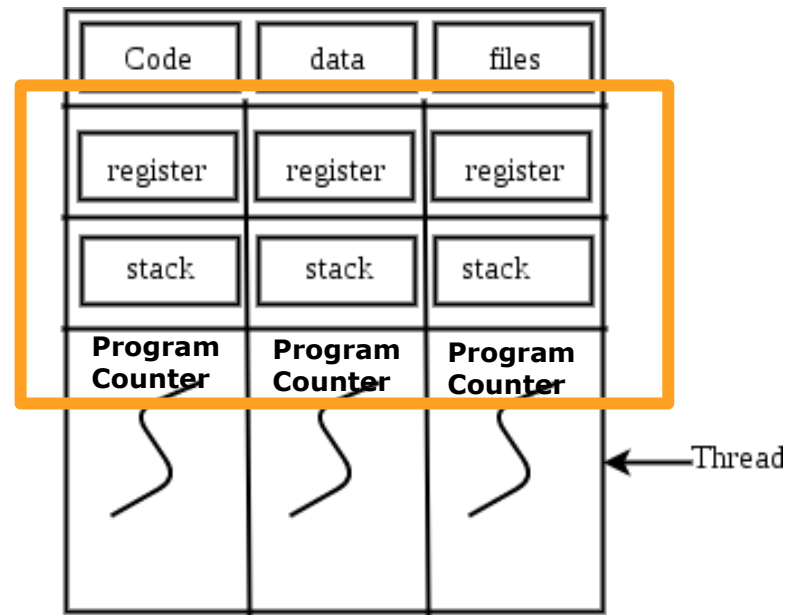
Multithreaded Process

Threads

- But, like process, a thread **has its own**
 - program counter (PC),**
 - register set, and**
 - stack space.**



Single Threaded process



Multithreaded Process

Prof. Shweta Dhawan Chachra

Process vs Thread?

- Threads within the same process run in a shared memory space, while processes run in separate memory spaces.
- Threads are not independent of one another like processes

Process & Threads

- Opening a new browser (say Chrome, etc) is an example of creating a process. At this point, a new process will start to execute.
- On the contrary, opening multiple tabs in the browser is an example of creating the thread.

Advantages of Thread over Process

1. **Responsiveness:** If the process is divided into multiple threads, **if one thread completes its execution, then its output can be immediately returned.**
2. **Faster context switch:**
 - Context switch time between threads is lower compared to process context switch.
 - Process context switching requires more overhead from the CPU.

Advantages of Thread over Process

3. *Resource sharing*:

Resources like **code, data, and files** can be shared among all threads within a process.

Note: **stack and registers can't be shared among the threads.**

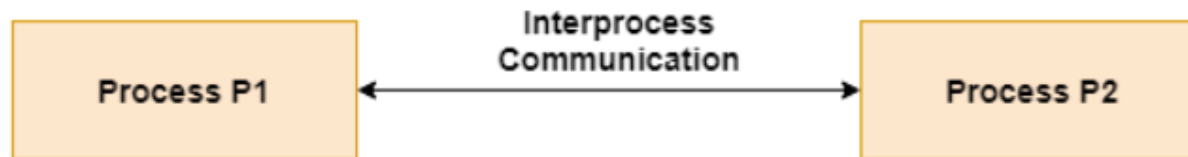
Each thread has its own stack and registers.

Advantages of Thread over Process

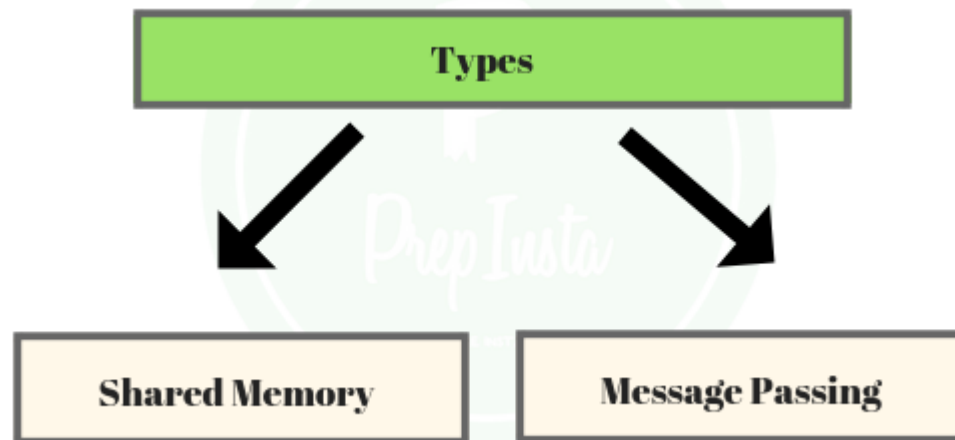
4. Communication:

- Communication between multiple threads is easier, as the **threads shares common address space**,
- while in process we have to follow some **specific communication technique** for communication between two process.

- Cooperating processes need **interprocess communication (IPC)** mechanism to exchange data and information



- Two models of IPC
 - Shared memory
 - Message passing



Advantages of Thread over Process

5. *Effective utilization of multiprocessor system:*
If we have multiple threads in a single process, then we can **schedule multiple threads on multiple processor**.
This will have faster execution .

Types of Threads

There are two types of threads.

- User Level Thread
- Kernel Level Thread

User Threads and Kernel Threads

- **User threads** - management done by **user-level threads library**
- Three primary thread libraries:
 - POSIX **Pthreads**
 - Windows threads
 - Java threads
- **Kernel threads** - **Supported by the Kernel**
- Examples – virtually all general purpose operating systems, including:
 - Windows
 - Solaris
 - Linux
 - Tru64 UNIX
 - Mac OS X

User Threads

- Implemented by a **thread library at the user level**

- The library provides support for
 - **thread creation,**
 - **scheduling and**
 - **management** in User space
 - with **no support /no intervention from the kernel.**

Kernel Threads

- Supported **Directly by the OS**
- Kernel performs
 - **thread creation,**
 - **scheduling and**
 - **management** in Kernel space.

User Threads

- Generally **fast to create**
- If the **Kernel is single threaded**,
 - then **any user level thread performing a blocking system call**
 - **will cause the entire process to block**,
 - even if other threads are available to run within the application

Kernel Threads

- Generally **slower to create** and manage than user threads
- If a thread performs a **blocking system call**,
 - **the kernel can schedule another thread** in the application for execution.

Multithreading Models

Many systems provide support for both user and kernel threads, resulting in different multithreading models

- 1) **Many-to-One**
- 2) **One-to-One**
- 3) **Many-to-Many**

Imp

- By default, an application begins with a single thread and begins running in that thread.
- This application and its thread are allocated to a single process managed by the kernel.

ULT and KLT

- At any time that the application is running (the process is in the Running state), the application may spawn a new thread to run within the same process.

ULT and KLT

- Spawning is done by invoking the spawn utility in the threads library.
- Control is passed to that utility by a procedure call.

ULT and KLT

- The threads library creates a data structure for the new thread
- Then passes control to one of the threads within this process that is in the Ready state, using some scheduling algorithm.

ULT and KLT

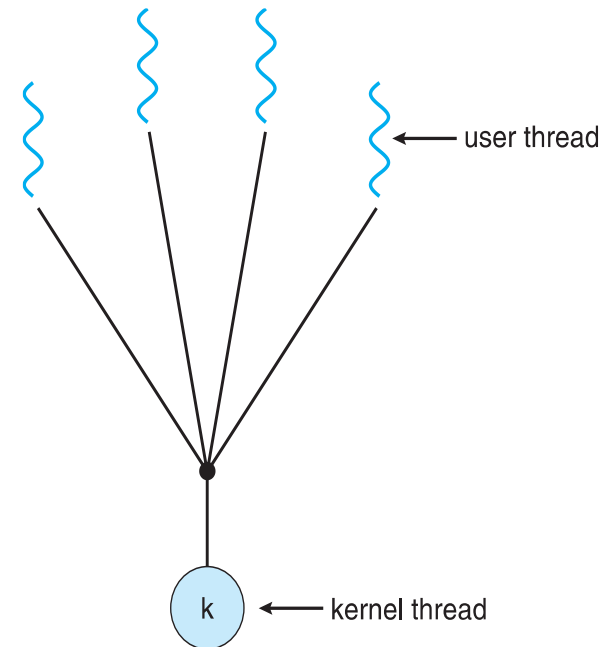
- When control is passed to the library, the context of the current thread is saved,
- and when control is passed from the library to a thread, the context of that thread is restored.
- The context essentially consists of the contents of user registers, the program counter, and stack pointers.

ULT and KLT

- All of the activity described in the preceding above takes place in user space and within a single process.
- The kernel is unaware of this activity. The kernel continues to schedule the process as a unit and assigns a single execution state (Ready, Running, Blocked, etc.) to that process

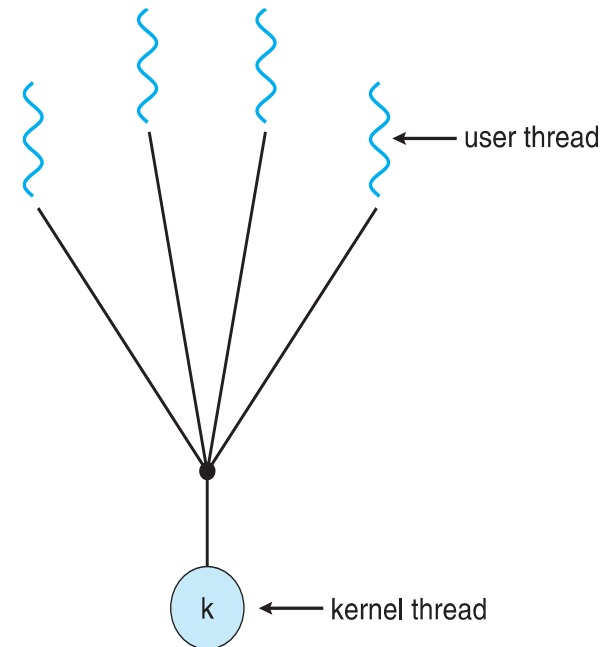
Many-to-One

- Many user-level threads mapped to single kernel thread
- Only one thread can access the Kernel at a time.**
- Thread Management done in User space, so is efficient.
- If one thread makes a blocking system call, the entire process will block.**
- One thread blocking causes all to block



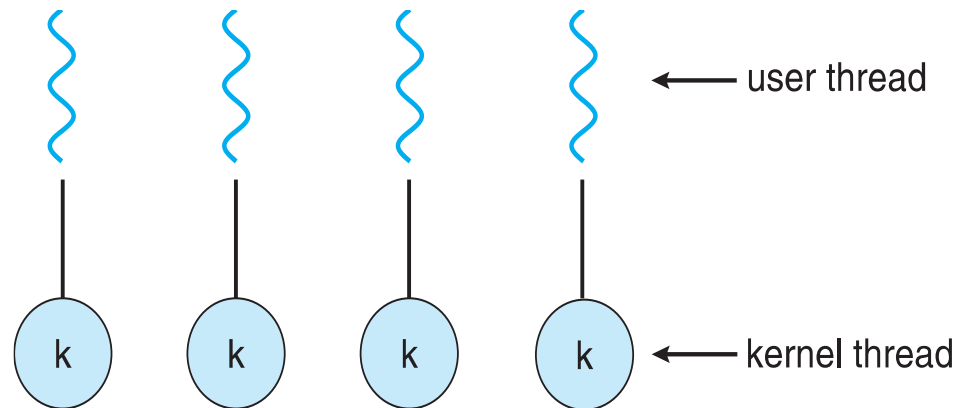
Many-to-One

- Multiple threads are unable to run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
 - Solaris Green Threads**
 - Green thread, a thread that uses this model
 - Available for Solaris 2
 - GNU Portable Threads**



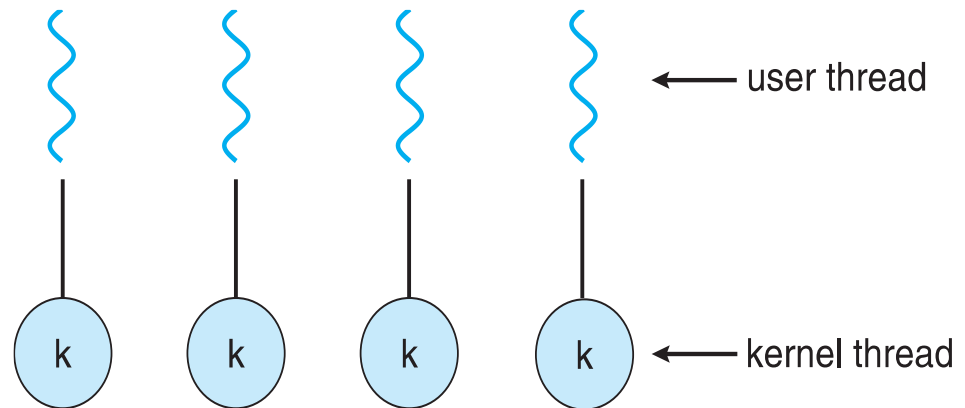
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread**
- More concurrency than many-to-one
 - By allowing another thread to run when a thread makes a blocking system call**
- Allows multiple threads to run in parallel on multiprocessors



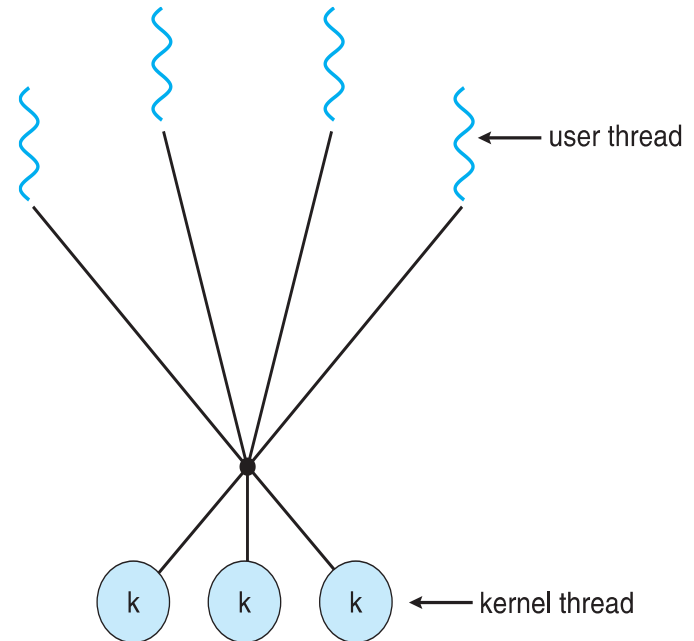
One-to-One

- Creating a user thread requires creating the corresponding kernel thread.
- Overhead of creating kernel threads can burden the performance of an application**
- Number of threads per process sometimes restricted due to overhead**
- Examples
 - Windows
 - Linux
 - Solaris 9 and later



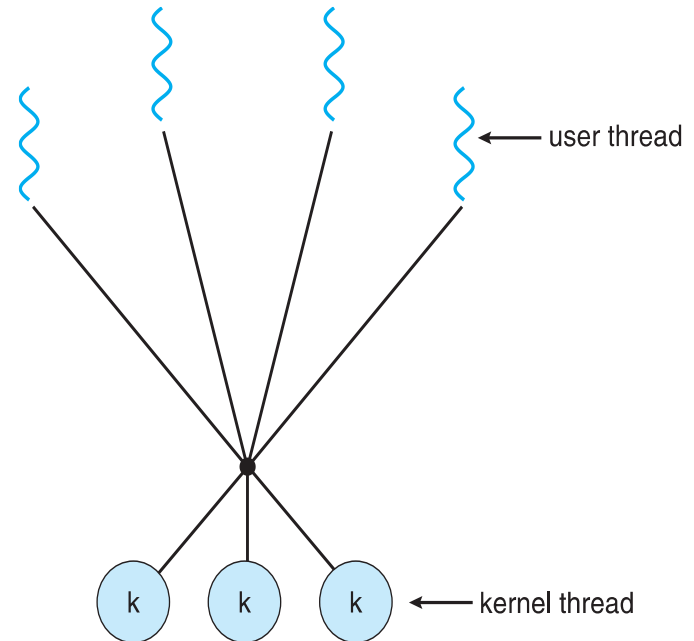
Many-to-Many Model

- **Allows many user level threads to be mapped to many kernel threads**
- Allows the operating system to create a sufficient number of kernel threads



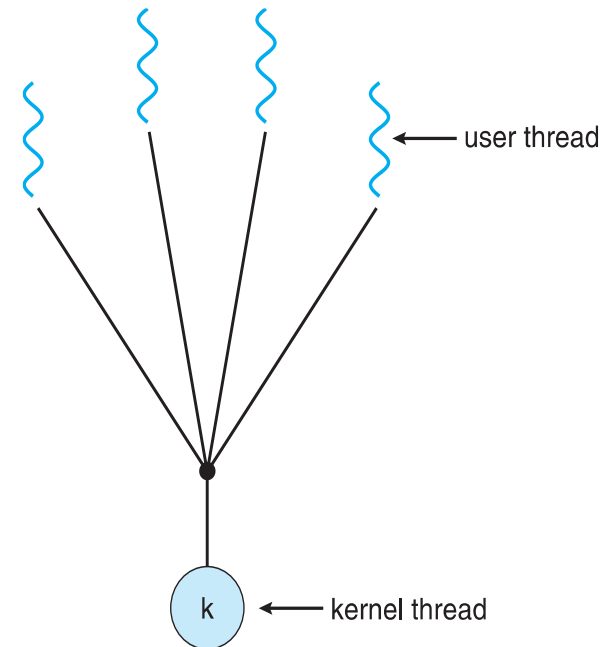
Many-to-Many Model

- The number of kernel threads may be specific to
 - Either a particular application
 - Or a particular machine
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



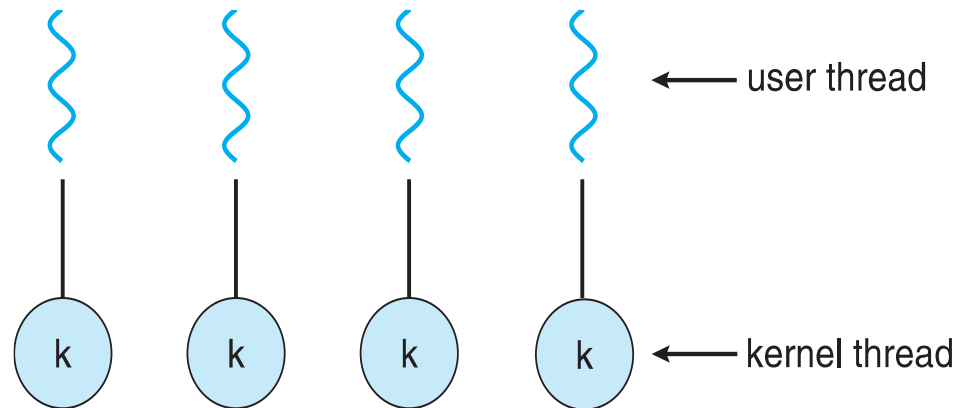
Drawbacks of Many-to-One Model

- **Many to One**
 - Allows developer to create as many user threads as needed
- **But true concurrency is not gained because kernel can schedule only one thread at a time**



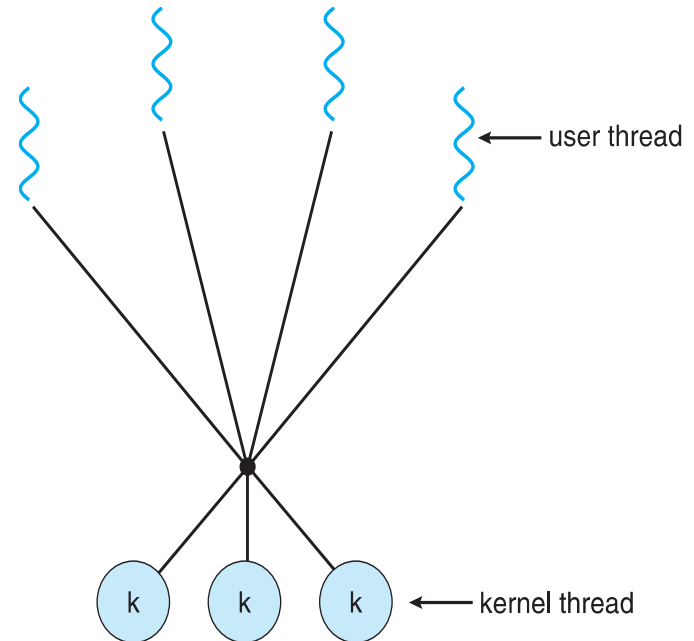
Drawbacks of One-to-One Model

- **One to One**
 - Allows for greater concurrency, but
 - **the developer has to be careful not to create too many threads within an application**



Advantages of Many-to-Many Model

- Many to Many
 - Overcomes these shortcomings
- Developers can create as many user threads as necessary
- Corresponding Kernel threads can run in parallel on a multiprocessor
- When a thread performs a blocking system call, the kernel can schedule another thread for execution









Now,
Let's compare User and
Kernel Level Threads

Difference between User Level thread and Kernel Level thread

USER LEVEL THREAD	KERNEL LEVEL THREAD
User thread are implemented by users.	kernel threads are implemented by OS.
OS doesn't recognized user level threads.	Kernel threads are recognized by OS.
Implementation of User threads is easy.	Implementation of Kernel thread is complicated.
Context switch time is less.	Context switch time is more.
Context switch requires no hardware support.	Hardware support is needed.
If one user level thread perform blocking operation then entire process will be blocked.	If one kernel thread perform blocking operation then another thread can continue execution.
User level threads are designed as dependent threads.	Kernel level threads are designed as independent threads.
Example : Java thread, POSIX threads.	Example : Window Solaris.

Done with Process and Threads

Now,
Lets compare Process and Threads

S.NO	PROCESS	THREAD
1.	Process means any program is in execution. 	Thread means segment of a process. 
2.	Process takes more time to terminate.	Thread takes less time to terminate.
3.	It takes more time for creation.	It takes less time for creation.
4.	It also takes more time for context switching. 	It takes less time for context switching. 
5.	Process is less efficient in term of communication. 	Thread is more efficient in term of communication. 
6.	Process consume more resources.	Thread consume less resources.
7.	Process is isolated. 	Threads share memory. 
8.	Process is called heavy weight process.	Thread is called light weight process.
9.	Process switching uses interface in operating system.	Thread switching does not require to call a operating system and cause an interrupt to the kernel.

NO.	THREAD CONTEXT SWITCH	PROCESS CONTEXT SWITCH
1.	TCS occurs when the CPU saves the current state of the thread and switches to another thread of the same process.	PCS occurs when the operating system's scheduler saves the current state of the running Program (including the state of PCB) and switches to another program.
2.	TCS helps the CPU to handle multiple threads simultaneously.	PCS involves loading of the states of the new program for its execution.
3.	TCS does not involve switching of memory address spaces. All the memory addresses that the processor accounts remain saved.	PCS involves switching of memory address spaces. All the memory addresses that the processor accounts get flushed.
4.	Processor's cache and Translational Lookaside Buffer preserve their state.	Processor's cache and TLB get flushed.
5.	Though TCS involves switching of registers and stack pointers, it does not afford the cost of changing the address space. Hence it is more efficient.	PCS involves the heavy cost of changing the address space. Hence it is less efficient.
6.	TCS is a bit faster and cheaper.	PCS is relatively slower and costlier.

Prof. Shweta Dhawan Chachra

Process Scheduling

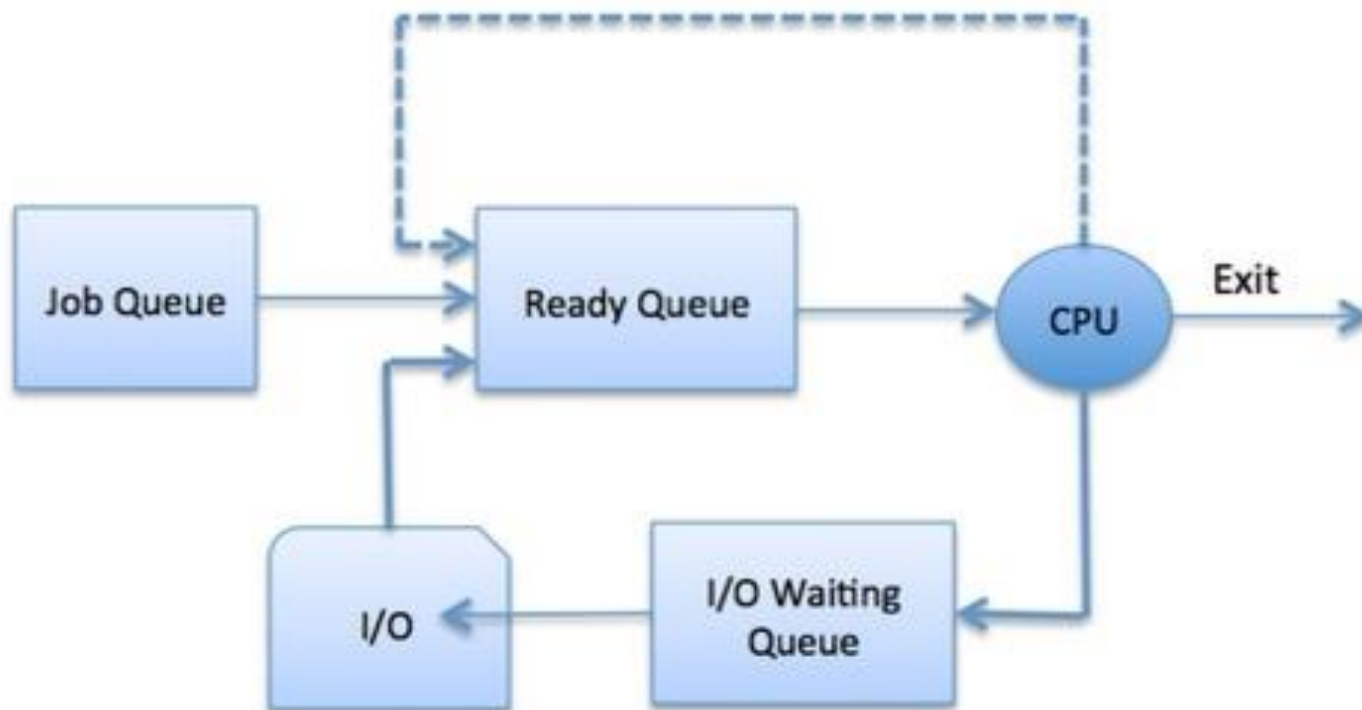
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU

Process Scheduling

- Maintains **scheduling queues** of processes
 - **Job queue**
 - **Ready queue**
 - **Device queues**

Process Scheduling

- Processes migrate among the various queues



Job queue –

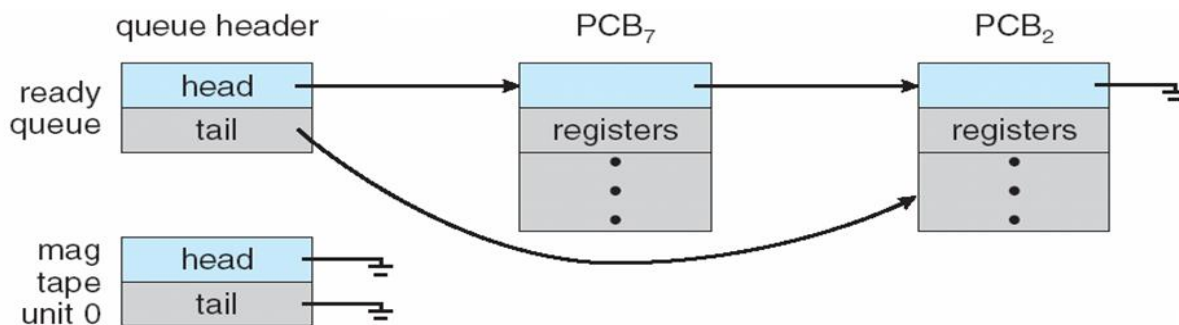
- Each new process goes into the job queue.
- Processes in the job queue reside on mass storage and await the allocation of main memory.
- as a process enters the system,
 - they are put on job queue..

Ready queue –

- set of all processes residing in main memory, **ready and waiting to execute**,
- The set of all processes that are in main memory and are waiting for CPU time are kept in the ready queue.

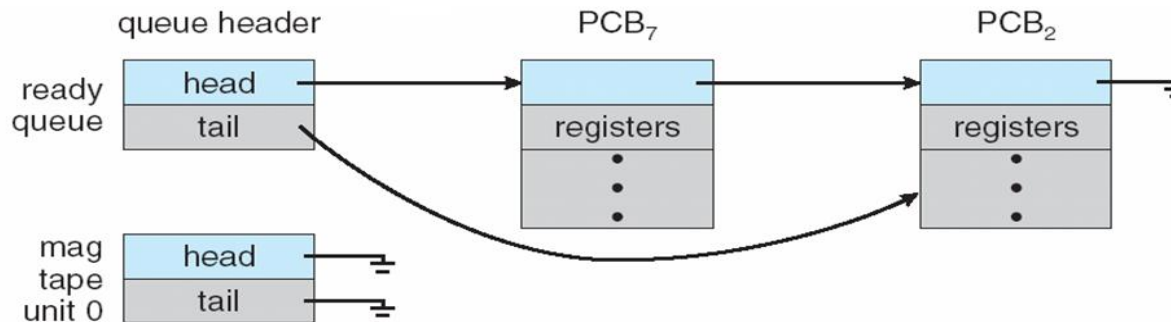
Ready queue –

- Generally **stored using a linked list**.
- A ready queue header contains **pointers to the first and final PCB in the list**.



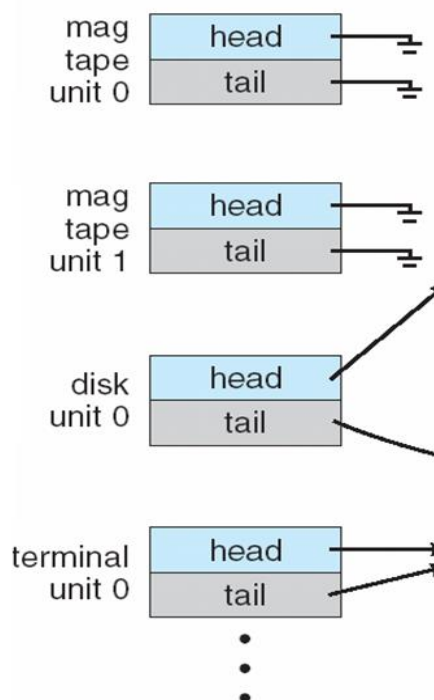
Ready queue –

- We extend each **PCB** to include a pointer field that points to the next PCB in the ready queue.



Device queues –

- Set of processes waiting for an I/O device.
- Each device has its own device queue



- Processes migrate among the various queues

Summary of Queues

- **Job Queue:**

- Each new process goes into the job queue.
- Processes in the job queue reside on mass storage and await the allocation of main memory.

- **Ready Queue:**

- The set of all processes that are in main memory and are waiting for CPU time is kept in the ready queue.

- **Waiting (Device) Queues:**

- The set of processes waiting for allocation of certain I/O devices is kept in the waiting (device) queue.

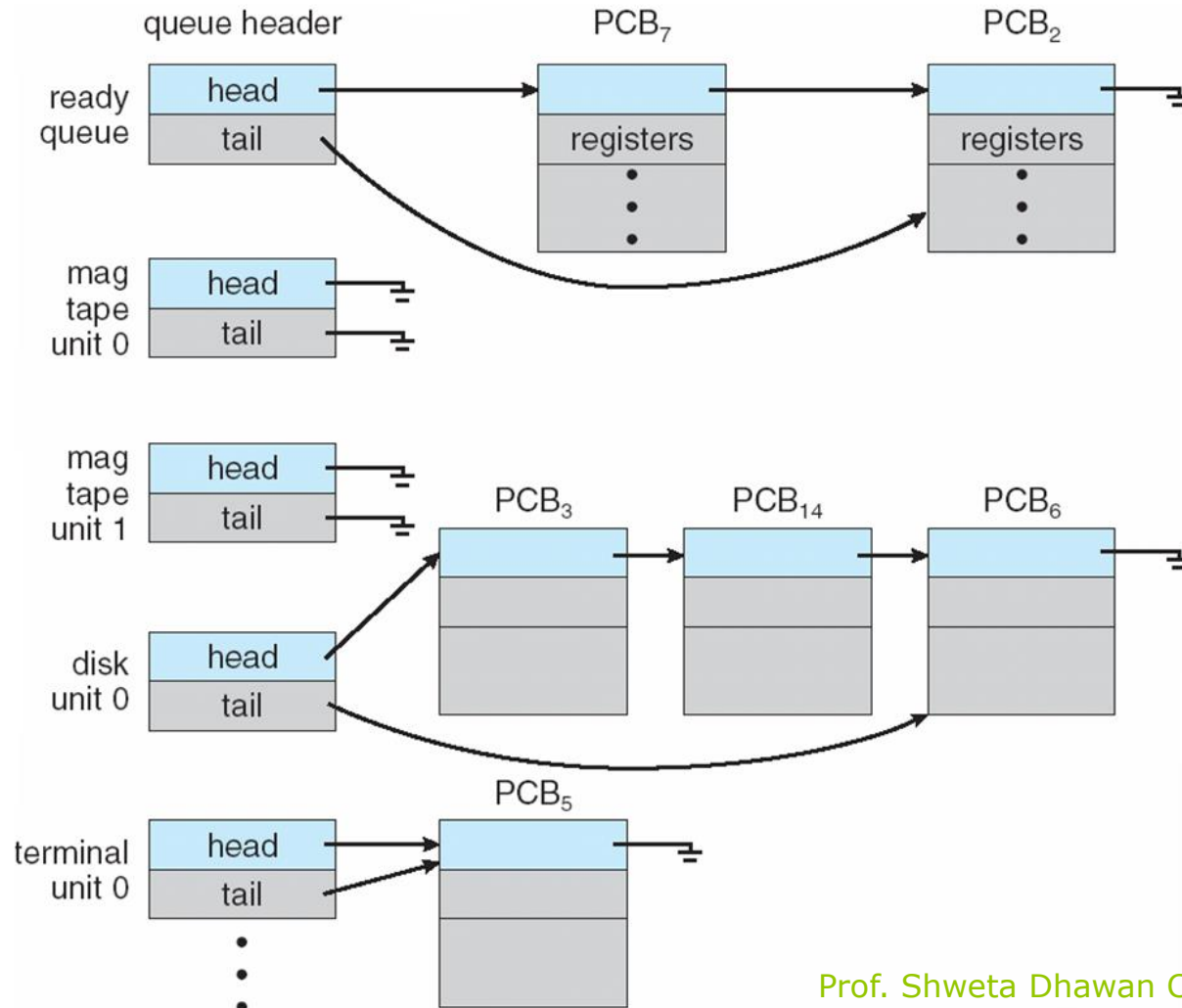
Prof. Shweta Dhawan Chachra

<https://cs.stackexchange.com/questions/1106/which-queue-does-the-long-term-scheduler-maintain>

Ready Queue And Various I/O Device Queues

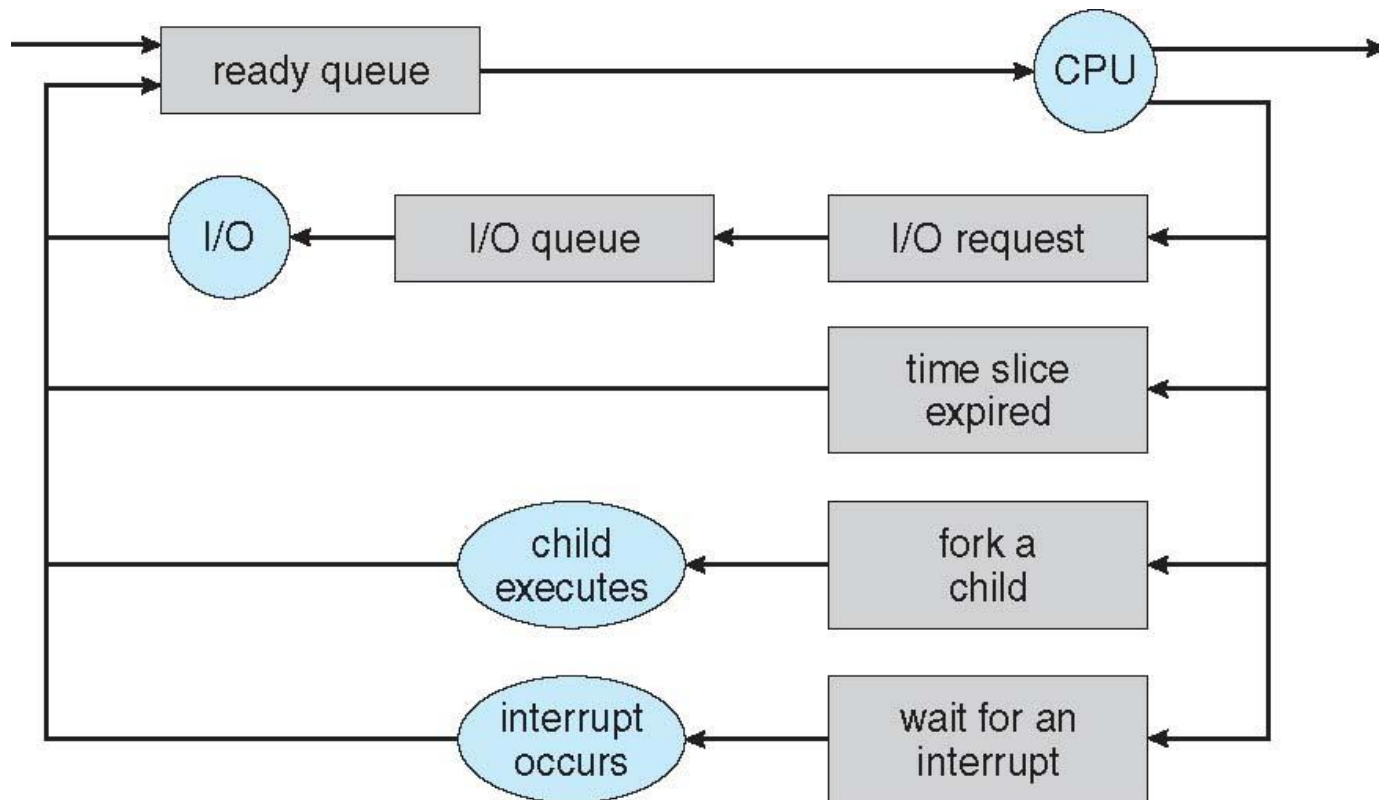
97

15/09/2024



Prof. Shweta Dhawan Chachra

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

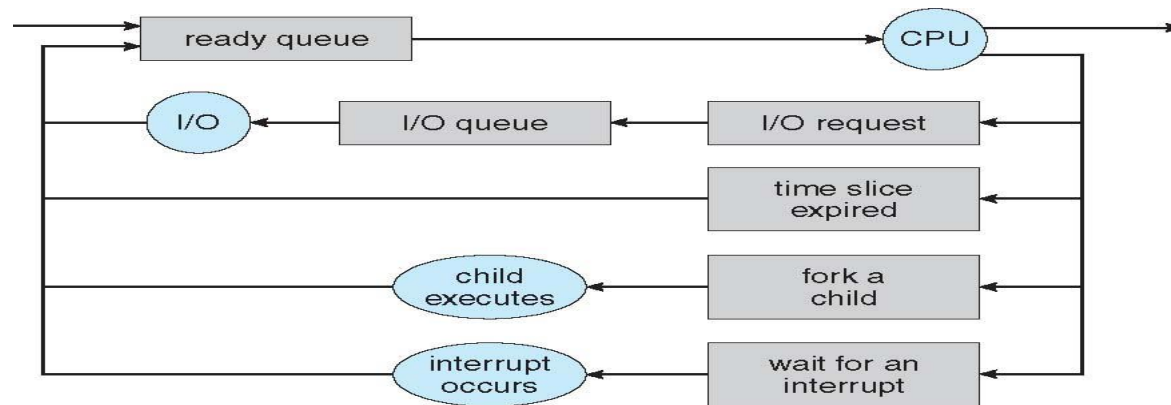


Representation of Process Scheduling

99

15/09/2024

- ❑ Rectangular box=queue
- ❑ Circles= resources that serve the queues
- ❑ Arrows=indicate the flow of processes



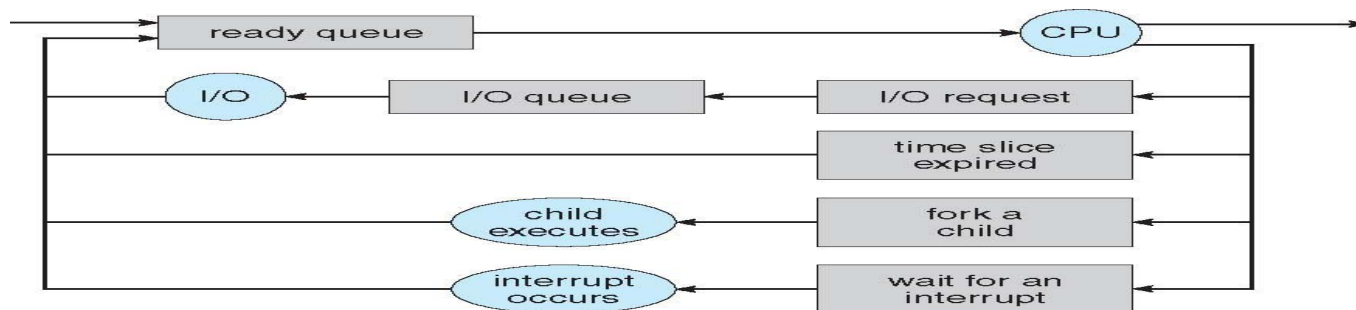
Prof. Shweta Dhawan Chachra

Representation of Process Scheduling

100

15/09/2024

- A new process is initially put in the ready queue, waits in the ready queue to be dispatched.
- Once assigned to the CPU and is executing, one of the several events may occur, **the process could:**
 - **Issue I/O request** and be placed in I/O queue
 - **Create a new sub process** and wait for its termination
 - **Be Removed forcibly** from CPU due to interrupt and put back in the ready queue
 - **Finally when terminates** is removed from all queues, PCBs and resources deallocated.



Prof. Shweta Dhawan Chachra

- OS must select processes from these queues in some fashion.
- The selection process is carried out by the appropriate scheduler
- The schedulers are :
 - **Long-term scheduler** (or **job scheduler**)
 - **Short-term scheduler** (or **CPU scheduler**)
 - **Medium-term scheduler**

Long-term scheduler

- Also known as **Job Scheduler**.
- Selects which **processes should be brought into the ready queue**
- Long-Term Scheduler changes the process state
 - **From New to Ready.**
 - **I.e. From Disk to Main Memory**
- **invoked infrequently (seconds, minutes) \Rightarrow (may be slow)**

Long-term scheduler (or job scheduler)

- The long-term scheduler controls
 - the **degree of multiprogramming**
 - **i.e. the number of processes in memory.**
 - **If the degree of multiprogramming is stable then:**
 - **average rate of process creation= average departure rate of process leaving the system**

Long-term scheduler (or job scheduler)

- On some systems, **the long term scheduler may be absent or minimal**
- **Time sharing systems like UNIX often has no LTS.** The stability of such system depends on
 - physical limitations
 - such as no of available terminals or
 - self adjusting nature of human users
 - If performance declines to unacceptable limits, some users will simply quit .

Long-term scheduler (or job scheduler)

- 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;
 - generates I/O requests infrequently,
 - few very long CPU bursts

Long-term scheduler (or job scheduler)

- Long-term scheduler strives for good **process mix of both**
- Why?**

Long-term scheduler (or job scheduler)

- Long-term scheduler strives for good **process mix of both**
 - **If all Processes are I/O bound,**
 - the ready queue will almost always be empty and
 - STS will have little to do
 - CPU goes underutilized
 - **If all Processes are CPU bound,**
 - I/O waiting queue will almost always be empty,
 - devices will go unused.

Short-term scheduler (or CPU scheduler)

- Selects processes from the ready queue and allocates the CPU for execution
- Selects which process should be executed next

Short-term scheduler (or CPU scheduler)

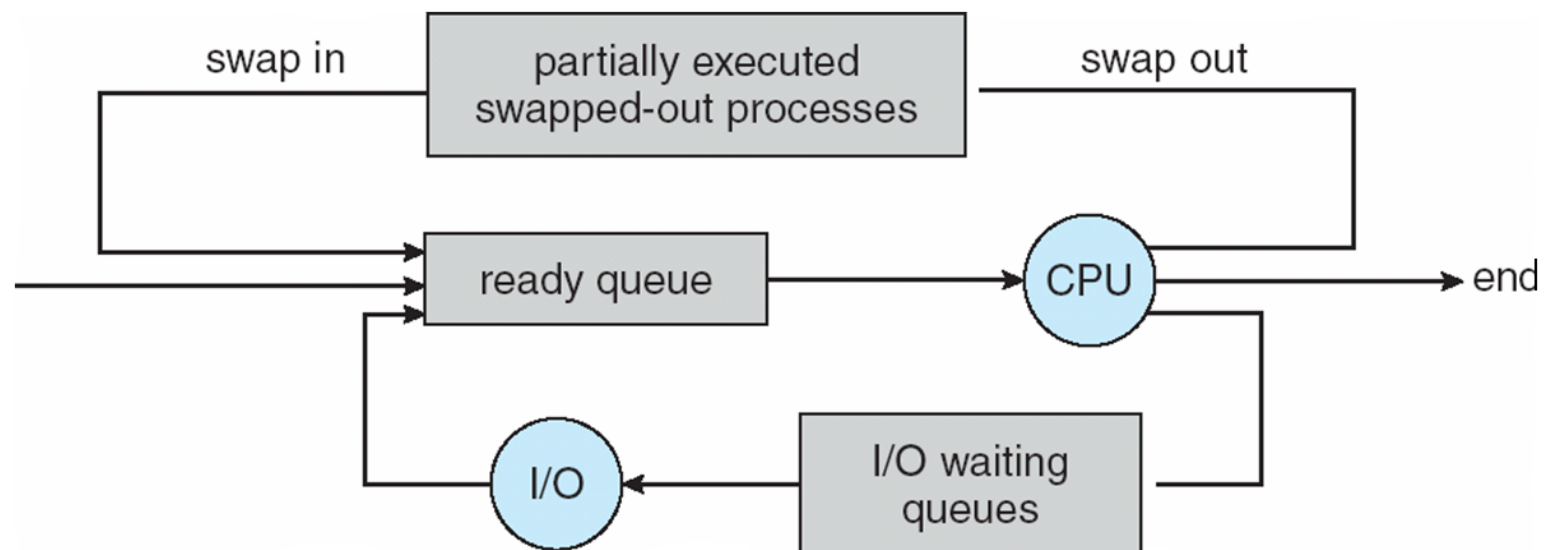
- ◉ **Sometimes the only scheduler** in a system
- ◉ Short-term scheduler is **invoked frequently (milliseconds)**
- ◉ Also known as **CPU scheduler**
- ◉ As it Selects a process from the ready queue and yields control of the CPU to the process.

Short-term scheduler (or CPU scheduler)

- It must be fast,
 - as a process may execute for only a few milliseconds
 - before waiting for an I/O request,
 - so STS needs to select another process from ready queue.
- Often STS executes at least **once every 100 milliseconds**.

Addition of Medium Term Scheduling

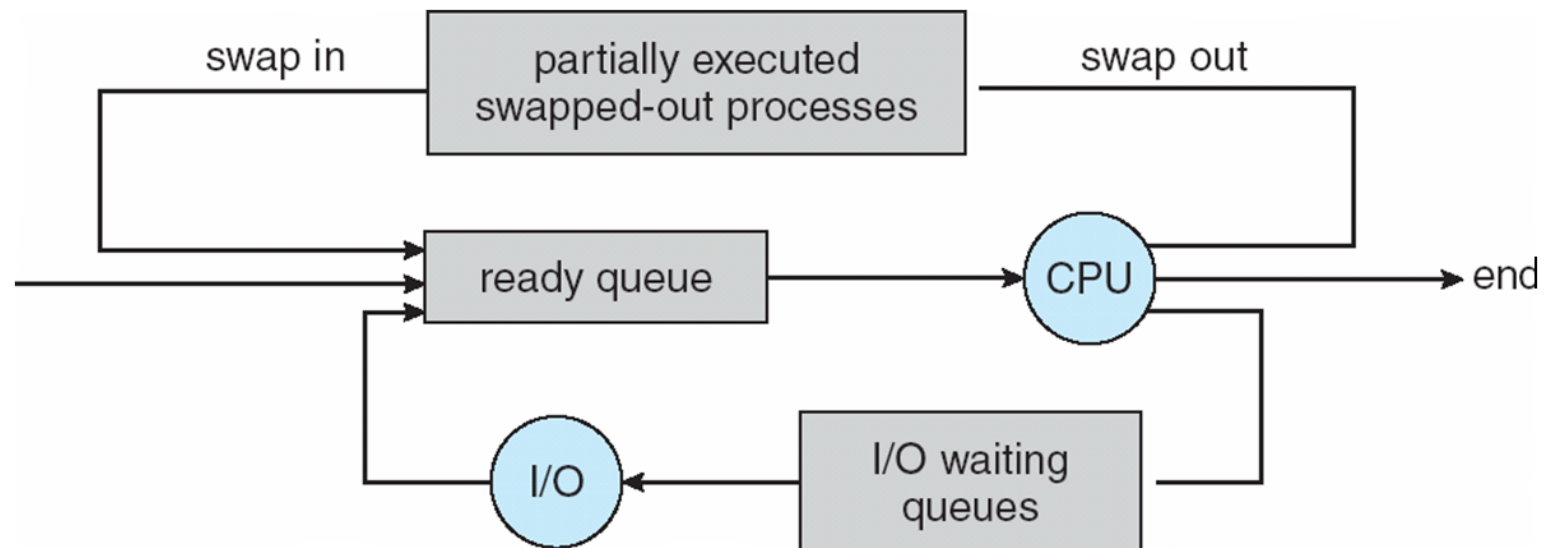
- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
 - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



Addition of Medium Term Scheduling

Why?

- ❑ To improve the process mix
- ❑ Memory requirements have overcommitted available memory, Requiring memory to be freed up.



- **Context** of a process represented in the PCB
- 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**
- Kernel saves context of old process in its PCB and
- loads the saved contents of new process from its PCB to run.

- Context-switch time is overhead;
 - **the system does no useful work while switching**
 - Varies from m/c to m/c
 - **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,
 - involves changing the pointer to the current register set.

- System must provide mechanisms for:
 - process creation,
 - process termination

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

- Resource sharing options(CPU time, memory, files, I/O devices)
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources (child obtains resources directly from OS)

- Restricting a child process to a subset of parents resources
 - prevents any process from overloading the system
 - by creating too many sub processes.

- **Execution options**

- Parent and children execute concurrently
- Parent waits until children terminate

- **In terms of Address space**
 - Child duplicate of parent
 - Child has a separate program loaded into it

Process Creation-In terms of Address space

- `fork()`
- `exec()`

Process Creation-In terms of Address space

- fork() creates a new process by duplicating the calling process,
- The new process, referred to as child, is an exact duplicate of the calling process, referred to as parent

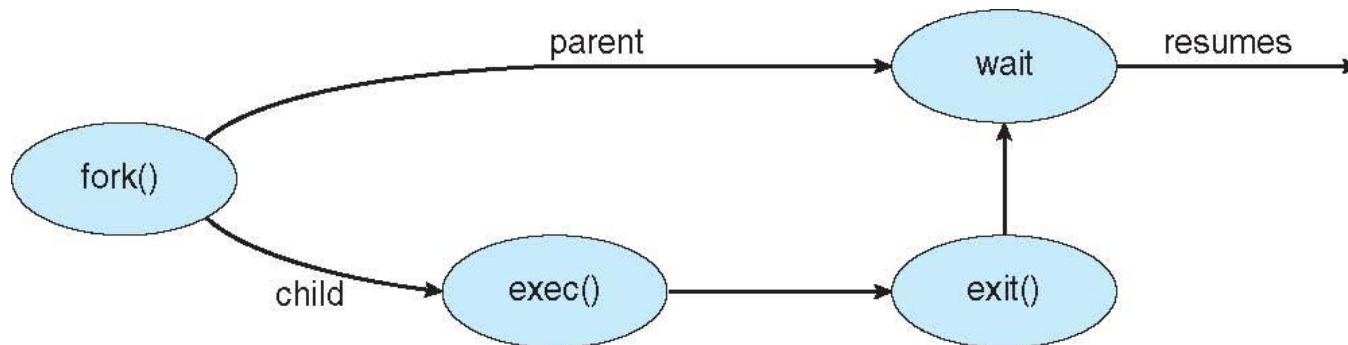
Process Creation-In terms of Address space

- fork()
 - 1) The child has its **own unique process ID**, and this PID **does not match the ID of any existing process group**.
 - 2) The **child's parent process ID is the same as the parent's process ID**.
 - 3) The child **does not inherit its parent's memory locks and semaphore** adjustments.
 - 4) The child **does not inherit outstanding asynchronous I/O** operations from its parent nor does it inherit any asynchronous I/O contexts from its parent.

Process Creation-In terms of Address space

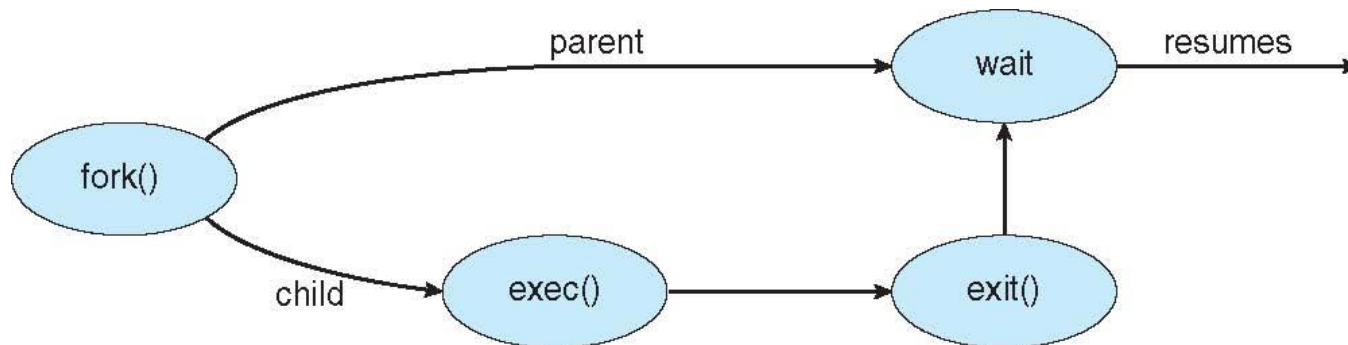
- The exec() family of functions
 - **replaces the current process image with a new process image.**
 - It loads the program into the current process space and runs it from the entry point.

- 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 vs Exec

- fork starts a new process which is a copy of the one that calls it, while
- exec replaces the current process image with another (different) one.**



execvp -- Overlay Calling Process and Run New Program

- The execvp function is most commonly used to overlay a process image that has been created by a call to the fork function.
- execvp replaces the calling process image with a new process image.
- This has the effect of running a new program with the process ID of the calling process.
 - Note that a new process is not started;
 - the new process image simply overlays the original process image.

Process Creation-In terms of Address space

- The exec() family consists of following functions
 - `int execl(const char *path, const char *arg, ...);`
 - `int execlp(const char *file, const char *arg, ...);`
 - `int execlx(const char *path, const char *arg, ..., char *const envp[]);`
 - `int execv(const char *path, char *const argv[]);`
 - `int execvp(const char *file, char *const argv[]);`
 - `int execvpe(const char *file, char *const argv[], char *const envp[]);`

execvp -- Overlay Calling Process and Run New Program

```
int execvp(const char *path, const char *arg0, ..., NULL);
```

- path
 - **identifies the location of the new process image** within the hierarchical file system (HFS).
- arg0, ..., NULL
 - is a **variable length list of arguments that are passed to the new process image.**
 - Each argument is specified as a null-terminated string, and the list must end with a NULL pointer.
 - **The first argument, arg0, is required and must contain the name of the executable file for the new process image.**

execvp -- Overlay Calling Process and Run New Program

```
int execvp(const char *path, const char *arg0, ..., NULL);
```

- RETURN VALUE-
 - A successful call to execvp **does not have a return value** because **the new process image overlays the calling process image**.
 - However, **-1 is returned if the call to execvp is unsuccessful**.

fork()

- Fork system call is used for creating a new process, which is called **child process**,
- which **runs concurrently** with the process that makes the fork() call (parent process).

fork()

- It takes no parameters and returns an integer value. Below are different values returned by fork().
 - **Negative Value:** creation of a child process was unsuccessful.
 - **Zero:** Returned to the newly created child process.
 - **Positive value:** Returned to parent or caller.
 - **The value contains process ID of newly created child process.**
 - **i.e. the PID of the child process is returned in the parent**

C Program Forking Separate Process

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

int main()
{
    pid_t pid;

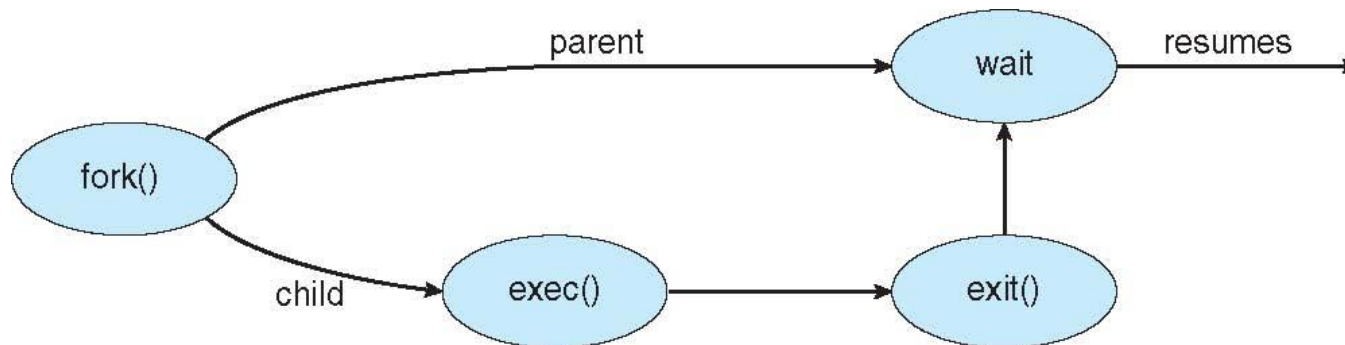
    /* fork a child process */
    pid = fork();

    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;
}
```

fork()

- The parent can create more children or
 - if it has nothing else to do while the child runs,
 - **it can issue a wait system call to move itself off the ready queue**
 - until the termination of the child.



Creating a Separate Process via Windows API

136

15/09/2024

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

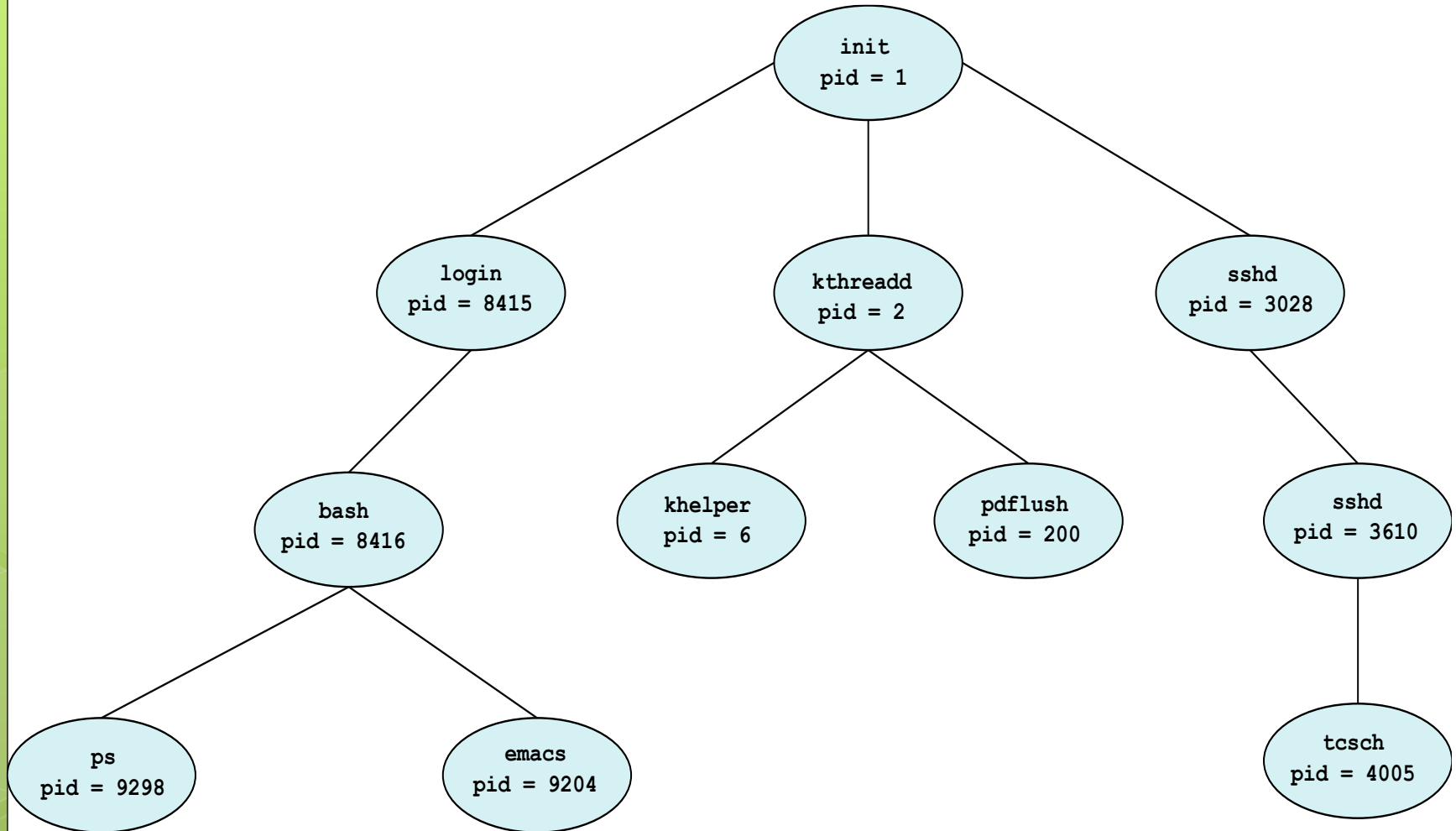
    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
        "C:\\\\WINDOWS\\system32\\mspaint.exe", /* command */
        NULL, /* don't inherit process handle */
        NULL, /* don't inherit thread handle */
        FALSE, /* disable handle inheritance */
        0, /* no creation flags */
        NULL, /* use parent's environment block */
        NULL, /* use parent's existing directory */
        &si,
        &pi))
    {
        fprintf(stderr, "Create Process Failed");
        return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Prof. Shweta Dhawan Chachra

A Tree of Processes in Linux



ISRO | ISRO CS 2018 | Question 63

The following C program

```
main()  
{  
    fork() ;  
    fork() ;  
    printf ("yes");  
}
```

If we execute this core segment, how many times the string yes will be printed ?

- (A)** Only once
- (B)** 2 times
- (C)** 4 times
- (D)** 8 times

ISRO | ISRO CS 2018 | Question 63

The following C program

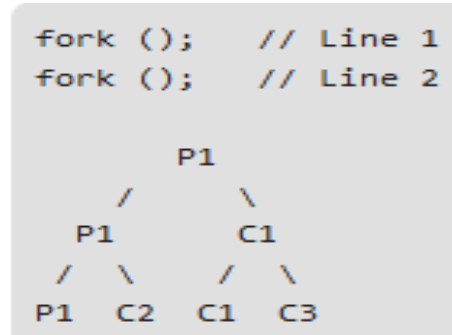
```
main()
```

```
{ fork() ; fork() ; printf ("yes"); }
```

If we execute this code segment, how many times the string yes will be printed ?

- (A) Only once
- (B) 2 times
- (C) 4 times
- (D) 8 times

Answer: (C)



Explanation: Number of times YES printed is equal to number of process created.

Total Number of Processes = 2^n where n is number of fork system calls.

So here $n = 2$, $2^2 = 4$

So, there are total 4 processes (3 new child processes and one original process).

Prof. Shweta Dhawan Chachra

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{

    // make two process which
    run same
    // program after this
    instruction
    fork();

    printf("Hello world!\n");
    return 0;
}
```

Output ?

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{

    // make two process which
    run same
    // program after this
    instruction
    fork();

    printf("Hello world!\n");
    return 0;
}
```

Output:

Hello world! Hello world!

After a new child process is created, both processes will execute the next instruction following the fork() system call.

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

Output:

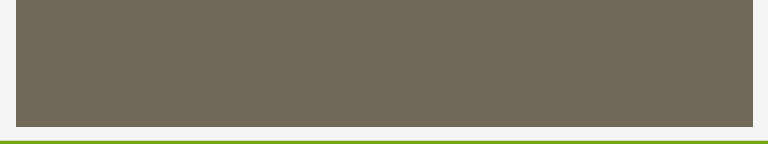
?

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

Output:

hello
hello
hello
hello
hello
hello
hello
hello

The number of times 'hello' is printed is equal to number of process created. Total Number of Processes = 2^n , where n is number of fork system calls. So here $n = 3$, $2^3 = 8$



```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
    // child process because return value zero
    if (fork() == 0)
        printf("Hello from Child!\n");

    // parent process because return value non-zero.
    else
        printf("Hello from Parent!\n");
}
int main()
{
    forkexample();
    return 0;
}
```



```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
    // child process because return
    // value zero
    if (fork() == 0)
        printf("Hello from Child!\n");

    // parent process because
    // return value non-zero.
    else
        printf("Hello from Parent!\n");
}
int main()
{
    forkexample();
    return 0;
}
```

Output-

1. Hello from Child!

Hello from Parent!

(or)

2. Hello from Parent!

Hello from Child!

Here, two outputs are possible because **the parent process and child process are running concurrently.**

So we don't know **whether the OS will first give control to the parent process or the child process.**

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

void forkexample()
{
    int x = 1;

    if (fork() == 0)
        printf("Child has x = %d\n",
++x);
    else
        printf("Parent has x = %d\n", --
x);
}
int main()
{
    forkexample();
    return 0;
}
```

Output?

Point To be Noted....

- Parent process and child process are running the same program,
 - **but it does not mean they are identical.**
- OS allocate
 - **different data and states for these two processes, and**
 - **the control flow of these processes can be different.**

```

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

void forkexample()
{
    int x = 1;

    if (fork() == 0)
        printf("Child has x = %d\n",
++x);
    else
        printf("Parent has x = %d\n", --
x);
}
int main()
{
    forkexample();
    return 0;
}

```

Output:

Parent has x = 0

Child has x = 2

(or)

Child has x = 2

Parent has x = 0

- Here, global variable change in one process does not affected two other processes **because data/state of two processes are different.**
- And also parent and child run simultaneously so two **outputs are possible.**

- Process executes last statement and then asks the operating system to delete it using the **exit()** system call.
- Return of data(output) from child to parent (via **wait()**)
- Process resources (physical and virtual memory, open files, I/O buffers) are deallocated by operating system

Child Process Termination

- Parent may terminate the execution of children processes using the **abort()** system call.
- Some reasons for doing so:
 - 1) **Child has exceeded allocated resources**, thus the parent needs a mechanism to inspect the state of its children.
 - 2) **Task assigned to child is no longer required**
 - 3) **The parent is exiting** and the operating systems does not allow a child to continue if its parent terminates

Child Process Termination

Cascading termination

- Some operating systems **do not allow child to exist if its parent has terminated.**
- If a process terminates, then all its children must also be terminated.**
- 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 wait system call returns status information and the process identifier (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**



What is the Zombie process?

- Also known as "**dead**" process.
- Ideally when a process completes its execution, **it's entry from the process table should be removed but this does not happen in case of a zombie process.**



Prof. Shweta Dhawan Chachra

What happens with the zombie processes?

- `wait()` system call is used for removal of zombie processes.
- `wait()` call ensures that the parent doesn't execute till the child process is completed.



Prof. Shweta Dhawan Chachra

Reaping of Child?

- When the child process completes executing ,
- **The parent process removes entries of the child process from the process table.**
- This is called "reaping of child".

Zombie Gone

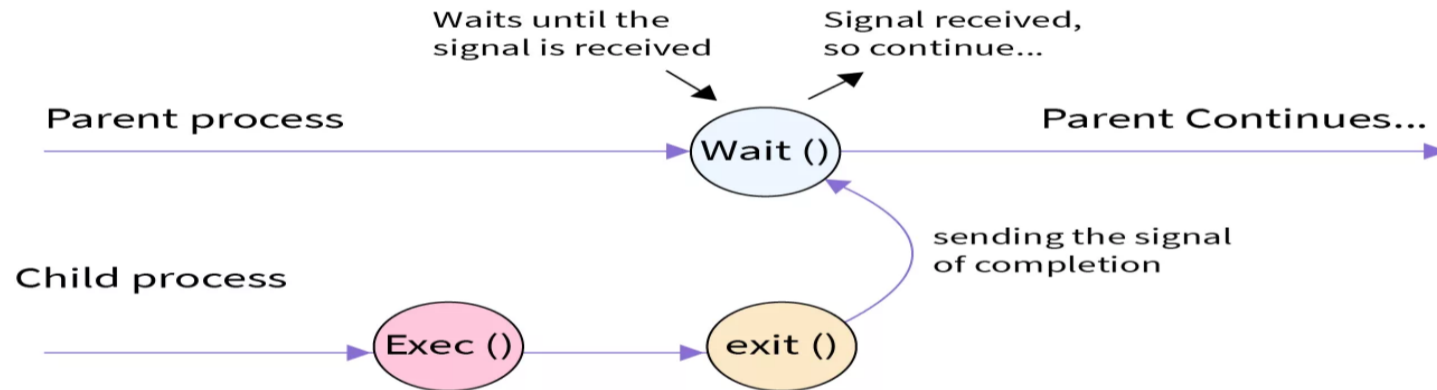


Prof. Shweta Dhawan Chachra

Zombie process?

158

15/09/2024



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 done with its 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

Prof. Shweta Dhawan Chachra