Unit II: Process Management

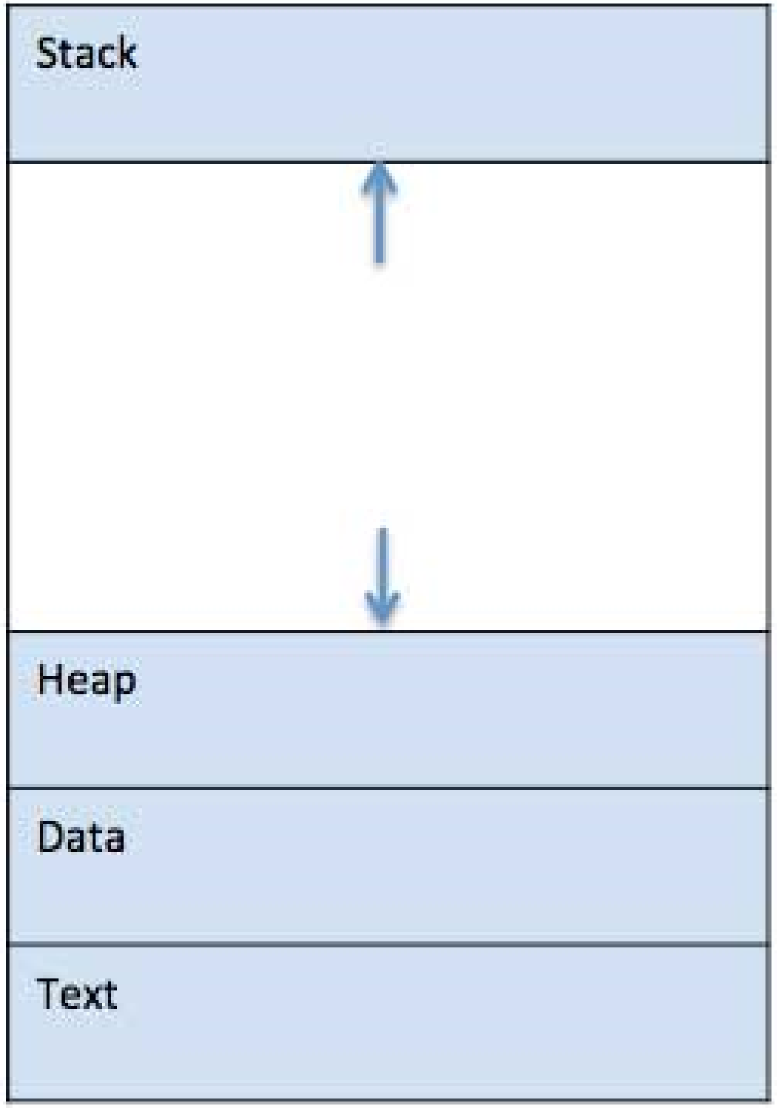
Process Concept

- Process
- Program
- Process State
- Process Control Block

Process

- A program in execution
- An instance of a program running on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by
  - A single sequential thread of execution
  - A current state
  - An associated set of system resources: memory image, open files, locks, etc.

- When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text and data.



S.N.	Component & Description
1	**Stack** The process Stack contains the temporary data such as method/function parameters, return address and local variables.
2	**Heap** This is dynamically allocated memory to a process during its run time.
3	**Text** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
4	**Data** This section contains the global and static variables.

Program

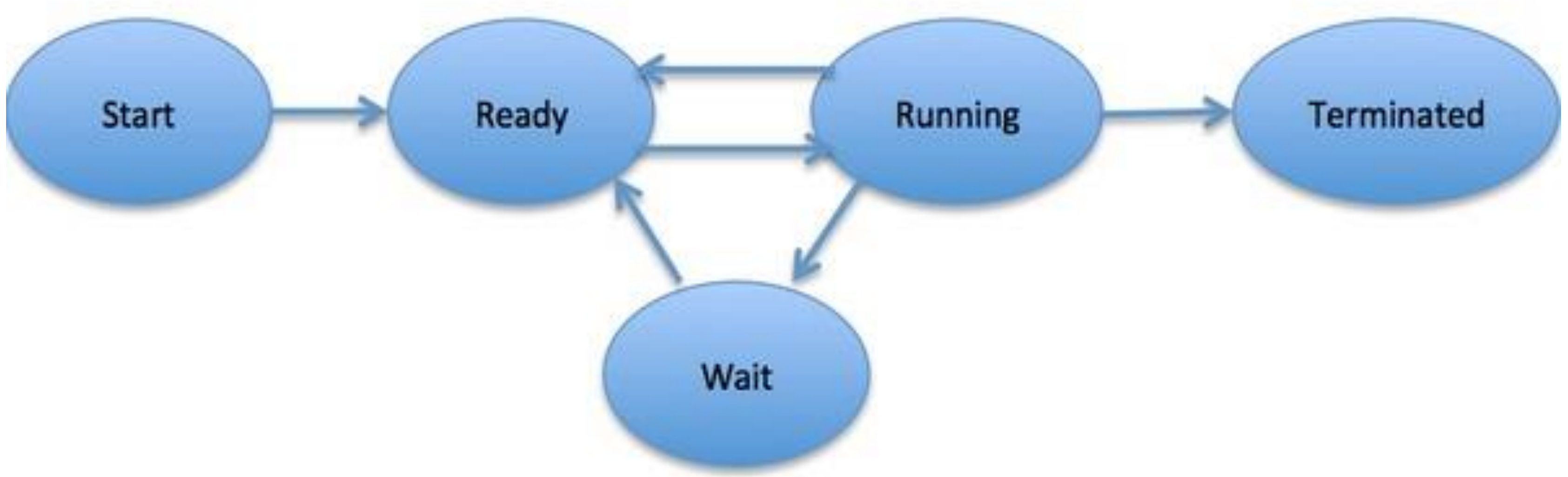
- A program is a piece of code which may be a single line or millions of lines.
- A computer program is usually written by a computer programmer in a programming language.

- A computer program is a collection of instructions that performs a specific task when executed by a computer.
- When we compare a program with a process, we can conclude that a process is a dynamic instance of a computer program.
- A part of a computer program that performs a well-defined task is known as an **algorithm**.
- A collection of computer programs, libraries and related data are referred to as a **software**.

Program vs Process

- When we write a program in C or C++ and compile it, the compiler creates a binary code. The original code and Binary code, both are programs. When we actually run the binary code, it becomes a process.
- A process is an ‘active’ entity as opposed to a program which is considered to be a ‘passive’ entity.
- A single program can create many processes when run multiple times, for example when we open a .exe or binary file multiple times, many instances begin (many processes are created).

Five State Process Model

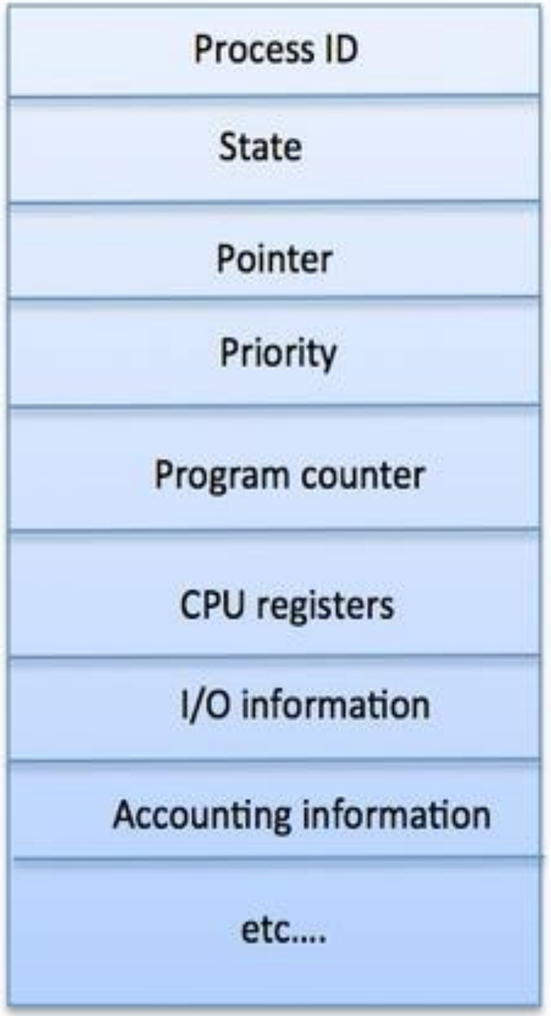


S.N.	State & Description
1	**Start** This is the initial state when a process is first started/created.
2	**Ready** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process.
3	**Running** Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
4	**Waiting** Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
5	**Terminated or Exit** Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.

Process Control Block (PCB)

- A Process Control Block is a data structure maintained by the Operating System for every process.
- The PCB is identified by an integer process ID (PID).
- A PCB keeps all the information needed to keep track of a process with the help of process attributes (also known as ***Context of the process)***.
- The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.
- PCB serves as a repository for any information that may vary from process to process

PCB



Process ID

State

Pointer

Priority

Program counter

CPU registers

I/O information

Accounting information

etc....

S.N.	Information & Description
1	**Process State** The current state of the process i.e., whether it is ready, running, waiting, or whatever.
2	**Process privileges** This is required to allow/disallow access to system resources.
3	**Process ID** Unique identification for each of the process in the operating system.
4	**Pointer** A pointer to parent process.
5	**Program Counter** Program Counter is a pointer to the address of the next instruction to be executed for this process.
6	**CPU registers** Various CPU registers where process need to be stored for execution for running state.
7	**CPU Scheduling Information** Process priority and other scheduling information which is required to schedule the process.
8	**Memory management information** This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
9	**Accounting information** This includes the amount of CPU used for process execution, time limits, execution ID etc.
10	**IO status information** This includes a list of I/O devices allocated to the process.

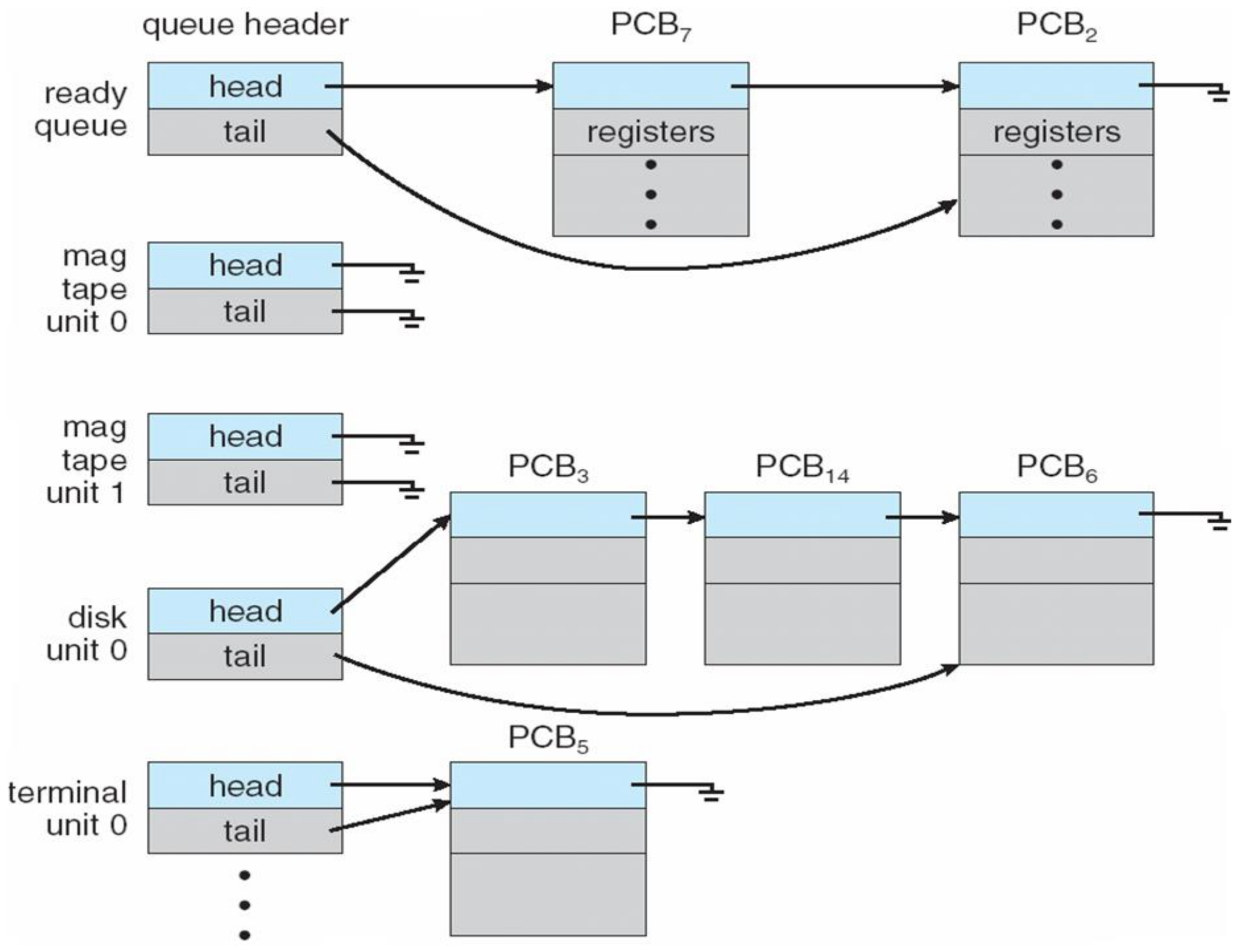
Process Scheduling

- Objective of multiprogramming is to have some processes running all times, to maximize CPU utilization
- Objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while running
- Uniprocessor system can have only 1 running process. If more exists they have to wait until CPU is free and can be rescheduled
- **Process scheduler** selects among available processes for next execution on CPU

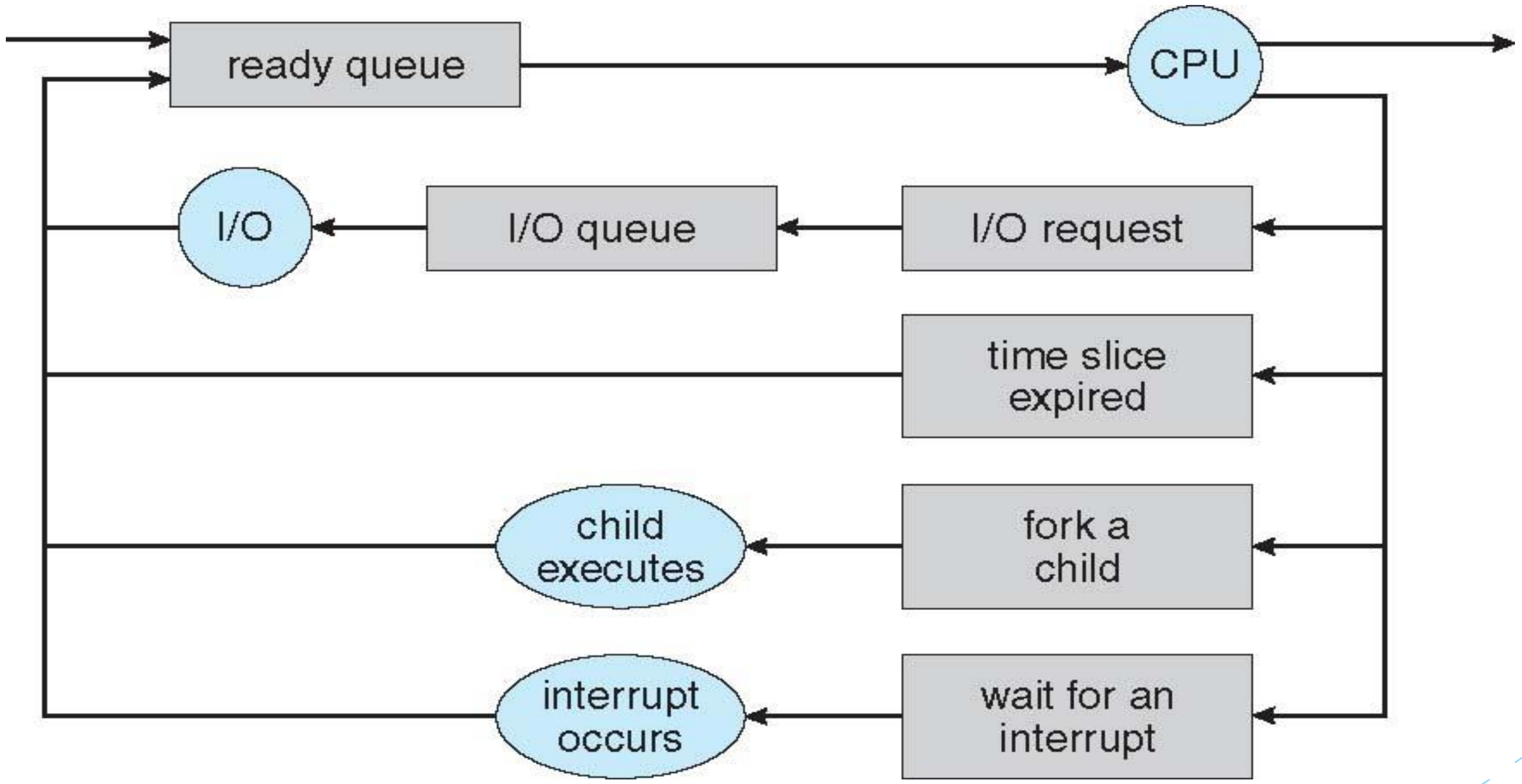
Scheduling Queues

- **Job queue**
  - set of all processes in the system
- **Ready queue**
  - set of all processes residing in main memory, ready and waiting to execute
  - Generally stored as a linked list
  - Ready queue header points to 1st and final PCBs in the list. Each PCB includes a pointer to next PCB in 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

Ready Queue And Various I/O Device Queues



Queuing- diagram Representation of Process Scheduling



Schedulers

- Processes migrate between various scheduling queues throughout its lifetime.
- The OS must select processes for scheduling from these queues in some fashion.
- This selection is carried out by schedulers.
- Processes are spooled to a mass storage device and kept there for later execution.

- **Long-term scheduler** (or job scheduler) – selects processes from this pool and loads them into main memory for execution.

- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Time sharing systems such as UNIX and Microsoft Windows have no long term schedulers, simply put every new process in memory for short term scheduler.

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
  - Executes at least once every 100 milliseconds

- Long-term scheduler is invoked very infrequently (seconds, minutes between creation of processes) ⇒ (may be slow)

- The long-term scheduler controls the *degree of multiprogramming-number of processes in memory*
  - *LTS may need to be invoked only when process leave the system*

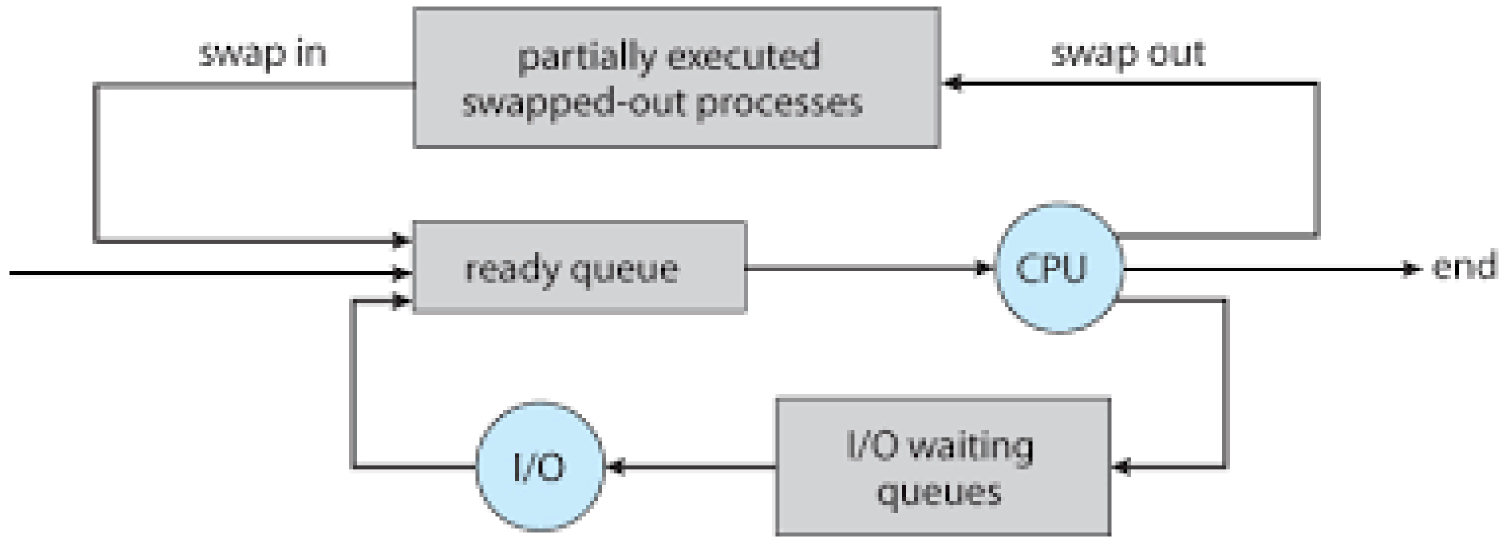
- Long term scheduler must make a careful selection between type of processes:
  - **I/O-bound process** – spends more time doing I/O than computations
  - **CPU-bound process** – spends more time doing computations

- Long term scheduler should select a good **process mix** of I/O bound and CPU bound process
- If all processes are I/O bound ready queue will be empty ⇒short term scheduler will have little to do
- If all processes are CPU bound, I/O waiting queue will almost always be empty, devices will go unused, system will be unbalanced
- System with best performance will have a combination of CPU bound and I/O bound processes

Medium Term Scheduling

- The medium term scheduler removes partially executed processes from memory (reduces the degree of multiprogramming)
- Some time later these processes can be reintroduced into memory and execution can be continued from where it left off. This scheme is called swapping. Process is swapped-in/ out by medium term scheduler
- Swapping may help to improve process mix or if memory has to be freed up

Addition of Medium Term Scheduling



Context Switch

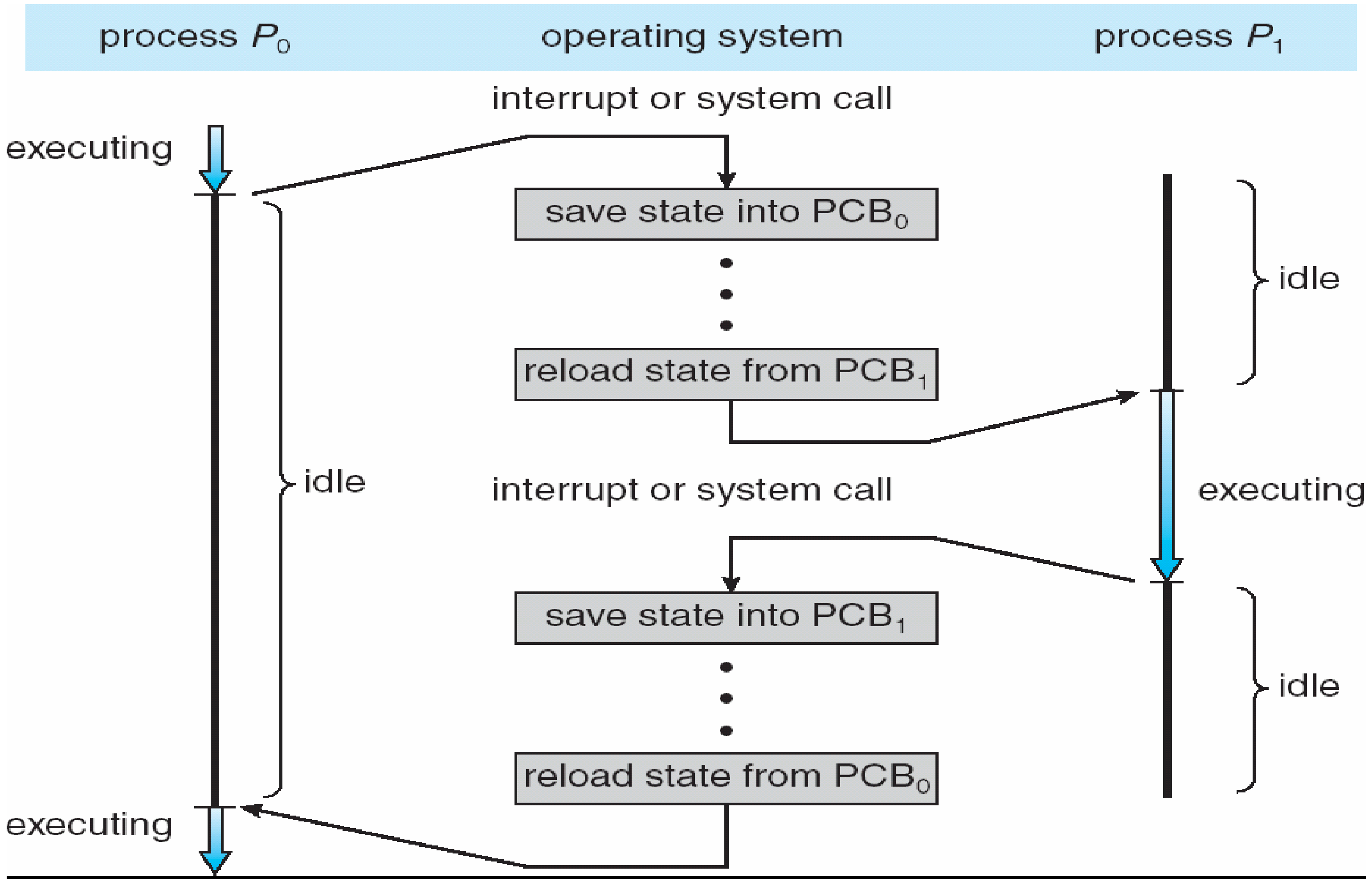
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a **context switch**.

- **Context** of a process represented in the PCB
  - CPU registers, state, memory management information

- When context switch occurs, kernel saves context of old process in its PCB and loads the saved context of the new process scheduled to run

- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB -> longer the context switch
- Context-switch times dependent on hardware support.
  - Some processors provide multiple set of registers. A context switch simply includes changing the pointer to current register set
  - If active processes exceeds register sets, system resorts to copying data to and from memory as before
- More complex the OS -> more work must be done during a context switch

CPU Switch From Process to Process



Preemptive/ non preemptive scheduling

- Non Preemptive/ cooperative:
  - Once a process is allocated CPU, it does not leave unless:
    - It terminates
    - Switches to the wait state
- Preemptive
  - OS can force (preempt) a process from CPU at anytime
  - Say, to allocate CPU to another higher-priority process

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the scheduler (short term/CPU).
- This function involves the following:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- Dispatcher should be as fast as possible, since it is invoked during every process switch.
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- Different CPU scheduling algo. have different properties. Few criteria for comparing CPU scheduling include the following:

  - **CPU utilization** – keep the CPU as busy as possible
  - **Throughput** – # of processes that complete their execution per time unit
  - **Turnaround time** – amount of time to execute a particular process
  - **Waiting time** – amount of time a process has been waiting in the ready queue
  - **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Scheduling Algorithms

1. First-Come, First-Served (FCFS) Scheduling (NP)
2. Shortest-Job-First (SJF) Scheduling/ shortest next CPU burst (P/NP)
3. Priority Scheduling (P/NP)
4. Round Robin (RR) scheduling (P)

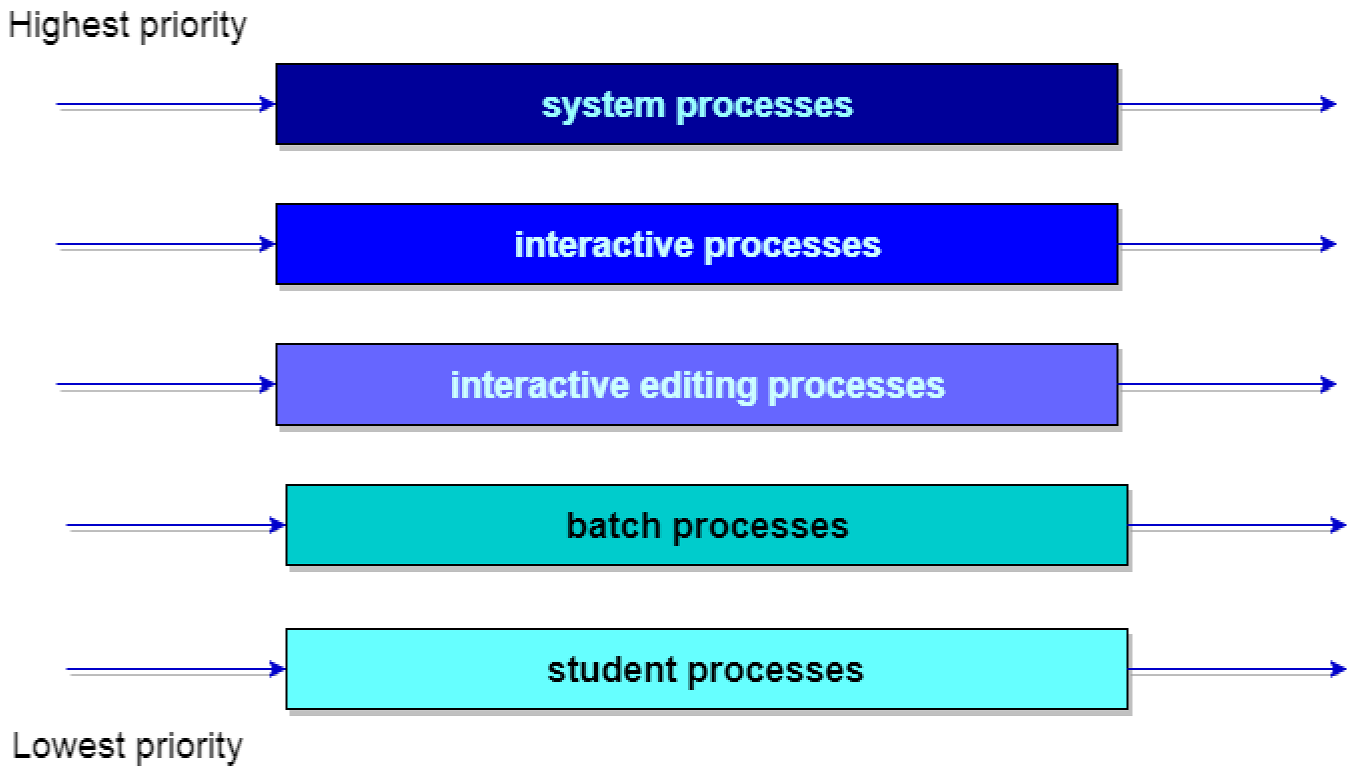
P1	**P2**	**P3**	**P1**	**P1**
0 4	7	10	14	…….30

5. Multilevel Queue Scheduling
6. Multilevel Feedback Queue Scheduling

Scheduling Algorithms (1-4) Problems

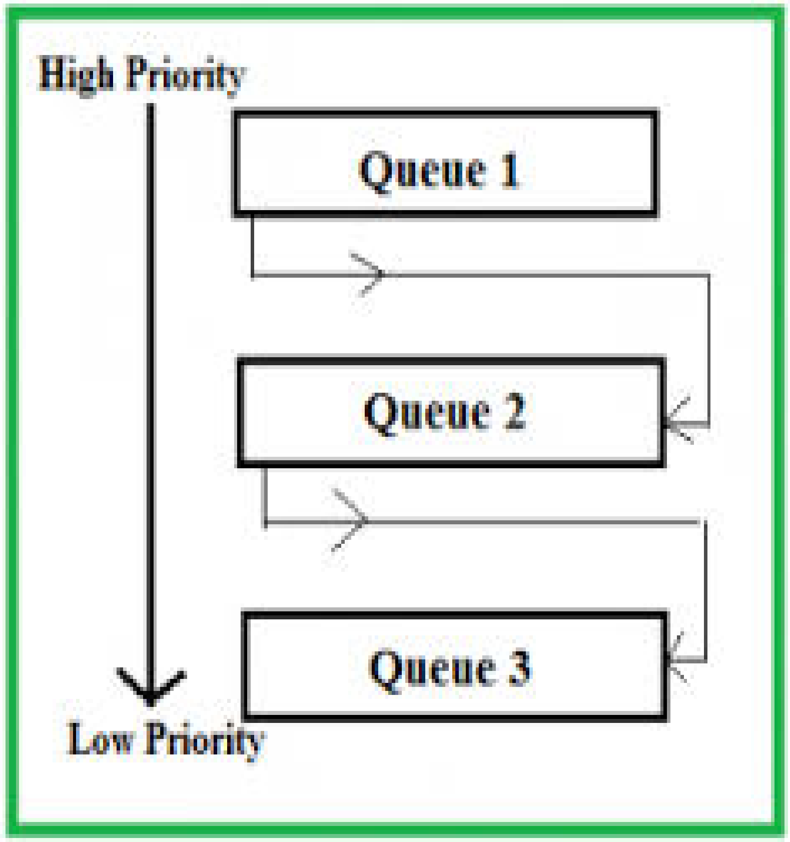
Multilevel Queue Scheduling (MQS)

- Another class of scheduling algorithms has been created for situations in which processes are easily classified (Eg: foreground (interactive), background (batch))
- Different types of processes have different response-time requirements, different scheduling needs, priorities
- MQS partitions ready queue into several separate queues.
- **Processes are permanently assigned to 1 queue** based on some property of process (memory size, priority, type)
- Each queue has its own scheduling algorithm
- There must be scheduling among the queues, commonly implemented as fixed priority preemptive scheduling.
  - Foreground Q may have **absolute priority** over background Q
    - No process in background Q could run unless Q for foreground process is empty
    - If foreground process entered the ready Q while a background process was running, the background process will be preempted
- Another possibility is to time slice among queues.
  - Eg: 1 queue is given 80% of CPU time for RR scheduling while other receives 20% of CPU to give to its processes on FCFS basis

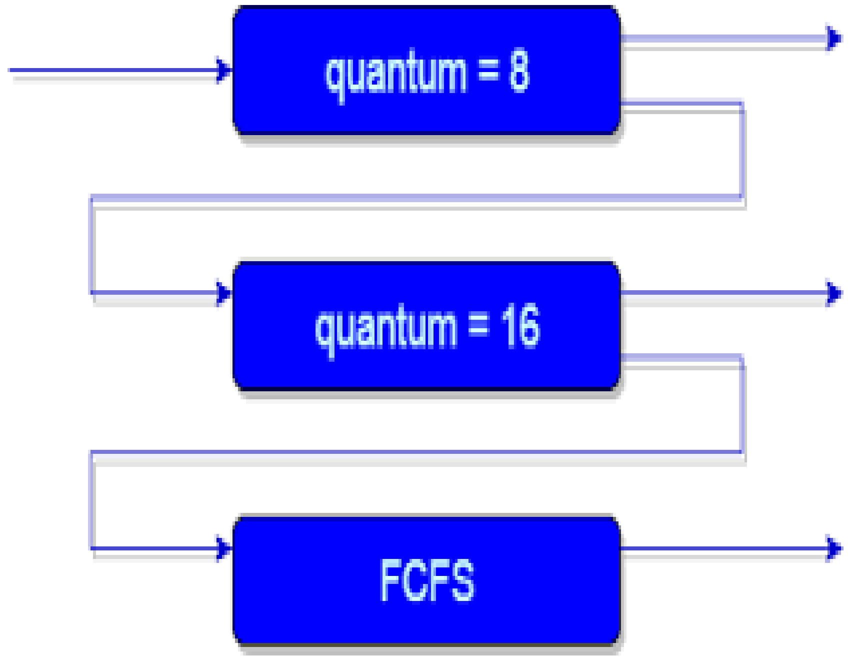

Highest priority
system processes
interactive processes
interactive editing processes
batch processes
student processes
Lowest priority

Multilevel Feedback Queue Scheduling (MFQS)

- MQS has the advantage of low scheduling overhead, but it is inflexible (processes do not move from 1 queue to other, since they don’t change their nature)
- MFQS allows processes to move between queues
- Separate processes according to their CPU burst.
- If a processes uses too much CPU time, it will be moved to lower priority queue
- I/O bound and interactive processes are in higher priority queues.
- Processes that waits for long in low priority queue may be moved to higher priority queue. This from of aging prevents starvation



- Scheduler will execute all processes in Q1 then Q2 and then Q3
- Process that arrives for Q2 will preempt a process in Q3. A process in Q2 will be pre-empted by a process arriving for Q1


quantum = 8
quantum = 16
FCFS

MFQS is defined by following parameters:

- The number of queues.
- The scheduling algorithm for each queue.
- The methods used to upgrade or demote processes from one queue to another. (Which may be different.)
- The method used to determine which queue a process enters initially.

Operations on Processes

- The processes in a system can execute concurrently, and must be created and deleted dynamically
- OS must provide a mechanism for process creation and termination
- ***Process creation:***
  - Process may create new process via create-process system call
  - **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
  - When OS creates a process at the explicit request of another process, the action is referred to as **process spawning.**
  - Generally, processes are identified and managed via **a process identifier** (**pid**)
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources

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

- Address space
  - Child duplicate of parent (has same program and data as the parent)
  - Child has a program loaded into it

Reason for Process Creation

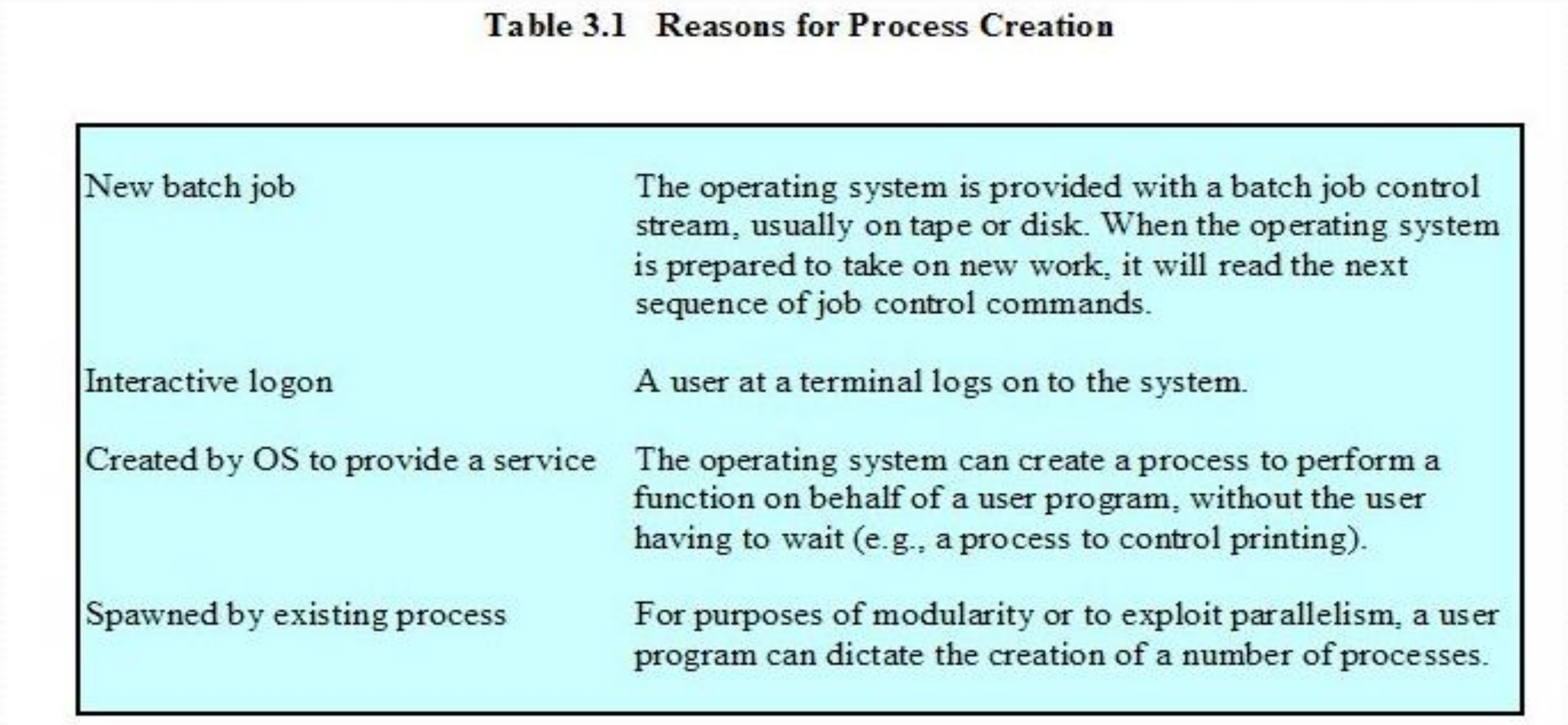


Table 3.1 Reasons for Process Creation

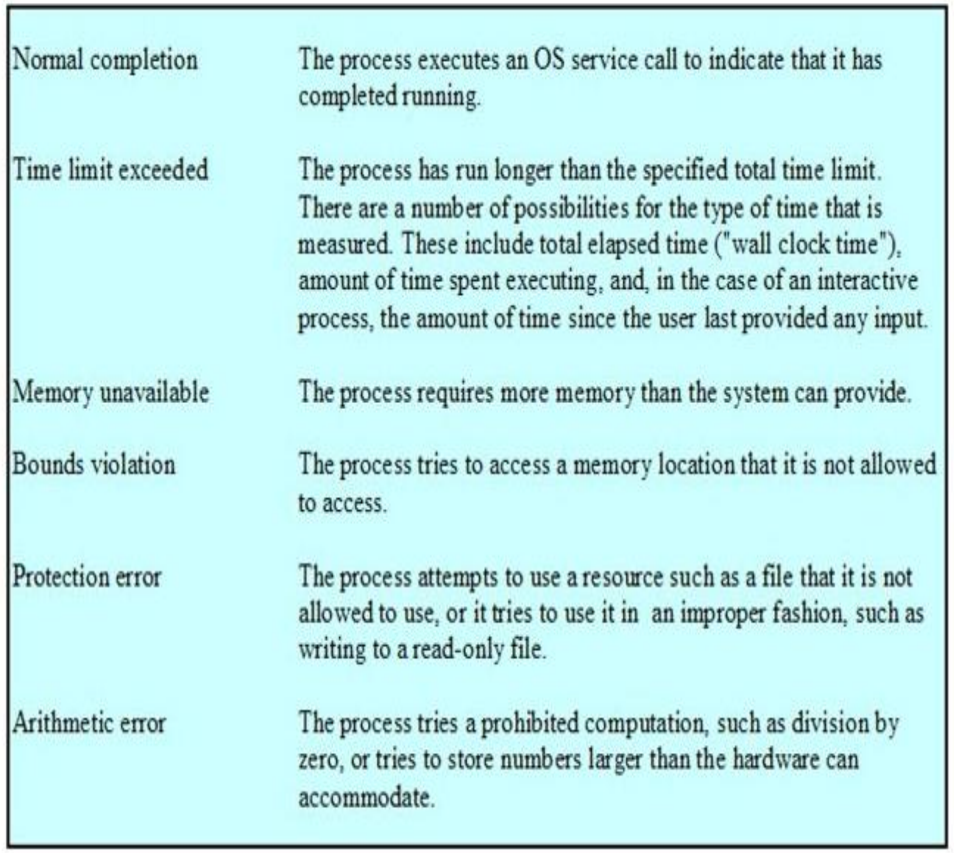
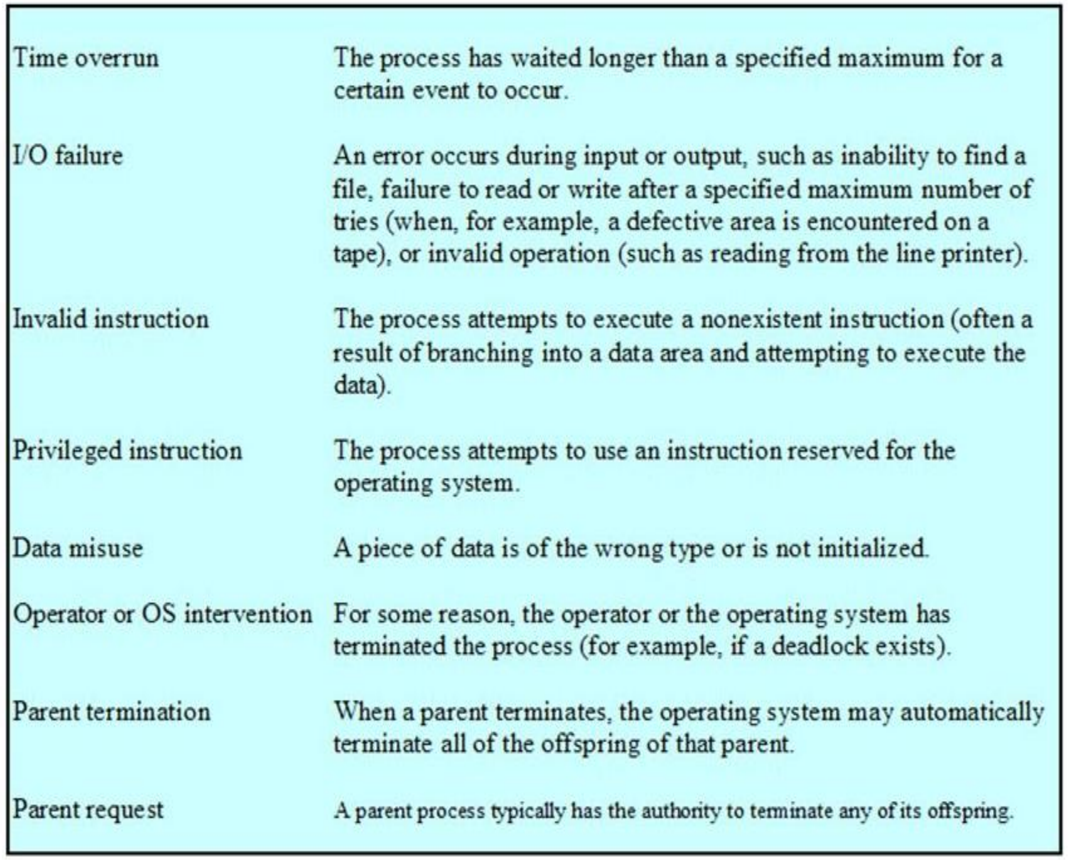
New batch job	The operating system is provided with a batch job control stream, usually on tape or disk. When the operating system is prepared to take on new work, it will read the next sequence of job control commands.
Interactive logon	A user at a terminal logs on to the system.
Created by OS to provide a service	The operating system can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.

Process Termination

- Process executes last statement and asks the operating system to delete it (**exit system call**)
    - Process may return data (output) to its parent (via **wait system call**)
    - Process' resources are deallocated by operating system

- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - **cascading termination**

Table 3.2 Reasons for Process Termination

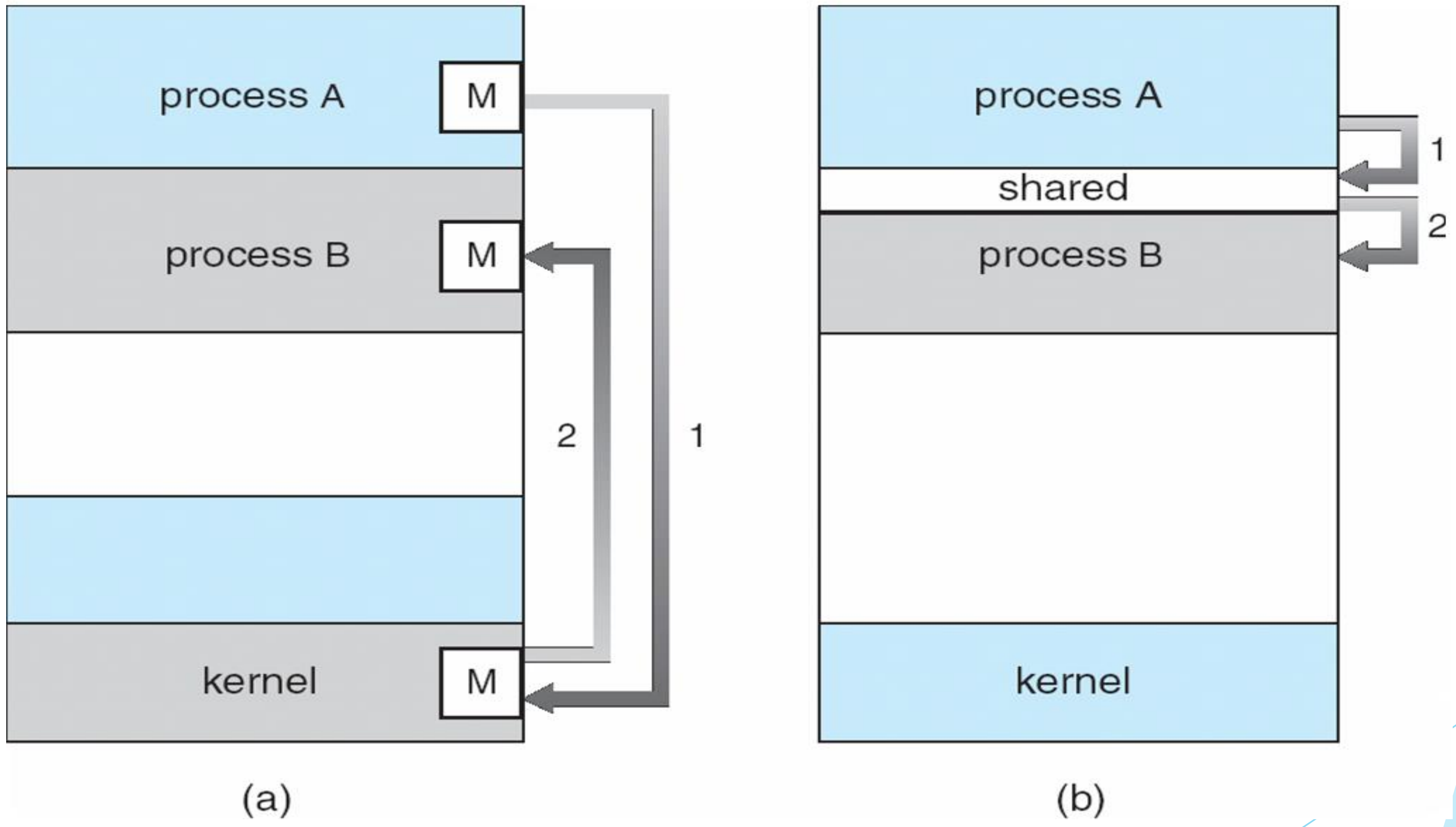



Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (for example, if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

Inter-process Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes executing in the system.
  - Process that shares data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
  - Shared memory
  - Message passing

Communications Models



Many OS implement both models

Message passing

- Useful for exchanging smaller amount of data, because no conflict need be avoided
- Easier to implement than shared memory
- Implemented using system calls, hence require more time consuming task of kernel intervention

Shared Memory

- Allows maximum speed and convenience of communication
- Faster than message passing
- System calls are required only to establish shared memory regions
- Once shared memory is established, no assistance from kernel is required (all accesses are treated as routine memory accesses)

Shared Memory Systems

- Shared memory Model
  - Use map memory system call to gain access to regions of memory owned by other processes
  - Information is exchanged by reading from and writing to the shared areas
- Example of cooperating processes: Producer-Consumer Problem
  - Compiler produces assembly code which is consumed by assembler
  - Assembler produces object modules, which are consumed by the loader
  - Web server produces(provides) HTML files, images, which are consumed (read) by the client web browser
- One solution to producer-consumer problem uses shared memory
- To allow producer-consumer to run concurrently, we must have available buffer of items that are filled by the producer and emptied by the consumer
- This buffer resides in the region of memory that is shared by producer and consumer processes.

Message Passing Systems

- Connection is opened
- Name of the other communicator must be known (can be known via get hostid(), get processid() system call)
- These identifiers are passed to open and close connection system calls
- Recipient gives permission via accept connection call
- Processes receiving connections are special purpose daemons
- Client (source) and server (daemon) exchange messages by read and write system calls
- Close connection terminates the communication

Message Passing Systems

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If processes *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - Not physical implementation(e.g., shared memory, hardware bus, network)
  - logical (e.g., logical properties)
    - Direct/ indirect communication
    - Synchronous/ asynchronous communication
    - Automatic/ explicit buffering

Naming

- Direct Communication
  - Processes must name each other explicitly:
    - **send** (*P*, *message*) – send a message to process P
    - **receive**(*Q*, *message*) – receive a message from process Q
  - Properties of communication link
    - Links are established automatically
    - A link is associated with exactly one pair of communicating processes
    - Between each pair there exists exactly one link
    - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Process can communicate with other process via a number of mailboxes

- Primitives are defined as:

  send(*A*, *message*) – send a message to mailbox A

  receive(*A*, *message*) – receive a message from mailbox A

- Properties of communication link
  - Link is established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

- Mailbox can be owned by process or OS

- If owned by process:
  - Mailbox is part of the address space of the process

- If owned by OS
  - Mailbox has existence of its own
  - It is independent and not attached to any particular process
  - OS must provide mechanism that allows a process to do the following operations
    - create a new mailbox
    - send and receive messages through mailbox
    - destroy a mailbox

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver retrieve a valid message or null
- Different combinations of send and receive are possible. When both send and receive are blocking, we have rendezvous between sender and receiver

Buffering

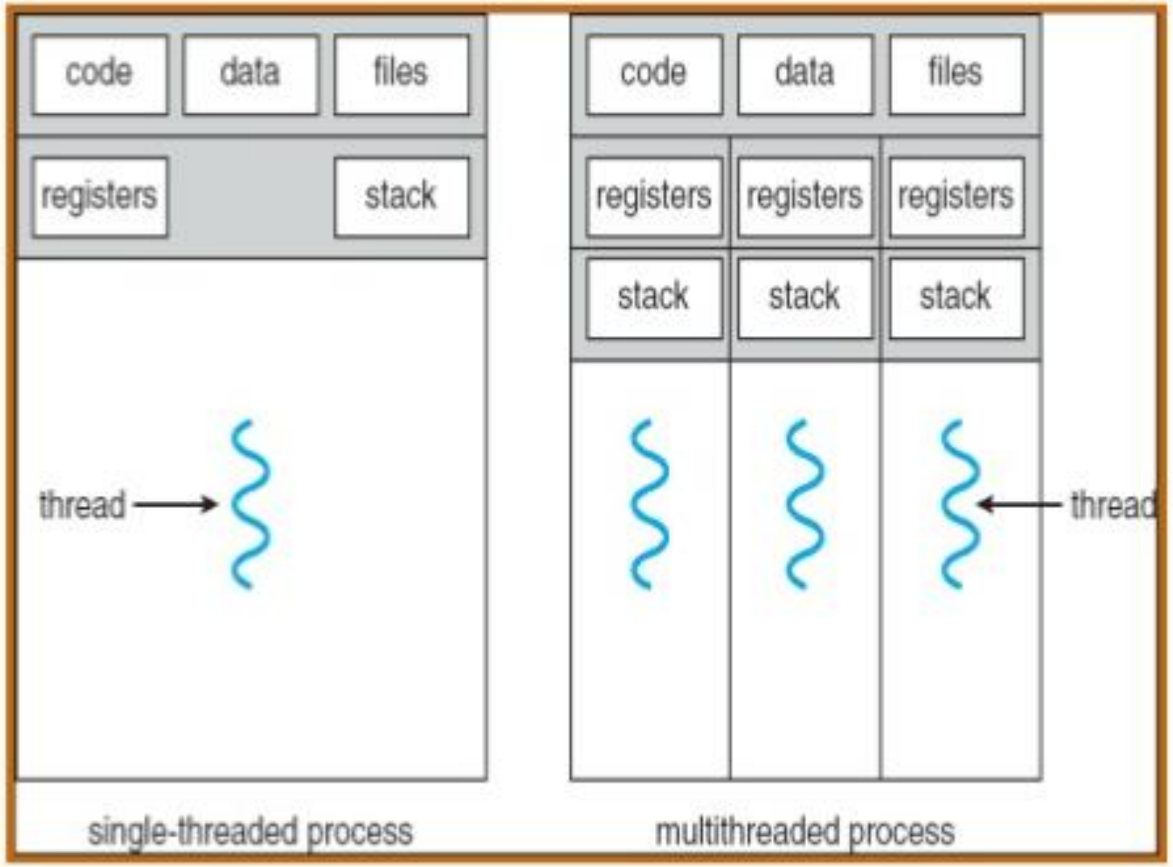
- Messages exchanged by communicating processes reside in a temporary queue
- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages (cannot have messages waiting in it)
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

Threads

- Process is a program that performs a single thread of execution
  - When a process is running a word-processor program, a single thread of instruction is being executed.
- Single thread of control allows the process to perform only 1 task a 1 time
- Modern OS allow a process to have multiple threads of execution
- Thus allowing the process to perform more than 1 task at a time
  - PCB is expanded to include information for each thread.

threads

- Basic unit of CPU utilization
- Sometimes called a lightweight process (LWP)
- Consists of
  - Thread id
  - Program counter
  - Register stack
  - Stack
- It shares with other threads belonging to the same process its code section, data section and other resources
- Heavyweight (traditional) process has single thread of control


code
data
files
registers
stack
thread
single-threaded process
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded process

Threads in use

- Web browser
  - 1 thread displays images/ text, another thread retrieves data from n/w
- Word processor
  - Thread for displaying graphics, responding to keystrokes, performing spelling and grammar check in background
- Web server
  - Several clients access it
  - If server ran a traditional single threaded process, it would serve only 1 client at a time (increases wait time of client)
  - Solution used before threads: every time server receives a request, it creates a separate process to service that request
    - Time consuming and resource intensive
    - New process performs the same task as existing process
  - Hence create 1 process with multiple threads
    - When a request is made, thread will service the request and resume listening for additional requests.

Benefits

- Responsiveness
  - Allows a program to continue even if a part is blocked
  - Multithreaded browser allows user interaction in 1 thread while another loads an image
- Resource Sharing
  - threads share memory and resources of process to which they belong
  - Application can have several threads within the same address space
- Economy
  - In general it is much more time consuming to create and manage processes than threads
  - More economical to create and context switch threads
- Utilization of Multiprocessor & Multicore Architectures
  - Single threaded process can run on one CPU (CPU moves between each thread to create an illusion of parallelism)
  - In multiprocessor architecture, each thread may be running in parallel on different processor
  - Multithreading on multiprocessor machine increases concurrency

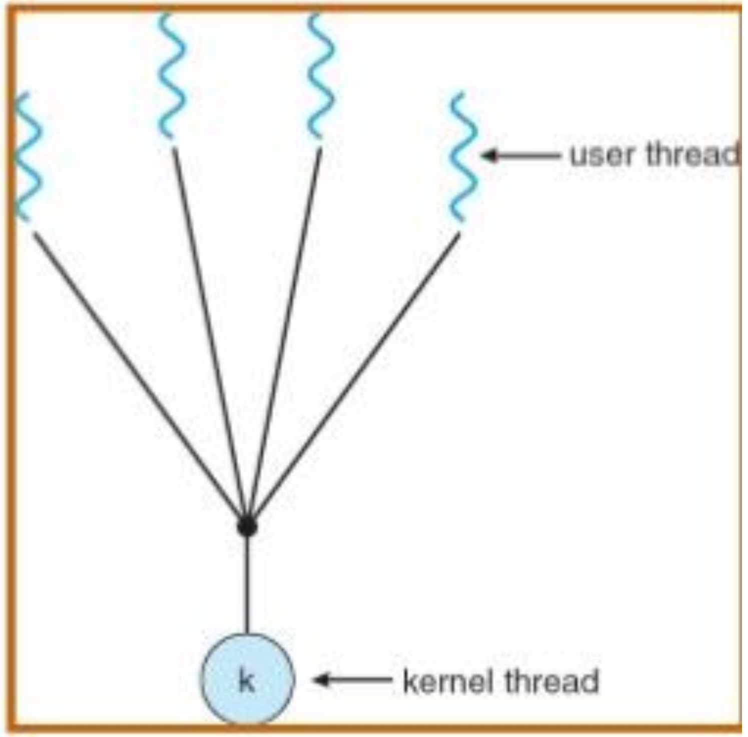
Types

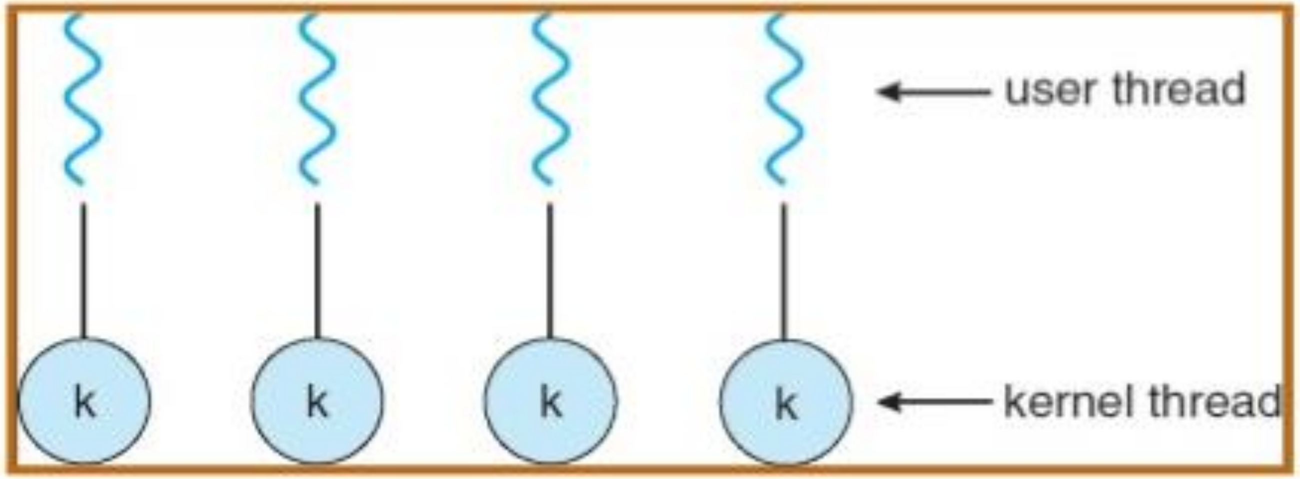
- User-level thread
  - Supported above kernel and implemented by thread library at user level
  - No support from kernel as kernel is unaware
  - Creation, scheduling, management is done in user space
  - User level threads are fast to create and manage

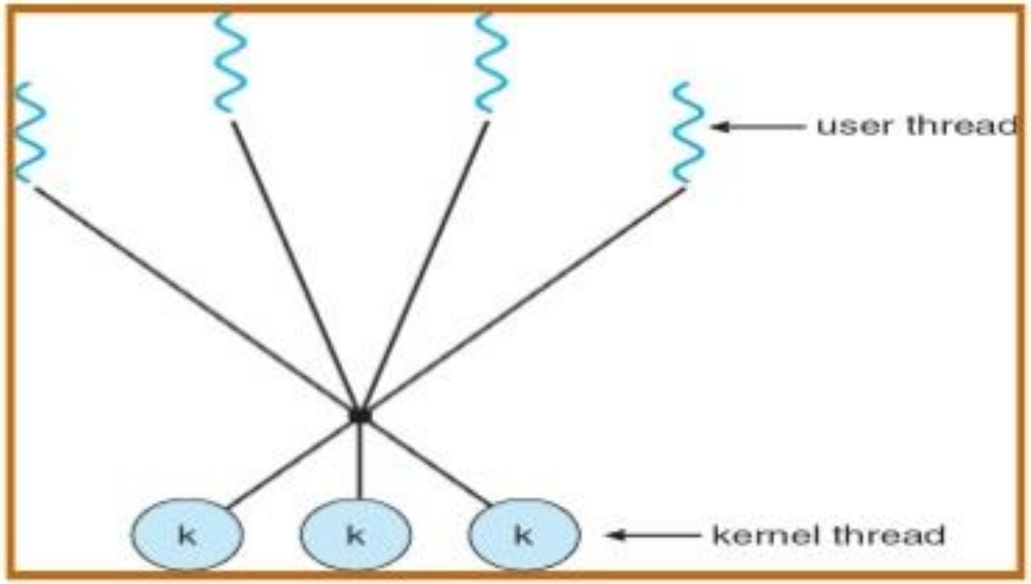
- Kernel-Level Thread
  - Supported directly by kernel (OS)
  - Kernel performs Creation, scheduling, management in kernel space
  - Slower to create and manage

Multithreading Models

- How user-level threads are mapped to kernel ones.
- Relationship must exists between user and kernel threads
  - Many-to-One
    - Many user-level threads mapped to single kernel thread
    - Only 1 thread can access kernel at a time, multiple threads are unable to run in parallel on multiprocessors
  - One-to-One
    - Each user-level thread maps to kernel thread
    - Provides more concurrency
    - Allows multiple threads to run in parallel on multiprocessors
    - Creating a user thread requires creation of a kernel thread (limit on user threads)
  - Many-to-Many
    - Allows many user level threads to be mapped to many kernel threads
    - Developers can create n number of user threads and the corresponding kernel thread can run in parallel on multiprocessor


user thread
k
kernel thread


user thread
k
k
k
k
kernel thread


user thread
k
k
k
kernel thread

Thread libraries

- Provides the programmer with API for creating and managing threads.
- 2 ways of implementing thread library
  1. Provide a library in user space with no kernel support.
     1. All code and data structures for library exists in user space.
     2. Invoking a function in the library results in a local function call in user space and not a system call.

  2. Implement a kernel level library supported directly by OS.
     1. All code and data structures for library exists in kernel space.
     2. Invoking a function in the API for the library typically results in a system call.
- 3 main thread libraries
  - POSIX Pthreads
    - Can be provided as user or kernel level library
  - Win32
    - Kernel level library on windows
  - Java
    - Allows threads to be created and managed in JAVA programs
    - Since JVM runs on top of host OS, java thread API is implemented using thread library available on host system

Threading Issues

- fork() and exec() system calls
  - If one thread forks, is the entire process copied, or is the new process single-threaded?
  - 2 versions of fork()
    - Duplicate all threads
    - Duplicate the thread that invoked fork()
  - If thread invokes exec() system call, the program specified in the parameter to exec() will replace the entire process-including all threads

- Thread cancellation
  - Terminating a thread before it has finished (target thread)
  - Two general approaches:
    - Asynchronous cancellation terminates the target thread immediately
    - Deferred cancellation allows the target thread to periodically check if it should be cancelled

- Signal handling
  - Signals are used in UNIX systems to notify a process that a particular event has occurred
  - 2 types of signal:
    - Synchronous: illegal memory access, / by 0
      - If a running program performs these actions, signal is generated
      - Delivered to same process
    - Asynchronous
      - When signal is generated by an event external to a running process, that process receives the signal asynchronously
        - Eg: terminating a process with ctrl+c
      - Asynchronous signal is sent to another process
  - A signal handler is used to process signals
    - Signal is generated by particular event
    - Signal is delivered to a process
    - Signal is handled

- 2 types of signal handlers
  - Default –run by kernel
  - User defined –user defined function is called rather than default action
- Signals are delivered to the process in single threaded programs
- Where should be a signal delivered to in multithreaded programs?
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

- Thread pools
  - Create a number of threads at process start up and place in a pool where they await work
  - Advantages: (over thread on demand approach)
    - Usually slightly faster to service a request with an existing thread than create a new thread
    - Allows the number of threads in the application(s) to be bound to the size of the pool

- Thread specific data
  - Allows each thread to have its own copy of data
  - Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

- Scheduler activations
  - Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
  - Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library
  - This communication allows an application to maintain the correct number kernel threads

Thread Scheduling

- On OS that support threads, it is kernel level threads-not processes-that are in fact being scheduled by OS (terms process scheduling and thread scheduling are used interchangeably)
- Kernel is unaware about user level thread.
- To run on CPU, user level threads must be mapped to an associated kernel level thread
- Contention Scope
  - On systems implementing many-many and many-1 models, the thread library schedules user level thread to run on available LWP (kernel level thread) [Process Contention Scope-PCS]
  - OS should schedule kernel thread onto a physical CPU
  - To decide which kernel thread to schedule on CPU, kernel uses System Contention Scope-SCS

Multiprocessor Scheduling

- Concerns in multi processor scheduling
  - Approaches to Multiple Processor Scheduling
  - Processor Affinity
  - Load Balancing
  - Multicore Processors
  - Virtualization and Scheduling