Chapter 2

Processes

Operating Systems (ECEg-4181)

Mequanent Argaw Muluneh

Wednesday, April 29, 2020

Outline

- ❖ Process Concept
- ❖ Process Scheduling
- ❖ Operations on Processes
- ❖ Interprocess Communication (IPC)
- ❖ Threads

Objectives

- ❖ To introduce the notion of a process—a program in execution, which forms the basis of all computation.
- ❖ To describe the various features of processes, including scheduling, creation, and termination.
- ❖ To explore interprocess communication using shared memory and message passing.

Process Concept

- **Process** is a program in execution.
- Program is a ***passive*** entity stored on disk (**executable file**) where as process is ***active entity*** being executed.
  - Program becomes process when an executable file is loaded into memory.
- Execution of program can be started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes.
  - Consider multiple users executing the same program as an example.

Process Concept ...

Process in Memory

❖ A process is more than the program code or the **text section**.

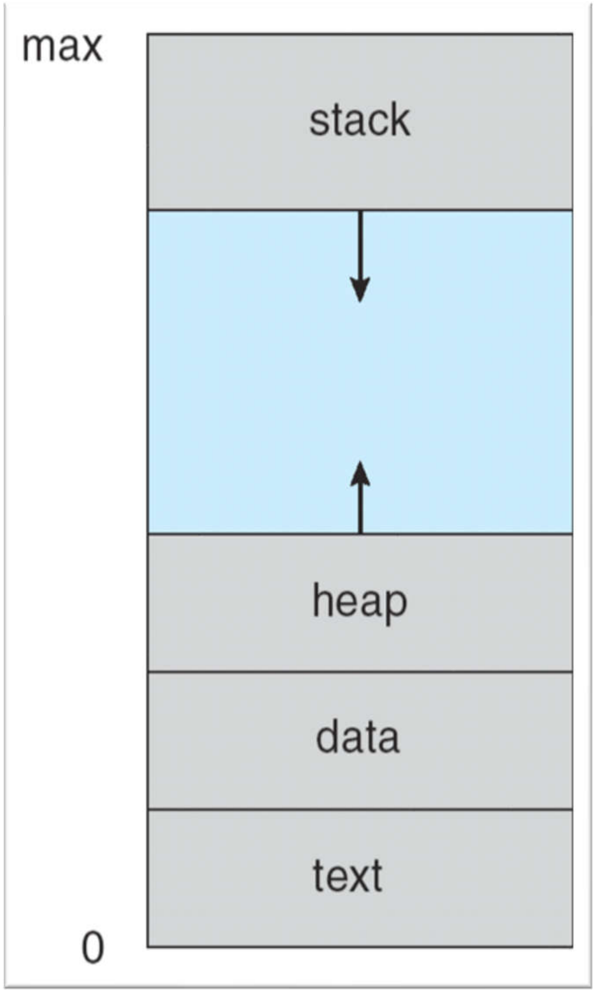
❖ It also includes:

  ❖ program counter to indicate next instruction.

  ❖ stack as a temporary data storage for parameters, return addresses and local variables.

  ❖ data section for global variables.

  ❖ heap dynamically allocated for the process at runtime.



Process Concept ...

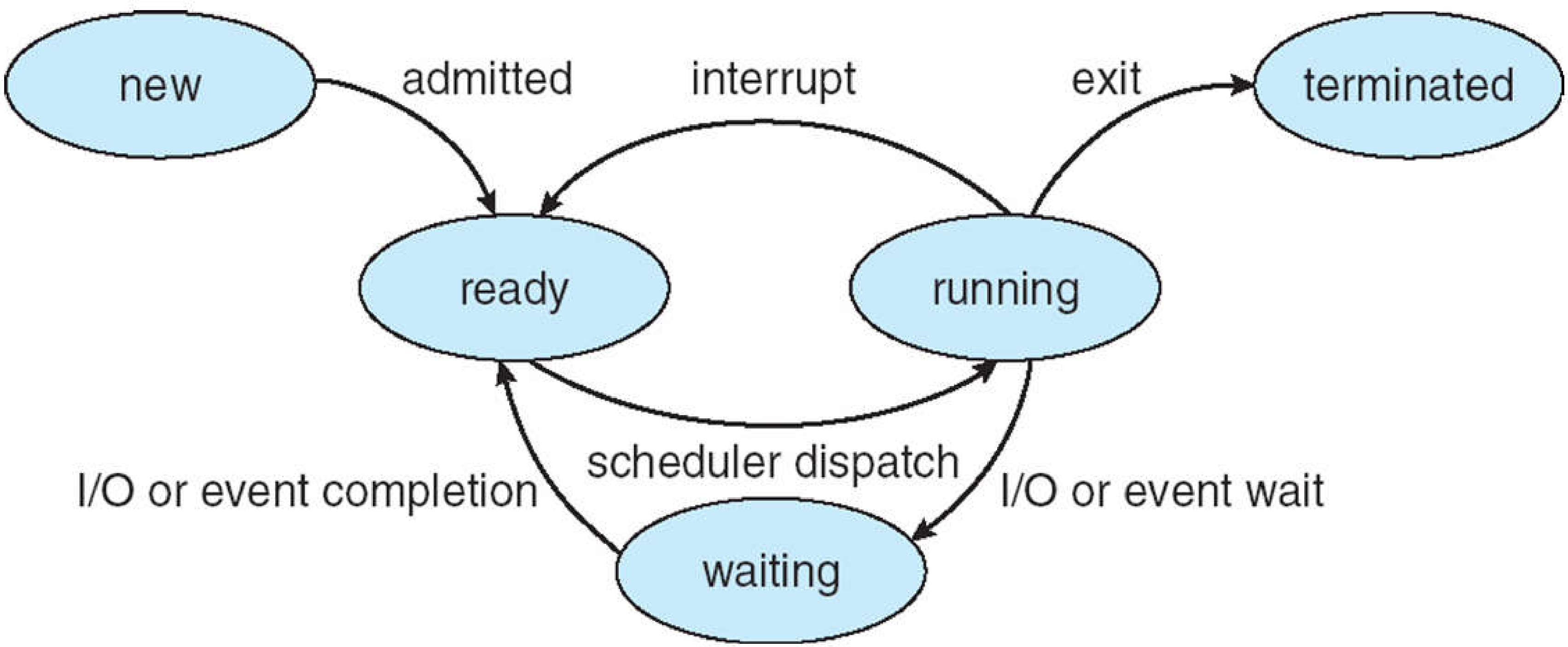
Process State

- As a process executes, it changes **state.** A process may be in one of the following states:
  - **new**: the process is being created.
  - **running**: instructions are being executed.
  - **waiting**: the process is waiting for some event to occur.
  - **ready**: the process is waiting to be assigned to a processor.
  - **terminated**: the process has finished execution.

Process Concept ...

Process State ...

Diagram of Process State

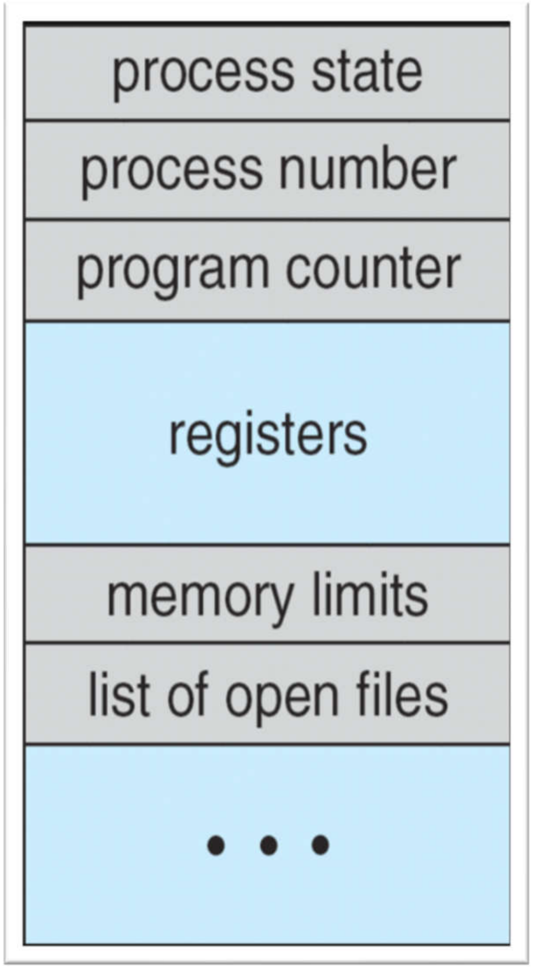


Process Concept ...

Process Control Block (PCB)

PCB (**task control block**) contains many pieces of information associated with a specific process, including these:

- ❖ Process state – new, ready, running, waiting, etc.
- ❖ Program counter – location of instruction to execute next.
- ❖ CPU registers – contents of all process-centric registers.
- ❖ CPU scheduling information- priorities, scheduling queue pointers.
- ❖ Memory-management information – memory allocated to the process, values of the base and index registers.
- ❖ Accounting information – includes: CPU used, real time used, time limits, job or process numbers.
- ❖ I/O status information – I/O devices allocated to process, list of open files.



Process Scheduling

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- To meet these objectives, the **process scheduler** selects among available processes for execution on CPU.

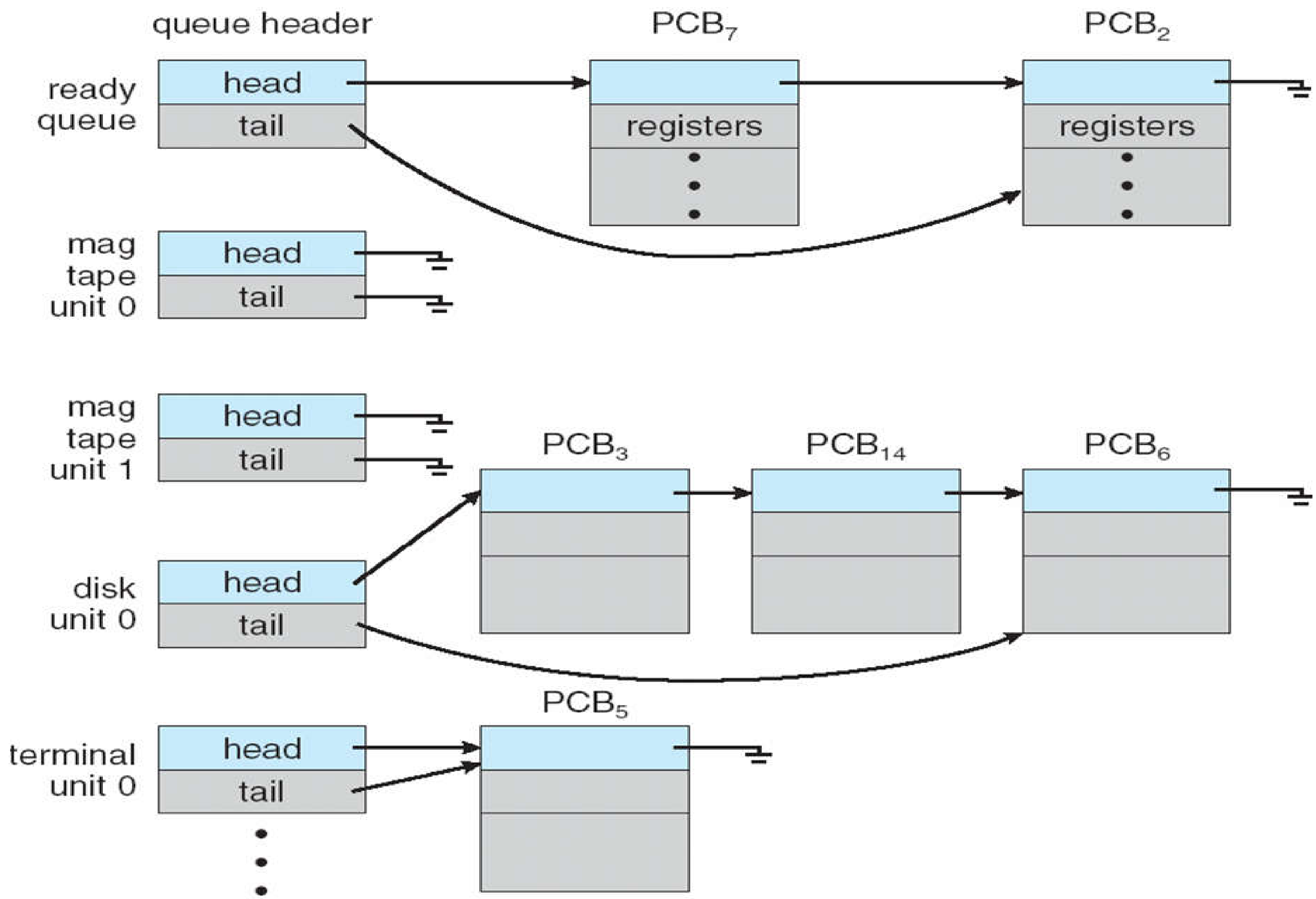
Process Scheduling ...

Scheduling Queues

- As processes enter the system, they are put into a job queue.
  - **Job queue** – consists of all processes in the system.
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute. This queue is stored as a linked list.
  - **Device queues** – set of processes waiting for a particular I/O device. Each device has its own device queue.
  - Processes migrate among the various queues.

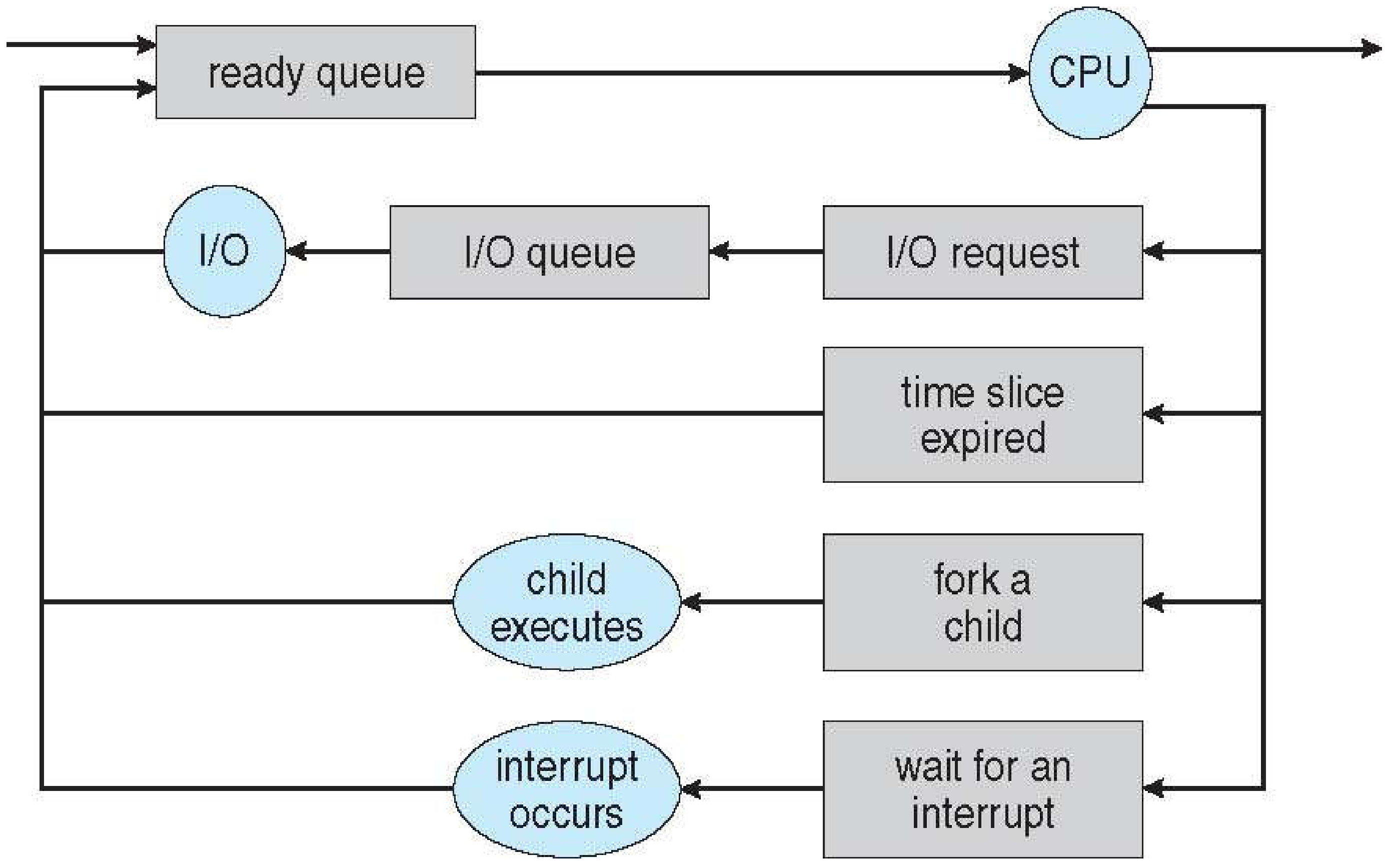
Process Scheduling …

Ready Queue and Various I/O Device Queues



Process Scheduling …

Representation of Process Scheduling



Queuing diagram represents queues, resources, flows

Process Scheduling ...

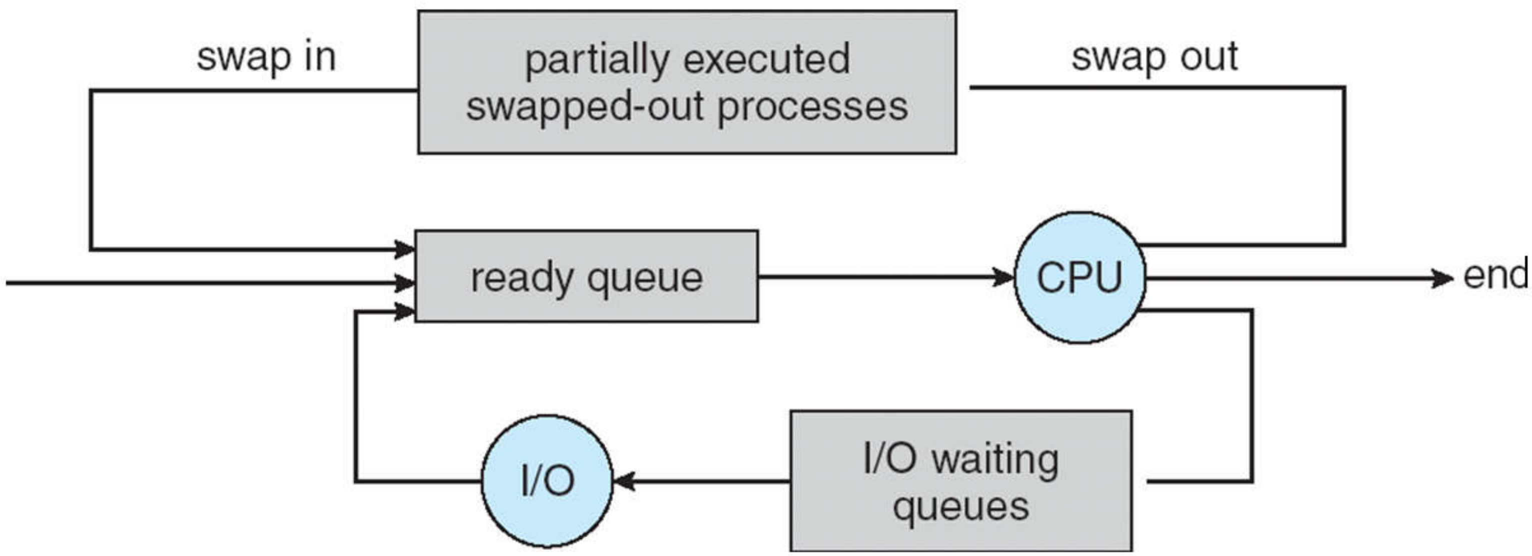
Schedulers

- **Short-term scheduler** (**CPU scheduler**) – selects which process should be executed next and allocates CPU.
  - Sometimes the only scheduler in a system.
  - Short-term scheduler is invoked frequently (milliseconds) $\Rightarrow$ (must be fast)
- **Long-term scheduler** (**job scheduler**) – selects which processes should be brought into the ready queue.
  - Long-term scheduler is invoked infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations.
  - **CPU-bound process** – spends more time doing computations.
- Long-term scheduler selects a good ***process mix*** of I/O-bound and CPU-bound processes.

Process Scheduling …

Addition of Medium Term Scheduling

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



Process 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 is represented in the PCB.
- ❖ Context-switch time is overhead since the system does no useful work while switching.
  - ❖ The more complex the OS and the PCB ➔ the longer the context switch.
- ❖ Context switch is highly dependent on hardware support.
  - ❖ Some processors provide multiple sets of registers.

Operations on Processes

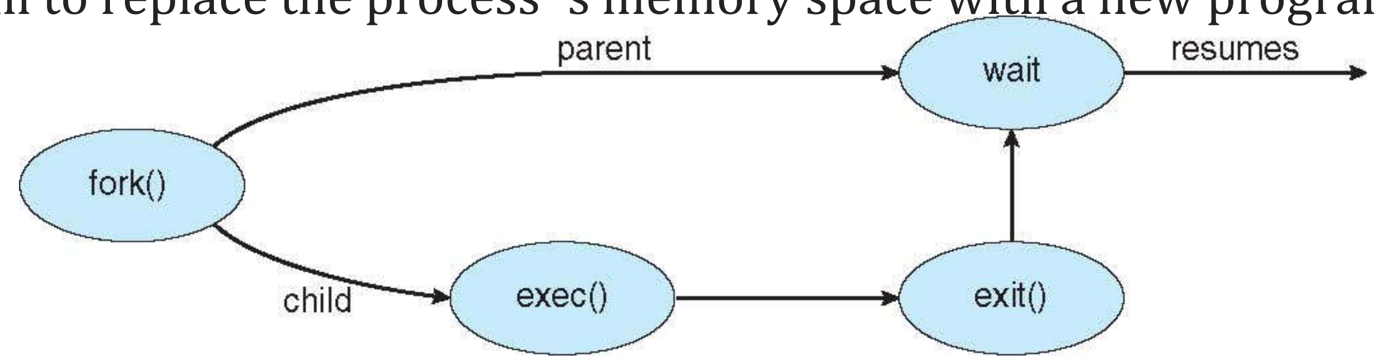
Process Creation

- **Parent** process creates **children** processes, which, in turn create other processes, forming a **tree** of processes.
- Generally, process is identified and managed via a **process identifier** (**pid**).
- Resource sharing options between the parent & child processes.
  - Parent and children share all resources.
  - Children share subset of parent' s resources.
  - Parent and child share no resources.
- Execution options
  - Parent and children may execute concurrently.
  - Parent waits until its children have terminated.

Operations on Processes …

Process Creation …

- ❖ Address space possibilities for a child process:
  - ❖ Child is duplicate of its parent (the same program and data as parent).
  - ❖ Child has a new program loaded into it.
- ❖ UNIX examples
  - ❖ A new process is created by the **fork()** system call.
  - ❖ After a **fork()** system call, one of the processes uses the **exec()** system call to replace the process' s memory space with a new program.



Operations on Processes ...

Process Creation ...

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;
}
```

Operations on Processes …

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
    - The process may return a status value to its parent process via **`wait()`** system call.
    - All process' resources are deallocated by the operating system
- A parent may terminate the execution of one of its children for a variety of reasons, such as these:
    - The child has exceeded the allocated resources
    - Task assigned to child process is no longer required.
    - The parent is exiting and the operating system does not allow a child to continue if its parent terminates.

Operations on Processes ...

Process Termination ...

- If a process terminates, then all its children must also be terminated. This phenomenon is referred to as **cascading termination.**
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the **`pid`** of the terminated process
  - **`pid = wait(&status);`**
- A terminated process whose parent has not yet called **`wait()`** is a **zombie** process.
- If parent terminated without invoking **`wait()`**, process is an **orphan**.

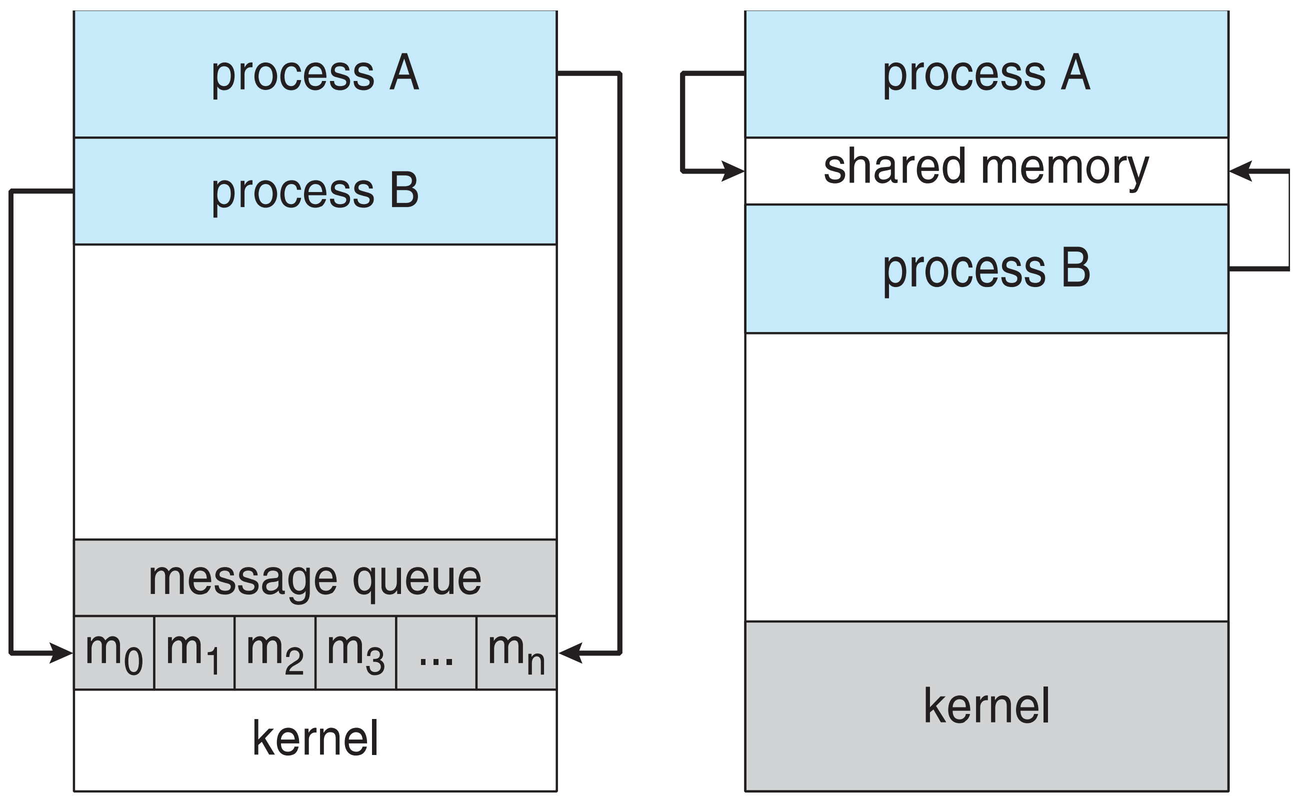
Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating.***
- ***Independent*** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by other processes, including sharing data.
- Any process that shares data with other processes is a cooperating process.
- Reasons for cooperating processes:
  - Information sharing: several users may want to access the same data concurrently.
  - Computation speedup: subdividing a task to run faster if the system is multicore.
  - Modularity: dividing the system functions into separate processes or threads.
  - Convenience: users may work many tasks at same time.
- Cooperating processes need **interprocess communication** (**IPC**) mechanism.

Interprocess Communication …

❖ There are two fundamental models of IPC.

(a) Message passing. (b) shared memory.



(a) (b)

Interprocess Communication …

Shared Memory

- Shared memory is an area of memory shared among the processes that wish to communicate
- The communication is under the control of the user processes not the operating system.
- Major issue is to provide a mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Processes are not allowed to write simultaneously.

Interprocess Communication …

Shared Memory …

Producer-Consumer Problem

- It is a common paradigm for cooperating processes.
- *Producer* process produces information that is consumed by a *consumer* process.
- There must be a buffer of items that can be filled by the producer and emptied by the consumer. The buffer may be:
  - **unbounded-buffer** places no practical limit on the size of the buffer. Producer produces without limit while the consumer waits when the buffer is empty.
  - **bounded-buffer** assumes that there is a fixed buffer size. **Producer** waits when buffer is **full** and **consumer** waits when buffer is **empty**.

Interprocess Communication ...

Shared Memory ...

Producer-Consumer Problem ...

Bounded-Buffer Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Interprocess Communication …

Shared Memory …

Producer-Consumer Problem …

Bounded-Buffer: Producer

The producer process has a local variable `next_produced` in which the new item to be produced is stored.

```
item next_produced;
while (true) {
  /* produce an item in next produced */
  while (((in + 1) % BUFFER_SIZE) == out)
    ; /* do nothing  -- no free buffers */
  buffer[in] = next_produced;
  in = (in + 1) % BUFFER_SIZE;
}
```

Interprocess Communication ...

Shared Memory ...

Producer-Consumer Problem ...

Bounded-Buffer: Consumer

```
item next_consumed;
while (true) {
   while (in == out)
   ; /* do nothing */
   next_consumed = buffer[out];
 out = (out + 1) % BUFFER_SIZE;

 /* consume the item in next consumed */
}
```

Interprocess Communication ...

Message Passing

- Message passing provides a mechanism for processes to communicate and to synchronize their actions without sharing the same address space.
- It is particularly useful in a distributed environment.
- IPC facility provides at least two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size can be either fixed or variable

Interprocess Communication …

Message Passing …

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them.
  - Exchange messages via send/receive.
- Here are several methods for logically implementing a link and the send()/receive() operations:
  - Direct or indirect communication
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering

Interprocess Communication …

Message Passing …

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 in this scheme:
  - Links are established automatically if processes to communicate know each other's identity.
  - A link is associated with exactly two processes.
  - Between each pair of processes, there exists exactly one link.

Interprocess Communication …

Message Passing …

Indirect Communication

- Messages are sent to and received from **mailboxes**, or **ports**.
- Each mailbox has a unique id and processes can communicate only if they have a shared a mailbox.
  - **`send`** (*A, message*) – send a message to mailbox A.
  - **`receive`**(*A, message*) – receive a message from mailbox A.
- Properties of communication link
  - The link is established only if processes share a common mailbox.
  - A link may be associated with more than two processes.
  - Each pair of processes may share several communication links.

Interprocess Communication …

Message Passing …

Synchronization

- Message passing may be either blocking or non-blocking.
- **Blocking** is considered **synchronous**
  - **Blocking send --** the sender is blocked until the message is received.
  - **Blocking receive --** the receiver blocks until a message is available.
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send --** the sender sends the message and continues.
  - **Non-blocking receive --** the receiver receives either a valid message or null.
- Different combinations of send() and receive() are possible.
  - If both send() and receive() are blocking, we have a **rendezvous** (like planned meeting with a certain time and place.)

Interprocess Communication …

Message Passing …

Buffering

- Messages exchanged by communicating processes reside in a temporary queue.
- Such queues can be implemented in three ways:
  1. **Zero capacity** – no messages are queued on the link. Sender must block to wait for receiver (rendezvous).
  2. **Bounded capacity** – the queue has finite length *n,* thus n of messages. Sender must block (wait) if link full.
  3. **Unbounded capacity** – infinite queue length. Sender never blocks.

Interprocess Communication …

Message Passing …

Examples: POSIX Shared Memory

- POSIX Shared Memory is organized using memory-mapped files.
  - Process first creates shared memory segment

```
int shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

  - The last parameter establishes the directory permissions of the shared-memory object. Also used to open an existing segment to share it.
  - Set the size of the object

```
ftruncate(shm_fd, 4096);
```

  - Now the process could write to the shared memory

```
sprintf(shm_fd, "Writing to shared memory");
```

Interprocess Communication …

Message Passing …

POSIX Producer

```
#include <stdio.h>
#include <stlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDRW, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

POSIX Consumer

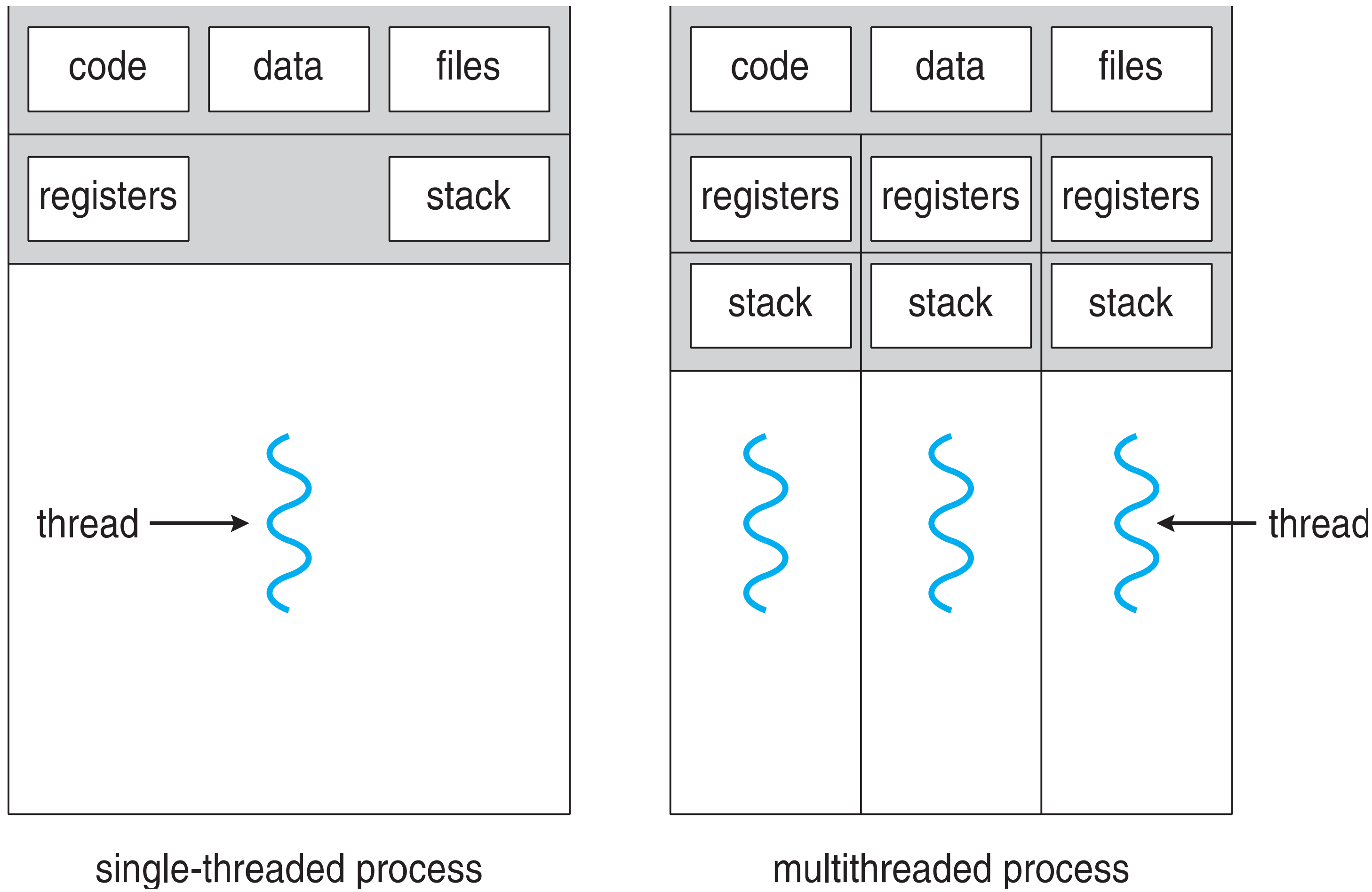
```
#include <stdio.h>
#include <stlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```

Threads

- A thread is a basic unit of CPU utilization.
- It comprises a threadID, a program counter, a register set, and a stack.
- It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.
- A traditional (or heavy weight) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

Threads ...

Single and Multithreaded Processes



Threads …

Benefits of Multithreaded Programming

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces.
- **Resource Sharing –** threads share memory and the resources of process, easier than shared memory or message passing.
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching.
- **Scalability –** process can take advantage of multiprocessor architectures.

Threads …

Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Identifying tasks:** involves examining applications to find areas that can be divided into separate, concurrent tasks.
  - **Balance:** programmers must also ensure that the tasks perform equal work of equal value.
  - **Data splitting:** as tasks divide, the data used to run them need to be divided.
  - **Data dependency:** data accessed by tasks must be checked for dependency and synchronized.
  - **Testing and debugging:** is more difficult in parallel tasks passing different paths.
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

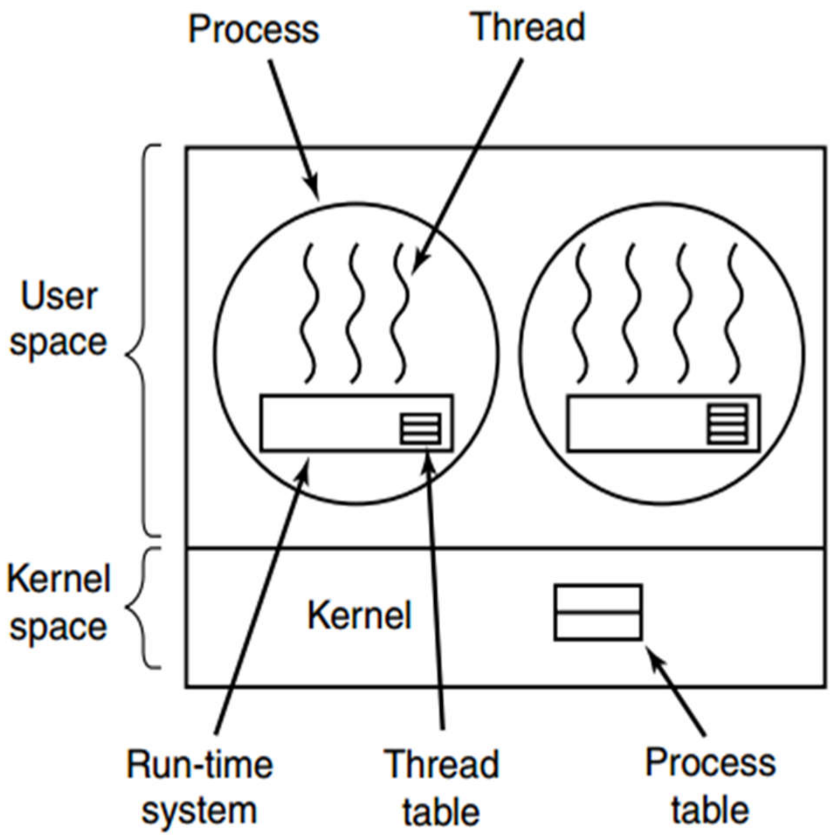
Threads ...

User Threads and Kernel Threads

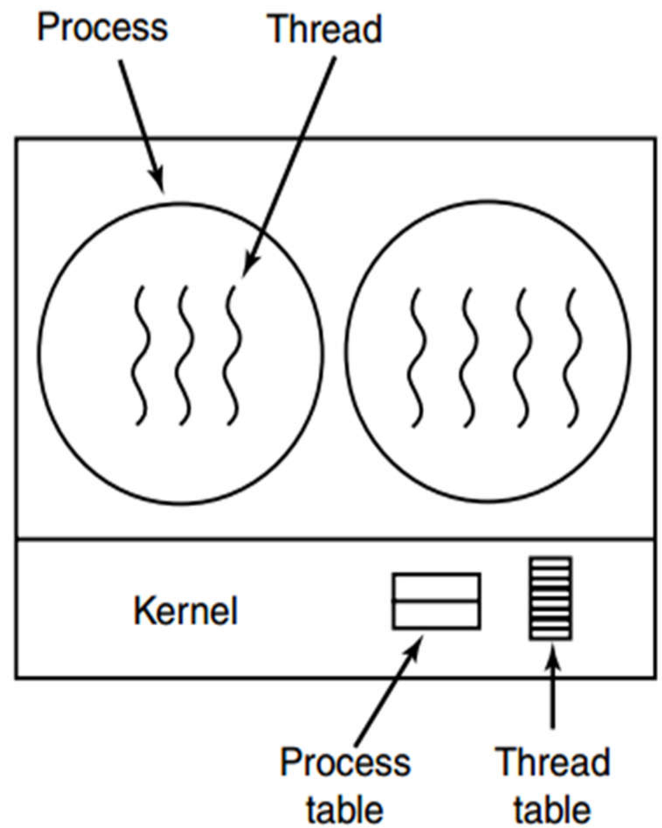
- There are **two** primary ways of implementing a thread library.
  - To implement a library entirely in **user space** with no kernel support.
  - To implement a **kernel-level** library supported directly by the OS.
- **Three primary** thread libraries are in use today:
  - **Pthreads:** the threads extension of the POSIX standard. Pthreads may be provided as either a user-level or a kernel-level library.
  - **Windows thread** library: is a kernel-level library on Windows systems.
  - **Java threads:** created and managed directly in Java programs.

Threads ...

User Threads and Kernel Threads ...



(a) A user-level threads package



(b) A threads package managed by the kernel.

Threads …

Kernel Threads

- A **kernel thread**, also known as a **lightweight process**, is a thread that the operating system knows about.
- Switching between kernel threads of the same process requires a small context switch.
- The values of registers, program counter, and stack pointer must be changed.
- Memory management information does not need to be changed since the threads share an address space.
- The kernel must manage and schedule threads (as well as processes), but it can use the same process scheduling algorithms.
- Switching between kernel threads is slightly faster than switching between processes.

Threads ...

User-Level Threads

- A user-level thread is a thread that the OS does not know about.
- The OS only knows about the process containing the threads.
- The OS only schedules the process, not the threads within the process.
- The programmer uses a thread library to manage threads (create and delete them, synchronize them, and schedule them).

Threads ...

User-Level Threads ...

Advantages ...

- There is no context switch involved when switching threads.
- User-level thread scheduling is more flexible.
  - A user-level code can define a problem-dependent thread scheduling policy.
  - Each process might use a different scheduling algorithm for its own threads.
  - A thread can voluntarily give up the processor by telling the scheduler it will yield to other threads.
- User-level threads do not require system calls to create them or context switches to move between them.
- User-level threads are typically much faster than kernel thread.

Threads …

User-Level Threads …

Disadvantages …

- Since the OS does not know about the existence of the user-level threads, it may make poor scheduling decisions:
  - It might run a process that only has idle threads.
  - If a user-level thread is waiting for I/O, the entire process will wait.
  - Solving this problem requires communication between the kernel and the user-level thread manager.
- Since the OS just knows about the process, it schedules the process the same way as other processes, regardless of the number of user threads.
- For kernel threads, the more threads a process creates, the more time slices the OS will dedicate to it.

Threads …

Multithreading Models

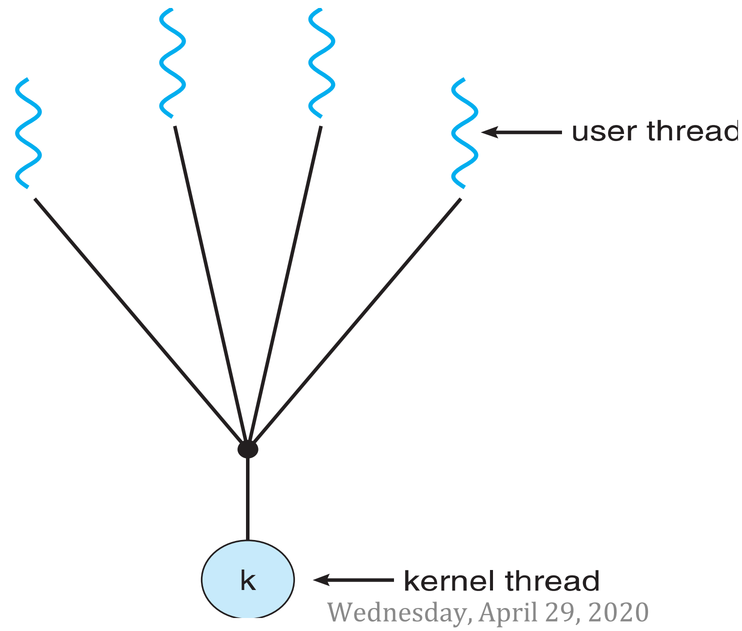
- A relationship must exist between user threads and kernel threads.
- Three common ways of establishing such a relationship:
  - Many-to-One
  - One-to-One
  - Many-to-Many

Threads …

Multithreading Models …

Many-to-One

- ❖ Many user-level threads mapped to single kernel thread.
- ❖ Entire process will be blocked if a thread makes a blocking system call.
- ❖ Multiple threads cannot run in parallel on multicore system because only one thread can access the kernel at a time.
- ❖ Few systems currently use this model.
- ❖ Examples:
  - ❖ Solaris Green Threads
  - ❖ GNU Portable Threads

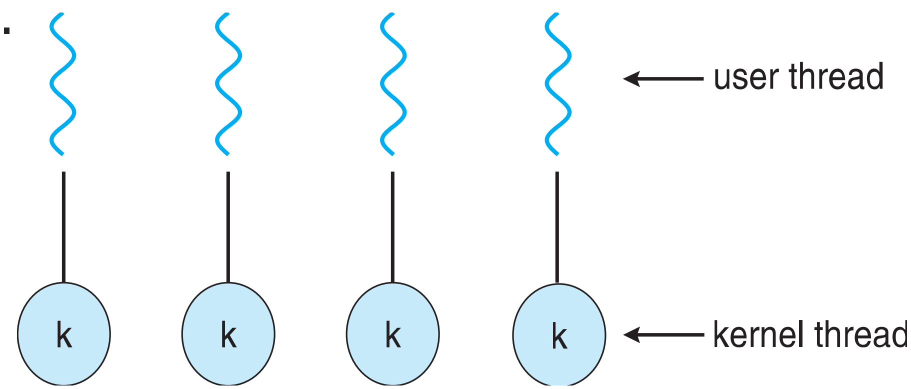


Threads …

Multithreading Models …

One-to-One

- Each user-level thread mapped to a kernel thread.
- This allows more concurrency than many-to-one model.
- Creating a user-level thread requires creating a corresponding kernel thread.
- Number of threads per process sometimes restricted due to the overhead creating kernel threads.
- Examples
  - Windows, Linux, Solaris 9 and later

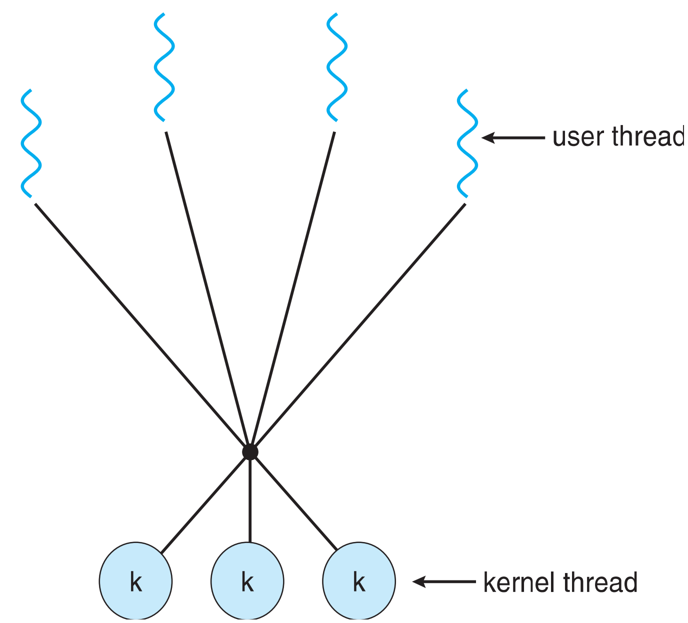


Threads …

Multithreading Models …

Many-to-Many Model

- Multiplexes many user-level threads to a smaller or equal number of kernel threads.
- Allows developers to create sufficient number of user threads.
- E.g. Solaris older versions then version 9.



Threads …

Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- There are two primary ways of **implementing** a thread library.
  - To provide a library entirely in user space with no kernel support.
  - To implement a kernel-level library supported directly by the operating system.

Threads ...

PThreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization.
- This is a ***Specification*** for thread behavior, not ***implementation.***
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Threads …

PThreads Example

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
    return -1;
  }
```

Threads ...

PThreads Example ...

```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Reference: Silberschatz et al., Operating System Concepts, Ninth Edition, 2013.

End of Chapter 2

Questions???