Unit II
PROCESS

1. **Process Concept**
    - **1.1. The Process**
    - **1.2. Process State**
    - **1.3. Process Control Block**
    - **1.4. Threads**
2. **Process Scheduling**
    - **2.1. Scheduling Queues**
    - **2.2. Schedulers**
    - **2.3. Context Switch**
3. **Operations on Processes**
    - **3.1. Process Creation**
    - **3.2. Process Termination**
4. **Interprocess Communication**
    - **4.1. Shared-Memory Systems**
    - **4.2. Message-Passing Systems**

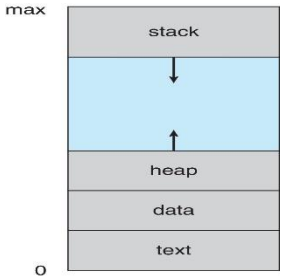
Unit II
PROCESS CONCEPT

1. Process Concept

- A process is the unit of work in a modern time-sharing system.
- An operating system executes a variety of programs that run as a process.

1.1. The Process

- Process – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process
- Multiple parts of a process:
  - The program code, also called text section
  - Stack containing temporary data
    - ✓ Function parameters, return addresses, local variables
  - Data section containing global variables
  - Heap containing memory dynamically allocated during run time

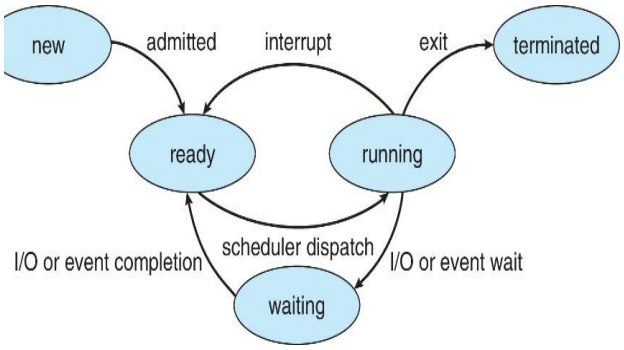


- Program is passive entity stored on disk (executable file): process is active
  - Program becomes process when an executable file is loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
  - Consider multiple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

1.2. Process State

As a process executes, it changes state

- New: The process is being created
- Ready: The process is waiting to be assigned to a processor
- Running: Instructions are being executed
- Waiting: The process is waiting for some event to occur
- Terminated: The process has finished execution

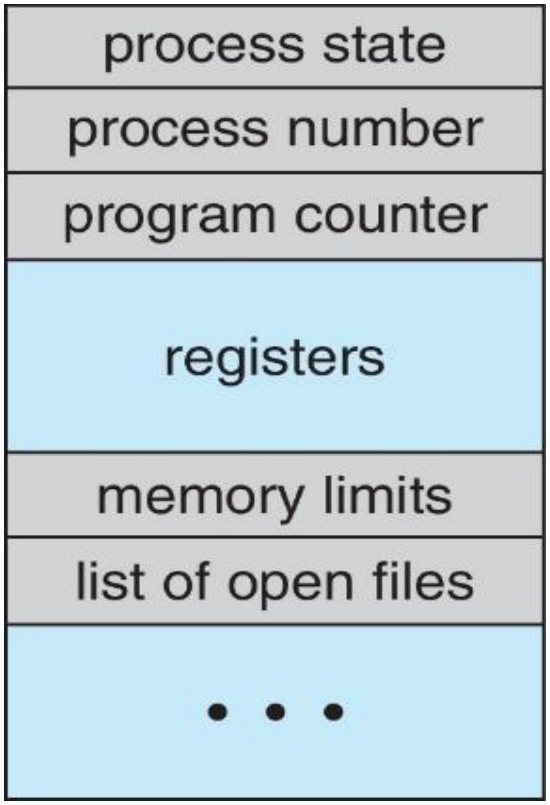


- These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems.
- Only one process can be running on any processor core at any instant of time. Many processes may be ready and waiting, however. The state diagram corresponding to these states is presented in Figure above

1.3.Process Control Block

Each process is represented in the operating system by a process control block (PCB) also called a task control block.

- Process state – the state many be new, running, waiting, halted, and so on.
- Program counter – location of next instruction to execute
- CPU registers – contents of all process-centric registers, they include accumulators, index registers, stack pointers, and general-purpose registers.
- CPU scheduling information- process priority, pointers to scheduling queues, and any other scheduling parameters.
- Memory-management information – memory allocated to the process: value of the base and limit registers and the page tables, or the segment tables,
- Accounting information – amount of CPU time used, time limits, account numbers, job or process numbers
- I/O status information – List of I/O devices allocated to process, list of open files and so on.



- In brief, the PCB simply serves as the repository for any information that may vary from process to process.

1.4.Threads

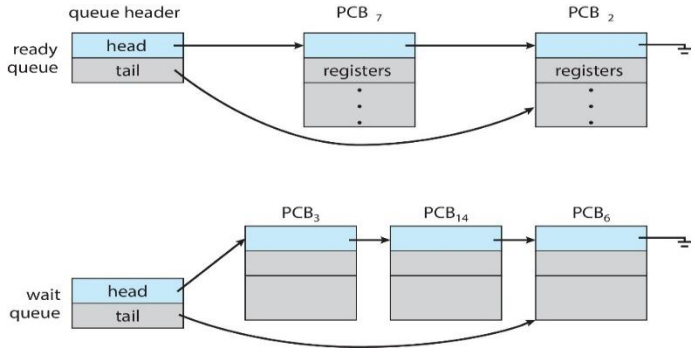
- A Process is a program that performs a single thread of execution.
- For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread control allows the process to perform only one task at a time. The user cannot simultaneously type in characters and run the spell checker within the same process.
- Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time.
- This feature is especially beneficial on multicore systems, where multiple threads can run in parallel.
- On a system that supports threads, the PCB is expanded to include information for each thread.

2. Process Scheduling

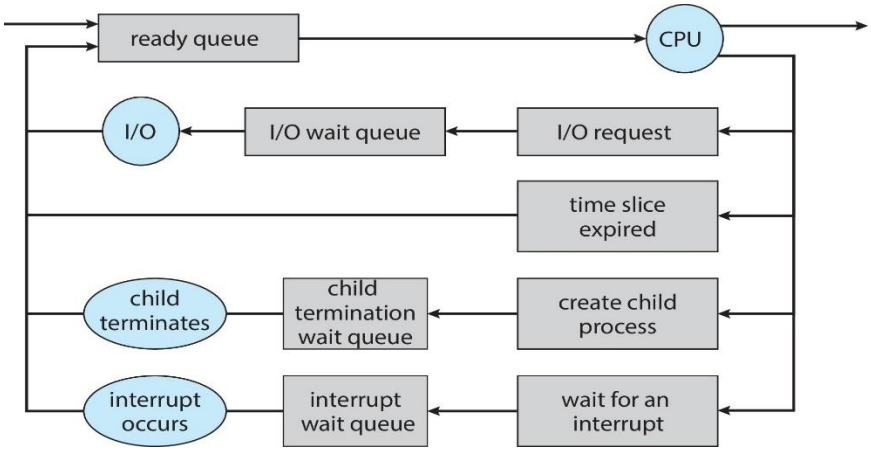
- The objective of multiprogramming is to have process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- To meet these objectives, the process ***scheduler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

2.1. Scheduling Queues

- As process enter the system, they put into a ***job queue***, which consists of all processes in the system.
- Maintains scheduling queues of processes
  - Ready queue – set of all processes residing in main memory, ready and waiting to execute
  - Wait queues – set of processes waiting for an event (i.e., I/O)
  - Processes migrate among the various queues
- This queue is generally stored as a linked list. Each PCB includes a pointer field that points to the next PCB in the ready queue.



- When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request.
- Suppose the process makes an I/O request to a shared device, such as disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process may have to wait for the disk.
- The list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue.



- A common representation for a discussion of process scheduling is a queueing diagram, such as that in Figure. Each rectangular box represents a queue.
- A new process is initially put in the ready queue. It waits there until it is selected for execution, or is dispatched. Once the process is allocated the CPU and is executing, one of several events could occur:
  - The process could issue an I/O request and then be placed in an I/O queue.
  - The process could create a new subprocess and wait for the subprocess's termination.
  - The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

2.2. Schedulers

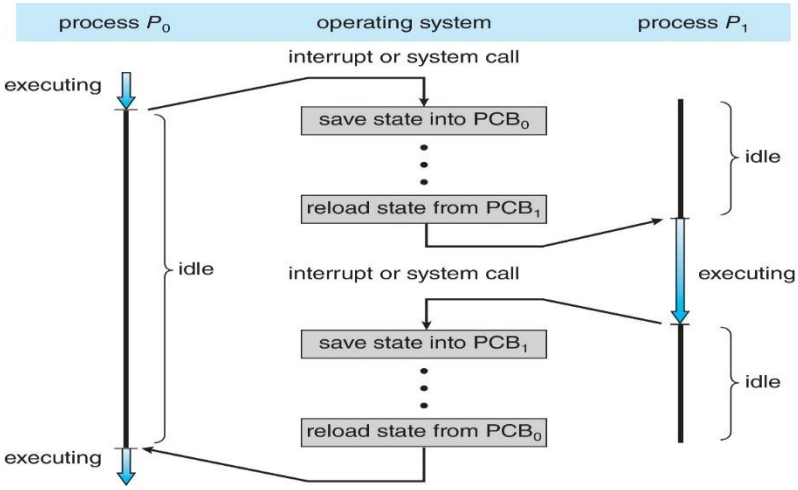
- A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate scheduler.
- Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device, where they are kept for later execution.
- The long-term scheduler, or job scheduler, selects processes from this pool and loads them into memory for execution.
- The short-term scheduler, or CPU scheduler, selects from among the processes that are ready to execute and allocates the CPU to one of them.
- The primary distinction between these two schedulers lies in frequency of execution.
- The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 100 milliseconds. Because of the short time between executions, the short-term scheduler must be fast.

- The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next. The long-term scheduler controls the degree of multiprogramming (the number of processes in memory).
- If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. Thus, the long-term scheduler may need to be invoked only when a process leaves the system.
- In general, most processes can be described as either I/O bound or CPU bound.
- An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations.
- A CPU bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations.
- The long-term scheduler select a good process mix of I/O-bound and cpu-bound processes. If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do. If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced.
- The system with the best performance will thus have a combination of CPU-bound and I/O-bound processes.
- Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling - medium-term scheduler.
- The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called ***swapping***.
- The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.

2.3. Context Switch

- A Context Switch occurs when the CPU switches from one process to another.
- The context is represented in the PCB of the process; it includes the value of the CPU registers, the process state, and memory-management information.
- 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
- Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a ***context switch***.

- Context-switch time is pure overhead, because the system does no useful work while switching.
- Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions.



3. Operations on Processes

- The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and process termination.

3.1.Process Creation

- During the course of execution, a process may create several processes. Parent process create children processes, which in turn create other processes, forming a tree of processes.
- In general, when a process creates a child process that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task.
- A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of resources of the parent process.
- The parent may have to partition its resources among its children, or it may be able to share some resources among several of its children.
- When a process creates a new process, two possibilities for execution exist:
    1. The parent continues to execute concurrently with its children
    2. The parent waits until some or all of its children have terminated
- There are also two address-space possibilities for the new process:
    1. The child process is a duplicate of the parent process.
    2. The child process has a new program loaded into it.

Example: Process creation in UNIX Operating System

- In UNIX, each process is identified by its identifier, which is a unique integer.

- A new process is created by the **fork()** system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process.
- Both processes continue execution at the instruction after the **fork()**, with one difference: the return code for the **fork()** is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.
- After a **fork()** system call, one of the two processes typically uses the **exec()** system call to replace the process's memory space with a new program.
- **The exec()** system call loads a binary file into memory (destroying the memory image of the program containing the exec 0 system call) and starts its execution.
- In this manner, the two processes are able to communicate and then go their separate ways. The parent can then 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.

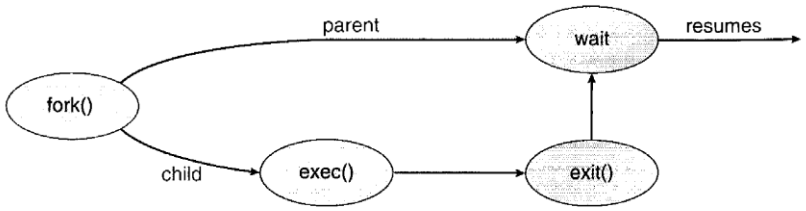


Figure 3.11 Process creation.

3.2.Process Termination

- Process executes last statement and then and asks the operating system to delete it using the **exit()** system call. At that point, the process may return a status value to its parent process.
- All the resources of the process including physical and virtual memory, open files, and I/O buffers are deallocated by the operating system.
- Termination can occur in other circumstances as well.
- A process can cause the termination of another process via an operating system call. Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrary kill each other's jobs.
- A parent may terminate the execution of one of its children for a variety of reasons, such as these:
  - The child has exceeded its usage of some of the resources that it has been allocated.
  - The task assigned to the child is no longer required.
  - The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

- Some system do not allow a child to exist if its parent has terminated. In such systems, if a process terminates, then all its children must also be terminated. This phenomenon, referred to as **cascading termination**, is normally initiated by the operating system.
- To illustrate process execution and termination, consider that, in LINUX and UNIX systems, we can terminate a process by using the e**xit()** system call, providing an exit status as parameter:

```
/* exit with status 1 */
exit(1);
```

- A parent process may wait for the termination of a child process by using the **wait()** system call. The **wait()** system call is passed a parameter that allows the parent to obtain the exit status of the child. This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

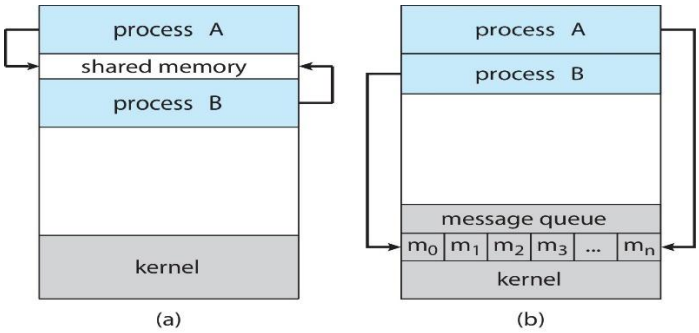
```
pid_t pid;
int status:
pid = wait(&status);
```

- When a process terminates, its resources are deallocated by the operating system.
- A process that has terminated, but whose parent has not yet called **wait()** is known as a **zombie** process. A parent process terminated without invoking **wait()** is known as **orphan** process**.**

4. Interprocess Communication

- Process executing concurrently in the operating system may be either independent processes or cooperating process.
- A process is **independent** if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent.
- A process in **cooperating** if it can affect or be affected by the other processes executing in the system. Clearly any process that shares data with other process is a cooperating process.
- Reasons for providing an environment that allows process cooperation:
  - **Information sharing**: Since several users may be interested in the same piece of information, we must provide an environment to allow concurrent access to such information.
  - **Computation speedup:** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.
  - **Modularity:** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
  - **Convenience:** Even an individual user may work on many tasks at the same time. For instance, a user many be editing, listening to music, and compiling in parallel.
- Cooperating processes need interprocess communication (IPC) mechanism that allow them to exchange data and information. There are two fundamental models of IPC:
  - Shared memory and
  - Message passing

- In the shared-memory model, a region of memory that is shared by cooperating process is established. Processes can then exchange information by reading and writing data to the shared region.
- In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.
- The two communications models are contrasted in Figure below



4.1. Shared-Memory Systems

- In the shared-memory model, a region of memory that is shared by cooperating process is established. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment.
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space. The processes can exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control.
- The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

Example: Producer-Consumer Problem

- A producer process produces information that is consumed by a consumer process.
- For example, a compiler may produce assembly code that is consumed by an assembler. The assembler in turn may produce object modules that are consumed by the loader.
- One solution to the producer – consumer problem uses shared memory.
  - To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
  - This buffer will reside in a region of memory that is shared by the producer and consumer processes. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.
- Two types of buffers can be used
  - The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.
  - The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.
- One solution to the producer – consumer problem uses shared memory.
  - The following variables reside in a region of memory shared by the producer and consumer processes:

```
#define BUFFER_SIZE 10
typedef struct {
```

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

- The shared buffer is implemented as a circular array with two logical pointers: **in** and **out.** The variable **in** points to the next free position in the buffer; **out** points to the first full position in the buffer.

- The code for the producer process is

```
item next_produced;
while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

- The code for the consumer process is

```
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 */
}
```

- The producer process has a local variable next_produced in which the new item to be produced is stored. The consumer process has a local variable next_consumed in which the item to be consumed is stored.
- Solution is correct, but can only use BUFFER_SIZE-1 elements

4.2. Message-Passing Systems

- In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.
- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.
- A message-passing facility provides at least two operations:
  send(message) receive(message)
- Message sent by a process can be either fixed or variable in size.
- If processes P and Q want to communicate
  - they must send messages to and receive messages from each other:
  - a communication link must exist between them.
- There are several methods for logically implementing a link and send() / receive() operations:
  - Direct or indirect communication
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering

4.2.1. Naming

- Under ***direct communication***, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send() and receive() primitives are defined as:
  - send(P, message) – Send a message to process P
  - receive(Q, message) – Receive a message from process Q
- This scheme exhibits ***symmetry*** in addressing; that is both, the sender process and receiver process must name the other to communicate.
- In ***asymmetric*** scheme, only the sender names the recipient; the recipient is not required to name the sender.
  - send(P, message) – Send a message to process P
  - receive(id, message) – Receive a message from any process
- The disadvantage in both of these schemes is the limited modularity of the resulting process definitions.
- With ***indirect communication,*** the messages are sent to and received from ***mailboxes***.
- A Mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
- Each mailbox has a unique identification. A process can communicate with another process via a number of different mailboxes, but two processes communicate only if they have a shared mailbox.
- The send() and received() primitives are defined as follows:
  - send(A, message) – Send a message to mailbox A.
  - receive(A, message) – Received a message from mailbox A.
- In this scheme. a communication link has the following properties
  - 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, with each link corresponding to one mailbox.
- Mailbox Sharing
  - Suppose that processes P1, P2, and P3 all share mailbox A
  - P1 sends a message to A, while P2 and P3 execute ***receive()*** operation
  - Who gets the message?
- The solution depends on which of the following methods we choose
  - Allow a link to be associated with two processes at most.
  - Allow at most one process at a time to execute a ***receive()*** operation.
  - Allow the system to select arbitrary which process will receive the message.

4.2.2. Synchronization

- Message passing may be either **blocking** or **nonblocking**.
- Blocking is considered **synchronous**
  - Blocking send -- the sender is blocked until the message is received
  - Blocking receive -- the receiver is blocked until a message is available
- Non-blocking is considered **asynchronous**
  - Non-blocking send -- the sender sends the message and continue
  - Non-blocking receive -- the receiver receives either a valid message, or a null message

- Different combinations of ***send()*** and ***receive()*** are possible. When both ***send()*** and ***receive()*** are blocking, we have a **rendezvous** between the sender and the receiver.
- The solution to the producer-consumer problem becomes trivial when we use blocking ***send()*** and ***receive()*** statements.
  - The producer merely invokes the blocking ***send()*** call and waits until the message is delivered to either the receiver or the mailbox.

```
message next_produced;
while (true) {
    /* produce an item in next_produced */
send(next_produced);
}
```

  - The consumer invokes ***receive()***, it blocks until a message is available.

```
message next_consumed;
while (true) {
receive(next_consumed)
/* consume the item in next_consumed */
}
```

4.2.3. Buffering

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:
  1. Zero capacity – The queue has a maximum length of zero; no messages are queued on a link. Sender must block until the recipient receives the message.
  2. Bounded capacity – The queue has a finite length of n; thus, at most n can reside in it. Sender must wait if link is full
  3. Unbounded capacity – The queue's length is potentially infinite; Sender never blocks.

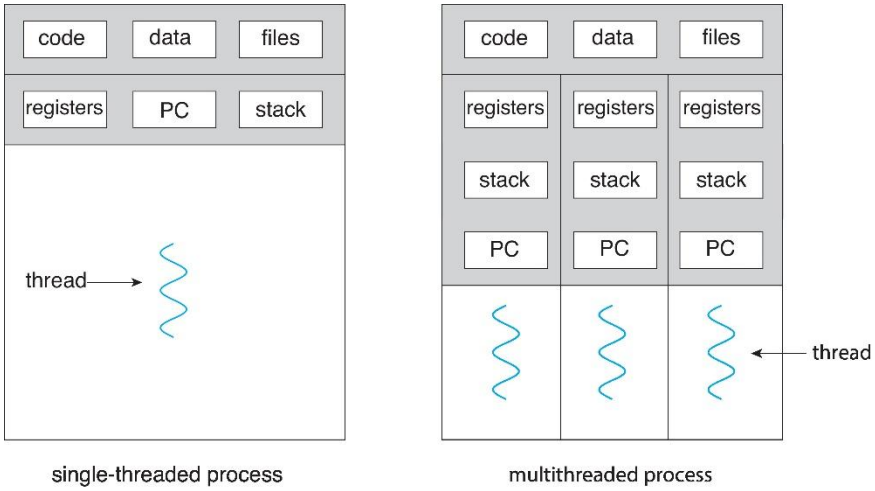
Unit II
THREADS AND CONCURRENCY
Threads

1. **Overview**
   - **1.1. Motivation**
   - **1.2. Benefits**
2. **Multicore Programming**
3. **Multithreading Models**
   - **3.1. Many-to-One Model**
   - **3.2. One-to-One Model**
   - **3.3. Many-to-Many Model**
4. **Thread Libraries**
   - **4.1. Pthreads**
   - **4.2. Win32 Threads**
   - **4.3. Java Threads**
5. **Threading Issues**
   - **5.1. The fork() and exec() System Calls**
   - **5.2. Cancellation**
   - **5.3. Signal Handling**
   - **5.4. Thread Pools**
   - **5.5. Thread-Specific Data**
   - **5.6. Scheduler Activations**

Unit II
THREADS AND CONCURRENCY

1. Overview

- A thread is a basic unit of CPU utilization; it comprises a thread ID, 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 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.



single-threaded process multithreaded process

1.1. Motivation

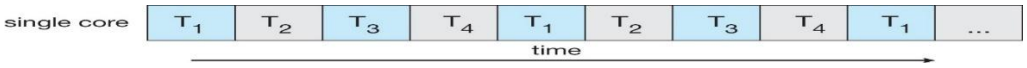
- Most software applications that run on modern computers are multithreaded.
- An application typically is implemented as a separated process with several threads of control.
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

1.2. Benefits

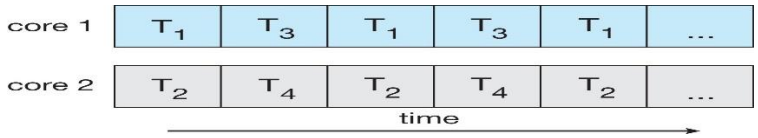
- **Responsiveness** – Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation. It is especially useful in designing user interfaces.
- **Resource Sharing** – Processes can share resources through techniques such as shared memory and message passing.
- **Economy** – Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.
- **Scalability** – The benefits of multithreading can be even greater in multiprocessor architecture, where threads may be running n parallel on different processing cores.

2. Multicore Programming

- Multithread programming provides a mechanism for more efficient use of multiple cores and improved concurrency.
- Consider an application with four threads.
- On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time, because the processing core is capable of executing only one thread at a time.



- On a system with multiple cores, however, concurrency means that the threads can run in parallel, because the system can assign a separate thread to each core.



- A system is ***parallel*** if it can perform more than one task simultaneously. In contrast, a ***concurrent*** system supports more than one task by allowing all the tasks to make progress.

2.1. Programming Challenges:

In general, five areas present challenges in programming for multicore systems

1. **Identifying tasks**: This involves examining applications to find areas that can be divided into separate, concurrent tasks.
2. **Balance:** While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value.
3. **Data splitting:** Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.
4. **Data dependency:** The data accessed by the tasks must be examined for dependencies between two or more tasks. When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency.
5. **Testing and debugging:** When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications.

2.2. Types of Parallelism

- In general, there are two types of parallelism: data parallelism and task parallelism.
- ***Data parallelism*** focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core.
- Consider, for example, summing the contents of an array of size *N*.
  - On a single-core system, one thread would simply sum the elements [0] … [*N*-1].
  - On a dual-core system, however, thread A, running on core 0, could sum the elements [0]…[*N/2*-1] while thread B, running on core 1, could sum the elements [*N*/2] … [*N*-1]. The two threads would be running in parallel on separate computing cores.
- **Task parallelism** involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data.

- Consider an example of task parallelism might involve two threads, each performing a unique statistical operation on the array of elements. The threads again are operating on separate computing cores, but each is performing a unique operation.

AMDAHL'S LAW

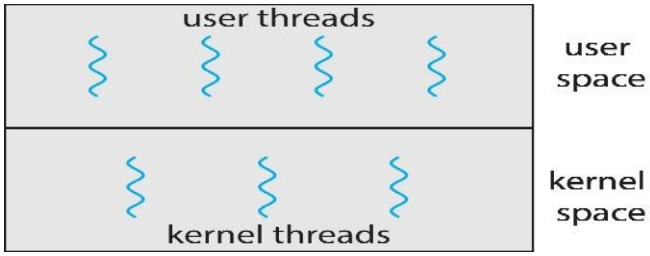
- AMDAHL'S Law is a formula that identifies potential performance gains from adding additional computing cores to an application that has both serial and parallel components.
- If S is the portion of the application that must be performed serially on a system with N processing cores, the formula is as follows:

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- As an example, if we have an application that is 75% parallel and 25% serial, moving from one to two cores results in speedup of 1.6 times. If we add two additional cores, the speedup is 2.28 times. As N approaches to infinity, speedup approaches to 1/S.
- Serial portion of an application has disproportionate effect on performance gained by adding additional cores

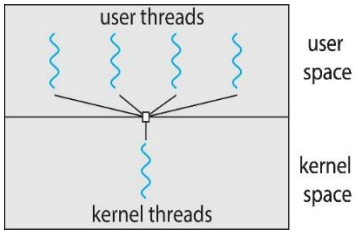
3. Mutlithreading Models

- User threads are managed by user-level thread libraries without the kernel support. Three primary thread libraries are POSIX Pthreads, Windows threads, Java Threads
- Kernel threads are supported and managed directly by the operating system. Virtually all general-purpose operating systems including Windows, Linux, Mac OS X, and Solaris support kernel threads.



3.1. Many-to-one Model

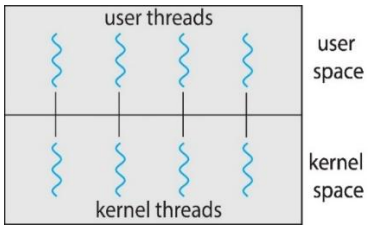
- The many-to-one model maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient.
- The entire process will block if a thread makes a blocking system call.
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time.Very few systems continue to use the model because of its inability to take the advantage of multiple processing cores.
- Examples: Solaris Green Threads and GNU Portable Threads



3.2. One-to-One Model

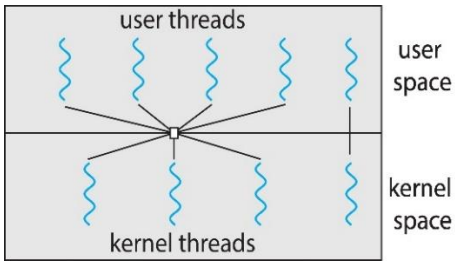
- The one-to-one model maps each user thread to a kernel thread.

- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call. It also allows multiple threads to run in parallel on multiprocessors.
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.
- Number of threads per process sometimes restricted due to overhead.
- Examples: Windows and Linux



3.3. Many-to-Many Model

- Allows many user-level threads to be mapped to many kernel threads.
- The many-to-one model allows the developer to create as many user threads as he wishes, it does not result in true concurrency, because the kernel can schedule only one thread at a time.
- The one-to-one model allows greater concurrency, but the developer has to be careful not to create too many threads within an application.
- The many-to-many model suffers from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
- One variation of the many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation is sometimes referred to as the two-level model.



4. Thread Libraries

- A thread library provides the programmer with a API for creating and managing threads.
- There are two primary ways of implementing a thread and library.
  1. Provide a library entirely in user space with no kernel support.
     - All code and data structures for the library exist in user space.
     - Invoking a function in the library results in a local function call in user space.
  2. Kernel-Level library supported directly by the operating system.
     - Code and data structures for the library exist in the kernel space.
     - Invoking a function in the API for the library typically results in a system call to the kernel
- Three main thread libraries are in use today: POSIX Pthreads, Windows and Java.

4.1. Pthreads

- Pthreads refer to the POSIX standard defining an API for the thread creation and synchronization. This is a ***specification*** for thread behavior, not an ***implementation.***

- Numerous systems implement the Pthreads specification; most are UNIX-type systems, including Linux, Mac OS X, and Solaris.
- Windows doesn't support Pthreads natively, some third-party implementations for Windows are available.
- The C program shown below demonstrates the basic Pthreads API for constructing a multithreaded program that calculates the summation of a non-negative integer in a separate thread.

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

#include <stdlib.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 */

   /* set the default attributes of the thread */
   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 execute 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);
}
```

- In a Pthreads program, separate threads begin execution in a specified function. When this program begins, a single thread of control begins in ***main***(). After some initialization, ***main***() creates a second thread that begins control in the ***runner***() function. Both threads share the global data sum.
- The statement *pthread_t_tid* declares the identifier for the thread we will create. The *pthread_attr_t attr* declaration represents the attributes for the thread.
- A separate thread is created with the *pthread_create()* function call.
- At this point, the program has two threads:
  - the initial thread in *main()* and
  - the summation thread performing the summation operation in the *runner()* function.
- This program follows the fork-join strategy: after creating the summation thread, the parent thread will wait for it to terminate by calling the *pthread_join()* function.
- The summation thread will terminate when it calls the function *pthread_exit()*. Once the summation thread has returned, the parent thread will output the value of the shared data sum.

4.2. Window Threads

- The technique for creating threads using the Windows thread library is similar to the Pthreads technique is several ways.

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 1; i <= Upper; i++)
      Sum += i;
   return 0;
}


int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   Param = atoi(argv[1]);
   /* create the thread */
   ThreadHandle = CreateThread(
      NULL, /* default security attributes */
      0, /* default stack size */
      Summation, /* thread function */
      &Param, /* parameter to thread function */
      0, /* default creation flags */
      &ThreadId); /* returns the thread identifier */

    /* now wait for the thread to finish */
   WaitForSingleObject(ThreadHandle,INFINITE);

   /* close the thread handle */
   CloseHandle(ThreadHandle);

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

- We define the *Summation()* function that is to be performed in a separate thread. This function is passed a pointer to a void, which Windows defines as LPVOID.
- Threads are created in the Windows API using the CreateThread() function, and – just as in Pthreads – a set of attributes for the thread is passed to this function.
- Once the summation thread is created, the parent must wait for it to complete before outputting the value of **Sum,** as the value is set by the summation thread.
- The parent thread wait for the summation thread using the *WaitForSingleObject()* function, which causes the creating thread to block until the summation thread has exited.
- In situations that require waiting for multiple threads to complete, the *WaitForMultipleObjects()* function is used. This function is passed four parameters:
  1. The number of objects to wait for
  2. A pointer to the array of objects
  3. A flag indicating whether all objects have been signaled
  4. A timeout duration

4.3. Java Threads

- Java Threads are managed by the JVM. Java language and its API provide a rich set of features for the creation and management of threads.

- All Java programs comprise at least a single thread of control – even a simple Java program consisting of only a *main()* method runs as a single thread in the JVM.
- There are two techniques for creating threads in a Java program.
  - One approach is to create a new class that is derived from the *Thread* class and to override its *run()* method.
  - An alternative technique is to define a class that implements the *Runnable* interface. The Runnable interface is defined as follows:

```
public interface Runnable
{
    public abstract void run();
}
```

- When a class implements *Runnable*, it must define a *run()* method. The code implementing the *run()* method is what runs a separate thread.

```
class Task implements Runnable
{
   public void run() {
      System.out.println("I am a thread.");
   }
}
```

- Thread creation is performed by creating an object instance of the Thread class and passing the constructor a *Runnable* object. Creating a Thread object does not specifically create the new thread; rather, the *start()* method creates the new thread. Calling the *start()* method for the new object does two things:
  1. It allocates memory and initializes a new thread in the JVM.
  2. It call the *run()* method, making the thread eligible to be run by the JVM.
- When the summation program runs, the JVM creates two threads. The first is the parent thread, which starts execution in the *main()* method. The second thread is created when the *start()* method on the *Thread* object is invoked.
- This child thread begins execution in the *run()* method of the *Summation* class. After outputting the value of the summation, this thread terminates when it exits from its *run*() method.
- The parent thread wait for the summation threads using the *join()* method to finish before proceeding. If the parent must wait for several threads to finish, the *join()* method can be enclosed in a *for* loop.

5. Threading Issues

5.1. The fork() and exec() System Calls

- The *fork()* system call is used to create a separate, duplicate process. The semantics of the *fork()* and *exec()* system calls change in a multithread program.
- Does **fork()** duplicate only the calling thread or all threads?
  - Some UNIX systems have chosen to have two versions of *fork(),* one that duplicates all threads and another duplicates only the thread that invoked the *fork()* system call.
- The *exec()* system call typically works as normal. If a thread invokes the *exec()* system call, the program specified in the parameter to *exec()* will replace the entire process – including all threads.

5.2. Signal Handling

- A signal is used in UNIX systems to notify a process that a particular even has occurred.

- A signal may be received synchronously or asynchronously, depending on the source of and the reason for the event being signaled.
- Examples of synchronous signal include illegal memory access and division by 0. If a running program performs either of these actions, a signal is generated.
- Examples of asynchronous signals include terminating a process with specific key strokes and having a timer expire.
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     - Default
     - user-defined
- Every signal has **default handler** that kernel runs when handling that signal. This default action can overridden by a user-defined signal handler that is called to handle the signal.
- Handling signals in single-threaded programs is straight forward: signals are always delivered to a process.
- Where should a signal be delivered for multi-threaded?
  - 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
- The standard UNIX function for delivering a signal is

  kill(pid_t pid, int signal)

The function specifies the process (pid) to which a particular signal(signal) is to be delivered.

- POSIX Pthreads provide the following function, which allows a signal to be delivered to a specific thread(tid):

  Pthread.kill(pthread_t tid, int signal)

5.3. Thread Cancellation

- Thread cancellation involves terminating a thread before it has completed. For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.
- A thread that is to be canceled is often referred to as the target thread.
- Cancellation of a target thread may occur in two different scenarios:
  - Asynchronous cancellation: One thread immediately terminates the target thread.
  - Deffered cancellation: The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.
- The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads.
- In Pthreads, thread cancelation is initiated using the *pthread_cancel()* function. The identifier of the target thread is passed as a parameter to the function. The following code illustrates creating and then cancelling a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

 . . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state. Pthreads supports three cancellation modes.

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- A thread may set its cancellation state and type using an API.
- Pthreads allow threads to disable or enable cancellation. If thread has cancellation disabled, cancellation remains pending until thread enables it
- The default cancellation type is deferred cancellation. Here, cancellation occurs only when a thread reaches a cancellation point.
- One technique for establishing a cancellation point is to invoke the *pthread_testcancel()* function. If a cancellation request is found to be pending, a function known as **cleanup handler** is invoked. This function allows any resources a thread may have acquired to be released before the thread is terminated.
  - On Linux systems, thread cancellation is handled through signals