Unit-2 Process management

Syllabus:

Process: concept, process control block, process state diagram, inter-process communication

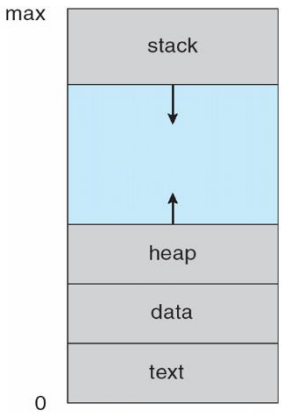
Process scheduling: types, first come first serve, shortest job first, round robin, priority-based scheduling

Threads: multi-core programming, multithreading models, implicit threading, threading issues

A process is a program in execution. A process is the unit of work in a modern time-sharing system. A system therefore consists of a collection of processes

A batch system executes jobs, whereas a time-shared system has user programs or tasks. For example, on a single-user system, a user may be able to run several programs at one time: a word processor, a Web browser, and an e-mail package. all of them are processes.

The Process:

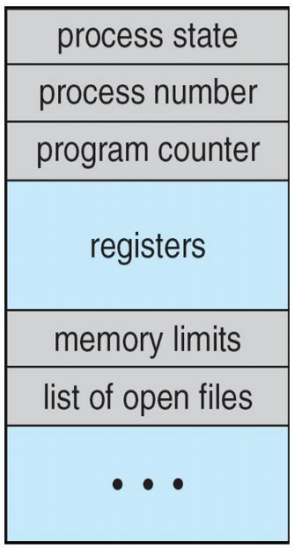


Process in Memory

- A process is a program in execution. It is more than the program code, known as the **text section**.
- It also includes the **current activity**, as represented by the value of the **program counter** and the contents of the processor's registers.

- A process generally also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables),
- It also includes a **data section**, which contains global variables.
- Then a **heap**, which is a memory that is dynamically allocated during process run time.
- **A program is a passive entity**, such as a file containing a list of instructions stored on disk (often called an executable file).
- In contrast, **a process is an active entity**, with a program counter specifying the next instruction to execute and a set of associated resources.
- A program becomes a process when an executable file is loaded into memory.
- Several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary.

Process control block

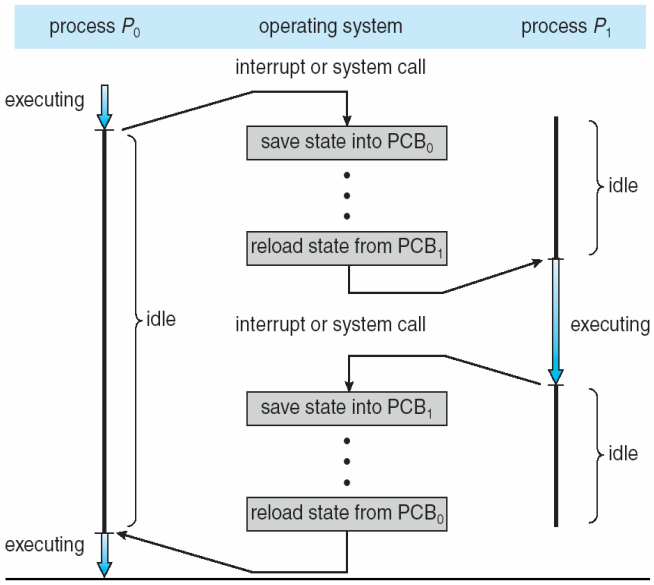


Each process is represented in the operating system by a process control block (PCB)—also called a task control block.
PCB contains many pieces of information associated with a specific process.

- **Process state** –
  The state may be new, ready, running, waiting, halted, and so on.

- **Program counter** – The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers** – The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers
- **CPU scheduling information-** This information includes a process priority, pointers to scheduling queues
- **Memory-management information** -This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system
- **Accounting information** – This information includes the amount of CPU and real-time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information.** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

Context switch



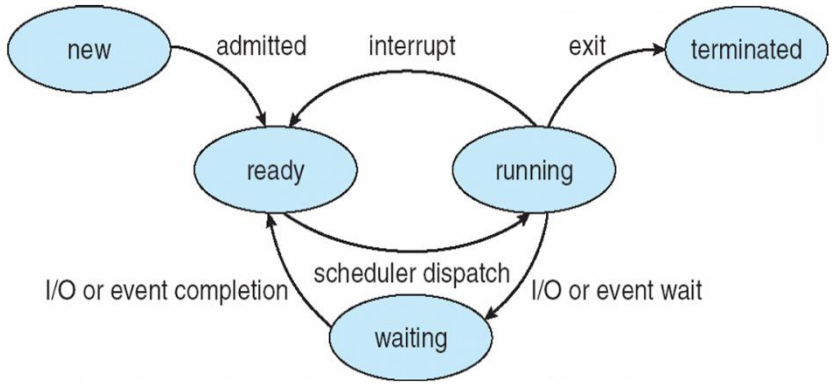
Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **context switch**. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.

Explanation of diagram:

- As we can see in the diagram, initially, the P0 process is running on the CPU to execute its task, and at the same time, another process, P1, is in the ready state.
- If an error or interruption has occurred or the process requires input/output operation, the P0 process switches its state from running to the waiting state.
- Before changing the state of process P0, context switching saves the context of process P0 in the form of registers and the program counter to the **PCB0**.
- After that, it loads the state of the P1 process from the ready state of the **PCB1** to the running state.

Process state diagram

As a process executes, it changes state. It 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 (such as an I/O completion or reception of a signal).
• **Ready**: The process is waiting to be assigned to a processor.
• **Terminated**: The process has finished execution.
It is important to realize that only one process can be running on any processor at any instant, many processes may be ready and waiting for CPU.



Process State Diagram

Explanation of diagram:

- When a new process is initiated, it is said to be in the new state, which means that the process is under creation.
- When the process is **ready** for execution, the **long-term scheduler** transfers it from the new state into the ready state. The process has now entered into the main memory of the system.
- In the **ready** state, the processes are scheduled by the **short-term schedulers**, and a queue is maintained in which the processes are serially sent to the processor.
- After maintaining the queue, the dispatcher then transfers the processes to the running state one by one as per the queue sequence.
- While being in the running state, the process utilizes the processor. But if there is a requirement for any input or output devices in between the processing, then the process is shifted to the **waiting** state.
- When the process is done with the input-output services, instead of directly going back to the running state, it is again sent to the ready state and is then scheduled again for going into the running state.
- This process keeps continuing and when the process completes its execution, it goes into the **termination** state, which means it exits from the main memory of the system.
- The black arrow in the diagram, which goes from the running to waiting for state exists only if the operating system follows pre-emption of processes, i.e. a current running process can be interrupted in between and can be replaced by some other process with high priority. If the OS does not allow pre-emption, then this switch is not allowed.

Inter-process communication:

- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
- 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 is cooperating if it can affect or be affected by the other processes executing in the system. Any process that shares data with other processes is a cooperating process.

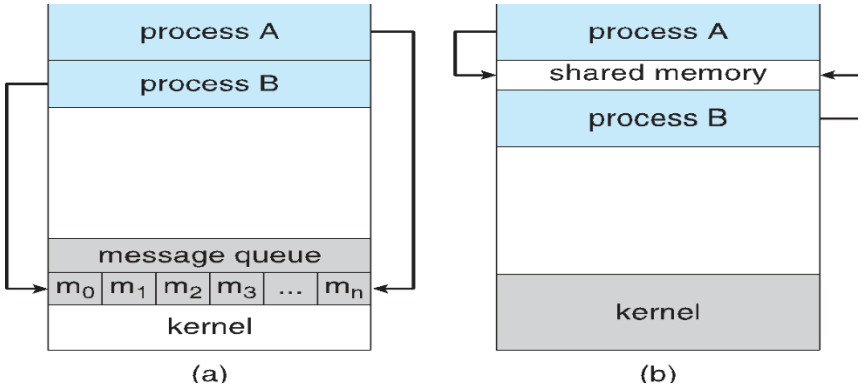
There are several reasons for providing an environment that allows process cooperation:

- Information sharing- Since several users may be interested in the same piece of information (for instance, a shared file), 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 executed in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
- 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 may be editing, listening to music, and compiling in parallel.

There are two fundamental models of inter-process communication: shared memory and message passing.

1. Shared Memory
2. Message Passing

1) **In the shared-memory model**, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.



To illustrate the concept of cooperating processes, let's consider the producer–consumer problem, which is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process.

The two processes share a common space or memory location known as a buffer where the item produced by the Producer is stored and from which the Consumer consumes the item if needed. There are two versions of this problem: the first one is known as the **unbounded buffer** problem in which the Producer can keep on producing items and there is no limit on the size of the buffer, the second one is known as the **bounded buffer** problem in which the Producer can produce up to a certain number of items before it starts waiting for Consumer to consume it.

Let us discuss the **bounded buffer** problem. First, the Producer and the Consumer will share some common memory, then the producer will start producing items. If the total produced item is equal to the size of the buffer, the producer will wait to get it consumed by the Consumer. Similarly, the consumer will first check for the availability of the item. If no item is available, the Consumer will wait for the Producer to produce it. If there are items available, Consumer will consume them.

2) Message Passing

In the message-passing model, communication takes place through messages exchanged between the cooperating processes. In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follows:

- Establish a communication link (if a link already exists, no need to establish it again.)
- Start exchanging messages using basic primitives. We need at least two primitives:
  – send(message, destination) or send(message)
  – receive(message, host) or receive(message)

The message size can be of fixed size or of variable size.

Message Passing through Communication Link.
1. Direct and Indirect Communication link

While implementing the link, some questions need to be kept in mind like:

1. How are links established?
2. Can a link be associated with more than two processes?
3. How many links can there be between every pair of communicating processes?
4. What is the capacity of a link? Is the size of a message that the link can accommodate fixed or variable?
5. Is a link unidirectional or bi-directional?

- A link has some capacity that determines the number of messages that can reside in it temporarily for which every link has a queue associated with it which can be of **zero capacity, bounded capacity, or unbounded capacity**. In zero capacity, the sender waits until the receiver informs the sender that it has received the message.
- In non-zero capacity cases, a process does not know whether a message has been received or not after the send operation.

Direct Communication links are implemented when the processes use a specific process identifier for the communication**.** For example, the print server.

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.

A communication link in this scheme has the following properties:

• A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.

• A link is associated with exactly two processes.

• Between each pair of processes, there exists exactly one link

In-direct Communication is done via a shared mailbox (port), which consists of a queue of messages. The sender keeps the message in the mailbox and the receiver picks them up. A mailbox may be owned either by a process or by the operating system. If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about which process should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears.

With indirect communication, the messages are sent to and received from mailboxes, or ports. 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

In this scheme, a communication link has the following properties:

• A link is established between a pair of processes only if both members of the pair have a shared mailbox.

• A link may be associated with more than two processes.

• Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

2) Synchronization

Communication between processes takes place through calls to send () and receive () primitives. There are different design options for implementing each primitive.

Message passing may be either **blocking or nonblocking**— also known as **synchronous and asynchronous**.

• Blocking send. The sending process is blocked until the message is received by the receiving process or by the mailbox.

• Nonblocking send. The sending process sends the message and resumes operation.
• Blocking receive. The receiver blocks until a message is available.
•Nonblocking receive. The receiver retrieves either a valid message or a null.

Process scheduling:

Various algorithms are used by the Operating System to schedule the processes on the processor efficiently.

1. First Come First Serve

It is the simplest algorithm to implement. The process with the minimal arrival time will get the CPU first. The lesser the arrival time, the sooner the process gets the CPU. It is the non-pre-emptive type of scheduling.

Advantages:

1. It is simple and easy to understand.
2. FCFS guarantees that every process will eventually get a chance to execute, as long as the system has enough resources to handle all the processes.
3. FCFS has low scheduling overhead since it does not involve frequent context switches or complex scheduling decisions.
4. FCFS is well-suited for long-running processes or workloads that do not have strict time constraints.

Disadvantages:

1. The process with less execution time suffers i.e. waiting time is often quite long.
2. Favors CPU-bound process than I/O bound process.
3. Here, the first process will get the CPU first, other processes can get the CPU only after the current process has finished its execution. Now, suppose the first process has a large burst time, and other processes have less burst time, then the processes will have to wait more unnecessarily, this will result in *more average waiting time*, i.e., the Convoy effect.
4. This effect results in lower CPU and device utilization.
5. The FCFS algorithm is particularly troublesome for multiprogramming systems, where each user must get a share of the CPU at regular intervals.

2. Shortest Job First (SJF):

Advantages:

1. Shortest jobs are favored.
2. It is probably optimal; in that it gives the minimum average waiting time for a given set of processes.

Disadvantages:

1. SJF may cause starvation if shorter processes keep coming.

It cannot be implemented at the level of short-term CPU scheduling.

3. Round Robin

In the Round Robin scheduling algorithm, the OS defines a time quantum (slice). All the processes will be executed cyclically. Each of the processes will get the CPU for a small amount of time (called time quantum) and then get back to the ready queue to wait for its next turn. It is a pre-emptive type of scheduling.

Advantages:

1. Every process gets an equal share of the CPU.
2. RR is cyclic in nature, so there is no starvation.

Disadvantages:

1. Setting the quantum too short increases the overhead and lowers the CPU efficiency, but setting it too long may cause a poor response to short processes.
2. The average waiting time under the RR policy is often long.
3. If the time quantum is very high then RR degrades to FCFS.

3. Priority-based scheduling

In this algorithm, the priority will be assigned to each of the processes. The higher the priority, the sooner will the process get the CPU. If the priority of the two processes is the same then they will be scheduled according to their arrival time.

Advantages:

- This provides a good mechanism where the relative importance of each process may be precisely defined.
- PB scheduling allows for the assignment of different priorities to processes based on their importance, urgency, or other criteria.

Disadvantages:

1. If high-priority processes use a lot of CPU time, lower-priority processes may starve and be postponed indefinitely. The situation where a process never gets scheduled to run is called starvation.

The Purpose of a Scheduling Algorithm

1. Maximum CPU utilization
2. Maximum throughput
3. Minimum turnaround time
4. Minimum waiting time
5. Minimum response time

Please refer to class notes on various problems of process scheduling.

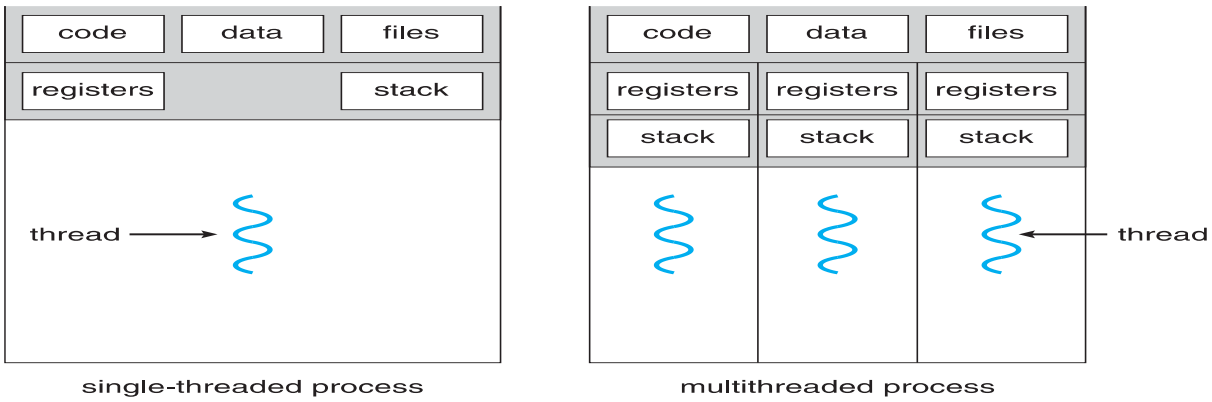
Threads

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. If a process has multiple threads of control, it can perform more than one task at a time. Most software applications that run on modern computers are multithreaded. An application typically is implemented as a separate process with several threads of control.

For example, a web browser might have one thread display images or text while another thread retrieves data from the network.

A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

Most operating-system kernels are now multithreaded. Several threads operate in the kernel, and each thread performs a specific task, such as managing devices, managing memory, or interrupt handling.



single-threaded process

multithreaded process

Benefits of multithreaded programming

1. 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, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces. For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation.

2. Resource sharing.

Processes can only share resources through techniques such as shared memory and message passing. Threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

3. 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. Empirically gauging the difference in overhead can be difficult, but in general, it is significantly more time-consuming to create and manage processes than threads.

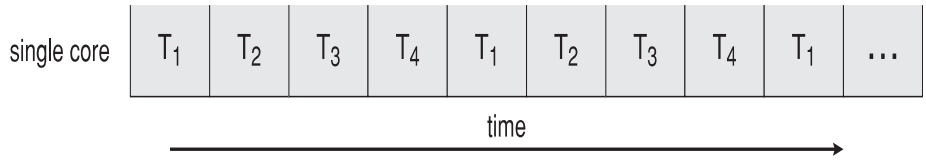
4. Scalability.

The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless of how many are available.

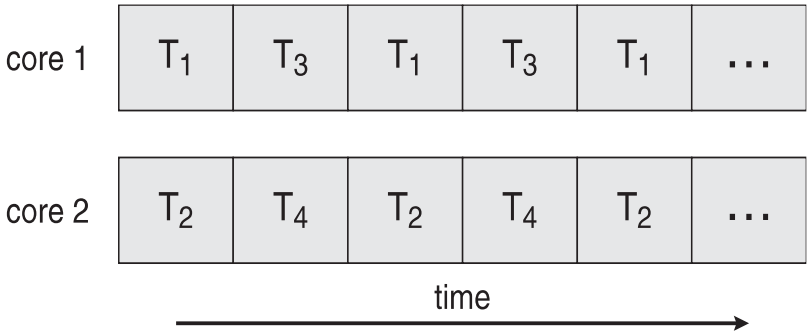
Multicore Programming

In response to the need for more computing performance, single-CPU systems evolved into multi-CPU systems. Each core appears as a separate processor to the operating system. Whether the cores appear across CPU chips or within CPU chips, we call these systems multicore or multiprocessor systems. Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency.

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.



Concurrent execution on a single-core system



Parallelism on a multi-core system:

Types of parallelism -

Data parallelism – distributes subsets of the same data across multiple cores, same operation on each

Task parallelism – distributing threads across cores, each thread performing unique operation

Challenges for programmers due to multicore systems:

1. Identifying tasks

This involves examining applications to find areas that can be divided into separate, concurrent tasks. Ideally, tasks are independent of one another and thus can run in parallel on individual cores.

2. Balance

While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value. In some instances, a certain task may not contribute as much value to the overall process as other tasks. Using a separate execution core to run that task may not be worth the cost.

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.

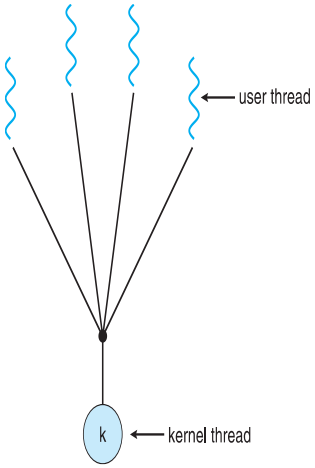
5. Testing and debugging

When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs are inherently more difficult than testing and debugging single-threaded applications.

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

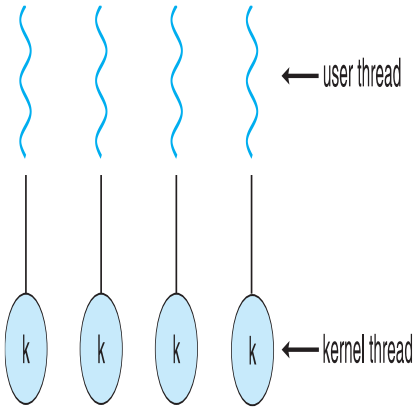
1. Many to One:



The many-to-one model maps many user-level threads to one kernel thread. the entire process will be blocked if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.

Examples: Solaris Green Threads, GNU Portable Threads

2. One to One:

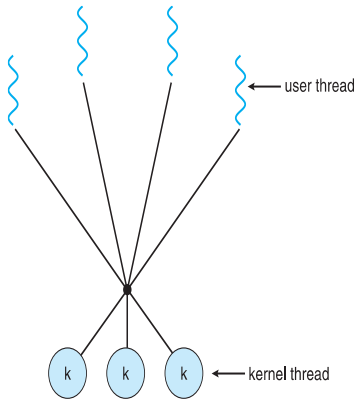


The one-to-one model maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call. It also allows multiple threads to run in parallel on multiprocessors.

The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system.

Ex. Windows, Linux, Solaris 9

3. Many to Many:



The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads. developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

Implicit Threading

One way to address the difficulties discussed under **challenges for programmers due to multicore systems** and to provide better support for the design of multithreaded applications is to transfer the creation and management of threading from application developers to compilers and run-time libraries Implicit threading is used over explicit threading.

It is implemented in 3 popular ways:

1. **Thread Pools**

The general idea behind a thread pool is to create several threads at process startup and place them into a pool, where they sit and wait for work. When a server receives a request, it awakens a thread from this pool—if one is available—and passes it the service request. Once the thread completes its service, it returns to the pool and awaits more work. If the pool contains no available thread, the server waits until one becomes free.

Thread pools offer these benefits:

1. Servicing a request with an existing thread is faster than waiting to create a thread.

2. A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.

3. Separating the task to be performed from the mechanics of creating the task allows us to use different strategies for running the task. For example, the task could be scheduled to execute after a time delay or to execute periodically

2. OpenMP is a set of compiler directives as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in shared-memory environments. OpenMP identifies parallel regions as blocks of code that may run in parallel. Application developers insert compiler directives into their code at parallel regions, and these directives instruct the OpenMP run-time library to execute the region in parallel. It uses directive as #pragma omp parallel

3. Grand Central Dispatch (GCD)—a technology for Apple's Mac OS X and iOS operating systems—is a combination of extensions to the C language, an API, and a run-time library that allows application developers to identify sections of code to run in parallel.

GCD identifies extensions to the C and C++ languages known as blocks. A block is simply a self-contained unit of work. It is specified by a caret ˆ inserted in front of a pair of braces { }. A simple example of a block is shown below: ˆ {printf ("I am a block");}

GCD identifies two types of dispatch queues: serial and concurrent. Blocks placed on a serial queue are removed in FIFO order. Once a block has been removed from the queue, it must complete execution before another block is removed. Blocks placed on a concurrent queue are also removed in FIFO order, but several blocks may be removed at a time, thus allowing multiple blocks to execute in parallel.

There are three system-wide concurrent dispatch queues, and they are distinguished according to priority: low, default, and high. Priorities represent an approximation of the relative importance of blocks. Quite simply, blocks with a higher priority should be placed on the high-priority dispatch queue.

Threading Issues

- **Semantics of fork () and exec () system calls**

  The problem lies in the semantics of fork () and exec () system calls when they are used in multithreaded applications. fork () creates 2 child processes of 1 parent process. Which further duplicates the functionality of the parent process. So there are chances of creating duplicacy in the operation. Even if exec () is called after fork (), it may create different execution paths for the same duplicated thread.

- **Signal handling**

  Signals are used in UNIX systems to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously. They follow the following events.

  1. A signal is generated by the occurrence of a particular event.

  2. The signal is delivered to a process.

  3. Once delivered, the signal must be handled.

  Synchronous signals are delivered to the same process that performed the operation that caused the signal.

  When a signal is generated by an event external to a running process, that process receives the signal asynchronously.

  A signal may be handled by one of two possible handlers:

1. A default signal handler 2. A user-defined signal handler

Every signal has a default signal handler that the kernel runs when handling that signal. This default action can be overridden by a user-defined signal handler that is called to handle the signal.

Delivering signals is more complicated in multithreaded programs, where a process may have several threads. Where, then, should a signal be delivered? In general, the following options exist:

1. Deliver the signal to the thread to which the signal applies.

2. Deliver the signal to every thread in the process.

3. Deliver the signal to certain threads in the process.

4. Assign a specific thread to receive all signals for the process.

- **Thread cancellation of the target thread**

Thread cancellation involves terminating a thread before it has been completed. A thread that is to be cancelled is often referred to as the target thread. Cancellation of a target thread may occur in two different scenarios:

1. Asynchronous cancellation. One thread immediately terminates the target thread.

2. Deferred cancellation. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

In Pthreads, thread cancellation 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 canceling— a thread:

```
pthread t tid;

/* create the thread */

pthread create (&tid, 0, worker, NULL); ...

/* cancel the thread */

pthread cancel(tid);
```

pthread supports 3 cancellation modes. Each mode is defined as a state and a type, as illustrated in the table below.

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

As the table illustrates, Pthreads allows threads to disable or enable cancellation. A thread cannot be cancelled if cancellation is disabled. However, cancellation requests remain pending, so the thread can later enable cancellation and respond to the request. The default cancellation type is deferred cancellation.

- **Thread-local storage**

  Threads share the data of the process to which it belongs to. This data sharing provides one of the benefits of multithreaded programming. However, in some circumstances, each thread might need its own copy of certain data. Such data is called **thread-local storage (or TLS)**.

  TLS is similar to static data. The only difference is that TLS data are unique to each thread. Most threads libraries- including Windows and Pthreads- provide some form of support for thread-local storage.

- **Scheduler Activations**

  Many systems implementing either the many-to-many or the two-level model place an intermediate data structure between the user and kernel threads. This data structure—is typically known as a lightweight process, or LWP.

  To the user-thread library, the LWP appears to be a virtual processor on which the application can schedule a user thread to run. Each LWP is attached to a kernel thread, and it is kernel threads that the operating system schedules to run on physical processors.

  If a kernel thread blocks (such as while waiting for an I/O operation to complete), the LWP blocks as well.

  One scheme for communication between the user-thread library and the kernel is known as scheduler activation. It works as follows: The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor. Furthermore, the kernel must inform an application about certain events. This procedure is known as an upcall. Upcalls are handled by the thread library with an upcall handler, and upcall handlers must run on a virtual processor.

****