**Introduction :-**

The discussion so far has presented the concept of a process as embodying two characteristics:

1. **Resource ownership**: A process includes a virtual address space to hold the process image; recall from Chapter 3 that the process image is the collection of program, data, stack, and attributes defined in the process control block. From time to time, a process may be allocated control or ownership of resources, such as main memory, I/O channels, I/O devices, and files. The OS performs a protection function to prevent unwanted interference between processes with respect to resources.

2. **Scheduling/execution**: The execution of a process follows an execution path (trace) through one or more programs (e.g., Figure 1.5 and Figure 1.26). This execution may be interleaved with that of other processes. Thus, a process has an execution state (Running, Ready, etc.) and a dispatching priority and is the entity that is scheduled and dispatched by the OS. Some thought should convince the reader that these two characteristics are independent and could be treated independently by the OS. This is done in a number of operating systems, particularly recently developed systems. To distinguish the two characteristics, the unit of dispatching is usually referred to as a thread or lightweight process, while the unit of resource ownership is usually still referred to as a process or task.

**Multithreading :-**

Multithreading refers to the ability of an OS to support multiple, concurrent paths of execution within a single process. The traditional approach of a single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach. The two arrangements shown in the left half of Figure 4.1 are single-threaded approaches. MS-DOS is an example of an OS that supports a single user process and a single thread. Other operating systems, such as some variants of UNIX, support multiple user processes but only support one thread per process. The right half of Figure 4.1 depicts multithreaded approaches. A Java run-time environment is an example of a system of one process with multiple threads. Of interest in this section is the use of multiple processes, each of which supports multiple threads. This approach is taken in Windows, Solaris, and many modern versions of UNIX, among others.

In a multithreaded environment, a process is defined as the unit of resource allocation and a unit of protection. The following are associated with processes:

- A virtual address space that holds the process image.
- Protected access to processors, other processes (for interprocess communication), files, and I/O resources (devices and channels).

Within a process, there may be one or more threads, each with the following:

- A thread execution state (Running, Ready, etc.).
- A saved thread context when not running; one way to view a thread is as an independent program counter operating within a process.
- An execution stack.
- Some per-thread static storage for local variables.
- Access to the memory and resources of its process, shared with all other threads in that process.

---

**Figure 4.1** Threads and Processes

**Figure 4.2** Single Threaded and Multithreaded Process Models
Figure 4.2 illustrates the distinction between threads and processes from the point of view of process management. In a single-threaded process model (i.e., there is no distinct concept of thread), the representation of a process includes its process control block and user address space, as well as user and kernel stacks to manage the call/return behavior of the execution of the process. While the process is running, it controls the processor registers. The contents of these registers are saved when the process is not running. In a multithreaded environment, there is still a single process control block and user address space associated with the process, but now there are separate stacks for each thread, as well as a separate control block for each thread containing register values, priority, and other thread-related state information.

Thus, all of the threads of a process share the state and resources of that process. They reside in the same address space and have access to the same data. When one thread alters an item of data in memory, other threads see the results if and when they access that item. If one thread opens a file with read privileges, other threads in the same process can also read from that file.

- The key benefits of threads derive from the performance implications:
  1. It takes far less time to create a new thread in an existing process than to create a brand-new process. Studies done by the Mach developers show that thread creation is ten times faster than process creation in UNIX.
  2. It takes less time to terminate a thread than a process.
  3. It takes less time to switch between two threads within the same process than to switch between processes.
  4. Threads enhance efficiency in communication between different executing programs. In most operating systems, communication between independent processes requires the intervention of the kernel to provide protection and the mechanisms needed for communication. However, because threads within the same process share memory and files, they can communicate with each other without invoking the kernel.

Thus, if there is an application or function that should be implemented as a set of related units of execution, it is far more efficient to do so as a collection of threads rather than a collection of separate processes.

- Thread Functionality

Like processes, threads have execution states and may synchronize with one another. We look at these two aspects of thread functionality in turn.

Thread States: As with processes, the key states for a thread are Running, Ready, and Blocked. Generally, it does not make sense to associate suspend states with threads because such states are process-level concepts. In particular, if a process is swapped out, all of its threads are necessarily swapped out because they all share the address space of the process.
There are four basic thread operations associated with a change in thread state:

- **Spawn**: Typically, when a new process is spawned, a thread for that process is also spawned. Subsequently, a thread within a process may spawn another thread within the same process, providing an instruction pointer and arguments for the new thread. The new thread is provided with its own register context and stack space and placed on the ready queue.

- **Block**: When a thread needs to wait for an event, it will block (saving its user registers, program counter, and stack pointers). The processor may now turn to the execution of another ready thread in the same or a different process.

- **Unblock**: When the event for which a thread is blocked occurs, the thread is moved to the Ready queue.

- **Finish**: When a thread completes, its register context and stacks are deallocated.

A significant issue is whether the blocking of a thread results in the blocking of the entire process. In other words, if one thread in a process is blocked, does this prevent the running of any other thread in the same process even if that other thread is in a ready state? Clearly, some of the flexibility and power of threads is lost if the one blocked thread blocks an entire process.

![Diagram of thread operations](image.png)

**Figure 4.3** Remote Procedure Call (RPC) Using Threads

We return to this issue subsequently in our discussion of user-level versus kernel level threads, but for now let us consider the performance benefits of threads that do not
block an entire process. Figure 4.3 shows a program that performs two remote procedure calls (RPCs) to two different hosts to obtain a combined result. In a single-threaded program, the results are obtained in sequence, so that the program has to wait for a response from each server in turn. Rewriting the program to use a separate thread for each RPC results in a substantial speedup. Note that if this program operates on a uniprocessor, the requests must be generated sequentially and the results processed in sequence; however, the program waits concurrently for the two replies. On a uniprocessor, multiprogramming enables the interleaving of multiple threads within multiple processes. In the example of Figure 4.4, three threads in two processes are interleaved on the processor. Execution passes from one thread to another either when the currently running thread is blocked or its time slice is exhausted.

Thread Synchronization: All of the threads of a process share the same address space and other resources, such as open files. Any alteration of a resource by one thread affects the environment of the other threads in the same process. It is therefore necessary to synchronize the activities of the various threads so that they do not interfere with each other or corrupt data structures. For example, if two threads each try to add an element to a doubly linked list at the same time, one element may be lost or the list may end up malformed. The issues raised and the techniques used in the synchronization of threads are, in general, the same as for the synchronization of processes.

**Figure 4.4 Multithreading Example on a Uniprocessor**

> An example—Adobe PageMaker4

An example of the use of threads is the Adobe PageMaker application running under a shared system. PageMaker is a writing, design, and production tool for desktop publishing. The thread structure for PageMaker used in the operating system OS/2, shown in Figure 4.5 was chosen to optimize the responsiveness of the application.
(similar thread structures would be found on other operating systems). Three threads are always active: an event-handling thread, a screen-redraw thread, and a service thread.

<table>
<thead>
<tr>
<th>Service Thread</th>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Import</td>
</tr>
<tr>
<td></td>
<td>Auto Flow</td>
</tr>
<tr>
<td></td>
<td>Printing</td>
</tr>
<tr>
<td>Event Handling Thread</td>
<td>Screen – Redraw Thread</td>
</tr>
</tbody>
</table>

Generally, OS/2 is less responsive in managing windows if any input message requires too much processing. The OS/2 guidelines state that no message should require more than 0.1 s processing time. For example, calling a subroutine to print a page while processing a print command would prevent the system from dispatching any further message to any applications, slowing performance. To meet this criterion, time-consuming user operations in PageMaker—printing, importing data, and flowing text—are performed by a service thread. Program initialization is also largely performed by the service thread, which absorbs the idle time while the user invokes the dialogue to create a new document or open an existing document. A separate thread waits on new event messages.

Synchronizing the service thread and event-handling thread is complicated because a user may continue to type or move the mouse, which activates the event handling thread, while the service thread is still busy. If this conflict occurs, PageMaker filters these messages and accepts only certain basic ones, such as window resize. The service thread sends a message to the event-handling thread to indicate completion of its task. Until this occurs, user activity in PageMaker is restricted. The program indicates this by disabling menu items and displaying a “busy” cursor. The user is free to switch to other applications, and when the busy cursor is moved to another window, it will change to the appropriate cursor for that application.

The screen redraw function is handled by a separate thread. This is done for two reasons:
1. PageMaker does not limit the number of objects appearing on a page; thus, processing a redraw request can easily exceed the guideline of 0.1 second.
2. Using a separate thread allows the user to abort drawing. In this case, when the user rescales a page, the redraw can proceed immediately. The program is less responsive if it completes an outdated display before commencing with a display at the new scale.

Dynamic scrolling—redrawing the screen as the user drags the scroll indicator—is also possible. The event-handling thread monitors the scroll bar and redraws the margin rulers (which redraw quickly and give immediate positional
feedback to the user). Meanwhile, the screen-redraw thread constantly tries to redraw the page and catch up.

Implementing dynamic redraw without the use of multiple threads places a greater burden on the application to poll for messages at various points. Multithreading allows concurrent activities to be separated more naturally in the code.

**User-Level and Kernel-Level Threads**

There are two broad categories of thread implementation: user-level threads (ULTs) and kernel-level threads (KLTs). The latter are also referred to in the literature as kernel-supported threads or lightweight processes.

![Diagram](image)

**User-Level Threads** In a pure ULT facility, all of the work of thread management is done by the application and the kernel is not aware of the existence of threads. Figure 4.6a illustrates the pure ULT approach. Any application can be programmed to be multithreaded by using a threads library, which is a package of routines for ULT management. The threads library contains code for creating and destroying threads, for passing messages and data between threads, for scheduling thread execution, and for saving and restoring thread contexts.

By default, an application begins with a single thread and begins running in that thread. This application and its thread are allocated to a single process managed by the kernel. At any time that the application is running (the process is in the Running state), the application may spawn a new thread to run within the same process. Spawning is done by invoking the spawn utility in the threads library. Control is passed to that utility by a procedure call. The threads library creates a data structure for the new thread and then passes control to one of the threads within this process that is in the Ready state, using some scheduling algorithm. When control is passed to the library, the context of the
current thread is saved, and when control is passed from the library to a thread, the context of that thread is restored. The context essentially consists of the contents of user registers, the program counter, and stack pointers.

All of the activity described in the preceding paragraph takes place in user space and within a single process. The kernel is unaware of this activity. The kernel continues to schedule the process as a unit and assigns a single execution state (Ready, Running, Blocked, etc.) to that process. The following examples should clarify the relationship between thread scheduling and process scheduling. Suppose that process B is executing in its thread 2; the states of the process and two ULTs that are part of the process are shown in Figure 4.7a. Each of the following is a possible occurrence:

1. The application executing in thread 2 makes a system call that blocks B. For example, an I/O call is made. This causes control to transfer to the kernel. The kernel invokes the I/O action, places process B in the Blocked state, and switches to another process. Meanwhile, according to the data structure maintained by the threads library, thread 2 of process B is still in the Running state. It is important to note that thread 2 is not actually running in the sense of being executed on a processor; but it is perceived as being in the Running state by the threads library. The corresponding state diagrams are shown in Figure 4.7b.

2. A clock interrupt passes control to the kernel and the kernel determines that the currently running process (B) has exhausted its time slice. The kernel places process
B in the Ready state and switches to another process. Meanwhile, according to the data structure maintained by the threads library, thread 2 of process B is still in the Running state. The corresponding state diagrams are shown in Figure 4.7c.

3. Thread 2 has reached a point where it needs some action performed by thread 1 of process B. Thread 2 enters a Blocked state and thread 1 transitions from Ready to Running. The process itself remains in the Running state. The corresponding state diagrams are shown in Figure 4.7d.

In cases 1 and 2 (Figures 4.7b and 4.7c), when the kernel switches control back to process B, execution resumes in thread 2. Also note that a process can be interrupted, either by exhausting its time slice or by being preempted by a higher-priority process, while it is executing code in the threads library. Thus, a process may be in the midst of a thread switch from one thread to another when interrupted. When that process is resumed, execution continues within the threads library, which completes the thread switch and transfers control to another thread within that process.

**Advantages to the use of ULTs instead of KLTs, including the following:**

1. **Thread switching does not require kernel mode privileges** because all of the thread management data structures are within the user address space of a single process. Therefore, the process does not switch to the kernel mode to do thread management. This saves the overhead of two mode switches (user to kernel; kernel back to user).

2. **Scheduling can be application specific.** One application may benefit most from a simple round-robin scheduling algorithm, while another might benefit from a priority-based scheduling algorithm. The scheduling algorithm can be tailored to the application without disturbing the underlying OS scheduler.

3. **ULTs can run on any OS.** No changes are required to the underlying kernel to support ULTs. The threads library is a set of application-level functions shared by all applications.

**Distinct disadvantages of ULTs compared to KLTs:**

1. **In a typical OS, many system calls are blocking.** As a result, when a ULT executes a system call, not only is that thread blocked, but also all of the threads within the process are blocked.

2. **In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing.** A kernel assigns one process to only one processor at a time. Therefore, only a single thread within a process can execute at a time. In effect, we have application-level multiprogramming within a single process. While this multiprogramming can result in a significant speedup of the application, there are applications that would benefit from the ability to execute portions of code simultaneously.

There are ways to work around these two problems. For example, both problems can be overcome by writing an application as multiple processes rather than multiple threads. But this approach eliminates the main advantage of threads: each switch...
becomes a process switch rather than a thread switch, resulting in much greater overhead.

Another way to overcome the problem of blocking threads is to use a technique referred to as jacketing. The purpose of jacketing is to convert a blocking system call into a nonblocking system call. For example, instead of directly calling a system I/O routine, a thread calls an application-level I/O jacket routine. Within this jacket routine is code that checks to determine if the I/O device is busy. If it is, the thread enters the Blocked state and passes control (through the threads library) to another thread. When this thread later is given control again, the jacket routine checks the I/O device again.

**Kernel-Level Threads:** In a pure KLT facility, all of the work of thread management is done by the kernel. There is no thread management code in the application level, simply an application programming interface (API) to the kernel thread facility. Windows is an example of this approach. Figure 4.6b depicts the pure KLT approach. The kernel maintains context information for the process as a whole and for individual threads within the process.

Scheduling by the kernel is done on a thread basis. This approach overcomes the two principal drawbacks of the ULT approach. First, the kernel can simultaneously schedule multiple threads from the same process on multiple processors. Second, if one thread in a process is blocked, the kernel can schedule another thread of the same process. Another advantage of the KLT approach is that kernel routines themselves can be multithreaded.

**The principal disadvantage of the KLT approach compared to the ULT approach is that the transfer of control from one thread to another within the same process requires a mode switch to the kernel.** To illustrate the differences, Table 4.1 shows the results of measurements taken on a uniprocessor VAX computer running a UNIX-like OS. The two benchmarks are as follows: Null Fork, the time to create, schedule, execute, and complete a process/thread that invokes the null procedure (i.e., the overhead of forking a process/thread); and Signal-Wait, the time for a process/thread to signal a waiting process/thread and then wait on a condition (i.e., the overhead of synchronizing two processes/threads together). We see that there is an order of magnitude or more of difference between ULTs and KLTs and similarly between KLTs and processes.

<table>
<thead>
<tr>
<th>Operation</th>
<th>User-Level Thread</th>
<th>Kernel-Level Thread</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null Fork</td>
<td>34</td>
<td>948</td>
<td>11,300</td>
</tr>
<tr>
<td>Signal Wait</td>
<td>37</td>
<td>441</td>
<td>1,840</td>
</tr>
</tbody>
</table>

Thus, on the face of it, while there is a significant speedup by using KLT multithreading compared to single-threaded processes, there is an additional significant speedup by using ULTs. However, whether or not the additional speedup is realized depends on the nature of the applications involved. If most of the thread
switches in an application require kernel mode access, then a ULT-based scheme may not perform much better than a KLT-based scheme.

**Combined Approaches:** Some operating systems provide a combined ULT/KLT facility (Figure 4.6c). In a combined system, thread creation is done completely in user space, as is the bulk of the scheduling and synchronization of threads within an application. The multiple ULTs from a single application are mapped onto some (smaller or equal) number of KLTs. The programmer may adjust the number of KLTs for a particular application and processor to achieve the best overall results. In a combined approach, multiple threads within the same application can run in parallel on multiple processors, and a blocking system call need not block the entire process. If properly designed, this approach should combine the advantages of the pure ULT and KLT approaches while minimizing the disadvantages.

Solaris is a good example of an OS using this combined approach. The current Solaris version limits the ULT/KLT relationship to be one-to-one.

<table>
<thead>
<tr>
<th>User-Level Thread</th>
<th>Kernel-Level Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel is not aware of thread mechanism.</td>
<td>Kernel is aware of thread mechanism.</td>
</tr>
<tr>
<td>Thread management is performed at User-Level</td>
<td>Thread management is performed at Kernel-Level</td>
</tr>
<tr>
<td>Thread Library is used for Thread Management.</td>
<td>API (Application Programming Interface) is used for Thread Management.</td>
</tr>
<tr>
<td>No mode switching is required.</td>
<td>Mode switching is required.</td>
</tr>
<tr>
<td>ULT is Faster.</td>
<td>KLT is Fastest.</td>
</tr>
<tr>
<td>Multiprocessing is not supported.</td>
<td>Multiprocessing is supported.</td>
</tr>
<tr>
<td>Synchronization of thread is performed at application level.</td>
<td>Synchronization of thread is performed at Operating System level.</td>
</tr>
</tbody>
</table>