Threads

Thread Functionality

A thread is a fundamental element of the central processing unit (CPU) utilization. It is generally composed of thread identification, a program counter, a set of registers, and a stack. Threads that belong in the same process share the same code section, data section, and other operating system (OS) resources. A traditional process has a single thread of control, and if a process has multiple threads of control, it can perform more than one task at a time (Silberschatz, Galvin, & Gagne, 2018). A thread should at least contain the following attributes:

- Thread execution state
- A saved thread context when not running
- Some per-thread static storage for local variables
- Access to the memory and resource of the process
- An execution stack

As an example, the list of attributes below characterizes a Windows thread object (Stallings, 2018):

Thread ID	A unique value that identifies a thread when it calls a server
Thread context	A set of register values and other volatile data that defines the execution state of a thread
Dynamic priority	The thread's execution priority at any given moment
Base priority	The lower limit of the thread's dynamic priority
Thread processor affinity	The set of processors on which the thread can run.
Thread execution time	The cumulative amount of time a thread has executed in user mode and in kernel mode
Alert status	A flag that indicates whether a waiting thread may execute an asynchronous procedure call
Suspension count	The number of times the thread's execution has been suspended without being resumed
Impersonation token	A temporary access token allowing a thread to perform operations on behalf of another process
Termination port	An inter-process communication channel to which the process manager sends a message when the thread terminates
Thread exit status	The reason for a thread's termination

The object-oriented structure of Windows OS facilitates the development of a general-purpose process facility that encompasses threads. Each process is represented by an object that includes specific attributes and encapsulates a number of actions or services that it may perform. A Windows process must also contain at least one (1) thread to execute. This thread may then create other threads. Some attributes of a thread object resemble those of a process, and in this case, the thread attribute value is actually derived from the process attribute value.

Note that one of the attributes of a thread object is context, which contains the values of the processor registers when the thread last ran. This information enables threads to be suspended and resumed. Furthermore, in Windows OS, it is possible to alter the behavior of a thread by altering its context while it is suspended.

To differentiate the resource ownership and program execution (scheduling) characteristics of a process, particularly for recently developed systems, the unit of resource ownership is usually referred to as *process* or *task*, while the unit of dispatching processes is usually referred to as *threads* or *lightweight processes*. The following process and thread arrangement or relationship may occur (Stallings, 2018):

- **One process: One thread (1:1)** – Each thread of execution is a unique process with its own address space and resources. The MS-DOS and the Classic UNIX are examples of OS that support this kind of relationship.
- **Multiple processes: One thread per process (M:1)** – A thread may migrate from one (1) process environment to another. This allows a thread to be easily moved among distinct systems. Some variants of the UNIX OS support this kind of arrangement.
- **One process: Multiple threads (1:M)** – A process defines a specific address space and a dynamic resource ownership. Thus, multiple threads can be created and executed within the process. The Java runtime environment is an example of a system that encompasses this arrangement.
- **Multiple processes: Multiple threads per processes (M:M)** – This has the combined attribute of the M:1 and 1:M arrangement.

The thread construct is also useful for systems with a single processor, since it simplifies the structure of a program that is logically performing several different functions. Threads are applied in a single-user multiprocessing system for the enhancement of process execution, for implementing a modular

program structure, for asynchronous processing, and for other foreground and background work enhancement.

Thread State. The key states for a thread are *Running*, *Ready*, and *Blocked*. If a process is swapped out, noted that all of its threads are automatically swapped out because they all share the address space of the process. The following are the four (4) basic thread operations associated with a change in thread state (Stallings, 2018):

1. **Spawn** – This operation typically transpires whenever a new process is created, since a thread for that specific process is also formed.
2. **Block** – This happens when a thread needs to wait for a particular event. In this operation, all the necessary information are automatically saved for the thread's execution resumption.
3. **Unblocked** – This operation moves a blocked thread into the ready queue for the continuation of its execution.
4. **Finish** – This happens whenever a thread completes its entire execution. Its register settings and stacks are deallocated.

As an example, the thread states used by Windows are illustrated below:

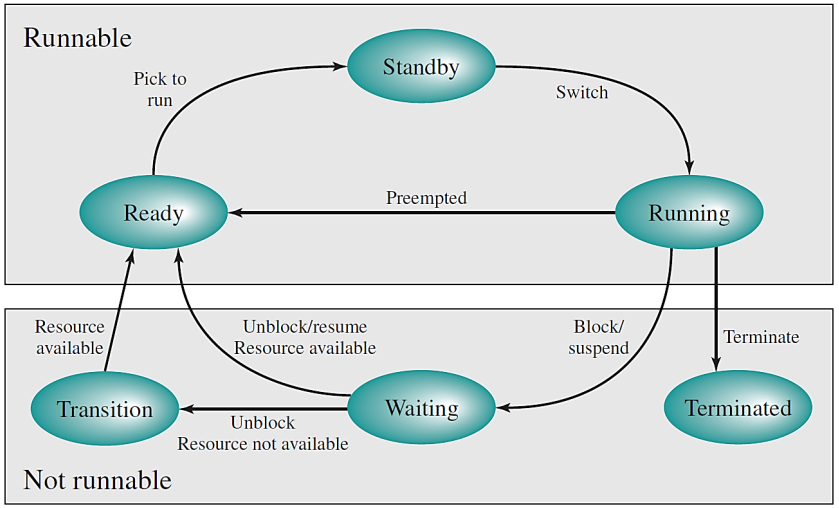

Figure 1. The Windows thread states.
Source: Operating Systems: Internal and Design Principles (9th ed.), 2018 p. 200

Thread Synchronization. All threads of a process share the same address space and resources. Hence, any alteration of a resource by one thread affects the environments of the other threads within the same process. It is therefore necessary to synchronize the activities of various threads in the process to eliminate interference and maintain the data structure of the process.

Thread Library. Any application can be programmed to be multithreaded with the use of thread libraries. A thread library technically provides programmers with an application programming interface (API) for creating and managing threads. It may contain codes for creating and destroying threads; for passing messages and data between threads; for scheduling thread execution; and for saving and restoring thread context/information.

Types of Threads (Stallings, 2018)

User-Level Threads

In implementing user-level threads, all the thread management is done by the application. Tthe system kernel is not aware of the existence of the threads and continues to schedule the process as a unit. Below are some advantages of implementing user-level threads:

- Thread switching does not require kernel-mode privileges since the overall thread data management structure is within the user address space of a single process. Thus, saves the overhead of two (2) mode switches.
- Scheduling can be application-specific. The scheduling algorithm can be tailored for the application without disturbing the operating system scheduler.
- User-level threads can run on any operating system. There are no changes required to the underlying kernel in order to support user-level threads.

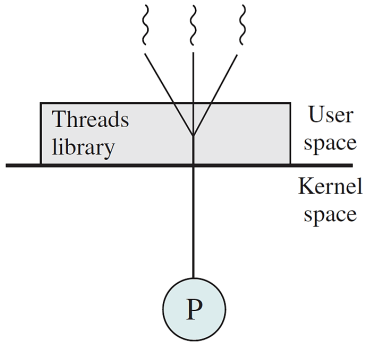

Figure 2. A pure user-level thread.
Source: Operating Systems: Internal and Design Principles (9th ed.), 2018 p. 184

Kernel-Level Threads

In implementing kernel-level threads, all thread management is performed by the operating system's kernel. There are no thread management code at the application level, only an application programming interface (API) to the kernel thread section. The kernel maintains the context information for the process as a whole and for the individual threads within the process. In addition, scheduling by the kernel is done on a thread basis. Below are some of the advantages of implementing kernel-level threads:

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors.
- If one thread in a process is blocked, the kernel can schedule another thread from the same process.
- The kernel routines themselves can be multithreaded.

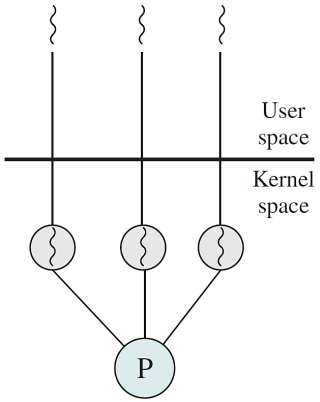

Figure 3. A pure kernel-level thread.
Source: Operating Systems: Internal and Design Principles (9th ed.), 2018 p. 184

Combined User-Level and Kernel-Level Approach

Some operating systems provide a combined user-level thread and kernel-level thread facility. Thread creation is performed completely in the user space, so is the scheduling and synchronization of threads within an application. Multiple user-level threads from a single application are then mapped onto specific kernel-level threads.

In this combined approach, multiple threads within the same application can run in parallel on multiple processors, and a blocking system call does not block the entire process. If properly designed, the combined approach should also combine the advantages of the pure user-level threads and the kernel-level threads. The Solaris is a good example of an operating system that implements the combined approach.

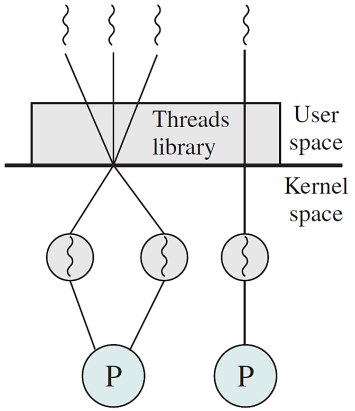

Figure 4. A combined user-level thread and kernel-level thread.
Source: Operating Systems: Internal and Design Principles (9th ed.), 2018 p. 184

Multithreading

Multithreading pertains to the ability of an operating system (OS) to support multiple, concurrent paths of execution within a single process. The use of *multicore systems* to provision applications with multiple threads affects the application design and performance. The potential performance benefits of a multicore system may also be subjected to the ability of a multithreaded application to effectively exploit the parallel resources available to the application. Moreover, an object-oriented multithreaded process is an efficient means of implementing a server application (Stallings, 2018).

Below are some of the general characteristics of multithreading (Gregg, 2021):

- The memory overhead in multithreading is small. It only requires an extra stack, register space, and space for thread-local data.
- The central processing unit (CPU) overhead is small since it uses application programming interface (API) calls.
- The communication within the process and with other processes are faster.
- The crash resilience in implementing multithreading is low. Any bug can crash the entire application
- The memory usage is monitored via a system allocator, which may incur some CPU contention from multiple threads, and fragmentation before the memory is reused.

The following are some examples of multithreaded applications:

- An application that creates thumbnails of photos from a collection of images
- A web browser that displays images or texts while retrieving data from the network
- A word processor that displays texts and responds to keystrokes or mouse clicks while performing spelling and grammar check

The benefits of multithreaded programming can be categorized as follows (Silberschatz, Galvin, & Gagne, 2018):

- **Responsiveness**: Incorporating the multithreading concept in an interactive application allows a program to continue running even if some part of it is blocked or performing a lengthy operation. Thus, increasing the responsiveness of the application to the user.
- **Resource Sharing**: Generally, threads share the memory and resources of the process to which they belong. This allows applications to have several different threads of activities within the same address space.
- **Economical**: Allocating memory and resources for process creation is inefficient, but thread creation consumes less time and memory, making it economical.
- **Scalability**: The benefit of multithreading is greater when dealing with multiprocessor architecture, where threads can run in parallel on different processing cores.

Windows is a good example of an OS that supports multithreading. Threads in different processes may execute concurrently or appear to run at the same time. Additionally, multiple threads within the same process may be allocated to different processors and execute simultaneously.

In a Windows OS, threads within the same process can exchange information through their common address space and access shared resources of the process. Threads in different processes can exchange information through some shared memory that has been set up between two (2) processes

References:
Gregg, B. (2021). *System performance: Enterprise and cloud (2nd ed.)*. Pearson Education, Inc.
Silberschatz, A., Galvin, P. & Gagne, G. (2018). *Operating systems concepts (10th ed.)*. John Wiley & Sons, Inc.
Stallings, W. (2018). *Operating systems: Internal and design principles (9th ed.)*. Pearson Education Limited