CMP 426 & 697 (Operating Systems)
Additional Reference on
Multi-threading models and implementations in various Operating Systems

References

- W. Stalling, *Operating Systems: Internals and Design Principles*, 7th edition, Prentice-Hall, 2012.
- F. Zabatta and K. Ying, “Thread Performance Comparison: Windows NT and Solaris”, in the Proceedings of the 2nd USENIX Windows NT Symposium, pp. 57-pp. 66
- *NPTL Optimization*, Sam-Sung Embedded Systems Group, 2010.
- Wikis about multithreading and POSIX (Portable Operating System Interface for Unix) thread specification
- https://en.wikipedia.org/wiki/Native_POSIX_Thread_Library

1. Introduction

1.1. Multithreading

Multithreading refers to the ability of an OS to support multiple, concurrent paths of execution within a single process. To distinguish processes from threads, the unit of scheduling and dispatching is usually referred to as a thread (or termed lightweight process), while the unit of resource ownership is usually referred to as a process (or task)

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 (process context):

- A virtual address space that holds the process image (code, data, stack, heap)
- Protected access to processors and possibly to other processes (for inter-process communications), files, and I/O resources (devices and channels such as DMAs)

Within a process, there may be one or more threads, each with the following (thread context)
- 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 (global data)

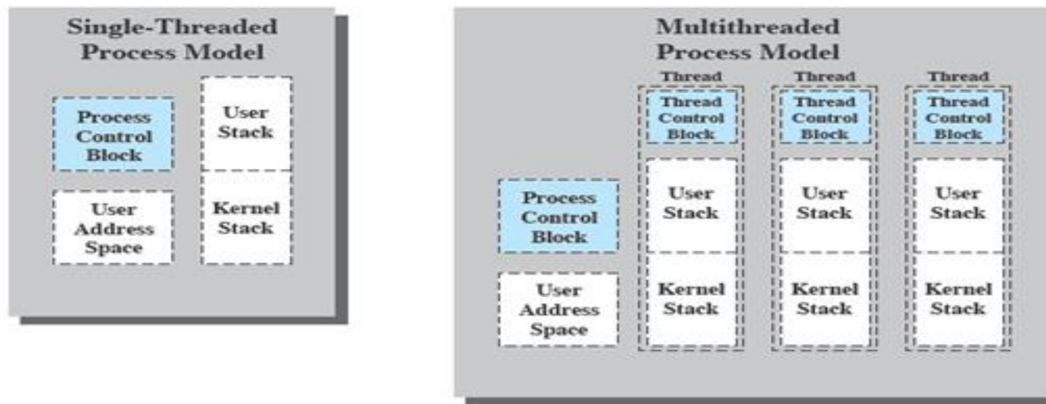


Figure 1.1: Process vs. Thread

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.

In an OS that supports threads, scheduling and dispatching is done on a thread basis; hence, most of the state information dealing with execution is maintained in thread-level data structures. There are, however, several actions that affect all of the threads in a process and that the OS must manage at the process level. For example, suspension involves swapping the address space of one process out of main memory to make room for the address space of another process. Because all threads in a process share the same address space, all threads are suspended at the same time. Similarly, termination of a process terminates all threads within that process.

1.2 General 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 (Note, however, some thread implementation may use the term suspended).

There are four basic thread operations associated with a change in thread state:

- **Spawn (new):** 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 de-allocated.

1.3. Example showing benefits of multithreading

Example: Single threaded RPC calls vs. Multithreaded RPC Calls

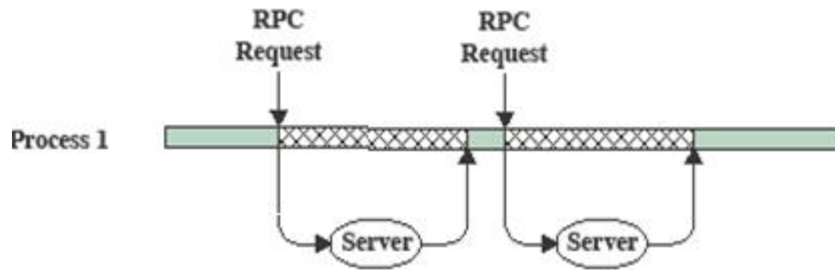


Figure 1.2: Single threaded RPC calls

An application performs two remote procedure calls (RPCs) 2 to two different hosts to obtain a combined result. In a single-threaded program, the results are obtained in sequence, so 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. In SMP, the thread can run simultaneously.

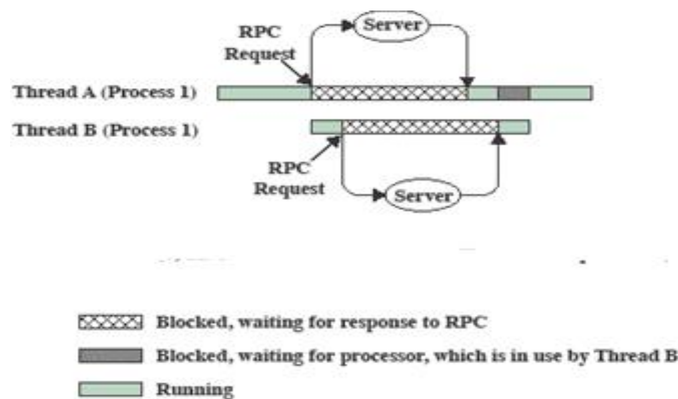


Figure 1.3: Multi-threaded RPC calls

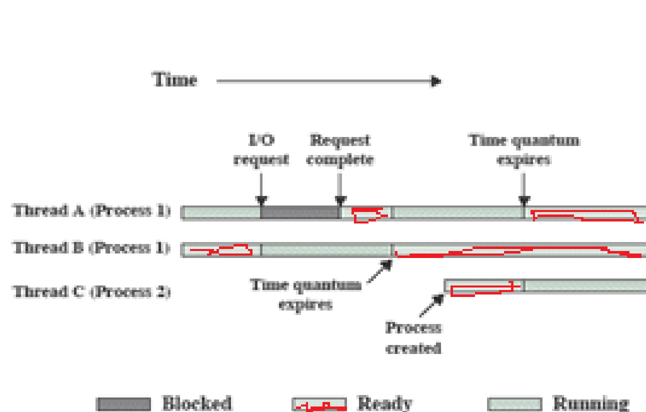


Figure 1.4: Interleaved executions of multi-threads

On a uniprocessor, multiprogramming enables the interleaving of multiple threads within multiple processes. In the example shown above, 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 when its time slice is exhausted.

1.4. *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.

2. Thread Implementation Approaches

There are two broad categories of thread implementation: user-level threads (**ULTs**) and kernel-level threads (**KLTs**) (known as kernel-supported threads or lightweight processes).

2.1 Pure User Level Threads (Many to One (M:1) model for CPU scheduling)

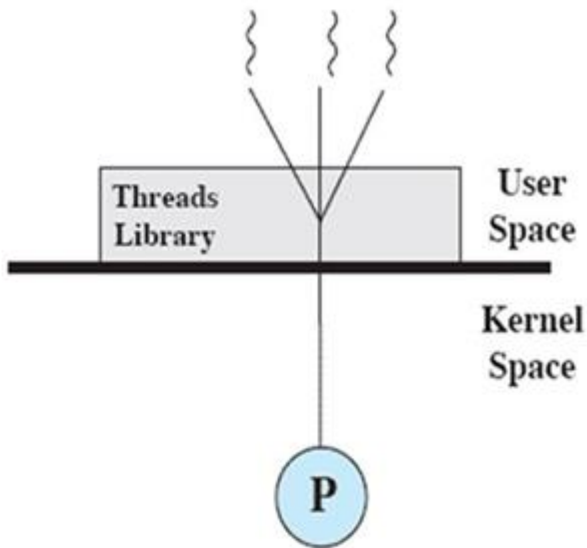


Figure 2.1: Pure user level threads (many to one mapping model)

An M:1 model implies that all application-level threads map to a single kernel-level scheduled entity; the kernel has no knowledge of the application threads. With this approach, context switching can be done very quickly and, in addition, it can be implemented even on simple kernels which do not support threading. One of the major drawbacks however is that it cannot benefit from the hardware acceleration

on multi-core processors or multi-processor computers: there is never more than one thread being scheduled at the same time. It is used by GNU Portable Threads and Solaris Green Thread.

In a pure ULT facility, all of the work of thread management is done in user address space (by application and threads library) and the kernel is not aware of the existence of threads. 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 function call. The threads library creates a data structure (in user address space) 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.

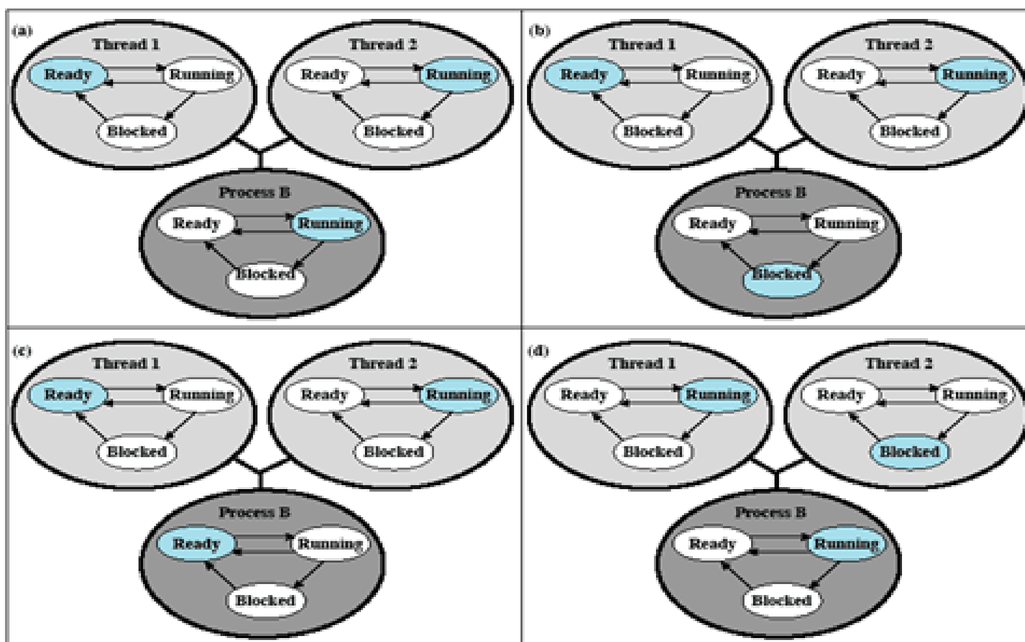


Figure 2.2: Relationships between ULT States and Process States

All the activities for thread management take 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 (refer to Figure 2.2). 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 above. 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 (see figure (b)).

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 (see figure (c)).

3. Thread 2 has reached a point where it needs some action performed by thread: Thread 2 enters a Blocked state, and thread 1 makes transition from Ready to Running. The process itself remains in the Running state (see figure (d)). 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.

There are a number of 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.

There are two 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 non-blocking 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.

2.2 Summary on ULTs (M:1 model)

User-level threads (ULT): Many-to-one thread mapping

- Implemented by user-level runtime libraries
    - Create, destroy, schedule, synchronize threads at user-level
    - Saving and restoring thread contexts
- OS is not aware of user-level threads
    - OS thinks each process contains only a single thread of control

Advantages

- Does not require OS support; Portable
    - Can run on any OS
- Can tune scheduling policy to meet application demands
    - You can create several scheduling policies in your thread library
    - Use one that is appropriate for your users (e.g. FIFO, SJF), rather than using a general purpose scheduler
    - Unfortunately: cooperative scheduling (non-preemptive). Why?
- Lower overhead thread operations since no system calls
    - Thread switching does not require kernel mode
    - All thread management data structures in user address space

Disadvantages

- Cannot utilize multiprocessors
    - OS assigns one process to only one processor at a time
    - In ULT scheme, OS only manages processes
- Entire process blocks when one thread blocks
    - Your thread library does not know if a read file operation will block (perform I/O to the disk) or not (read from memory).
    - Reason: OS does not inform the application that the system call is blocked
    - Hence, the thread library cannot switch running another thread
    - (e.g.,) 1 process in the system created 10 threads in the process, and thread-1 calls read(file), and file is not in memory. When thread-1 is waiting, the thread library cannot switch to another thread. CPU is underutilized!

2.3 *Pure Kernel-level Threads (1 to 1 Model for CPU scheduling)*

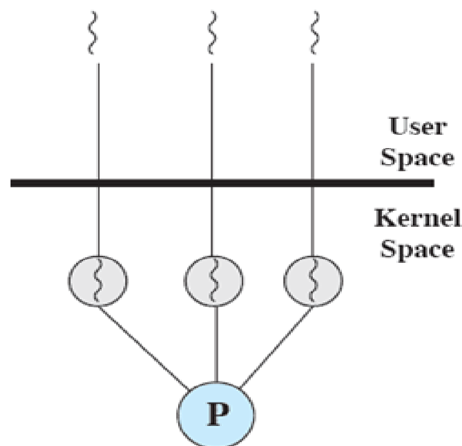


Figure 2.3: Pure kernel level threads (one to one mapping)

In 1:1 (Pure Kernel-level threading) model, threads are created by the user. Threads are in 1-1 correspondence with schedulable entities in the kernel. This is the simplest possible threading implementation. Win32 used this approach from the start Windows NT/XP/2000. Windows 7 is based on the same model (with better processor affinity for better utilizing SMP architecture). On Linux, the usual C library implements this approach (via the NPTL or older LinuxThreads). The same approach is used by Solaris (from 9 and later versions) and FreeBSD.

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. 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.

Uniprocessor DEC-VAX thread performance:

- Overhead for creating thread: ULT (37), KLT(948), Process(11,300) (Note: number represents unit time)
- Overhead for handling wait signal: ULT (37), KLT(441), Proces (1,840)

2.4. Summary on KLTs (1:1 model)

Kernel-level threads (KLT): One-to-one thread mapping

- OS provides each user-level thread with a kernel thread (e.g., Linux pthread implementation)
- Each kernel thread scheduled independently
- Thread operations (creation, scheduling, synchronization) performed by OS

Advantages
- Each kernel-level thread can run in parallel on a multiprocessor
- When one thread blocks, other threads from process can be scheduled

Disadvantages
- Higher overhead for thread operations (10-30x slower)
  - Requires a mode switch to the kernel
  - (e.g.,) running a user code, time-slice expires, change to kernel mode to perform thread scheduling
- OS must scale well with increasing number of threads

2.5. Combined Approaches (M:N Threading Model; or Hybrid Threading Model (M:N) && (1:1))

M:N maps some M number of application threads onto some N number of kernel entities, or "virtual processors." This is a compromise between kernel-level ("1:1") and user-level ("N:1") threading. In general, "M:N" threading systems are more complex to implement than either kernel or user threads, because changes to both kernel and user-space code are required. In the M:N implementation, the threading library is responsible for scheduling user threads on the available schedulable entities; this makes context switching of threads very fast, as it avoids system calls. However, this increases complexity and the likelihood of priority inversion as well as suboptimal scheduling without extensive (and expensive) coordination between the use-space thread scheduler and the kernel scheduler.

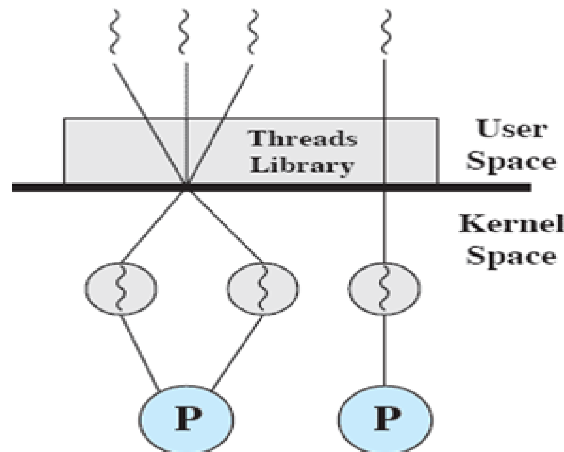


Figure 2.4: Combined Approach (hybrid many-to-many and one-to-one mappings)

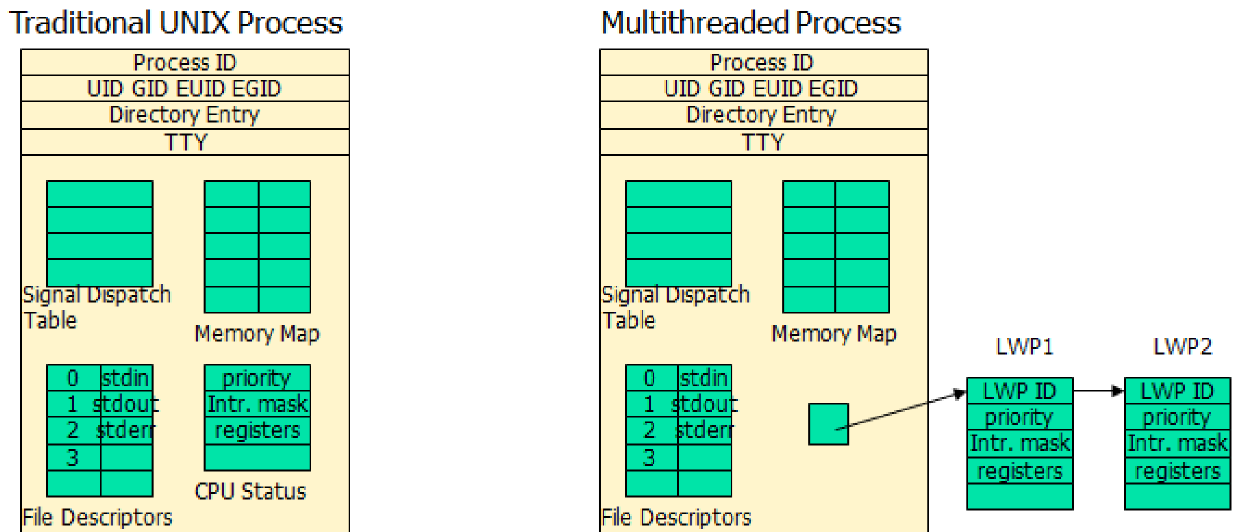


Figure 2.5: Process Structure in Traditional UNIX and Solaris

Some operating systems provide a combined ULT/ KLT facility (Figure 3). 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.

2.6. Summary on Combined Approach (M:N thread Model; M:N && 1:1 hybrid Model)

- Application creates m threads
- OS provides **pool** of n kernel threads
- Few user-level threads mapped to each kernel-level thread

Advantages

- Can get best of user-level and kernel-level implementations
- Works well given many short-lived user threads mapped to constant-size pool

Disadvantages

- Complicated (OS must export interfaces about the processors)
- How to select mappings?
  - Users are lazy
- How to determine the best number of kernel threads?
  - User specified
  - OS dynamically adjusts number depending on system load

3. Relationship between Threads and Processes

Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases.	TRIX

Figure 3.1: Relationship between threads and processes

The concepts of resource allocation and dispatching unit have traditionally been embodied in the single concept of the process—that is, as a 1:1 relationship between threads and processes. Recently, there has been much interest in providing for multiple threads within a single process, which is a many-to one relationship. However, as Figure 3.1 shows, the other two combinations have also been investigated, namely, a many-to-many relationship and a one-to-many relationship.

The idea of having a many-to-many relationship between threads and processes has been explored in the experimental operating system TRIX. In TRIX, there are the concepts of domain and thread. A domain is a static entity, consisting of an address space and "ports" through which messages may be sent and received. A thread is a single execution path, with an execution stack, processor state, and scheduling information.

As with the multithreading approaches discussed so far, multiple threads may execute in a single domain, providing the efficiency gains discussed earlier. However, it is also possible for a single user activity, or application, to be performed in multiple domains. In this case, a thread exists that can move from one domain to another.

4. Case Studies

Processes supported by different OS environments differ in a number of ways, including the following:

- How processes are named
- Whether threads are provided within processes
- How processes are represented
- How process resources are protected
- What mechanisms are used for inter-process communication and synchronization
- How processes are related to each other

4.1. Windows Thread Implementation

The native process structures and services provided by the Windows Kernel are relatively simple and general purpose, allowing each OS subsystem to emulate a particular process structure and functionality. Important characteristics of Windows processes are the following:

- Windows processes are implemented as objects.
- A process can be created as new process, or as a copy of an existing process.
- An executable process may contain one or more threads.
- Both process and thread objects have built-in synchronization capabilities.

4.1.1. Windows Kernels and APIs

As with most operating systems, Windows and UNIX both have kernels. The kernel provides the base functionality of the operating system. The major functionality of the kernel includes process management, memory management, thread management, scheduling, I/O management, and power management. It runs on the 7th major version revision of the NT kernel. XP, Vista, Windows 7 all run on the NT kernel (for example, Windows 7 has the version number NT 6.1). See more details about Windows History (http://en.wikipedia.org/wiki/History_of_Microsoft_Windows).

In UNIX, the API functions are called system calls. System calls are a programming interface common to all implementations of UNIX. The kernel is a set of functions that are used by processes through system calls.

Windows has an API for programming calls to the executive. In addition to this, each subsystem provides a higher-level API. This approach allows Windows operating systems to provide different APIs, some of which mimic the APIs provided by the kernels of other operating systems. The standard sub-system APIs include the Windows API (the Windows native API) and the POSIX API (the standards-based UNIX API).

4.2.2 Windows Subsystems

A subsystem is a portion of the Windows operating system that provides a service to application programs through a callable API. Subsystems come in two varieties, depending upon the application program that finally handles the request:

- Environment subsystems. These subsystems run in a user mode and provide functions through a published API. The Windows API subsystem provides an API for operating system services, GUI capabilities, and functions to control all user input and output. The Win32 subsystem and POSIX subsystem are part of the environment subsystems and are described as follows:
    - Win32 subsystem. The Win32 environment subsystem allows applications to benefit from the complete power of the Windows family of operating systems. The Win32 subsystem has a vast collection of functions, including those required for advanced operating systems, such as security, synchronization, virtual memory management, and threads. You can use Windows APIs to write 32-bit and 64-bit applications that run on all versions of Windows.

- POSIX subsystem and Windows Services for UNIX. To provide more comprehensive support for UNIX programs, Windows uses the Interix subsystem. Interix is a multiuser UNIX environment for a Windows-based computer. Interix conforms to the POSIX.1 and POSIX.2 standards. It provides all features of a traditional UNIX operating system, including pipes, hard links, symbolic links, UNIX networking, and UNIX graphical support through the X Windows system. It also includes case-sensitive file names, job control tools, compilation tools, and more than 300 UNIX commands and tools, such as KornShell, C Shell, awk, and vi. Because it is layered on top of the Windows kernel, the Interix subsystem is not an emulation. Instead, it is a native environment subsystem that integrates with the Windows kernel, just as the Win32 subsystem does. Shell scripts and other scripted applications that use UNIX and POSIX.2 tools run under Interix.
- Interix is the name of an optional, full-featured POSIX and Unix environment subsystem for Microsoft's Windows NT-based operating systems. Interix is a component of the Services for Unix (SFU). The most recent releases of Interix, 5.2 and 6.0, are components of the Windows Server 2003 R2, Windows Vista Enterprise and Ultimate editions and Windows Server 2008 under the name SUA (Subsystem for Unix-based Applications). Version 6.1 is included in Windows 7 (Enterprise and Ultimate) and Windows Server 2008 R2 (all editions).

Integral subsystems. These subsystems perform key operating system functions and run as a part of the executive or kernel. Examples are the user-mode subsystems, Local Security Authority subsystem (LSASS), and Remote Procedure Call subsystem (RPCSS).

4.2.3 Processes and Threads

In addition to being a preemptive multitasking operating system, Windows is also multithreaded, meaning that more than one thread of execution (or thread) can execute in a single task at once.

A process comprises:

- A private memory address space in which the process's code and data are stored.
- An access token against which Windows makes security checks.
- System resources such as files and Windows (represented as object handles).
- At least one thread to execute the code.

A thread comprises:

- A processor state including the current instruction pointer.
- A stack for use when running in user mode.
- A stack for use when running in kernel mode.

Since processes (not threads) own the access token, system resource handles, and address space, threads do NOT have their own address spaces nor do they have their own access token or system resource handles. Therefore, all of the threads in a process SHARE the same memory, access token, and system resources on a "per-process" rather than a "per-thread" basis. In a multithreaded program, the programmer

is responsible for making sure that the different threads don't interfere with each other by using these shared resources in a way that conflicts with another thread's use of the same resource.

The Windows kernel-mode process and thread manager handles the execution of all threads in a process.

4.2.4. Sharing a Single Address Space--Synchronizing Access To Data

Running each process in its own address space had the advantage of reliability since no process can modify another process's memory. However, all of a process's threads run in the same address space and have unrestricted access to all of the same resources, including memory. While this makes it easy to share data among threads, it also makes it easy for threads to step on each other. As mentioned before, multithreaded programs must be specially programmed to ensure that threads don't step on each other.

A section of code that modifies data structures shared by multiple threads is called a ***critical section*** . It is important than when a critical section is running in one thread that no other thread be able to access that data structure. Synchronization is necessary to ensure that only one thread can execute in a critical section at a time. This synchronization is accomplished through the use of some type of Windows NT synchronization object. Programs use Windows synchronization objects rather than writing their own synchronization both to save coding effort and for efficiency: when you wait on a Windows NT synchronization object, you do NOT use any CPU time testing the object to see when it's ready.

4.2.5. The Life Cycle of a Thread

Each thread has a ***dispatcher state*** that changes throughout its lifetime.

The most important dispatcher states are:

- Running: only one thread per processor can be running at any time.
- Ready: threads that are in the Ready state may be scheduled for execution the next time the kernel dispatches a thread. Which Ready thread executes is determined by their priorities.
- Waiting: threads that are waiting for some event to occur before they become Ready are said to be waiting. Examples of events include waiting for I/O, waiting for a message, and waiting for a synchronization object to become unlocked.

Intel has revealed that Windows 7 features new and improved multi-threading, which will help to improve power consumption and battery life. Previous versions of Windows often swapped threads around cores, which prevented them from entering lower power states and caused cache thrashing as separate cores raced to grab data processed by others.

The Windows 7 kernel changes this by improving thread affinity, locking threads to particular cores in order to allow unused CPU cores to enter low power C-states when they’re not in use – called thread parking - providing the CPU and motherboard supports this of course.

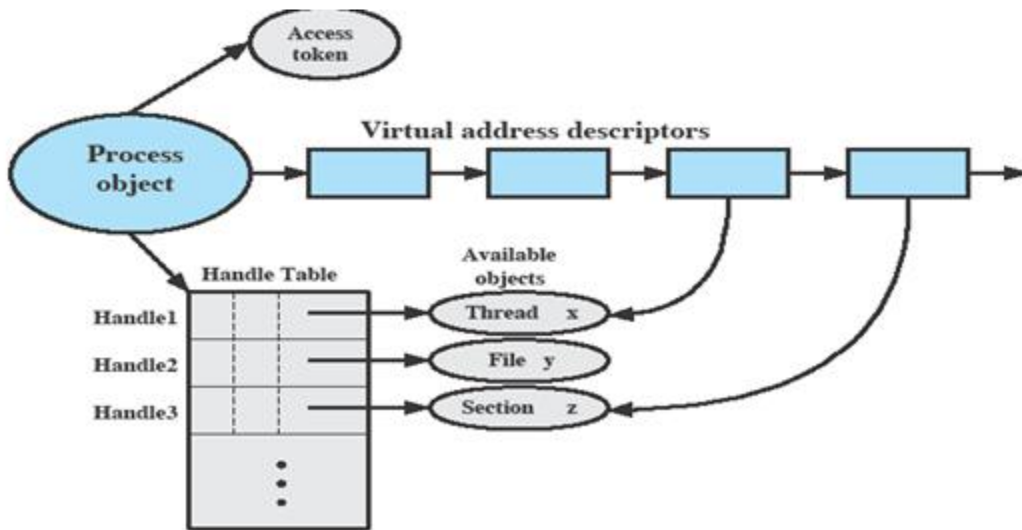


Figure 4.1: Windows Process and Its Resources

Thread Information Block (TIB) is a data structure that stores info about the currently running thread. This structure is also known as the Thread Environment Block (TEB).

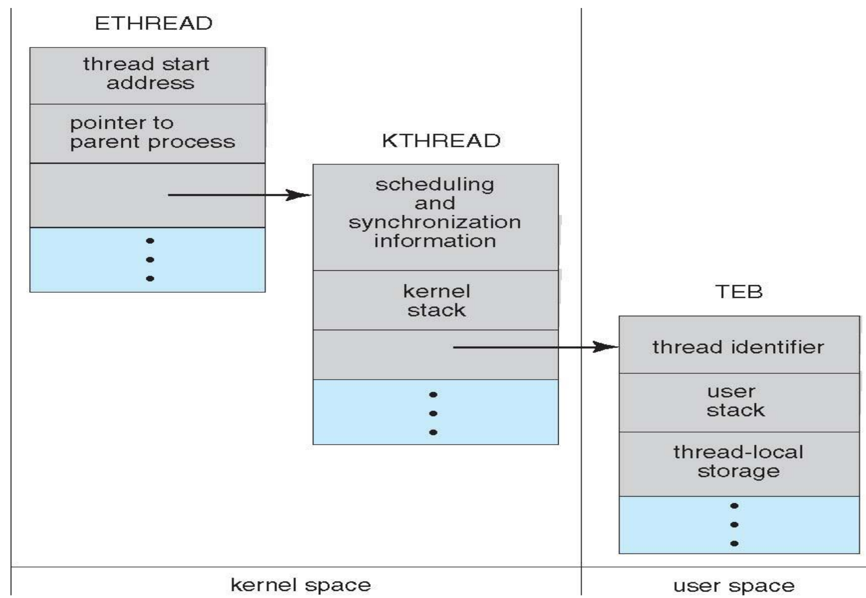


Figure 4.2: Thread Environment Block

Figure 4.1 illustrates the way in which a process relates to the resources it controls or uses. Each process is assigned a security access token, called the primary token of the process. When a user first logs on, Windows creates an access token that includes the security ID for the user. Every process that is created by or runs on behalf of this user has a copy of this access token. Windows uses the token to validate the user's ability to access secured objects or to perform restricted functions on the system and on secured

objects. The access token controls whether the process can change its own attributes. In this case, the process does not have a handle opened to its access token. If the process attempts to open such a handle, the security system determines whether this is permitted and therefore whether the process may change its own attributes.

Also related to the process is a series of blocks that define the virtual address space currently assigned to this process. The process cannot directly modify these structures but must rely on the virtual memory manager, which provides a memory-allocation service for the process.
Finally, the process includes an object table, with handles to other objects known to this process. Figure 4.1 shows a single thread. In addition, the process has access to a file object and to a section object that defines a section of shared memory.

Windows makes use of two types of process-related objects: processes and threads. A process is an entity corresponding to a user job or application that owns resources, such as memory and open files. A thread is a dispatchable unit of work that executes sequentially and is interruptible, so that the processor can turn to another thread.

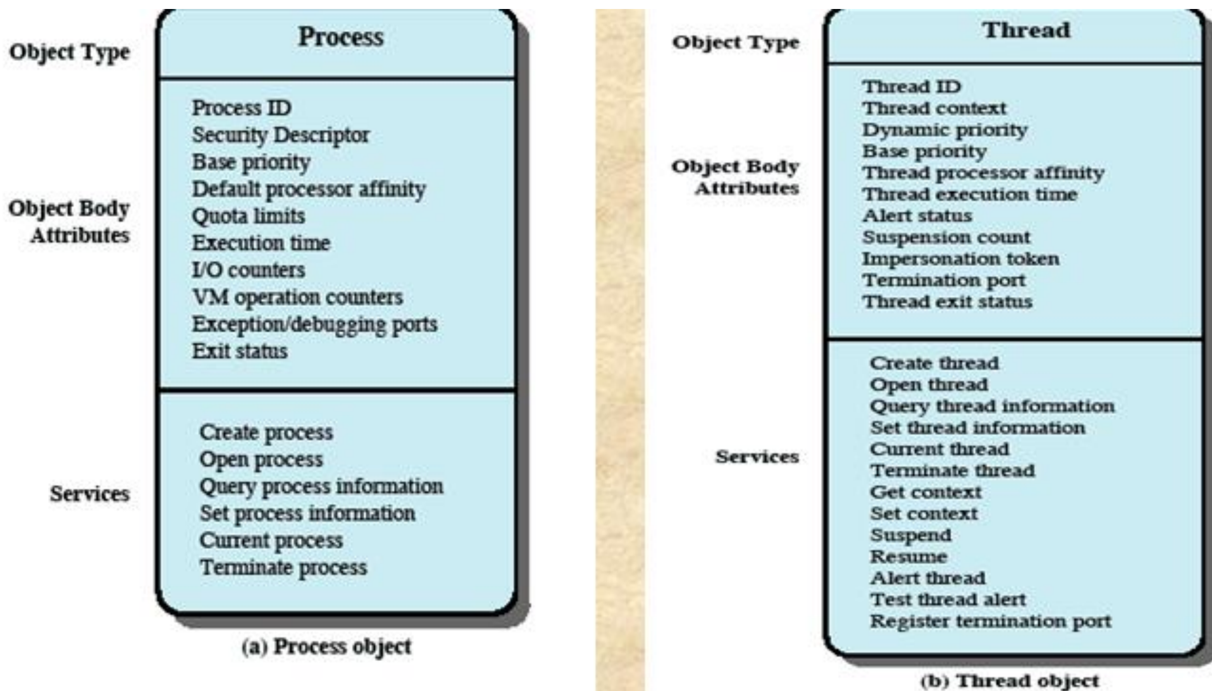


Figure 4.3: Windows Process and Thread Objects

Each Windows process is represented by an object whose general structure is shown in Figure 4.3. Each process is defined by a number of attributes and encapsulates a number of actions, or services, that it may perform. A process will perform a service when called upon through a set of published interface methods. When Windows creates a new process, it uses the object class, or type, defined for the Windows process as a template to generate a new object instance. At the time of creation, attribute values are assigned. Figure 4.3(b) depicts the object structure for a thread object.

A Windows process must contain at least one thread to execute. That thread may then create other threads. In a multiprocessor system, multiple threads from the same process may execute in parallel. Note that

some of the attributes of a thread resemble those of a process. In those cases, the thread attribute value is derived from the process attribute value. For example, the thread processor affinity is the set of processors in a multiprocessor system that may execute this thread; this set is equal to or a subset of the process processor affinity (we will study processor affinity in chapter 5 CPU Scheduling).

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, it is possible to alter the behavior of a thread by altering its context while it is suspended.

Windows supports concurrency among processes because threads in different processes may execute concurrently (appear to run at the same time). Moreover, multiple threads within the same process may be allocated to separate processors and execute simultaneously (actually run at the same time).

Process ID	A unique value that identifies the process to the operating system.
Security descriptor	Describes who created an object, who can gain access to or use the object, and who is denied access to the object.
Base priority	A baseline execution priority for the process's threads.
Default processor affinity	The default set of processors on which the process's threads can run.
Quota limits	The maximum amount of paged and nonpaged system memory, paging file space, and processor time a user's processes can use.
Execution time	The total amount of time all threads in the process have executed.
I/O counters	Variables that record the number and type of I/O operations that the process's threads have performed.
VM operation counters	Variables that record the number and types of virtual memory operations that the process's threads have performed.
Exception/debugging ports	Interprocess communication channels to which the process manager sends a message when one of the process's threads causes an exception. Normally, these are connected to environment subsystem and debugger processes, respectively.
Exit status	The reason for a process's termination.

Thread ID	A unique value that identifies a thread when it calls a server.
Thread context	The 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, which is a subset or all of the processor affinity of the thread's process.
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 (used by subsystems).
Termination port	An interprocess communication channel to which the process manager sends a message when the thread terminates (used by subsystems).
Thread exit status	The reason for a thread's termination.

A multithreaded process achieves concurrency without the overhead of using multiple processes. Threads within the same process can exchange information through their common address space and have access to the shared resources of the process. Threads in different processes can exchange information through shared memory that has been set up between the two processes. Multithreaded process is an efficient means of implementing a server application. For example, one server process can service a number of clients concurrently.

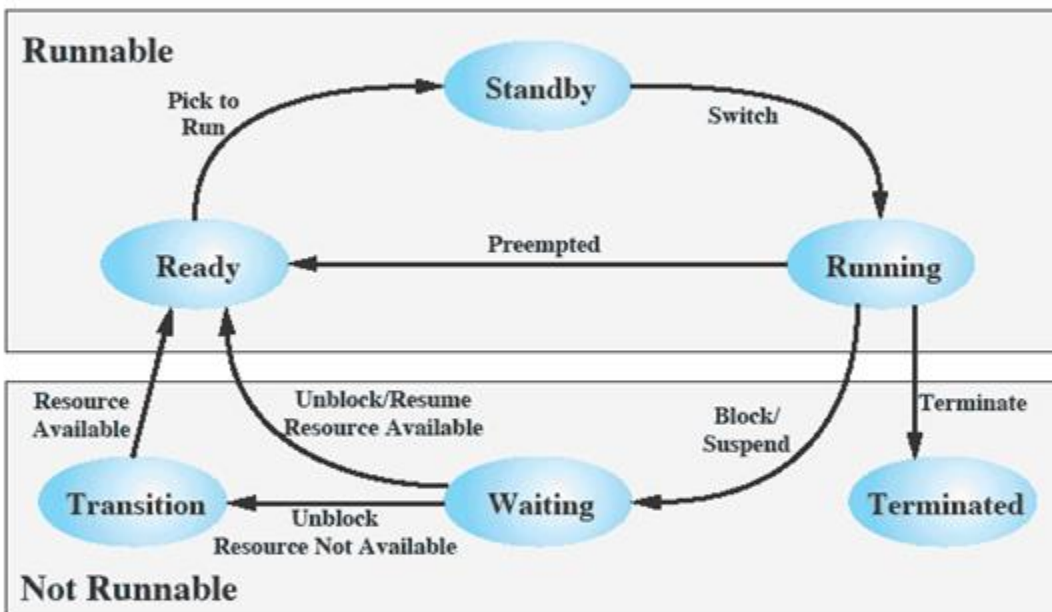


Figure 4.4: Windows Thread States

4.2.6. Early Window NT's Threads and Fibers

A thread is Windows NT's smallest kernel-level object of execution. Processes in NT can consist of one or more threads. When a process is created, one thread is generated along with it, called the primary thread. This object is then scheduled on a system wide basis by the kernel to execute on a processor. After the primary thread has started, it can create other threads that share its address space and system resources but have independent contexts, which include execution stacks and thread specific data. A thread can execute any part of a process' code, including a part currently being executed by another thread. It is through threads, provided in the Win32 application programmer interface (API), that Windows NT allows programmers to exploit the benefits of concurrency and parallelism.

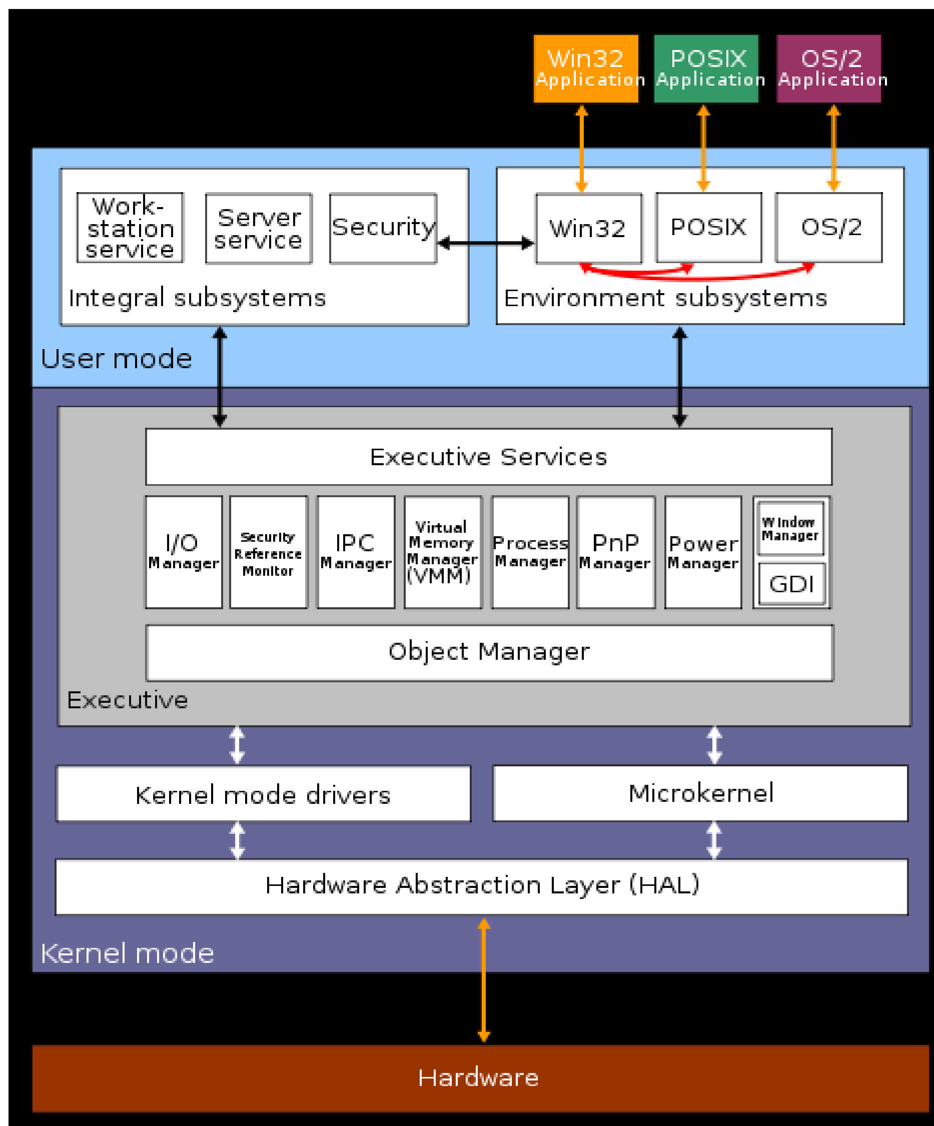


Figure 4.5: Windows NT Architecture

A fiber is NT's smallest user-level object of execution. It executes in the context of a thread and is unknown to the operating system kernel. A thread can consist of one or more fibers as determined by the application programmer. Some literature assumes that there is a one-to-one mapping of user-level objects to kernel-level objects, this is inaccurate. Fibers are not preemptively scheduled. You schedule a fiber by switching to it from another fiber. The system still schedules threads to run. When a thread running fibers is preempted, its currently running fiber is preempted but remains selected. The selected fiber runs when its thread runs. Windows NT does provide the means for many-to-many scheduling. However, NT's design is poorly documented and the application programmer is responsible for the control of fibers such as allocating memory, scheduling them on threads and preemption. An illustrative example of early NT's thread implementation approach is shown in Figure 4.6.

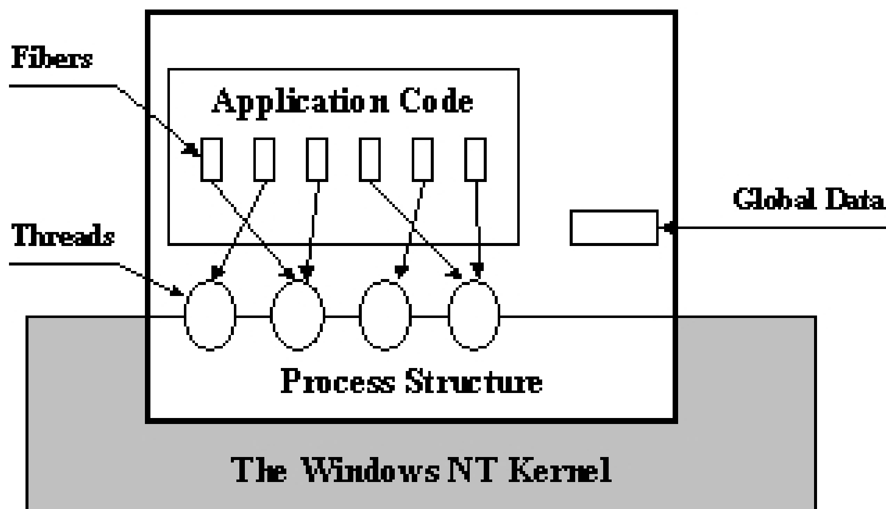
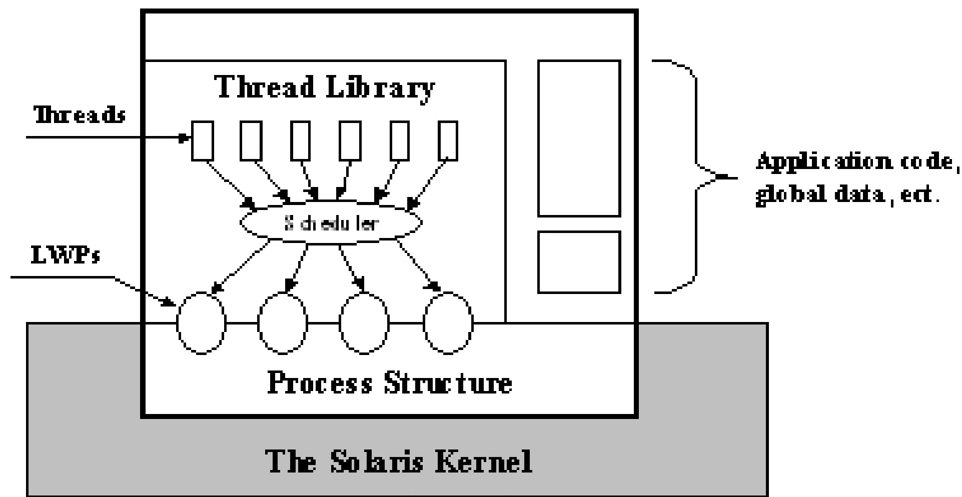


Figure 4.6: Windows NT Fibers and Threads

4.3.1. Solaris's LWPs and Threads

A light weight process (LWP) is Solaris's smallest kernel-level object of execution. A Solaris process consists of one or more light weight processes. In Solaris, a *thread* is the smallest user-level object of execution. Like Windows NT's fiber, they are not executable alone. A Solaris thread must execute in the context of a light weight process. Unlike NT's fibers, which are controlled by the application programmer, Solaris's threads are implemented and controlled by a system library. The library controls the mapping and scheduling of threads onto LWPs automatically. One or more threads can be mapped to a light weight process. The mapping is determined by the library or the application programmer. Since the threads execute in the context of a light weight process, the operating system kernel is unaware of their existence. The kernel is only aware of the LWPs that threads execute on. An illustrative example of this design is shown in.



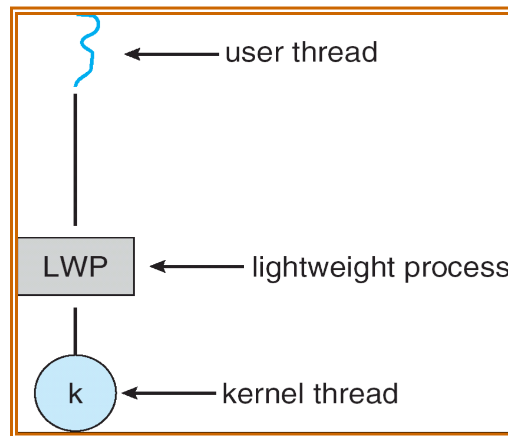


Figure 4.5: Solaris LWPs

Solaris makes use of four separate thread-related concepts:

- Process: This is the normal UNIX process and includes the user's address space, stack, and process control block.
- User-level threads: Implemented through a threads library in the address space of a process, these are invisible to the OS. A user-level thread (ULT) is a user-created unit of execution within a process.
- Lightweight processes: A lightweight process (LWP) can be viewed as a mapping between ULTs and kernel threads. Each LWP supports ULT and maps to one kernel thread. LWPs are scheduled by the kernel independently and may execute in parallel on multiprocessors.
- Kernel threads: These are the fundamental entities that can be scheduled and dispatched to run on one of the system processors.

Figure 4.5 illustrates the relationship among these four entities. Note that there is always exactly one kernel thread for each LWP.

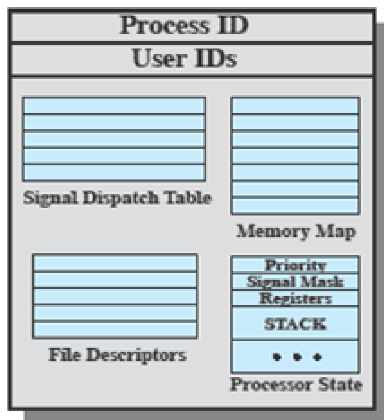
4.3.2. Is LWP in kernel address space or in user address space?

It depends on implementation:

- User address space: An LWP is visible within a process to the application. Thus, LWP data structures exist within their respective process address space. At the same time, each LWP is bound to a single dispatchable kernel thread, and the data structure for that kernel thread is maintained within the kernel's address space.
- Kernel address space: ULT library makes system call to request LWPs from the kernel.

A process may consist of a single ULT bound to a single LWP. In this case, there is a single thread of execution, corresponding to a traditional UNIX process. When concurrency is not required within a single process, an application uses this process structure. If an application requires concurrency, its process contains multiple threads, each bound to a single LWP, which in turn are each bound to a single kernel thread.

In addition, there are kernel threads that are not associated with LWPs. The kernel creates, runs, and destroys these kernel threads to execute specific system functions. The use of kernel threads rather than kernel processes to implement system functions reduces the overhead of switching within the kernel (from a process switch to a thread switch).



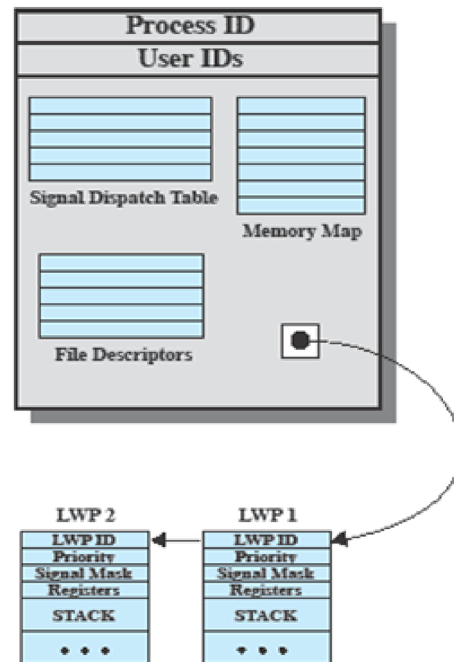


Figure 4.6 (Revisit Figure 2.5): Process Structure in UNIX and Solaris

Figure 4.6 compares, in general terms, the process structure of a traditional UNIX system with that of Solaris. On a typical UNIX implementation, the process structure includes the process ID; the user IDs; a signal dispatch table, which the kernel uses to decide what to do when sending a signal to a process; file descriptors, which describe the state of files in use by this process; a memory map, which defines the address space for this process; and a processor state structure, which includes the kernel stack for this

process. Solaris retains this basic structure but replaces the processor state block with a list of structures containing one data block for each LWP.

The LWP data structure includes the following elements:

- An LWP identifier
- The priority of this LWP and hence the kernel thread that supports it
- A signal mask that tells the kernel which signals will be accepted
- Saved values of user-level registers (when the LWP is not running)
- The kernel stack for this LWP, which includes
    - system call arguments,
    - - results, and
    - - error codes for each call level
- Resource usage and profiling data
- Pointer to the corresponding kernel thread
- Pointer to the process structure

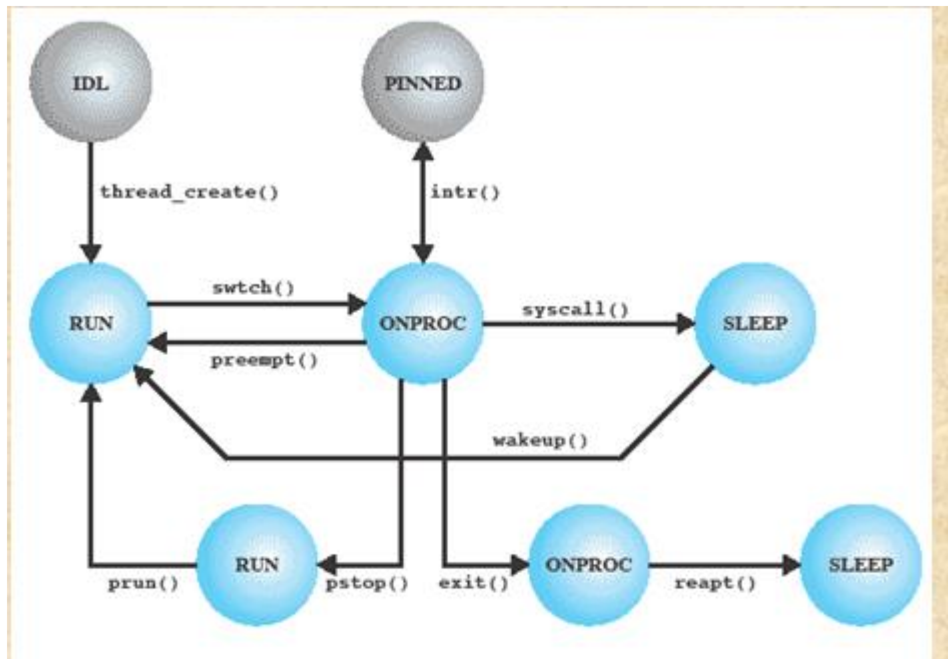


Figure 4.7: Solaris Thread States (Solaris 10) 1:1 mapping between ULTs and LPWs

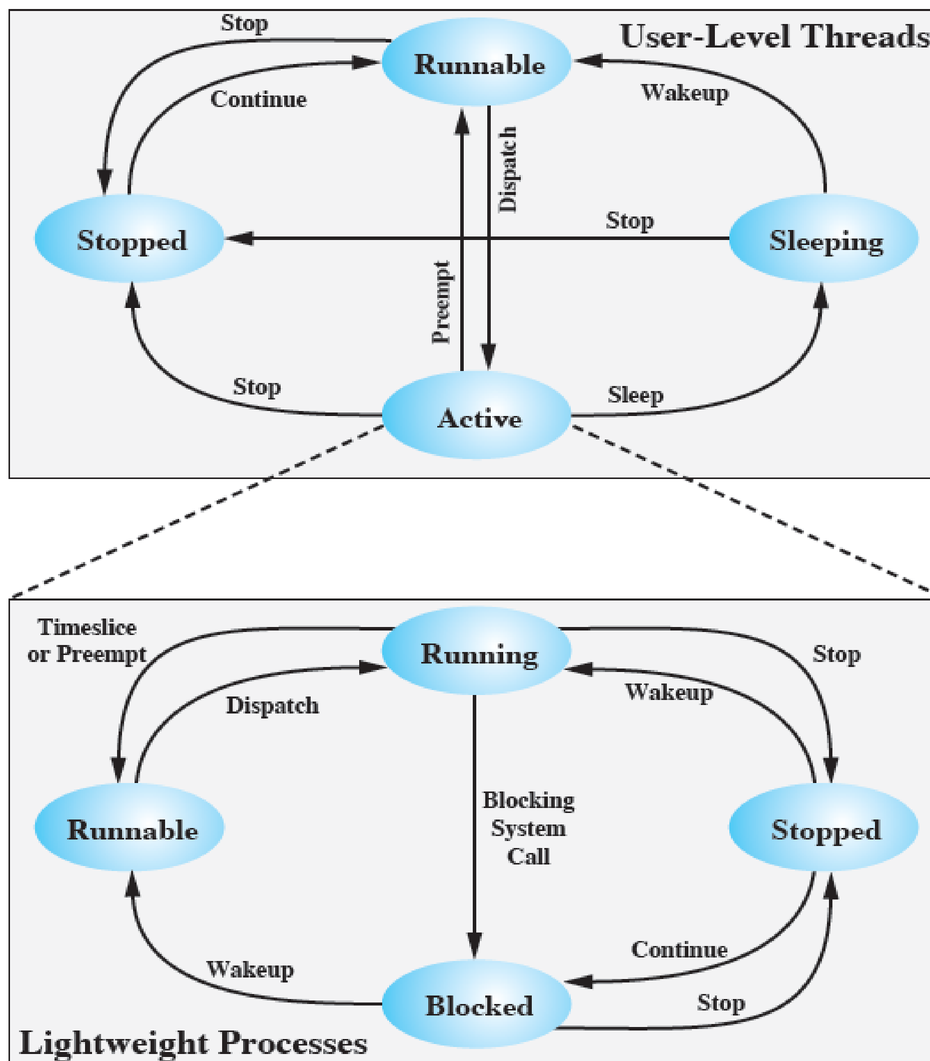


Figure 4.8: Solaris ULTs and LWT States (Solaris 8, 9)

Let us assume L represents number of LWPs and U represents the number of ULTs.

- No benefit to having L > U
- If U > L, some threads may have to wait for an LWP to run
  - Active thread - executing on an LWP
  - Runnable thread - waiting for an LWP
- Executing synchronous system call blocks LWP
  - Need at least k LWPs to allow k concurrent I/O operations
- Blocking on synchronization mutex, condition variable) - LWP switches to run another thread
- If N CPUs, need at least N LWPs to keep them all busy
- Kernel may take away LWPs if some are idle for a long time, or allocate more LWPs as needed

4.3.3. Combination of 1:1 and M:N models

- Supports both bound and unbound threads
    - Bound threads - permanently mapped to a single, dedicated LWP
    - Unbound threads - may move among LWPs in set
- Thread creation, scheduling, synchronization done in user space
- Flexible approach, “best of both worlds”
- Used in Solaris implementation of Pthreads and several other Unix implementations (IRIX, HP-UX)

4.3.4 Interrupt Handling(converting interrupts to kernel threads)

Most operating systems contain two fundamental forms of concurrent activity: processes and interrupts. Processes (or threads) cooperate with each other and manage the use of shared data structures by means of a variety of primitives that enforce mutual exclusion (only one process at a time can execute certain code or access certain data) and that synchronize their execution. Interrupts are synchronized by preventing their handling for a period of time. Solaris unifies these two concepts into a single model, namely kernel threads and the mechanisms for scheduling and executing kernel threads. To do this, interrupts are converted to kernel threads.

The motivation for converting interrupts to threads is to reduce overhead. Interrupt handlers often manipulate data shared by the rest of the kernel. Therefore, while a kernel routine that accesses such data is executing, interrupts must be blocked, even though most interrupts will not affect that data. Typically, the way this is done is for the routine to set the interrupt priority level higher to block interrupts and then lower the priority level after access is completed. These operations take time. The problem is magnified on a multiprocessor system. The kernel must protect more objects and may need to block interrupts on all processors.

When an interrupt occurs, it is delivered to a particular processor and the thread that was executing on that processor is pinned. A pinned thread cannot move to another processor and its context is preserved; it is simply suspended until the interrupt is processed. The processor then begins executing an interrupt thread. There is a pool of deactivated interrupt threads available, so that a new thread creation is not required. The interrupt thread then executes to handle the interrupt. If the handler routine needs access to a data structure that is currently locked in some fashion for use by another executing thread, the interrupt thread must wait for access to that data structure. An interrupt thread can only be preempted by another interrupt thread of higher priority. Experience with Solaris interrupt threads indicates that this approach provides superior performance to the traditional interrupt-handling strategy.

4.3.4 M:N vs. 1:1 Mapping

Solaris 8 ships with two threading libraries: many-to-many and one-to-one. The names refer to how the model maps Solaris threads to LWPs and kernel threads. By default, the many-to-many thread library is used in Solaris 8. In certain circumstances, this leads to thread starvation and even hangs (application crashes). From experience, it has been noted that Oracle and IBM WebSphere Application Server running on Solaris 8 seems particularly prone to these problems.

By default, the many-to-many thread library is used on Solaris 8. Switching to the one-to-one library reduces the possibility of thread starvation and improves scalability. This improves the overall performance of the system. One-to-one is the default implementation in Solaris 9 and subsequent releases.

4.4 Linux Threads

A process, or task, in Linux is represented by a task_struct data structure. The task_struct data structure contains information in a number of categories:

- **State:** The execution state of the process (executing, ready, suspended, stopped, zombie). This is described subsequently.
- **Scheduling information:** Information needed by Linux to schedule processes. A process can be normal or real time and has a priority. Real-time processes are scheduled before normal processes, and within each category, relative priorities can be used. A counter keeps track of the amount of time a process is allowed to execute.
- **Identifiers:** Each process has a unique process identifier and also has user and group identifiers. A group identifier is used to assign resource access privileges to a group of processes.
- **Inter-process communication:** Linux supports the IPC mechanisms found in UNIX SVR4, described (shared memory based, message based)
- **Links:** Each process includes a link to its parent process, links to its siblings (processes with the same parent), and links to all of its children.
- **Times and timers:** Includes process creation time and the amount of processor time so far consumed by the process. A process may also have associated one or more interval timers. A process defines an interval timer by means of a system call; as a result, a signal is sent to the process when the timer expires. A timer may be single use or periodic.
- **File system:** Includes pointers to any files opened by this process, as well as pointers to the current and the root directories for this process.
- **Address space:** Defines the virtual address space assigned to this process.
- **Processor-specific context:** The registers and stack information that constitute the context of this process.

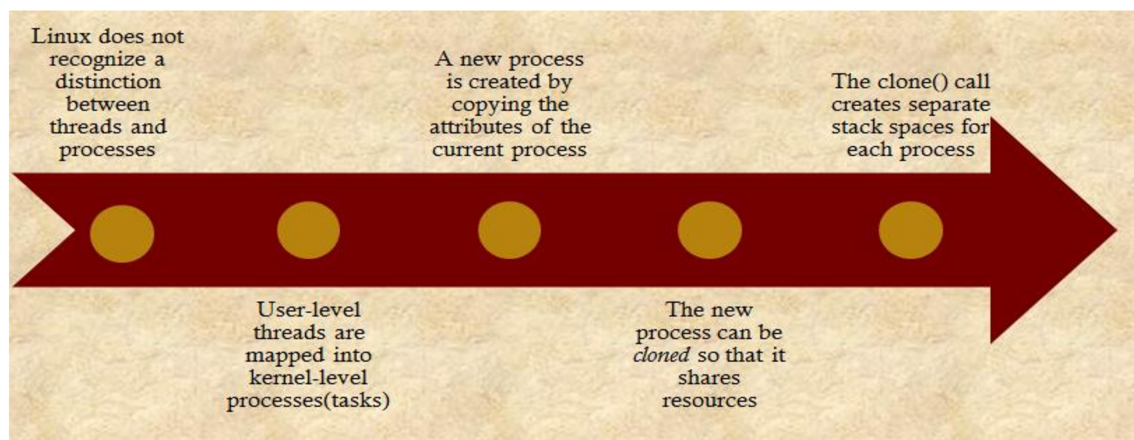


Figure 4.9: Linux Thread Creation

Unique solution:

- Does not recognize a distinction between threads and processes
- User-level threads are mapped into kernel-level processes
- These processes share the same group ID
    - To share resources and avoid the need
    - Avoid context switch when scheduler switches among processes in the same group
- A new thread is created → a process is “cloned” (rather than forked)

- Some clone flags to define shared elements

We have seen that modern versions of UNIX offer kernel-level threads. Linux provides a unique solution in that it does not recognize a distinction between threads and processes. Using a mechanism similar to the lightweight processes of Solaris, user-level threads are mapped into kernel-level processes. Multiple user-level threads that constitute a single user-level process are mapped into Linux kernel-level processes that share the same group ID. This enables these processes to share resources such as files and memory and to avoid the need for a context switch when the scheduler switches among processes in the same group.

A new process is created in Linux by copying the attributes of the current process. A new process can be *cloned so that it shares resources, such as files, signal* handlers, and virtual memory. When the two processes share the same virtual memory, they function as threads within a single process via clone() system call. However, no separate type of data structure is defined for a thread.

When the Linux kernel performs a switch from one process to another, it checks whether the address of the page directory of the current process is the same as that of the to-be-scheduled process. If they are, then they are sharing the same address space, so that a context switch is basically just a jump from one location of code to another location of code. Although cloned processes that are part of the same process group can share the same memory space, they cannot share the same user stacks. Thus the clone() call creates separate stack spaces for each process.

CLONE_CLEARID	Clear the task ID.
CLONE_DETACHED	The parent does not want a SIGCHLD signal sent on exit.
CLONE_FILES	Shares the table that identifies the open files.
CLONE_FS	Shares the table that identifies the root directory and the current working directory, as well as the value of the bit mask used to mask the initial file permissions of a new file.
CLONE_IDLETASK	Set PID to zero, which refers to an idle task. The idle task is employed when all available tasks are blocked waiting for resources.
CLONE_NEWNS	Create a new namespace for the child.
CLONE_PARENT	Caller and new task share the same parent process.
CLONE_PTRACE	If the parent process is being traced, the child process will also be traced.
CLONE_SETTID	Write the TID back to user space.
CLONE_SETTLS	Create a new TLS for the child.
CLONE_SIGHAND	Shares the table that identifies the signal handlers.
CLONE_SYSVSEM	Shares System V SEM_UNDO semantics.
CLONE_THREAD	Inserts this process into the same thread group of the parent. If this flag is true, it implicitly enforces CLONE_PARENT.
CLONE_VFORK	If set, the parent does not get scheduled for execution until the child invokes the *execve()* system call.
CLONE_VM	Shares the address space (memory descriptor and all page tables).

Figure 4.10: Linux clone() system call flags

4.4.1 Linux Thread Life-Cycle

Running: Corresponds to two states.

- A Running process is either executing or
- It is ready to execute.

Interruptible: A blocked state, in which the process is waiting for an event, such as the end of an I/O operation, the availability of a resource, or a signal from another process.

Uninterruptible:

- Another blocked state.
- The difference between the Interruptible state is that in this state, a process is waiting directly on hardware conditions and therefore will not handle any signals.

Stopped:

- The process has been halted and can only resume by positive action from another process.

- (e.g.,) A process that is being debugged can be put into the Stopped state.

Zombie: The process has been terminated but, for some reason, still must have its task structure in the process table.

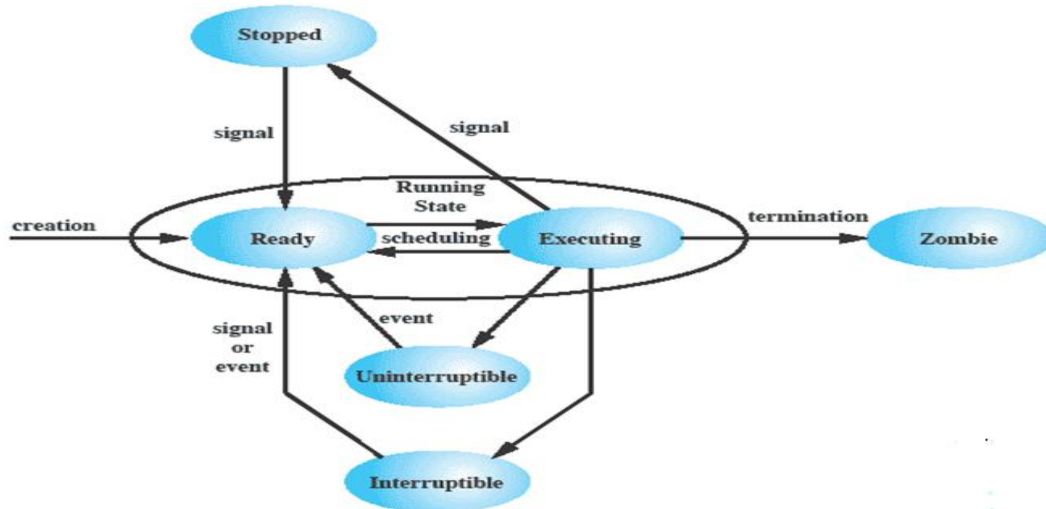


Figure 4.11: Linux Process/Thread Model

4.4.2 Native POSIX Thread Library (NPTL)

- Thread creation is done through clone() system call: clone() allows a child task to share the address space of the parent task (process)
- Linux refers to them as tasks rather than threads
  - In NPTL, all threads belong to a single process. NPTL threads are based on KSEs(kernel scheduling entities). A non-threaded process is also a KSE. A threaded process has more than one KSEs. All KSEs are scheduled by the kernel.
  - NPTL is based on fast mutual exclusion (futex) mechanism.
  - Not fully POSIX compliant
  - See the papers on OS course website about NPTL design and implantation.

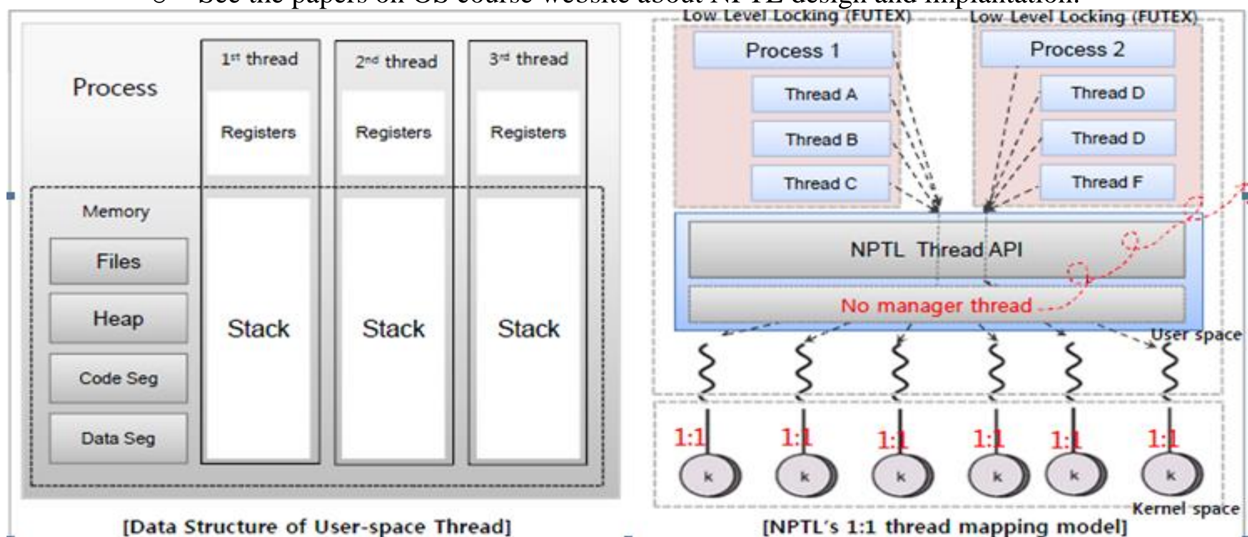
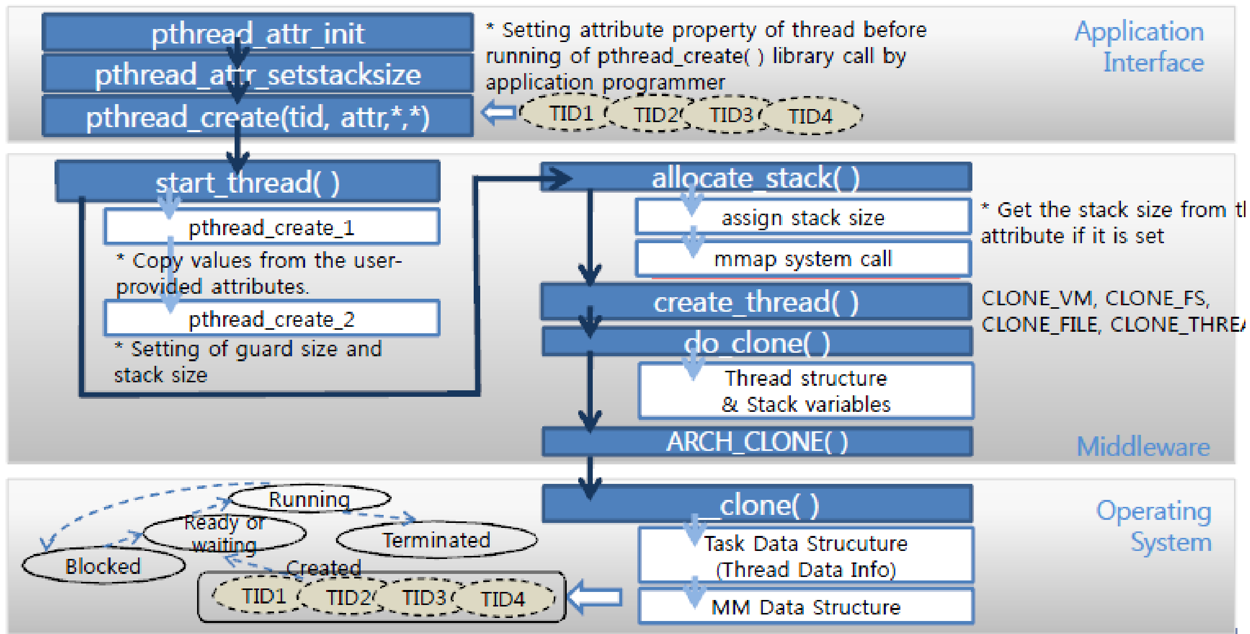


Figure 4.12: Linux Threads (1:1 Mapping)



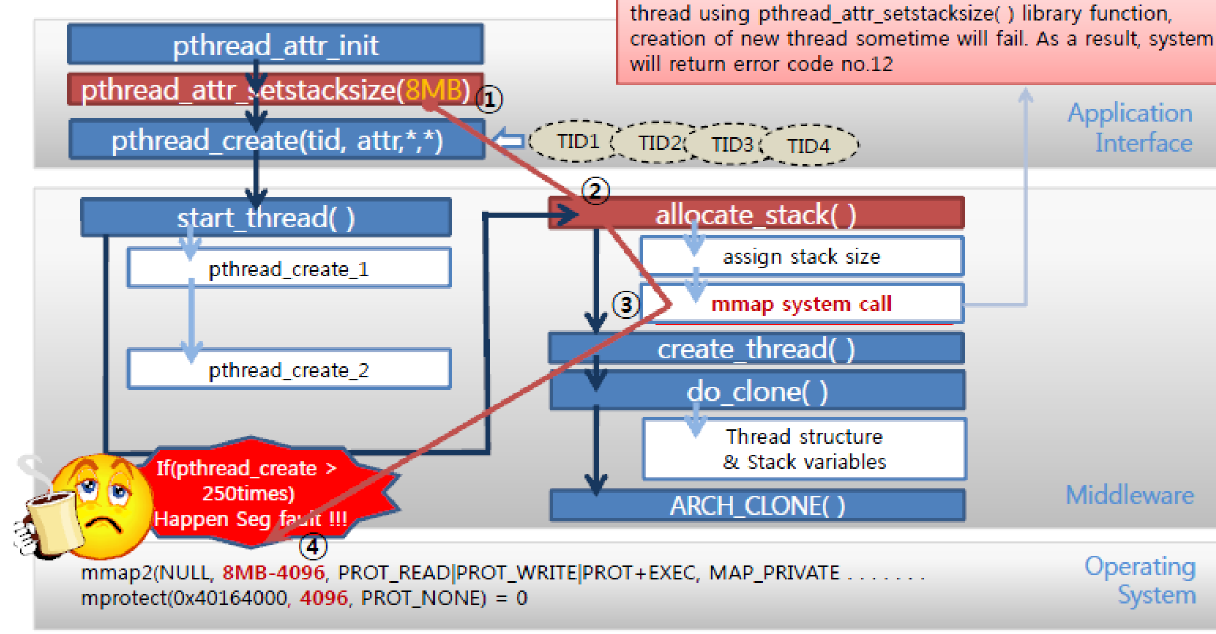


Figure 4.13: Thread creation by clone() system call

Utilize gettid( ) instead of getpid( ) in NPTL thread model to find out the unique number of thread executed in relevant function region to apply normal priority to threads produced as nice value.

Priority (user-space value)	PID/TID	Function Name (API)	Call (classification)	Thread Model	
				LinuxThread (interface)	NPTL (interface)
Normal Priority (from -20 to 19)	Process	setpriority( ) nice( )	System call	getpid( )	gettid( )
	Thread	setpriority( ) nice( )	System call	getpid( )	gettid( )
Real-time Priority (from 1 to 99)	Process	sched_setscheduler( ) sched_setparam( )	System call	getpid( )	gettid( )
	Thread	pthread_setschedprio( ) pthread_setschedparam( )	Library call	getpid( )	gettid( )

Architecture	File name	gettid( ) syscall No.
X86	./arch/i386/kernel/entry.S ./arch/x86/kernel/syscall_table_32.S	224
ARM	./arch/arm/kernel/call.S	224
MIPS	./arch/mips/kernel/scall32-o32.S ./arch/mips/kernel/scall64-932.S	4,222
PPC	./arch/ppc/kernel/misc.S	207

Figure 4.14: Linux gettid()

5. Summary on Thread Implementations on various Operating Systems

There are many different and incompatible implementations of threading. These include both kernel-level and user-level implementations. However, they often follow more or less closely the POSIX Threads interface.

Kernel-level implementation examples

- Light Weight Kernel Threads (LWKT) in various BSDs
- M:N threading
- Native POSIX Thread Library (NPTL) for Linux, an implementation of the POSIX Threads (pthreads) standard
- Apple Multiprocessing Services version 2.0 and later, uses the built-in nanokernel in Mac OS 8.6 and later which was modified to support it.
- Microsoft Windows from Windows NT and later versions.

User-level implementation examples

- GNU Portable Threads
- Solaris Green Thread
- Netscape Portable Runtime (includes a user-space fibers implementation)

Hybrid implementation examples

- Scheduler activations used by the NetBSD native POSIX threads library implementation (an M:N model as opposed to a 1:1 kernel or userspace implementation model)
- The OS for the Tera/Cray MTA
- Microsoft Windows 7 (????? Because of fibers support??)

Fiber implementation examples

Fibers can be implemented without operating system support, although some operating systems or libraries provide explicit support for them.

- Win32 supplies a fiber API (Windows NT 3.51 SP3 and later)
- Ruby as Green threads

6. Summary on user threads vs. kernel threads

- User-level threads
  - created and managed by a threads library that runs in the user space of a process
  - a mode switch is not required to switch from one thread to another
  - only a single user-level thread within a process can execute at a time
  - if one thread blocks, the entire process is blocked
- Kernel-level threads
  - threads within a process that are maintained by the kernel
  - a mode switch is required to switch from one thread to another
  - multiple threads within the same process can execute in parallel on a multiprocessor
  - blocking of a thread does not block the entire process
- Process/related to resource ownership
- Thread/related to program execution

Some operating systems distinguish the concepts of process and thread, the former related to resource ownership and the latter related to program execution. This approach may lead to improved efficiency and coding convenience. In a multithreaded system, multiple concurrent threads may be defined within a single process. This may be done using either user-level threads or kernel-level threads. User-level threads are unknown to the OS and are created and managed by a threads library that runs in the user space of a process. User-level threads are very efficient because a mode switch is not required to switch from one thread to another. However, only a single user-level thread within a process can execute at a time, and if one thread blocks, the entire process is blocked. Kernel-level threads are threads within a process that are maintained by the kernel. Because they are recognized by the kernel, multiple threads within the same process can execute in parallel on a multiprocessor and the blocking of a thread does not block the entire process. However, a mode switch is required to switch from one thread to another.