Threads

Dr. Ailbhe Gill

Textbook: Operating Systems: Internals and Design Principles, 9th Edition by William Stallings

Today's Lecture

- Processes and threads
- Types of threads
- Linux threads
- Multicore and multithreading

Learning Objectives

After studying this chapter, you should be able to:

- Understand the distinction between process and thread.
- Describe the basic design issues for threads.
- Explain the difference between user-level threads and kernel-level threads.

Processes and Threads

Processes and Threads

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

1. **Resource Ownership**
   - Process includes a virtual address space to hold the process image
   - 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**
   - Execution of a process follows an execution path (trace) through one or more programs
   - This execution may be interleaved with other processes
     - ➤ 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

Multithreading

Multithreading

Technique in which a process, executing an application, is divided into threads that can run concurrently

Thread

- Dispatchable unit of work
- Includes a processor context and its own data area for a stack
- Executes sequentially and is interruptible

Process (Definition in terms of threads)

- A collection of one or more threads and associated system resources

*By breaking a single application into multiple threads, a programmer has greater control over the modularity of the application and the timing of application-related events

Processes and Threads Summary

- The unit of dispatching is referred to as a **thread** or lightweight process
- The unit of resource ownership is referred to as a **process or task**
- **Multithreading** - The ability of an OS to support multiple, concurrent paths of execution within a single process

Single Threaded Approaches

Single-threaded Approach

A single thread of execution per process, in which the concept of a thread is not recognized.

- MS-DOS is an example of an OS that supports a single user process and a single thread.
- The left half of the figure depicts single-threaded approaches

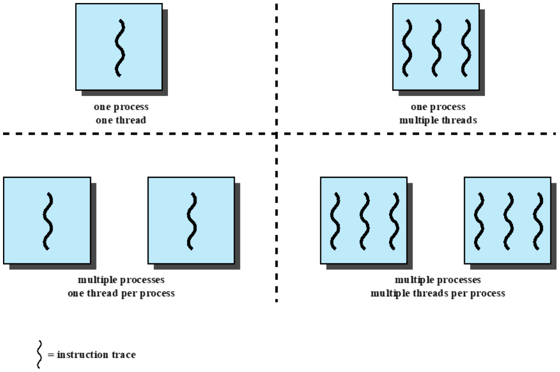
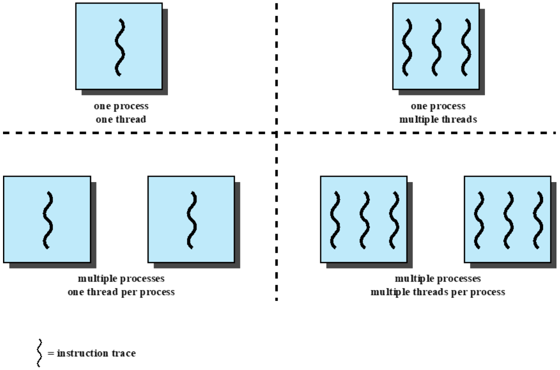


Figure 1: Threads and Processes

Multithreaded Approaches

- The right half of the figure depicts multithreaded approaches
- A Java run-time environment is an example of a system of one process with multiple threads



Process

Defined in a multithreaded environment 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
  - I/O resources (devices and channels)

One or More Threads in a Process

Each thread has:

- An execution state (Running, Ready, etc.)
- A saved thread context when not running
- An execution stack
- Some per-thread static storage for local variables
- Access to the memory and resources of its processes, shared with all other threads in that process

Single-Threaded and Multithreaded Process Models

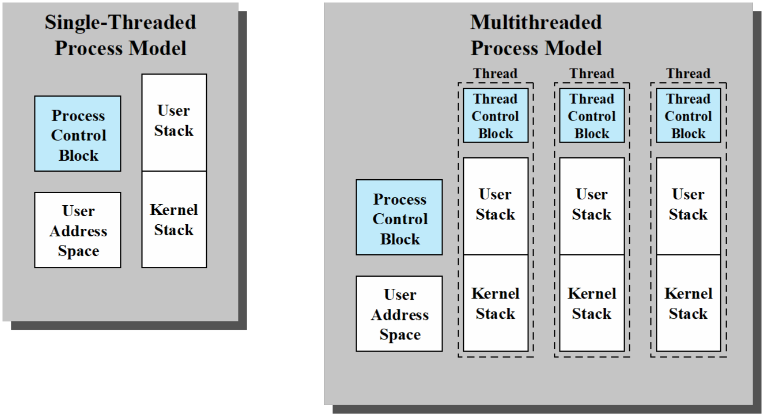


Figure 2: Single-Threaded and Multithreaded Process Models

Key Benefits of Threads

Key Benefits of Threads:

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Switching between two threads takes less time than switching between processes
- Threads enhance efficiency in communication between programs

Thread Use in a Single-User System

Four examples of the uses of threads in a single-user multiprocessing system:

1. **Foreground and background work**
    - e.g. in a spreadsheet program, one thread could display menus and read user input, while another thread executes user commands and updates the spreadsheet.
2. **Asynchronous processing**
    - e.g. as a protection against power failure, a thread can be created whose sole job is periodic backup
3. **Speed of execution**
    - a multithreaded process can compute one batch of data while reading the next batch from a device
4. **Modular program structure**
    - programs that involve a variety of activities or a variety of sources and destinations of input and output may be easier to design and implement using threads.

Threads

- In an OS that supports threads, scheduling and dispatching is done on a thread basis
- Most of the state information dealing with execution is maintained in thread-level data structures
    - Suspending a process involves suspending all threads of the process
    - Termination of a process terminates all threads within the process

Thread Execution States

The key states for a thread are:

- Running
- Ready
- Blocked

Thread operations associated with a change in thread state are:

- **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 it's info. 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.

Remote Procedure Call (RPC) Using Threads

Example

Consider 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 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.

This example is demonstrated in the figure on the next slide

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.

Remote Procedure Call (RPC) Using Threads

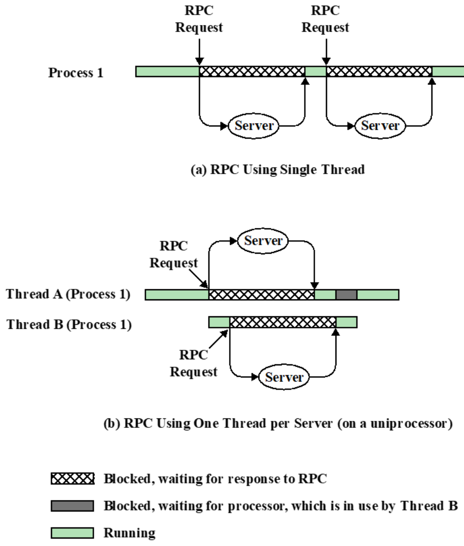

RPC Request
RPC Request
Process 1
Server
Server
(a) RPC Using Single Thread
RPC Request
Server
Thread A (Process 1)
Thread B (Process 1)
RPC Request
Server
(b) RPC Using One Thread per Server (on a uniprocessor)
Blocked, waiting for response to RPC
Blocked, waiting for processor, which is in use by Thread B
Running

Figure 3: Remote Procedure Call (RPC) Using Threads

Multithreading Example on a Uniprocessor

On a uniprocessor, multiprogramming enables the interleaving of multiple threads within multiple processes.

Example

For example, 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.

This example is demonstrated in the figure on the next slide

Multithreading Example on a Uniprocessor

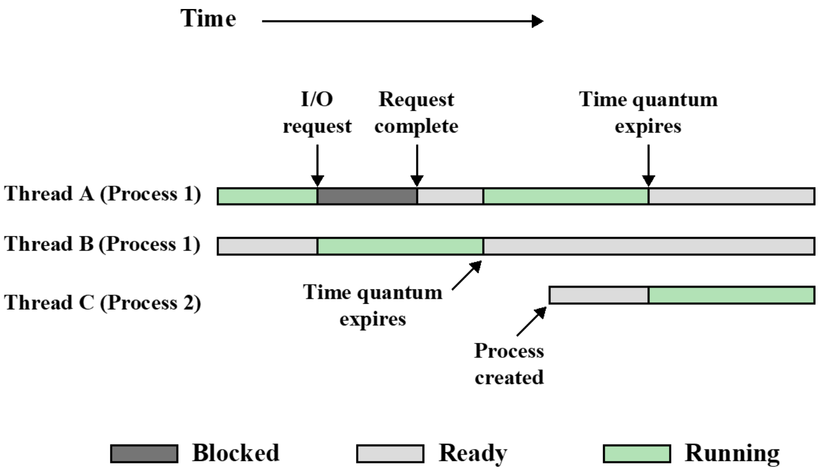


Figure 4: Multithreading Example on a Uniprocessor

Thread Synchronization

- All threads of a process share the same address space and other resources
- Any alteration of a resource by one thread affects the other threads in the same process
  - 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.
- Therefore, it is necessary to synchronize the activities of the various threads

Types of Threads

There are two broad categories of thread implementation:

- User Level Thread (ULT)
- Kernel level Thread (KLT)

User-Level Threads (ULTs)

- All thread management is done by the application
- The kernel is not aware of the existence of threads

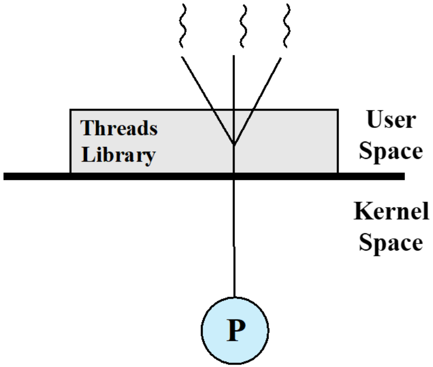


Figure 5: Pure user-level

Relationships Between User-Level Thread States and Process States

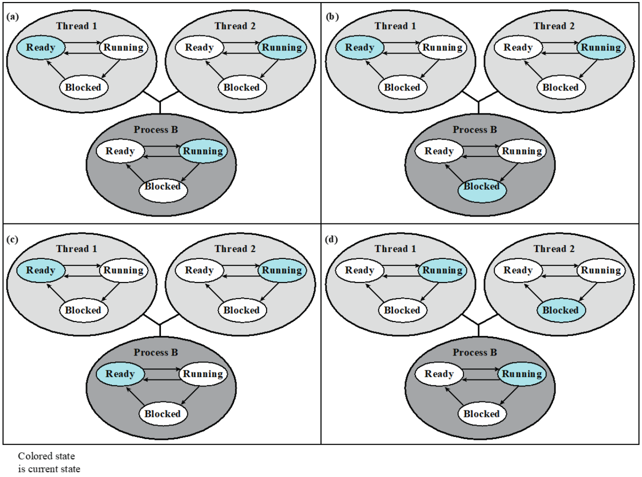


Figure 6: Examples of the Relationships Between User-Level Thread States and Process States

Advantages and Disadvantages of ULTs

Advantages of ULTs

- Thread switching does not require kernel mode privileges
  - ➤ Since all of the thread management data structures are within the user address space of a single process, saving the overhead of two mode switches.
- Scheduling can be application specific
  - ➤ The scheduling algorithm can be tailored to the application without disturbing the underlying OS scheduler.
- ULTs can run on any OS

Disadvantages of ULTs

- When a ULT executes a system call, not only is that thread blocked, but all of the threads within the process are blocked as well.
  - ➤ And in a typical OS many system calls are blocking.
- 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

Overcoming ULT Disadvantages

1. Jacketing

    Purpose is to convert a blocking system call into a non-blocking system call

2. Writing an application as multiple processes rather than multiple threads

    However, this approach eliminates the main advantage of threads

Kernel-Level Threads (KLTs)

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

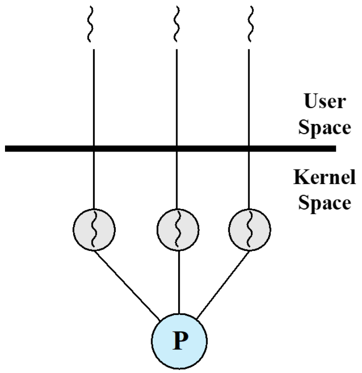


Figure 7: Pure kernel-level

Advantages and Disadvantages of KLTs

Advantages of KLTs

- 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 of the same process
- Kernel routines themselves can be multithreaded

Disadvantage of KLTs

- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

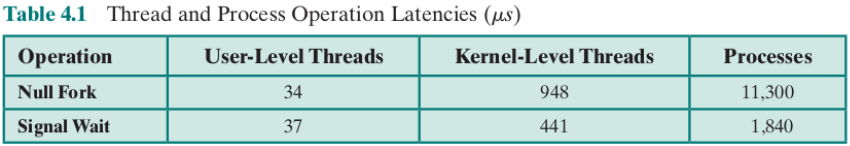


Table 4.1 Thread and Process Operation Latencies (μs)

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Combined Approaches

- Thread creation is done completely in the 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
- Combines the advantages of the pure ULT and KLT approaches while minimizing the disadvantages
- Solaris is a good example

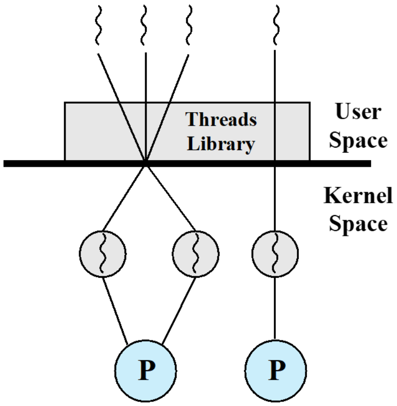


Figure 8: Combined

Relationship between Threads and Processes

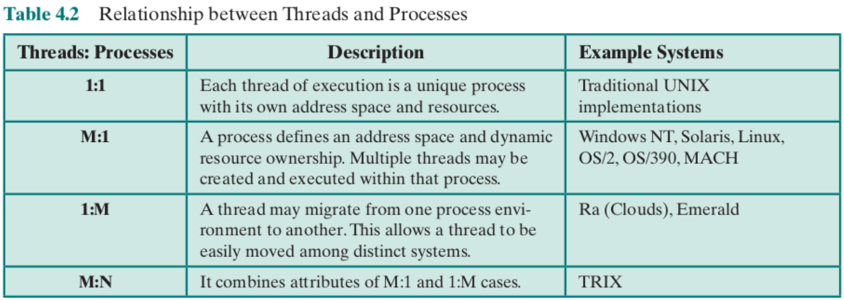


Table 4.2 Relationship between Threads and Processes

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	It combines attributes of M:1 and 1:M cases.	TRIX

Figure 9: Relationship between Threads and Processes

Linux Process/Thread Model

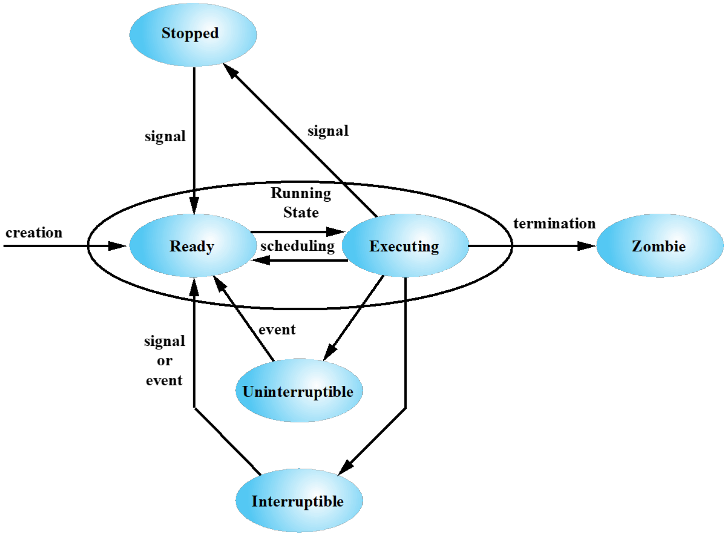


Figure 10: Linux Process/Thread Model

Execution States

Running:
Corresponds to two states:

- Running process is either executing or
- 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.

Linux Threads

- Linux does not recognize a distinction between threads and processes
- User-level threads are mapped into kernel-level processes
- A new process is created by copying the attributes of the current process
- The new process can be cloned so that it shares resources
- The clone() call creates separate stack spaces for each process

Multiprocessing, Multicore and Multithreading

Symmetric Multiprocessing (SMP)

Term that refers to a computer hardware architecture and also to the OS behavior that exploits that architecture

OS Behaviour with Symmetric Multiprocessing (SMP)

- The OS of an SMP schedules processes or threads across all of the processors
- The OS must provide tools and functions to exploit the parallelism in an SMP system
- An attractive feature of an SMP is that the existence of multiple processors is transparent to the user

Symmetric Multiprocessor OS Considerations

A multiprocessor OS must provide all the functionality of a multiprogramming system plus additional features to accommodate multiple processors

Key design issues:

Simultaneous concurrent processes or threads
Kernel routines need to be reentrant to allow several processors to execute the same kernel code simultaneously

Scheduling
Any processor may perform scheduling, which complicates the task of enforcing a scheduling policy

Synchronization
With multiple active processes having potential access to shared address spaces or shared I/O resources, care must be taken to provide effective synchronization

Memory management
The reuse of physical pages is the biggest problem of concern

Reliability and fault tolerance
The OS should provide graceful degradation in the face of processor failure

Multicore OS Considerations

The design challenge for a many-core multicore system is to efficiently harness the multicore processing power and intelligently manage the substantial on-chip resources efficiently

Potential for parallelism exists at three levels:

1. Hardware parallelism within each core processor, known as instruction level parallelism
2. Potential for multiprogramming and multithreaded execution within each processor
3. Potential for a single application to execute in concurrent processes or threads across multiple cores