CS415 Module 3 Part A - Threads Overview

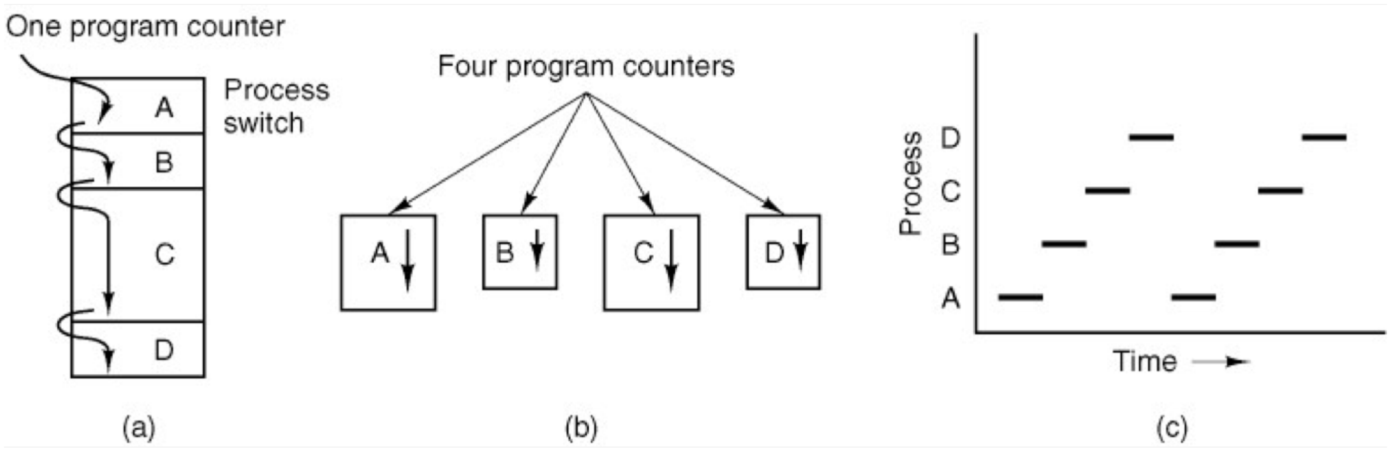
Athens State University

Contents

Processes do two things

- *Resource ownership*
  - Process includes a virtual address space to hold the process image
  - The OS provides a protection function to prevent unwanted interference between processes for resources
- *Scheduling/Execution*
  - A process follows an execution path that can be interleaved with other processes
  - A process has an execution state (Running, Ready,...) and a dispatching priority that the operating system uses to schedule the process

One Program Counter

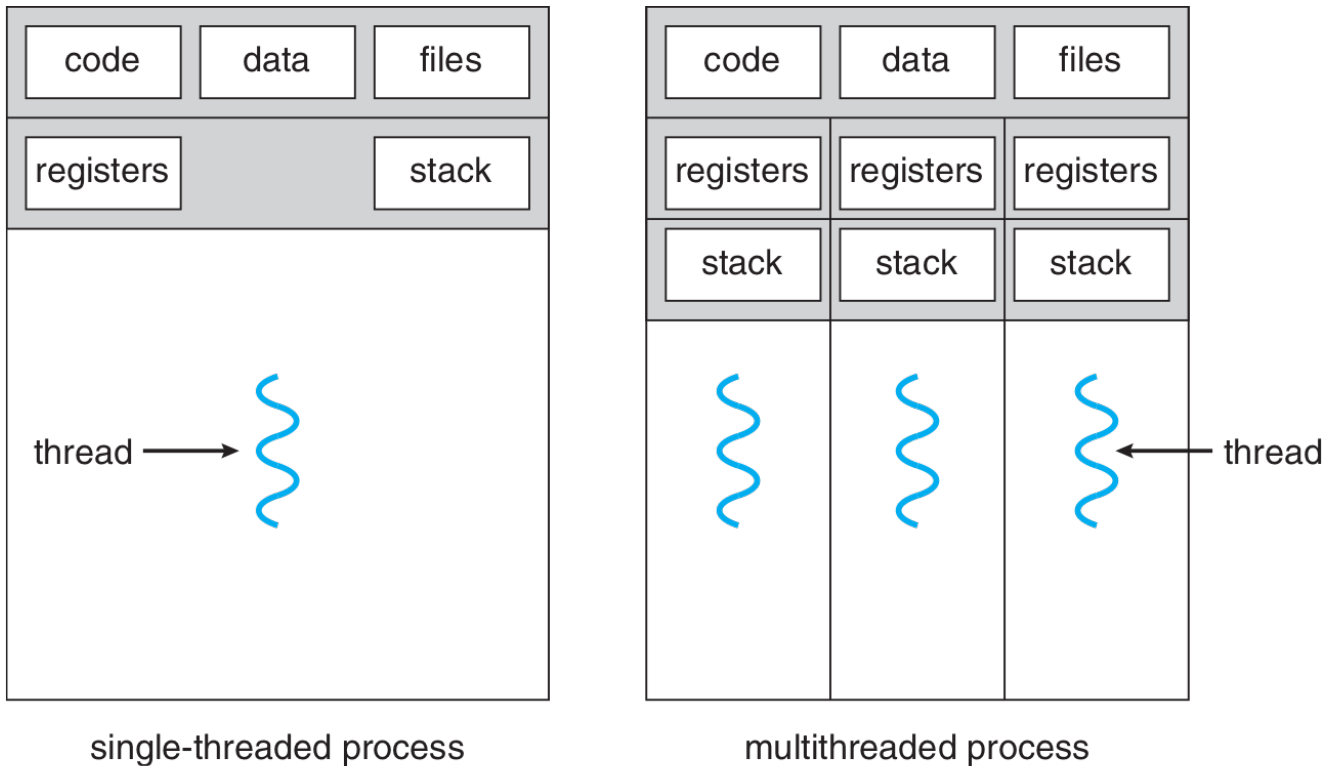


We have been operating under the assumption from Classical UNIX that a process operates with a single program counter. The multiprocessing aspects of scheduling in the OS switches between processes by swapping these program counts. The interleaving of processes over time is what makes it appear that the machine is doing more than one thing at a time.

Threads

- Consider what might transpire if we were to allow multiple program counters per process
  - Multiple locations can execute at once
    - **Thread**: multiple threads of control
  - Must have storage for thread details and multiple program counters in the PCB
- Started by considering the simple case of one thread per process
- Now we consider the case where a process can have many threads per process

Single Threaded vs. Multithreaded Processes



Revising Our Definition of Process

- A process is the unit of resource allocation and a unit of protection within the operating system
- A virtual address space that holds the process in memory
- Provide protected access to
  - Processors, other processes, files, I/O resources

One or More Threads In a Process

- Each thread has:
  - An execution state (running, ready, ...)
  - 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 process (shared by all threads)

How Threads Fit Into a Linux Process

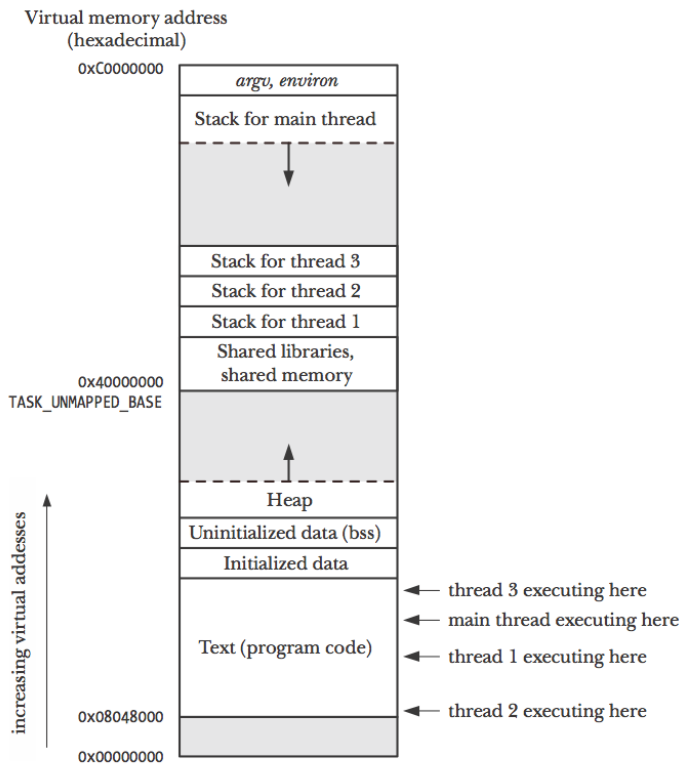


Figure 29-1: Four threads executing in a process (Linux/x86-32)

Benefits of Threads

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

Threads? But I'm The Single User of the System

- Foreground and background work: Dealing with input and output at the same time
- Asynchronous processing: implementing behind the scenes work
- Speed of execution: do part of a computation while another thread reads the data for the next part
- Modular program structure: Many design problems lend themselves to this type of structure

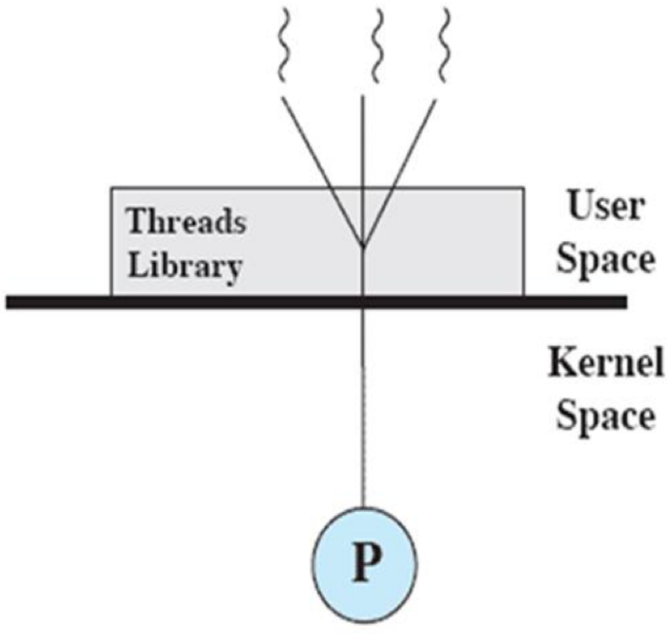
1 Threads As The Base Unit of Scheduling

The Base Unit of Scheduling

- In a thread-capable OS, scheduling and dispatching occurs on a thread basis
- Most of the state information dealing with execution is maintained in thread-level data structures
  - Suspending a process means we suspend all threads of the process
  - Terminating a process means that we terminate all threads within the process

1.1 Types of Threads

Types of Threads



(a) Pure user-level

- **User Level Thread** (ULT):
  - All thread management is done by the application
  - Kernel is unaware that threads exist

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

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

Advantages of ULTs

- ULTs can run on any OS

- Scheduling can be application specific
- Thread switching does not require kernel mode privileges

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.

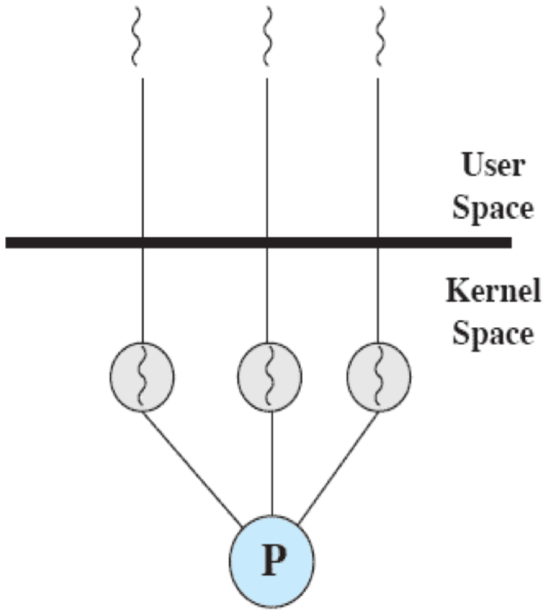
Disadvantages of ULTs

- 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 all of the threads within the process are blocked
- In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing

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 multi-programming can result in a significant speedup of the application, there are applications that would benefit from the ability to execute portions of code simultaneously.

Types of Threads



(b) Pure kernel-level

- **Kernel Level Thread** (kLT):
  - All thread management is done by the kernel
  - No thread management is done by the application

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. This figure depicts the pure KLT approach. The kernel maintains context information for the process as a whole and for individual threads within the process.

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 can be multi-threaded

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.

Disadvantage of KLTs

The transfer of control from one thread to another requires calling a system call with matching switch to privileged mode

Operation	ULTs	KLTs	Processes
Null fork	34	948	11,300
Signal wait	37	441	1840

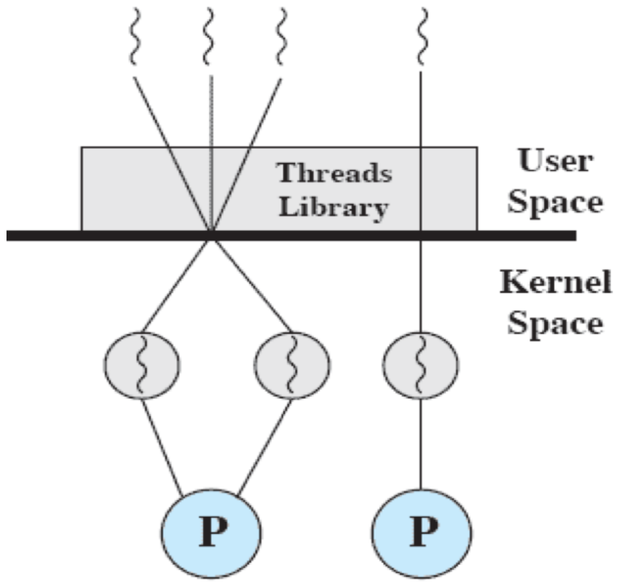
The principal disadvantage of the KLT approach compared to the ULT approach is that the transfer of control from one thread to another within the same process requires a mode switch to the kernel. To illustrate the differences, the table in the slide shows the results of measurements taken on a uniprocessor VAX computer running a UNIX-like OS. The two benchmarks are as follows:

1. Null Fork, the time to create, schedule, execute, and complete a process/thread that invokes the null procedure (i.e., the overhead of forking a process/thread)

2. Signal-Wait, the time for a process/thread to signal a waiting process/thread and then wait on a condition (i.e., the overhead of synchronizing two processes/threads together).

We see that there is an order of magnitude or more of difference between ULTs and KLTs and similarly between KLTs and processes.

Combined Approaches



(c) Combined

- Thread creation is done in the user space
- Bulk of scheduling and synchronization of threads occurs in user space
- Each user thread gets mapped to one or more threads in the kernel

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.

Many implementations of the POSIX `pthreads` standard use this approach (Linux, macOS).

2 Thread Implementation Issues

Semantics of `fork()` and `exec()` System Calls

- Does `form()` duplicate only the calling thread or all threads?
  - Some versions of UNIX have two versions of `fork()`
- The `exec()` usually works as in the single-threaded case: replace the running process including all threads

Signal Handling

- Where should a signal be delivered for a multi-threaded process?
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

Thread-Local Storage

- **Thread-Local Storage**: portion of process memory where a thread can store data that is static to that thread
- Different that both local variables and global storage
  - Static data unique to each thread
  - Unlike local variables, preserved from function call to function call
  - Not visible to other threads

3 Key Points

Key Points

- Process: Unit of resource allocation
- Thread: Unit of execution
- A process can one many threads
- Different ways to implmement
  - User-Level Threads
  - Kernel-Level Threads
  - Mix of both

- Threading issues
  - What happens to `fork()` and `exec()`?
  - Signals?
  - Thread Local Storage?