Study Set for Lecture 05: Threads

Due Sep 28 at 11:59pm **Points** 10 **Questions** 8
Available Sep 22 at 12pm - Dec 8 at 8pm 3 months **Time Limit** None
Allowed Attempts Unlimited

Instructions

Here are the examples from the lecture: **lect05code.zip**. Download the file, unzip, and then compile and experiment with the examples.

Review lecture notes from **lect05_Threads.pdf**.

Then answer the questions from this study set and submit them to gain access to the further part of the course.

Take the Quiz Again

Attempt History

	Attempt	Time	Score
LATEST	Attempt 1	47 minutes	10 out of 10

ⓘ Correct answers are hidden.

Score for this attempt: **10** out of 10
Submitted Sep 22 at 2:44pm
This attempt took 47 minutes.

Question 1 — 0 / 0 pts

Describe with details the differences between a process and a thread? Does every process have a thread? What's the minimum and maximum threads per process?
Can a thread span multiple processes?

Your Answer:

Process
Each process provides the resources needed to **execute** a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.

Thread
A thread is an entity within a process that can be **scheduled for execution**. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The thread context includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process. Threads can also have their own security context, which can be used for impersonating clients.

Does every process have a thread? What's the minimum and maximum threads per process?

Threads exist within a **process** — **every process has** at least one. **Threads** share the **process's** resources, including memory and open files.

Can a thread span multiple processes?

Multiple threads can exist within one **process**, executing concurrently and sharing resources such as memory, while different **processes** do not share these resources.

You'd prefer multiple threads over multiple processes for two reasons:

1. Inter-thread communication (sharing data etc.) is significantly simpler to program than inter-process communication.
2. Context switches between threads are faster than between processes. That is, it's quicker for the OS to stop one thread and start running another than do the same with two processes.

Example:

Applications with GUIs typically use one thread for the GUI and others for background computation. The spellchecker in MS Office, for example, is a separate thread from the one running the Office user interface. In such applications, using multiple processes instead would result in slower performance and code that's tough to write and maintain.

Question 2

0 / 0 pts

Why use threads?

Your Answer:

Light Weight:

- When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.

- For example, the following table compares timing results for the **`fork()`** subroutine and the **`pthread_create()`** subroutine. Timings reflect 50,000 process/thread creations, were performed with the `time` utility, and units are in seconds, no optimization flags.

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

Efficient Communications/Data Exchange:

- The primary motivation for considering the use of Pthreads in a high performance computing environment is to achieve optimum performance. In particular, if an application is using MPI for on-node communications, there is a potential that performance could be improved by using Pthreads instead.

- MPI libraries usually implement on-node task communication via shared memory, which involves at least one memory copy operation (process to process).

- For Pthreads there is no intermediate memory copy required because threads share the same address space within a single process. There is no data transfer, per se. It can be as efficient as simply passing a pointer.

- In the worst case scenario, Pthread communications become more of a cache-to-CPU or memory-to-CPU bandwidth issue. These speeds are much higher than MPI shared memory communications.

- For example: some local comparisons, past and present, are shown below

Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
Intel 2.6 GHz Xeon E5-2670	4.5	51.2
Intel 2.8 GHz Xeon 5660	5.6	32
AMD 2.3 GHz Opteron	1.8	5.3
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 2.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

Other Common Reasons:

- Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways:
  - Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads.
  - Priority/real-time scheduling: tasks which are more important can be scheduled to supersede or interrupt lower priority tasks.
  - Asynchronous event handling: tasks which service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.

- A perfect example is the typical web browser, where many interleaved tasks can be happening at the same time, and where tasks can vary in priority.

Question 3

0 / 0 pts

Which of the following components of program state are shared across threads in a multithreaded process, and which ones are not?

- Register values

- Heap memory (dynamic allocation)
- Global variables
- Stack memory(call stack)

Explain with details why some elements on the list are shared while others cannot be shared.

Your Answer:

Multi threaded process share :

Heap memory

Global variables

Each thread has separate set of :

Register values

Stack memory

Question 4 **0 / 0 pts**

What are the differences between user-level threads and kernel-level threads?
Discuss the ways in which a user thread library can be implemented using kernel threads?

Your Answer:

User - Level Threads

The user-level threads are implemented by users and the kernel is not aware of the existence of these threads. It handles them as if they were single-threaded processes. User-level threads are small and much faster than kernel level threads. They are represented by a program counter(PC), stack, registers and a small process control block. Also, there is no kernel involvement in synchronization for user-level threads.

Advantages of User-Level Threads

Some of the advantages of user-level threads are as follows −

- User-level threads are easier and faster to create than kernel-level threads. They can also be more easily managed.
- User-level threads can be run on any operating system.
- There are no kernel mode privileges required for thread switching in user-level threads.

Disadvantages of User-Level Threads

Some of the disadvantages of user-level threads are as follows −

- Multithreaded applications in user-level threads cannot use multiprocessing to their advantage.
- The entire process is blocked if one user-level thread performs blocking operation.

Kernel-Level Threads

Kernel-level threads are handled by the operating system directly and the thread management is done by the kernel. The context information for the process as well as the process threads is all managed by the kernel. Because of this, kernel-level threads are slower than user-level threads.

Advantages of Kernel-Level Threads

Some of the advantages of kernel-level threads are as follows −

- Multiple threads of the same process can be scheduled on different processors in kernel-level threads.
- The kernel routines can also be multithreaded.
- If a kernel-level thread is blocked, another thread of the same process can be scheduled by the kernel.

Disadvantages of Kernel-Level Threads

Some of the disadvantages of kernel-level threads are as follows −

- A mode switch to kernel mode is required to transfer control from one thread to another in a process.
- Kernel-level threads are slower to create as well as manage as compared to user-level threads.

User threads are implemented with kernel threads by mapping user threads to kernel threads using one of the multi-threaded models:

- many-to-one
- one-to-one
- many-to-many

Question 5

0 / 0 pts

What is a signal? Discuss the issues with signals targeted at multithreaded processes. What options do operating system designers have? How is the problem solved by POSIX?

Your Answer:

Signals are software interrupts sent to a program to indicate that an important event has occurred. The events can vary from user requests to illegal memory access errors. Some signals, such as the interrupt signal, indicate that a user has asked the program to do something that is not in the usual flow of control.

Signals are similar to **interrupts** (https://en.wikipedia.org/wiki/Interrupt), the difference being that interrupts are mediated by the processor and handled by the **kernel** (https://en.wikipedia.org/wiki/Kernel_(operating_system)) while signals are mediated by the kernel (possibly via system calls) and handled by processes. The kernel may pass an interrupt as a signal to the process that caused it (typical examples are **SIGSEGV** (https://en.wikipedia.org/wiki/SIGSEGV), **SIGBUS** (https://en.wikipedia.org/wiki/SIGBUS), **SIGILL** (https://en.wikipedia.org/wiki/Signal_(IPC)#SIGILL) and **SIGFPE** (https://en.wikipedia.org/wiki/Signal_(IPC)#SIGFPE)).

There is a issue when mixing signals and threads. Operating system designers have avoided this by mask signals before the threads are created.

In POSIX, signals target process rather than a thread. aarbitrary thread is selected to handle the signal.

Question 6

0 / 0 pts

What will happen if a Linux multi-threaded process is cloned with fork()?

What will happen if a Linux multi-threaded process morphs into another program using one of the exec() functions?

Your Answer:

The **fork( )** system call creates an exact duplicate of the address space from which it is called, resulting in two address spaces executing the same code. Problems can occur if the forking address space has multiple threads executing at the time of the **fork( )**. When multithreading is a result of library invocation, threads are not necessarily aware of each other's presence, purpose, actions, and so on. Suppose that one of the other threads (any thread other than the one doing the **fork( )**) has the job of deducting money from your checking account. Clearly, you do not want this to happen twice as a result of some other thread's decision to call **fork( )**.

One solution to the problem of calling **fork( )** in a multithreaded environment exists. (Note that this method will not work for server application code or any other application code that is invoked by a callback from a library.) Before an application performs a **fork( )** followed by something other than **exec( )**, it must cancel all of the other threads. After it joins the canceled threads, it can safely **fork( )** because it is the only thread in existence. This means that libraries that create threads must establish cancel handlers that propagate the cancel to the created threads and join them. The application should save enough state so that the threads can be recreated and restarted after the **fork( )** processing completes.

Does fork() duplicate only the calling thread or all threads?

- some systems have two versions of fork()

● in Linux, only the calling thread is cloned

● i.e., the process context with the program counter, stack, and registers specific to the executing thread

What about exec()?

● usually, it replaces whole process

● including overwriting the threads

● the new process will have only one (main) thread

Question 7

0 / 0 pts

Explain what a deferred thread cancellation is and how it enhances program reliability.

Your Answer:

Terminating a thread before it has completed is called **Thread cancellation**. For an example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled. Another situation might be occurred when a user presses a button on a web browser that stops a web page from loading any further. Often, using several threads a web page loads — each image is loaded in a separate thread. When the stop button is pressed by a user on the browser, all threads loading the page are canceled. A thread which is to be cancelled is often referred to as the target thread. Cancellation of a target thread may occur in two different cases −

- **Asynchronous cancellation** − One thread terminates immediately the target thread.
- **Deferred cancellation** − The target thread checks periodically whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

The definition above is the main reason why it enhances program reliability

Question 8

10 / 10 pts

I have submitted answers to all questions in this study set.

- (selected) True
- False

Quiz Score: **10** out of 10