

CS420: Operating Systems

Threading Issues

James Moscola

Department of Engineering & Computer Science

York College of Pennsylvania



Threading Issues

- **There are a variety of issues to consider with multithreaded programming**
 - Semantics of `fork()` and `exec()` system calls
 - Thread cancellation
 - Asynchronous or deferred
 - Signal handling
 - Synchronous and asynchronous
 - Thread pooling
 - Thread-specific data
 - Create facility needed for data private to thread

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

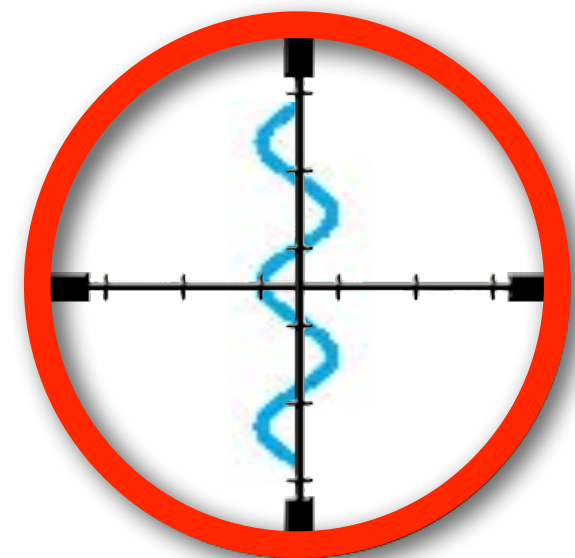
- Recall that when `fork()` is called, a separate, duplicate process is created
- How should `fork()` behave in a multithreaded program?
 - Should all threads be duplicated?
 - Should only the thread that made the call to `fork()` be duplicated?
- In some systems, different versions of `fork()` exist depending on the desired behavior
 - Some UNIX systems have `fork1()` and `forkall()`
 - `fork1()` only duplicates the calling thread
 - `forkall()` duplicates all of the threads in a process
 - In a POSIX-compliant system, `fork()` behaves the same as `fork1()`

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

- **The `exec()` system call continues to behave as expected**
 - Replaces the entire process that called it, including all threads
- **If planning to call `exec()` after `fork()`, then there is no need to duplicate all of the threads in the calling process**
 - All threads in the child process will be terminated when `exec()` is called
 - Use `fork1()`, rather than `forkall()` if using in conjunction with `exec()`

Thread Cancellation

- **Thread cancellation** is the act of terminating a thread before it has completed
 - Example - clicking the stop button on your web browser will stop the thread that is rendering the web page
- The thread to be cancelled is called the **target thread**
- Threads can be cancelled in a couple of ways
 - **Asynchronous cancellation** terminates the target thread immediately
 - Thread may be in the middle of writing data ... not so good
 - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
 - Allows thread to terminate itself in an orderly fashion



Signal Handling

- **Signals** are used in **UNIX** systems to notify a process that a particular event has occurred
 - CTRL-C is an example of an **asynchronous signal** that might be sent to a process
 - An asynchronous signal is one that is generated from outside the process that receives it
 - Divide by 0 is an example of a **synchronous signal** that might be sent to a process
 - A synchronous signal is delivered to the same process that caused the signal to occur
- **All signals follow the same basic pattern:**
 - A signal is generated by particular event
 - The signal is delivered to a process
 - The signal is handled by a **signal handler** (all signals are handled exactly once)

Signal Handling

- **Signal handling is straightforward in a single-threaded process**
 - The one (and only) thread in the process receives and handles the signal
- **In a multithreaded program, where should signals be delivered?**
 - Options:
 - (1) Deliver the signal to the thread to which the signal applies
 - (2) Deliver the signal to every thread in the process
 - (3) Deliver the signal only to certain threads in the process
 - (4) Assign a specific thread to receive all signals for the process

Signal Handling

- **Option 1 - Deliver the signal to the thread to which the signal applies**
 - Most likely option when handling synchronous signals (e.g. only the thread that attempts to divide by zero needs to know of the error)
- **Option 2 - Deliver the signal to every thread in the process**
 - Likely to be used in the event that the process is being terminated (e.g. a CTRL-C is sent to terminate the process, all threads need to receive this signal and terminate)

Thread Pools

- **In applications where threads are repeatedly being created/destroyed thread pools might provide a performance benefit**
 - Example: A server that spawns a new thread each time a client connects to the system and discards that thread when the client disconnects
- **A thread pool is a group of threads that have been pre-created and are available to do work as needed**
 - Threads may be created when the process starts
 - A thread may be kept in a queue until it is needed
 - After a thread finishes, it is placed back into a queue until it is needed again
 - Avoids the extra time needed to spawn new threads when they're needed

Thread Pools

- **Advantages of thread pools:**

- Typically faster to service a request with an existing thread than create a new thread (performance benefit)
- Bounds the number of threads in a process
 - The only threads available are those in the thread pool
 - If the thread pool is empty, then the process must wait for a thread to re-enter the pool before it can assign work to a thread
 - Without a bound on the number of threads in a process, it is possible for a process to create so many threads that all of the system resources are exhausted

Thread-Specific Data

- **Thread-specific data** - in some applications it may be useful for each thread to have its own copy of data
 - May also be referred to as **Thread-local storage** or **Thread-static variables**
 - The `errno` variable is thread-specific

```
// In C#  
class FooBar {  
    [ThreadStatic] static int foo;  
}
```

```
// In Java  
private static ThreadLocal<Integer> threadLocalInt =  
    new ThreadLocal<Integer>();
```

```
// In a POSIX-compliant system  
// see pthread_key_create()  
//      pthread_setspecific()  
//      pthread_getspecific()
```

Thread Examples - Windows XP

- **Implements threads using the one-to-one thread model**
- **Also implements a **fiber** that uses a many-to-many model**
 - A **fiber** is a unit of execution that must be manually scheduled by the application
- **Each thread contains**
 - A thread id
 - Register set
 - Separate user and kernel stacks
 - Private data storage area

Linux Threads

- Linux oftentimes uses the term **task** rather than **process** or **thread**
- Thread creation is done through the **clone()** system call
 - The **clone()** can create either 'threads' or 'processes' depending on the options passed to **clone()**
 - The options passed to clone determine how much sharing is taking place between the parent and the child
 - A 'process' can still be created using the **fork()** system call
 - Provided a specific set of options and the **clone()** and **fork()** systems calls behave identically

Linux Threads (Cont.)

- The following table shows the various flags that can be passed to clone to determine how much sharing is taking place between the parent and the child

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.