Computer Network Programming

UNIX Threads

Dr. Sam Hsu
Computer Science & Engineering
Florida Atlantic University

UNIX Threads

- Motivation for Threads
- Thread Resources
- Thread Implementations
- Unix Threads
- POSIX Threads
- Thread Operations
- MT Safe Functions
- Forking Process in Threads
- Dealing with Locks in Threads

What Is a Thread?

- A thread is an execution stream within a process with its own stack, local variables, and program counter.
  - There may be more than one execution stream in a process.
- A thread shares resources with other threads executing in the same address space.
- A multi-threaded process can perform several tasks concurrently.

Motivation for Threads

- Fork is expensive.
- IPC is required for data exchange between parent and child.
- A thread is lightweight.
  - Thread creation is about 10-100 times faster than process creation.

All Threads Are Siblings

- All threads executing in the same process address space are called ***sibling*** threads.
- A thread can create as many threads as it pleases. However there is no relationship among them after creation.
  - No parent/child relationship between the creator thread and the createe thread.
  - They are peers in the same process.

Threads Resources (1/3)

- All threads within a process share:
  - Text segment
  - Data segment
  - Heap
  - Open files
  - Signal handlers
  - Current working directory
  - UID and GID

Threads Resources (2/3)

- However, each thread has its own (known as *thread-private*):
  - TID
  - Set of registers, including program counter and stack pointer
  - Stack (for local variables and return addresses)
  - ***errno*** variable
  - Signal mask
  - Priority

Threads Resources (3/3)

- A thread also has its *thread-specific* data (TSD).
  - Data structures of TSD depending on applications.

Thread Implementations

- Threads may be implemented as:
  - A kernel-level abstraction,
    - Also called kernel-supported threads
  - A user-level abstraction, or
  - A combination of the two.

Kernel-level Abstraction

- Kernel-supported threads require kernel data structures.
  - The OS is aware of each thread.
  - Kernel threads are required to support user-level threads.
  - The kernel must contain system-level code for each specified thread function.
  - This approach is good for supporting parallelism with multiple threads running on multiple processors.

User-level Abstraction

- A user-level abstraction is represented by data structures within a process's own address space.
  - It does not require direct support from the OS.
    - It runs on top of the OS and is transparent to it.
    - The OS maintains a runtime system to manage thread activities.
- Has the potential to execute only when associated with a kernel process.
  - User-level threads are *multiplexed* onto a kernel process for execution.
- In general, user-levels threads are designed to share resources with other threads within their own process space running on a single processor.

A Combination of Two

- The combined model offers both *multiplexed* and *bound* user-level threads.
  - A user-level thread is ***bound*** (one-to-one mapping) to a kernel thread.
    - A kernel thread is also known as a ***lightweight process*** (LWP).
    - LWPs are also called ***virtual processor***s by some authors.
  - Or, multiple user-level threads are *multiplexed* onto a kernel LWP(s).
    - The number of kernel LWPs available for the multiplexed user threads may be either implementation-dependent, or tunable by the application.

UNIX Threads

- UNIX threads implementation is system dependent.
  - The UNIX Threads Interface does not define the implementation.
  - However, it does provide for both *multiplexed* and *bound* threads.
    - Both implementations can support exactly the same APIs.
- The relations between user-level threads and kernel LWPs may be:
  - 1-to-1
  - M-to-1 (Many-to-One)
  - M-to-N (Many-to-Many)

POSIX Threads

- POSIX threads are portable among UNIX systems that are POSIX-compliant.
  - It is known as ***IEEE 1003.1c***
  - AKA ***Pthreads***

Other Threads Implementations

- Some other threads implementations:
  - Light Weight Kernel Threads (LWKT) in BSDs
  - Native POSIX Thread Library (NPTL) for Linux
  - Win32 Threads
  - GNU Portable Threads
  - Mac OS Threads
  - Solaris Threads
  - Java Threads

Thread Operations

- Thread creation.
- Thread execution.
- Thread termination.
- Thread management.
- Thread synchronization.
- Thread scheduling.
- TSD manipulation.
- Thread errono handling.

Thread Creation (1/2)

- A thread is created by another thread.
- Once a thread is created, it
  - Has its own set of attributes.
    - Either given by the initiating thread or system default.
  - Has its own execution stack.

Thread Creation (2/2)

- Inherits its signal mask and scheduling priority from the calling thread.
- Does not inherit any pending signals.
- Does not inherit any TSD data.

- Primitivies:
  - int pthread_create()

Thread Execution (1/2)

- Threads of a process execute in a single UNIX process environment.
  - All resources available in this environment are shared by the sibling threads and one or more thread execution environments.
    - A thread execution environment contains the scheduling policy, priority, and the disposition of signals for a thread.

Thread Execution (2/2)

- In terms of execution, a process has one or more kernel LWPs that provide the execution vehicle for the threads.
  - The threads of a process are either multiplexed onto an available kernel LWP, or are *bound* (mapped one-to-one) to a specific LWP for execution.
    - In case a kernel call blocks, the corresponding user level thread(s) also blocks.

Thread Termination (1/2)

- A thread terminates when either its execution reaches the last statement in the thread, is signaled to quit or it exits voluntarily (a call to pthread_exit()).
  - When a thread exits, normally a sibling can request the exit status of the terminated thread.
- However, all threads terminate if one thread calls exit(), or execution falls off the bottom of main().
  - Use pthread_exit() in main() to avoid premature termination of the program.

Thread Termination (2/2)

- A UNIX process will terminate when its last thread exits.
- Primitivies:
  - void pthread_exit()
  - int pthread_cancel()

Thread Management (1/2)

- A thread can be either *detached* or *nondetached*.
  - A detached thread will clean up after itself upon termination.
    - Resources to return for reuse include its thread structure, TSD array, stack, and heap.
  - A nondetached thread will clean up after itself only after it has been *joined*.
    - Nondetached threads are the default.

Thread Management (2/2)

- Primitivies:
  - int pthread_join()
  - int pthread_detach()

Thread Synchronization (1/6)

- Mutual exclusion locks
  - A mutual exclusion (*mutex*) lock indicates that the use of a shared resource is mutually exclusive between competing threads.
    - To use a resource, a thread must first **lock** the mutex guarding the resource.
    - When the use is complete, the thread must **unlock** the mutex, thereby permitting other threads to use the resource.

Thread Synchronization (2/6)

- The section of code manipulating the shared resource is often referred to as a ***critical section***.
  - The integrity of the shared resource is ensured only if all threads using the resource follow the **lock-unlock** convention.

Thread Synchronization (3/6)

- Condition variables
  - A convenient mechanism to notify interested threads of an event.
  - How it works:
    - A thread obtains a mutex (a condition variable always has an associated mutex) and evaluates the condition under the mutex's protection.
    - If the condition is true, the thread completes its task, releasing the mutex when appropriate.

Thread Synchronization (4/6)

- If the condition is false, the mutex is released by the system and the thread goes to sleep on the condition variable.
- When the value of the condition variable is changed by another thread, it can wake up the thread(s) sleeping on the variable.
  - The awakened thread will reevaluate the condition variable again.

- A typical example of using a condition variable would be for a thread to suspend its execution until a message is received.

Thread Synchronization (5/6)

- Barriers
  - A mechanism for a set of threads to sync up.
    - A barrier is initialized to the number of threads to be using it. When a thread reaches it, its execution is suspended until all of the participating threads arrive at the barrier.
    - At this point, all threads are permitted to resume execution.
  - A barrier provides a rendezvous point for threads cooperating in the barrier.

Thread Synchronization (6/6)

- Primitivies:
  - int int pthread_mutexattr_init()
  - int pthread_mutexattr_setpshared()
  - int pthread_mutex_init()
  - int pthread_mutex_lock()
  - int pthread_mutex_unlock()
  - int pthread_mutex_trylock()
  - int pthread_mutex_destroy()

Thread Scheduling (1/4)

- Common scheduling policies:
  - First-come-first-serve
  - Shortest-job first
  - Priority-based
  - Round-robin

Thread Scheduling (2/4)

- Global and local scheduling
  - If a thread is *bound* (one-to-one with a LWP), its scheduling is determined by the kernel scheduling algorithms.
    - It is known as *global scheduling*.
    - Its scheduling class is said to have a *System Contention Scope*.
  - If a thread is *unbound*, the thread library has full control which thread will be scheduled on an LWP.
    - It is known as *local scheduling*.
    - It is said to have a *Process Contention Scope*.

Thread Scheduling (3/4)

- Scheduling of threads involves three factors:
  - Contention scope
  - Scheduling policy
  - Thread priority
- Note: Most thread implementations today use a priority-based, preemptive (a thread can be removed by a thread of higher priority), non-time slicing algorithm to schedule thread activities. It is also recommended that you, as a programmer, to spend little time thinking about issues of thread scheduling.

Thread Scheduling (4/4)

- Primitives:
  - void pthread_setschedparam()
  - void pthread_getschedparam()

TSD Manipulation (1/3)

- TSD (Thread Specific Data) provides a mechanism of handling global data in a thread.
  - TSD is globally accessible to all functions in a thread but still unique to the thread.
    - A TSD value is referenced using a thread specific pointer and an associated key.
  - To make use of TSD, a thread must create and bind (associate) the key with the TSD data.
    - The TSD keys in a thread are global to all functions in the thread.

TSD Manipulation (2/3)

- A destructor function for cleanup can be specified at the time of creating a TSD key.
  - Dynamically allocated memory in TSD needs to be explicitly freed in the destructor.
- To ensure data integrity, mutual exclusion is desired for accessing TSD.

TSD Manipulation (3/3)

- Primitives:
  - pthread_key_create()
  - pthread_key_delete()
  - pthread_getspecific()
  - pthread_setspecific()

Thread errno Handling

- In general, pthread functions do not set the standard UNIX errno variable. When an error occurs, the errno value is the return value of the function.
  - On may need to use a variable to save the return value. Therefore, each thread has, in effect, its own errno variable.

Thread Attributes (1/3)

- Attributes defined and their values:
  - contentionscope
    - PTHREAD_SCOPE_PROCESS
    - PTHREAD_SCOPE_SYSTEM
  - detachstate
    - PTHREAD_CREATE_JOINABLE
    - PTHREAD_CREATE_DETACHED
  - stackaddr
    - NULL
    - *nnn* (valid address)

Thread Attributes (2/3)

- stacksize
  - NULL
  - *nnn* (valid address)
- policy
  - SCHED_OTHER
  - SCHED_FIFO
  - SCHED_RR
- inheritsched
  - PTHREAD_EXPLICIT_SCHED

Thread Attributes (3/3)

- Policy
  - The data type for thread priorities is int sched_priority. It is defined in the sched_param structure found in the header file <sched.h>. However, POSIX gives no advice on how to use the priority levels provided.

Getting/Setting Attributes (1/2)

- Primitives:
  - int pthread_attr_init()
  - int pthread_attr_getscope()
  - int pthread_attr_setscope()
  - int pthread_attr_getdetachstate()
  - int pthread_attr_setdetachstate()
  - int pthread_attr_getstackaddr()
  - int pthread_attr_getstackaddr()
  - int pthread_attr_setstackaddr()

Getting/Setting Attributes (2/2)

- int pthread_attr_getstacksize()
- int pthread_attr_setstacksize()
- int pthread_attr_getschedparam()
- int pthread_attr_setschedparam()
- int pthread_attr_getschedpolicy()
- int pthread_attr_setschedpolicy()
- int pthread_attr_getinheritsched()
- int pthread_attr_setinheritsched()

Signals in Threads

- Two types of signals in threads
  - Synchronous: Signals delivered to the thread that generated the exception.
    - Ex: SIGFPE (divide by zero)
  - Asynchronous: Signals delivered to a non-specific or non-offending thread.
    - Ex: SIGHUP (hang up)

Different Uses of Signals

- Three applications
  - Error reporting
  - Situation reporting
  - Interruption
- Methods of handling signals
  - These 3 different situations are mixed together in single threaded processes, and handled indifferently.
  - In multithreaded programming, the distinctions become important. They are handled differently.

Signal Delivery

- For error reporting, the thread library guarantees that a signal will be delivered to the offending thread.
- For situation reporting, the thread library decides which thread should receive a specific signal and arranges for the execution of the associated signal handler.
- For interruption, there is no general method of ensuring that a signal gets delivered to the intended thread.
  - A dirty fix, mask out the signal on all but one thread.

Process-wide Signal Handlers

- Be aware that signal handlers are process-wide.
  - Only one set of signal handlers per process.
  - No thread-specific signal handlers.
- However, each thread can have its own signal mask.

Rationale for One Set

- Rationale of having one set of signal handlers for all threads in a process:
  - Signals are used for asynchronous events. However, multithreading is itself asynchronous enough.
    - A multithreaded program can simply spawn a new thread to wait for an event of interest.

Handling Signals in Threads

- To handle signals effectively in threads, a programmer,
  - Needs to be concerned of the most is probably the thread signal mask.
  - Employs a simple solution by designating one thread to take care of signals in a process.
    - Masking out all asynchronous signals on all threads but one, and let this one handles the asynchronous signals of the process.

Singal Primitives

- Primitives:
  - int pthread_kill()
  - int pthread_sigmask()
  - int sigwait()
  - int sigtimedwait()
  - int sigwaitinfo()

MT Safe Functions

- MT safe means that a function can be called from multiple threads concurrently.
  - The function can be a C library function, a system call, etc.
- To be MT safe, a function must:
  - Lock any shared data it uses.
  - Call only other MT safe functions.
  - Use the correct error number (errno).
    - Be aware that errno is process-wide.
- Note: It is OK to use an MT unsafe function in an MT program, but just don't call it concurrently.

Forking Processes in Threads

- There are two semantics in defining fork():
  - Only the calling thread is replicated (fork1()).
    - POSIX uses this one.
  - All threads and LWPs are replicated (*forkall()*).

Dealing with Locks in Threads

- Be cautious about touching any locks that might be held by threads that do not exist in the child process.
  - One may arrive at a deadlock.
- Suggestion: Have the child process call exec() immediately after the fork1() call to avoid a potential deadlock.
- Also, POSIX defines pthread_atfork() to help solve the *deadlock-in-the-child* problem.

Some References

- http://www.llnl.gov/computing/tutorials/pthreads/
- http://math.arizona.edu/~swig/documentation/pthreads
- http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html#SYNCHRONIZATION /
- http://liinwww.ira.uka.de/bibliography/Os/threads.html
- Chapter 26, UNIX Network Programming, Volume 1, 3rd ed., W. Richard Stevens.