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 multi-threaded process can perform several tasks concurrently.
  • A thread shares resources with other threads executing in the same address space.
  • All threads executing in the same process address space are called sibling threads.
  • No parent/child relationship between the creator thread and the createe thread.
    ✦ A thread can create as many threads as it pleases. But there is no relationship among them after creation.

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

− Thread implementations
  • Threads may be implemented as a kernel-level abstraction (kernel-supported threads), user-level abstraction, or a combination of the two.
  ° 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.
  ° A user-level abstraction is represented by data structures within a process’ own address space.
  • 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.

° They are peers in the same process.

• 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.

• Thread resources
  ° All threads within a process share:
    ✦ Text segment
    ✦ Data segment
    ✦ Heap
    ✦ Open files
    ✦ Signal handlers
    ✦ Current working directory
    ✦ UID and GID
In general, user-level threads are designed to share resources with other threads within their own process space running on a single processor.

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 processors** 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.

The UNIX Threads Interface does not define the implementation.

However, it does provide for both **multiplexed** and **bound** threads.

- 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)

- Both implementations can support exactly the same APIs.
- Different UNIX systems may have their own implementation of threads.
  - POSIX threads are portable among UNIX systems that are POSIX-compliant.
    - It is known as **IEEE 1003.1c**
    - AKA **Pthreads**
  - Other major threads implementations (not a complete list):
    - Win32 Threads

- OS/2 Threads
- DCE Threads
- Solaris Threads

- Thread operations
  - Thread creation
    - 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.
      - inherits its signal mask and scheduling priority from the calling thread.
      - does not inherit any pending signals.
      - does not inherit any TSD data.
    - Primitives:
      ```c
      int pthread_create()
      ```

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

• Thread management
  ° 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.
  ° Primitivies:
    int pthread_join()
    int pthread_detach()

• Thread synchronization
  ° 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.
    ◆ 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.
  ° 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.
      ◆ 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.
  ° Primitivies:
    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()

° 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 scheduling
  ° Common scheduling policies:
    ✦ First-come-first-serve

✦ Shortest-job first
✦ Priority-based
✦ Round-robin

° 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.

° 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.

° Primitivies:
  void pthread_setschedparam()
  void pthread_getschedparam()

• TSD manipulation
  ° TSD 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.

° Also, data storage for TSD needs to be explicitly allocated in the thread.
  - A destructor function for cleanup can be specified at the time of creating a TSD key.
  - However, the current implementation still requires the destructor be invoked explicitly.

° To ensure data integrity, mutual exclusion is desired for accessing TSD.

° Primitivies:
  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

- Attributes defined and their values:
  - `contentionscope`
    - `PTHREAD_SCOPE_PROCESS`
    - `PTHREAD_SCOPE_SYSTEM`
  - `detachstate`
    - `PTHREAD_CREATE_JOINABLE`
    - `PTHREAD_CREATE_DETACHED`
  - `stackaddr`
    - `NULL` (valid address)
  - `stacksize`
    - `NULL` (`nnn` within system limits)

Signal handling 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)

Signal delivery

- 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.
  - Only one set of signal handlers per process
    - No thread-specific signal handlers.
    - However, each thread can have its own signal mask.
    - In terms of 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.

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.

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()).

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.

Threads are still evolving. Don’t be surprised to know some things working differently in the newer versions of UNIX later.

Some relevant web sites

http://www.math.arizona.edu/swig/pthreads/threads.html
http://www.best.com/~bos/threads-faq/
http://liinwww.ira.uka.de/bibliography/OS/threads.html

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

There are two semantics in defining fork():

Only the calling thread is replicated (fork1()).

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.

Be aware that signal handlers are process-wide.

No thread-specific signal handlers.

Primitivies:

int pthread_kill()
int pthread_sigmask()
int sigwait()
int sigtimedwait()
int sigwaitinfo()