Implementing Lightweight Threads

- paper “Implementing Lightweight Threads” by Stein and Shah
  - implementation report
  - thread-specific data, signals and threads
  - signal handling also in Nichols et al. Ch. 5

Library Model

- described library is many-to-many
  - scheduling user-level threads onto LWPs
  - threads can be bound and unbound
- this is not a Pthreads implementation, but is close

Thread Structure

- Each thread has the standard set of data associated with it
  - thread ID
  - execution context
  - signal mask
  - priority
  - pointer to thread stack
- Stacks
  - have a ‘red zone’ if they are allocated automatically (page after stack is invalid)
  - or users can specify own stack and do whatever they want

Thread-Specific Data

- thread-specific data useful
  - Pthreads library offers an interface for maintaining thread-specific data
- implementation techniques:
  - through a library API (as with Pthreads)
  - with compiler/linker support
    - compiler recognizes special variables (e.g., `#pragma unshared`) and the linker defines a symbol to represent their size
    - the library then allocates space from the base of the stack
    - object files with thread-specific data cannot be loaded dynamically (`dlopen()`) since the stack will then be already active
    - h/w support makes this more convenient: the %g7 register on a SPARC is reserved (compilers should not use it when generating code), and can be used to hold a pointer to thread-specific data

Preemption

- Two cases:
  - a newly runnable thread has a higher priority than an active one
  - an active thread's priority is lowered below that of a runnable thread
- The library handles preemption by
  - recognizing these cases (they happen through library calls)
  - sending an `SIGLWP` signal to the LWP that needs to take action (a handler is installed by the library)
- Possible problem:
  - the preemted thread may be in the kernel executing a system call
  - library has no way of doing anything, (it can only see the user stack)
  - the call will be restarted (unsafe!)

Threads and LWPs

- The thread library offers a way to ask for a number of LWPs
  - `thr_setconcurrency()`
- New LWPs are created then all existing ones are blocked and runnable threads exist
  - the kernel sends a `SIGWAITING` signal when all LWPs in a process are blocked
- LWP can *idle* – wait for more work, or can be *parked* – wait for specific thread to become runnable again
  - for bound threads always *park*

Traditional Signal Processing

- A process can associate an action with the signals it may receive (sigaction)
  - `SIG_IGN`: ignore signal
  - `SIG_DFL`: default action
    - usually terminate or ignore, but can be stop (`SIGSTOP`), suspend (`SIGTSTP`), resume (`SIGCONT`)
  - execute handler
- Signals can be generated by the system or by other processes, can be synchronous or asynchronous
- A process can block (mask) signals. If they are later unmasked, they get delivered then, but they are not queued!
  - total number of signals received <= number of signals sent

Signal handling in thread libraries

- a single-threaded process should continue to use signals the way it always has, even if linked with a threads library
  - solution: signals are still delivered to the entire process, i.e. sigaction table is common for entire process
- in a multithreaded process one thread must be selected to process the signal (run signal handler)
  - solution: per thread signal masks let the threads declare what they want to handle
- a signal handler has to work in a way such that it does not interfere with the thread execution
  - solution: ‘async safety’

Signal handling

- kinds of signals
  - synchronous undirect
    - `SIGFPE` (e.g. divide by zero), `SIGSEGV` (access to protected memory), `SIGPIPE` (use of broken pipe)
  - synchronous directed: thread_kill
  - asynchronous (undirected): kill
    - `SIGKILL, SIGALRM`,...
- the first two kinds are delivered to the thread directly. the last can be delivered to any one thread that can handle it (only one!)

Thread libraries and Async Signals

- problem: user-level threads and their masks are invisible to the kernel
- signal delivery depends on the thread signal mask
- kernel cannot see it, so cannot determine whether to deliver an asynchronous signal to the thread or not
- kernel cannot deliver signal to a thread that has it masked, but cannot drop it just because currently executing thread has blocked it

Async Safety

- another goal of thread libraries: async safe critical sections
  - a function is async-safe if it is reentrant while handling a signal
  - if a call is not async-safe, then calling it again from a signal handler may result in a deadlock (i.e. the signal handler will block waiting for the interrupted code)
  - Pthreads mutexes and condition variables are not async-safe (POSIX semaphores are)
- Async safety is a standard property of system routines

Async Safety in User libraries

- how does a library enable some operations to be async-safe? just mask the signals in the critical section
  - e.g., an async-safe mutex_lock operation will mask the signals and a mutex_unlock will unmask them
  - such critical sections could be in user code or the thread library code itself
  - since the first signal handler to run is inside the library, masking signals could have very low overhead (just change library data, no system call)

Solutions in Thread Libraries

how can signals be handled by a user-level thread library?

- the LWP signal mask is set to the union of all thread signal masks
  - or at least it's less restrictive than currently active thread
- the library installs its own signal handlers – signals are not first handled by the current thread, but by the library
- when a signal is delivered, the library signal handler finds a thread that can receive it and delivers it to it (this possibly causes an inactive thread to be scheduled)
- if the signal is 'masked' by all threads (not really masked, but deferred for async safety), the LWP signal mask is changed and the signal is resent to the process (if unidirected) or the LWP (if directed) so that the default action is taken

similarly inside the kernel! (substitute 'process' for 'LWP' and 'LWP' for 'thread')

Debugging threads

- Interesting implementation point:
  - the debugger knows nothing about threads
  - a special thread debugging dynamic library is used
  - advantage: the debugger does not need to change when the threads library implementation changes (e.g., internal data structures, implementation of mutexes, etc.)

Destroying threads

- Deteached threads on exit -> put on ‘deathrow’ and periodically destroyed by reaper thread
- Undeteached threads which exit are reaped on thread_join (join code should implment appropriate operations)
- Thread strucutures/stacks are placed on stack -> not to wait for new page allocs, etc.