POSIX Threads

Processes vs Threads

- the kernel maintains a process structure (in kernel space) for each process; the information stored in the process structure includes
  - the memory map
  - the signal dispatch table
  - the file descriptor table
  - IDs (user, group etc), current working directory
  - the signal mask
  - the CPU state (e.g. registers)
  - the kernel stack (for executing system calls)
- when context-switching between 2 processes, the kernel saves the registers in the process structure, changes some virtual memory pointers & loads the CPU registers from the process structure of the other process
- threads within the same process share (within kernel space)
  - the memory map
  - the signal dispatch table
  - the file descriptor table
  - IDs (user, group etc), current working directory
- information about a thread is stored in a thread structure in user space; the information stored includes
  - the thread ID
  - the stack
  - the signal mask
  - the scheduling priority
  - CPU state (including stack pointer & program counter)
- within user space, threads in the same process share
  - global data
  - user code
- when context-switching between 2 threads, the CPU state is switched, but the memory map doesn't need to be changed; hence it's faster than switching between 2 processes

Basics of POSIX Threads

- specified by IEEE 1003.1c (Pthreads)
- prototypes are in the header `pthread.h`
- it may be necessary to define a macro when compiling (e.g. `_REENTRANT`) & link with a special library (e.g. by specifying `-lpthread` to the compiler)
- each thread has a thread ID (TID) which is of type `pthread_t`; this is an opaque type
- a thread can find its own TID by calling `pthread_self`

```
pthread_t pthread_self(void);
```

- we can find out if two threads have the same TID by calling `pthread_equal`

```
int pthread_equal(pthread_t tid1, pthread_t tid2);
/* returns nonzero value if both tid1 & tid2 refer to the same thread;
   otherwise returns 0 */
```

- when a program starts executing, the `main()` function is executed in a thread called the "initial thread" or the "main thread"
- we can create new threads by calling the `pthread_create` function

```
int pthread_create(pthread_t *tid, pthread_attr_t *attr, void *(*start)(void *), void *arg);
/* returns 0 on success, a non-zero error code on error */
```

  - note that functions in Pthreads do not use the `errno` variable; most return 0 on success or an error code (from the `errno.h` header file) on error; use `strerror` to get the corresponding error message
    (Although Pthreads functions do not use `errno`, each thread has its own `errno` which may be set by other functions)
  - the TID is returned through the pointer in the first argument
  - the `attr` parameter specifies the attributes of the new thread (more on this later); we usually use NULL to specify default attributes
  - `start` is a pointer to a function to be executed by the new thread; this function should take one argument of type `void *` & should return a value of the same type
  - `arg` is the argument passed to the function specified by `start`
- a thread created by `pthread_create` terminates either explicitly, by calling `pthread_exit`, or implicitly, by returning from the `start` function; the latter case is equivalent to calling `pthread_exit` with the result returned by the `start` function as the exit code

```
void pthread_exit(void *retval);
```

- note that if the main thread returns (from `main()`), an implicit call to `exit` is made & the whole process exits; also when any thread in a process calls `exit`, the process exits; if the main thread calls `pthread_exit`, only that thread terminates & the process still continues
- a thread can wait for another thread to terminate by calling the `pthread_join` function

```
int pthread_join(pthread_t tid, void **thread_return);
```

  - the thread identified by `tid` must be joinable — it must not be detached; when a joinable thread terminates, its memory resources are not deallocated until another thread calls `pthread_join` on it
  - if `thread_return` is not NULL, the return value (which is of type `void *`) of the thread identified by `tid` is stored in the location `thread_return` points to
  - the return value of the thread is the return value of the function executed by the thread or the argument in its call to `pthread_exit` (or PTHREAD_CANCELLED if the thread was cancelled)
- we can detach a thread by calling `pthread_detach`; detaching a thread just tells the system that the thread's resources can be reclaimed when the thread terminates

```
int pthread_detach(pthread_t tid);
```

Mutexes

- stands for mutual exclusion
- used to protect a critical section so that only one thread (or process) at a time executes the code in the section
- a thread locks the mutex before executing the critical section; when it is through, it unlocks the mutex. Since only one thread can lock (own) the mutex at a time, mutual exclusion is achieved
- a POSIX mutex has type `pthread_mutex_t` (an opaque type); if statically allocated, it can be initialized by assigning it the value PTHREAD_MUTEX_INITIALIZER; if dynamically allocated, it can be initialized by calling `pthread_mutex_init`
- the functions to lock & unlock a mutex are

```
int pthread_mutex_lock(pthread_mutex_t *mutex));
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

  - as before, all these functions return either 0 or an error code
  - the difference between `pthread_mutex_lock` & `pthread_mutex_trylock` is that the former blocks if the mutex is locked by some other thread while the latter returns EBUSY in that case
- caveats
  - you should never make a copy of a mutex; the result of using a copied mutex is undefined
  - only the thread that owns a mutex may unlock it; an erroneous unlock attempt may return an error or may succeed
  - if a thread needs to hold more that one mutex simultaneously, you should use either
    - a fixed locking hierarchy: e.g. always lock mutex A before locking mutex B; or
    - a try & backoff strategy: after locking the first mutex, use `pthread_mutex_trylock` to lock additional needed mutexes; if an attempt fails, unlock all acquired mutexes & start again

Condition Variables

- a condition variable allows threads to suspend execution until some condition (also called predicate) on shared data is satisfied
- the basic operations on condition variables are: signal the condition (when the condition becomes true), and wait for the condition (suspend the thread execution until another thread signals the condition)
- a condition variable must always be associated with a mutex
- the way to use a condition variable is as follows:
  - a thread obtains the mutex before testing the condition
  - if the condition is true, the thread performs its task & releases the mutex when appropriate
  - otherwise, it "waits" for the condition; this releases the mutex & puts the thread to sleep on the condition variable
  - when some other thread "signals" the condition, one sleeping thread is awakened
  - the awakened thread returns from its "wait" operation & automatically relocks the mutex; it should then reevaluate the condition & will either succeed or go back to sleep again
  - it is also possible to do a broadcast rather than a signal; this will awaken all threads waiting on the condition variable

- a POSIX condition variable has type `pthread_cond_t`; if statically allocated, it can be initialized by assigning it the value PTHREAD_COND_INITIALIZER; if dynamically allocated, it can be initialized by calling `pthread_cond_init` (see below)
- the functions to wait, signal & broadcast are

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

  - `pthread_cond_wait` atomically unlocks the mutex & waits for the condition variable `cond` to be signaled; the mutex must be locked by the calling thread on entrance to `pthread_cond_wait`; before returning, `pthread_cond_wait` re-locks the mutex
  - `pthread_cond_signal` restarts one of the threads that are waiting on the condition variable `cond`; if no threads are waiting, nothing happens; if several threads are waiting, exactly one is restarted, but it is not specified which
  - `pthread_cond_broadcast` restarts all threads that are waiting on the condition variable `cond`; nothing happens if no threads are waiting on `cond`
- using the above functions, the way to use a condition variable is:
  - waiting for a condition

```
pthread_mutex_lock(&mutex);  /* must acquire mutex before testing */
while (condition != TRUE)    /* need to re-test condition when awakened */
  pthread_cond_wait(&cond, &mutex);
do_thing();
pthread_mutex_unlock(&mutex);
```

  - signaling a condition

```
pthread_mutex_lock(&mutex);  /* acquire mutex before changing condition */
condition = TRUE;
pthread_mutex_unlock(&mutex);
pthread_cond_signal(&cond);
```

- it is also possible to do a "timed" wait in which you can limit the duration the thread can block

```
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

  - the `timespec` structure is given by

```
struct timespec {
  time_t  tv_sec;   /* seconds */
  long    tv_nsec;  /* nanoseconds */
};
```

  - note that `abstime` is the absolute time measured from the Epoch (January 1, 1970, UTC)
  - `pthread_cond_timedwait` returns ETIMEDOUT if the call is timed out (the condition is still not true when the function returns)