CU CS 4995-03: Internet Systems Programming

POSIX Threads

November 14, 1999

Overview

- thread model (review)
- thread creation
- locks, mutexes
- killing threads

Thread Model

- threads share address space (plus signals, ...) within a process
- "light-weight processes": no copy, easier IPC
- typically, implemented by OS (sometimes user)
- threads may have own thread-specific storage
- *must* compile with `-lpthread`

e.g., *Multithreaded Programming Guide*; Stevens, *Unix Network Programming*, 2nd ed.

Threads Share

- process instructions
- most data
- open files
- signal handlers and dispositions
- current working directory
- user and groups IDs
- locale

Threads Have Their Own

- thread ID
- set of registers (PC, SP)
- stack
- `errno`
- signal mask
- scheduling priority

Thread Variants

- POSIX (pthreads)
- Sun threads (mostly obsolete)
- Java threads

Thread Creation

```
int pthread_create(pthread_t *tid,
  const pthread_attr_t *,
  void *(*func)(void *), void *arg);
```

- Start function in new thread, return 0 if ok, > 0 if error.
- Careful with `arg` argument.

Thread Attributes: Scheduling

```
int pthread_attr_init(pthread_attr_t *attr);
pthread_attr_setscope(pthread_attr_t *attr,
  int scope);
pthread_attr_setschedpolicy(pthread_attr_t *attr,
  int policy);
```

Set scheduling to `PTHREAD_SCOPE_SYSTEM` ("bound" to an LWP) or `PTHREAD_SCOPE_PROCESS` (unbound, default); with policy as `SCHED_FIFO`, `SCHED_RR` and `SCHED_OTHER`.

Joining Threads

Similar to `waitpid`:

```
pthread_join(pthread_t tid, void **status);
```

Detached (`pthread_attr_setdetachstate`) can't be joined; also:

```
pthread_detach(pthread_self());}
```

Find out thread id via `pthread_self()`

Leaving a Thread

- threads can return value, but typicall `NULL`
- just return from function (return `void *`)
- main process exits
- `pthread_exit(void *status)`

Thread-Specific Data

- pass block of thread-specific data on thread creation
- table of keyed thread data items
- need to set up table first:
  `pthread_key_create(&key, destructor)`
- destructor called upon thread exit
- read and write data with `pthread_getspecific` and `pthread_setspecific`
- comparison to plain malloc?

Thread Synchronization

- mutual exclusion, locks: *mutex*
- synchronization: *condition variables*
- semaphores

Mutex

- protect shared static/global data structures
- if locked, block until other thread unlocks
- order of execution not predictable

```
pthread_mutex_t mp = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_init(&mp, NULL);  /* with attributes */
pthread_mutex_lock(&mp);
/* critical region */
pthread_mutex_unlock(&mp);
```

Condition Variables

- wait until some condition occurs

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
pthread_cond_broadcast(pthread_cond_t *c);
```

- `_wait` unblocks mutex and goes to sleep
- broadcast: send to *all* threads waiting
- allows to wait for several sources (UDP, TCP, . . .)

Condition Variable: Signaling

```
pthread_mutex_lock(&var_mutex);
var = ...;
pthread_cond_signal(&var_cond);
pthread_mutex_unlock(&var_mutex);
```

Condition Variable: Waiting

```
pthread_mutex_lock(&var_mutex);
while (var == 0)
  pthread_cond_wait(&var_cond, &var_mutex);
pthread_mutex_unlock(&var_mutex);
```

Why a Mutex?

```
while (!done) {
  /* other thread signals and sets 'done' */
  pthread_cond_wait(&var_cond, &var_mutex);
}
```

Lost wakeup problem if all of:

- thread calls `pthread_cond_signal()` or `pthread_cond_broadcast`
- another thread is between test of condition and `pthread_cond_wait`
- no threads are waiting.

Standard Patterns

Producer-Consumer: finite buffer (queue), ≥ 1 producer, ≥ 1 consumer $\longrightarrow$ read from network, process packets

Reader-Writer: limit concurrent access

Condition Variable: Timed Waiting

```
pthread_cond_timedwait(pthread_cond_t *c, pthread_mutex_t *m,
  struct timespec *abstime);
```

- return `ETIME` if time out
- used for time-outs, e.g., in protocols

```
for (attempt = 0; attempt < limit; attempt++) {
  write(s, request, ...);
  if (pthread_cond_timedwait() != ETIME) {
    /* get message */
    break;
  }
}
```

Semaphores

- counting: thread must wait until count is positive, then decrement
- similar “power” to condition variables, but lower-level
- can be used intra-process:

```
r = sem_init(&sem, 0, initial_count);
```

- or shared between processes (via shared memory):

```
r = sem_init(&sem, 1, initial_count);
```

Semaphores

P, wait: `sem_wait(sem);`

V, post: `sem_post(sem);`, unblocks process or thread

try: `sem_trywait(sem);`, return `EAGAIN` if it would block

Generally good for sharing access to multiple resources.

Named Semaphores

```
sem_t *sem;
/* note slash! */
sem = sem_open("/name", O_CREAT, 0777, 1);
```

Read-Write Locks

Allow multiple readers (while no writer) or single writer

```
pthread_rwlock_init(&rwlock, attr);
pthread_rwlock_rdlock(&rwlock);
pthread_rwlock_wrlock(&rwlock);
pthread_rwlock_unlock(&rwlock);
```

Killing (Cancelling) Threads

- threads can be terminated from “outside”:

```
e = pthread_cancel(thread_id);
```

- but only if it is in “cancellable” state

```
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &old);
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &old);
```

- cancellation type: deferred (default) or asynchronous
- cancellation handlers

```
pthread_cleanup_push(handler, arg);
pthread_cleanup_pop(execute_flag);
```

Cancellation Points

- explicit:

  `pthread_testcancel();`

- `pthread_cond_wait`
- `pthread_join`
- `sigwait`
- standard library calls: `read`, `write`, `sleep`, ...
- not: mutex!