Threads

Table of Contents

Overview

- Introduction
  - multiple tasks within a single process
  - they can access to the same process components
  - synchronization for shared resources
- A Thread Consists of
  - thread ID
  - register values
  - stack content
  - a signal mask
  - an errno variable
  - scheduling priority and policy
  - thread specific data
- Unix Thread Standard
  - POSIX.1-2001, known as pthreads
- Linux Implementation of POSIX Threads
  - via clone system call
  - two flavors
    - LinuxThreads
    - Native POSIX Thread Library (NPTL)
      - better conformance to POSIX.1
        - e.g. POSIX.1 requires threads of a process obtaining same PID by `getpid()`, but LinuxThreads does not follow it
      - better performance
      - require supports from the C library and the kernel
    - both are 1:1 thread model
      - i.e. each thread maps to a kernel scheduling entity
- Thread Identification
  - `pthread_t` data type, and it can be a structure

- test equivalence
    - `int pthread_equal(pthread_t tid1, pthread_t tid2);`
    - return: nonzero if equal, otherwise zero
- get the current thread ID
    - `pthread_t pthread_self(void);`

Thread Creation

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

- Create Function
    - `thread`: be declared previously
    - `attr`: customize thread attributes
    - `start_routine`: routine function
    - `arg`: arguments passed to `start_routine`
    - return: 0 if OK, `errno` on failure
- Note
    - no guarantee which thread runs first
    - threads have access to the process address space
    - threads inherit followings from the calling process
        - floating-point environment
        - signal mask
    - return error code, not use `errno` variable
    - per thread copy of `errno` is still provided for compatibility
- Example
    - code

```
pthread_t ntid;
void printids(const char *s) {
  pid_t pid = getpid();
  pthread_t tid = pthread_self();
  printf("%s pid %u tid %lu (%#lx)\n", s, pid, tid, tid);
}

void *thr_fn(void *arg) {
  printids("new thread: ");
  return NULL;
}

int main(void) {
```

```
  int err = pthread_create(&ntid, NULL, thr_fn, NULL);
  if (err != 0) {
    return 1;
  }
  printids("main thread:");
  sleep(1);
  return 0;
}
```

- result
  - differs on different platforms
  - `pthread_t` may be not an integer
  - `getpid()` may return different values

```
main thread: pid 17242 tid 873604928 (0x34122740)
new thread:  pid 17242 tid 865298176 (0x33936700)
```

Thread Termination

- Terminate
  - entire process
    - `exit(3)`, `_Exit(2)` or `_exit(2)` was called
    - received signal with default action of terminating process
  - signal thread
    - `pthread_exit(3)` was called
    - return from the start routine
    - canceled by another thread in the same process
- Termination Status
  - `wait(2)` can retrieve exit status of a thread
  - `pthread_join(3)`: `int pthread_join(pthread_t thread, void **retval);`
    - return: 0 OK, `errno` on failure
    - suspends the calling thread unless the target has already terminated
    - `retval` stores exit status if it is not NULL
    - the target thread is then placed in a detached state
    - storage of a thread can be released right on its termination
  - `pthread_detach(3)`: `int pthread_detach(pthread_t thread);`
    - return: 0 OK, `errno` on failure
    - cannot be joined, such attempts will return `EINVL`
  - example

```c
void errquit(const char *msg) {
  perror(msg);
  exit(-1);
}

void *thr_fn1(void *arg) {
  printf("thread 1 returning\n");
  return ((void *)1);
}

void *thr_fn2(void *arg) {
  printf("thread 2 exiting\n");
  pthread_exit((void *)2);
}

int main(void) {
  int err;
  pthread_t tid1, tid2;
  int ret;

  err = pthread_create(&tid1, NULL, thr_fn1, NULL);
  if (err != 0) errquit("create thread 1");
  err = pthread_create(&tid2, NULL, thr_fn2, NULL);
  if (err != 0) errquit("create thread 2");
  err = pthread_join(tid1, (void **)&ret);
  if (err != 0) errquit("join thread 1");
  printf("thread 1 exit code %d\n", ret);
  err = pthread_join(tid2, (void **)&ret);
  if (err != 0) errquit("join thread 2");
  printf("thread 2 exit code %d\n", ret);
}
```

    - `void *`
        - can be used to pass a data structure
        - however, the data structure should not be placed on the stack
            - it might be reused by other threads when the thread is terminated
- Canceling a Thread
    - `pthread_cancel(3)`: `int pthread_cancel(pthread_t thread);`
        - return: 0 OK, `errno` on failure
        - sends a cancellation request to `thread`
        - similar to `thread` calls `pthread_exit(PTHREAD_CANCELED)`
        - `thread` can select to ignore or control how it is canceled
        - **does not wait for the thread to terminate**
- Cleanup Functions
    - recall: `atexit(3)` can register functions executed when termination
    - `pthread_cleanup_push(3)`: `void pthread_cleanup_push(void (*routine)(void *), void *arg);`
    - `pthread_cleanup_pop(3)`: `void pthread_cleanup_pop(int execute);`

  - registered routines is executed when
    - calls `pthread_exit(3)`
    - responding to a cancellation request
    - calls `pthread_cleanup_pop(3)` with a nonzero argument
      - if zero argument, it just remove the routine on stack top
- Comparison of Process and Thread Primitives

Process	Thread Primitive	Description
`fork(2)`	`pthread_create(3)`	Create a new flow of control
`exit(3)`	`pthread_exit(3)`	Exit from an existing flow of control
`waitpid(2)`	`pthread_join(3)`	Get exit status from flow of control
`atexit(3)`	`pthread_cleanup_push(3)`	Register function to be called at exit from flow of control
`getpid(2)`	`pthread_self(3)`	Get ID for flow of control
`abort(3)`	`pthread_cancel(3)`	Request abnormal termination of flow of control

Thread Synchronization

- Mutexes
  - `pthread_mutex_init(3p)`: `int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);`
    - alternative `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER`
  - `pthread_mutex_destroy(3p)`: `int pthread_mutex_destroy(pthread_mutex_t *mutex);`
  - `pthread_mutex_trylock(3p)`: `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
  - `pthread_mutex_lock(3p)`: `int pthread_mutex_lock(pthread_mutex_t *mutex);`
  - `pthread_mutex_unlock(3p)`: `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
    - return: 0 OK, error number on failure
  - example: protect data structure

```
// foo_alloc allocate memory and initialize f_lock

void foo_hold(struct foo *fp) {
  pthread_mutex_lock(&fp->f_lock);
  fp->f_count++;
  pthread_mutex_unlock(&fp->f_lock);
}

void foo_rele(struct foo *fp) {
  pthread_mutex_lock(&fp->f_lock);
  if (--fp->f_count == 0) {
    pthread_mutex_unlock(&fp->f_lock);
    pthread_mutex_destroy(&fp->f_lock);
```

```
        free(fp);
    } else {
        pthread_mutex_unlock(&fp->f_lock);
    }
}
```

Deadlock Avoidance

- Solution
    - lock performs in the same order
    - `pthread_mutex_trylock(3p)` can be used to check locks

Reader-Writer Lock

- Reader-Writer Lock
    - similar to mutexes, but higher degree of parallelism
    - mutexes: locked or unlocked
    - reader-writer lock: locked in read mode, locked in write mode, or unlocked
    - **multiple reader locks can be acquired simultaneously**
    - **but only one can lock in write mode**
    - **if a reader/writer locks, then the coming writer/reader must wait until it unlocks**
    - `pthread_rwlock_init(3p)`: `int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);`
        - alternative `pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;`
    - `pthread_rwlock_destroy(3p)`: `int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);`
    - `pthread_rwlock_tryrdlock(3p)`: `int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);`
    - `pthread_rwlock_rdlock(3p)`: `int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);`
    - `pthread_rwlock_trywrlock(3p)`: `int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);`
    - `pthread_rwlock_wrlock(3p)`: `int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);`
    - `pthread_rwlock_unlock(3p)`: `int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);`
        - return: 0 OK, error number on failure

Condition Variable

- Condition Variable
    - condition is protected by a mutex
    - a thread must lock the mutex to change the condition state
    - allow a thread to wait in a race-free way for arbitrary conditions to occur

- `pthread_cond_init(3p)`: `int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);`
    - alternative `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
- `pthread_cond_destroy(3p)`: `int pthread_cond_destroy(pthread_cond_t *cond);`
- `pthread_cond_wait(3p)`: `int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);`
    1. unlock `mutex` (`mutex` must be locked when it is called)
    2. wait for `cond` to occur (thread start to sleep)
    3. if being signaled, lock `mutex`
- `pthread_cond_timedwait(3p)`: `int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime);`
    - occur error `ETIMEOUT` if `abstime` is passed
    - absolute expire time may need `gettimeofday(2)` to set

    ```
    struct timespec {
      long int tv_sec;
      long int tv_nsec;
    };
    ```

- `pthread_cond_broadcast(3p)`: `int pthread_cond_broadcast(pthread_cond_t *cond);`
    - wake up all waiting threads
- `pthread_cond_signal(3p)`: `int pthread_cond_signal(pthread_cond_t *cond);`
    - wake up one waiting thread
    - POSIX.1 allows the implementation wakes up more than one threads
    - waked up threads have to contend for the mutex
    - return: 0 OK, error number on failure

- example: producer and consumer

```
void process_msg(void) {
  struct msg *mp;
  for (;;) {
    pthread_mutex_lock(&qlock);
    /* spurious wakeup */
    /* because other threads may get the resource and set workq to
NULL */
    while (workq == NULL) pthread_cond_wait(&qready, &qlock);
    mp = workq;
```

```
    workq = mp->m_next;
    pthread_mutex_unlock(&qlock);
    /* now process the message mp */
  }
}

void enqueue_msg(struct msg *mp) {
  pthread_mutex_lock(&qlock);
  mp->m_next = workq;
  workq = mp;
  pthread_mutex_unlock(&qlock);
  pthread_cond_signal(&qready);
}
```

```
process: (work, NULL) and wait
enqueue: (work, 1)
enqueue: (work, 2) -> (1)
enqueue: (work, 3) -> (2) -> (1)
process: (work, 2) -> (1)
enqueue: (work, 4) -> (2) -> (1)
process: (work, 2) -> (1)
process: (work, 1)
process: (work, NULL) and wait
```

- example: job queue

```
#define N_WORKERS 3
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
std::list<Job> jobqueue;

int do_the_job(long id, int ch) {
  if (ch == -1) return -1;
  printf("worker-%ld: %c\n", id, ch);
  return 0;
}

void *worker_main(void *arg) {
  long id = (long)arg;
  printf("# worker-%ld created\n", id);
  while (1) {
    Job j;
    pthread_mutex_lock(&mutex);
    pthread_cond_wait(&cond, &mutex);
    // signaled, means at least one job
    j = jobqueue.front();
    if (j.getId() == 0 || pthread_equal(pthread_self(), j.getId())) {
      jobqueue.pop_front();
#ifdef ORDERED  // follow string order
```

```
      if (do_the_job(id, j.getChar()) < 0) {
        pthread_mutex_unlock(&mutex);
        break;
      }
#endif
    } else {
      pthread_mutex_unlock(&mutex);
      continue;
    }
    pthread_mutex_unlock(&mutex);
#ifndef ORDERED  // may not follow string order
    if (do_the_job(id, j.getChar()) < 0) break;
#endif
  }
  printf("# worker-%ld terminated\n", id);
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t workers[N_WORKERS];
  if (argc < 2) {
    fprintf(stderr, "usage: %s input-string\n", argv[0]);
    return -1;
  }
  // create workers
  for (int i = 0; i < N_WORKERS; i++) {
    if (pthread_create(&workers[i], NULL, worker_main, (void *)
(long)i) != 0) {
      fprintf(stderr, "create worker[%d] failed\n", i);
      exit(-1);
    }
  }
  // create jobs, signal every time a new job is pushed in queue
  for (char *ptr = argv[1]; *ptr; ptr++) {
#ifdef ASSIGNID
    Job j(*ptr, workers[(ptr - argv[1]) % N_WORKERS]);
#else
    Job j(*ptr);
#endif
    pthread_mutex_lock(&mutex);
    jobqueue.push_back(j);
    pthread_mutex_unlock(&mutex);
    pthread_cond_signal(&cond);
  }
  // terminate workers
  for (int i = 0; i < N_WORKERS; i++) {
#ifdef ASSIGNID
    Job j(-1, workers[i]);
#else
    Job j(-1);
#endif
    pthread_mutex_lock(&mutex);
    jobqueue.push_back(j);
    pthread_mutex_unlock(&mutex);
```

```
    pthread_cond_signal(&cond);
  }
  // process all jobs
  size_t jobs;
  do {
    pthread_mutex_lock(&mutex);
    jobs = jobqueue.size();
    pthread_mutex_unlock(&mutex);
    pthread_cond_signal(&cond);
  } while (jobs > 0);
  // wait for all workers
  for (int i = 0; i < N_WORKERS; i++) {
    void *ret;
    pthread_join(workers[i], &ret);
  }
  return 0;
}
```

Barrier

- `pthread_barrier_init(3p)`: `int pthread_barrier_init(pthread_barrier_t *restrict barrier, const pthread_barrierattr_t *restrict attr, unsigned count);`
- `pthread_barrier_destroy(3p)`: `int pthread_barrier_destroy(pthread_barrier_t *barrier);`
- `pthread_barrier_wait(3p)`: `int pthread_barrier_wait(pthread_barrier_t *barrier);`
  - return: 0 OK, error number on failure
- Example

```
#define N 5
#ifdef HAS_BARRIER
static pthread_barrier_t barrier;
#endif

void *worker(void *arg) {
  long i, id = (long)arg;
  for (i = 0; i < id + 1; i++) {
    fprintf(stderr, "%ld", id + 1);
  }
  fprintf(stderr, "[%ld/done]\n", id + 1);
#ifdef HAS_BARRIER
  pthread_barrier_wait(&barrier);
#endif
  return NULL;
}
```

```c
int main() {
  long i;
  pthread_t tid;
#ifdef HAS_BARRIER
  pthread_barrier_init(&barrier, NULL, N + 1);
#endif
  for (i = 0; i < N; i++) {
    if (pthread_create(&tid, NULL, worker, (void *)i) != 0) {
      fprintf(stderr, "pthread_create failed.\n");
      return -1;
    }
  }
#ifdef HAS_BARRIER
  pthread_barrier_wait(&barrier);
  pthread_barrier_destroy(&barrier);
#endif
  fprintf(stderr, "all done.\n");
  return 0;
}
```