Lab 6. pthread Library

XUE Jin

jinxue@cse.cuhk.edu.hk

Lab Six

- In this lab, you will learn how to use pthread library to implement multi-threading programs. In particular, we will use the following functions:
  - pthread_create
  - pthread_exit
  - pthread_join
  - pthread_mutex_init
  - pthread_mutex_lock
  - pthread_mutex_trylock()
  - pthread_mutex_timedlock()
  - pthread_mutex_unlock()
  - pthread_cond_init()
  - pthread_cond_wait()
  - pthread_cond_signal()

Thread Creation

```
#include <pthread.h>

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

- `thread`: Used to interact with this thread.
- `attr`: Used to specify any attributes this thread might have.
  - Stack size, Scheduling priority, etc…
- `start_routine`: the pointer to the function this thread execute.
- `arg`: the argument to be passed to the function (`start_routine`)
  - *a void pointer* allows us to pass in *any type of* argument.
- It will return 0 if thread is created successfully. Otherwise returns the error code.

Thread Creation

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

- `retval`: A pointer to the return value

```
int pthread_join(pthread_t thread, void** value_ptr);
```

- `thread`: Specify which thread *to wait for*
- `value_ptr`: A pointer to the return value
  - Because `pthread_join()` routine changes the value, you need to pass in a pointer to that value.

Passing data from child thread to calling threads

- Be careful with how value is returned from a child thread.

```
1   void* thr_func(void* arg) {
2       thread_data_t* data = (thread_data_t*)arg;
3       printf("%d + %d\n", data->a, data->b);
4       int* retptr = malloc(sizeof(int));
5       *retptr = data->a + data->b;
6       return retptr;
7   }
```

- A wrong implementation.

```
1   void* thr_func(void* arg) {
2       thread_data_t* data = (thread_data_t*)arg;
3       printf("%d + %d\n", data->a, data->b);
4       int retval = data->a + data->b;
5       return (void*)&retval;
6   }
```

Locks

- All locks must be properly initialized.
  - One way: using `PTHREAD_MUTEX_INITIALIZER`

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

  - The dynamic way: using `pthread_mutex_init()`

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

Locks

- Provide mutual exclusion to a critical section
  - Interface

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

  - Usage (*lock initialization* and *error check*)

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

Locks

- Check errors code when calling lock and unlock
  - An example wrapper

```
 // Use this to keep your code clean but check for failures
void pthread_mutex_lock(pthread_mutex_t *mutex) {
     int rc = pthread_mutex_lock(mutex);
     assert(rc == 0);
 }
```

Non-blocking locks

- Provide a non-blocking way to lock mutex.
  - Interface

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            const struct timespec* abs_timeout);
```

  - Usage

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
int rc = pthread_mutex_trylock(&lock);
if (rc == EBUSY) {
  // failed to get the lock due to the resource busy
}
else {
  x = x + 1; // or whatever your critical section is
  pthread_mutex_unlock(&lock);
}
```

Non-blocking locks

- Provide a non-blocking way to lock mutex.
  - Interface

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            const struct timespec* abs_timeout);
```

  - Usage

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
int rc = pthread_mutex_timedlock(&lock);
if (rc == ETIMEDOUT) {
  // failed to get the lock due to the resource busy
}
else {
  x = x + 1; // or whatever your critical section is
  pthread_mutex_unlock(&lock);
}
```

Condition Variable

- All conds must be properly initialized.
  - One way: using `PTHREAD_COND_INITIALIZER`

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

  - The dynamic way: using `pthread_cond_init()`

```
int rc = pthread_cond_init(&cond, NULL);
assert(rc == 0); // always check success!
```

Condition Variable

- Provide conditional access to a critical section
  - Interface

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

  - Usage (*main thread*)

```
pthread_mutex_lock(&m);
while (x < 8) // or whatever else conditions are
    pthread_cond_wait(&c, &m);
pthread_mutex_unlock(&m);
```

  - Usage (*child thread*)

```
pthread_mutex_lock(&m);
x = x + 1; // or whatever your critical section is
pthread_cond_signal(&c);
pthread_mutex_unlock(&m);
```

Exercise

- You are required to implement two functions similar to pthread_join() and pthread_exit() based on condition variables

- You may also enjoy the Snake game implementation with pthread library. (https://github.com/anayjoshi/naga.git)