Introduction to Multi-Thread Programming (Part 1)

Yifan ZHU <i@zhuyi.fan>

October 8, 2021

1. Low-Level C API
   - Compile
   - Create, Join, Cancel and Kill
   - Synchronization
     - Mutex
     - RwLock
     - Barrier
     - Conditional Variable
   - TLS and Initialization
     - Dynamic Storage Initialization
     - Thread Local Storage

Compiling Pthread Programs

```
clang/clang++ <source-file> -pthread
```

Creating Threads

https://man7.org/linux/man-pages/man3/pthread_create.3.html

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

Joining Threads

https://man7.org/linux/man-pages/man3/pthread_join.3.html

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

Canceling and Killing Threads

https://man7.org/linux/man-pages/man3/pthread_cancel.3.html
https://man7.org/linux/man-pages/man3/pthread_kill.3.html

```c
int pthread_cancel(pthread_t thread);
int pthread_kill(pthread_t thread, int sig);
```

Mutex

```
#include <pthread.h>
pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t  recmutex  =  PTHREAD_RECURSIVE_MUTEX_INI-
TIALIZER_NP;
pthread_mutex_t errchkmutex =
↪  PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
int   pthread_mutex_init(pthread_mutex_t   *mutex,   const
pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex));
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Mutex

- Recursive Mutex: Allows a mutex to be locked multiple times in the same thread without causing deadlock.
- Error-Check Mutex: Allows a mutex to be locked exactly one times before unlocking; but it will report error on repeated locks.

RwLock

```c
#include <pthread.h>
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
    const pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_rdlock (pthread_rwlock_t *__rwlock);
int pthread_rwlock_wrlock (pthread_rwlock_t *__rwlock);
int pthread_rwlock_unlock (pthread_rwlock_t *__rwlock);
int pthread_rwlock_tryrdlock (pthread_rwlock_t *__rwlock);
```

RwLock

- Read Lock: can be held by multiple thread; when a lock is in the read mode, it cannot be locked as a writer lock.
- Write Lock: can be held by only one thread; when a lock is in the write mode, it cannot be locked as a read lock.

RwLock

Upgrading/Downgrading Sequence

```
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
pthread_rwlock_rdlock(&rwlock);
/* Now Upgrade to write lock */
pthread_rwlock_wrlock(&rwlock);
/* write lock (and read lock) are held here.*/
/* We have effectively upgraded to a write lock */
/* `Downgrade' back to a only the read lock */
pthread_rwlock_unlock(&rwlock);
/* unlock the read lock */
pthread_rwlock_unlock(&rwlock);
```

Barrier

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_init(pthread_barrier_t *restrict
↪   barrier,
    const pthread_barrierattr_t *restrict attr, unsigned
↪   count);
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Conditional Variable

```
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *restrict cond,
    const pthread_condattr_t *restrict attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_wait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

Conditional Variable

```
pthread_cond_wait(mutex, cond):
    value = cond->value; /* 1 */
    pthread_mutex_unlock(mutex); /* 2 */
    pthread_mutex_lock(cond->mutex); /* 10 */
    if (value == cond->value) { /* 11 */
        me->next_cond = cond->waiter;
        cond->waiter = me;
        pthread_mutex_unlock(cond->mutex);
        unable_to_run(me);
    } else
        pthread_mutex_unlock(cond->mutex); /* 12 */
    pthread_mutex_lock(mutex); /* 13 */

pthread_cond_signal(cond):
    pthread_mutex_lock(cond->mutex); /* 3 */
    cond->value++; /* 4 */
    if (cond->waiter) { /* 5 */
        sleeper = cond->waiter; /* 6 */
        cond->waiter = sleeper->next_cond; /* 7 */
        able_to_run(sleeper); /* 8 */
    }
    pthread_mutex_unlock(cond->mutex); /* 9 */
```

Figure: Conditional Variable Explanation

Dynamic Storage Initialization

```c
static int random_is_initialized = 0;
extern int initialize_random();


int random_function()
{
    if (random_is_initialized == 0) {
        initialize_random();
        random_is_initialized = 1;
    }
    /* Operations performed after initialization. */
}
```

Dynamic Storage Initialization

Use Mutex? Too Costly!

Dynamic Storage Initialization

```c
#include <pthread.h>
static pthread_once_t random_is_initialized =
↪  PTHREAD_ONCE_INIT;
extern int initialize_random();


int random_function()
{
    (void) pthread_once(&random_is_initialized,
↪  initialize_random);
    ... /* Operations performed after initialization. */
}
```

Thread Local Storage

```
#include <pthread.h>
int pthread_key_create(pthread_key_t *key, void
↪  (*destr_function) (void *));
int pthread_key_delete(pthread_key_t key);
int  pthread_setspecific(pthread_key_t  key,  const void
↪  *pointer);
void * pthread_getspecific(pthread_key_t key);
```

Thread Local Storage

```c
/* Key for the thread-specific buffer */
static pthread_key_t buffer_key;
/* Once-only initialisation of the key */
static pthread_once_t buffer_key_once = PTHREAD_ONCE_INIT;
/* Allocate the thread-specific buffer */
void buffer_alloc(void)
{
        pthread_once(&buffer_key_once, buffer_key_alloc);
        pthread_setspecific(buffer_key, malloc(100));
}
```

Thread Local Storage

```c
/* Return the thread-specific buffer */
char * get_buffer(void)
{
        return (char *) pthread_getspecific(buffer_key);
}
/* Allocate the key */
static void buffer_key_alloc()
{
        pthread_key_create(&buffer_key, buffer_destroy);
}
/* Free the thread-specific buffer */
static void buffer_destroy(void * buf)
{
        free(buf);
}
```