고려대학교 KOREA UNIVERSITY

Global KU Frontier Spirit

Thread

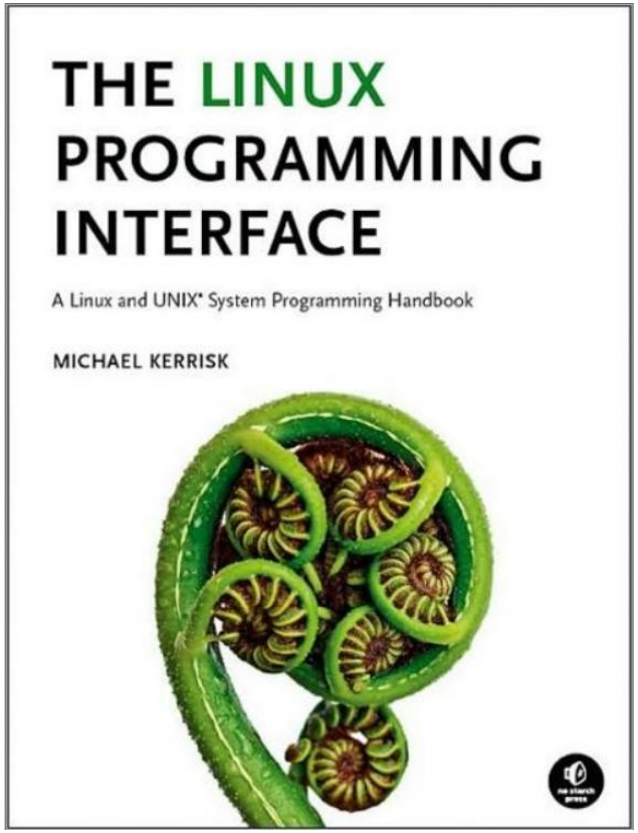
Jinwon Jeong

jin4812@korea.ac.kr

Thread

Reference

- The Linux Programming Interface
  - Published in October 2010
  - No Starch Press
  - ISBN 978-1-59327-220-3



Threads: Introduction

- We describe POSIX threads, often known as ***Pthreads***.
- Like processes, threads are a mechanism that permits an application to perform **multiple tasks concurrently**.
- A single process can **contain multiple threads**.
- All of these threads are **independently executing** the same program.
- They **all share** the same global memory and heap segments.

Threads: Introduction

- On a multiprocessor system, multiple threads can be **executed in parallel**.

  – If one thread is blocked on I/O,
  other threads are still eligible to execute.

Threads: Introduction

- Processes does have the following **limitations** in some applications:

  – It is **difficult to share information** between processes.

  – Since the **parent and child don't share memory** (other than the read-only text segment), we **must use some form** of interprocess communication in order to exchange information between processes.

Threads: Introduction

- Process creation with *fork()* is **relatively expensive**.

- Even with the copy-on-write technique, the need to duplicate various process attributes such as **page tables and file descriptor tables** means that a *fork()* call is **still time-consuming**.

Threads: Introduction

- Threads address both of these problems:
  - Sharing information between threads is easy and fast.
  - It is just a matter of copying data into shared (global or heap) variables
    - Synchronization techniques are needed.
  - Thread creation is faster than process creation
    - Typically 10 times faster or better.
    - Thread creation is faster because many of the attributes that must be duplicated in a child created by fork() are instead shared between threads.

Threads: Introduction

- Pthreads data types

Data type	Description
pthread_t	Thread identifier
pthread_mutex_t	Mutex
pthread_mutexattr_t	Mutex attributes object
pthread_cond_t	Condition variable
pthread_condattr_t	Condition variable attributes object
pthread_key_t	Key for thread-specific data
pthread_once_t	One-time initialization control context
pthread_attr_t	Thread attributes object

- When compile, use *–pthread* or *–lpthread* option.

Threads: Introduction

- Thread Creation

  – The *pthread_create()* function creates a new thread.

```
#include <pthread.h>

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

                                   Return 0 on success, or a positive error number on error
```

Threads: Introduction

- After a call to *pthread_create()*, a program has no guarantees about which thread will next be scheduled to use the CPU.

- Programs that implicitly rely on a particular order of scheduling are open to the same sorts of **race conditions**.

- If we need to enforce a particular order of execution, we must use one of the synchronization techniques.

Threads: Introduction

- Thread Termination

– The execution of a thread terminates in one of the following ways:

- The thread's start function performs a return specifying a return value for the thread.
- The thread call *pthread_exit().*
- The thread is canceled using *pthread_cancel().*
- Any of the threads calls *exit()*, or the main thread performs a return.(in the *main()* function)
  - It causes all threads in the process to terminate immediately.

Threads: Introduction

- The *pthread_exit()* function terminates the calling thread, and specifies a return value that can be obtained in another thread by calling *pthread_join().*

```
#include <pthread.h>

void pthread_exit(void *retval);
```

Threads: Introduction

```
1  #include<pthread.h>
2  #include<stdlib.h>
3  #include<stdio.h>
4
5  void *hello_thread (void *arg)
6  {
7      printf("thread : hello₩n");
8      return arg;
9  }
10
11 main()
12 {
13     pthread_t tid;
14     int status;
15
16     status = pthread_create(&tid, NULL, hello_thread, NULL);
17     if(status != 0)
18         perror("create Thread");
19     pthread_exit(NULL);
20 }
```

Threads: Introduction

```c
#include<stdlib.h>
#include<pthread.h>
#include<stdio.h>

#define NUM_THREADS 3

void *hello_thread(void *arg)
{
    printf("Thread %ld : Hello, world₩n", (long int)arg);
    return arg;
}

main()
{
    pthread_t tid [NUM_THREADS];
    long int i, status;

    for(i=0; i<NUM_THREADS; i++)
    {
        status = pthread_create(&tid[i], NULL, hello_thread, (void *)i);
        if(status != 0)
        {
            fprintf(stderr, "create thread %ld : %ld₩n", i, status);
            exit(1);
        }
    }
    pthread_exit(NULL);
}
```

Threads: Introduction

- Joining with a Terminated Thread.

– The *pthread_join()* function waits for the thread to terminate.
  - If that thread has already terminated, *pthread_join()* returns immediately.

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);

                                    Returns 0 on success, or a positive error number on error
```

Threads: Introduction

- If a thread is not detaches, then we must join with it using *pthread_join().*

  - If we fail to do this, then, when the thread terminates, it produces the thread equivalent of a zombie process.

  - Aside from wasting system resources, we won't be able to create additional threads.

Threads: Introduction

join.c

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#define NUM_THREADS 5
pthread_t tid[NUM_THREADS];

void *hello_thread(void *arg){

        int i = 0;
        while(1){
                if(i == 5){
                        break;
                }
                i++;
                sleep(1);
        }
        printf("thread%ld terminated... \n", (intptr_t) arg);
        pthread_exit( (void*)(intptr_t)i);
}
```

Threads: Introduction

```
void main(){

        long int i, status;
        printf("pid : %d\n", getpid());

        for(i=0;i<=NUM_THREADS; i++){
                status = pthread_create(&tid[i], NULL, &hello_thread, (void *)(intptr_t)i);
                if(status !=0){
                        fprintf(stderr, "create  thread %ld, status : %ld\n", i, status);
                        exit(1);
                }
                printf("thread%ld is created...\n", i);
        }

        for(i=0; i<=NUM_THREADS; i++){
                void *return_value;
                pthread_join(tid[i], &return_value);
                printf("thread%ld's return value is %ld\n", i, (intptr_t)return_value);

        }
```

Threads: Introduction

- Detaching a Thread

  – By default, a thread is *joinable*, meaning that when it terminates, another thread can obtain its return status using *pthread_join().*

  – Sometimes we don't care about the thread's return status.

    - We simply want the system to automatically clean up and remove the thread when it terminates.

Threads: Introduction

– In this case, we can mark the thread as detached, by making a call to pthread_detach() specifying the thread's identifier in thread.

```
#include <pthread.h>

int pthread_detach(pthread_t thread);

                                    Returns 0 on success, or a positive error number on error
```

Threads: Introduction

– Once a thread has been detached, it is no longer possible to use *pthread_join()*.

  - The thread can't be made joinable again.

Threads: Introduction

detach.c

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#define NUM_THREADS 5

int end[NUM_THREADS];
pthread_t tid[NUM_THREADS];

void *hello_thread(void *arg){

        int i;
        double result = 0.0;
        int state;
        state = pthread_detach(pthread_self());
        if(state == 0){
                printf("thread%d(%d) is detached!\n",(int)(intptr_t) arg, getpid());
        }else{
                printf("pthread_detach() got a error.,,\n");
        }

        while(!end[(int)(intptr_t) arg]){
                for(i=0; i<50000; i++){
                        result = result + (double)random();
                }
        }
        printf("thread%d's result : %e\n",(int)(intptr_t) arg, result);
        printf("thread%d terminated...\n", (int)(intptr_t) arg);

}
```

Threads: Introduction

```
void main(){

        long int i, status;
        printf("pid : %d\n", getpid());
        for(i=0;i<=NUM_THREADS; i++){
                end[i] = 0;
                status = pthread_create(&tid[i], NULL, hello_thread, (void *)(intptr_t)i);
                if(status !=0){
                        fprintf(stderr, "create  thread %ld, status : %ld\n", i, status);
                        exit(1);
                }
                printf("thread%ld is created...\n", i);
        }
        sleep(1);
        for(i=0;i<=NUM_THREADS; i++){
                end[i] = 1;
        }
        sleep(5);
        printf("++pid%d is finished...++\n", getpid());

        return;
}
```

Threads: Thread Synchronization

- We describe two tools that threads can use to synchronize their actions:
  - Mutexes
  - Condition variables

Threads: Thread Synchronization

- Mutexes allow threads to synchronize their use of a shared resource.
  - So that one thread doesn't try to access a shared variable at the same that as another thread is modifying it.

- Condition variables perform a complementary task.
  - They allow threads to inform each other that a shared variable (or other shared resource) has changed state.

Threads: Thread Synchronization

- The term critical section is used to refer to a section of code that accesses a shared resource and whose execution should be *atomic (All or Nothing)*.

  – Its execution should not be interrupted by another thread that simultaneously accesses the same shared resource.

Mutexes

- To avoid the problems that can occur when threads try to update a shared variable, we must use a *mutex* (short for *mutual exclusion*) to ensure that only one thread at a time can access the variable.

- A mutex has two states:

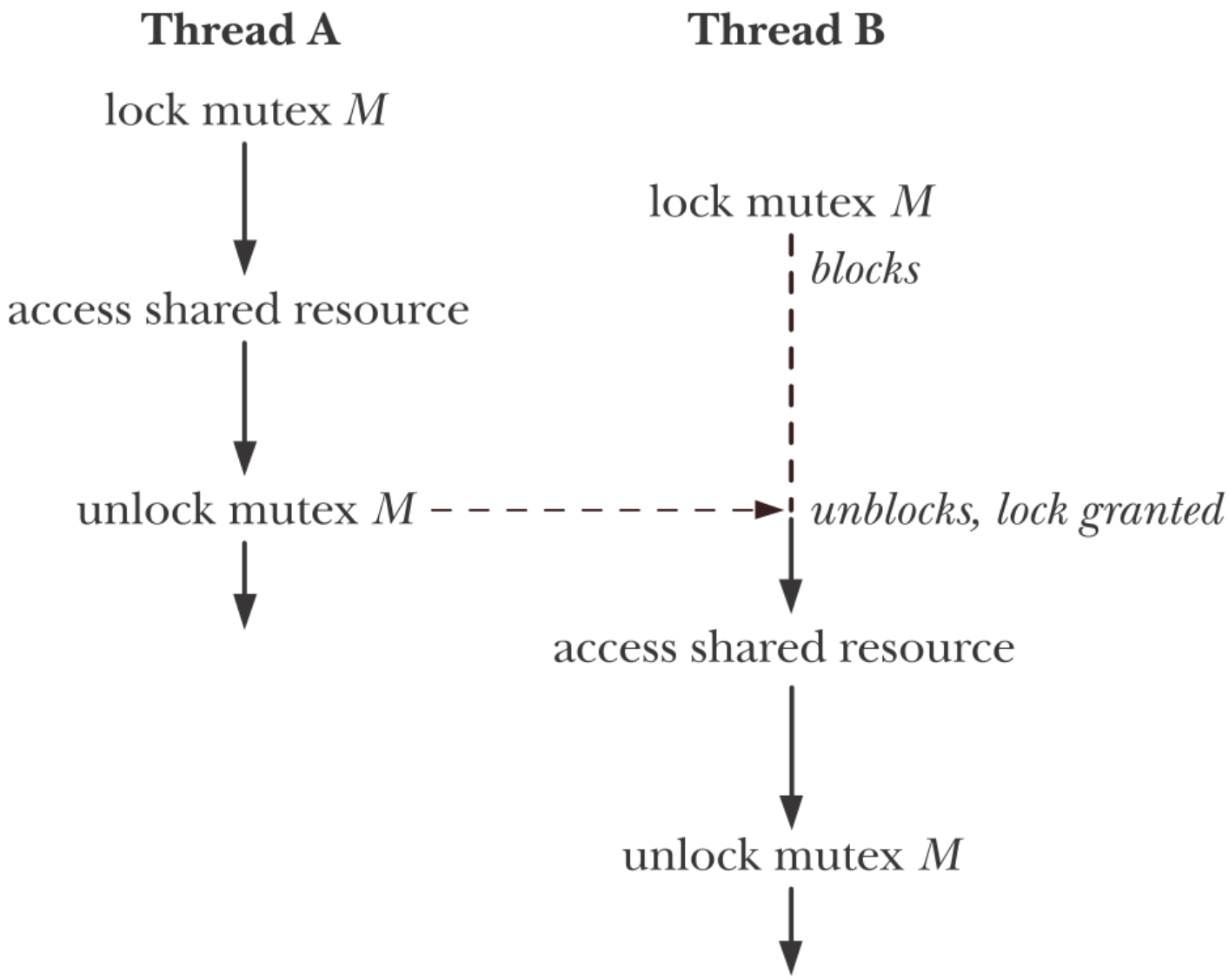
  - *locked*
  - *unlocked*

Mutexes

- At any moment, at most one thread may hold the lock on a mutex.

- Attempting to lock a mutex that is already locked either blocks or fails with an error.

Mutexes


Thread A
Thread B
lock mutex M
lock mutex M
blocks
access shared resource
unlock mutex M
unblocks, lock granted
access shared resource
unlock mutex M

Mutexes

- A mutex is a variable of the type *pthread_mutex_t*.
- Before it can be used, a mutex must always be initialized.

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

Returns 0 on success, or a positive error number on error

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Returns 0 on success, or a positive error number on error

Mutexes

– Locking and Unlocking a Mutex.

- After initialization, a mutex is unlocked.
- To lock and unlock a mutex, we use the *pthread_mutex_lock()* and *pthread_mutex_unlock()* functions.

```
#include <pthread.h>

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

                    Both return 0 on success, or a positive error number on error
```

Mutexes

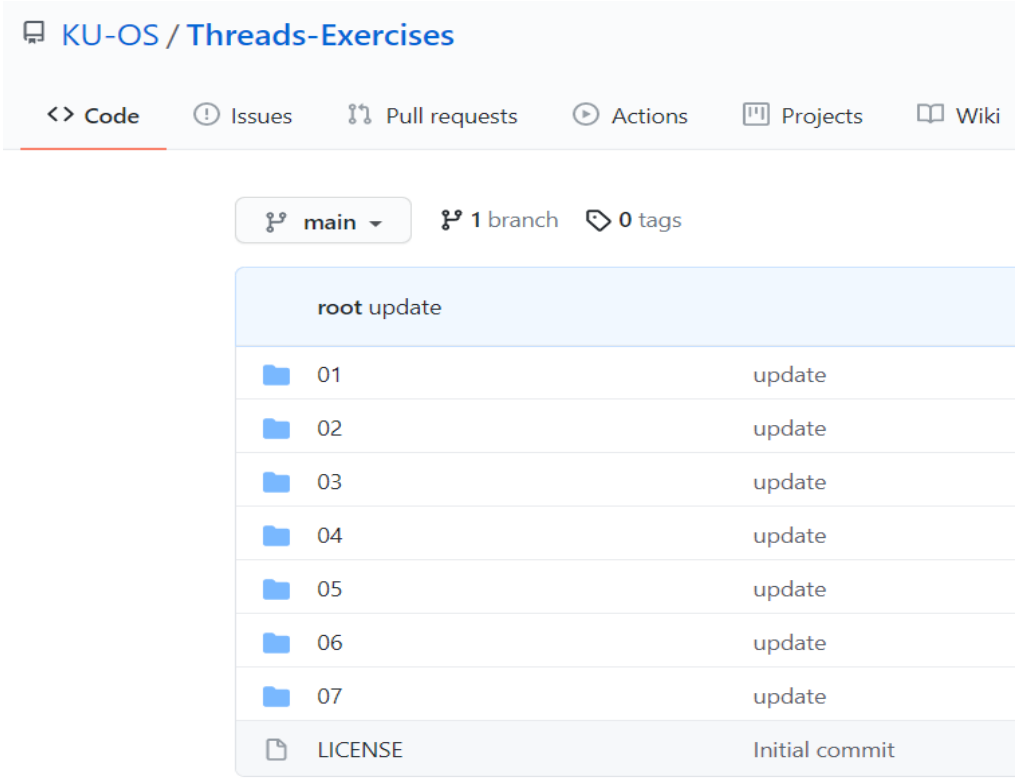
```c
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 3

pthread_mutex_t mutex;
long int sum;
void *mutex_thread(void *arg)
{
        pthread_mutex_lock (&mutex);
        sum +=(long int) arg;
        pthread_mutex_unlock (&mutex);

        return arg;
}
```

```c
int main (int argc, char *argv[])
{
        pthread_t tid[NUM_THREADS];
        long int arg[NUM_THREADS], i;
        void *result;
        int status;

        if (argc < 4)
        {
                fprintf( stderr,"usage: mutexthread number1 number2 number3 \n");
                exit(1);
        }

        for (i=0; i< NUM_THREADS; i++)
                arg[i] = atoi (argv[i+1]);

        pthread_mutex_init (&mutex, NULL);

        for (i=0; i< NUM_THREADS; i++)
        {
                status = pthread_create ( &tid[i], NULL, mutex_thread, (void *) arg [i]);
                if (status !=0)
                {
                        fprintf(stderr, "create thread %d: %d" , (int)i, (int)status);
                        exit(1);
                }
        }
        for (i=0; i< NUM_THREADS; i++)
        {
                status = pthread_join ( tid[i], &result);
                if (status !=0)
                {
                        fprintf(stderr, "join thread %d: %d" , (int)i, (int)status);
                        exit(1);
                }
        }

        status = pthread_mutex_destroy (&mutex);
        if (status != 0)
                perror ("Desotry mutex");
        printf(" sum is %ld\n", sum);
        pthread_exit (result);
}
```

Assignment

Visit → https://github.com/KU-OS/Threads-Exercises

Do : 01 ~ 07



Assignment

리눅스 환경(가상머신) 또는 macOS에서 실습 진행

1) 각 문제의 빈칸에 들어갈 코드 작성(전체 코드 작성할 필요 없음)

<?1/> xxx
<?2/> xxx
…

! 06과 07번 문제는 빈칸에 왜 그러한 코드를 작성하였는지에 대한 이유를 전체 코드의 흐름과 연관하여 서술할 것.
(07번 문제는 빈칸이 없으므로 적절한 공간에 코드를 작성하고 이유 서술.)

2) 각 문제마다 출력 결과 스크린샷(계정 보이게)

```
root@kali:~# gcc -o thread1 thread1.c -lpthread
root@kali:~# ./thread1
thread : hello
```

PDF 파일로 제출