

Interlude: Thread API

Operating System (OS)
안인규

강의자료는 KAIST 원유집 교수님 OS 강의 자료를 참고했습니다.

Thread Creation

- How to create and control threads?

```
#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. (생성된 thread를 식별하기 위한 handle을 저장할 pointer)
- `attr`: Used to specify any attributes this thread might have. (thread 속성 설정)
 - Stack size, Scheduling priority, ...
- `start_routine`: the function this thread start running in. (thread가 시작할 함수의 주소)
- `Arg`: the argument to be passed to the function (`start routine`) (thread의 입력 인자)
 - *a void pointer* allows us to pass in *any type of* argument.

Thread Creation (Cont.)

- If `start_routine` instead requires another type argument, the declaration would look like this:
 - An integer argument:

```
int
pthread_create(..., // first two args are the same
                void*  (*start_routine)(int),
                int    arg);
```

- Return an integer:

```
int
pthread_create(..., // first two args are the same
                int   (*start_routine)(void*),
                void* arg);
```

Example: Creating a Thread

```
#include <pthread.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    rc = pthread_create(&p, NULL, mythread, &args);
    ...
}
```

Wait for a thread to complete

```
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 the pointer** to that value.

Example: Waiting for Thread Completion

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <assert.h>
4  #include <stdlib.h>
5
6  typedef struct __myarg_t {
7      int a;
8      int b;
9  } myarg_t;
10
11 typedef struct __myret_t {
12     int x;
13     int y;
14 } myret_t;
15
16 void *mythread(void *arg) {
17     myarg_t *m = (myarg_t *) arg;
18     printf("%d %d\n", m->a, m->b);
19     myret_t *r = malloc(sizeof(myret_t));
20     r->x = 1;
21     r->y = 2;
22     return (void *) r;
23 }
24
```

```
25 int main(int argc, char *argv[]) {
26     int rc;
27     pthread_t p;
28     myret_t *m;
29
30     myarg_t args;
31     args.a = 10;
32     args.b = 20;
33     pthread_create(&p, NULL, mythread, &args);
34     pthread_join(p, (void **) &m);
35     // this thread has been waiting inside of
36     // the pthread_join() routine.
37     printf("returned %d %d\n", m->x, m->y);
38     return 0;
39 }
```

Example: Dangerous code

- Be careful with how values are returned from a thread.

```
1 void *mythread(void *arg) {
2     myarg_t *m = (myarg_t *) arg;
3     printf("%d %d\n", m->a, m->b);
4     myret_t r; // ALLOCATED ON STACK: BAD!
5     r.x = 1;
6     r.y = 2;
7     return (void *) &r;
8 }
```

- When the variable `r` returns, it is automatically **de-allocated**.

Example: Simpler Argument Passing to a Thread

- Just passing in a single value using casting (형 변환)

```
1  void *mythread(void *arg) {
2      int m = (int) arg;
3      printf("%d\n", m);
4      return (void *) (m + 1);
5  }
6
7  int main(int argc, char *argv[]) {
8      pthread_t p;
9      int rc, m;
10     pthread_create(&p, NULL, mythread, (void *) 100);
11     pthread_join(p, (void **) &m);
12     printf("returned %d\n", m);
13     return 0;
14 }
```

Locks

- Provide **mutual exclusion**(상호 배제) to a critical section

- Interface

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

- Usage (w/o *lock initialization* and *error check*)

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

→ *Opaque type(불투명 타입)*

- Lock의 상태, Lock을 잡고있는 thread ID 등이 포함

- No other thread holds the lock → the thread will acquire the lock and **enter the critical section**.
- If another thread hold the lock → the thread will **not return from the call** until it has acquired the lock.

Locks (Cont.)

- 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 (Cont.)

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

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

- These two calls are used in lock acquisition(획득)

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

- trylock: return failure if the lock is already held
- timedlock: return after a timeout or after acquiring the lock
- Deadlock 감지 / Real-time system에서 활용 가능

Condition Variables

- **Condition variables** are useful when some kind of **signaling** must take place between threads.

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

- `pthread_cond_wait`:
 - Put the calling thread to sleep.
 - Wait for some other thread to signal it.
- `pthread_cond_signal`:
 - Unblock at least one of the threads that are blocked on the condition variable

Condition Variables

- A thread calling wait routine:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t init = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (initialized == 0)
    pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
```

- The wait call **releases the lock** when putting said caller to sleep.
- Before returning after being woken, the wait call **re-acquire the lock**.
- A thread calling signal routine:

```
pthread_mutex_lock(&lock);
initialized = 1;
pthread_cond_signal(&init);
pthread_mutex_unlock(&lock);
```

Condition Variables

- The waiting thread **re-checks** the condition **in a while loop**, instead of a simple if statement.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t init = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (initialized == 0)
    pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
```

- Without rechecking, the waiting thread will continue thinking that the condition has changed *even though it has not*.

Condition Variables (Cont.)

- Don't ever do this.
 - A thread calling wait routine:

```
while(initialized == 0)  
    ; // spin
```

- A thread calling signal routine:

```
initialized = 1;
```

- It performs poorly in many cases. → just wastes CPU cycles.
- It is error prone.

Compiling and Running

- To compile them, you must include the header `pthread.h`
 - Explicitly link with the `pthread` library, by adding the `-pthread` flag.

```
prompt> gcc -o main main.c -Wall -pthread
```

- For more information,

```
man -k pthread
```