

OS01 7주차 과제 - 201702081

- Concurrency

1. 코드 이미지 및 설명

- (1) atomicity

```
main: begin
t1: before check
t1: after check
                t2: begin
                t2: set to NULL
t1: use!
[1] 42939 segmentation fault ./atom_raw
```

```
if (thd->proc_info) {
    printf("t1: after check\n");
    sleep(2);
    printf("t1: use!\n");
    printf("%d\n", thd->proc_info->pid);
}
<- thread 1
```

```
printf("                t2: begin\n");
sleep(1); // change to 5 to make the code "work"...
// // // // THREAD2가 THREAD1 접근 전에 proc_info를 밀어버려서 fault
printf("                t2: set to NULL\n");
thd->proc_info = NULL;
<- thread 2
```

➔ thread1에서는 thd의 정보를 사용하고, thread2에서 thd의 정보를 밀어버린다.

thread2가 더 먼저 동작하면, thread1가 use할 때 접근할 대상이 없어 오류가 발생한다.

따라서 세마포어로 순서를 정하고, lock / unlock으로 mutex를 보장해주었다.

```
typedef struct {
    int pid;
} proc_t;

typedef struct {
    proc_t *proc_info;
} thread_info_t;

proc_t p;
thread_info_t *thd;

// 1. SEMAPHORE & LOCK
sem_t sem;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *thread1(void *arg);
void *thread2(void *arg);

int main(int argc, char *argv[]) {
    thread_info_t t;
    p.pid = 100;
    t.proc_info = &p;
    thd = &t;

    // 2. SEMAPHORE INIT : 0
    sem_init(&sem, 0, 0);

    pthread_t p1, p2;
    printf("main: begin\n");
    pthread_create(&p1, NULL, thread1, NULL);
    pthread_create(&p2, NULL, thread2, NULL);
    // join waits for the threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

➔ Semaphore 변수 생성, mutex 변수 초기화 / Semaphore 초기화

```

void *thread1(void *arg) {

    pthread_mutex_lock(&mutex);           // 3. LOCK
    printf("t1: before check\n");
    if (thd->proc_info) {
        printf("t1: after check\n");
        sleep(2);
        printf("t1: use!\n");
        printf("%d\n", thd->proc_info->pid);
    }
    pthread_mutex_unlock(&mutex);         // 3. UNLOCK
    sem_post(&sem);                       // 4. SEMIGNAL

    return NULL;
}

void *thread2(void *arg) {

    sem_wait(&sem);                       // 4. SEMWAIT
    pthread_mutex_lock(&mutex);           // 3. LOCK
    printf("          t2: begin\n");
    sleep(1); // change to 5 to make the code "work"...
    // THREAD2가 THREAD1 접근 전에 proc_info를 밀어버려서 fault
    printf("          t2: set to NULL\n");
    thd->proc_info = NULL;
    pthread_mutex_unlock(&mutex);         // 3. UNLOCK

    return NULL;
}

```

- ➔ thread 1, 2의 각 시작과 끝에 lock과 unlock을 걸어 공유변수 접근에 간섭을 받지 않도록 하고,
- ➔ thread2의 시작에 sem_wait / thread1의 끝에 sem_post를 넣어 thread1 완료 후에 2가 실행되도록 하였다.

(2) ordering

```
ordering: begin
routine: begin
[1] 43001 segmentation fault ./ord_raw
```

```
pthread_create(&p->th, NULL, start_routine, NULL);
// turn the sleep off to avoid the fault, sometimes...
sleep(1);      // 자는동안 스레드에서 thd에 접근하는데, 리턴이 되지 않았으므로
| | | | | // 값이 안들어있음 -> fault
return p;
```

<- myCreateThread

```
thd = myCreateThread(routine);
```

<- main

```
printf("routine: begin\n");
printf("routine: state is %d\n", thd->state);
```

<- routine

➔ myCreateThread 에서 반환된 값이 main에서 전역변수 thd에 저장되고, 이후에 스레드의 routine에서 thd의 값을 사용해야 한다.

하지만, myCreateThread가 리턴되지 않고 sleep 하는 동안 스레드에서 thd의 값을 사용하려 할 시 에러가 발생한다.

➔ 따라서 세마포어로 순서를 정하고, lock / unlock으로 mutex를 보장해주었다.

```
typedef struct {
    pthread_t th;
    int state;
} my_thread_t;

my_thread_t *thd;

// 1. SEMAPHORE & LOCK
sem_t sem;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

➔ Semaphore 변수 생성, mutex 변수 초기화

```

void *routine(void *arg) {
    sem_wait(&sem);                // 4. SEMWAIT
    pthread_mutex_lock(&mutex);    // 3. LOCK
    printf("routine: begin\n");
    printf("routine: state is %d\n", thd->state);
    pthread_mutex_unlock(&mutex);  // 3. UNLOCK
    return NULL;
}

void myWaitThread(my_thread_t *p) {
    pthread_join(p->th, NULL);
}

my_thread_t *myCreateThread(void *(*start_routine)(void *)) {
    my_thread_t *p = malloc(sizeof(my_thread_t));
    if (p == NULL)
        return NULL;
    p->state = STATE_INIT;
    pthread_create(&p->th, NULL, start_routine, NULL);
    // turn the sleep off to avoid the fault, sometimes...
    sleep(1);          // 자는동안 쓰레드에서 thd에 접근하는데, 리턴이 되지 않았으므로
    // | | | | | // 값이 안들어있음 -> fault
    return p;
}

```

```

int main(int argc, char *argv[]) {
    // 2. SEMAPHORE INIT : 0
    sem_init(&sem, 0, 0);

    printf("ordering: begin\n");
    pthread_mutex_lock(&mutex);    // 3. LOCK
    thd = myCreateThread(routine);
    pthread_mutex_unlock(&mutex);  // 3. UNLOCK
    sem_post(&sem);                // 4. SEMSIGNAL
    myWaitThread(thd);
    printf("ordering: end\n");
    return 0;
}

```

- ➔ thread routine / main 의 공유변수 접근부에 lock과 unlock을 걸어 서로 간섭을 받지 않도록 하고,
- ➔ thread routine의 시작에 sem_wait / main에 sem_post를 넣어 main의 thd 설정 완료 후에 thread routine이 실행되도록 하였다.

(3) deadlock prevention

Hold and wait : 필요한 Lock L1, L2를 모두 할당받은 상태에서만 각 스레드가 시작되도록 한다. L1, L2를 한 곳에서 독점해야 하기 때문에, 두 스레드의 실행 시작 순서를 세마포어를 통해 제어하면 독점시킬 수 있다. 자원을 다 가진 상태의 스레드는 wait할 필요가 없으므로 Hold and wait가 깨진다.

No preemption : 한 스레드가 lock을 얻으려고 할 때마다, 다른 쪽 스레드에서 unlock시키면서 wait하게 한다. 다른 스레드의 자원을 강제로 뺏으므로, No preemption이 깨진다.

Circular wait : 자원을 요구하는 순번을 정의하고, 그에 따라 접근한다. 순서가 바뀌어 순환구조를 벗어나므로, Circular wait가 깨진다.

| THREAD1 | THREAD2 | <- 전 |
|-----------|-----------|------|
| lock l1 | lock l2 | |
| lock l2 | lock l1 | |
| unlock l1 | unlock l1 | |
| unlock l2 | unlock l2 | |
| | | |
| THREAD1 | THREAD2 | <- 후 |
| lock l1 | lock l2 | |
| unlock l1 | lock l1 | |
| lock l2 | unlock l2 | |
| unlock l2 | unlock l1 | |

2. 실행 결과

(1) atomicity

```
main: begin
t1: before check
t1: after check
t1: use!
100
                                t2: begin
                                t2: set to NULL
main: end
```

t1의 sem_post가 수행되고 나서, sem_wait를 통해 기다리던 t2가 시작된다.

각 스레드의 동작은, lock / unlock으로 mutex가 보장되어있기 때문에 섞이지 않는다.

마지막에 pthread_join이 호출되면 스레드를 정리하고 끝났음을 표시한다.

(2) ordering

```
ordering: begin
routine: begin
routine: state is 0
ordering: end
```

ordering의 sem_post가 수행되고 나서, sem_wait를 통해 기다리던 routine이 시작된다.

ordering과 routine의 동작은, lock / unlock으로 mutex가 보장되어있기 때문에 섞이지 않는다.

마지막에 pthread_join이 호출되면 스레드를 정리하고 끝났음을 표시한다.