KWANGWOON UNIVERSITY

2022년 2학기 **운영체제실습** 10주차

Synchronization

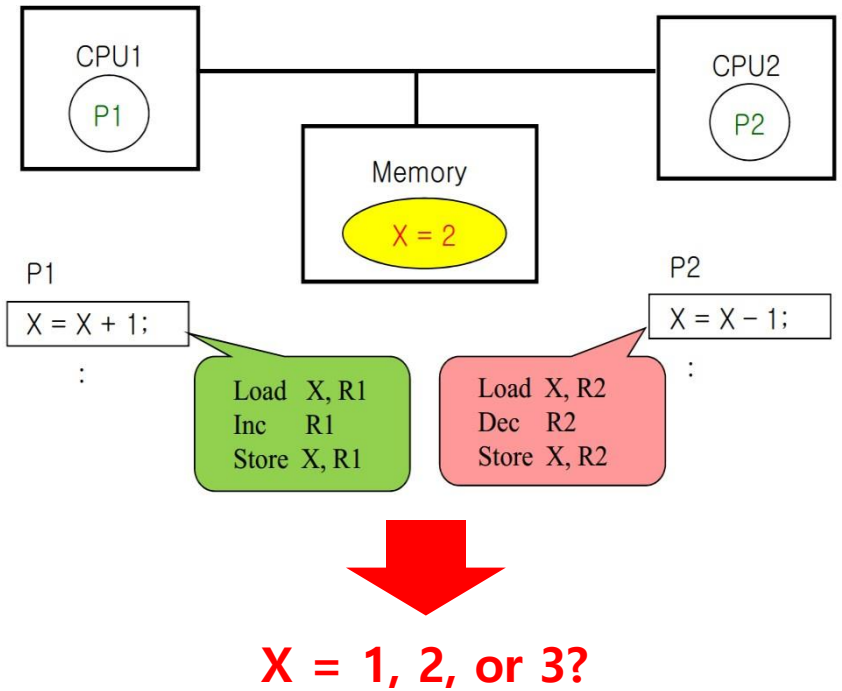
System Software Laboratory
School of Computer and Information Engineering
Kwangwoon Univ.

Contents

- **Synchronization**
- **Spin Lock**
- **Mutex**
  - Lab. 1
- **Semaphore**
  - Lab. 2
- **Deadlock**

Synchronization

- **멀티프로세서 환경 또는 시분할 방식에서, 병렬성(parallelism)을 활용하여 처리율, 응답 시간 등의 성능 향상을 얻을 수 있음**
  - 다만, 공유 자원에 동시에 접근 하여 경쟁 조건이 발생할 수 있으므로 이를 방지하여야 함
    - 동기화의 필요성
    - 공유 자원 (shared resource)
      - 시스템 자원의 대부분이 공유될 수 있음
    - 경쟁 조건 (race condition)
      - 하나 이상의 프로세스가 동일한 자원을 접근하기를 원하는 상태



Synchronization

- **동기화 순서**
  - 1) 임계 구역 (critical section)을 정의
    - 공유자원을 접근하기 위한 코드의 일부
  - 2) 상호 배제 (mutual exclusion) 메커니즘의 사용
    - 한 시점에 하나의 프로세스만 공유 자원을 접근할 수 있음
- **대표적인 동기화 방법**
  - 락, 세마포어, 파이프 등
- **주의 사항**
  - 데드락이 일어나지 않도록 해야 함
    - 데드락(deadlock): 절대 발생하지 않을 일을 무한정 기다리는 현상

Spin Lock

- **스핀 락 (spin lock)**
  - 리눅스 커널에서 가장 일반적인 락킹 기법.
  - 다른 스레드가 이미 가진 스핀락을 얻으려고 시도한다면?
    - 그 락을 얻을 때까지 바쁜 루프(busy loop)로 대기

- **특징**
  - 문맥 교환이 필요하지 않음
  - 바쁜 루프로 인한 CPU 소모
  - 짧은 기간 대기에 효과적임
    - 스핀락을 사용하는 시간이 두 번의 문맥교환 시간보다 짧은 것이 효율적
  - Linux에서는 SMP 환경에서만 사용
  - Symmetric Multi-Processing (SMP)
    - 두 개 이상의 프로세서가 한 개의 공유된 메모리를 사용하는 구조

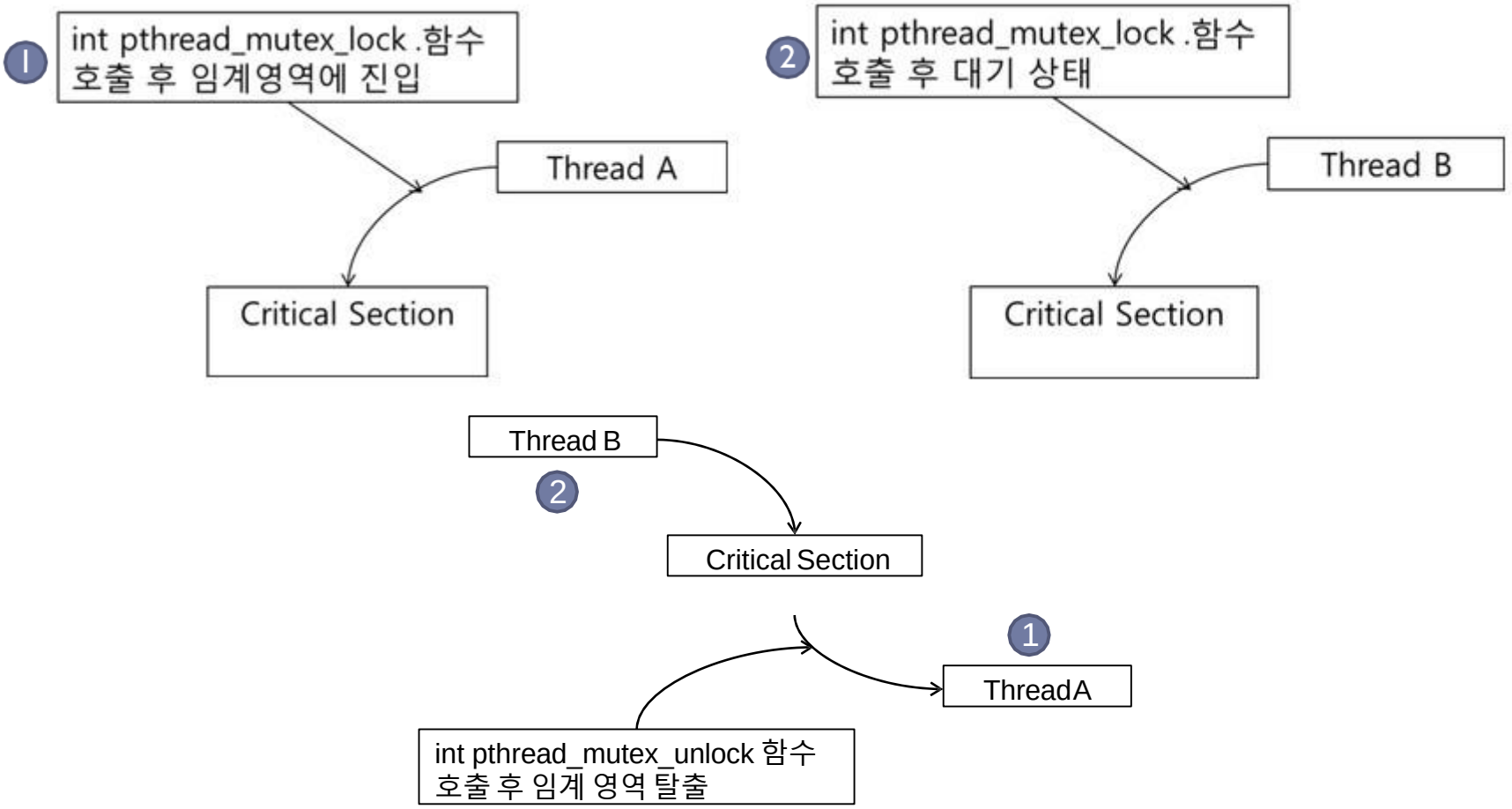
Mutex: Mutual Exclusion

- **Mutex 제약조건**
  - Mutex의 사용 카운트 값은 항상 1
  - Mutex를 얻은 곳에서만 다시 해제할 수 있음
  - Mutex를 가지고 있는 동안에는 프로세스의 종료가 불가능
  - 주어진 공식 API를 통해서만 관리할 수 있음

- **pthread의 mutex 관련 함수**

int **pthread_mutex_init** (pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)
int **pthread_mutex_lock** (pthread_mutex_t *mutex)
int **pthread_mutex_unlock** (pthread_mutex_t *mutex)
int **pthread_mutex_destroy** (pthread_mutex_t *mutex)

Mutex: Mutual Exclusion



Mutex: Mutual Exclusion

- **APIs**
  - pthread_mutex_init()
    - 사용할 mutex 변수를 초기화 함

```
#include <pthread.h>

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

    - pthread_mutex_t * mutex : 사용할 mutex 변수의 주소 값
    - const pthread_mutex_attr *attr : Mutex 속성 값. 기본 특징을 이용하고자 한다면, NULL

  - pthread_mutex_lock()
    - Mutex lock을 얻기 위해 사용
    - 이미 다른 thread가 mutex lock을 얻고 있다면 얻을 때 까지 대기함

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
```

    - pthread_mutex_t * mutex : 사용할 mutex 변수의 주소 값

Mutex: Mutual Exclusion

- **APIs (cont'd)**
  - pthread_mutex_unlock()
    - Mutex lock을 반환하고자 할 때 사용

```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

    - pthread_mutex_t * mutex : 사용할 mutex 변수의 주소 값

  - pthread_mutex_destroy()
    - 뮤텍스 객체를 제거하기 위해서 사용

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

    - pthread_mutex_t * mutex : 제거할 mutex 변수의 주소 값

Mutex: Mutual Exclusion

- **Lab. 1**

- "MUTEX"가 정의되어 있으면 mutex를 사용한다는 의미
- 이는 컴파일 시 "-DMUTEX"로 on 할 수 있음

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

void* thread_inc(void* arg);
void* thread_dec(void* arg);

#ifdef MUTEX
pthread_mutex_t mutex;
#endif

int a;
int num = 0;


int main()
{
        int state;
        pthread_t t1, t2;
        void *thread_result1, *thread_result2;
#ifdef MUTEX
        state = pthread_mutex_init(&mutex, NULL);
        if(state)
        {
                printf("mutex init error!\n");
                return 1;
        }
#endif
        pthread_create(&t1, NULL, thread_inc, "thread1");
        pthread_create(&t2, NULL, thread_dec, "thread2");
        pthread_join(t1, &thread_result1);
        pthread_join(t2, &thread_result2);

        printf("[main] %d\n", num);
#ifdef MUTEX
        pthread_mutex_destroy(&mutex);
#endif
        return 0;
}

void* thread_inc(void* arg)
{
        int i, j = 10000;
        for(i=0; i<10; ++i)
        {
#ifdef MUTEX
        pthread_mutex_lock(&mutex);
#endif
        printf("[%s] %d\n", (char*)arg, num);
        a = i;
        while(--j);
        j =  100000;
        num += a;
        printf("[%s] %d\n", (char*)arg, num);
#ifdef MUTEX
        pthread_mutex_unlock(&mutex);
#endif
        }
        return (void*)num;
}


void* thread_dec(void* arg)
{
        int i, j = 10000;
        for(i=0; i<5; ++i)
        {
#ifdef MUTEX
                pthread_mutex_lock(&mutex);
#endif
                printf("[%s] %d\n", (char*)arg, num);
                a = i + 1;
                while(--j);
                j = 10000;
                num -= a;
                printf("[%s] %d\n", (char*)arg, num);
#ifdef MUTEX
                pthread_mutex_unlock(&mutex);
#endif
        }
        return (void*)num;
}
```

Mutex: Mutual Exclusion

- **Lab. 1**

```
default:
	$(CC) -pthread -o mutex_test mutex_test.c
mutex:
	$(CC) -DMUTEX -pthread -o mutex_test mutex_test.c
```

- Mutex 없이 컴파일 한 경우

```
sslab@ubuntu:~/mutex$ make
cc -pthread -o mutex_test mutex_test.c
```

- Mutex를 사용하도록 컴파일 한 경우

```
sslab@ubuntu:~/mutex$ make mutex
cc -DMUTEX -pthread -o mutex_test mutex_test.c
```

Mutex: Mutual Exclusion

- **Result**
  - Mutex 미사용

```
sslab@ubuntu:~/mutex$ ./mutex_test
[thread1] 0
[thread2] 0
[thread1] 0
[thread1] 0
[thread2] -1
[thread2] -1
[thread2] -3
[thread2] -3
[thread2] -6
[thread2] -6
[thread2] -10
[thread2] -10
[thread2] -15
[thread1] -10
[thread1] -10
[thread1] -8
[thread1] -8
[thread1] -5
[thread1] -5
[thread1] -1
[thread1] -1
[thread1] 4
[thread1] 4
[thread1] 10
[thread1] 10
[thread1] 17
[thread1] 17
[thread1] 25
[thread1] 25
[thread1] 34
[main] 34
sslab@ubuntu:~/mutex$
```

```
sslab@ubuntu:~/mutex$ ./mutex_test
[thread2] 0
[thread2] -1
[thread2] -1
[thread2] -3
[thread2] -3
[thread2] -6
[thread2] -6
[thread2] -10
[thread2] -10
[thread2] -15
[thread1] 0
[thread1] -15
[thread1] -15
[thread1] -14
[thread1] -14
[thread1] -12
[thread1] -12
[thread1] -9
[thread1] -9
[thread1] -5
[thread1] -5
[thread1] 0
[thread1] 0
[thread1] 6
[thread1] 6
[thread1] 13
[thread1] 13
[thread1] 21
[thread1] 21
[thread1] 30
[main] 30
sslab@ubuntu:~/mutex$
```

KWANGWOON UNIVERSITY

Mutex: Mutual Exclusion

- **Result**
  - Mutex 사용

```
sslab@ubuntu:~/mutex$ ./mutex_test
[thread1] 0
[thread1] 0
[thread1] 0
[thread1] 1
[thread1] 1
[thread1] 3
[thread1] 3
[thread1] 6
[thread1] 6
[thread1] 10
[thread1] 10
[thread1] 15
[thread1] 15
[thread1] 21
[thread1] 21
[thread1] 28
[thread1] 28
[thread1] 36
[thread1] 36
[thread1] 45
[thread2] 45
[thread2] 44
[thread2] 44
[thread2] 42
[thread2] 42
[thread2] 39
[thread2] 39
[thread2] 35
[thread2] 35
[thread2] 30
[main] 30
sslab@ubuntu:~/mutex$
```

Semaphore

- **정의**
  - 여러 프로세스들이 한정된 수의 자원을 이용할 때, 한정된 수의 프로세스만 이용할 수 있게 하는 방법을 제시하는 개념

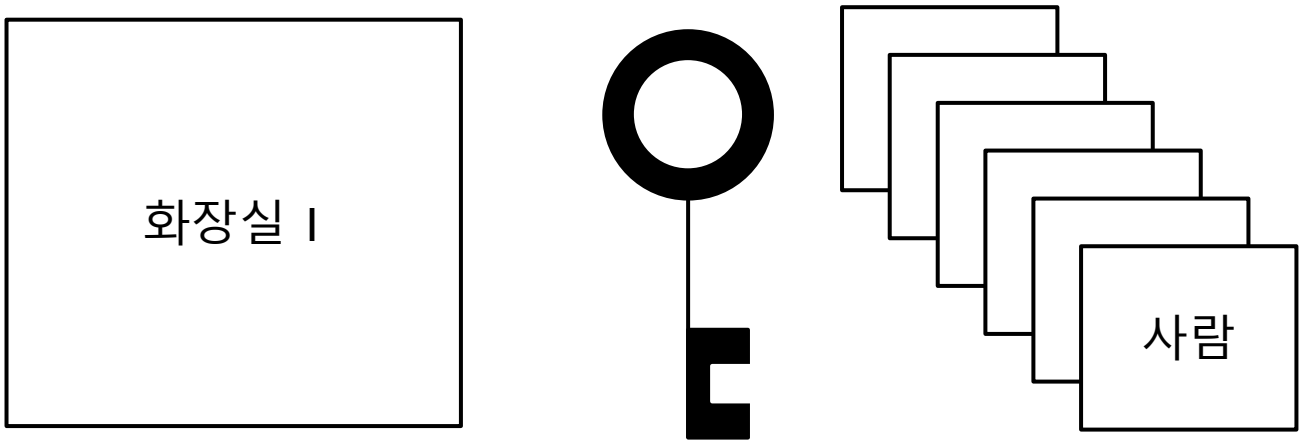
- **Linux의 세마포어**
  - 다른 태스크가 이미 가지고 있는 세마포어를 요청 시 그 태스크는 sleep
  - 오랜 시간 동안 잡게 되는 락에 적합
  - 프로세스 문맥에서만 사용
    - 인터럽트 문맥에서는 사용 불가 -> 인터럽트 문맥은 sleep 상태가 되면 안 됨

Semaphore

- **세마포어의 종류**
  - 바이너리 세마포어 (binary semaphore)
    - 공유데이터가 한 개일 경우, 0과 1값을 사용
    - 1개 이하만 동시 접근 가능
    - 뮤텍스(mutex)라고 불리기도 함
  - 카운팅 세마포어 (counting semaphore)
    - 공유할 수 있는 데이터가 둘 이상일 경우
    - 다수의 스레드의 동시 접근 허용
    - 상호 배제를 보장하지 않음

Semaphore

- **세마포어 예시 (화장실 – 열쇠 비유)**
  - 바이너리 세마포어 : 화장실1을 쓰기 위해 열쇠1을 획득
    - 작업 후 반납 열쇠가 있는 경우 0, 없는 경우 1 -> 바이너리 세마포어 (뮤택스)
    - 한 사람(thread)은 한 번에 한 화장실(shared resource)에 접근 가능
    - 세마포어 값의 범위: 0~1

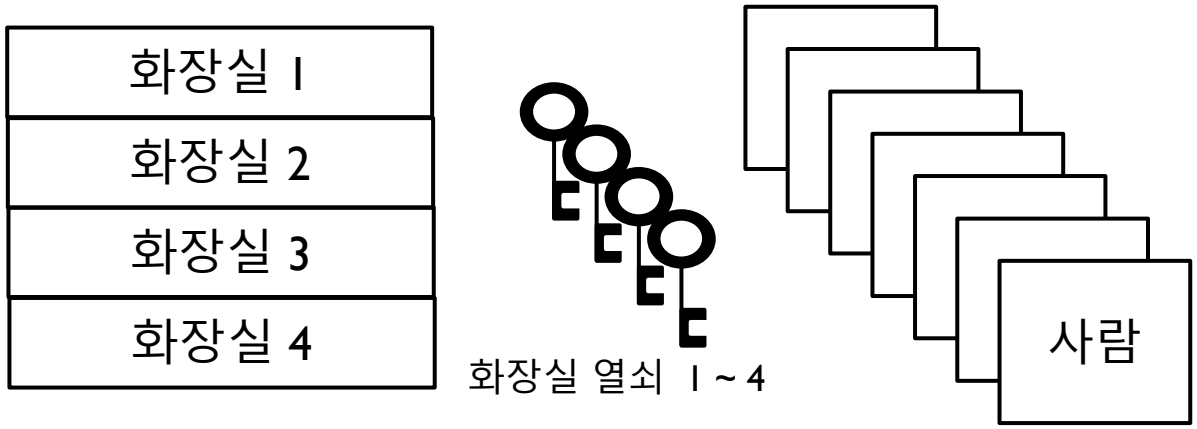


화장실 열쇠 I

세마포어 값	상태
0	열쇠 없음 (접근 불가)
1	열쇠 있음

Semaphore

- **세마포어 예시 (화장실 – 열쇠 비유)**
  - 카운팅 세마포어 : 화장실1~4 을 쓰기 위해 열쇠1~4 를 획득 후 반납
    - 사람(thread)이 각 화장실(shared resource) 에 접근 가능 -> 카운팅 세마포어
    - 여러 thread가 메모리에 접근 할 수 있음
    - 세마포어 값의 범위: 0~4



세마포어 값	상태
0	열쇠 없음 (접근 불가)
1	열쇠 1개
2	열쇠 2개
3	열쇠 3개
4	열쇠 4개

Semaphore

- **세마포어의 연산**
  - P() 연산과 V() 연산
    - 각각 lock의 잠금과 해제 역할
    - Linux에서는 각각 down()과 up()으로 사용
  - down()
    - 카운트를 1만큼 줄여서 세마포어를 얻음.
    - 만일 카운트가 0이상이면 락을 얻고 태스크가 임계구역으로 진입
    - 카운트가 음수인 경우에는 태스크가 sleep 됨
  - up()
    - 임계구역에서의 수행을 마치고 세마포어 반납
    - 카운트 값을 증가시키고, 세마포어를 기다리며 sleep 중인 태스크를 깨움

Semaphore

- **APIs**
  - int **semget** (key_t **key**, int **nsems**, int **semflg**);
    - 세마포어를 생성하거나, 존재하는 세마포어의 ID를 받기 위해 사용
    - 세마포어의 ID 값을 반환
  - key
    - 다른 세마포어와 구별하는데 사용되므로 유일한 값을 넘겨주어야 함
    - 세마포어가 생성된 후 semget() 함수로 key값을 이용하여 세마포어의 ID를 얻을 수 있음
  - nsems
    - 사용하고자 하는 세마포어의 수 (e.g. 1일 경우 binary 세마포어)
    - 기존 세마포어의 id를 얻고자 하는 경우, 0을 넘겨주면 됨
  - semflg
    - 세마포어에 접근할 때 사용하는 플래그
    - ipc_perm 구조체의 필드 설정을 통해 세마포어의 생성과 접근 권한 등을 수행할 수 있음
      - IPC_CREAT : 지정된 key 값을 이용하여 세마포어를 생성
      - IPC_EXCL : IPC_CREAT와 함께 사용
        - 만일 key를 가진 세마포어가 이미 존재하면 에러(-1)를 리턴
      - 접근권한 지정 : 일반 파일에 접근 권한을 지정하듯이 숫자의 집합을 사용

Semaphore

- **APIs (cont'd)**
  - int **semop** (int **sem_id**, struct sembuf ***ops**, unsinged **nsops**);
    - 한 세마포어 집합에 대한 일련의 연산들을 **원자적으로** 수행
    - 즉, 임계 구역에 들어가기 위해 세마포어 값을 감소시키는 작업이나, 임계구역에서 나오면서 세마포어를 증가시키는 작업을 본 함수로 수행
  - sem_id
    - 세마포어의 ID
  - ops
    - 세마포어의 값을 계산하기 위해 설정하는 값
  - nops
    - 연산의 개수

Semaphore

- Lab. 2

```c
#include <stdio.h>
#include <linux/sem.h>
#include <pthread.h>
#include <sys/types.h>

#define MAX_THREAD      5
#define MAX_RESOURCE    3

int g_semid = 0;

char *getTime(char *buffer);
void *thread_func(void *arg);


int main()
{
        int i = 0;
        union semun sem_union;
        pthread_t thread[MAX_THREAD];

        g_semid = semget((key_t)0x1234, 1, IPC_CREAT|0666);
        if(g_semid < 0)
        {
                printf("semget failed\n");
                return 1;
        }

        sem_union.val = MAX_RESOURCE;
        if(semctl(g_semid, 0, SETVAL, sem_union) == -1)
        {
                printf("semctl failed\n");
                return 1;
        }

        for(i=0; i < MAX_THREAD; ++i)
                pthread_create(&thread[i], NULL, thread_func, NULL);
        for(i=0; i < MAX_THREAD; ++i)
                pthread_join(thread[i], NULL);

        if(semctl(g_semid, 0, IPC_RMID, 0) == -1)
        {
                printf("semctl failed\n");
                return 1;
        }
        return 0;
}

char *getTime(char *buffer)
{
        time_t now = time(NULL);
        struct tm *t = localtime(&now);

        sprintf(buffer, "%02d:%02d:%02d", t-> tm_hour, t->tm_min, t->tm_sec);
        return buffer;
}

void *thread_func(void *arg)
{
        int i;
        unsigned int tid = pthread_self();
        struct sembuf s;
        char buffer[256];
        s.sem_num = 0;
        s.sem_flg = SEM_UNDO;
        for(i=0; i<5; ++i)
        {
                s.sem_op = -1;
                if(semop(g_semid, &s, 1) == -1)
                        printf("semop fail\n");
                printf("%u %s enter\n", tid, getTime(buffer));
                sleep(2);
                printf("%u %s leave\n", tid, getTime(buffer));

                s.sem_op = 1;
                if(semop(g_semid, &s, 1) == -1)
                        printf("semop fail\n");
        }
        return NULL;
}
```

Semaphore

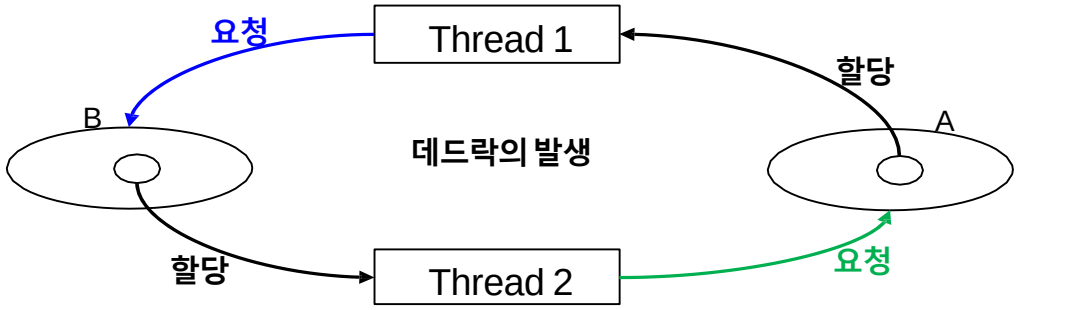
- **Result**

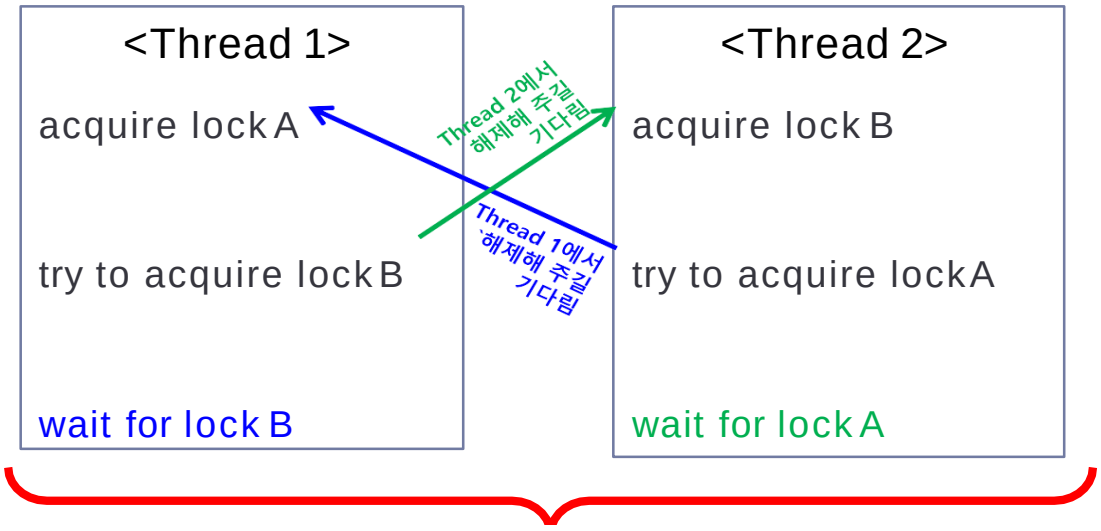
```
sslab@ubuntu:~/semaphore$ ./a.out
2312828672 18:16:14 enter
2304435968 18:16:14 enter
2296043264 18:16:14 enter
2312828672 18:16:16 leave
2304435968 18:16:16 leave
2296043264 18:16:16 leave
2312828672 18:16:16 enter
2206197504 18:16:16 enter
2214590208 18:16:16 enter
2312828672 18:16:18 leave
2304435968 18:16:18 enter
2206197504 18:16:18 leave
2296043264 18:16:18 enter
2214590208 18:16:18 leave
2312828672 18:16:18 enter
2304435968 18:16:20 leave
2296043264 18:16:20 leave
2206197504 18:16:20 enter
2214590208 18:16:20 enter
2312828672 18:16:20 leave
2304435968 18:16:20 enter
2304435968 18:16:22 leave
2296043264 18:16:22 enter
2214590208 18:16:22 leave
2312828672 18:16:22 enter
2206197504 18:16:22 leave
2304435968 18:16:22 enter
2304435968 18:16:24 leave
2312828672 18:16:24 leave
2296043264 18:16:24 leave
2304435968 18:16:24 enter
2206197504 18:16:24 enter
2214590208 18:16:24 enter
2304435968 18:16:26 leave
2312828672 18:16:26 enter
2206197504 18:16:26 leave
2296043264 18:16:26 enter
2214590208 18:16:26 leave
2206197504 18:16:26 enter
2312828672 18:16:28 leave
2296043264 18:16:28 leave
2296043264 18:16:28 enter
2214590208 18:16:28 enter
2206197504 18:16:28 leave
2206197504 18:16:28 enter
2296043264 18:16:30 leave
2214590208 18:16:30 leave
2214590208 18:16:30 enter
2206197504 18:16:30 leave
2214590208 18:16:32 leave
sslab@ubuntu:~/semaphore$
```

Semaphore

- **데드락의 정의**
  - 프로세스들이 결코 발생되지 않을 이벤트를 기다리는 상태
  - 각 프로세스들이 자원을 기다리고 있지만, 모든 자원이 이미 점유됨

- **Example**





Deadlock

- **데드락 방지 방법**
  - 중첩된 락은 반드시 같은 순서로 락/언락이 수행되어야 함
    - Example: 무조건 A->B 순서대로 락을 얻음

<Thread 1>

acquire lock A

acquire lock B

unlock B

<Thread 2>

try to acquire lock A

wait for lock A

  - 기아현상을 방지, 즉 코드가 종료되는지 확인
  - 락 설계는 최대한 단순하게

Kernel Lock Examples

- **Spin lock**

```
DEFINE_SPINLOCK(my_lock);
spin_lock (&mr_lock);
    /* Critical section */
spin_unlock (&mr_lock);
```

  - spin_lock_irqsave(), …

- **RCU(Read Copy Update)**

```
rwlock_t my_lock = RW_LOCK_UNLOCKED;
read_lock(&mr_rwlock);
    /* Critical section (read only) */
read_unlock(&mr_rwlock);
write_lock(&mr_rwlock);
    /* Critical section (read and write) */
write_unlock(&mr_rwlock);
```

- **Semaphore**

```
struct semaphore my_sem;
sema_init(&my_sem, 1);  // Count is 1
if (down(&my_sem)){
            /* Received a signal to get semaphore */
}
    /* Critical section */
up(&my_sem);
```

Appendix. Structures for Semaphore

- **Related structures**
  - struct semid_ds
    - semaphore의 집합

필드명	내용
struct ipc_perm **sem_perm**	세마포어의 접근권한
time_t **sem_otime**	최종 semop 호출시간
time_t **sem_ctime**	최종 수정시간
unsigned long int **sem_nsems**	세마포어 배열에 있는 세마포어들의 수

  - struct sem
    - semaphore 집합에 포함되는 각 semaphore

필드명	내용
unsigned short **semval**	세마포어의 값 (항상 0 이상)
pid_t **sempid**	마지막으로 연산을 수행한 프로세스의 pid

    - 세마포어 값 : 임계 구역에서 메모리에 접근을 통제하는 값
      - 0 이면 모든 메모리가 사용되어 있어 접근 불가, 통제하려 하는 메모리가 1개일 경우 binary값 을 가지고 여러 개인 경우 integer 값이 된다

Appendix. Structures for Semaphore

- **Related structures (cont'd)**
  - struct sembuf
    - semop() 함수는 이 구조체의 array를 사용

필드명	내용
unsigned short **sem_num**	array내에서 세마포어의 번호 (0, 1, …, nsems-1)
short **sem_op**	세마포어 operation (음수, 0, 양수)
short **sem_flg**	세마포어의 플래그를 세팅. 일반적으로 SEM_UNDO로 세팅. 프로세스 종료시 자동으로 세마포어 해제

  - union semun
    - semctl() 함수 등에서 사용

필드명	내용
int **val**	SETVAL을 위한 값으로 활용
short semid_ds ***buf**	IPC_STAT, IPC_SET을 위한 버퍼로 사용
unsigned short ***array**	GETALL, SETALL 명령을 위한배열
struct seminfo ***buf**	IPC_INFO를 위한 버퍼로 사용

Assignment 4

- 제출 기한: 2022. 11.10(목) ~ 2022.12.1(목) 23:59:59
- Delay 없음
- 업로드 양식에 어긋날 경우 감점 처리
- Hardcopy 제출하지 않음