



2016년 2학기 운영체제실습 10주차

Synchronization

Dept. of Computer Engineering,
Kwangwoon Univ.

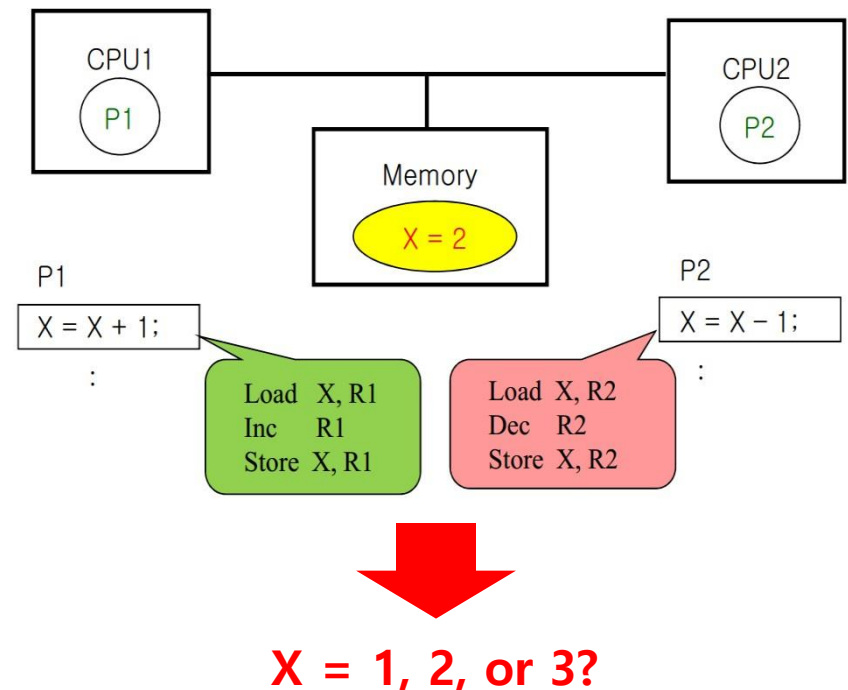
Contents

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

Synchronization

▶ 동기화의 필요성

- ▶ 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
 - ▶ 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

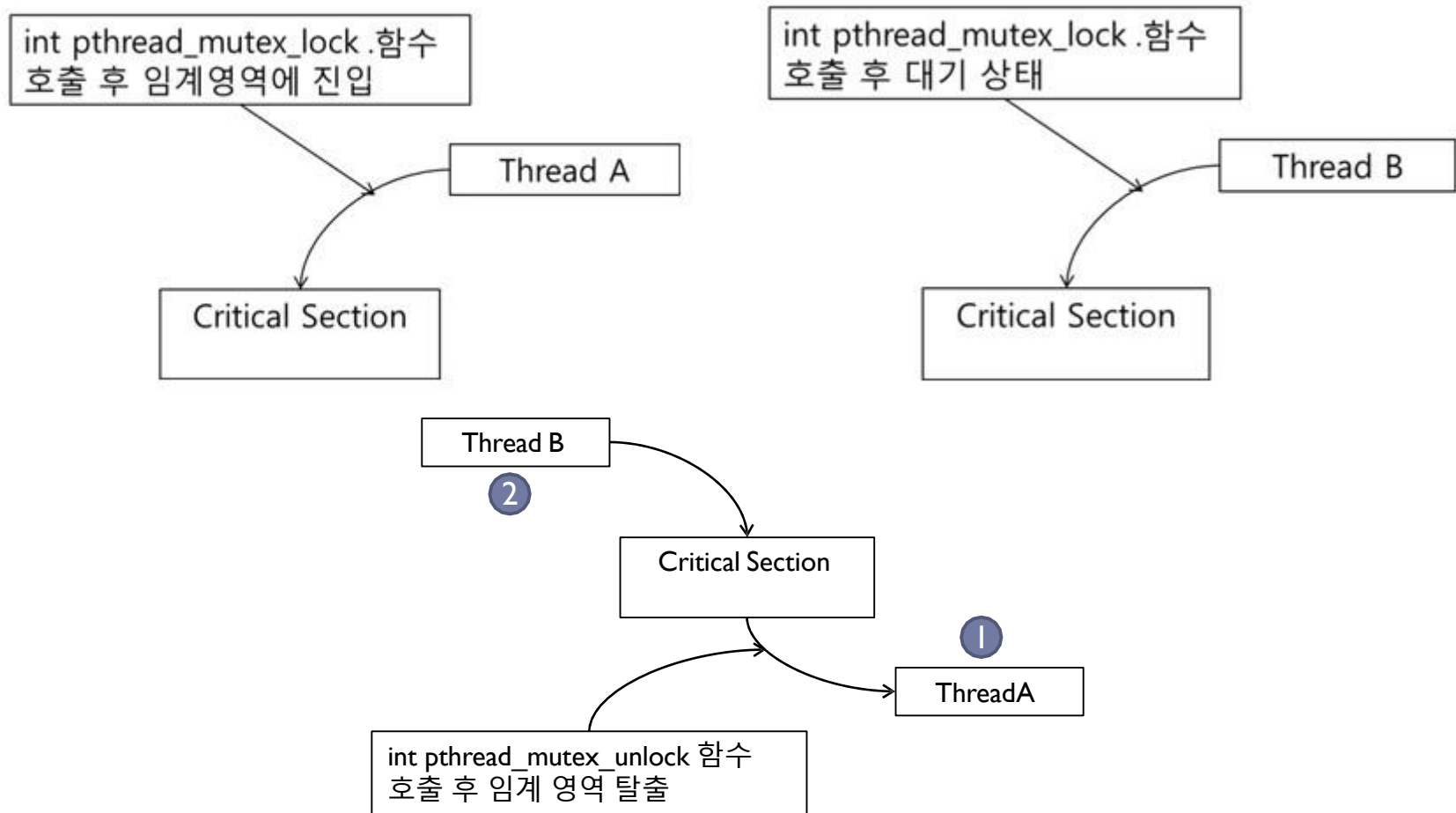
▶ 특징

- ▶ 문맥 교환이 필요하지 않음
- ▶ 바쁜 루프로 인한 CPU 소모
- ▶ 짧은 기간 대기에 효과적임
- ▶ Linux에서는 SMP 환경에서만 사용
 - ▶ **Symmetric Multi-Processing (SMP)**
 - 두 개 이상의 프로세서가 한 개의 공유된 메모리를 사용하는 구조

Mutex: Mutual Exclusion

▶ Mutual Exclusion

- ▶ 여러 thread 간 동시간 접근을 허용하지 않도록 하는 것 (lock을 얻지 못하면 대기상태)



Mutex: Mutual Exclusion

▶ Mutual Exclusion

- ▶ 여러 thread 간 동시간 접근을 허용하지 않도록 하는 것

▶ 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

▶ 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()

- ▶ 뮤텝스 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 (p.11 참고)

```
ssangkong — ssangkong@ssangkong-sslab: ~ — ssh — 126x41

1 #include <stdio.h>
2 #include <pthread.h>
3
4 void* thread_inc(void* arg);
5 void* thread_dec(void* arg);
6
7 #ifdef MUTEX
8 pthread_mutex_t mutex;
9 #endif
10 int a;
11 int num = 0;
12
13 int main()
14 {
15     int state;
16     pthread_t t1, t2;
17     void* thread_result1, *thread_result2;
18     #ifdef MUTEX
19         state = pthread_mutex_init(&mutex, NULL);
20         if( state )
21         {
22             printf("mutex init error!\n");
23             return 1;
24         }
25     #endif
26     pthread_create(&t1, NULL, thread_inc, "thread1");
27     pthread_create(&t2, NULL, thread_dec, "thread2");
28     pthread_join(t1, &thread_result1);
29     pthread_join(t2, &thread_result2);
30
31     printf("[main] %d\n", num);
32     #ifdef MUTEX
33         pthread_mutex_destory(&mutex);
34     #endif
35     return 0;
36 }
37
38 void* thread_inc(void* arg)
39 {
40     int i, j = 10000;
41     for( i=0 ; i<10 ; ++i )
42     {
43         #ifdef MUTEX
44             pthread_mutex_lock(&mutex);
45         #endif
46         printf("[%s] %d\n", (char*)arg, num);
47         a = i;
48         while( --j );
49         j = 10000;
50         num += a;
51         printf("[%s] %d\n", (char*)arg, num);
52     }
53     #ifdef MUTEX
54         pthread_mutex_unlock(&mutex);
55     #endif
56     return (void*)num;
57 }
58
59 void* thread_dec(void* arg)
60 {
61     int i, j = 10000;
62     for( i=0 ; i<5 ; ++i )
63     {
64         #ifdef MUTEX
65             pthread_mutex_lock(&mutex);
66         #endif
67         printf("[%s] %d\n", (char*)arg, num);
68         a = i + 1;
69         while( --j );
70         j = 10000;
71         num -= a;
72         printf("[%s] %d\n", (char*)arg, num);
73     }
74     #ifdef MUTEX
75         pthread_mutex_unlock(&mutex);
76     #endif
77     return (void*)num;
78 }
```

mutex_test.c 31,2-5 Top mutex_test.c 71,1-4 Bot

Mutex: Mutual Exclusion

▶ Lab. 1 (cont'd)

```
ssangkong — ssangkong@ssangkong-sslab: ~ — ssh — 74x7
1 default:
2  $(CC) -pthread -o mutex_test mutex_test.c
3 mutex:
4  $(CC) -DMUTEX -pthread -o mutex_test mutex_test.c
5
Makefile 1,1 All
:wq
```

Mutex 없이
컴파일 한 경우

```
ssangkong@ssangkong-sslab:~$ make
cc -pthread -o mutex_test mutex_test.c
ssangkong@ssangkong-sslab:~$ ./mutex_test
[thread1] 0
[thread1] 0
[thread1] 0
[thread2] 0
[thread1] 1
[thread1] 1
[thread2] -1
[thread2] -1
[thread1] 1
[thread1] 1
[thread2] -2
[thread2] -2
[thread1] 1
[thread1] 1
[thread2] -3
[thread2] -3
[thread1] 1
[thread1] 1
[thread2] -4
[thread2] -4
[thread1] 1
[thread1] 1
[thread2] -5
[thread1] 1
[thread1] 1
[thread1] 8
[thread1] 8
[thread1] 16
[thread1] 16
[thread1] 25
[main] 25
ssangkong@ssangkong-sslab:~$
```

```
ssangkong@ssangkong-sslab:~$ ./mutex_test
[thread1] 25
[main] 25
ssangkong@ssangkong-sslab:~$ ./mutex_test
[thread1] 0
[thread1] 0
[thread1] 0
[thread2] 0
[thread2] 0
[thread2] -2
[thread2] -2
[thread1] 1
[thread1] -2
[thread2] -4
[thread2] -4
[thread1] 0
[thread1] 0
[thread2] -3
[thread2] -3
[thread1] 2
[thread1] 2
[thread2] -2
[thread1] 2
[thread1] 2
[thread1] 7
[thread1] 7
[thread1] 13
[thread1] 13
[thread1] 20
[thread1] 20
[thread1] 28
[thread1] 28
[thread1] 37
[main] 37
ssangkong@ssangkong-sslab:~$
```

Mutex: Mutual Exclusion

▶ Lab. 1 (cont'd)

Mutex를 사용하도록
컴파일 한 경우

```
ssangkong@ssangkong-sslab:~$ make mutex
cc -DMUTEX -pthread -o mutex_test mutex_test.c
ssangkong@ssangkong-sslab:~$ ./mutex_test
[thread1] 0
[thread1] 0
[thread1] 0
[thread1] 1
[thread1] 1
[thread1] 3
[thread1] 3
[thread1] 6
[thread2] 6
[thread2] 5
[thread2] 5
[thread2] 3
[thread2] 3
[thread2] 0
[thread2] 0
[thread2] -4
[thread2] -4
[thread2] -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
ssangkong@ssangkong-sslab:~$
```

```
ssangkong@ssangkong-sslab:~$ ./mutex_test
[thread1] 30
[main] 30
ssangkong@ssangkong-sslab:~$ ./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
ssangkong@ssangkong-sslab:~$
```

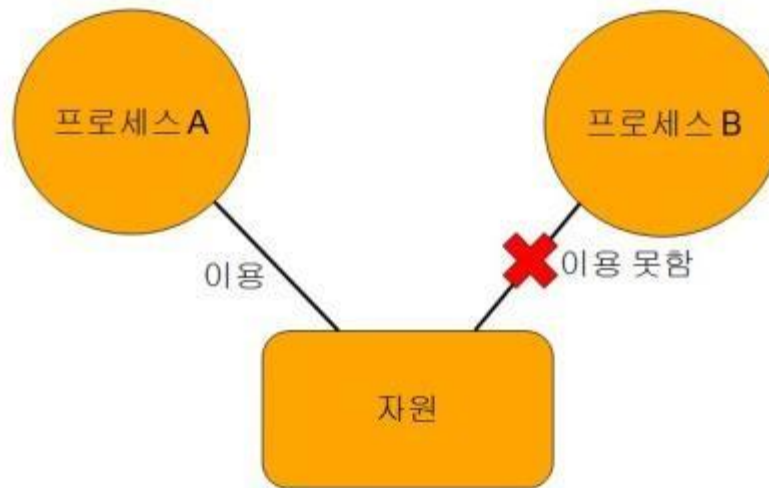
Semaphore

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 요청시 그 태스크는 휴면
- 오랜 시간 동안 잡게 되는 락에 적합
- 프로세스 문맥에서만 사용
 - 인터럽트 문맥에서는 사용 불가 → 인터럽트 문맥은 휴면상태가 되면 안 됨



Semaphore

▶ 세마포어의 종류

- ▶ 세마포어는 어느 순간에 여러 스레드의 동시 접근을 허용함.
- ▶ **바이너리** 세마포어 (binary semaphore)
 - ▶ 공유데이터가 한 개일 경우, 0과 1값을 사용
 - ▶ **1개 이하만 동시 접근** 가능
 - ▶ 뮤텝스(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)

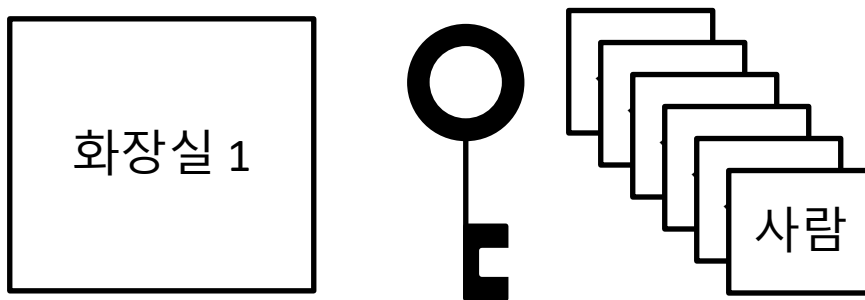
▶ 바이너리 세마포어

: 화장실1을 쓰기 위해 열쇠1을 획득

➔ 작업 후 반납 열쇠가 있는 경우 0, 없는 경우 1 ➔ 바이너리 세마포어 (뮤텍스)

➔ 한 사람(thread)은 한 번에 한 화장실(shared resource)에 접근 가능

➔ 세마포어 값의 범위: 0~1



화장실 열쇠 1

세마포어 값	상태
0	열쇠 없음 (접근 불가)
1	열쇠 있음

Semaphore

▶ 세마포어 예시 (화장실 - 열쇠 비유)

▶ 카운팅 세마포어

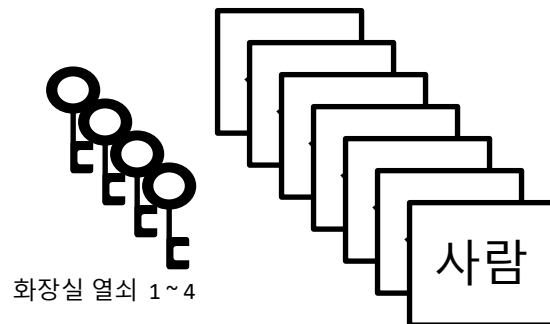
: 화장실1~4 을 쓰기 위해 열쇠1~4 를 획득 후 반납

➔ 사람(thread)이 각 화장실(shared resource) 에 접근 가능 ➔ 카운팅 세마포어

➔ 여러 thread가 메모리에 접근 할 수 있음

➔ 세마포어 값의 범위: 0~4

화장실 1
화장실 2
화장실 3
화장실 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 휴면 됨
- ▶ up()
 - ▶ 임계구역에서의 수행을 마치고 세마포어 반납
 - ▶ 카운트 값을 증가시키고, 세마포어를 기다리며 휴면 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 `key`값을 이용하여 semaphore ID를 얻을 수 있음

- ▶ `nsems`
 - 사용하고자 하는 세마포어의 수 (e.g. 1일 경우 binary semaphore)
 - 기존 세마포어의 id를 얻고자 하는 경우, 0을 넘겨주면 됨

- ▶ `semflg`
 - Semaphore에 접근할 때 사용하는 플래그
 - `ipc_perm` 구조체의 필드 설정을 통해 semaphore의 생성과 접근 권한 등을 수행할 수 있음
 - `IPC_CREAT` : 지정된 `key` 값을 이용하여 semaphore를 생성
 - `IPC_EXCL` : `IPC_CREAT`와 함께 사용. 만일 `key`를 가진 semaphore가 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, unsigned 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`
 - `semget`에서 얻어온 semaphore의 ID
- ▶ `ops`
 - semaphore에 대해 수행하고자 하는 연산을 지정한 `sembuf` 타입의 배열에 대한 포인터.
- ▶ `nsops`
 - 연산의 개수

Semaphore

▶ APIs (cont'd)

- ▶ `int semctl (int sem_id, int semnum, int cmd, union semun arg);`
 - ▶ Semaphore를 제어하는 함수. Semaphore을 초기화 할 때도 사용.
 - ▶ `sem_id`
 - `semget`을 통해 얻어온 semaphore의 ID
 - ▶ `semnum`
 - semaphore의 멤버 번호
 - ▶ `cmd`
 - semaphore에 수행할 명령. 아래 표 참고.

Name	Function
IPC_STAT	<code>semid_ds</code> 구조체를 <code>arg.buf</code> 에 저장하여 semaphore 집합의 정보 획득
IPC_SET	<code>arg.buf</code> 의 <code>semid_ds</code> struct가 가진 내용을 바탕으로 semaphore의 접근 권한을 변경
IPC_RMID	semaphore 삭제
GETVAL	<code>semnum</code> 으로 지정된 <code>sem.semval</code> 값을 획득
SETVAL	<code>semnum</code> 으로 지정된 <code>sem.semval</code> 값을 <code>arg.val</code> 로 설정
GETPID	마지막으로 <code>semop()</code> 함수를 실행한 프로세스의 PID 획득
GETALL	모든 semaphore 값을 얻은 뒤, <code>arg.array</code> 가 가리키는 배열에 저장
SETALL	모든 semaphore의 값을 <code>arg.array</code> 가 가리키는 배열의 값으로 세팅
GETNCNT	<code>semnum</code> 으로 지정된 <code>sem.semval</code> 이 현재 값보다 커지길 기다리는 프로세스 개수
GETZCNT	<code>semnum</code> 으로 지정된 <code>sem.semval</code> 이 0이 되길 기다리는 프로세스 개수

Semaphore

▶ Lab. 2

```
ssangkong — ssangkong@ssangkong-sslab: ~ — ssh — 157x42

1 #include <stdio.h>
2 #include <linux/sem.h>
3 #include <pthread.h>
4 #include <sys/types.h>
5
6 #define MAX_THREAD    5
7 #define MAX_RESOURCE   3
8
9 int g_semid = 0;
10
11 char* getTime(char *buffer);
12 void* thread_func(void *arg);
13
14 int main()
15 {
16     int i = 0;
17     union semun sem_union;
18     pthread_t thread[MAX_THREAD];
19
20     g_semid = semget((key_t)0x1234, 1, 0666 | IPC_CREAT);
21     if( g_semid < 0 )
22     {
23         printf("semget failed\n");
24         return 1;
25     }
26
27     sem_union.val = MAX_RESOURCE;
28     if( semctl(g_semid, 0, SETVAL, sem_union) == -1 )
29     {
30         printf("semctl failed\n");
31         return 1;
32     }
33
34     for( i=0 ; i<MAX_THREAD ; ++i )
35         pthread_create(&thread[i], NULL, thread_func, NULL);
36
37     for( i=0 ; i<MAX_THREAD ; ++i )
38         pthread_join(thread[i], NULL);
39
40     if( semctl(g_semid, 0, IPC_RMID, 0) == -1 )
41     {
42         printf("semctl failed\n");
43         return 1;
44     }
45     return 0;
46 }
47
48 char* getTime(char *buffer)
49 {
50     time_t now = time(NULL);
51     struct tm *t = localtime(&now);
52
53     sprintf(buffer, "%02d:%02d:%02d", t->tm_hour, t->tm_min, t->tm_sec);
54     return buffer;
55 }
56
57 void* thread_func(void *arg)
58 {
59     int i;
60     unsigned int tid = pthread_self();
61     struct sembuf s;
62     char buffer[256];
63     s.sem_num = 0;
64     s.sem_flg = SEM_UNDO;
65     for( i=0 ; i<5 ; ++i )
66     {
67         s.sem_op = -1;
68         if( semop(g_semid, &s, 1) == -1 )
69             printf("semop fail\n");
70
71         printf("%u %s enter\n", tid, getTime(buffer));
72         sleep(2);
73         printf("%u %s leave\n", tid, getTime(buffer));
74
75         s.sem_op = 1;
76         if( semop(g_semid, &s, 1) == -1 )
77             printf("semop fail\n");
78     }
79     return NULL;
80 }
```

세마포어 제거

세마포어 생성

세마포어 수 변경

P연산 수행

V연산 수행

semaphore_test.c 12,1 Top semaphore_test.c 79,1-4 Bot

Deadlock

▶ 데드락의 정의

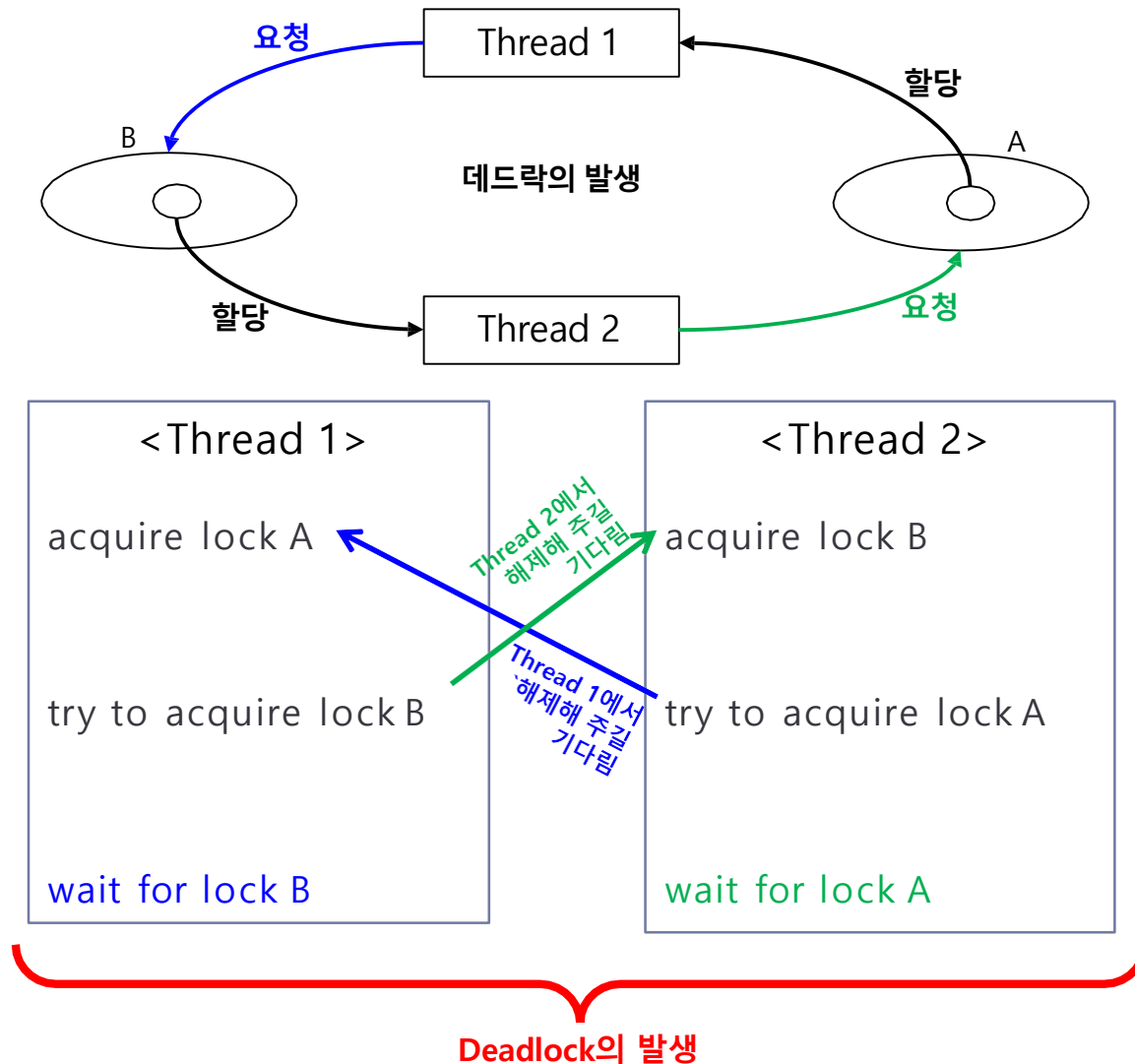
- ▶ 프로세스들이 결코 발생되지 않을 이벤트를 기다리는 상태

▶ 데드락 방지 방법

- ▶ 중첩된 락은 반드시 같은 순서로 잠겨져야 함.
- ▶ 기아현상을 방지, 즉 코드가 과연 종료하는지 확인.
- ▶ 락 설계는 최대한 단순하게.

Deadlock

▶ Example



Appendix. Structures for Semaphore

▶ Related structures

▶ struct semid_ds

▶ semaphore의 집합

필드명	내용
struct ipc_perm sem_perm	세마포어의 접근 권한
time_t sem_otime	최종 semop 호출 시간
time_t sem_ctime	최종 수정 시간
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 위한 배열
struct seminfo *buf	IPC_INFO를 위한 버퍼로 사용