7주차 2차시 스레딩 동기화

【학습목표】

1. 스레딩 동기화의 개념을 설명할 수 있다.
2. 데드락의 개념을 설명할 수 있다.

학습내용1 : 스레딩 동기화

1. 필요성

하나의 프로세스에서 2개의 스레드가 같이 공유하고 있는 자원(code, data, files)에 동시에 접근하는 것을 막아야 함.
하나의 자원(code, data, files)에 하나의 스레드가 접근하도록 제어 필요
다수의 스레드가 하나의 자원에 순차적으로 접근할수 있도록 제어 필요

2. 임계영역

서로 공유하는 자원들을 순차적으로 접근하기 위한 제어 공간
다수의 프로세스 및 스레드가 접근 가능하지만 어느 한 순간에서는 프로세스 및 스레드 하나만 사용 가능

3. 동기화 기법

① 상호배제(MUTEX : MUTual EXclusion)
- pthread_mutex_t 타입의 변수를 뮤텍스라고 표현함
- 한 스레드가 자원을 사용하기 위해 임계영역(Critical Section)으로 들어가면 다른 스레드는 임계영역으로 들어가지 못하도록 하는 기법
- 작동 기본 원리 : 임계영역에 들어갈 때 뮤텍스를 잠그고 들어감,
임계영역에서 빠져 나올 때 뮤텍스를 풀고 나감

* 상호배제의 조건
두 프로세스는 동시에 공유 자원에 진입 불가.
프로세스의 속도나 프로세서 수에 영향 받지 않음
공유 자원을 사용하는 프로세스만 다른 프로세스 차단 가능
프로세스가 공유 자원을 사용하려고 너무 오래 기다려서는 안 됨

* 주의점
Deadlock
mutex의 파괴
다른 쓰레드가 lock한 뮤텍스를 unlock하는 상황

② Lock and unlock(Binary Semaphore)
세마포 S를 상호배제에 사용, 1 또는 0으로 초기화, P와 V의 연산 교대 실행

4. 세마포어

① 세마포어란?
- 프로세스 사이의 동기를 맞추는 기능 제공
- 한 번에 한 프로세스만 작업을 수행하는 부분에 접근해 잠그거나, 다시 잠금을 해제하는 기능을 제공하는 정수형 변수
- 세마포어를 처음 제안한 에츠허르 데이크스트라가 사용한 용어에 따라 잠금함수는 p로 표다익스트라가 테스 명령어의 문제 해결을 위해 제안, 세마포어 값은 true나 false로, P와 V연산과 관련. 네덜란드어로 P는 검사Proberen, V 증가Verhogen 의미. 음이 아닌 정수 플래그 변수
- 세마포를 의미하는 S는 표준 단위 연산 P(프로세스 대기하게 하는 wait 동작, 임계 영역에 진입하는 연산)와 V(대기 중인 프로세스 깨우려고 신호 보내는 signal 동작, 임계 영역에서 나오는 연산)로만 접근하는 정수 변수시하고 해제함수는 v로 표시

② 세마포어 기본 동작 구조
중요 처리부분(critical section)에 들어가기 전에 p 함수를 실행하여 잠금 수행
처리를 마치면 v 함수를 실행하여 잠금 해제

```
p(sem);  // 잠금
중요한 처리 부분
v(sem);  // 잠금 해제
```

③ p 함수의 기본 동작 구조

```
p(sem) {
    while sem=0 do wait;
    sem 값을 1 감소;
}
```

sem의 초기값은 1
sem이 0이면 다른 프로세스가 처리부분을 수행하고 있다는 의미이므로 1이 될 때까지 기다린다.
sem이 0이 아니면 0으로 만들어 다른 프로세스가 들어오지 못하게 함

④ v 함수의 기본 동작 구조

```
v(sem) {
    sem 값을 1 증가;
    if (대기중인 프로세스가 있으면)
        대기중인 첫 번째 프로세스를 동작시킨
}
```

5. 세마포어 관련 함수

① 세마포어 생성: semget(2)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

nsems : 생성할 세마포어 개수
semflg : 세마포어 접근 속성 (IPC_CREAT, IPC_EXCL)

* semid_ds 구조체

```
struct semid_ds {
    struct ipc_perm sem_perm;
    struct sem *sem_base;
    ushort_t sem_nsems;
    time_t sem_otime;
    int32_t sem_pad1;
    time_t sem_ctime;
    int32_t sem_pad2;
    int sem_binary;
    long sem_pad3[3];
};
```

sem_perm: IPC공통 구조체
sem_base: 세마포어 집합에서 첫번째 세마포어의 주소
sem_nsems: 세모포어 집합에서 세마포어 개수
sem_otime: 세마포어 연산을 수행한 마지막시간
sem_ctime: 세마포어 접근권한을 마지막으로 변경한 시간
sem_binary: 세마포어 종류를 나타내는 플래그

* sem 구조체
; 세마포어 정보를 저장하는 구조체

```
struct sem {
    ushort_t    semval;
    pid_t       sempid;
    ushort_t    semncnt;
    ushort_t    semzcnt;
    kcondvar_t  semncnt_cv;
    kcondvar_t  semzcnt_cv;
};
```

semval : 세마포어 값
sempid : 세마포어 연산을 마지막으로 수행한 프로세스 PID
semncnt: 세마포어 값이 현재 값보다 증가하기를 기다리는 프로세스 수
semzcnt: 세마포어 값이 0이 되기를 기다리는 프로세스 수

② 세마포어 제어: semctl(2)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```

semnum : 기능을 제어할 세마포어 번호
cmd : 수행할 제어 명령
... : 제어 명령에 따라 필요시 사용할 세마포어

* 공용체 주소(선택사항)

```
union semun {
    int             val;
    struct semid_df * buf;
    ushort_t        *array;
} arg;
```

* cmd에 지정할 수 있는 값
IPC_RMID, IPC_SET, IPC_STAT : 메시지 큐, 공유 메모리와 동일 기능
GETVAL : 세마포어의 semval 값을 읽어온다.
SETVAL : 세마포어의 semval 값을 arg.val로 설정한다.
GETPID : 세마포어의 sempid 값을 읽어온다.
GETNCNT, GETZNCNT : 세마포어의 semncnt, semzcnt 값을 읽어온다.
GETALL : 세마포어 집합에 있는 모든 세마포어의 semval 값을 arg.array에 저장
SETALL : 세마포어 집합에 있는 모든 세마포어의 semval 값을 arg.array의 값으로 설정

③ 세마포어 연산: semop(2)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, size_t nsops);
```

sedmid : semget()함수로 생성한 세마포어 식별자
sops : sembuf 구조체 주소
nsops : sops가 가리키는 구조체 크기

세마포어 연산을 의미

```
struct sembuf {
    ushort_t    sem_num;
    short       sem_op;
    short       sem_flg;
};
```

* 세마포어 연산
sembuf 구조체의 sem_op 항목에 지정

```
if (sem_op < 0) {              /* 세마포어 잠금 */
    wait until semval >= | sem_op |;
    semval -= | sem_op |;
}
else if (sem_op > 0)           /* 세마포어 잠금 해제 */
    semval += sem_op;
else
    wait until semval is 0;
```

* 1. sem_op가 음수 : 세마포어 잠금 기능 수행
- semval 값이 sem_op의 절댓값과 같거나 크면 semval 값에서 sem_op의 절댓값을 뺀다.
- semval 값이 sem_op 값보다 작고 sem_flg에 IPC_NOWAIT가 설정되어 있으면 semop 함수는 즉시 리턴
- semval 값이 sem_op 값보다 작은데 sem_flg에 IPC_NOWAIT가 설정되어 있지 않으면 semop 함수는 semncnt 값을 증가시키고 다음 상황을 기다린다.
- semval 값이 sem_op의 절대값보다 같거나 커진다. 이 경우 semncnt 값은 감소하고 semval 값에서 sem_op의 절대값을 뺀다.
- 시스템에서 semid가 제거된다. 이 경우 errno가 EIDRM으로 설정되고 -1을 리턴한다.
- semop 함수를 호출한 프로세스가 시그널을 받는다. 이 경우 semncnt 값은 감소하고 시그널 처리함수를 수행한다.

* 2. sem_op가 양수면 이는 세마포어의 잠금을 해제하고 사용중이던 공유자원을 돌려준다. 이 경우 sem_op 값이 semval 값에 더해진다.

* 3. sem_op 값이 0일 경우
- semval 값이 0이면 semop 함수는 즉시 리턴한다.
- semval 값이 0이 아니고, sem_flg에 IPC_NOWAIT가 설정되어 있으면 semop 함수는 즉시 리턴한다.
- semval 값이 0이 아니고, sem_flg에 IPC_NOWAIT가 설정되어 있지 않으면 semop 함수는 semzcnt 값을 증가시키고 semval 값이 0이 되길 기다린다.

[예제 10-7] (1) 세마포어 생성과 초기화 ex10_7.c

```
...
09  union semun {
10      int val;
11      struct semid_ds *buf;
12      unsigned short *array;
13  };
14
15  int initsem(key_t semkey) {
16      union semun semunarg;
17      int status = 0, semid;
18
19      semid = semget(semkey, 1, IPC_CREAT | IPC_EXCL | 0600);
20      if (semid == -1) {
21          if (errno == EEXIST)
22              semid = semget(semkey, 1, 0);
23      }
24      else {
25          semunarg.val = 1;
26          status = semctl(semid, 0, SETVAL, semunarg);
27      }
28
29      if (semid == -1 || status == -1) {
30          perror("initsem");
31          return (-1);
32      }
33
34      return semid;
35  }
```

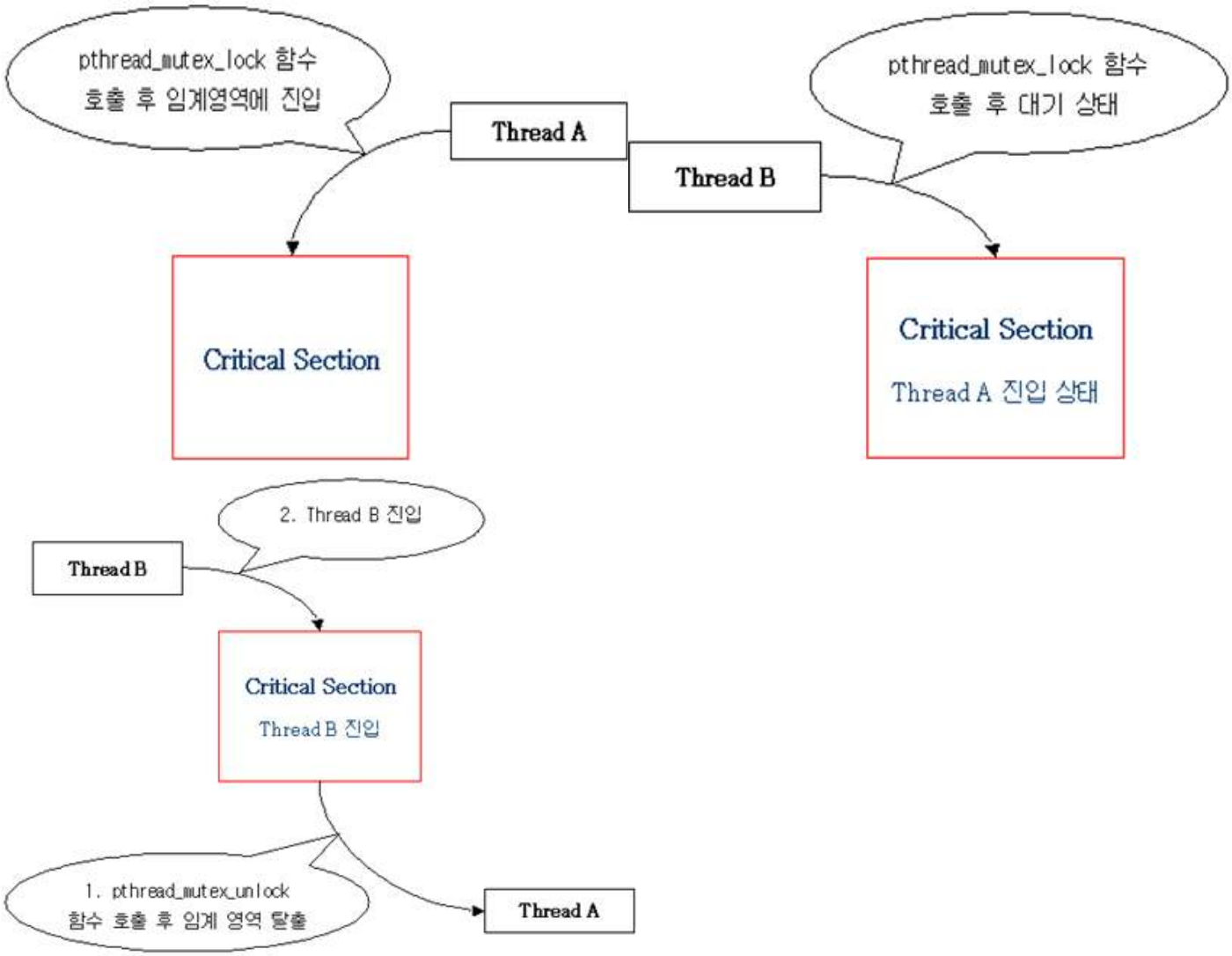
semun 공용체 선언

세마포어 생성 및 초기화 함수

세마포어 생성

세마포어 값을 1로 초기화

6. MUTEX 동기화 원리



7. 뮤텍스 조작 함수

기능	함수명
뮤텍스 초기화 함수	pthread_mutex_init()
뮤텍스 소멸 함수	pthread_mutex_destroy()
뮤텍스 잠금 함수	pthread_mutex_lock()
뮤텍스 잠금 해제 함수	pthread_mutex_unlock()

① Pthread_mutex_init()
사용 : 뮤텍스를 초기화
성공 시 : 0
실패 시 : 0이 아닌 errno 설정

errno	원인
EAGAIN	초기화할 만한 메모리 이외의 자원이 부족
ENOMEM	초기화할 만한 메모리가 부족
EPERM	호출자가 적절한 권한을 가지고 있지 않음

```
#include <pthread.h>

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

mutex : 초기화 할 뮤텍스 객체의 포인터
attr : 뮤텍스에 지정할 속성, dafault는 NULL

```
int error;
pthread_mutex_t mylock;

if(error = pthread_mutex_init(&mylock, NULL))
    fprintf(stderr,"Failed to init mylock");
```

② Pthread_mutex_destroy()
사용 : 뮤텍스를 파괴
성공 : 0
실패 : non-zero 리턴, errno 설정

errno	원인
EBUSY	뮤텍스가 이미 lock되어 있음

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Mutex : 제거 할 뮤텍스 객체의 포인터

```
int error;
pthread_mutex_t mylock;

if(error = pthread_mutex_destory(&mylock))
    fprintf(stderr, "Failed to destroy mylock");
```

③ Pthread_mutex_lock()
사용 : 뮤텍스를 lock 시킴
만약 뮤텍스가 lock된 상태라면 unlock 될때까지 block됨
성공 시 : 0 리턴
실패 시 : non-zero 리턴, errno 설정

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Mutex : lock 시킬 뮤텍스 객체의 포인터

```
int error;
pthread_mutex_t mylock;

if(error = pthread_mutex_lock(&mylock))
    fprintf(stderr, "Failed to lock mylock");
```

④ Pthread_mutex_trylock
사용 : pthread_mutex_lock의 non-blocking버젼
성공 : 0 리턴
실패 : non-zero 리턴, errno 설정

```
#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

mutex : trylock 시킬 뮤텍스 객체의 포인터

```
int error;
pthread_mutex_t mylock;

if(error = pthread_mutex_trylock(&mylock))
    fprintf(stderr, "Failed to trylock mylock");
```

⑤ Pthread_mutex_unlock()
사용 : lock된 뮤텍스를 unlock 시킴
성공 : 0 리턴
실패 : non-zero 리턴, errno 설정

```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

mutex : unlock 시킬 뮤텍스 객체의 포인터

```
int error;
pthread_mutex_t mylock;

if(error = pthread_mutex_unlock(&mylock))
    fprintf(stderr, "Failed to unlock mylock");
```

⑥ 예제

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <pthread.h>
5
6  Void *thread_increment (void *arg);
7  Char thread1[] = “A Thread”;
8  Char thread2[] = “B Thread”;
9
10 pthread_mutex_t mutx;
11 int number = 0;
12 int main(int argc, char **argv) {
13       pthread_t t1, t2;
14       void *thread_result;
15       int state;
16
17       state = pthread_mutex_init(&mutex, NULL);
18       if(state) {
19                puts(“Failed to init mutex”);
20                exit(1);
21       }
22
23       pthread_create(&t1, NULL, thread_increment, &thread1);
24       pthread_create(&t1, NULL, thread_increment, &thread1);
25
26       pthread_join(t1, &thread_result);
27       pthread_join(t2, &thread_result);
28
29       printf(“final number : %d ₩n”,number);
30       pthread_mutex_destroy(&mutx);
31       Return 0;
32 }
33
34 void *thread_increment(void *arg)
35 {
36       int i;
37       for(i = 0; i < 5; i++) {
38                pthread_mutex_lock(&mutx);
39                sleep(1);
number++;
41                printf(“execute : %s, number : %d ₩n”,(char*)arg,                number);
```

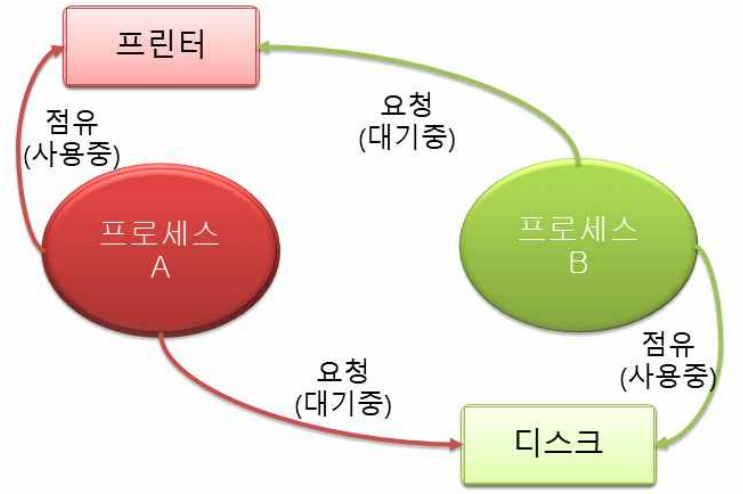
```
42                  pthread_mutex_unlock(&mutx);
43          }
44 }
```

```
#gcc -lpthread exThr_3.c -o exThr_3
#./exThr_3
execute : A Thread, number : 1
execute : B Thread, number : 2
execute : A Thread, number : 3
execute : B Thread, number : 4
execute : A Thread, number : 5
execute : B Thread, number : 6
execute : A Thread, number : 7
execute : B Thread, number : 8
execute : A Thread, number : 9
execute : B Thread, number : 10
final number : 10
```

학습내용2 : 데드락

1. deadlock(교착 상태)란?

① 둘 이상의 프로세스가 서로 남이 가진 자원을 요구하면서 양쪽 모두 작업 수행을 할 수 없이 대기 상태로 놓이는 상태.
② 다중 프로그래밍 시스템에서 일어날 수 있는 현상
③ 두 프로세스가 사용하는 자원(비공유) 서로 기다리고 있을 때 발생
④ 프린터를 점유(사용)하고 있는 프로세스 A와 디스크를 점유(사용)하고 있는 프로세스B가 서로 상대방의 자원(프린터, 디스크)를 요구하면서 기다리는 현상
⑤ A태스크가 B라는 태스크의 종료 후에 실행되기 위해 대기 상태에 있을 때 B라는 태스크도 A의 종료 후에 실행을 종료시키는 상태에 있으면 모두 대기 상태가 된 채 언제까지나 실행이 시작되지 않아 컴퓨터가 마치 정지해 있는 상태



2. 프로세스의 자원 사용 순서

자원 요청 : 프로세스가 필요한 자원 요청. 해당 자원 다른 프로세스가 사용 중이면 요청을 수락 때까지 대기.
자원 사용 : 프로세스가 요청한 자원 획득하여 사용
자원 해제 : 프로세스가 자원 사용 마친 후 해당 자원 되돌려(해제) 줌

3. 교착 상태 발생의 네가지 조건

① 상호배제
자원을 최소 하나 이상 비공유. 즉, 한 번에 프로세스 하나만 해당 자원을 사용할 수 있어야 함
사용 중인 자원을 다른 프로세스가 사용하려면 요청한 자원 해제될 때 까지 대기
② 점유와 대기
자원을 최소한 하나 정도 보유, 다른 프로세스에 할당된 자원 얻으려고 대기하는 프로세스 있어야 함
③ 비선점
자원 선점 불가. 즉, 자원은 강제로 빼앗을 수 없고, 자원 점유하고 있는 프로세스 끝나야 해제
④ 순환(환형) 대기

4. 교착 상태 해결 방법 세 가지

① 예방prevention
* 보통 교착 상태 예방 방법
자원의 상호배제 조건 방지
점유와 대기 조건 방지
비선점 조건 방지
순환(환형) 대기 조건 방지

② 회피avoidance
* 교착 상태의 모든 발생 가능성을 미리 제거하는 것이 아닌 교착 상태 발생할 가능성 인정하고(세 가지 필요조건 허용), 교착 상태가 발생하려고 할 때 적절히 회피하는 것
* 교착 상태의 회피 방법
- 프로세스의 시작 중단 : 프로세스의 요구가 교착 상태 발생시킬 수 있다면 프로세스 시작 중단
- 자원 할당 거부(알고리즘Banker's algorithm) : 프로세스가 요청한 자원 할당했을 때 교착 상태 발생할 수 있다면 요청한 자원 할당 않음

③ 탐지detection 와 회복
* 쇼사니Shoshani와 코프만Coffman이 제안
* 은행가 알고리즘에서 사용한 자료구조들과 비슷
- Available : 자원마다 사용 가능한 자원 수를 표시하는 길이 m 인 벡터
- Allocation : 각 프로세스에 현재 할당된 각 형태들의 자원 수 표시하는n ×m 행렬
- Request : 각 프로세스의 현재 요청 표시하는 n×m 행렬. Request[i, j]일 때 프로세스 Pi에 필요한 자원 수가 k개라면, 프로세스 Pi는 자원 Ri의 자원 k개 더 요청

* 회복 방법
프로세스 중단
자원 선점

【학습정리】

1. 필요성
- 하나의 프로세스에서 2개의 스레드가 같이 공유하고 있는 자원(code, data, files)에 동시에 접근하는 것을 막아야 함.
- 하나의 자원(code, data, files)에 하나의 스레드가 접근하도록 제어 필요
- 다수의 스레드가 하나의 자원에 순차적으로 접근할수 있도록 제어 필요

2. 임계영역
- 서로 공유하는 자원들을 순차적으로 접근하기 위한 제어 공간
- 다수의 프로세스 및 스레드가 접근 가능하지만 어느 한 순간에서는 프로세스 및 스레드 하나만 사용 가능

* 스택 영역
- 함수 호출 시 되돌아갈 주소 값 및 지역변수(함수 안에서 선언한)를 저장하기 위한 메모리 공간
- 함수 호출 시 필요한 메모리 영역

* 코드 영역
- 코드영역의 main(), add(), subtract(), Devide()등을 다수의 스레드가 공유하면서 호출 가능

3. 동기화 기법
* 상호배제(MUTEX : MUTual EXclusion)
- pthread_mutex_t 타입의 변수를 뮤텍스라고 표현함
- 한 스레드가 자원을 사용하기 위해 임계영역(Critical Section)으로 들어가면 다른 스레드는 임계영역으로 들어가지 못하도록 하는 기법
- 작동 기본 원리 : 임계영역에 들어갈 때 뮤텍스를 잠그고 들어감, 임계영역에서 빠져 나올 때 뮤텍스를 풀고 나감

* Lock and unlock(Binary Semaphore)
- 세마포 S를 상호배제에 사용, 1 또는 0으로 초기화, P와 V의 연산 교대 실행

4. deadlock(교착 상태)란?
- 둘 이상의 프로세스가 서로 남이 가진 자원을 요구하면서 양쪽 모두 작업 수행을 할 수 없이 대기 상태로 놓이는 상태.

5. 교착 상태 발생의 네 가지 조건
- 상호배제
- 점유와 대기
- 비선점
- 순환(환형) 대기

6. 교착 상태 해결 방법 세 가지
- 예방prevention
- 회피avoidance
- 탐지detection와 회복