OS Assignment#3

세마포어와 철학자 문제 실습

학부 : 소프트웨어 학부

학번 : 20150291

이름 : 하현수

1. 소개

과제 개요

본 과제는 세마포어를 쓰레드 대상으로 구현하고 철학자들의 만찬 (Dining Philosopher)의 동기화 문제를 해결하는 것을 목표로 합니다. Sem.c에 tsem_try_wait()함수를 성공적으로 구현하였으며 이를 바탕으로 dining.c의 코드를 변경하여 동기화 문제를 성공적으로 해결하였습니다.
구현에 대한 테스트는 Vmware Workstation 12 버전을 통한 우분투 16.04 가상머신을 통해 진행하였습니다.
관련 연구는 Deadlock에 대한 간략한 설명과, Dining Philosophers 문제에 대한 Deadlock 기반의 분석, 이를 해결하기 위한 세마포어를 포함한 각종 동기화 기법에 대한 조사와 pthread API함수 조사로 구성되어 있으며, 데드락, 동기화에 대해서 간단히 설명하고, 다양한 동기화 기법과 특성에 대한 조사를 하였습니다. API 함수 부분에서는 본 과제의 핵심 라이브러리인 pthread API함수를 조사하였습니다.

2. 관련 연구

Deadlock(교착상태)

Deadlock(교착상태)이란, 두 개 이상의 작업이 각각 서로의 작업이 끝나기를 기다리며 결과적으로 아무것도 진행되지 않는 상태를 의미한다. 즉, 운영체제나 소프트웨어가 자원 관리를 잘못하여 둘 이상의 프로그램(프로세스) 혹은 쓰레드가 다운되거나 시스템 자체가 멈춰버리는 현상이다.

교착 상태의 조건은 4가지가 존재한다.
1. 상호 배제(Mutual Exclusion) : 상호 배제는 프로그램들이 공유 자원을 동시에 쓸 수 없는 상황을 일컫는다.
2. 순환성 대기(Circular wait) : 대기가 꼬리에 꼬리를 문 상황이다. 연쇄 대기를 해결하려고 해도 결국 자기 자신으로 돌아와버리니 해결할 방법이 없다.
3. 점유 상태로 대기 (Hold and Wait) : 공유 자원을 점유한 상태에서 다른 자원을 기다리는 것이다. 자원을 계속 점유하게 될 경우, 그 자원이 필요한 프로세스는 계속 대기하게 된다. 만약 여기에 순환성 대기 (A->B->C->A)까지 발생한다면 문제는 더욱 악화된다.
4. 선점 불가(No Preemption) : 자원을 어떤 프로세스가 점유 중일때, 다른 프로세스가 그 자원을 뺏을 수 없다는 조건이다.

위 사례를 바탕으로 식사하는 철학자(Dining Philosopher)문제를 분석해보자,
5명의 철학자가 원탁에 앉아있고 각자의 앞에는 스파게티가 있다. 그리고 양 옆에 포크(본 과제에서는 젓가락)이 하나씩 있다. 각각의 철학자는 스파게티를 먹으려면 포크를 2개 사용해야하며, 다른 철학자에게 말을 걸 수 없고 포크를 빼앗을 수 없다는 조건(선점 불가)이 있다. 본 문제에서 포크(젓가락)은 공유 자원을 의미하며, 각 철학자는 프로세스 혹은 쓰레드에 대입될 수 있다.
철학자들은 포크를 공유할 수 없고(상호 배제), 왼쪽에 있는 자신의 포크를 들고 자신의 오른쪽에 앉은 철학자가 포크를 놓을 때 까지 기다린다.(점유 상태로 대기), 모든 철학자들은 자신의 오른쪽 철학자의 포크(젓가락)을 대기한다. (순환성 대기)
이러한 교착 상태를 회피하기 위해서, 본 과제에서는 세머포어를 이용하여 점유 상태로 대기 상태를 해제하였다. 오른쪽 포크를 집을 수 없을 때, 왼쪽 포크를 식탁위에 내려놓는 방법을 사용하였다. 이는 구현 설명에서 자세히 설명한다.

Synchronization(동기화)

동기화란 사건이 동시에 일어나거나, 일정한 간격을 두고 일어날 수 있도록 작업들 사이의 수행 시기를 적절히 조정하는 것을 의미한다.

동기화의 두가지 관점

1. 실행 순서의 동기화

쓰레드 들이 정해진 특정 순서로 실행되어야 할 때 필요하다.

보통 이벤트 오브젝트를 이용한 방법이 많이 사용된다.

2. 메모리 접근의 동기화

대다수의 멀티 쓰레드 기반 시스템이 메모리 접근의 동기화를 구현한다.

Critical Section, Mutex, Semaphore등 다양한 방법이 있다.

다양한 동기화 방식

InterlockedXXX 계열 함수

만약 하나의 전역 변수를 동기화하는 것이 목적이라면, InterlockedXXX 계열의 함수만으로도 동기화가 가능하다. InterlockedXXX계열의 함수들은 한 순간에 하나의 쓰레드만 접근하는 것을 보장해 주는 Atomic Access가 가능하도록 해준다.

CriticalSection

간단한 설명을 통해서 설명을 하자면, 임계 영역을 화장실이라고 가정하자,

이 화장실에 들어가기 위해서는 화장실 열쇠가 필요하다.

화장실에 아무도 없다면 열쇠를 가져가 볼일을 보고, 다시 열쇠를 제자리에 가져다 두어야 한다. 그래야 다음 사람이 다시 열쇠를 가져가 화장실에 갈 수 있기 때문이다. 열쇠를 가진 사람만이 화장실(임계 영역)에 들어갈 수 있다.

이것이 CriticalSection 방식의 핵심 개념이다.

SpinLock

Spinlock은 CriticalSection의 단점을 극복하는데서 착안된 동기화 기법이다.
그 단점이란, 쓰레드가 임계 영역을 획득하지 못하게 되면(Lock을 못잡게 되면) 쓰레드가 블로킹 되는 것이다. 쓰레드 블로킹은 이후 쓰레드에 컨텍스트 스위칭을 불러오게 되어 성능 하락을 유발 시킨다.
이를 극복하기 위해 SpinLock은 락을 점유하지 못할 때 쓰레드가 Back-Off되어 다른 쓰레드에게 넘기는 것이 아니라, Loop를 돌면서 해당 쓰레드를 Busy-Waiting 상태로 만들어 버린다. 이를 통해 쓰레드 스위칭이 발생하지 않게 되어 컨텍스트 스위칭이 발생하지 않게 되는 것이다.

Mutex

CriticalSection에서의 화장실 열쇠가 Mutex방식에서의 Mutex 오브젝트이다.
Mutex는 커널 오브젝트의 이름을 명명할 수 있으며, 이를 통해 프로세스의 중복 실행을 방지하는데 사용한다. Mutex 오브젝트는 NON-Signaled 상태에 있다가, 특정 상황이 되면 Signaled 상태로 바뀌는데, 누군가에 의해 획득이 가능 할 때 Signaled 상태에 놓인다. 즉 일단 화장실 열쇠를 차지하고(락을 걸고), 열쇠를 다시 반환할 때 까지(Mutex를 반환할 때 까지)기다리는 것이다.

Semaphore

보통 세마포어는 Mutex와 상당히 유사하다고들 얘기한다.
세마포어는 그 종류가 몇가지 되는데, 그 중 하나가 Mutex이다.
즉, Mutex는 엄밀히 얘기해 세마포어의 여러 종류 중 하나인 셈이라고도 할 수 있다.
따라서 세마포어 또한 커널 오브젝트 형태이고, 뮤텍스와 처리 방식이 유사함을 알 수 있다. 하지만, Mutex와 다르게 세마포어는 Count기능을 제공한다.
CriticalSection과 Mutex에서는 한번에 한명만 들어갈 수 있는 화장실을 예로 들었지만, 세마포어는 카운터기능을 설명하기 위해 식당(임계 영역)과 테이블로 설명한다.
유명한 식당에 테이블은 딱 10개 뿐인데, 손님이 50명이 대기중이다.

즉 생성된 쓰레드는 50개고, 임계 영역에 동시 접근할 수 있는 쓰레드는 10개가 되는 셈이다. 뮤텍스는 임계 영역에 접근가능한 쓰레드 개수를 조절하는 기능이 없고, 세마포어에는 존재한다. 이것이 바로 Count 기능이다.
세마포어는 카운트가 0인 경우 NON-Signaled 상태가 되고, 1이상인 경우 Signaled 상태가 된다.

Pthread API 함수 연구

Pthread_create 함수

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,

void *(*start_routine)(void *), void *arg);

원형은 위와 같으며, 첫번째 Argument인 thread는 쓰레드가 성공적으로 생성되었을 때 생성된 쓰레드를 식별하기 위한 쓰레드 식별자이며, 두번째 Argument는 쓰레드의 특성을 의미하며, 기본 쓰레드 특성을 사용하고자 할 경우 NULL을 사용한다. 3번째 Argument인 start_routine은 분기시켜서 실행할 쓰레드 함수이며, 4번째 Argument인 arg는 쓰레드 함수의 인자이다.
쓰레드를 생성할 때 사용되며, 성공적으로 생성되면 0을 리턴한다.

Pthread_join 함수

int pthread_join(pthread_t th, void **thread_return);
원형은 위와 같으며, 첫번째 인자인 th는 Join할 쓰레드 식별자이며, 두번째 인자 thread_return은 쓰레드의 리턴 값이다.

Pthread_detach 함수

int pthread_detach(pthread_t th);
함수 원형은 위와 같으며, pthread_create 를 이용해 생성된 쓰레드를 분리시킨다. 이 함수는 식별번호 th인 쓰레드를 detach시키며, 종료된 프로세스의 자원은 pthread_join의 호출 없이도 모두 해제(free)된다.

Pthread_exit함수

void pthread_exit(void *retval);

함수 원형은 위와 같으며, 현재 실행중인 쓰레드를 종료시키고자 할 때 사용한다.

Pthread_self함수

pthread_t pthread_self(void);

함수 원형은 위와 같으며, 해당 함수를 호출하는 현재 쓰레드의 쓰레드 식별자를 리턴한다.

Pthread_mutex_init함수

int pthread_mutex_init(pthread_mutex_t * mutex,

const pthread_mutex_attr *attr);

pthread_mutex_init는 mutex 객체를 초기화 시키기 위해서 사용한다. 첫번째 인자로 주어지는 mutex객체 mutex를 초기화 시키며, 두번째 인자 attr를 이용하여 mutex의 특성을 변경할 수 있다. Mutex의 특성에는 fast, recurisev, error checking의 종류가 있으며, 기본적으로 fast가 사용된다.

Pthread_mutex_destroy함수

int pthread_mutex_destroy(pthread_mutex_t *mutex);

인자로 주어진 mutex 객체를 제거하기 위해서 사용된다. Mutex는 pthread_mutex_init 함수를 이용해서 생성된 Mutex 객체이다.
Pthread_mutex_destroy를 이용해서 제대로 mutex를 삭제하려면 mutex가 반드시 unlock상태여야 한다.

Pthread_mutex_lock함수

int pthread_mutex_lock(pthread_mutex_t *mutex);

원형은 위와 같으며, 임계 영역에 들어가기 위해서 mutex lock을 요청한다. 이미 다른 쓰레드에서 mutex 락을 가지고 있다면 다른 쓰레드에서 mutex lock을 해제할 때 까지 블럭된다. 락을 다 사용하면, pthread_mutex_unlock을 호출하여 mutex lock을 반환한다.

Pthread_mutex_unlock함수

int pthread_mutex_unlock(pthread_mutex_t *mutex);

원형은 위와 같으며, 임계영역에서의 모든 작업을 마치고 mutex lock을 반환하기 위해서 사용된다.

Pthread_cond_init함수

int pthread_cond_init(pthread_cond_t *cond,

const pthread_cond_attr *attr);

원형은 위와 같으며, 조건 변수 cond를 초기화하기 위해서 사용한다. Attr을 이용해서 조건변수의 특성을 변경할 수 있으며, NULL을 줄경우 기본 특성으로 초기화된다.

Pthread_cond_signal함수

int pthread_cond_signal(pthread_cond_t *cond);

원형은 위와 같으며, 조건변수 cond에 시그널을 보낸다. 시그널을 보낼 경우 cond에서 기다리는 쓰레드가 있다면 쓰레드를 깨우게 된다.

Pthread_cond_broadcast함수

int pthread_cond_broadcast(pthread_cond_t *cond);

원형은 위와 같으며, 모든 쓰레드에게 신호를 보내서 깨운다는 점을 제외하고는 cond_signal과 동일하다.

Pthread_cond_wait함수

int pthread_cond_wait(pthread_cond_t cond, pthread_mutex_t *mutex);

원형은 위와 같으며, cond를 통해서 신호가 전달될때까지 블록된다. 만약 신호가 전달되지 않는다면 영원히 블록될 수도 있다. 블록되기 전에 자동으로 mutex락을 반환한다.

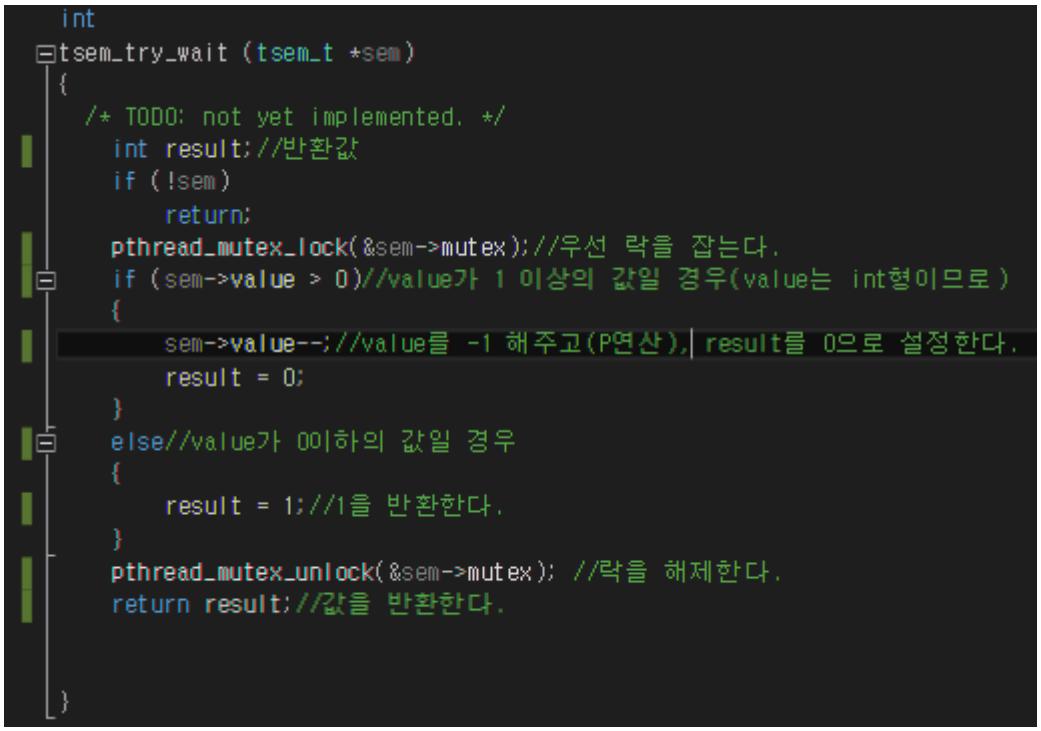
Pthread_cond_destroy

int pthread_cond_destroy(pthread_cond_t *cond);

원형은 위와 같으며, pthread_cond_init을 통해서 생성한 조건변수 cond에 대한 자원을 해제한다. destroy함수를 호출하기 전에 어떤 쓰레드도 cond에서의 시그널을 기다리지 않는 걸 확인해야 한다. 만약 cond 시그널을 기다리는 쓰레드가 존재한다면 이 함수는 실패하고 EBUSY를 리턴한다.

3. 추가 기능 구현 방법

Tsem_try_wait 구현 방법



```
int
tsem_try_wait (tsem_t *sem)
{
  /* TODO: not yet implemented. */
    int result;//반환값
    if (!sem)
        return;
    pthread_mutex_lock(&sem->mutex);//우선 락을 잡는다.
    if (sem->value > 0)//value가 1 이상의 값일 경우(value는 int형이므로)
    {
        sem->value--;//value를 -1 해주고(P연산), result를 0으로 설정한다.
        result = 0;
    }
    else//value가 0이하의 값일 경우
    {
        result = 1;//1을 반환한다.
    }
    pthread_mutex_unlock(&sem->mutex); //락을 해제한다.
    return result;//값을 반환한다.


}
```

Tsem_try_wait 함수를 구현하였다.

우선 wait와 마찬가지로 처음에 pthread_mutex_lock함수를 통해서 lock을 잡아 준 후, sem->value가 1 이상일 경우 value를 1 감소시켜주고 0을 리턴하게,

그리고 sem->value가 0 이하일 경우 바로 1을 리턴하게 설정하였다.

리턴 전에는 반드시 pthread_mutex_unlock을 통해서 락을 해제한다.

```
root@server:~/Assignment3_현수# ./test
thread[0] entered...
thread[1] entered...
thread[2] entered...
thread[3] entered...
thread[4] entered...
thread[5] entered...
thread[6] entered...
thread[7] entered...
thread[8] entered...
thread[9] entered...
thread[0] leaved...
thread[1] leaved...
thread[2] leaved...
thread[3] leaved...
thread[4] leaved...
thread[5] leaved...
thread[6] leaved...
thread[7] leaved...
thread[8] leaved...
thread[9] leaved...
```

test.c 수행 결과, entered와 leaved가 순서대로 출력된다.

즉 명세서의 추가기능 1번 사항을 만족한다.

dining.c 수정 내용

```
//=====================================Edit Code=============================================
do
{
    if (tsem_try_wait(chopstick[i]) == 0)//우선 왼쪽 젓가락을 사용할 수 있는지 살핀다.
    {
        if (tsem_try_wait(chopstick[k]) == 0)//왼쪽 젓가락을 사용할 수 있을 경우 오른쪽 젓가락을 사용할 수 있는지 살핀다.
        {
            update_status(i, 1);//두 젓가락을 모두 얻었으므로, 음식을 먹는다.
            update_status(i, 0);//음식을 먹고 나서 다시 think로 돌아간다.
            tsem_signal(chopstick[i]);//왼쪽 젓가락을 내려 놓는다.
            tsem_signal(chopstick[k]);//오른쪽 젓가락을 내려 놓는다.
        }
        else
        {
            tsem_signal(chopstick[i]);//만약 오른쪽 젓가락을 잡지 못하는 상황일 경우, 왼쪽 젓가락도 내려 놓는다.
        }
    }//=============================Edit COde=============================================
```

Dining.c의 코드는 Deadlock이 발생하는 코드였다.

본 과제에서는 Deadlock을 해결하기 위해서, 점유상태로 대기 상태를 해제하도록 하였다. 즉, 왼쪽 젓가락을 들고 오른쪽 젓가락을 내려놓기를 기다리는 상태의 Deadlock이 일어나지 않도록 수정하였다.

우선 tsem_wait함수를 모두 tsem_try_wait함수로 변경하였다.

그 후, if문을 삽입하여,

if(tsem_try_wait(chopstick[i])==0)라는 문장을 넣었다. 만약 젓가락을 정상적으로 점유할 수 있다면 리턴값이 0일 것이므로, if문 안에 들어왔다면 왼쪽 젓가락을 정상적으로 점유하였음을 의미한다. 이 상태에서 오른쪽 젓가락을 점유할 수 있는지에 대한 여부를 살핀다.

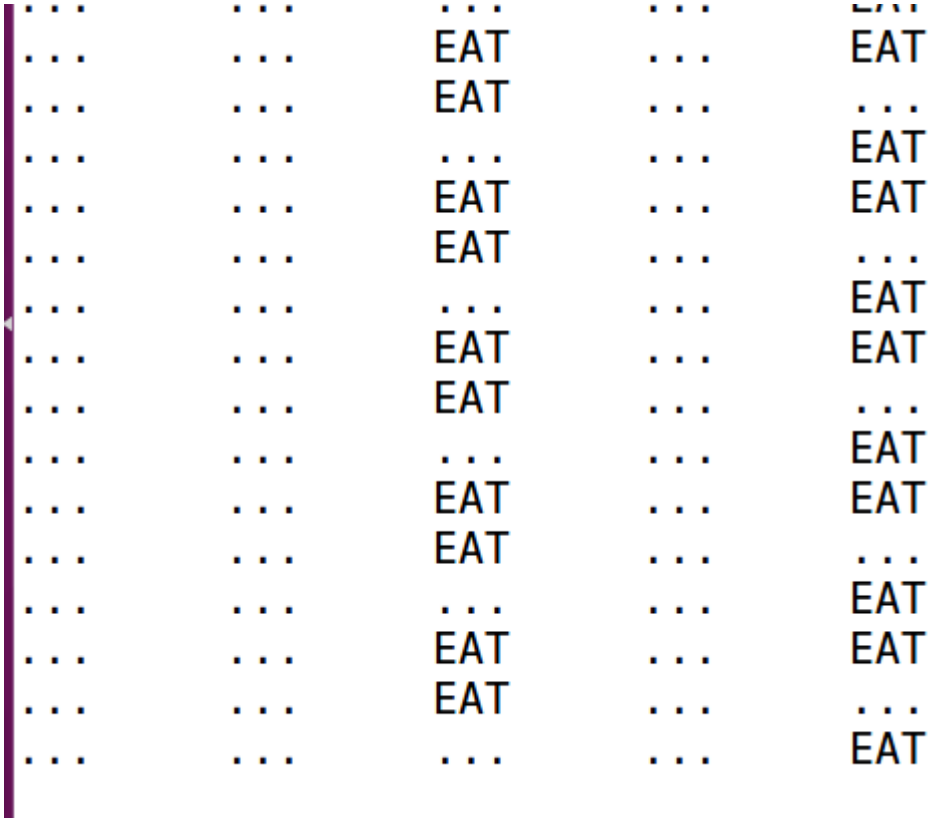
if(tsem_try_wait(chopstick[k])==0) 만약 오른쪽 젓가락도 점유할 수 있다면 리턴값은 0으로 조건문안에 들어올 것이므로, update_status를 통해 음식을 먹은 것을 표시하고, signal을 통해 chopstick[i](왼쪽 젓가락)과 chopstick[k](오른쪽 젓가락)을 반환한다.

만약 tsem_try_wait(chopstick[k])의 반환값이 0이 아니라 1일 경우, 왼쪽 젓가락을

점유하더라도, 오른쪽 젓가락을 차지 할 수 없으므로 signal을 통해서 왼쪽 젓가락을 내려놓도록 한다.
즉 이를 통해 왼쪽 젓가락을 든 상태로 오른쪽 젓가락을 기다리는, 점유 상태로 대기(Hold and Wait) 상태를 해제하였다. 이를 통해 데드락이 해결되었다.

실행 결과

```
...     EAT     ...     ...     EAT
...     EAT     ...     ...     ...
...     ...     ...     ...     EAT
...     EAT     ...     ...     EAT
...     EAT     ...     ...     ...
...     ...     ...     ...     EAT
...     EAT     ...     ...     EAT
...     EAT     ...     ...     ...
...     ...     ...     ...     EAT
...     EAT     ...     ...     EAT
...     EAT     ...     ...     ...
...     ...     ...     ...     EAT
...     EAT     ...     ...     EAT
...     EAT     ...     ...     ...
...     ...     ...     ...     EAT
...     EAT     ...     ...     EAT
...     EAT     ...     ...     ...
EAT     ...     ...     EAT     ...
EAT     ...     ...     ...     ...
EAT     ...     ...     EAT     ...
EAT     ...     ...     ...     ...
EAT     ...     ...     EAT     ...
EAT     ...     ...     ...     ...
EAT     ...     ...     EAT     ...
EAT     ...     ...     ...     ...
EAT     ...     ...     EAT     ...
EAT     ...     ...     ...     ...
EAT     ...     ...     EAT     ...
EAT     ...     ...     ...     ...
EAT     ...     ...     EAT     ...
EAT     ...     ...     ...     ...
```



실행결과, 데드락 문제 없이 프로그램이 실행되는 것을 확인하였으며,
철학자들이 1명 혹은 2명씩 (양 옆이 동시에 먹는 문제 없이) EAT을 잘 진행하고 있는 것을 확인할 수 있었다.

이상 보고서를 마치겠습니다.

감사합니다!