Operating Systems Project #3

과목명.	운영체재론
담 당.	오 희 국 교수님
제출일.	2023년 05월 10일
소프트웨어융합대학	컴퓨터학부
3 학년,	학 번. 2019033936
이 름.	이승섭

HANYANG UNIVERSITY

설계한 두 가지 해법에 대한 설명

우선 공유 버퍼 문제인 bounded_buffer.c 에서는 producer가 버퍼에 아이템을 채우고 consumer가 버퍼에서 아이템을 제거하는 방식으로 이루어지는데 이 과정에서 buffer 배열에 동시에 접근하기 때문에 공유 변수에 대한 문제가 생긴다. 이를 해결하기 위해 producer 함수와 consumer 함수 각각에 lock을 걸어줘야 하는데 stdatomic.h 라이브러리 안에 있는 atomic_compare_exchange_weak() 함수를 사용해준다. 우선 첫번째 인자로 lock 이 들어가는데 원자적 변수로 받아와야되고 lock 값과 비교해주는 expected 도 두번째 인자로 들어가게 된다. 그리고 세번째 인자로 lock의 값을 false 에서 true로 바꿔 주기 위해서 true를 넣어준다. 또한 한개의 스레드씩 critical section 에 들어가야 함으로 반복문에서 이 함수를 계속 돌아가게 하면서 스레드를 대기시킨다. Producer 에서는 버퍼가 가득 차면 버퍼에 아이템을 그만 넣어줘야 하기 때문에 while문을 통과하고 critical section에 들어가기 전 조건문을 통해 counter 값과 버퍼의 사이즈인 BUFSIZE를 비교해 만약 같으면 버퍼가 가득 차 있다는 의미로 lock을 풀어주고 continue를 통해 whie문의 처음으로 돌아가게 한다.

```
expected = false;
while(!atomic_compare_exchange_weak(&lock, &expected, true))
    expected = false;
if(counter == BUFSIZE){
    lock = false;
    continue;
}
```

Consumer에서도 동일한 방식으로 lock을 걸어주는데 consumer에서는 버퍼에 아이템이 한개라도 들어가 있어야지 버퍼에서 아이템을 뺄 수 있기 때문에 while문을 통과하고 critical section 에 들어가기 전 조건문을 통해 counter 의 값이 0이면 버퍼에 아무런 아이템도 없다는 의미이므로 lock을 풀어주고 continue를 통해 while문의 처음으로 돌아가게 해주는 방식으로 동기화를 진행한다.

```
expected = false;
while(!atomic_compare_exchange_weak(&lock, &expected, 1))
    expected = false;
if(counter == 0){
    lock = false;
    continue;
}
```

유한 대기 문제인 bounded_waiting.c 에서는 여러 개의 스레드가 임계구역에 동시에 접근할 경우 값들이 섞여서 나오는 문제가 생기므로 락을 사용하여 한개의 스레드씩 임계구역에 접근하게 만들어주는 방식으로 동기화를 진행한다. atomic_compare_exchange_weak() 함수를 이용하여 반복문 안에 임계구역에 들어가야 하는 스레드인 waiting[i]와 함께 사용해 spinlock을 구현한다. Worker 에서 임계구역으로 들어가기 전 lock을 걸어주고 한개의 스레드가 임계구역에 들어가도록 만든다.

또한 임계구역에서 나오면 다음에 들어갈 스레드를 결정해주기 위해서 반복문과 조건문을 사용해 첫번째 스레드가 끝나면 두번째 스레드가 들어가도록 만들어주고 lock을 풀어준다.

```
while (waiting[i] && !(atomic_compare_exchange_weak(&lock, &expected, 1)))
    expected = 0;
waiting[i] = false;
```

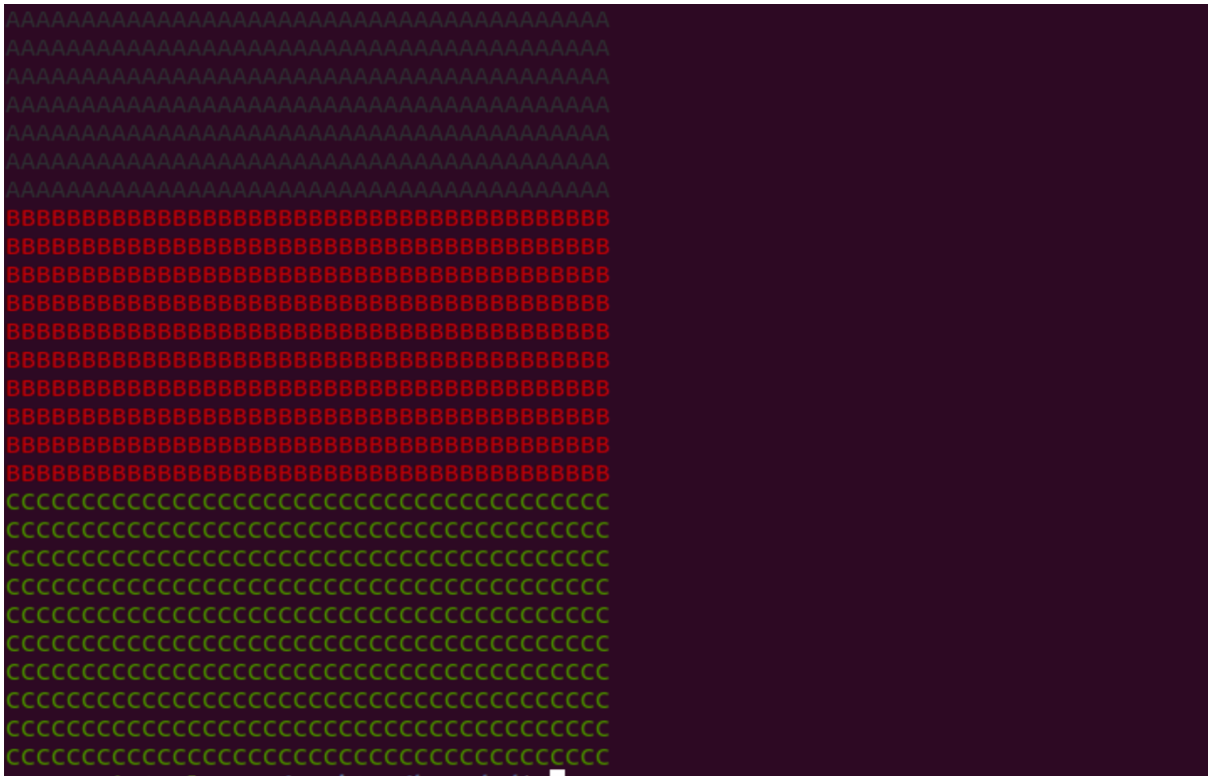
컴파일 과정을 보여주는 화면 캡처

```
os@os-VirtualBox:~/Desktop/bounded$ gcc -o bounded_waiting bounded_waiting.c -p
thread
os@os-VirtualBox:~/Desktop/bounded$ gcc -o bounded_buffer bounded_buffer.c -pth
read
```

실행 결과물의 주요 장면 발췌 및 상세한 설명

```
<P4,25>
<C1,25>
<P4,26>
<C1,26>
<P4,27>
<C1,27>
<P4,28>
<C1,28>
<P4,29>
<C1,29>
<P4,30>
<C2,30>
<C0,31>
<P4,31>
Total 32 items were produced.
Total 32 items were consumed.
```

해당 결과물은 bounded_buffer.c 의 결과물로 producer와 consumer 가 버퍼에 아이템을 넣고 빼면서 실행되고 모든 프로세스가 무한루프를 빠져나와 종료되면 아이템을 얼마나 생성했는지, 아이템을 얼마나 소비했는지 보여준다.



해당 결과물은 bounded_waiting.c의 결과물로 각 알파벳이 40개씩 10줄 총 400개가 연속으로 출력되고 알파벳이 바뀌면 알파벳의 색깔이 바뀌는 결과를 얻을 수 있다.

과제를 수행하면서 경험한 문제점과 느낀점

우선 유한대기 문제 bounded_waiting.c 는 6강 강의동영상과 수업자료를 통해 비교적 쉽게 구현할 수 있었다. 스핀락의 개념인 스레드가 한개씩 임계구역에 들어가도록 락을 걸어주는 방식으로 동기화를 해주며 임계구역에서 스레드가 빠져나오면 그 다음 스레드가 들어가도록 설정해주는 방식을 사용했다. 처음에는 compare_and_swap() 함수를 직접 구현해서 코드를 짰고 실행이 잘 되는 것을 확인 할 수 있었다. 이후에 stdatomic.h 라이브러리 안에 있는 함수를 활용해야 한다는 사실을 알고 공유버퍼 문제를 해결하는 과정에서 스핀락을 어떤 방식으로 걸어줘야 할지 배웠기 때문에 금방 해결할 수 있었다. 하지만 공유버퍼 문제 bounded-buffer.c 는 구현하는데 굉장히 시간이 오

래 걸렸다. 처음에는 mutex_lock 과 semaphore의 개념을 활용해 아래와 같은 방식으로

```
typedef struct semaphore {
    int value;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} semaphore_t;

void sem_init(semaphore_t *s, int value)
{
    s->value = value;
    pthread_mutex_init(&s->mutex, NULL);
    pthread_cond_init(&s->cond, NULL);
}

void sem_wait(semaphore_t *s)
{
    pthread_mutex_lock(&s->mutex);
    while (s->value <= 0)
        pthread_cond_wait(&s->cond, &s->mutex);
    s->value--;
    pthread_mutex_unlock(&s->mutex);
}

void sem_signal(semaphore_t *s)
{
    pthread_mutex_lock(&s->mutex);
    s->value++;
    pthread_cond_signal(&s->cond);
    pthread_mutex_unlock(&s->mutex);
}

semaphore_t m;
semaphore_t empty;
semaphore_t full;

/*
```

wait와 signal 개념을 이용해 producer에서 임계구역에 들어가기 전 sem_wait(&empty), sem_wait(&m) 을 해주고 임계구역에서 나온 후 sem_signal(&m), sem_signal(&full) 해주는 방식으로 동기화를 진행하였고, consumer에서 임계구역에 들어가기 전 sem_wait(&full), sem_wait(&m)을 해주고 임계구역에서 빠져나오면 sem_signal(&m), sem_signal(&empty) 를 해주어 동기화를 해주었고 결과가 잘 나오는 것을 확인할 수 있었다. 하지만 bounded_buffer.c 도 스핀락을 이용해 구현해줘야 한다는 사실을 수업시간에 듣고 수정했는데 스핀락을 어떤 방식으로 걸어주어야 할지 그리고 락을 빠져나올 때 조건문을 사용해 버퍼가 가득 차 있을 때나 비어 있을 때를 어떤 방식으로 구현해야 할지 고민하는 과정에서 stdatomic 라이브러리를 통해 그 해답을 얻을 수 있었다.

atomic_compare_exchange_weak() 함수를 통해 락을 걸어주고 임계구역에 들어가기 전 조건문으로 버퍼가 비어 있는지 가득 차 있는지 확인해주고 임계구역에 나오면 락을 풀어주는 방식으로 구현

하면서 이 과정에서 스핀락에 대해 이해할 수 있었고 락을 어떤 방식으로 걸어줘야 할지 그리고 멀티프로세서 환경에서 동기화가 얼마나 중요한지 알 수 있었다. 또한 변수 expected를 전역변수로 설정해야하는지 지역변수로 설정해야하는지에 대해서도 알 수 있었다. 당연한 말이지만 전역변수로 사용해야하는데 지역변수로서 expected를 사용하면 스핀락이 확실하게 걸리지 않고 동기화가 제대로 되지 않으며 반대의 경우도 마찬가지였다. 동기화를 해주지 않으면 공유변수로 인해 원하는 결과와 다른 값이 나온다는 사실을 이번 과제를 하면서 처음 알게 되었고 스핀락을 이해하면서 동기화 문제를 잘 해결할 수 있었다.