운영체제론 PROJ3 보고서

소프트웨어학부 2019007329 이지현

1. 코드 설명

- bounded_buffer

bounded_buffer는 생산자와 소비자 함수로 나뉘어 중복생산, 중복소비, 미소비 문제를 해결하는 코드이다. 공유변수로 bool expected = false, atomic_bool lock = false를 사용하였다.

producer 함수는 아이템을 생성하여 버퍼에 넣는 역할을 하며 생산자 스레드로 실행되어진다. while(alive)문 안으로 들어가면, 아이템을 생성하기 전 하나의 스레드만 허용하기 위해 atomic_compare_exchange_weak 명령어를 사용하였다. lock이 false이면 critical section에 스레드가 들어가 있지 않아 접근이 가능하다는 뜻이고, true이면 다른 스레드가 critical section에 있으니 접근을 막는다. lock이 false여야 아이템 생성이 가능하므로 expected 값은 false로 지정한다. atomic_compare_exchange_weak 명령어는 lock과 expected 값이 같으면 lock을 true로 바꾸어준 뒤 true를 return한다. cae명령어 앞에 ! 를 사용하여 true가 return 되면 !true => false가 되어 while문을 빠져나가게 만들어준다. 만약 lock과 expected 값이 다르면 expected 값을 lock 값으로 바꾸어 준 뒤 false를 return한다. !false = true이므로 lock이 true일 경우 while문을 빠져나가지 못하고 loop한다. expected값은 항상 false여야 하므로 while문 안에서 expected = false로 값을 유지하도록 하였다. lock이 false여서 while문을 빠져나왔다면 critical section에 진입하게 된다. producer는 버퍼가 꽉 차면 더이상 아이템을 생성할 수 없기 때문에 버퍼에 아이템이 몇 개 남았는지 알려주는 공유변수인 counter를 이용하여 현재 아이템을 생성할 수 있는 상태인지 확인한다. 만약 counter >= BUFSIZE라면 아이템을 생성할 수 없기 때문에 아이템이 소비된 뒤에 생성을 진행할 수 있다. consumer함수에서 소비자 스레드가 아이템을 소비할 수 있도록 lock = false로 락을 풀어주고, 다시 while문을 돌며 락을 획득하는 과정을 거칠 수 있도록 continue를 사용하였다. 만약 counter < BUFSIZE로 아이템을 소비할 수 있는 상태라면 item = next_item++으로 아이템을 생성한 뒤 버퍼의 현재 위치(in)에 item을 저장하고, 버퍼의 다음 위치에 아이템을 저장할 수 있도록 in을 (in + 1) % BUFSIZE 로 한칸 이동한다. 버퍼에 아이템이 하나 추가되었기 때문에 counter 변수를 하나 늘려준다. 그 다음 task_log를 이용해 생산자를 기록하고 중복생산이 아닌지 검증한다. 검증이 끝나면 lock = false로 락을 풀어주고 생산된 아이템을 출력한다.

counuser 함수는 버퍼에서 아이템을 읽고 출력하는 역할을 하며 소비자 스레드로 실행되어진다. while(alive)문 안으로 들어가면, atomic_compare_exchange_weak 명령어를 사용하여 하나의 스레드만 critical section에 접근할 수 있도록 한다. lock은 producer에서 사용하는 lock과 같은 변수를 사용하여 생산과 소비가 동시에 진행되지 않도록 한다. lock이 false이면 expected 와 같기 때문에 lock = true로 락이 걸린 뒤 true가 return된다. 조건문이 !true가 되기 때문에 while문을 빠져나온

다. lock이 true이면 expected와 다르기 때문에 expected = true가 되고 false가 return되어 조건문이 !false가 되기 때문에 while문을 빠져나오지 못한다. expected 값이 true가 되는 경우 false로 바꾸어주기 위해 while문 안에 expected를 false로 바꾸어주는 코드를 삽입하였다. while문을 빠져나오면 버퍼에서 아이템을 소비할 수 있는 상태인지 판단한다. 버퍼에 아이템이 하나도 없다면 소비할 수 없기 때문에 counter <= 0 일 경우 producer 함수에서 아이템을 생성하도록 lock = false로 락을 풀어주고, countinue를 사용하여 아이템 소비를 진행하지 않도록 한다. 만약 counter > 0 이라면 buffer[out]에서 item을 가져오고 소비할 버퍼의 위치를 (out + 1) % BUFSIZE로 한 칸 이동한다. 버퍼에서 아이템 하나를 소비했기 때문에 counter 변수를 하나 감소한다. task_log를 이용해 소비자를 기록하고 미생산 또는 중복소비가 발생했는지 검증한다. 검증이 끝나면 lock = false로 락을 풀어주고 소비한 아이템을 출력한다.

- bounded_waiting

bounded_waiting은 N개의 스레드가 순서에 맞게 critical section에 들어가야 하고, 정해진 차례 안에서 스레드가 락을 얻을 수 있도록 보장한다. 순서에 맞게 critical section에 들어갈 수 있도록 조정하기 위해 turn 공유변수를 사용하였다.

worker 함수의 while(alive)문 안으로 들어가면, critical section에 들어가야 하는 순서인 스레드만 허용하기 위해 while(turn != i || !atomic_compare_exchange_w eak(&lock, &expected, true)) 를 사용했다. turn은 현재 critical section에 들어와야 할 스레드 번호 i를 뜻한다. critical section 공간에 다른 스레드가 없는 경우 lock은 false일 것이고, 스레드 i의 turn이 온다면 바로 while문을 빠져나올 수 있다. 만약 turn이 아닌 다른 스레드가 진입을 원하거나 critical section에 다른 스레드가 진입하여 lock이 true로 잠겨있는 경우, while문을 빠져나올 수 없다. turn과 lock 두가지 조건을 모두 만족하여 while문을 빠져나왔다면, critical section에 진입하여 알파벳 문자 'A'+i를 한 줄에 40개씩 10줄을 출력한다. 출력이 끝나면 다른 스레드가 들어올 수 있게 turn과 lock값을 지정한다. turn은 다음 순서의 스레드가 진입할 수 있도록 turn = (i + 1) % N 로 설정한다. 스레드 i가 할 일을 모두 끝냈기 때문에 lock = false로 락을 풀어준다. turn의 범위는 0부터 7까지이므로 8개의 스레드가 적어도 8번 안에는 자신의 차례가 돌아온다.

2. 컴파일

```
os@os-VirtualBox:~/Documents/theory_assignment/proj3-1$ gcc -o bounded_buffer bo
unded_buffer.c -lpthread
os@os-VirtualBox:~/Documents/theory_assignment/proj3-1$ gcc -o bounded_waiting b
ounded_waiting.c -lpthread
```

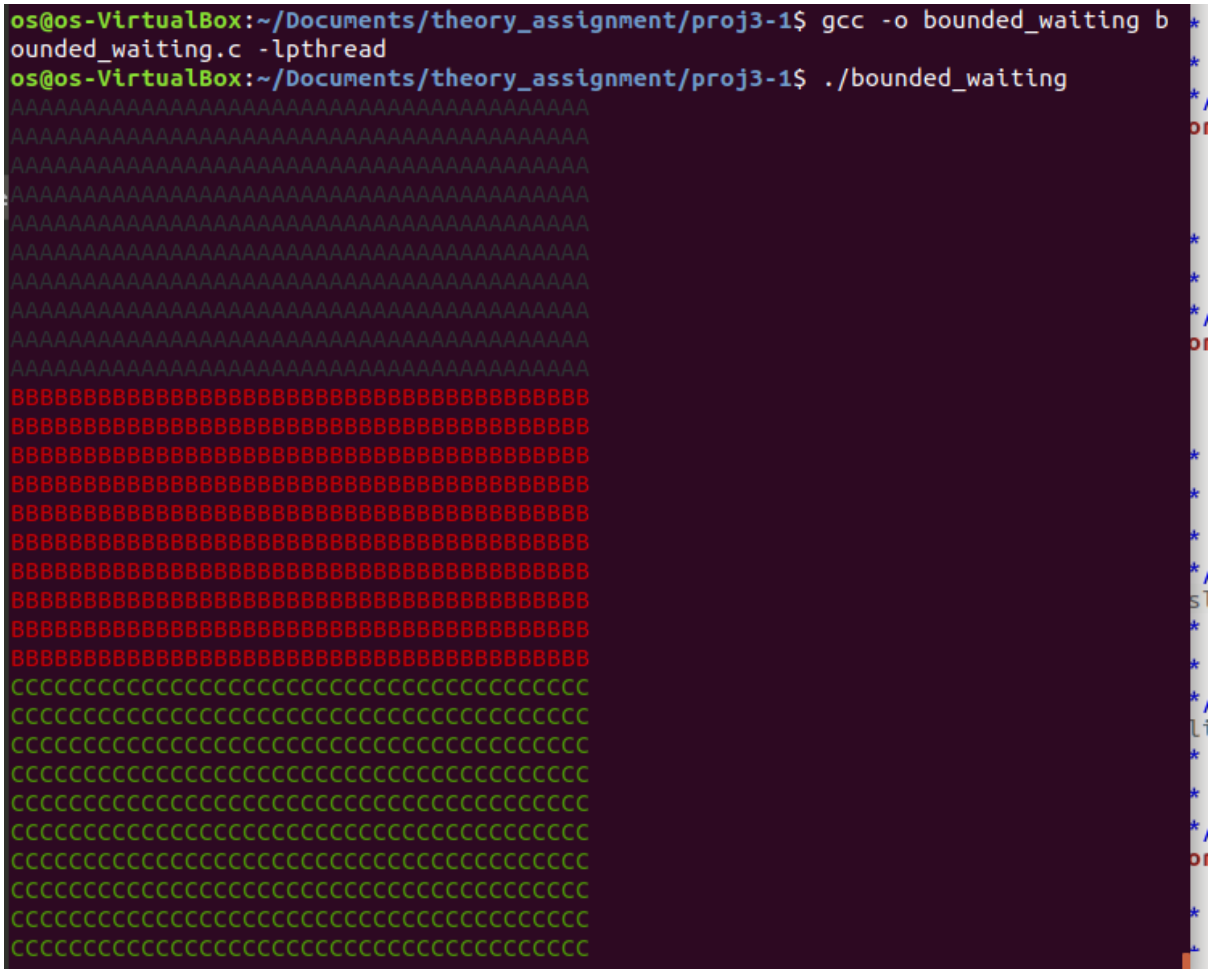
3. 실행 결과

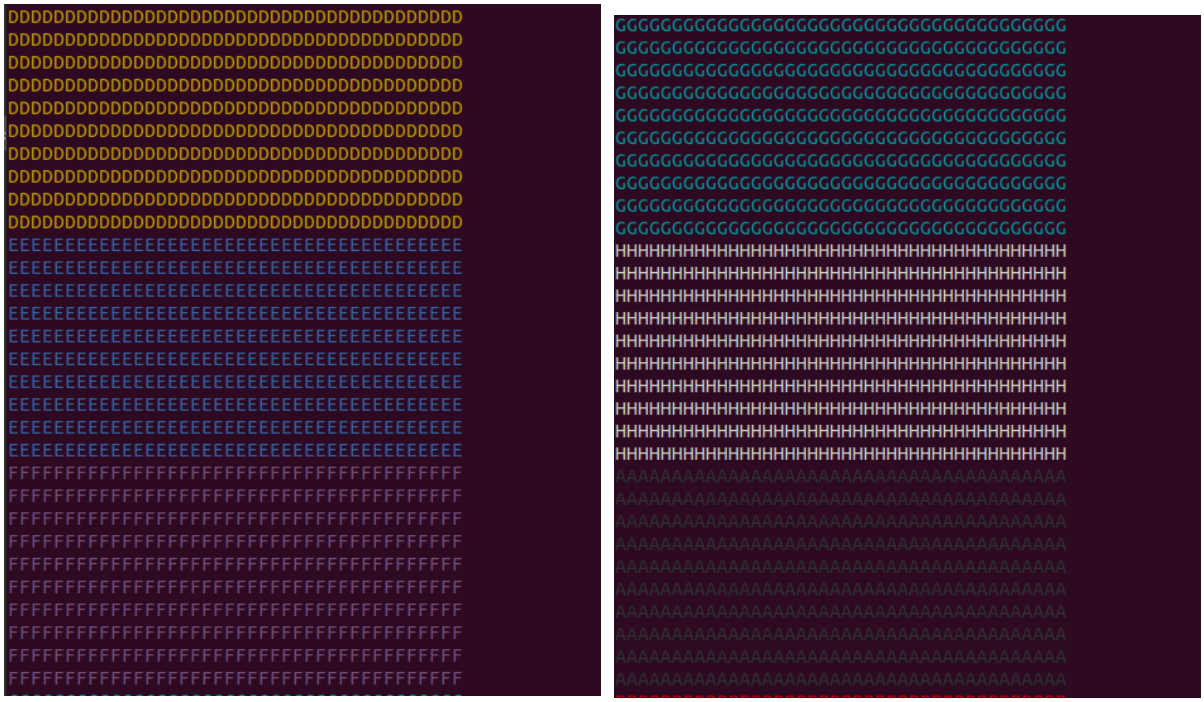
- bounded_buffer

```
os@os-VirtualBox:~/Documents/theory_assignment/proj3-1$ ./bounded_buffer
<P4,0>
<P7,1>
<P7,2>
<C0,1>
<P7,3>
<C3,2>
<P6,5>
<P6,7>
<P6,8>
<P6,9>
<C0,3>
<C1,4>
<C0,6>
<C1,7>
<C0,8>
<C0,9>
<C0,10>
<P4,6>
<C2,0>
<P4,11>
<P4,12>
<P4,13>
<P4,14>
<P4,15>
<C2,13>
<P5,17>
<P7,4>
<C0,12>
<C1,11>
<C1,14>
<P7,18>
<P7,19>
<P7,20>
<P4,16>
<C1,17>
<P6,10>
<C0,16>
<C0,18>
<C0,19>
<P6,22>
<P5,24>
<C1,20>
<C1,21>
<C2,15>
<C1,22>
<C1,24>
<C1,25>
<C1,26>
<C2,23>
<P7,21>
<P7,27>
<P7,28>
<P7,29>
<P7,30>
<P7,31>
<C2,29>
<C3,5>
<P4,23>
<C3,30>
<C3,31>
<C2,32>
<C3,33>
<P5,25>
<P6,26>
<P6,34>
<P6,35>
<C1,27>
<C1,36>
<P7,32>
<P7,37>
<P7,38>
<P7,39>
<P7,40>
<C0,28>
<C0,37>
<C0,38>
<C0,39>
<C1,40>
<C1,41>
<P4,33>
<P4,42>
<P4,43>
<P4,44>
<P4,45>
<P4,46>
<P4,47>
<C2,35>
<C2,44>
<C3,34>
<C3,45>
<P6,36>
<P6,50>
<P7,41>
<C1,42>
<C1,47>
<C1,48>
<P7,51>
<C1,49>
<P6,52>
<C1,51>
<P6,54>
<P6,55>
<P7,53>
<C2,50>
<C1,52>
<C1,53>
<C1,54>
<C1,55>
<C3,46>
<C0,43>
<P4,48>
<P4,57>
<P7,56>
<C1,56>
<C2,57>
<P6,58>
<P5,49>
Total 59 items were produced.
Total 58 items were consumed.
```

bounded_buffer를 실행한 결과, 총 59개의 아이템이 생산되고 58개의 아이템이 소비되었다. 아이템을 출력하는 과정은 critical section 밖에 위치해있기 때문에 소비, 생산 순서와 관계없이 출력되었다. 같은 아이템이 중복으로 생산 및 소비되거나, 생산되지 않은 아이템을 소비하는 문제가 발생하지 않은 것을 볼 수 있다.

- bounded_waiting





총 8개의 스레드를 생성하기 때문에 알파벳이 A부터 H까지 출력되는 것을 볼 수 있다.

알파벳 문자는 출력되는 도중에 다른 알파벳이 출력되지 않고, 하나의 알파벳이 한줄에 40개씩 10줄이 출력된다. 문자 순서는 A, B, C, D, E, F, G, H 순서로 출력된 뒤, 다시 A부터 알파벳 순서대로 실행된다.

4. 문제점과 느낀점

bounded_buffer의 경우 소비자와 생산자 모두 같은 락을 사용하기 때문에 소비하는 동안에는 생산하지 못하고 생산하는 동안에는 소비하지 못한다. counter의 수에 따라 생산과 소비를 동시에 진행할 수 있는 경우가 있는데, 이를 고려하지 않아 스레드의 병행성이 높지 않다. 대신 하나의 락을 사용해서 더욱 간단하고 직관적인 코드를 작성할 수 있었다. 임계 구역과 락의 개수에 따라 어떤 방향으로 코드를 작성할지 선택할 수 있다는 것을 알 수 있었다.

bounded_waiting의 경우 스레드의 순서가 정해졌을때 어떤 방식으로 스레드를 제어해야 할지 알 수 있었던 프로젝트였다. 특정 순서의 스레드가 하나만 들어와야 하므로 turn 변수를 사용하였는데, bounded_waiting은 하나의 함수만 고려하면 되기 때문에 비교적 어렵지 않게 문제를 해결할 수 있었다. 만약 bounded_buffer와 같이 여러개의 함수에서 스레드를 제어해야 했다면 더욱 어려웠을 것이다.

스핀락을 구현하면서 가장 어려웠던 것은 데드락을 해결하는 부분이었다. 어느 부분에서 문제가 발생하는지 찾기 어려웠고, while문의 개수와 while문의 조건이 많을수록 데드락이 많이 발생하는

것 같아 최대한 while문을 적게 사용하여 문제를 해결하려고 노력했다. bounded_buffer와 bounded_waiting 문제를 cae 방식 외에도 수업시간에 배운 다른 방법을 이용해 해결해 보고 싶다.