스핀락

1. bounded_buffer 문제 해결

기존 코드에서는 생산자와 소비자가 각각 버퍼에 접근할 때, *버퍼의 공간 여부에 상관 없이 생산과 소비를 진행*했습니다. 또한, *여러 스레드가 버퍼에 동시에 접근*하면서 원하는 결과를 얻지 못하는 동기화 문제도 있었습니다.

이를 해결하기 위해 **생산자** 입장에서는 *버퍼에 공간이 남아있는지 확인*해야하고, **소비자** 입장에서는 *버퍼에 아이템이 최소 하나는 들어있는지 확인*한 후에 임계구역에 접근해야 합니다. 또한, 하나의 스레드가 임계구역에 진입한 경우에는 *락을 걸어 다른 스레드가 접근하지 못하도록* 막아야 합니다.

이를 구현하기 위해 **counter 변수**와 **atomic_compare_exchange_weak 함수**를 사용했습니다.

producer 함수

뼈대 코드에서 producer는 **버퍼에 빈 공간이 없음에도 임계구역에 접근하는 문제점**이 있습니다. 따라서 아이템을 추가하는 작업 이전에 아래의 코드를 추가하였습니다.

```
68          /*
69           * 일단 락을 얻는다.
70           * 락을 우선 얻은 이유는, 락을 얻은 시점에서의 counter 값으로 접근 가능 여부를 판단해야하기
                 때문이다. 만약 락을 얻기도 전에 counter 값으로 접근 가능 여부를 판단하면, counter 값이
                 임계구역에 있는 스레드에 의해 변경되면서 잘못된 결과를 초래할 가능성이 생기기 때문이다.
71           */
72          while(!atomic_compare_exchange_weak(&lock, &expected, true)) {
73              expected = false;
74          }
75          /*
76           * 만약 아이템을 생성할 자리가 없다면, 생산자는 버퍼에 접근해선 안된다.
77           * 따라서 이 경우에는 락을 해제시킨 후 while문 첫 부분에 돌아가서 락을 다시 대기한다.
78           */
79          if (counter == BUFSIZE) {
80              lock = false;
81              continue;
82          }
```

코드에서는 락을 얻은 상태에서 counter의 값이 BUFSIZE와 동일한지(= 버퍼가 가득 찼는지) 확인합니다. 락을 얻는 과정에서 **atomic_compare_exchange_weak** 함수를 사용하여 락을 얻을 수 있는 상태인지 계속해서 검사하는 것(= 스핀락)을 확인할 수 있습니다.

여기서 락을 먼저 얻는 이유는, 락을 걸지 않은 상태에서 counter 값을 비교하게 되면 임계구역에 있는 다른 스레드가 counter값을 변경하면서 비교 연산이 제대로 진행되지 않을 수 있기 때문입니다.

따라서 락을 얻었지만 버퍼가 꽉 찬 상태라면, 현재 이 스레드는 버퍼에 값을 추가할 기회를 얻을 수 없으며 다음 기회를 잡아야 합니다. 그렇기 때문에 이 경우에는 락을 해제시키고 다시 락을 대기하도록 흐름을 변경했습니다.

생산자가 아이템 생산 작업을 모두 마쳤다면 이제 다른 스레드가 버퍼에 접근할 수 있도록 락을 해제시켜야 합니다. 따라서 아래와 같이 락을 해제하는 코드를 추가하였습니다.

```
93          /*
94           * 생산자를 기록하고 중복생산이 아닌지 검증한다.
95           */
96          if (task_log[item][0] == -1) {
97              task_log[item][0] = i;
98              produced++;
99          }
100         else {
101             printf("<P%d,%d>....ERROR: 아이템 %d 중복생산\n", i, item, item);
102             continue;
103         }
104
105         /*
106          * 아이템 추가 및 생산 기록을 끝내면 락을 해제시킨다.
107          */
108         lock = false;
109
```

위 코드에서 락을 If-else문 이후에 해제한 이유는, `task_log` 배열을 소비자 함수 내부에서 사용하고 있기 때문입니다. 즉, 생산자가 `task_log` 기록까지 완전히 마친 후에 락을 해제시켜야 로그 값이 동기화 될 수 있기 때문입니다.

consumer 함수

소비자 함수도 마찬가지로 **버퍼가 비었음에도 임계구역에 접근하여 아이템을 꺼내오는 문제점**이 있습니다. 따라서 아이템을 꺼내오는 작업 이전에 아래의 코드를 추가하였습니다.

```
129         /*
130          * 락을 얻는다.
131          */
132         while(!atomic_compare_exchange_weak(&lock, &expected, true)) {
133             expected = false;
134         }
135         /*
136          * 만약 버퍼가 비었다면, 소비자는 버퍼에 접근해선 안된다.
137          * 따라서 이 경우에는 락을 해제시킨 후 while문 첫 부분에 돌아가서 락을 다시 대기한다.
138          */
139         if (counter == 0) {
140             lock = false;
141             continue;
142         }
143
144         /*
145          * 버퍼에서 아이템을 꺼내고 관련 변수를 갱신한다.
146          */
```

마찬가지로 소비자도 스핀락 방식으로 락을 먼저 얻은 후 counter 값을 통해 버퍼가 비었는지 판단합니다. 버퍼가 비어있다면 아이템을 소비할 수 없으므로, 락을 해제한 후에 다시 락을 기다릴 수 있도록 작성했습니다.

소비자가 아이템을 소비한 후에는 다른 스레드가 접근할 수 있도록 락을 해제합니다.

```
        * 버퍼에서 아이템을 꺼내고 관련 변수를 갱신한다.
        */
       item = buffer[out];
       out = (out + 1) % BUFSIZE;
       counter--;

       /*
        * 아이템 소비가 끝나면 락을 해제시킨다.
        */
       lock = false;
```

producer 함수와 달리 consumer 함수에서는 아이템을 소비한 후 바로 락을 해제시켰습니다. 소비자는 `task_log`에 있는 값을 읽어오기만 할 뿐, 내부 값은 수정하고 있지 않기 때문에 이로 인해 다른 스레드에 영향을 주지 않기 때문입니다.

2. bounded_waiting 문제 해결

기존 코드에서는 문자를 출력하는 부분이 동기화 되어있지 않아 서로 다른 문자가 섞여서 출력되는 문제가 있었습니다. 따라서 하나의 문자가 출력되는 도중에 다른 문자가 출력되지 않도록 동기화 해주어야 합니다.

또한, 하나의 문자에 대해서 N회 안에 자신의 순서가 돌아오는 것을 보장할 수 있도록 N만큼의 크기를 갖는 waiting 배열을 활용해야 합니다.

따라서 문자를 출력하기 전, 아래와 같이 **waiting 배열**과 **atomic_compare_exchange_weak** 함수를 활용하여 락을 얻는 코드를 추가해 동기화 문제를 해결했습니다.

```
       /*
        * 스레드 i가 critical section 접근을 원함
        */
       waiting[i] = true;
       /*
        * lock의 상태가 해제(false)된 상태인지 확인하기 위한 기댓값
        */
       bool expected = false;

       /*
        * 스레드 i가 대기상태이면서 lock이 해제 상태가 아닌 동안 계속 대기한다.
        * - waiting[i]가 false -> 스레드 i가 더이상 대기하지 않고 cs에 접근 가능함을 의미(다른
            스레드에 의해 대기상태가 해제)
        * - lock이 false -> cs의 접근 권한이 허용되었음을 의미
        * 위 두 조건 중 하나라도 성립하면 임계구역에 접근 가능하다.
        */
       while (waiting[i] && !atomic_compare_exchange_weak(&lock, &expected, true))
           {
           expected = false; // expected가 true로 변경 -> false로 재설정
       }

       waiting[i] = false;

       /*
        * 임계구역: 알파벳 문자를 한 줄에 40개씩 10줄 출력한다.
        */
```

위 코드의 while문을 살펴보면, *현재 스레드가 다른 스레드에 의해 대기 상태가 해제*되거나 *대기상태에서 락이 해제되면* 락을 얻는 것을 볼 수 있습니다. 그리고 락을 얻게 되면 더이상 대기상태에 머무르지 않아도 되므로 waiting 배열의 값을 false로 바꾸는 것을 볼 수 있습니다.

출력 과정 이후에는 아래와 같이 코드를 추가하였습니다.

```
        /*
         * turn을 다음 순서의 스레드 번호로 변경한다.
         */
        int turn = (i + 1) % N;
        /*
         * 자신의 순번으로 다시 넘어오기 전까지 모든 스레드의 대기상태를 검사한다.
         * 만약 대기중인 스레드가 있다면, 해당 스레드에게 접근 권한을 넘겨주기 위함이다.
         */
        while ((turn != i) && !waiting[turn]) {
            turn = (turn + 1) % N;
        }

        /*
         * 자신의 순번으로 되돌아왔다는 것은 현재 대기중인 스레드가 없다는 뜻이다.
         * 따라서 권한을 곧바로 넘겨주지 않고, lock을 그냥 해제한다.
         */
        if (turn == i) lock = false;
        /*
         * 대기중인 스레드가 있다면, lock을 해제하지 않고 해당 스레드에게 cs접근 권한을 넘겨준다.
         * -> waiting을 false로 바꾸면 해당 스레드가 while문을 빠져나와 cs에 접근 가능하게 된다.
         */
        else waiting[turn] = false;
```

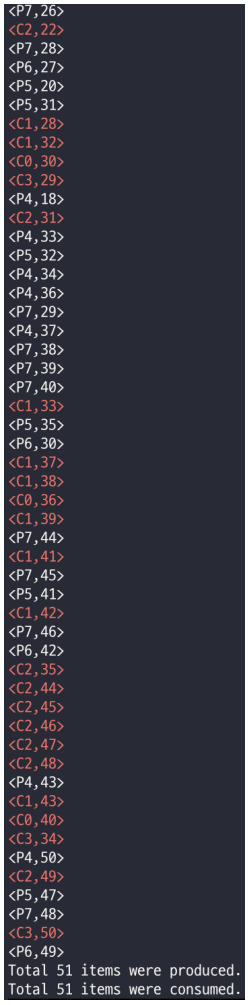
코드에서 대기상태인 스레드가 남아있는 경우에는 락 해제 과정 없이 다음 스레드의 waiting 값을 false로 변경시켜 해당 스레드가 while문을 빠져나올 수 있도록 하였습니다. 즉, *락의 해제/잠금 과정 없이 곧바로 락을 다음 스레드에게 승계할 수 있습*니다.

또한 while문 내부에서 *자신의 차례가 올 때까지 다음 스레드의 대기 상태를 확인*하기 때문에, **N번 안에 임계 구역에 접근하는 것을 보장**할 수 있습니다.

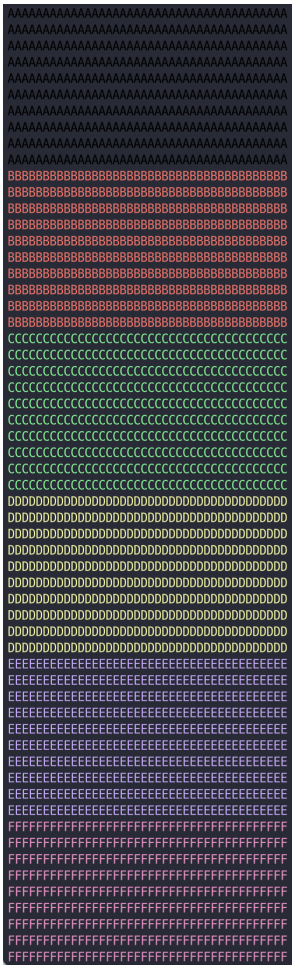
3. 컴파일 과정

```
 parkminseon@bagminseon-ui-MacBookPro  ~/Documents/3-1/OS/assignment/proj3-1
gcc bounded_buffer.c -o bounded_buffer
 parkminseon@bagminseon-ui-MacBookPro  ~/Documents/3-1/OS/assignment/proj3-1
gcc bounded_waiting.c -o bounded_waiting
 parkminseon@bagminseon-ui-MacBookPro  ~/Documents/3-1/OS/assignment/proj3-1
```

4. 실행 결과물 설명



bounded_buffer 실행 결과 (일부)



bounded_waiting 실행 결과 (일부)

bounded_buffer의 경우 기존 코드에서는 동기화가 이뤄지지 않아서 중복생산, 중복소비 등의 문제가 있었지만 코드를 수정한 후에는 문제 없이 잘 동작하는 것을 확인할 수 있습니다. 출력만 보면 소비 이후에 생산되는 것처럼 보이기도 하지만, 이는 출력 부분까지는 동기화하지 않았기 때문에 발생하는 것으로 버퍼 동기화에는 문제가 없음을 알 수 있습니다.

bounded_waiting도 마찬가지로 기존 코드에서는 동기화가 이뤄지지 않아 실행 결과에서 문자가 서로 섞여서 출력되었지만, 코드 수정 이후에는 순서가 보장되며 각 문자가 40줄씩 출력된 후에 이어서 다른 문자가 출력되는 것을 볼 수 있습니다. 특히 각 문자마다 N회 안에는 자신의 순서가 되돌아오는 것을 확인할 수 있습니다.

5. 과제 수행 중 경험한 문제점 및 느낀점

```
59          /*
60           * 버퍼가 비어있지 않거나 lock 상태인 경우 버퍼에 접근할 수 없다.
61           * 따라서 위 조건이 성립하는동안 대기시킨다.
62           * + || 연산의 특징에 의해, 버퍼가 가득 차있는 경우에는 뒤 조건을 검사하지 않는다. 즉, lock
                 상태가 아니더라도 버퍼가 가득 차있으면 lock을 걸 수 없다.
63           */
64          while (counter == BUFSIZE || !atomic_compare_exchange_weak(&lock,
                &expected, true)) {
65              expected = false;
66          }
```

처음 bounded_buffer.c 를 수정할 때는 제출 코드와 달리, counter 값을 통해 버퍼의 상태를 먼저 확인한 후에 atomic_compare_exchange_weak 함수를 통해 락을 얻으려고 했습니다. 하지만 이렇게 작성하여 실행한 결과 오류가 발생하였고, 이와 관련해서 해결 방법을 한참 고민하기도 했습니다. 고민 끝에 counter 변수가 임계구역에 있는 다른 스레드에 의해 변경될 수 있다는 점을 간과했다는 것을 깨닫고, 코드를 올바르게 수정할 수 있었습니다.

저는 이 과정에서 ***공유 변수에 접근하기 위해서는, 다른 스레드가 해당 공유 변수를 수정하지 못하도록 락을 얻은 후 접근해야 한다***는 점을 깊이 깨달을 수 있었습니다. 이론 수업에서 그냥 이해하고 넘어갔던 부분을 직접 코드를 작성하며 겪다보니 이론 수업 때 보다 더 크게 와닿았던 것 같습니다.

그리고 처음에는 **스핀락에서의 락 해제 시점에 대한 중요성**을 크게 깨닫지 못하고 락을 해제시키는 위치를 무작정 각 함수의 마지막 부분에 배치하기도 했습니다. 하지만 스핀락의 경우에는 락을 소유하는 시간을 최소한으로 하는 것이 중요하다는 점을 인지하고, 락 해제 시점을 앞 당겨 코드를 개선할 수 있었습니다.