Operating Systems

Project #3

2019041094 김정민

```
void *worker(void *arg)
{
    int i = *(int *)arg;

    while(alive) {
        bool expected = false;
        while(!atomic_compare_exchange_weak(&waiting[i], &expected, true)) expected = false;

        for(int k = 0; k < 400; ++k){
            printf("%s%c%s", color[i], 'A'+i, color[N]);
            if((k+1) % 40 == 0) printf("\n");
        }

        waiting[i] = true;
        waiting[i+1] = false;

    }
    pthread_exit(NULL);

}
```

Bounded_waiting.c에서 스핀락을 구현할 때는 waiting 배열에 모두 true값을 입력 받고 처음에 0번째 인덱스에 해당하는 곳에 false를 입력하여 스핀락을 풀어준다. 그러고 문자열을 출력해준 뒤 해당 인덱스를 true로 바꿔주고 다음 인덱스를 false로 바꿔주어서 다음 문자열에 해당하는 스레드의 스핀락을 풀어준다.

```
void *consumer(void *arg)
{
    int i = *(int *)arg;
    int item;

    while (alive) {
    bool expected = false;
    while(!atomic_compare_exchange_weak(&lock_c[i%BUFSIZE], &expected, true)) expected = false;
        while(counter == 0);
     /*
        * 버퍼에서 아이템을 꺼내고 관련 변수를 갱신한다.
        */

        item = buffer[out];
        out = (out + 1) % BUFSIZE;
        counter--;
        /*
        * 소비자를 기록하고 미생산 또는 중복소비 아닌지 검증한다.
        */
        if (task_log[item][0] == -1) {
            printf(RED"<C%d,%d>"RESET"....ERROR: 아이템 %d 미생산\n", i, item, item);
            continue;
        }
        else if (task_log[item][1] == -1) {
            task_log[item][1] = i;
            consumed++;
        }
        else {
            printf(RED"<C%d,%d>"RESET"....ERROR: 아이템 %d 중복소비\n", i, item, item);
            continue;
        }
        /*
        * 소비할 아이템을 빨간색으로 출력한다.
        */
        printf(RED"<C%d,%d>"RESET"\n", i, item);
    lock_p[(i+1)%BUFSIZE] = false;
    }
    pthread_exit(NULL);
}
```

소비자 스핀락 응용

스핀락을 통과한 뒤 동작을 완료하고 인덱스 + 1 에 해당하는 생산자의 스핀락을 풀어준다.

```c
void *producer(void *arg)
{
    int i = *(int *)arg;
    int item;

    while (alive) {

    bool expected = false;
        while(!atomic_compare_exchange_weak(&lock_p[i%BUFSIZE], &expected, true)) expected = false;

    while(counter == BUFSIZE);
        /*
        * 새로운 아이템을 생산하여 버퍼에 넣고 관련 변수를 갱신한다.
        */
        item = next_item++;
        buffer[in] = item;
        in = (in + 1) % BUFSIZE;
        counter++;
        /*
        * 생산자를 기록하고 중복생산이 아닌지 검증한다.
        */
        if (task_log[item][0] == -1) {
            task_log[item][0] = i;
            produced++;
        }
        else {
            printf("<P%d,%d>....ERROR: 아이템 %d 중복생산\n", i, item, item);
            continue;
        }
        /*
        * 생산한 아이템을 출력한다.
        */
        printf("<P%d,%d>\n", i, item);
    lock_c[i%BUFSIZE] = false;
    }
    pthread_exit(NULL);
}
```

생산자 스핀락 응용

스핀락을 통과한 뒤 동작을 완료하고 인덱스에 해당하는 소비자의 스핀락을 풀어준다.

Bounded_buffer.c 에서는 소비자와 생산자가 번갈아가면서 작동할 수 있도록 구현하기 위해 lock_p와 lock_c 배열을 만들어서 각각의 스핀락을 관리해준다. 소비자가 작동이 끝나면 생산자의 락 배열을 false 값으로 바꿔주고, 생산자의 작동이 끝나면 소비자의 락 배열을 false 값으로 바꿔줘서 소비자와 생산자가 번갈아가면서 나올 수 있게 동기화해준다.

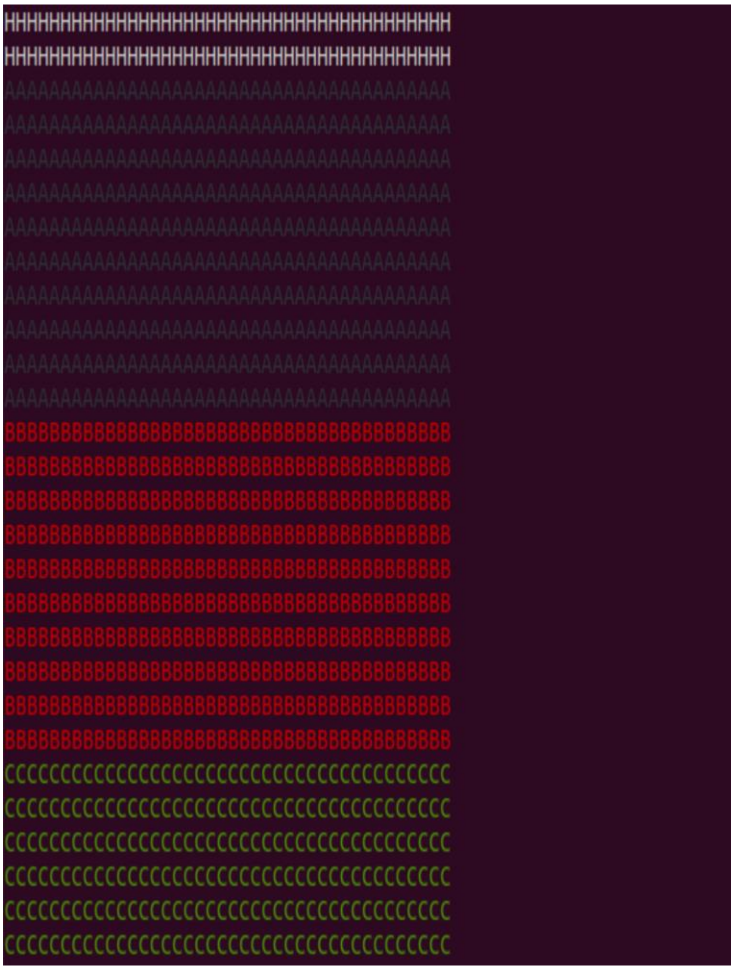
```
jeongmin@jeongmin-VirtualBox:~/Desktop/proj3-1$ gcc bounded_buffer.c -o test1 -
lpthread
jeongmin@jeongmin-VirtualBox:~/Desktop/proj3-1$ gcc bounded_waiting.c -o test2
-lpthread
```

컴파일 과정

```
<C1,70>
<P4,74>
<C0,73>
<P6,75>
<C2,74>
<P5,76>
<C1,75>
<P7,77>
<C3,76>
<P6,78>
<C2,77>
<P4,79>
<C0,78>
<P7,80>
<C3,79>
<P4,81>
<C0,80>
<P5,82>
<C1,81>
<P6,83>
<C2,82>
<P7,84>
<C3,83>
<P4,85>
Total 86 items were produced.
Total 84 items were consumed.
```

생산자와 소비자가 번갈아가면서 나오는 모습

총 85개 아이템을 생산하고 84개 아이템을 소비했다.



A~H까지 순서대로 각자 다른 색깔로 40개씩 10줄 나오는 모습

알파벳들이 서로 섞이지 않고 출력된다.

어려웠던 점, 느낀 점

과제를 수행하면서 스핀락의 개념에 대해 깊이 이해할 수 있었습니다. 스핀락을 통해 여러 개의 스레드가 공유되는 자원에 동시에 접근할 때 발생할 수 있는 경합 조건과 같은 문제를 해결할 수 있습니다. 또한, 소비자 생산자 문제를 다루면서 공유 버퍼에 동시에 접근할 때 동기화를 해주어 문제를 해결할 수 있었습니다. 이를 위해 락을 사용하여 상호배타를 보장하였으며, 이로 인해 임계구역에 대한 접근을 조절하여 안정적인 프로그램을 구현할 수 있다는 사실을 깨달았고, 스핀락은 정말 중요한 개념이라고 생각하게 되었습니다.