OS 실습 5

semaphore

```
assignment > assignment_1_2 > C message_buffer_semaphore.c
120 vint produce(MessageBuffer **buffer, int sender_id, int data, int account_id) {
          (*buffer)->messages[account_id].sender_id = sender_id;
          (*buffer)->messages[account_id].data += data;
          (*buffer)->account_id = account_id;
          (*buffer)->is_empty = 0;
          s_quit();
          printf("produce message\n");
          return 0;
138 ∨ int consume(MessageBuffer **buffer, Message **message) {
          /* TODO 4 : consume message
          s_wait();
          if((*buffer)->is_empty == 1){
            s_quit();
             return -1;
          *message = &((*buffer)->messages[(*buffer)->account_id]);
          (*buffer)->is_empty = 1;
          s_quit();
          return 0;
```

- **line 124**: produce 함수에서, 공유 메모리에 해당하는 *buffer에 접근하기 전에 s_wait() 을 써서 세마포어로 락을 걸어주어 원자성을 보장한다.
- **line 129**: produce 함수에서, 공유 메모리 작업이 끝난 뒤 s_quit() 을 써서 교착이 일어나지 않도록 락을 release한다.
- line 142: consume 함수에서, 공유 메모리에 접근하기 전에 s_wait() 을 써서 원자성을 보장한다.
- **line 144**: 만약 메모리가 버퍼가 비어있어 consume할 게 없을 경우, 작업을 끝내고 리턴하게 되므로 s_quit()을 써서 교착이 일어나지 않도록 release한다.
- line 149: 마찬가지로 작업이 끝난 뒤 s_quit() 을 써서 락을 release 한다.
- → 위 코드들로 공유 메모리에 동시 접근 시 발생하는 race condition 문제를 해결할 수 있다.

OS 실습 5 1

assignment_1_2에 대한 test.sh

```
#!/bin/bash

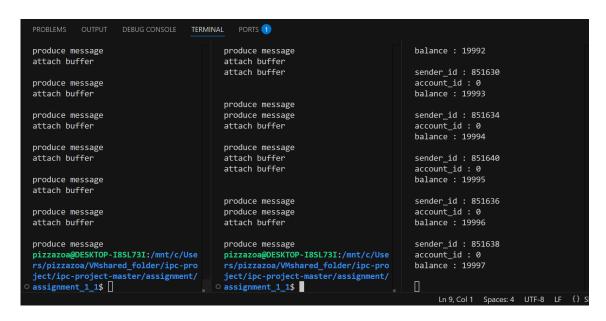
for j in {1..100}; do
    for i in {1..100}; do
        ./producer 0 1 &
        done

done

wait
```

- 터미널 3개를 띄워서, 하나는 ./consumer를 실행하여 consumer를 켠다.
- 나머지 2개의 터미널에서 ./test.sh를 동시에 실행한다. 동시에 producer가 실행될 때 발생할 수 있는 캐시 비일관성 문제에 대해 1_1과 1_2에서의 결과를 비교한다.

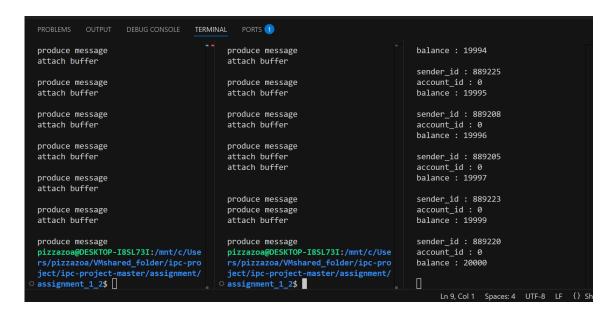
1_1에서의 결과 (세마포어 없을 때)



• 예상 기대값 20000과 불일치하는 문제가 발생했다.

1_2에서의 결과 (세마포어 있을 때)

OS 실습 5



• 세마포어로 임계 구역을 보호한 덕에 공유 메모리에 대한 연산이 원자적으로 수행되어 예상 기대값 20000과 일치하는 모습을 확인할 수 있다. (중간에 값(19998)이 비는 문제는 producer의 연산이 consumer보다 빠르게 두 번 수행되었기 때문으로 해석할 수 있다)

OS 실습 5