# (운영체제 Homework #4)

## < Reader-Writer Locks: no Starvation >

소프트웨어학과 20162820 김영민

★ Read-Writer Lock이 개선되었을 때의 Thread Trace 구현 ★

◎ 쓰레드별 상태 추적 출력하기

- 파라미터 입력 방법 구현 (예: command line parameter 처리)
- 다양한 시나리오 생성 및 설명
- 각 시나리오에 대한 실제 실행 결과

◎ Reader-Writer Lock 개선

- Starvation 현상이 제거되도록 개선(Writer가 기다리는데 Reader가 계속 도착하면 발생

◎ 개선된 Reader-Writer Lock을 사용해서 사용한 여러 시나리오를 실행하고 결과를 비교

## < 쓰레드별 상태 추적 출력하기 >

### Q1) 파라미터 입력 방법 구현 (예: command line parameter 처리)

- 파라미터 입력 방법은 Command line argument으로 구현 했다.

→ /reader-writer -n 6 -a 0:0:5,0:1:8,1:3:4,0:5:7,1:6:2,0:7:4

먼저 int num_workers =6;을 넣어주어 총 6개의 job이 실행할 수 있는 환경을 만들어 주었다. 첫 번째 JoB(0:0:5)의 경우로 예를 들자면 각 job의 read/write 타입을 첫 번째 파라미터로 설정해 주었다. 0이기 때문에 read로 설정 돼있다. 두 번째 파라미터는 언제 실행하는지 도착시간 arrival_delay를 설정해주었다. 첫 번째 JoB은 0 이므로 0초에 시작 됐다. 즉 제일 첫 번째 running을 시켜주었다. 그리고 마지막 파라미터 5의 경우는 running time 즉 5를 넣어주어 5time만큼 running 된다고 설정해주었다.

```
int num_t[6] = {0,0,1,0,1,0}; // Job -> read/write type
int num_w[6] = {0,1,3,5,6,7}; // Job -> arrival_delay
int num_c[6] = {5,5,4,3,2,4}; // job -> running_time
 // int num_workers =6; 이므로 Job이 총 6개이기 때문에 각 6개씩 값 넣어준다.
```

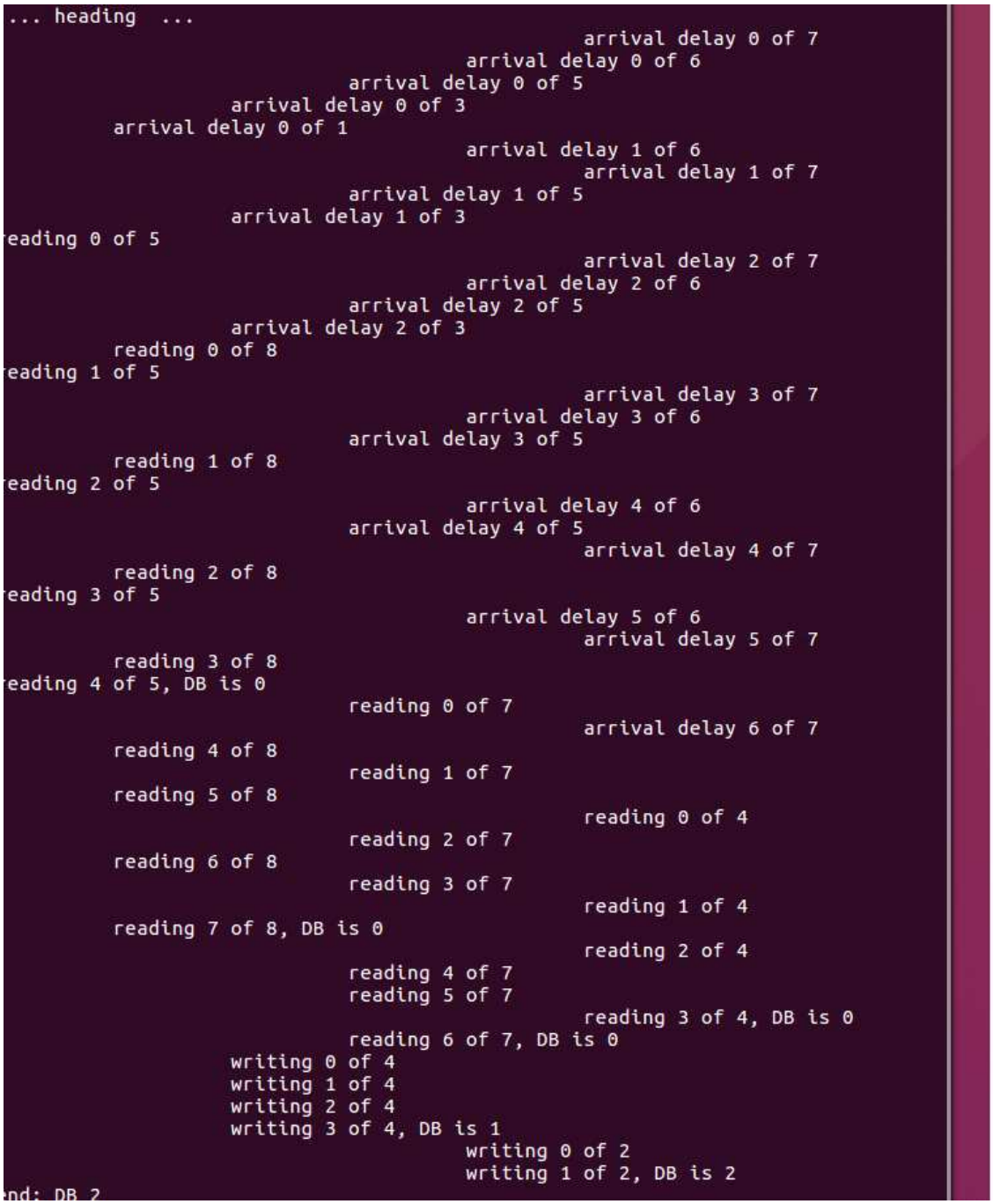
배열 선언을 해준뒤 for문으로 그 배열의 값을 집어 넣어 a[i] 구조체에 Job의 type, arrival_day, running_time을 설정해주었다.

```
for (i = 0; i < num_workers; i++) {
    a[i].thread_id = i;
    a[i].job_type = num_t[i];
    a[i].arrival_delay = num_w[i];
    a[i].running_time = num_c[i];
}
```

# Q2) 다양한 시나리오 생성 및 설명

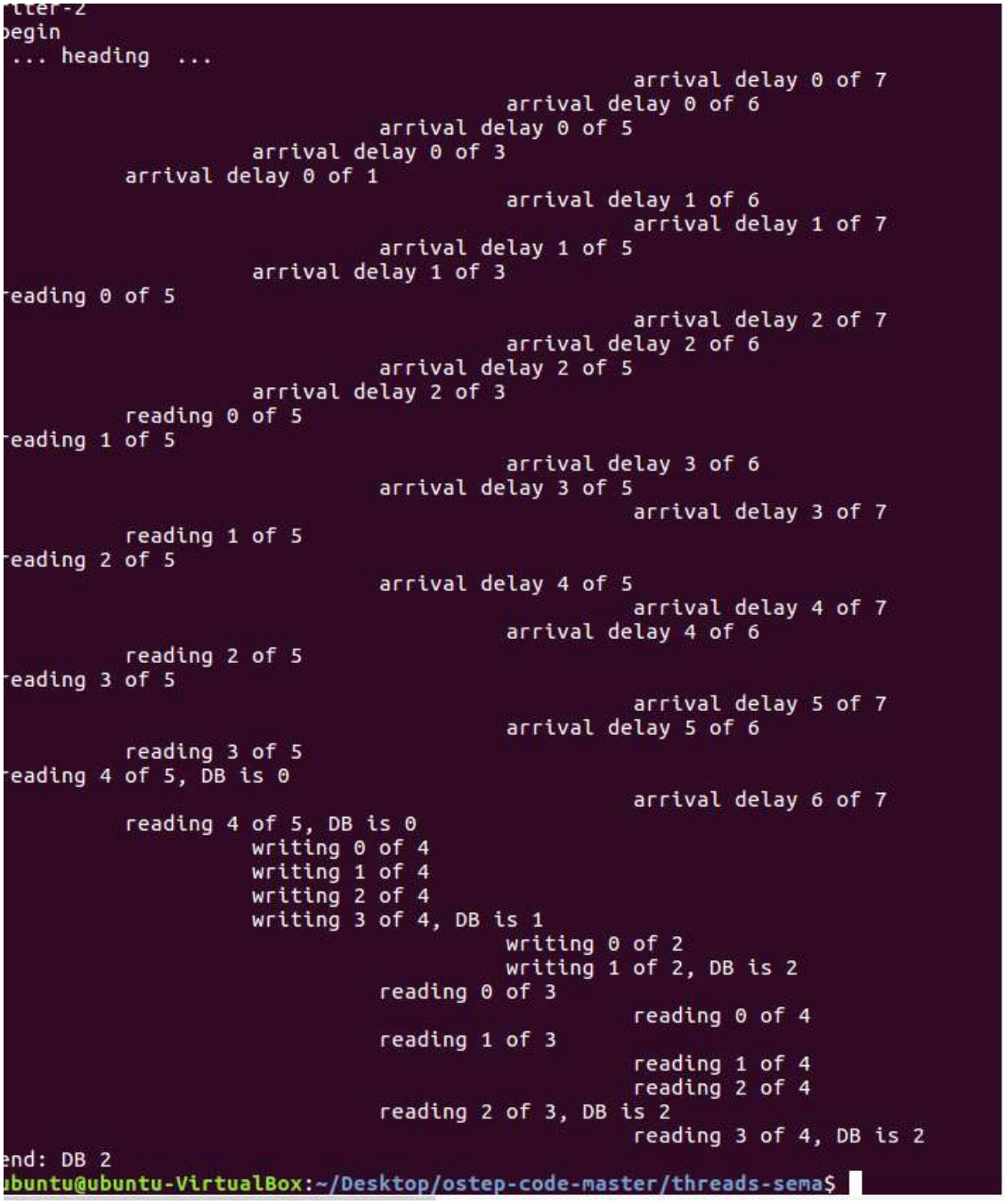
(Starvation)을 해결하지 못한 시나리오.

시나리오 1) `/reader-writer -n 6 -a 0:0:5,0:1:8,1:3:4,0:5:7,1:6:2,0:7:4`




starvation이 해결되지 못했기 때문에 writing이 맨 마지막에 실행이 된다. 이 시나리오는 처음에 제시해준 버전의 시나리오 이다. 두 번째 시나리오는 러닝타임, read/write Job순서를 바꿔주어 실행해 보았다.

시나리오 2)  /reader-writer  -n 6 -a 0:0:3, 0:1:5. 1:2:4, 1,4:7, 0:5:3, 0:6:4




마찬가지로 starvation이 해결하지 못한 상황으로 read Job들이 끝나야만 비로소 wrighting이 running이 되는상황을 연출해준다.

# < Reader-Writer Lock 개선 >

먼저 기존코드에 버그가 있는 부분부터 수정을 해주었다, 첫 번째는 void void *worker(void *arg)의 순서이다. void void *worker(void *arg) {함수는 void *writer(void *arg)와 void *reader(void *arg) 선언 이후에 나와야 하기 때문에 그 이후로 위치를 옮겨 주었다.
두 번째는  space(args->thread_id); printf("writing %d of %d, DB is %d\n", i, args->running_time, Value)에서 Value값을 DB값으로 고쳐주었다 그렇게 하여 버그를 고쳤다.

**<starvation 해결>**

먼저 starvation을 해결하기위해 기존 코드에서 int AR 뿐만아니라 AW,WR,WW를 rwlock_t 구조체에 추가해주었다. 그리고 semaphore를 이용하여 no starvation하게 코드를 짜주었다. sem_t에 write가 대기할수 있는 큐인 okTowrite와 read가 대기할 수 있는 큐인 oKToread를 구현해주어 rwlock_t 구조체를 완성시켰다 이런 구조체를 이용하여 세마포어형태의 nostarvation 락을 만들어 줄 수 있다.
먼저 첫 번째 로 init 형태를 구현해주었다.

```
void rwlock_init(rwlock_t *rw) {
    rw->AR = 0; rw->AW = 0; rw->WR = 0; rw->WW=0;
    Sem_init(&rw->mutex, 1);
    Sem_init(&rw->okToRead, 0);
    Sem_init(&rw->okToWrite, 0);
}
```

AR,AW,WR,WW의 초기값은 0 이고 mutex의 세마포어 값은 1로 설정해주었다. 그리고 okToread와 okTowrite의 값은 0으로 설정해줌으로써 각각 wait가 걸릴시 0에서 -1로 어느 job이 대기하고 있다는 것을 알 게 해주었다.

```
void rwlock_acquire_readlock(rwlock_t *rw) {
    Sem_wait(&rw->mutex);
    while((rw->AW+rw->WW)>0){
        rw->WR++;
        Sem_post(&rw->mutex);
        Sem_wait(&rw->okToRead);
        rw->WR--;
    }
    rw->AR++;
    Sem_post(&rw->mutex);}
```

두 번째 는 reader가 실행 됐을 때 의 상황인데 read job이 실행된다면 일단 뮤텍스락을 걸어주고 AR을 ++한다 하지만 write가 실행중일때는 okToread에 큐를 대기시켜준다. 이때 Sem_wait(&rw->okToRead); 구현전에 무조건 Sem_post(&rw->mutex);를 해줘야 데드락이 발생하지 않는다 이것이 바로 semaphore의 특징이다.

```
void rwlock_release_readlock(rwlock_t *rw) {
    Sem_wait(&rw->mutex);
    rw->AR--;
    if (rw->AR == 0 && (rw->WW>0))
        Sem_post(&rw->okToWrite);
    Sem_post(&rw->mutex);
}
```

여기서는 read를 깨우는 과정이다. rw->AR에 실행중인 reader가 없거나 okTowrite에 있는 큐가 >0개 이상일 경우 그것을 깨어주어 실행시키게 해준다 이 함수 역시 mutex락과 언락이 필수적이다.

```
void rwlock_acquire_writelock(rwlock_t *rw) {
    Sem_wait(&rw->mutex);
    while((rw->AW + rw->AR)>0){
        rw->WW++;
        Sem_post(&rw->mutex);
        Sem_wait(&rw->okToWrite);
        rw->WW--;
    }
    rw->AW++;
    Sem_post(&rw->mutex);
```

이함수도 앞서말한 acquire_readlock과 마찬가지로 Sem_wait(&rw->okToWrite); 실행전에 Sem_post(&rw->mutex);는 필수적으로 같이 와줘야한다. 그리고 writer가 okTowrite에 들어가 대기하는 조건은 rw_>AW가 실행중이고 rw->AR이 실행중이면 큐에 들어가 대기를 해준다.

```
void rwlock_release_writelock(rwlock_t *rw) {
    Sem_wait(&rw->mutex);
    rw->AW--;
    if(rw->WW > 0){
        Sem_post(&rw->okToWrite);
    }
    else if(rw->WR > 0){
        sem_post(&rw->okToRead);
        sem_post(&rw->okToRead);
    }
    Sem_post(&rw->mutex);
}
```

마지막으로 void rwlock_release_writelock(rwlock_t *rw) 함수이다. 이함수는 앞서말한 void rwlock_release_readlock(rwlock_t *rw) { 와 마찬가지로 세마포어로 구성된 writer job이 끝날 때 실행되는 함수이다. 이때  else if(rw->WR > 0){

sem_post(&rw->okToRead);

sem_post(&rw->okToRead);

가 실행되는데 이건 컨디션락 일때는 broadcast로 모든 대기 job을 깨웠으나 semaphore일때는 각각의 job을 깨우기 위해 post를 두 번해주었다.

<개선된 Reader-Writer Lock을 사용해서 사용한 여러 시나리오를 실행하고 결과를 비교>

(starvation을 해결한 시나리오)

시나리오 1) /reader-writer -n 6 -a 0:0:5,0:1:8,1:3:4,0:5:7,1:6:2,0:7:4

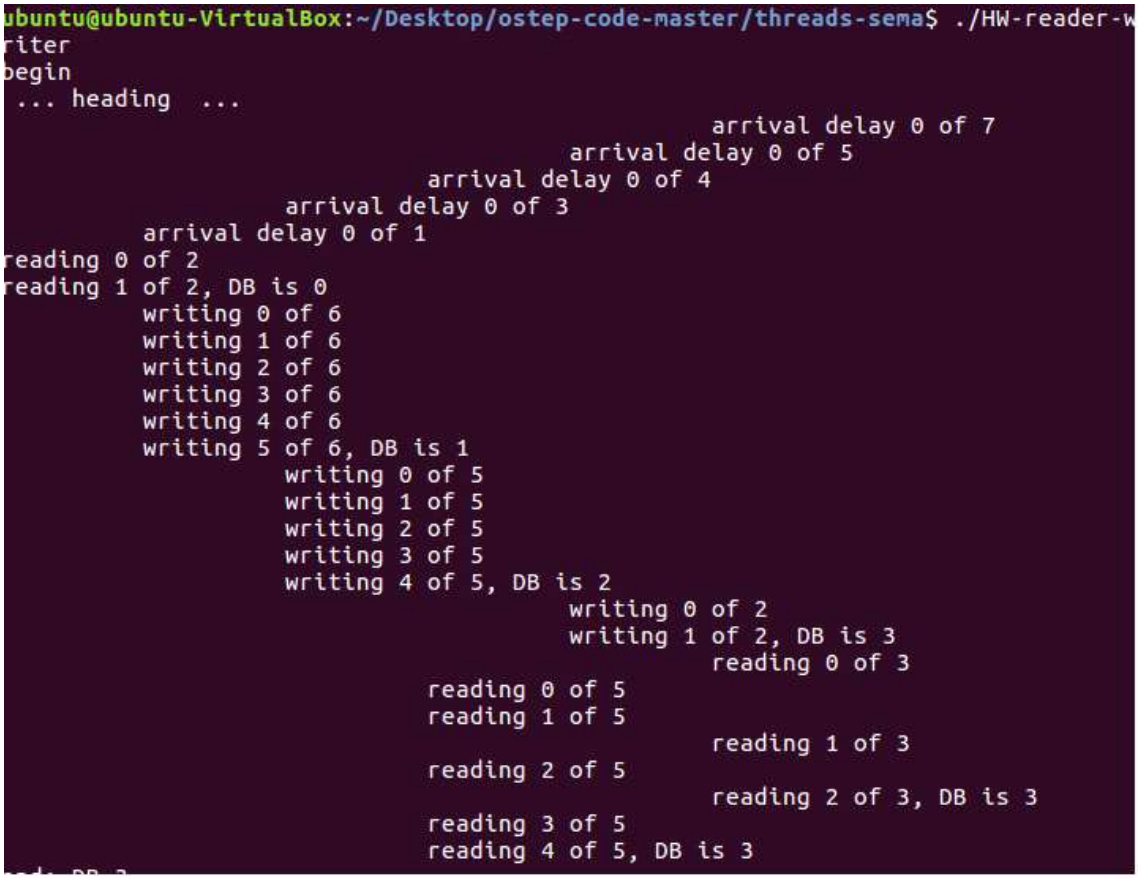
```
riter
begin
 ... heading  ...
                                                  arrival delay 0 of 7
                                        arrival delay 0 of 6
                              arrival delay 0 of 5
                    arrival delay 0 of 3
          arrival delay 0 of 1
                                        arrival delay 1 of 6
                                                  arrival delay 1 of 7
                              arrival delay 1 of 5
                    arrival delay 1 of 3
reading 0 of 5
                                                  arrival delay 2 of 7
                              arrival delay 2 of 5
                                        arrival delay 2 of 6
                    arrival delay 2 of 3
          reading 0 of 5
reading 1 of 5
                                                  arrival delay 3 of 7
                              arrival delay 3 of 5
                                        arrival delay 3 of 6
          reading 1 of 5
reading 2 of 5
                                        arrival delay 4 of 6
                              arrival delay 4 of 5
                                                  arrival delay 4 of 7
          reading 2 of 5
reading 3 of 5
                                                  arrival delay 5 of 7
                                        arrival delay 5 of 6
          reading 3 of 5
reading 4 of 5, DB is 0
                                                  arrival delay 6 of 7
          reading 4 of 5, DB is 0
                    writing 0 of 4
                    writing 1 of 4
                    writing 2 of 4
                    writing 3 of 4, DB is 1
                                        writing 0 of 2
                                        writing 1 of 2, DB is 2
                              reading 0 of 3
                                                  reading 0 of 4
                              reading 1 of 3
                                                  reading 1 of 4
                                                  reading 2 of 4
                              reading 2 of 3, DB is 2
                                                  reading 3 of 4, DB is 2
end: DB 2
```

-> 개선된 시나리오를 통해 제대로 동작하는 reea/write lock이 형성됐음을 알 수 있다. 전에 개선전의 시나리오를 보면 writing을하는 3번째 job이 마지막에 위치해있었으나 개선하고 나서는 중간에 실행을하는 것을 보여준다.

시나리오 2)  /reader-writer  -n 6 -a 0:0:3, 0:1:5. 1:2:4, 1,4:7, 0:5:3, 0:6:4


-> 이것또한 깔끔하게 writing과 reading의 과정이 나온다. 앞서 개선전의 시나리오는 들쭉날쭉하게 더럽게(?) 구현됐으나 개선된버전은 깔끔하게 표현이된다.

시나리오 3) /reader-writer -n 6 -a 0:0:2, 1:1:6, 1:3:5, 0:4,5, 1:5,2, 0:7:3




마지막시나리오는 개선전 과정에는 포함하지 않았으나 write를 3개를 주어 상호배재하게 표현을 하였다 arrival delay 타임이 표현되는게 조금 덜 깔끔한 코드라 생각하여 for문의 조건에 I가 1이이상이면 더 이상 표현되지 않게끔 고쳐주었다.