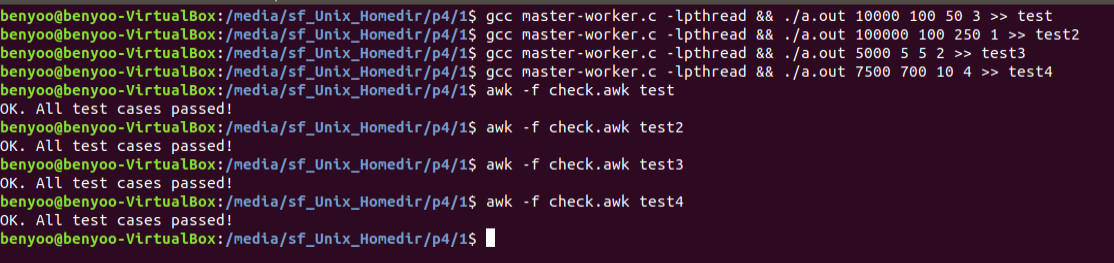
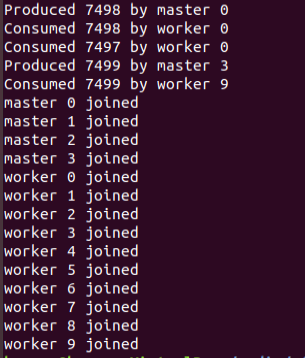
* **실행결과**

1. Master-Worker Thread Pool 구현

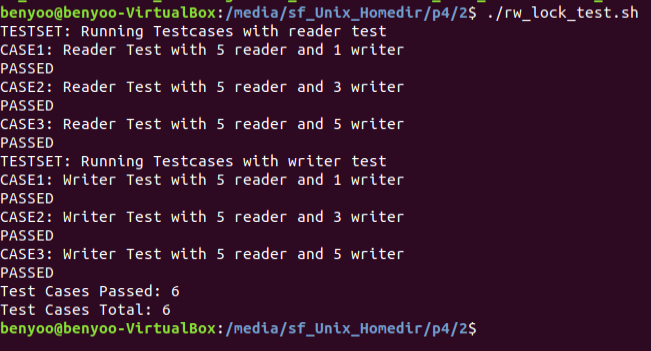


구현한 master-worker.c에 다양한 인자를 주어 파일로 출력을 한 다음 주어진 check.awk를 통해 출력된 파일들을 검증하였다.



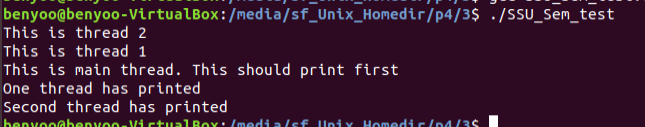
그 중 test4 파일의 내용을 보면, 0~7499까지 총 7500개의 파일을 produce하고 consume 하는 것을 확인할 수 있고 4개의 master와 9개의 worker가 성공적으로 조인한 것을 확인할 수 있다.

1. Reader-Writer Locks 쓰레드 프로그램 구현

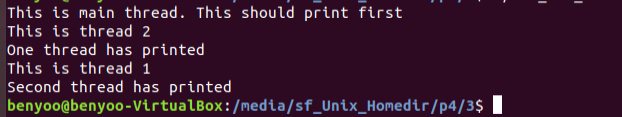


구현한 프로그램을 rw\_lock.c, rw\_lock.h는 공통적으로 포함하고, rw\_lock-r-test.c와 rw\_lock-w-test.c 등의 프로그램으로 차이점을 두어 최종 프로그램을 읽기 쓰레드를 기본 설정으로 컴파일할지 아니면 쓰기 쓰레드의 기본 설정으로 컴파일할지를 결정하게 되어있다. rw\_lock\_test.sh에는 두가지 모드로 프로그램을 컴파일한다음 각 모드에 따라 3종류의 인자를 통해 프로그램을 검증한다.

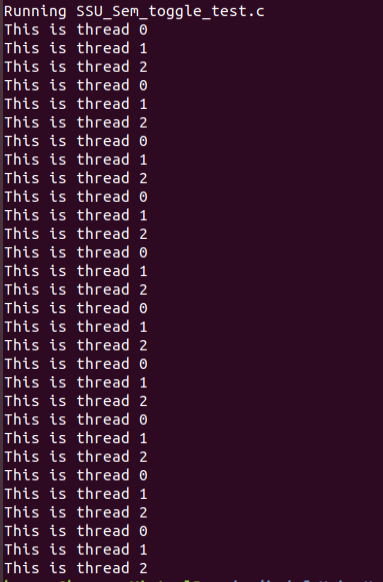
1. Pthread를 이용한 사용자 수준 세마포어(SSU\_Sem) 구현



세모포어를 구현하기전에 실행시킨 모습 과제의 명세에 적힌대로 main thread보다 thread1, thread2가 먼저 실행된 모습이다.



세마포어를 구현한 다음 실행시킨 모습 세마포어가 정상적으로 동작하여 main thread가 먼저 실행되고 이 후 다른 thread들이 실행된다.



구현한 세마포어를 사용하여 SSU\_Sem\_toggle.c를 완성시켜 출력이 일정하게 정렬된 것을 확인할 수 있다.

* **소스코드**

1. Master-Worker Thread Pool 구현

void \*generate\_requests\_loop(void \*data)

{

int thread\_id = \*((int \*)data);

while(1)

{

pthread\_mutex\_lock(&mutex);

if(item\_to\_produce >= total\_items) {

pthread\_mutex\_unlock(&mutex);

pthread\_cond\_broadcast(&cond);

return 0;

}

if(curr\_buf\_size >= max\_buf\_size)

{

pthread\_cond\_signal(&cond);

pthread\_cond\_wait(&cond,&mutex);

pthread\_mutex\_unlock(&mutex);

continue;

}

buffer[curr\_buf\_size++] = item\_to\_produce;

print\_produced(item\_to\_produce, thread\_id);

item\_to\_produce++;

pthread\_mutex\_unlock(&mutex);

}

return 0;

}

Master가 숫자를 생성하는 부분이다. 생성한 아이템의 개수가 총 생산해야 되는 아이템의 개수와 같아질 때 return 하도록 설계되었다. curr\_buf\_size 라는 변수로 현재 버퍼의 상태를 확인하며, 버퍼에 더 이상 원소가 들어갈 수 없으면 pthread\_cond\_wait을 통해 wait 상태에 들어가며 들어가기 전에 pthread\_cond\_signal을 호출한다. 아이템을 생성할 때마다 curr\_buf\_size와 item\_to\_produce를 증가시킨다.

void \*consume\_loop(void \*data)

{

int thread\_id = \*((int \*)data);

int i;

while(1)

{

pthread\_mutex\_lock(&mutex);

if(consumed\_item >= total\_items)

{

pthread\_mutex\_unlock(&mutex);

pthread\_cond\_broadcast(&cond);

return 0;

}

if(curr\_buf\_size<=0)

{

pthread\_cond\_signal(&cond);

pthread\_cond\_wait(&cond,&mutex);

pthread\_mutex\_unlock(&mutex);

continue;

}

curr\_buf\_size--;

print\_consumed(buffer[curr\_buf\_size],thread\_id);

consumed\_item++;

pthread\_mutex\_unlock(&mutex);

}

}

Consumer가 숫자를 소비하는 부분이다. Consume한 아이템의 개수가 총 아이템과 같아지면 프로그램이 종료되도록 설계되었고 현재 버퍼 안에 들어있는 아이템의 숫자가 0개 일 경우 pthread\_cond\_signal을 호출하고 wait 상태에 들어간다.

int main(int argc, char \*argv[])

{

int \*master\_thread\_id;

pthread\_t \*master\_thread;

int \*worker\_thread\_id;

pthread\_t \*worker\_thread;

item\_to\_produce = 0;

consumed\_item=0;

curr\_buf\_size = 0;

int i;

if (argc < 5) {

printf("./master-worker #total\_items #max\_buf\_size #num\_workers #masters e.g. ./exe 10000 1000 4 3\n");

exit(1);

}

else {

num\_masters = atoi(argv[4]);

num\_workers = atoi(argv[3]);

total\_items = atoi(argv[1]);

max\_buf\_size = atoi(argv[2]);

}

buffer = (int \*)malloc (sizeof(int) \* max\_buf\_size);

//create master producer threads

master\_thread\_id = (int \*)malloc(sizeof(int) \* num\_masters);

master\_thread = (pthread\_t \*)malloc(sizeof(pthread\_t) \* num\_masters);

for (i = 0; i < num\_masters; i++)

master\_thread\_id[i] = i;

worker\_thread\_id = (int \*)malloc(sizeof(int)\* num\_workers);

worker\_thread = (pthread\_t \*)malloc(sizeof(pthread\_t) \* num\_workers);

for(i=master\_thread\_id[num\_masters-1]+1;i<num\_workers;i++)

worker\_thread\_id[i]=i;

for (i = 0; i < num\_masters; i++)

pthread\_create(&master\_thread[i], NULL, generate\_requests\_loop, (void \*)&master\_thread\_id[i]);

//create worker consumer threads

for(i=0;i <num\_workers;i++)

{

pthread\_create(&worker\_thread[i],NULL,consume\_loop,(void\*)&worker\_thread\_id[i]);

}

//wait for all threads to complete

for (i = 0; i < num\_masters; i++)

{

void \*status;

pthread\_join(master\_thread[i], status);

printf("master %d joined\n", i);

}

for(i=0;i<num\_workers;i++)

{

void \*status;

pthread\_join(worker\_thread[i],status);

printf("worker %d joined\n",i);

}

pthread\_mutex\_destroy(&mutex);

pthread\_exit(NULL);

/\*----Deallocating Buffers---------------------\*/

free(buffer);

free(master\_thread\_id);

free(master\_thread);

free(worker\_thread\_id);

free(worker\_thread);

return 0;

}

주어진 메인함수에 worker 쓰레드들을 생성해주는 부분과 join 받는 부분을 추가했다.

1. Reader-Writer Locks 쓰레드 프로그램 구현

//rw\_lock.h

struct rw\_lock

{

pthread\_mutex\_t mutex;

pthread\_cond\_t cond;

pthread\_cond\_t cond2;

int read\_thread\_count;

int write\_thread\_flag;

int read\_waiting\_count;

int write\_waiting\_count;

};

Mutex하나와 2개의 cond\_t 그리고 lock을 얻기위해 대기중인 read, write thread의 개수, 그리고 현재 읽고/쓰고 있는 thread의 개수를 멤버 변수로 가지고 있는 rw\_lock 구조체이다.

//rw\_lock-r-test.c -> 읽기쓰레드 기본설정

#include "rw\_lock.h"

void init\_rwlock(struct rw\_lock \* rw)

{

// Write the code for initializing your read-write lock.

pthread\_mutex\_init(&(rw->mutex),NULL);

pthread\_cond\_init(&(rw->cond),0);

pthread\_cond\_init(&(rw->cond2),0);

rw->read\_thread\_count=0;

rw->write\_thread\_flag=0;

rw->read\_waiting\_count=0;

rw->write\_waiting\_count=0;

}

void r\_lock(struct rw\_lock \* rw)

{

rw->read\_thread\_count++;

}

void r\_unlock(struct rw\_lock \* rw)

{

rw->read\_thread\_count--;

if(rw->read\_thread\_count==0 )

pthread\_cond\_signal(&(rw->cond2));

}

void w\_lock(struct rw\_lock \* rw)

{

if(rw->write\_thread\_flag<=0)

{

pthread\_mutex\_t t\_mtx;

pthread\_mutex\_init(&t\_mtx,NULL);

pthread\_cond\_wait(&(rw->cond2),&t\_mtx);

pthread\_mutex\_destroy(&t\_mtx);

}

rw->write\_thread\_flag++;

pthread\_mutex\_lock(&(rw->mutex));

}

void w\_unlock(struct rw\_lock \* rw)

{

pthread\_cond\_broadcast(&(rw->cond2));

pthread\_mutex\_unlock(&(rw->mutex));

rw->write\_thread\_flag--;

}

init에서 멤버변수의 값들을 모두 기본값으로 설정해준다. r\_lock 에선 단순히 현재 읽기에 들어간 쓰레드의 개수를 증가시켜준다. r\_unlock 에선 읽기에 마무리 되었음으로 읽는 중인 쓰레드의 개수를 하나 제거해주고, 만약 현재 읽고 있는 쓰레드가 없다면 write thread 쪽으로 signal을 보내준다. w\_lock 에선 현재 쓰고 있는 쓰레드가 없으면 우선적으로 읽기 쓰레드 쪽에서 작업을 마무리하기를 기다린다. 읽기 쪽에서 작업을 마무리하고 시그널을 보내면, 쓰기를 시작한다. 현재 작업중인 쓰기 쓰레드의 숫자를 올리고, mutex로 lock을 걸어준다. w\_unlock에선 현재 읽고 있는 쓰레드의 개수를 하나 빼고 mutex를 unlock한다음, 쓰기 작업이 마무리 되었다고 broadcast해준다.

//rw\_lock-r-test.c -> 쓰기 쓰레드 기본설정

#include "rw\_lock.h"

void init\_rwlock(struct rw\_lock \* rw)

{

// Write the code for initializing your read-write lock.

pthread\_mutex\_init(&(rw->mutex),NULL);

pthread\_cond\_init(&(rw->cond),0);

pthread\_cond\_init(&(rw->cond2),0);

rw->read\_thread\_count=0;

rw->write\_thread\_flag=0;

rw->read\_waiting\_count=0;

rw->write\_waiting\_count=0;

}

void r\_lock(struct rw\_lock \* rw)

{

pthread\_mutex\_lock(&(rw->mutex));

while(rw->write\_waiting\_count!=0)

pthread\_cond\_wait(&(rw->cond),&(rw->mutex));

rw->read\_thread\_count++;

pthread\_mutex\_unlock(&(rw->mutex));

// Write the code for aquiring read-write lock by the reader.

}

void r\_unlock(struct rw\_lock \* rw)

{

pthread\_mutex\_lock(&(rw->mutex));

rw->read\_thread\_count--;

pthread\_mutex\_unlock(&(rw->mutex));

pthread\_cond\_signal(&(rw->cond2));

// Write the code for releasing read-write lock by the reader.

}

void w\_lock(struct rw\_lock \* rw)

{

pthread\_mutex\_lock(&(rw->mutex));

rw->write\_waiting\_count++;

if(rw->read\_thread\_count || rw->write\_thread\_flag)

{

pthread\_cond\_wait(&(rw->cond2),&(rw->mutex));

}

rw->write\_thread\_flag++;

pthread\_mutex\_unlock(&(rw->mutex));

// Write the code for aquiring read-write lock by the writer.

}

void w\_unlock(struct rw\_lock \* rw)

{

pthread\_mutex\_lock(&(rw->mutex));

rw->write\_waiting\_count--;

rw->write\_thread\_flag--;

if(rw->write\_waiting\_count>0)

pthread\_cond\_signal(&(rw->cond2));

else

pthread\_cond\_broadcast(&(rw->cond));

pthread\_mutex\_unlock(&(rw->mutex));

// Write the code for releasing read-write lock by the writer.

}

Init 부는 reader부분과 동일하다. r\_lock에서 쓰기를 대기하고 있는 쓰레드가 확인되면 추가적인 read 작업을 wait을 통해 잠시 멈춘다. r\_unlock에선 읽고있는 쓰레드의 개수를 하나 빼고 write 쓰레드쪽으로 signal을 전송한다. w\_lock이 호출되면 바로 대기하고 있는 쓰기 쓰레드의 개수를 증가시키고 현재 읽고있는 쓰레드의 개수나 쓰고있는 쓰레드의 개수가 0이 아니라면 우선 대기한다. 대기가 끝나면 현재 쓰고 있는 쓰레드의 숫자를 증가시킨다. r\_unlock에선 대기하고있는 쓰레드의 개수, 쓰고 있는 쓰레드의 개수를 감소시킨다. 그리고 만약 대기하고 있는 쓰기 쓰레드가 있으면 쓰기 쓰레드 쪽으로 시그널을 보내고 아니라면 읽기 쓰레드쪽으로 시그널을 전송한다.

1. Pthread를 이용한 사용자 수준 세마포어(SSU\_Sem) 구현.

//SSU\_Sem.h

typedef struct SSU\_Sem {

int count;

pthread\_cond\_t cond;

pthread\_mutex\_t mutex;

} SSU\_Sem;

Count, pthread\_cond\_t, pthread\_mutex\_t 3개의 멤버가 있는 구조체를 사용했다.

//SSU\_Sem.c

void SSU\_Sem\_init(SSU\_Sem \*s, int value) {

s->count=value;

pthread\_cond\_init(&(s->cond),NULL);

pthread\_mutex\_init(&(s->mutex),NULL);

}

void SSU\_Sem\_down(SSU\_Sem \*s) {

pthread\_mutex\_lock(&(s->mutex));

if(s->count==0)

{

pthread\_cond\_wait(&(s->cond),&(s->mutex));

}

s->count--;

pthread\_mutex\_unlock(&(s->mutex));

}

void SSU\_Sem\_up(SSU\_Sem \*s) {

pthread\_mutex\_lock(&(s->mutex));

s->count++;

pthread\_cond\_signal(&(s->cond));

pthread\_mutex\_unlock(&(s->mutex));

}

init부분은 세마포어의 첫 값을 count에 value를 인자로 받아 저장하고 cond, mutex 2개의 변수를 초기값으로 설정해준다.

SSU\_Sem\_down 에서는 count가 0일 시에는 thread를 wait을 사용하여 block해주고, count를 1 감소시킨다.

마지막으로 SSU\_Sem\_up은 count를 1증가시키는 역할과, pthread\_cond\_signal을 통해 대기중인 쓰레드를 깨워준다.

//SSU\_Sem\_toggle.c

SSU\_Sem \*sem;

int count;

void \*justprint(void \*data)

{

int thread\_id = \*((int \*)data);

int j;

for(int i=0; i < NUM\_ITER; i++)

{

SSU\_Sem\_down(sem+(thread\_id));

printf("This is thread %d\n", thread\_id);

SSU\_Sem\_up(sem+(thread\_id+1)%NUM\_THREADS);

}

return 0;

}

세마포어를 쓰레드의 숫자만큼 배열로 만들어 세마포어를 통해 숫자출력이 정렬되게 하였다.

int main(int argc, char \*argv[])

{

count=0;

pthread\_t mythreads[NUM\_THREADS];

int mythread\_id[NUM\_THREADS];

sem= malloc(sizeof(SSU\_Sem) \* NUM\_THREADS);

int i;

for(i=0;i<NUM\_THREADS;i++)

{

SSU\_Sem\_init(sem+i,0);

}

SSU\_Sem\_init(sem,1);

앞 소스코드의 메인부분이며 SSU\_Sem 구조체의 배열을 초기화하고, 첫번째 세마포어의 초기값만 1로 설정하여 첫 쓰레드부터 출력이 되도록 하였다.