오퍼레이팅 시스템 7주차 과제

12180626 성시열

6. Homework

1) Try below (ex1.c) and explain the result.

#include <stdio.h>

unsigned long long sum=0;

void main(){

int x=fork();

if (x==0){ // child

int i, j;

for(i=0;i<20000;i++)

for(j=0;j<2000;j++)

sum++;

printf(“child sum:%llu \n”,sum);

}else{ // parent

int i, j;

for(i=0;i<20000;i++)

for(j=0;j<2000;j++)

sum++;

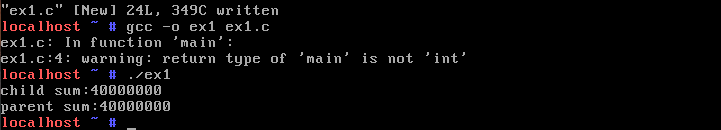
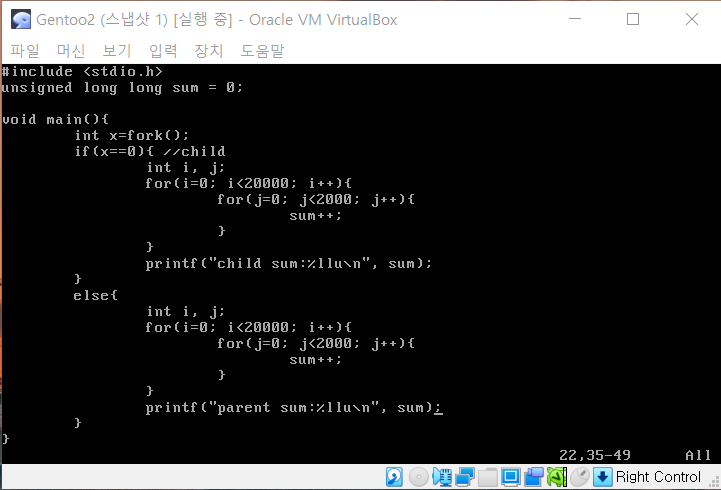
printf(“parent sum:%llu\n”,sum);

}

}

#gcc -o ex1 ex1.c

#./ex1



이중 포문으로 인해 20000\*2000 = 40000000번 반복되며, 부모와 자식에서 모두 sum:40000000임을 알 수 있다.

2) Try below (th.c) and explain the result.

#include <stdio.h>

#include <string.h>

#include <pthread.h>

#include <stdlib.h>

#include <unistd.h>

pthread\_t t1, t2; // thread 1, thread 2

unsigned long long sum=0;

void \* foo1(void \*arg){

int i,j;

for(i=0;i<20000;i++){

for(j=0;j<2000;j++)

sum += 1;

}

printf(“thread 1 sum:%llu\n”, sum);

return NULL;

}

void \* foo2(void \*arg){

int i,j;

for(i=0;i<20000;i++){

for(j=0;j<2000;j++)

sum += 1;

}

printf(“thread 2 sum:%llu\n”, sum);

return NULL;

}

int main(void){

pthread\_create(&t1, NULL, &foo1, NULL);

pthread\_create(&t2, NULL, &foo2, NULL);

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

return 0;

}

# gcc -o th -lpthread th.c

#./th

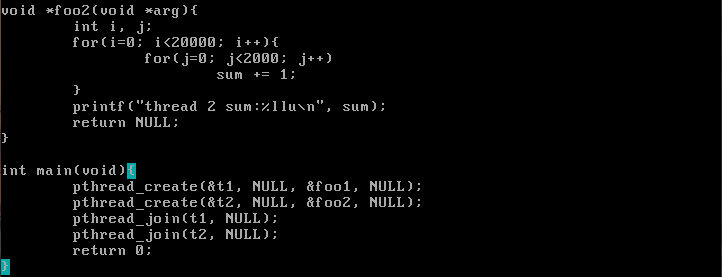
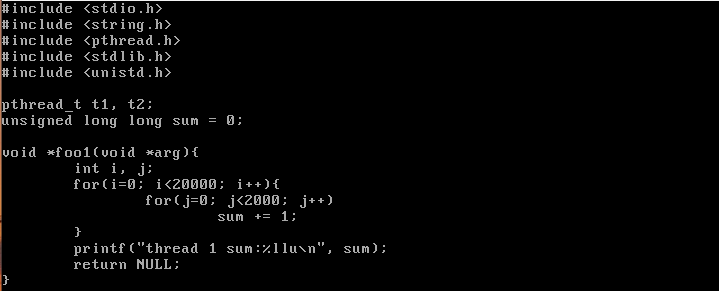
....

#./th

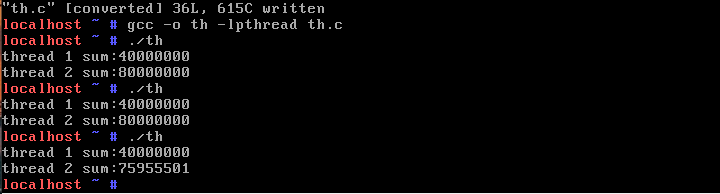
....

#./th

....



코드를 그대로 작성해보았다. foo1, foo2에서 sum++을 40000000씩 반복한다. Sum은 공유 변수이기 때문에 foo1과 foo2에서의 sum 값이 그대로 이어져 sum 값을 더한다.



실행시켜보면 foo1과 foo2에서 각각 2000000번 반복하였을 때 thread1값이 40000000일 것이다. 이후 나머지 시간을 실행시키면 thread2 값이 80000000이 된다. Thread 1 값은 40000000보다 크거나 같아야 하며 thread 2값이 80000000이어야 오류가 발생하지 않는 것이다. 실행을 3번정도 반복하여 해본 결과 마지막 실행에서 thread 2값이 7595501이 나왔음을 알 수 있고, 이러한 현상을 race condition이라고 한다. 이는 함수를 진행하며 critical section을 진행할 때 도중에 함수가 끊기지 않도록 semaphore로 보호해주어야 한다.

3. Try below(th2.c) and explain the result.

#include <stdio.h>

#include <string.h>

#include <pthread.h>

#include <stdlib.h>

#include <unistd.h>

pthread\_t t1, t2; // thread 1, thread 2

pthread\_mutex\_t lock; // semaphore

unsigned long long sum=0;

void \* foo1(void \*arg){

int i,j;

for(i=0;i<20000;i++){

pthread\_mutex\_lock(&lock);

for(j=0;j<2000;j++)

sum += 1;

pthread\_mutex\_unlock(&lock);

}

printf(“thread 1 sum:%llu\n”, sum);

return NULL;

}

void \* foo2(void \*arg){

int i,j;

for(i=0;i<20000;i++){

pthread\_mutex\_lock(&lock);

for(j=0;j<2000;j++)

sum += 1;

pthread\_mutex\_unlock(&lock);

}

printf(“thread 2 sum:%llu\n”, sum);

return NULL;

}

int main(void){

pthread\_mutex\_init(&lock, NULL);

pthread\_create(&t1, NULL, &foo1, NULL);

pthread\_create(&t2, NULL, &foo2, NULL);

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

pthread\_mutex\_destroy(&lock);

return 0;

}

# gcc -o th2 -lpthread th2.c

#./th2

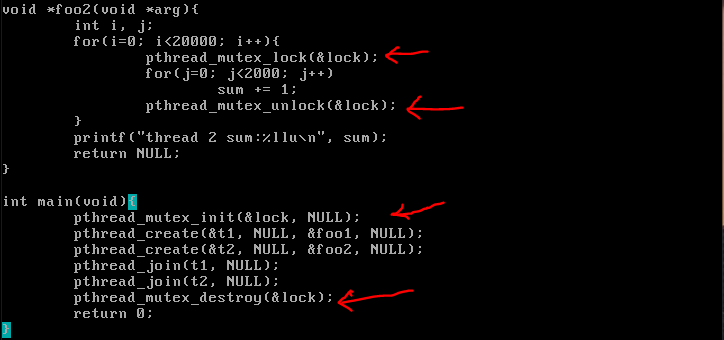
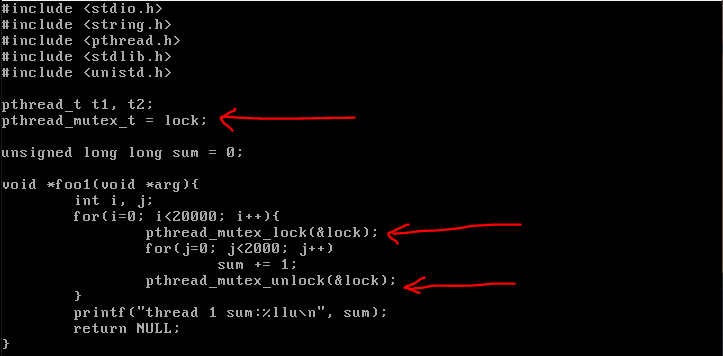
...

#./th2

...

#./th2

.....



위 th.c 파일에서 해당 표시한 부분만 추가하여 코드를 작성한다. sum+=1 과정을 반복하는 반복문. 즉 Critical Section 전후로 pthread\_mutex\_lock(&lock)과 pthread\_mutex\_unlock(&lock)을 통해 CS를 보호한다.



컴파일 후 실행한 결과 3번 실행문 모두 race condition 없이 잘 출력됨을 확인하였다. Semaphore를 통해 Critical Section에서 정지되는 것을 방지한다.(CS에 프로세스가 들어가면서 sema = 1로 바뀜)

4. (Deadlock) Try below(th3.c) and explain the result. Modify the code so that it won't have a deadlock.

#include <stdio.h>

#include <string.h>

#include <pthread.h>

#include <stdlib.h>

#include <unistd.h>

pthread\_t t1, t2; // thread 1, thread 2

pthread\_mutex\_t lock1; // semaphore 1 for sum 1

pthread\_mutex\_t lock2; // semaphore 2 for sum 2

unsigned long long sum1=0;

unsigned long long sum2=0;

void \* foo1(void \*arg){

int i,j;

for(i=0;i<20000;i++){

pthread\_mutex\_lock(&lock1);

pthread\_mutex\_lock(&lock2);

for(j=0;j<2000;j++)

sum1 += 1;

pthread\_mutex\_unlock(&lock1);

for(j=0;j<2000;j++)

sum2 += 1;

pthread\_mutex\_unlock(&lock2);

}

printf(“thread 1 sum1:%llu\n”, sum1);

printf(“thread 1 sum2:%llu\n”, sum2);

return NULL;

}

void \* foo2(void \*arg){

int i,j;

for(i=0;i<20000;i++){

pthread\_mutex\_lock(&lock2);

pthread\_mutex\_lock(&lock1);

for(j=0;j<2000;j++)

sum1 += 1;

pthread\_mutex\_unlock(&lock1);

for(j=0;j<2000;j++)

sum2 += 1;

pthread\_mutex\_unlock(&lock2);

}

printf(“thread 2 sum1:%llu\n”, sum1);

printf(“thread 2 sum2:%llu\n”, sum2);

return NULL;

}

int main(void){

pthread\_mutex\_init(&lock1, NULL);

pthread\_mutex\_init(&lock2, NULL);

pthread\_create(&t1, NULL, &foo1, NULL);

pthread\_create(&t2, NULL, &foo2, NULL);

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

pthread\_mutex\_destroy(&lock1);

pthread\_mutex\_destroy(&lock2);

return 0;

}

# gcc -o th3 -lpthread th3.c

#./th3

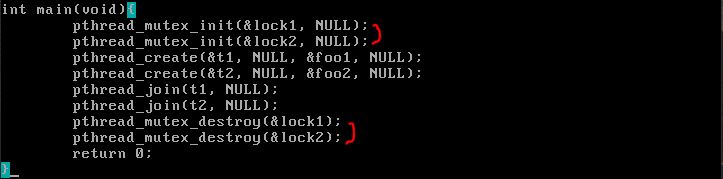
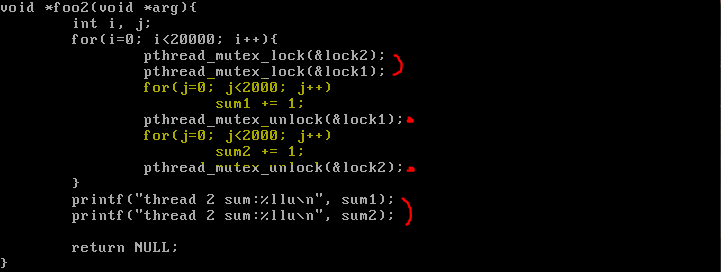
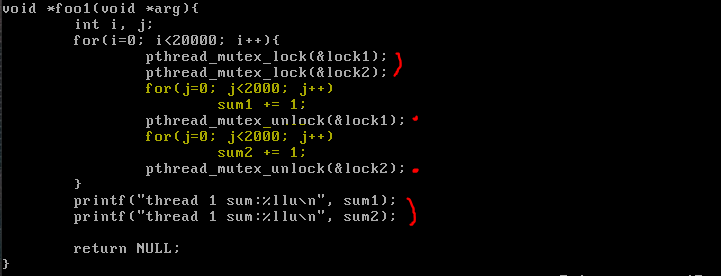
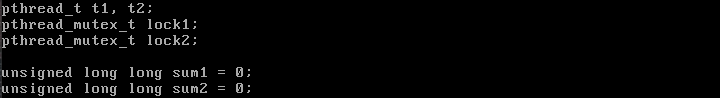
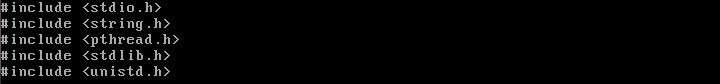
...

#./th3

...

#./th3

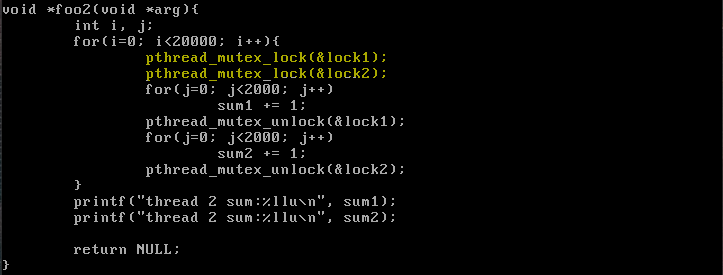
.....



Deadlock은 공유변수가 2개이고 semaphore 변수가 2개일 때 locking과 releasing의 순서가 꼬이면 발생하는 오류이다. 두 공유변수는 samephore를 통해 프로세스가 Critial Section을 탈출할 때까지 기다리는데 locking/releasing의 순서가 꼬이게 되면 서로의 semaphore를 기다리는 상태가 된다.



실행과 동시에 아무것도 출력되지 않고 무한루프에 빠진 듯이 대기함을 알 수 있다.

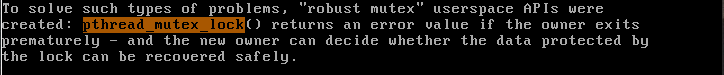
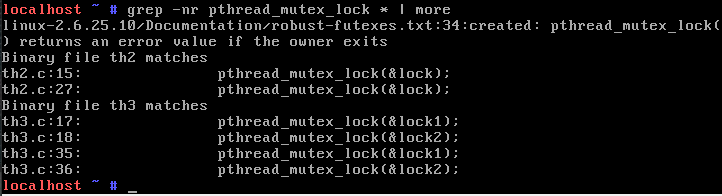


Deadlock 해제를 위해 locking 순서를 알맞게 변경해주었다.



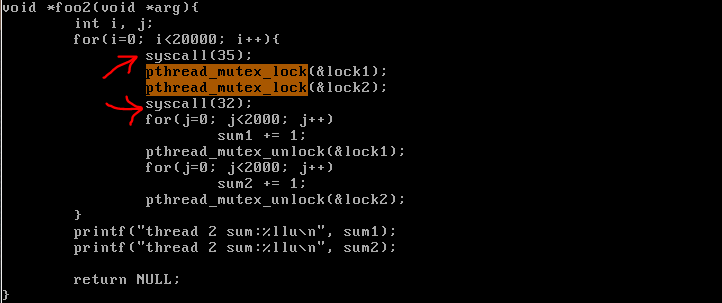
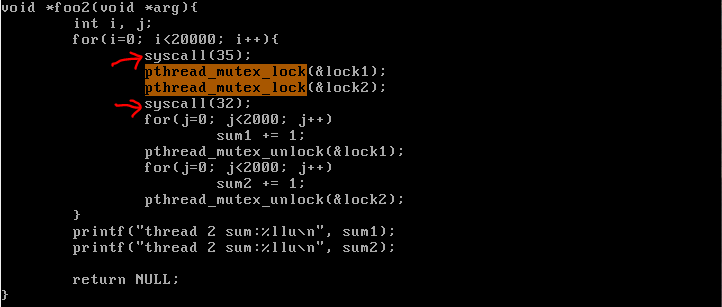
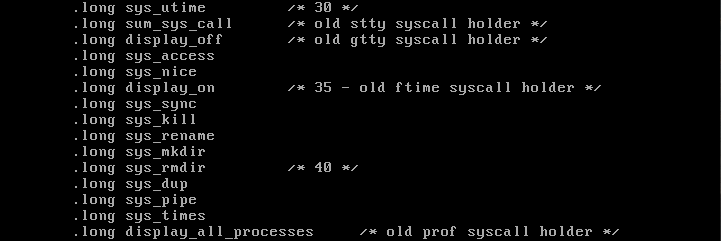
컴파일 후 실행한 결과 두번째 쓰레드에서 출력이 모두 80000000으로 race condition 없이 잘 실행된 모습이다.

5. Find out the ISR2 function for pthread\_mutex\_lock() and trace the code. You can do kernel tracing also in the following site: <https://elixir.bootlin.com/linux/latest/ident/>. Select the right version (v2.6.25.10) and type the ISR2 name in the search box.

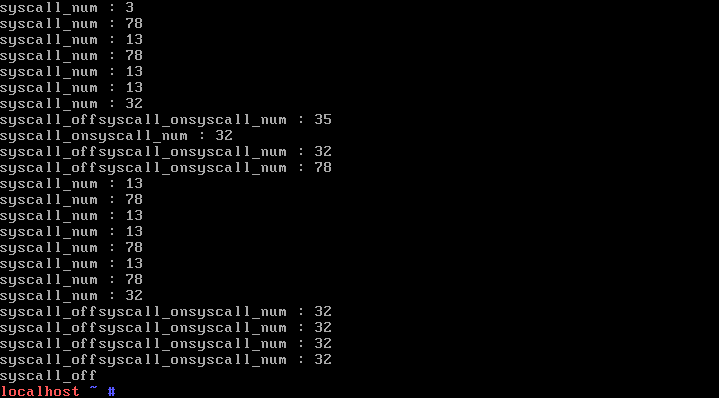


위 방법으로 함수가 정의된 부분을 파악하고 실제 동작이 무엇인지 확인해보려 했으나 pthread\_mutex\_lock은 printf처럼 C 내부의 함수이고, 리눅스 자체 함수가 아니기 때문에 적절히 찾지 못하였다.

arch/x86/kernel/syscall\_table\_32.S에서 system call 호출하는 함수를 찾기



컴파일 후 echo 8 > /proc/sys/kernel/printk 후 실행해본 결과



무수히 출력되어 ctrl+c를 통해 중지하고 살펴보았다.

3:read

13:time

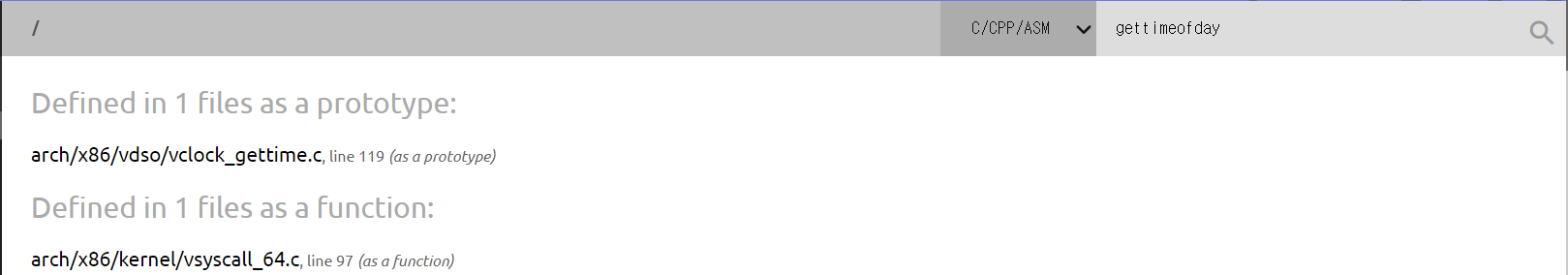
32:display off

35:display on

78:gettimeofday

78과 13이 반복되며 계산되고 32와 35로 함수의 시작과 끝을 알 수 있었다.

사이트를 통해 검색해보았다.



해당 위치로 가서 함수 정의를 확인해보았다.

