**2018008304 컴퓨터소프트웨어학부 박경하**

**운영 체제 HW#5**

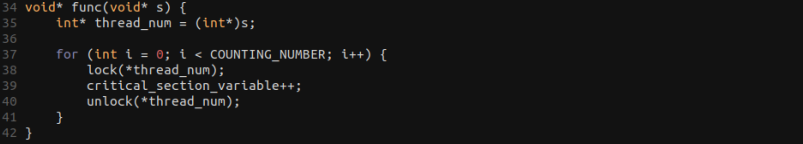
**제출 일자: 2021/04/20**

1. **과제 A**
2. **자료구조 설명**

****

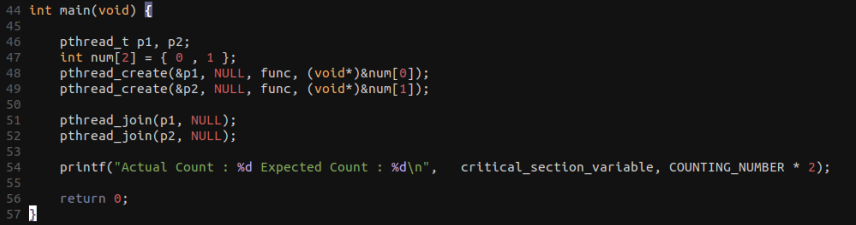
<atomic> 자료구조는 정수형 같은 타입의 산술연산을 atomic하게 수행하도록 도와주는 클래스로, CPU가 명령어 1개를 처리하는 중간에 다른 스레드가 끼어들 여지가 전혀 없도록 합니다. atomic\_int은 atomic한 int형 변수를 선언하게 되며, lock 또는 store와 같이 메모리에 접근할 때 어떠한 방식으로 접근하는지를 memory\_order로 정할 수 있습니다. 여기서 memory\_order\_seq\_cst는 메모리 명령의 순차적 일관성을 보장해주는 디폴트이며, 그 외에도 조금 더 느슨한 memory\_order\_relexed, 명령 앞의 메모리 명령들이 재배치되는 것을 금지하는 memory\_order\_release, 그 반대인 memory\_order\_acquire 등이 있습니다.

1. **동기화 방법 설명**

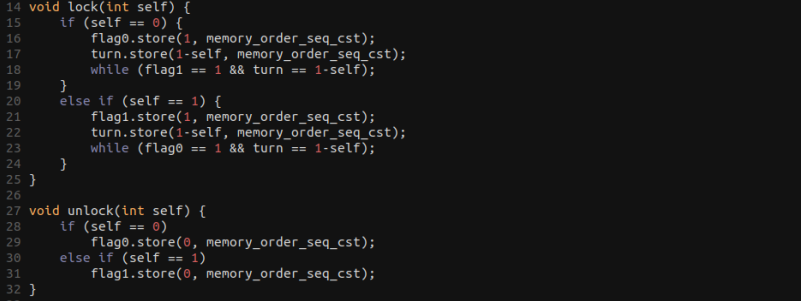
****

Race Condition을 해결하기 위해, 즉 critical section을 atomic하게 실행하기 위해 수업에서 배운 Peterson Algorithm을 적용하였습니다. 이는 Mutual Exclusion 동기화 기법의 한 종류로, 한 프로세스가 critical section에 진입할 때에는 다른 프로세스가 진입하지 못하도록 막습니다. 이를 구현하기 위해 lock, unlock 를 사용하였으며, 이는 만약 다른 스레드가 critical section 안에 있다면 while문을 이용해 무한정으로 기다린 뒤, 그 스레드가 작업을 마친 뒤 critical section에서 나와 flag를 flase로 바꾸게 되면 비로소 ciritical section에 들어갈 수 있게 되는 원리입니다. 이로 인해 한 스레드가 작업을 마칠 때까지 다른 스레드를 critical section에 접근하지 못하게 막음으로써 race condition을 막게 됩니다.

1. **프로그램 구조 설명**

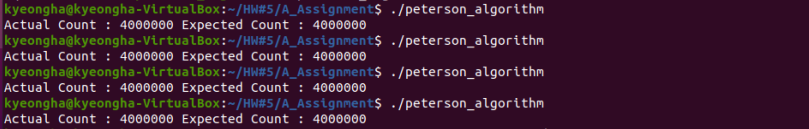
****

문제를 해결하기 위한 프로그램 구조는 위와 같습니다. 먼저 스레드 [0], [1]으로 2개 생성해줄 때 인자로 func함수로 넘어가게 됩니다. func 함수에서는 2번의 사진과 같이 해당 index를 인자로 넘겨 받은 뒤, lock과 unlock을 통해 race condition을 해결하며 해당 함수에서는 아래와 같은 동작을 수행합니다.

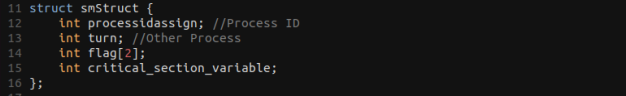


lock에서는 동기화를 위해 현재 자신의 스레드의 flag를 ture(1)로 넣어줌으로, 자신이 critical section에 들어가고 싶음을 알립니다. 그리고 while문을 통해 다른 스레드가 있는지를 무한정으로 확인한 뒤, critical section에서 나오면 자신의 스레드가 그 section에 들어가게 됩니다. 이 때, 실습에서는 동기화를 하더라도 기대하는 Count의 값과 실제 동작한 Count의 값이 여전히 차이가 발생했습니다. 그 이유는 CPU가 자동으로 최적화를 진행하면서 실행의 순서가 미세하게나마 달라져 약간의 차이가 발생하게 되는데, 이 문제를 해결하기 위해 과제에서는 1번의 자료구조처럼 atomic 자료구조를 이용하였습니다. atomic 자료구조의 memory ordering을 통해 메모리 명령 재배치를 하지 않도록 하여 우리가 시뮬레이션하여 원하는 그대로의 결과값이 나오도록 합니다. 그래서 실습에서와는 달리 flag와 turn 변수에 값을 저장할 때 store을 이용하여 memory ordering을 설정해주었습니다.

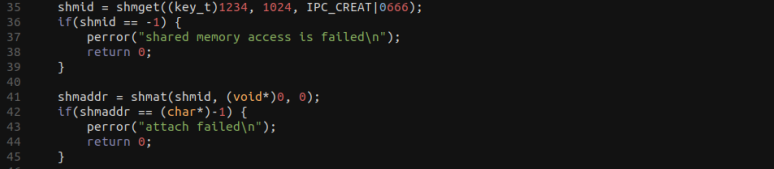
그 결과, 원하는 것처럼 기대값과 실제로 작동한 값이 일치하는 것을 볼 수 있었습니다.



1. **과제 B**
2. **자료구조 설명**

****

과제B에서 사용한 자료구조 중, 먼저 smStruct는 shared\_memory 의 데이터들을 저장할 struct이며, 자식 프로세스 2개를 선언하여 모두 이 smStruct의 데이터들을 공유하게 됩니다. 이 때, 각각의 자식 프로세스에서 공유한 smStruct의 critical\_section\_count 값을 올려줌으로써 동기화가 잘 되었음을 확인할 수 있게 됩니다.

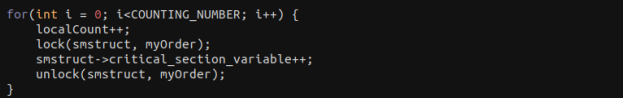


<B\_Assignment.c>

shared memory는 위에서 설명한 바와 같이 둘 이상의 프로세스가 공통의 메모리에 접근할 수 있도록 하며, shmget으로 공유할 메모리 공간을 커널에 요청하며, 1234값을 가진 공유의 공유메모리, 사이즈는 1024, 권한은 0666으로 주었습니다. 그리고 그 값을 shmat함수를 이용하여 int형의 식별자로 바꾸고, shmaddr에 int형 주소를 저장한 뒤, 위의 자료구조 smStruct와 맵핑시킴으로써 shmaddr의 주소를 가진 smStruct의 공유메모리를 만들었습니다.

또한, fork()를 이용해 프로세스를 생성한 뒤, child.c 코드를 실행하는 프로세스로 이어지게 되며, waitpid함수를 이용해 pid의 status가 0으로 반환되어 자식프로세스가 모두 종료될 때까지 기다린 후, 종료합니다.

1. **동기화 방법 설명**

****

<child.c>

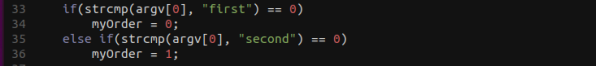
과제B에서도 마찬가지로 동기화를 위해 peterson algorithm을 이용하였습니다. myorder에 자신이 몇 번째 프로세스인지 저장한 뒤에, 해당 프로세스가 critical section에 들어가길 원하는 것을 flag에 1을 넣어줌으로써 알리며, 과제A와 동일하게 while문에서 무한반복을 거쳐 critical section에 프로세스가 존재하지 않을 때까지 기다립니다.

1. **프로그램 구조 설명**

****

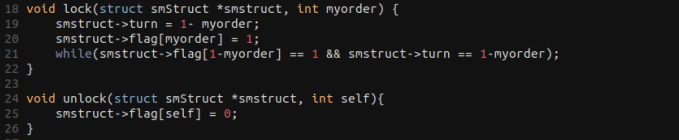
<B\_Assignment.c>

부모 프로세스에서 1번과 같이 smStruct를 가진 공유메모리를 생성해준 뒤에, fork()를 이용해 2개의 자식프로세스를 만들게 되며 자식프로세스로 넘어간 뒤, waitpid를 이용하여 2개의 자식프로세스가 모두 종료될 때까지 부모프로세스는 종료하지 않도록 합니다.



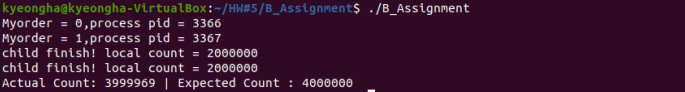
<child.c>

이 때, execl 인자에 프로세스 idx인 0을 char\* 형태로 넣어주어야 하는데, (char\*)0을 넣게 되면 인자를 넣지 않은 것으로 인식하기 때문에 “first”, “second”로 각각 넣어주었습니다. 그래서 자식프로세스에서 first이면 0을 idx로 인식하여 myorder에 저장하며, second이면 1을 idx로 인식하여 myorder에 저장하는 방식으로 설계하였습니다.



<child.c>

그 뒤, 마찬가지로 child 프로세스에서도 공유메모리에 접근하기 위해 부모 프로세스와 동일한 key값과 메모리 공간을 가진 공유메모리를 생성하여 smStruct와 매핑시켜줍니다. 그 뒤, 자식에서 연산한 횟수를 저장하기 위해 localCount를 1씩 증가시키고, 총 연산횟수를 저장하기 위해 critical\_section\_variable을 1씩 증가시켜줍니다. 이 때, critical\_section\_variable은 공유메모리에 저장하여 race condition이 일어나게 하지 않기 위해 peterson algorithm을 이용합니다. lock과 unlock에서는 공유메모리 안의 데이터 값에 저장하는 것을 제외하고는 과제A와 동일하게 진행하였습니다.



그리고, 과제B에서는 따로 atomic 자료구조를 이용하는 것과 같이 CPU의 최적화를 해결하기 위한 memory ordering을 해주지 않았기 때문에 결과값은 실제 기대하는 값과 상이하는 현상이 발생했습니다.



<child.c>

참고로, process pid값을 불러오기 위해 getpid()를 이용하였으며, 이 값을 공유메모리 idassign에 저장하여 출력하였습니다. 만약 공유메모리와 Peterson Algorithm을 사용하지 않았다면, 이 값 또한 변동이 있었을 거라고 예상합니다.