**2017029589 컴퓨터소프트웨어학부 류지범**

**운영 체제 HW#5**

**제출 일자: 2021/04/07**

1. **과제 A**
2. **자료구조 설명**

COUNTING\_NUMBER \* 2까지 숫자를 증가시키는데, 쓰레드 두 개를 생성해서 각각 COUNTING\_NUMBER만큼 critical\_section\_variable을 증가시키도록 한다.

피터슨 알고리즘을 사용해서 lock과 unlock을 통해 동기화를 시킨다.

1. **동기화 방법 설명**

우선 피터슨 알고리즘을 이용해서 lock과 unlock을 통해 두 쓰레드가 동시에 변수에 접근하지 못하도록 한다. 하지만 현대 CPU의 최적화로 인해 실제 instruction의 실행 순서가 미세하게 달라져서 우리가 원하는 결과를 얻지 못한다. 이에 순서를 지칭하는 turn과 flag를 atomic 변수로 선언한다. Atomic 클래스는 volatile 변수 없이 연산을 가능하게 해주므로 atomic 클래스의 함수들을 이용만 하면 동기화를 시킬 수 있다.

우선 기본적인 코드의 틀은 유지했다. 문제점은 cpu가 최적화를 통해서 서로 의존성이 없다고 판단했을 때 코드의 순서를 마음대로 바꾼다는 것인데, flag와 turn값을 저장하는 과정의 순서도 이때 바뀔 수 있으므로 atomic 함수를 사용해서 의도적으로 이 부분의 코드가 상호 배제가 깨지지 않도록 순차적으로 실행되도록 유도하도록 한다. 이를 위해서 atomic 클래스의 store함수를 사용했다. Store 함수를 사용함으로써 flag = true, turn = 1 – self의 순서를 유지해주고, lock 과정이 atomic하게 이루어지도록 했다. Lock 과정을 atomic 하게 해줌으로써 두 쓰레드가 피터슨 알고리즘을 우리가 의도한대로 실행한다.

1. **프로그램 구조 설명**

**텍스트이(가) 표시된 사진

자동 생성된 설명**우선 프로그램 구조는 실습에서 Peterson\_algorithm을 구현한 것과 동일하게 작성했다. 쓰레드를 두 개를 만들고 쓰레드의 번호를 인자값으로 넘겨주면서 func을 실행하도록 했다. 쓰레드가 종료될 때까지 pthread\_join을 통해 기다리도록 하고, 다 종료되면 실제 값과, 기대한 값을 출력하도록 한다. 쓰레드가 실행할 함수는 COUNTING\_NUMBER 까지 값을 올리는 것이며, critical\_section\_variable을 건드리는 것이므로 피터슨 알고리즘을 통해 lock과 unlock을 통해 동기화를 해주어서, 변수에 동시에 접근하지 못하도록 한다. 현대 CPU의 최적화 문제로 인해 완벽한 동기화가 이루어지지 못하는 부분은 atomic 변수를 사용했다. Lock은 쓰레드 번호에 해당하는 쓰레드가 들어가겠다고 자신의 flag 값을 1로 변경하고, turn은 다른 쓰레드의 값으로 지정해주고, 다른 쓰레드가 값을 변경할 때까지(while(flag1 && turn == 1);) 기다리도록 한다. Lock을 실행한 후 critical\_section\_variable의 값을 올리고, unlock을 한다. Unlock은 자신의 flag 값을 0으로 바꾸는 것으로 한다.

1. **과제 B**
2. **자료구조 설명**

자식 프로세스와 부모프로세스가 통신할 수 있도록 shared\_memory IPC를 사용했다. 공유 메모리를 생성하고, attach해서 메모리를 붙이고, 역할을 다 하면 detach 후 delete하는 과정을 진행한다. Shared\_memory에서 사용할 것으로 구조체를 사용했고, 이 구조체에는 프로세스의 ID, turn, flag, critical\_section\_variable이 담겨있다. 자식 프로세스를 생성하는 것은 fork()를 통해 진행했고, execl로 자식 프로세스가 프로그램을 실행하도록 했다.

1. **동기화 방법 설명**

동기화는 피터슨 알고리즘을 사용했다. 구조체에 들어있는 공유변수인 critical\_section\_variable을 두 개의 자식 프로세스가 접근해서 값을 올린다.

각각 자식프로세스는 COUNTING\_NUMBER 만큼 critical\_section\_variable을 올리는데, 두 프로세스가 변수에 동시에 접근하면 안되기 때문에 피터슨 알고리즘의 lock과 unlock을 이용했다. Lock 과정에선 자신의 프로세스 번호에 해당하는 flag 값을 1로 바꿔주고 turn은 상대방의 프로세스 번호에 해당하는 1 – self 값으로 바꿔준다. 이후 다른 프로세스가 값을 변경할 때까지 while(smstruct->flag[1-self] && smstruct->turn == 1-self);를 통해 기다려준다. 기다린 이후 critical\_section\_variable의 값을 올리고 unlock을 한다. Unlock은 자신의 flag 값을 0으로 바꿔준다. Lock과 unlock 과정 사이에 critical\_section\_variable을 올림으로써 동기화가 이루어지고, 우리가 원하는 결과를 얻을 수 있게 된다.

1. **프로그램 구조 설명**

부모 프로세스와 자식 프로세스는 공유 메모리를 통해 통신하기 때문에, 이를 위해 사용할 구조체를 둘 다 선언해준다. smStruct라는 구조체를 만들었는데 이 안에는 프로세스id, turn, flag, critical\_section\_variable 변수가 존재한다. 부모 프로세스는 우선 구조체를 선언하고, key 값에 해당하는 공유 메모리를 만든다. 공유 메모리를 attach하고 smstruct에 공유메모리를 매핑해준다. 이후 turn 값을 0, 프로세스 id를 0, flag값들을 0, critical\_section\_variable의 값을 0으로 초기화해준다. 이후 fork를 2번을 해주어서 자식 프로세스를 두 개를 만들고 각각 ./child를 실행하도록 한다. 첫 번째 fork로 실행되는 자식은 processidassign 값으로 0을 가진채로 child를 실행하도록 했고, 두 번째 fork로 실행되는 자식은 processidassign 값으로 1을 가지도록 선언해주고 child를 실행하도록 했다. 다만 이 때 fork가 거의 동시에 이루어져서 processidassign의 값이 우리가 의도한대로 서로 다르게 들어가지 않는 경우가 발생하기 때문에, usleep(2000)을 통해 의도적으로 두 번째 자식 프로세스가 조금 더 늦게 실행되도록 유도했다. 두 자식 프로세스는 각각 processidassign 값으로 0, 1을 가지게 된다. 부모 프로세스는 자식 프로세스가 모두 종료될 때까지 waitpid를 통해 기다리고, 실제 count 값과 기대했던 count값을 출력한 후 공유 메모리를 detach하고 삭제한 후 종료한다.

자식 프로세스도 부모 프로세스와 마찬가지로 key값에 해당하는 공유 메모리를 생성하고, attach해서 붙인다. 이후 smstruct에 공유메모리를 매핑한다. 프로세스의 순서에 해당하는 Myorder에 이전에 우리가 넣었던 processidassign 값을 할당해준다. 이후 자식 프로세스는 본인의 순서를 출력하고 자신의 pid를 출력한다. 이후 값을 증가시키는 함수를 실행하는데 인자 값으로 구조체와, Myorder를 넣어준다. Func은 자식 프로세스의 로컬 변수인 localcount를 COUNTING\_NUMBER만큼 올려주고, lock과 unlock을 사용해서 critical\_section\_variable의 값을 올려준다. Lock과 unlock의 인자 값으로도 func과 동일하게 구조체와 Myorder을 넘겼다. Lock은 피터슨 알고리즘에 따라 자신의 flag 값을 1로 바꾸고, turn은 상대방으로 바꾼다. 이후 상대방이 값을 바꾸는 것을 마칠 때까지 while문을 통해서 기다리도록 한다. 기다린 이후 critical\_section\_variable의 값을 올려주고, unlock을 통해 자신의 flag값을 0으로 바꿔줌으로써 자신의 값 변경이 다 끝났음을 알린다.

Func을 실행한 이후 자식 프로세스는 자신의 로컬 변수인 localcount를 출력하고 종료한다. 이 로컬변수는 우리가 지정해놓은 값과 동일하게 나오게 된다.텍스트이(가) 표시된 사진

자동 생성된 설명