**2019062833 컴퓨터 소프트웨어학부 김유진**

**운영 체제 HW#5**

**제출 일자: 2021/04/15**

1. **과제 A**
2. **자료구조 설명**

1)atomic

-atomic : atomic 변수를 선언할 수 있는 데이터형

-fetch\_add : atomic의 멤버 함수로, 값을 증가시키는 함수

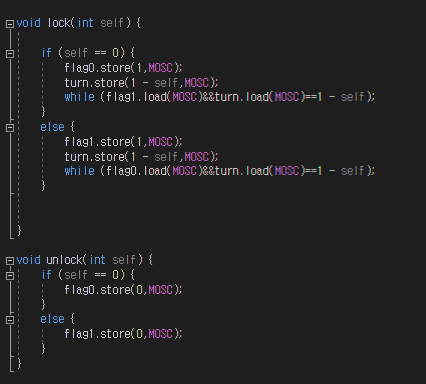
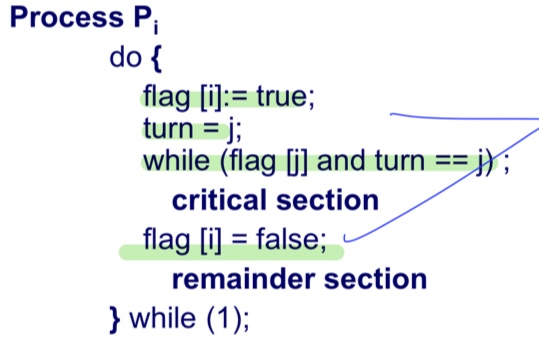
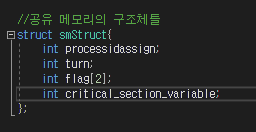
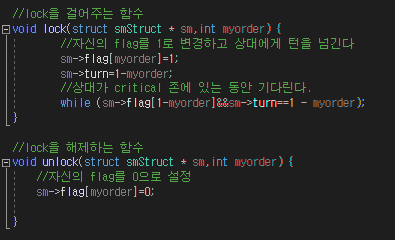
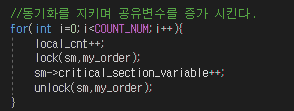
-fetch\_sub : atomic의 멤버 함수로, 값을 감소시키는 함수

-atomic으로 설정된 변수에는 초기에만 값 대입이 가능하고 그 후로는 증감 역할만 수행

-atomic으로 선언된 변수를 이용해 어떤 동작을 하는 중에는 다른 스레드들이 해당 atomic 변수를 절대 사용 불가

-따라서 동작이 매우 오래 걸려 성능을 저하시킬 수 있음

-멀티 스레드 환경에서 lock을 사용하지 않고 한 줄의 명령으로 방해받지 않고 실행시킬 수 있음

1. **동기화 방법 설명**
   1. std::atomic으로 turn, flag 변수 선언
      1. memory ordering 방식(아래 추가 레포트 참고) 중 memory\_order\_seq\_cst 사용
      2. 이 방식에서 모든 스레드들은 똑같은 순서의 연산을 봐야하기 때문에 연산들이 재정렬될 수 없음
   2. lock/unlock -Peterson 알고리즘 사용
      1. 
   3. Peterson 알고리즘
      1. mutex exclusion을 구현한 알고리즘 중 하나
      2. flag변수와 turn 변수를 사용
      3. 
      4. 먼저 자신의 flag를 true로 변경-> 자신이 들어갈 의사가 있음을 밝힌다
      5. 이 후 상대에게 턴을 넘겨준다 (turn 변수가 없으면 서로 교착상태에 빠지는 경우가 발생 -> turn은 j또는i 중 하나이기 때문에 반드시 둘 중 하나만 진입 가능)
      6. 만약 상대가 들어갈 의사를 밝히고 상대의 턴이면(critical section에 있음을 의미) while문에서 기다림
      7. 턴이 자신에게 넘어오거나 상대가 들어갈 의지가 없다면 자신이 critical section에 진입 가능
      8. 작업을 마친 이후에는 flag를 내려 줌
2. **프로그램 구조 설명**
   1. 두 스레드를 생성하고 공유변수를 증가시키는 작업을 정해진 횟수만큼 반복한다
   2. 각 스레드는 공유변수 수정전에 lock 함수를 호출하고 작업이 끝나면 unlock을 호출
   3. 두 스레드의 일이 모두 끝날 때까지 부모에서 기다린다
   4. 모든 스레드가 일을 마치면 공유변수의 값과 예상한 값을 각각 출력한다.
3. **과제 B**
4. **자료구조 설명**
   1. shared memory
      1. 공유메모리
         1. 두개 이상의 프로세스가 특정한 메모리 영역을 공유하는 것을 말함
         2. 공유메모리 영역에 대한 쓰기와 읽기 작업을 통해 데이터 통신도 가능
      2. 순서
         1. shmget 함수로 공유메모리를 생성/불러오기
         2. shmat 함수로 자신의 메모리 공간에 공유메모리를 붙이고 그 주소 반환
         3. shmdt로 자신의 메모리 공간에서 공유메모리를 떼어 낸다
         4. shmct로 공유메모리를 제거한다.
      3.  공유해서 사용해야하는 구조체
5. **동기화 방법 설명**
   1. shared memory에서의 locking 방식을 사용
   2. lock과 unlock은 피터슨 알고리즘(과제 A 참고)을 이용하여 직접구현
      1. 
   3. critical section에 진입하기 전에 lock함수를 호출하고 작업이 모두 끝나면 unlock 함수를 호출
      1. 
6. **프로그램 구조 설명**
   1. parent
      1. 공유메모리를 필요한 사이즈(구조체 크기)만큼으로 생성한다.
      2. 그 공유메모리를 자신의 메모리 공간에 붙인 후 해당 공간의 변수 초기화
      3. 자식 프로세스를 2개 생성한 후 일을 끝마칠 때까지 기다린다
      4. 각 자식은 child 프로그램으로 exec되어서 일을 수행한다
      5. 모든 자식이 일을 끝마친 이후 실제 count 값과 예상한 값을 출력한다
      6. 프로그램 종료 전 공유메모리를 메모리 공간에서 떼고 공유메모리를 제거한다.
   2. child
      1. 부모에서와 마찬가지로 공유메모리를 먼저 생성/존재한다면 가져오기
      2. 공유메모리를 자신의 메모리공간에 붙인다.
      3. 자식들은 동일한 코드를 사용하기 때문에 각 프로세스마다 작업을 분리해서 하려면 구분 방법이 필요
      4. 부모에서 fork 사용 후 구조체의 변수 하나를 더 먼저 생성된 프로세스의 아이디로 설정해두고 만약 그 id와 해당 프로세스의 아이디가 동일하다면 먼저 생성된 프로세스이므로 0번 order을 부여
      5. 그게 아니라면 1번 order을 부여하고 Peterson 알고리즘에 인자로 order을 넘겨줌
      6. lock/unlock을 통해 동기화를 지키며 공유변수를 정해진 횟수만큼 증가시킨다
      7. 모든 작업이 끝나면 자신이 증가시킨 횟수(지역변수)를 출력한다
      8. 공유메모리를 떼어준다.
7. **추가 레포트**

**[과제A] – c++에서의 원자적 연산**

**1. 원자적 연산**

-C++11에서는 atomic을 이용하여 뮤텍스와 같이 lock, unlock을 사용하지 않고 값을 증가시키거나 값을 감소시킬 수 있는 기능을 제공

-더 자세한 정보는 과제 A의 내용 참고

**2. 메모리 순서(6 타입)**

**1)순차적으로 일관된 순서(memory\_order\_seq\_cst)**

-프로그램의 행동이 세상의 간단한 순차적 관점과 일치하기 때문에 가장 이해하기 쉬운 메모리 순서

-모든 스레드들은 똑같은 순서의 연산을 봐야하기 때문에 연산들이 재정렬될 수 없음

-모든 스레드들 간의 전역 동기화를 요구하므로 가장 비용이 비쌈

-멀티프로세스 시스템에서 프로세스 간에 꽤 많은 시간이 소모되는 통신이 될 수 있음

**2)획득-해제 순서(memory\_order\_consume, memory\_order\_acquire, memory\_order\_release, memory\_order\_acq\_rel)**

-비순차적 순서. 즉 동일한 연산에 대해 다른 스레드는 다른 순서를 볼 수 있음을 의미하며 컴파일러가 명령어를 재배치할 수 있음.

-완화된 순서와 순처적 순서의 중간단계

-release(해제-쓰기)는 해당 명령 이전의 모든 메모리 명령들이 해당 명령 이후로 재배치되는 것을 금지

-만약 같은 변수를 acquire(획득-읽기)로 읽는 스레드가 있다면 release 이전에 오는 모든 메모리 명령들이 해당 스레드에서 관찰될 수 있어야함

-memory\_order\_acquire의 경우, release 와는 반대로 해당 명령 뒤에 오는 모든 메모리 명령들이 해당 명령 위로 재배치되는 것을 금지

-memory\_order\_acq\_rel 은 acquire 와 release를 모두 수행. 읽기와 쓰기를 모두 수행하는 명령들, 예를 들어서 fetch\_add 와 같은 함수에서 사용

- C++ 17 현재, memory\_order\_consume의 정의가 수정 중에 있기 때문에 memory\_order\_consume의 사용이 권장되지 않음

**3)완화된 순서(memory\_order\_relaxed)**

-비순차적 일관성 순서

-단일 스레드 내의 같은 변수에 대한 연산은 happen-before 관계를 만족하지만 다른 스레드에 관련된 순서는 관여하지 않음

-같은 스레드로부터 단일 원자 변수에 대한 접근은 재배치되지 않는 다는 요구사항만 존재

-절대적으로 필요한 상황이 아니라면 사용하지 말자. 사용하게 되면 주의.

**3.잠금 없는 데이터 구조의 장점과 단점**

-뮤텍스를 쓰지 않는 잠금이 없는 구조=논블로킹

-장점: 동시성을 최대화할 수 있음. 견고함을 위해 사용.

-단점: 쓰기 동작이 잠금을 사용하는 것보다 훨씬 어려움. 데드락은 없는 대신 라이브락(live-lock)이 발생할 수 있음. 라이브락은 성능을 저하 가능성 존재

**4.다른 동기화 방법**

**1)mutex의 lock/unlock**

- 스레드가 이미 작업 중이거나 작업을 하다가 말았으면 다른 스레드들이 못 건드리게 잠궈둠

- 단점

unlock을 반드시 해줘야 다른 스레드가 작업을 수행할 수 있는데 프로그래머가 명시하는 것을 깜빡할 수도 있음

예외가 발생하거나 했을 때 unlock을 건너뛸 수도 있음

**2)lock\_guard**

- 스레드가 이미 작업 중이거나 작업을 하다가 말았으면 다른 스레드들이 못 건드리게 잠궈둠

- mutex의 lock 함수, unlock 함수와 기능은 동일

- lock/unlock 함수와 다르게 스마트 포인터처럼 자신의 영역을 벗어나면 자동으로 unlock()을 호출

**3) scope\_lock**

- 스레드가 이미 작업 중이거나 작업을 하다가 말았으면 다른 스레드들이 못 건드리게 잠궈둠

- mutex의 lock 함수, unlock 함수와 기능은 동일

- 스마트 포인터처럼 자신의 영역을 벗어나면 자동으로 unlock()을 호출

- C++ 17에서만 사용 가능하며 std::lock\_guard 함수보다 권장