Operating Systems Producer consumer problem

컴퓨터공학과

12121451

김수환

**< 목 차 >**

1. **개요**
2. **정의**
3. Semaphore
4. Mutex
5. **구현**
6. Producer Consumer Problem
7. **개발환경**
8. **개요**

2개의 Monitor Thread가 추가된 Producer Consumer Problem과제를 Semaphore와 Mutex를 이용하여 구현합니다. 이를 통해 Semaphore와 Mutex로 Synchronization하는 방법을 익히고, Critical Section Problem을 해결합니다.

1. **정의**
2. Semaphore

Semaphore란 두개의 Atomic한 함수로 제어되는 변수(정수)로 멀티 프로그래밍 환경에서 공유 변수에 대한 접근을 제어하는 방식입니다. 즉, 1개의 공유되는 변수에 대한 접근은 일정 개수의 Process 또는 Thread만 가능하도록 합니다.

[ 자료형 ]

* sem\_t

Semaphore의 Atomic한 함수로 제어되는 변수입니다.

[ 함수 ]

* sem\_init(sem\_t \*sem, int pshared, unsigned int value)

sem의 자원을 value값으로 초기화하고, 성공하면 0을 실패하면 -1을 반환합니다.

* sem\_wait(sem\_t \*sem)

sem의 자원을 1감소시킵니다. 만약 한 Process 또는 Thread가 sem\_wait을 호출하여 자원을 감소하고 0이 된다면 다른 Process 또는 Thread는 Block되고, 자원이 증가되기를 기다립니다.

* sem\_post(sem\_t \*sem)

sem의 자원을 1증가시킵니다.

* sem\_destroy(sem\_t \*sem)

sem의 자원을 해제합니다.

1. Mutex

Mutex란 Semaphore의 변수에 1의 자원을 할당한 것과 비슷합니다. 공유 변수에 접근하는, 즉 Critical Section에 접근하는 Process 또는 Thread를 오직 1개로 제한합니다.

[ 자료형 ]

* pthread\_mutex\_t

Semaphore의 Atomic한 함수로 제어되는 변수입니다.

[ 함수 ]

* pthread\_mutex\_init(pthread\_mutex\_t \*mutex, pthread\_mutex\_attr \*attr)

mutex변수를 초기화하고, 성공하면 0을 실패하면 -1을 반환합니다.

* pthread\_mutex\_lock(pthread\_mutex\_t \*mutex)

mutex변수에 lock을 걸어줍니다. 만약 mutex변수에 이미 lock이 걸려있는 상태라면 lock이 풀릴 때까지 호출한 thread는 Block되어있습니다.

* pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex)

mutex변수에 lock을 풀어줍니다.

* pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex)

mutex변수에 lock을 시도합니다. 만약 mutex변수에 이미 lock이 걸려있는 상태라면 EBUSY를 반환하고 함수를 빠져나옵니다. 그리고 mutex에 lock이 풀려있는 상태라면 lock을 걸어주고 0을 반환하고 함수를 빠져나옵니다.

* pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex)

mutex변수의 자원을 해제합니다.

1. **구현**
2. Producer Consumer Problem

요구 조건

* 첫번째 조건

item을 버퍼로 생산과 소비하는 과정에서 성공하면 0을 실패하면 -1을 반환합니다.

* 두번째 조건

Producer Monitor Thread는 생산자가 생산한 item을 1~50의 수 인지 확인합니다. 또한 Monitoring동안에는 Producer Thread는 Block되어있어야 합니다. 즉, insert하기 전에 Monitoring 되어야합니다.

* 세번째 조건

Consumer Monitor Thread는 소비자가 소비할 item을 1~25의 수면 그대로, 26~50의 수면 2로 나누어 소비할 것을 알립니다. 역시 Monitoring동안에는 Consumer Thread는 Block되어있어야 합니다.

사용한 변수 및 함수 설명

* typedef enum {false, ture} bool

C언어에서는 bool 자료형이 존재하지 않으므로 직접 정의.

* int front, rear, produceItem, consumeItem

front : buffer에서 소비할 item의 index

rear : 저장해야하는 buffer의 index

produceItem : 생산자가 생산한 item을 producerMonitoring()에서

Monitoring하기 위한 전역 변수

consumeItem : 소비자가 소비할 item을 condumerMonitoring()에서

Monitoring하기 위한 전역 변수

* bool producerFlag, consumerFlag

producerFlag : producerMonitoring()의 확인 결과를 저장할 변수

consumerFlag : condumerMonitoring()의 확인 결과를 저장할 변수

* int semWaitError, semPostError, mutexLockError, mutexUnlockError

sem\_wait(), sem\_post(), pthread\_mutex\_lock(), pthread\_mutex\_unlock()오류 유무를 저장할 변수

* sem\_t full, empty

full : buffer에 있는 item의 개수를 나타내는 semaphore변수입니다.

초기화는 0으로 합니다.

empty : buffer에 남은 공간의 개수를 나타내는 semaphore변수입니다.

초기화는 buffer의 size로 합니다.

* pthread\_mutex\_t mutex, conMutex, proMutex, proMoniMutex1, proMoniMutex2, conMoniMutex1, conMoniMutex2;

mutex : buffer(공유 자원)의 값을 Atomic하게 수정하도록 정의한 변수

conMutex, proMutex : 생산자와 소비자의 Mutual Exclusion을 위한 변수

proMoniMutex1, proMoniMutex2 : producer와 producerMonitoring() 의

Synchronization을 위한 변수

conMoniMutex1, conMoniMutex2 : consumer와 producerMonitoring() 의

Synchronization을 위한 변수

* void \*producerMonitoring()

생산자가 생산한 Item을 모니터링합니다.

* void \*consumerMonitoring()

소비자가 소비할 Item을 모니터링합니다.

* int insert\_item(buffer\_item \*item)

producerMonitoring에 의해 확인된 item을 버퍼에 삽입합니다.

* int remove\_item(buffer\_item \*item)

consumerMonitoring에 의해 확인된 item을 버퍼에서 가져옵니다.(출력)

* void \*producer(void \*param)
* void \*consumer(void \*param)

동작 원리

**(생산자, 소비자, 2개의 Monitoring Thread의 semaphore, mutex사용에 대해 기술)**

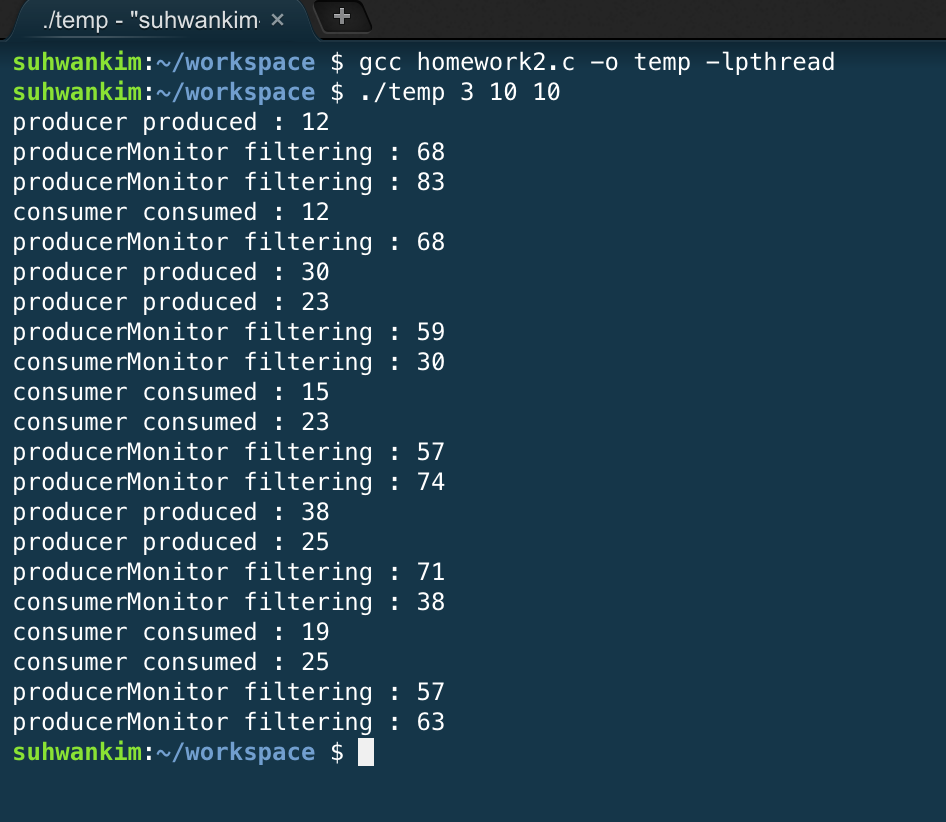
main()을 먼저 살펴보면 사용자로부터 프로그램의 구동시간, producer thread의 개수, consumer thread의 개수를 입력 받습니다. 그리고 크기가 10인 buffer를 동적 할당하고, semaphore와 mutex를 초기화합니다. 그 후 입력받은 producer, consumer의 개수만큼 thread를 생성합니다. 그리고 Monitoring을 담당하는 2개의 thread는 필요하기 전까지는 Block되어 있어야 하므로 먼저 proMoniMutex2와 conMoniMutex2에 lock을 걸어줍니다. 이후에 2개의 Monitor thread를 생성하고 sleep()을 호출하여 입력받은 프로그램의 구동시간 동안 기다립니다. 그리고 구동시간 이후에 사용한 semaphore와 mutex의 자원을 해제합니다.

main()에서 생성된 producer thread는 pthread\_mutex\_trylock(&proMoniMutex1)을 가장 처음에 호출하여 proMoniMutex1에 lock을 걸어줍니다. 이때 trylock()은 lock()과는 다르게 lock이 이미 걸려있는 상태라면 Block이 되지 않고 EBUSY를 반환한 후 함수를 탈출합니다. 그리고 1~3초의 시간 동안 sleep()하고, critical section에 접근할 준비를 마칩니다. 다음으로 pthread\_mutex\_lock(&proMutex)를 호출하여 proMutex에 lock을 걸어주고 critical section에 접근합니다. critical section에서는 다음과 같이 진행됩니다. 먼저 1~100사이의 정수를 생산하고 producerMonitoring()에서 확인하기 위해 생산한 item을 produceItem에 저장합니다. 그리고 Monitoring 하기 위해 pthread\_mutex\_unlock(&proMoniMutex2)를 호출하여 main()에서 걸어줬던lock을 풀어줍니다. 따라서 producerMonitoring()이 Block상태에서 해제됩니다. 그리고 바로 pthread\_mutex\_lock(&proMoniMutex1)을 호출하는데 이미 while문 처음에 trylock()을 이용하여 porMoniMutex1에 lock을 걸어줬으므로 producer thread는 Monitoring동안 Block되어있습니다. 따라서 **두번째 조건이 성립**합니다. 그리고 Monitoring의 과정을 살펴보면 먼저 삽입 여부를 저장할 변수 producerFlag를 true로 초기화합니다. 그 다음 조건문에서 생산한 item이 50을 초과할 경우 producerFalg를 false로 수정합니다. 이렇게 삽입 여부가 결정되었으므로 Block되어있던 producer thread를 재개하기 위해 pthread\_mutex\_unlock(&proMoniMutex1)을 호출하여 lock을 풀어줍니다. 그리고 다시 producerMonitoring()은 while문의 맨 처음으로 돌아가pthread\_mutex\_lock(&proMoniMutex2)를 호출하여 Block됩니다. 이로써 producer thread는 삽입 여부를 producerFlag를 통해 알 수 있습니다. 삽입이 가능하다면 insert\_item(&item)을 호출하여 buffer에 item을 저장하고 rear를 1증가 시킵니다. 그리고 만약 삽입이 정상적으로 이루어질 경우 0, 정삭적으로 이루어지지 않을 경우 -1을 반환합니다. 따라서 **첫번째 조건이 성립**합니다. 마지막으로 다른 producer thread가 접근할 수 있도록 proMutex의 lock을 풀어줍니다. 이러한 과정을 반복하며 producer thread가 동작합니다.

**main()에서 생성된 consumer thread 또한 앞서 설명한 producer thread와 동작 원리는 동일합니다.** 하지만 몇가지 다른 점이 있습니다. 따라서 consumer thread는 이 몇가지 다른 점에 대해서만 설명하겠습니다.

먼저 critical section에서 buffer[front]에 접근하여 소비할 item을 갖고 옵니다. 그리고 이 item에 대한 정제 여부를 확인하기 위해 consumerMonitoring()의 Block을 해제합니다. 그리고 Monitoring을 통해 정제가 필요하다고 확인되면 다시 해당 item위치의 buffer index에 2로 나누어 저장합니다. 그리고 remove\_item(&item)을 호출하여 item을 소비(출력)하고 buffer를 비우기 위해 0으로 수정합니다. 그리고 front를 1증가 시키고 remove\_item()을 빠져나옵니다. 앞서 언급하였듯이 이렇게 몇가지 다른 점을 제외하고는 producer thread와 동작 원리가 같으므로 역시 **세번째 조건이 성립**합니다.

위와 같은 과정을 통해 Producer Consumer Problem의 Synchronization을 구현하였습니다.

 결과

producer produced : 12

consumer consumed : 12

producer produced : 30

consumer consumed : 15

producer produced : 23

consumer consumed : 23

producer produced : 38

consumer consumed : 19

producer produced : 25

consumer consumed : 25

< 생산 소비 과정 확인 >

위의 그림을 통해 2개의 Monitoring Thread가 정상적으로 작동하는 것을 검증할 수 있습니다. 예를 들어 producer가 30을 삽입하고 consumer가 소비할 때 consumerMonitoring()이 작동하여 2로 나누어진 값인 15를 소비합니다. 그리고 producer가 1~50이 아닌 59의 숫자를 생성할 경우 producerMonitoring()이 작동하여 삽입하지 않을 것을 결정하여 buffer에 삽입되지 않습니다.

1. **개발 환경**

OS : Ubuntu 14.04.3

Compiler : gcc version 4.8.4

Language : C