Operating Systems

Producer consumer problem

컴퓨터공학과
12121451
김수환

< 목 차 >

1. 개요

2 개의 Monitor Thread 가 추가된 Producer Consumer Problem 과제를 Semaphore 와 Mutex 를 이용하여 구현합니다. 이를 통해 Semaphore 와 Mutex 로 Synchronization 하는 방법을 익히고, Critical Section Problem 을 해결합니다.

2. 정의

(1) Semaphore

Semaphore 란 두개의 Atomic 한 함수로 제어되는 변수(정수)로 멀티 프로그래밍 환경에서 공유 변수에 대한 접근을 제어하는 방식입니다. 즉, 1 개의 공유되는 변수에 대한 접근은 일정 개수의 Process 또는 Thread 만 가능하도록 합니다.

[자료형]

- sem_t

  Semaphore 의 Atomic 한 함수로 제어되는 변수입니다.

[함수]

- sem_init(sem_t *sem, int pshared, unsigned int value)

  sem 의 자원을 value 값으로 초기화하고, 성공하면 0 을 실패하면 -1 을 반환합니다.
- sem_wait(sem_t *sem)

  sem 의 자원을 1 감소시킵니다. 만약 한 Process 또는 Thread 가 sem_wait 을 호출하여 자원을 감소하고 0 이 된다면 다른 Process 또는 Thread 는 Block 되고, 자원이 증가되기를 기다립니다.
- sem_post(sem_t *sem)

  sem 의 자원을 1 증가시킵니다.
- sem_destroy(sem_t *sem)

  sem 의 자원을 해제합니다.

(2) Mutex

Mutex 란 Semaphore 의 변수에 1 의 자원을 할당한 것과 비슷합니다. 공유 변수에 접근하는, 즉 Critical Section 에 접근하는 Process 또는 Thread 를 오직 1 개로 제한합니다.

[자료형]

- pthread_mutex_t
  Semaphore 의 Atomic 한 함수로 제어되는 변수입니다.

[함수]

- pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutex_attr *attr)
  mutex 변수를 초기화하고, 성공하면 0 을 실패하면 -1 을 반환합니다.
- pthread_mutex_lock(pthread_mutex_t *mutex)
  mutex 변수에 lock 을 걸어줍니다. 만약 mutex 변수에 이미 lock 이 걸려있는 상태라면 lock 이 풀릴 때까지 호출한 thread 는 Block 되어있습니다.
- pthread_mutex_unlock(pthread_mutex_t *mutex)
  mutex 변수에 lock 을 풀어줍니다.
- pthread_mutex_trylock(pthread_mutex_t *mutex)
  mutex 변수에 lock 을 시도합니다. 만약 mutex 변수에 이미 lock 이 걸려있는 상태라면 EBUSY 를 반환하고 함수를 빠져나옵니다. 그리고 mutex 에 lock 이 풀려있는 상태라면 lock 을 걸어주고 0 을 반환하고 함수를 빠져나옵니다.
- pthread_mutex_destroy(pthread_mutex_t *mutex)
  mutex 변수의 자원을 해제합니다.

3. 구현

(1) Producer Consumer Problem

① 요구 조건

- 첫번째 조건
  item 을 버퍼로 생산과 소비하는 과정에서 성공하면 0 을 실패하면 -1 을 반환합니다.

- 두번째 조건

  Producer Monitor Thread 는 생산자가 생산한 item 을 1~50 의 수 인지 확인합니다. 또한 Monitoring 동안에는 Producer Thread 는 Block 되어있어야 합니다. 즉, insert 하기 전에 Monitoring 되어야합니다.

- 세번째 조건

  Consumer Monitor Thread 는 소비자가 소비할 item 을 1~25 의 수면 그대로, 26~50 의 수면 2 로 나누어 소비할 것을 알립니다. 역시 Monitoring 동안에는 Consumer Thread 는 Block 되어있어야 합니다.

② 사용한 변수 및 함수 설명

- typedef enum {false, ture} bool

  C 언어에서는 bool 자료형이 존재하지 않으므로 직접 정의.

- int front, rear, produceItem, consumeItem

  front : buffer 에서 소비할 item 의 index

  rear : 저장해야하는 buffer 의 index

  produceItem : 생산자가 생산한 item 을 producerMonitoring()에서 Monitoring 하기 위한 전역 변수

  consumeItem : 소비자가 소비할 item 을 condumerMonitoring()에서 Monitoring 하기 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 : buffer 에 있는 item 의 개수를 나타내는 semaphore 변수입니다. 초기화는 0 으로 합니다.

  empty : buffer 에 남은 공간의 개수를 나타내는 semaphore 변수입니다. 초기화는 buffer 의 size 로 합니다.

- pthread_mutex_t mutex, conMutex, proMutex, proMoniMutex1, proMoniMutex2, conMoniMutex1, conMoniMutex2;

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

  conMutex, proMutex : 생산자와 소비자의 Mutual Exclusion 을 위한 변수

  proMoniMutex1, proMoniMutex2 : producer 와 producerMonitoring() 의 Synchronization 을 위한 변수

  conMoniMutex1, conMoniMutex2 : consumer 와 producerMonitoring() 의 Synchronization 을 위한 변수

- void *producerMonitoring()
  생산자가 생산한 Item 을 모니터링합니다.
- void *consumerMonitoring()
  소비자가 소비할 Item 을 모니터링합니다.
- int insert_item(buffer_item *item)
  producerMonitoring 에 의해 확인된 item 을 버퍼에 삽입합니다.
- int remove_item(buffer_item *item)
  consumerMonitoring 에 의해 확인된 item 을 버퍼에서 가져옵니다.(출력)
- void *producer(void *param)
- void *consumer(void *param)

③ 동작 원리

(생산자, 소비자, 2 개의 Monitoring Thread 의 semaphore, mutex 사용에 대해 기술)

main()을 먼저 살펴보면 사용자로부터 프로그램의 구동시간, producer thread 의 개수, consumer thread 의 개수를 입력 받습니다. 그리고 크기가 10 인 buffer 를 동적 할당하고, semaphore 와 mutex 를 초기화합니다. 그 후 입력받은 producer, consumer 의 개수만큼 thread 를 생성합니다. 그리고 Monitoring 을 담당하는 2 개의 thread 는 필요하기 전까지는 Block 되어 있어야 하므로 먼저 proMoniMutex2 와 conMoniMutex2 에 lock 을 걸어줍니다. 이후에 2 개의 Monitor thread 를 생성하고 sleep()을 호출하여 입력받은 프로그램의 구동시간 동안 기다립니다. 그리고 구동시간 이후에 사용한 semaphore 와 mutex 의 자원을 해제합니다.

main()에서 생성된 producer thread 는 pthread_mutex_trylock(&proMoniMutex1)을 가장 처음에 호출하여 proMoniMutex1 에 lock 을 걸어줍니다. 이때 trylock()은 lock()과는 다르게 lock 이 이미 걸려있는 상태라면 Block 이 되지 않고 EBUSY 를 반환한 후 함수를 탈출합니다. 그리고 1~3 초의 시간 동안 sleep()하고, critical section 에 접근할 준비를 마칩니다. 다음으로 pthread_mutex_lock(&proMutex)를 호출하여 proMutex 에 lock 을 걸어주고 critical section 에 접근합니다. critical section 에서는 다음과 같이 진행됩니다. 먼저 1~100 사이의 정수를 생산하고 producerMonitoring()에서 확인하기 위해 생산한 item 을 produceItem 에 저장합니다. 그리고 Monitoring 하기 위해 pthread_mutex_unlock(&proMoniMutex2)를 호출하여 main()에서 걸어줬던 lock 을 풀어줍니다. 따라서 producerMonitoring()이 Block 상태에서 해제됩니다. 그리고 바로 pthread_mutex_lock(&proMoniMutex1)을 호출하는데 이미 while 문 처음에 trylock()을 이용하여 porMoniMutex1 에 lock 을 걸어줬으므로 producer thread 는 Monitoring 동안 Block 되어있습니다. 따라서 **두번째 조건이 성립**합니다. 그리고 Monitoring 의 과정을 살펴보면 먼저 삽입 여부를 저장할 변수 producerFlag 를 true 로 초기화합니다. 그 다음 조건문에서 생산한 item 이 50 을 초과할 경우 producerFalg 를 false 로 수정합니다. 이렇게 삽입 여부가 결정되었으므로 Block 되

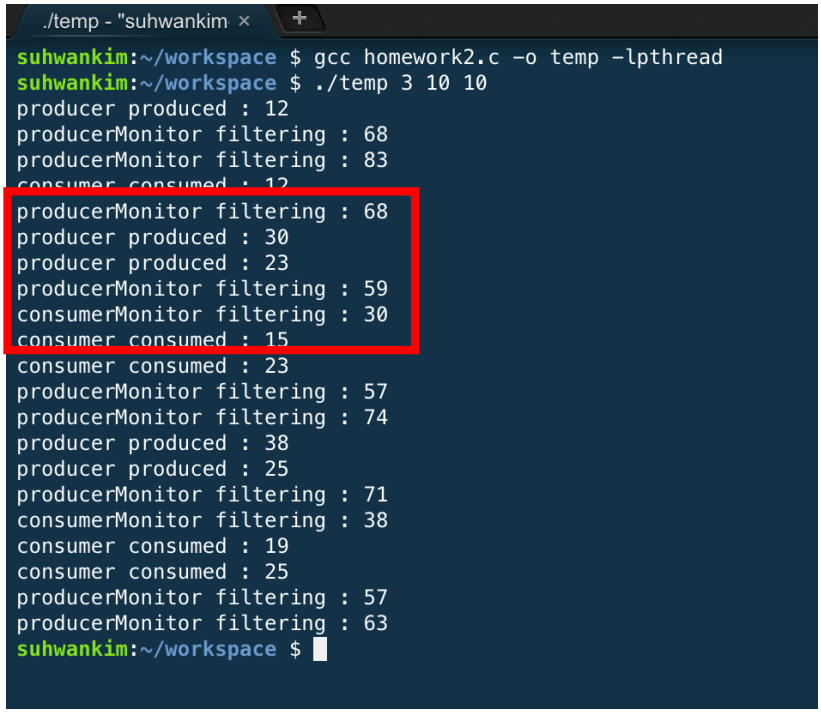
어있던 producer thread 를 재개하기 위해 pthread_mutex_unlock(&proMoniMutex1)을 호출하여 lock 을 풀어줍니다. 그리고 다시 producerMonitoring()은 while 문의 맨 처음으로 돌아가 pthread_mutex_lock(&proMoniMutex2)를 호출하여 Block 됩니다. 이로써 producer thread 는 삽입 여부를 producerFlag 를 통해 알 수 있습니다. 삽입이 가능하다면 insert_item(&item)을 호출하여 buffer 에 item 을 저장하고 rear 를 1 증가시킵니다. 그리고 만약 삽입이 정상적으로 이루어질 경우 0, 정삭적으로 이루어지지 않을 경우 -1 을 반환합니다. 따라서 **첫번째 조건이 성립**합니다. 마지막으로 다른 producer thread 가 접근할 수 있도록 proMutex 의 lock 을 풀어줍니다. 이러한 과정을 반복하며 producer thread 가 동작합니다.

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

먼저 critical section 에서 buffer[front]에 접근하여 소비할 item 을 갖고 옵니다. 그리고 이 item 에 대한 정제 여부를 확인하기 위해 consumerMonitoring()의 Block 을 해제합니다. 그리고 Monitoring 을 통해 정제가 필요하다고 확인되면 다시 해당 item 위치의 buffer index 에 2 로 나누어 저장합니다. 그리고 remove_item(&item)을 호출하여 item 을 소비(출력)하고 buffer 를 비우기 위해 0 으로 수정합니다. 그리고 front 를 1 증가 시키고 remove_item()을 빠져나옵니다. 앞서 언급하였듯이 이렇게 몇가지 다른 점을 제외하고는 producer thread 와 동작 원리가 같으므로 역시 **세번째 조건이 성립**합니다.

위와 같은 과정을 통해 Producer Consumer Problem 의 Synchronization 을 구현하였습니다.

④ 결과



< 생산 소비 과정 확인 >

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

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

4. 개발 환경

OS : Ubuntu 14.04.3

Compiler : gcc version 4.8.4

Language : C