RePORT

제 목 : Producer-consumer problem



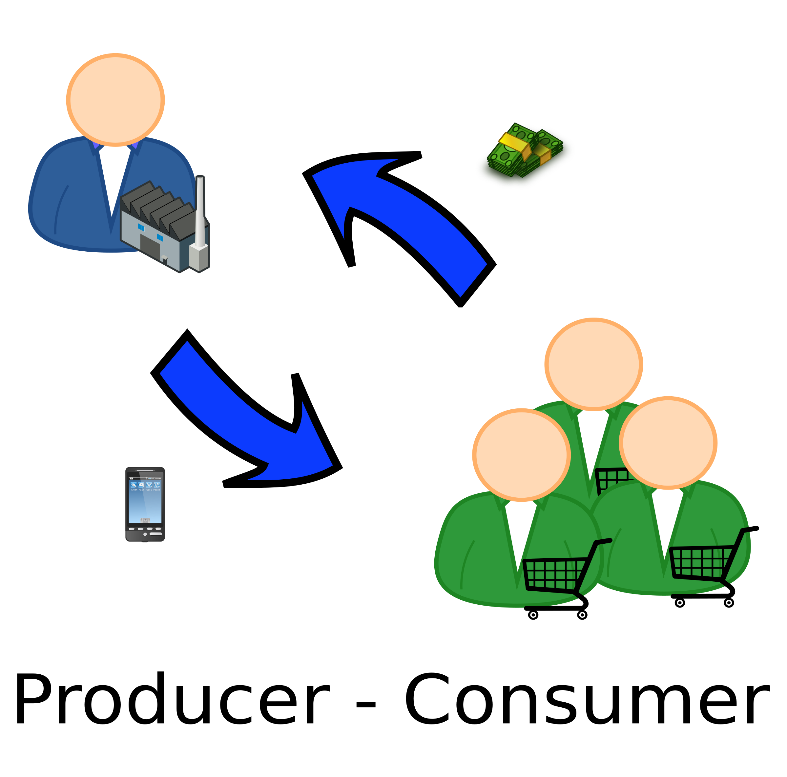
|  |  |  |
| --- | --- | --- |
| 과 목 명 | : | 오퍼레이팅시스템 |
| 학 과 | : | 컴퓨터정보공학과 |
| 학 번 | : | 12111453 |
| 담당교수 | : | 송민석 교수님 |
| 제 출 일 | : | 17년 05월 13일 |
| 성 명 | : | 김 우 진 |

1. 개요

- 목표

Producer-consumer 문제를 이해하고 binary semaphore인 mutex를 이용하여 해당 문제를 해결한다.

- Producer-consumer problem



<출처 : openclipart.org>

Producer-consumer problem이란 여러 개의 process를 어떻게 동기화 할 것인가에 대한 문제로 bounded-buffer problem이라고도 한다. 여러 명의 producer와 consumer 유한한 item을 저장할 수 있는 buffer를 공유할 때 발생하는 문제이다. Producer는 item을 계속해서 만들어서 buffer에 저장하는 역할을 하고, consumer는 지속적으로 item을 buffer에서 가져오는 역할을 한다. 이 때 문제는 producer는 buffer가 가득 찼을 때, item의 생산을 잠시 멈춰야 한다는 것이고, consumer의 경우는 buffer에 item이 없을 때 아무것도 가지고 갈 수 없다는 것이다.

2. 요구되는 개념

- Race condition

Race condition이란 실행 결과가 접근 순사에 따라 달라지는 것을 의미한다. 이러한 현상은 여러 process들이 같은 data를 concurrent하게 혹은 parallelism하게 접근하거나 조작할 때 발생한다. 이것을 해결하기 위해서는 data를 조작하는 부분을 atomic하게 구현하여야 한다. Preemptive kernel일 때 특히 race condition들이 발생할 가능성이 농후하다.

- Critical section

Critical section이란 공유변수를 접근하는 영역을 가리킨다. Critical section에서는 race condition과 같은 문제가 발생할 수 있는데, 이러한 문제를 해결하기 위해서는 mutual exclusion, progress, bounded waiting을 만족하여야 한다. Mutual exclusion이란 하나의 process만 critical section에 접근이 가능하다는 것을 의미한다. Progress는 하나의 process가 critical section에서 작업을 마치면 다음 process가 해당 구역에 들어가야 한다는 것을 뜻한다. Bounded waiting은 critical section의 접근을 기다리는 순서의 bound가 존재해야 한다는 것이다.

- Deadlock

해당 프로그램을 잘못 작성 하였을 경우에 deadlock 문제가 발생할 수 있기 때문에, 개념을 다시한번 살펴보았다. Deadlock이란 두개 혹은 그 이상의 process들이 무기한으로 event를 기다리는 것으로, 대표적인 예로는 Dining-Philosophers problem이 있다. 이러한 문제의 해결 방법으로는 deadlock이 발생할 것 같을 시에 잠시 프로그램의 수행을 중지하는 deadlock avoidance, deadlock의 필요조건인 mutual exclusion, hold and wait, no preemption, circular wait 중 하나를 발생하지 않도록 하는 deadlock prevention, 그리고 deadlock이 발생된 후에 deadlock을 detection하여 process를 roll back하거나 kill하는 방식이 있다.

- Queue

Wikipedia의 producer-consumer problem의 정의에 fixed-size buffer used as a queue라고 나타나 있는 것을 보고, 처음에 remove\_item의 stack으로 구현하여 LIFO로 된 것을 다시 queue로 구현하여 FIFO가 되게끔 하였다. Queue의 개념을 다시 한번 살펴 보았다. Queue란 선형 자료구조인 배열 혹은 list등으로 구현이 가능하며, 먼저 넣은 data가 먼저 나오는 FIFO(First In First Out)의 구조로 저장되는 자료구조이다. Queue의 ADT로는 create, destroy, enqueue, deque, isEmpty등이 있다.

- 필요함수

(열혈 TCP/IP 소켓 프로그래밍 참조)

int pthread\_mutex\_init(phtread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*attr);

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

- return : 성공 시 0, 실패 시 0 이외의 값 반환

- mutex : mutex 생성시에는 mutex의 참조 값 저장을 위한 변수의 주소 값 전달, 그리고 mutex 소멸 시에는 소멸하고자 하는 mutex의 참조 값을 저장하고 있는 변수의 주소 값 전달.

- attr : 생성하는 mutex의 특성정보를 담고 있는 변수의 주소 값 전달, 별도의 특성을 지정하지 않을 경우에는 NULL 전달.

int phread\_mutex\_lock(phtread\_mutex\_t \*mutex);

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

- return : 성공 시 0, 실패 시 0 이외의 값 반환

- mutex : mutex 생성시에는 mutex의 참조 값 저장을 위한 변수의 주소 값 전달, 그리고 mutex 소멸 시에는 소멸하고자 하는 mutex의 참조 값을 저장하고 있는 변수의 주소 값 전달.

3. 코드설명

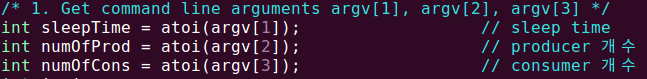
Version 1.

- item을 monitoring할 때마다 monitoring thread를 생성

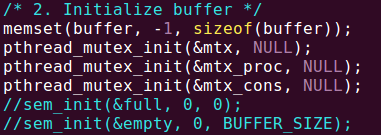
1) main코드 설명

- 주석에 알맞게 코드를 채워 넣었다.

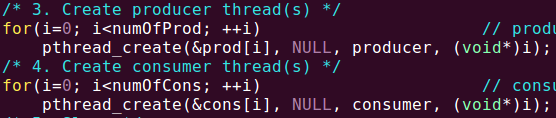
(1) command line의 argument 이용

 Sleep time의 크기를 받는 sleepTime함수와 producer의 개수를 받는 numOfProd, consumer의 개수를 받는 numOfCons를 선언과 동시에 초기화 하였다.

(2) buffer와 각종 변수들 초기화

 Buffer를 memset을 이용하여 초기화를 해주었다. 이 때 buffer가 0의 값을 가질 수 있기 때문에 -1로 초기화를 하였다. mutex로 사용이 되는 변수들도 함께 초기화를 해주었다.

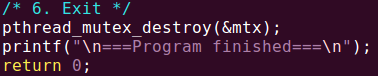
(3) producer와 consumer thread 생성

 앞에서 받은 producer의 개수와 consumer의 개수를 이용하여, 필요한 만큼의 producer thread와 consumer thread들을 생성하였다.

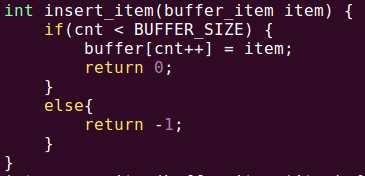
(4) main thread sleep

C:\Users\wooji\AppData\Local\Microsoft\Windows\INetCache\Content.Word\4.png앞서 받은 sleep시간 동안에 producer thread와 consumer thread들이 연산을 하기 위하여 main thread를 sleep시켜두었다.

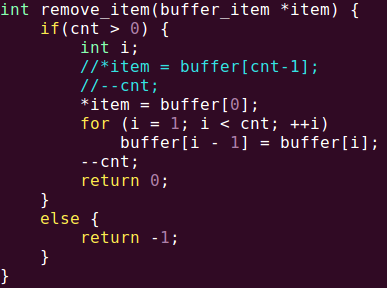
(5) mutex사용 종료 및 프로그램 종료

올바른 coding 습관을 위하여 생성하였던 mutex 변수들을 더 이상 사용하지 않는다는 것을 알리기 위하여 pthread\_mutex\_destroy()함수를 사용하였다.

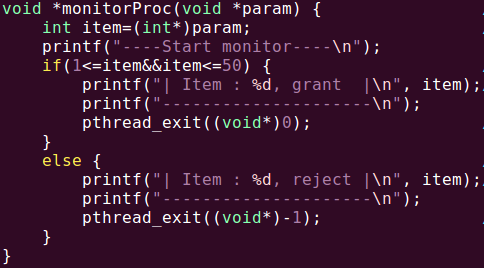
2) Buffer에 item 넣기 위한 함수

Insert\_item()함수는 buffer에 item을 넣기 위하여 만든 함수로, 만약 현재 buffer에 있는 item의 개수인 cnt의 값이 item 개수의 최대 크기인BUFFER\_SIZE보다 작으면 item을 집어넣고 0을 반환하지만 BUFFER\_SIZE보다 클 경우에 값을 집어 넣지 못하게 하고 -1을 반환하도록 하였다.

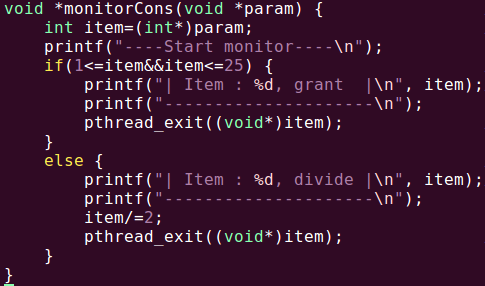
3) Buffer에서 item 가져오기 위한 함수

remove\_item()함수는 buffer에서 item을 가져오기 위한 함수로 item이 buffer에 하나라도 있을 시에 가져올 수 있으나 그렇지 않을 경우에 가져오지 못하므로 -1을 반환하도록 구현하였다. 처음 구현을 하였을 때 stack방식으로 가장 최근에 들어온 item을 먼저 가져오도록 작성 하였으나, queue방식으로 처음 들어온 item을 먼저 가져가도록 다시 구현하였다.

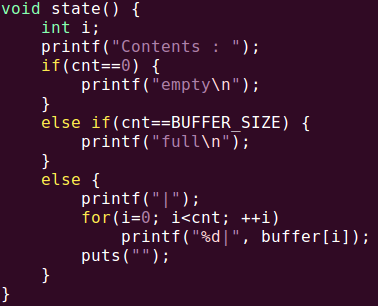
4) producer를 monitoring 하기위한 함수

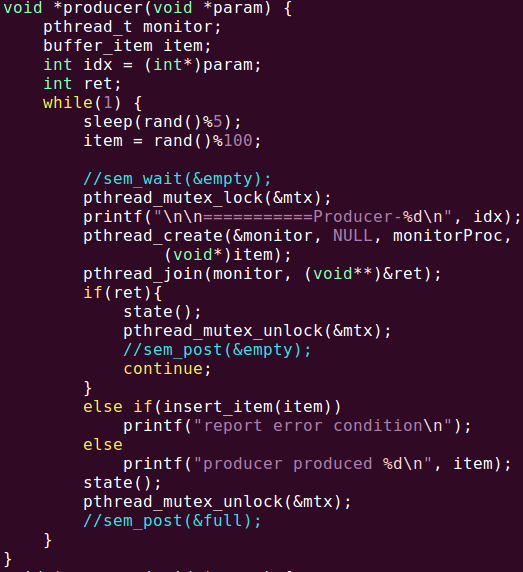
 monitorProc()함수는 producer thread들을 monitoring 하기 위한 함수로 item이 1보다 크거나 같거나 50보다 작거나 같을 경우에만 buffer에 넣을 수 있도록 한다. 만약 조건을 만족할 시에 item의 값과 승인을 출력하였고 0을 반환하여 buffer에 값을 저장할 수 있도록 하였. 조건을 만족하지 않을 시에 item의 값과 거절을 출력하고, buffer에 item을 넣지 못하도록 -1을 반환하였다.

5) consumer를 monitoring 하기위한 함수

 monitorCons()함수는 consumer thread들을 monitoring 하기위한 함수로 해당 item의 값의 범위가 1보다 크거나 같고 25보다 작거나 같을 경우 그대로 해당 item을 반환해주고 그렇지 않을 경우 item을 2로 나눈 값을 반환하도록 구현하였다. 즉, consumer가 1~25 사이의 값이 아닌 값을 buffer에서 가져왔을 경우 2로 나눈 값으로 가져가게 하기 위함이다.

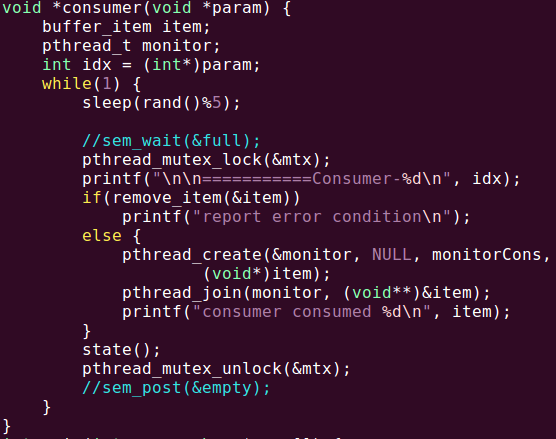
6) 현재 buffer의 상태를 출력

 state()함수는 현재 buffer에 들어있는 item들을 출력하기 위해서 만든 함수로 만약 buffer에 아무것도 없을 시에 empty를 출력하게 하였고, buffer가 가득 찼을 경우에는 full을 출력하도록 하였다. Buffer가 empty이거나 full이 아닌 경우에는 buffer에 들어온 item을 먼저 들어온 순서대로 출력하게 끔 작성하였다.

7) producer 함수

producer()함수는 producer thread들을 생성하기 위하여 작성된 함수로 각 thread마다 각기 다른 0~4초간의 시간동안 sleep을 해준 뒤에 0~99의 특정 값을 갖는 item을 생성하도록 하였다. 그 후 공유변수인 buffer에 접근하기 때문에 race condition의 발생을 막기 위하여 pthread\_mutex\_lock()으로 하나의 thread만을 critical section에 접근할 수 있게끔 구현을 하였다. Critical section에 접근한 후에 앞서 생성한 item의 값이 적절한지를 확인하기 위하여 monitoring을 실시하였다. 이 때 monitoring의 조건에 “Meanwhile, the producer thread must be blocked.”라고 되어있었기 때문에 pthread\_join을 통하여 monitoring이 실시되는 thread의 종료를 기다리도록 하였다. 그 후 monitoring의 결과 값을 토대로 해당 값이 적절하다고 판단 되었을 시에 buffer에 값을 저장하고 그렇지 않을 경우에는 다시 새로운 item을 생성하도록 하였다. 값이 적절하다고 판단이 되었을 경우에도, 만약 buffer가 가득 찼다면 “report error condition”을 출력하여 값을 집어 넣을 수 없다는 것을 출력하였다.

8) consumer 함수

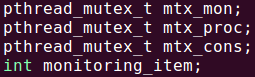
 consumer()함수는 consumer thread들을 생성하기 위하여 작성된 함수로 각 thread마다 각기 다른 0~4초간의 시간동안 sleep을 해준 뒤에, critical section에 들어가도록 하였다. 이 때 해당 buffer에 아무 값도 들어 있지 않다면 “report error condition”을 출력하고 만약 값이 있다면 해당 값이 조건에 맞는지 확인을 하기 위하여 monitoring을 한다. 이 때 monitoring의 조건인 “Meanwhile, the consumer thread must be blocked.”를 만족 시키기 위하여 pthread\_join()으로 monitoring thread의 종료를 기다려서, 해당 monitoring의 결과로 얻은 조건에 맞는 item을 consumer가 소비하였다고 출력하였다.

Version 2.

- 처음에 생성한 producer monitor thread와 consumer monitor thread로 주어진 item들을 검사

- Version 1과의 차이점을 기준으로 설명

1) 전역변수

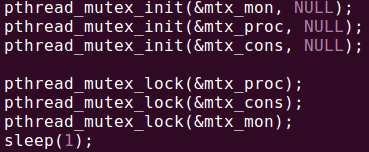
추가적인 동기화가 필요하기 때문에 전역변수로, 3개의 mutex변수를 선언 하였다. mtx\_mon은 monitoring을 수행동안 producer 혹은 consumer thread를 block하기 위해 필요하다. mtx\_proc은 producer monitoring thread가 item을 검사하지 않을 때 block하기 위해 생성하였고, mtx\_cons는 consumer monitoring thread가 item을 검사하지 않을 때 block하기 위해서 생성하였다. monitoring\_item은 monitoring할 값을 전달 받고, 결과를 보내주는 용도로 사용되는 공유 변수이다.

2) main코드 설명

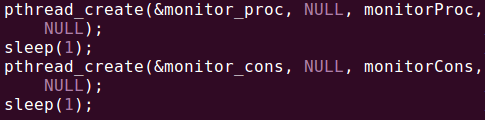
(1) pthread\_t 변수 추가

C:\Users\Woojinee\AppData\Local\Microsoft\Windows\INetCache\Content.Word\1.pngmain에서 monitoring thread를 생성하기 위하여 pthread\_t 변수 두개를 추가 생성하였다.

(2) 추가적인 mutex변수 초기화 및 처리

 추가적인 동기화로 인해 나타난 mutex변수들을 초기화하는 부분이 추가되었고, 처음 producer와 consumer monitoring thread가 생성되었을 때, block상태로 두기 위하여 mtx\_proc과 mtx\_cons에 대해 pthread\_mutex\_lock을 각각 호출 하였다. 또 producer와 consumer thread 역시 monitoring을 할 때 바로 block이 되기 위하여 mtx\_mon에 대해서 pthread\_mutex\_lock을 호출하였다. sleep을 사용한 이유는 thread 생성 전에 mutex값이 변하는 것을 확실시 하기 위해서이다.

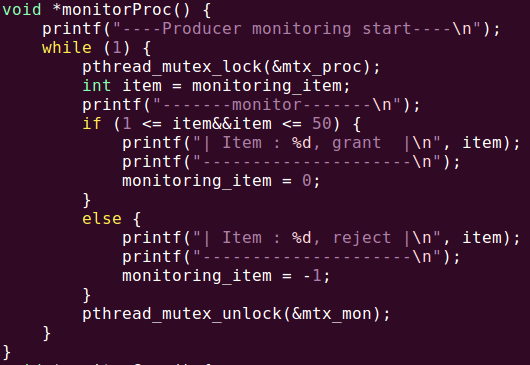
(3) main에서 monitoring thread 생성

해당 시점에서 parameter를 사용할 필요가 없기 때문에, parameter 전달 없이 thread를 생성하였다. sleep을 사용한 이유는 producer thread가 consumer thread보다 먼저 생성되는 것을 확실시 하기 위해서이다.

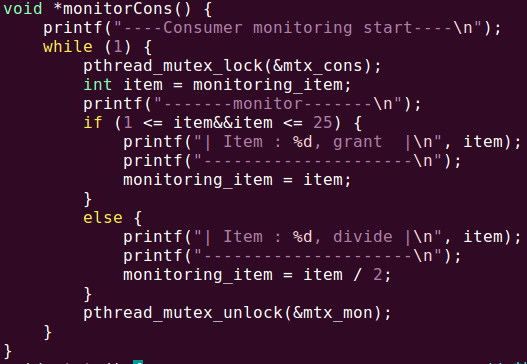
(4) mutex사용 종료 및 프로그램 종료

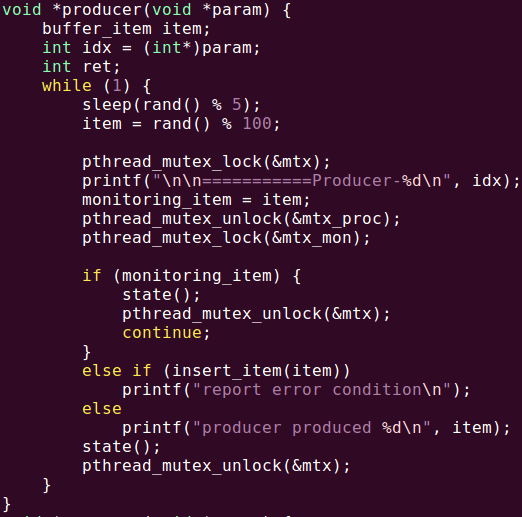
C:\Users\Woojinee\AppData\Local\Microsoft\Windows\INetCache\Content.Word\4.png동기화를 위한 mutex가 더 추가되었기 때문에 mutex사용 종료를 위한 pthread\_mutex\_destroy 역시 추가되었다.

3) producer를 monitoring 하기위한 함수

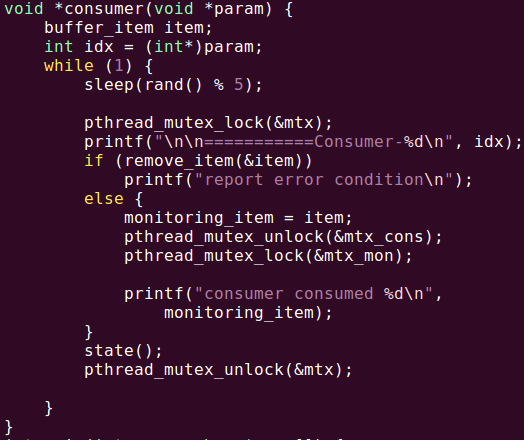
monitorProc은 version 1에서 설명했듯이 producer thread에서 생산한 item이 조건에 맞는지 확인하기 위하여 만들어진 함수로 version 2에서는 처음 main에서 호출이 되었을 때, producer monitoring이 시작되었음을 알리고, producer에서 item을 생성하고 검사를 받기위해 monitoring\_item에 값을 넣을 때까지 pthread\_mutex\_lock(&mtx\_proc)에 의하여 block 상태에 놓인다. Monitoring한 item이 조건에 맞지 않는다면 -1을 producer에 전달하고, 조건에 맞다면 0을 전달해 준다. Monitoring 중일 때 producer는 block 상태이기 때문에 가장 아랫줄에 위치한 pthread\_mutex\_unlock(&mtx\_mon)을 통하여 producer가 다시 동작하도록 한다. Monitoring은 while(1)로 계속 이러한 동작을 실행하지만, pthread\_mutex\_lock(&mtx\_proc)으로 인해 다시 block 상태에 놓인다.

4) consumer를 monitoring 하기위한 함수

 monitorCons()는 version 1에서 설명하였듯이, consumer thread에서 받은 item이 조건을 만족하는지를 판단하기 위해 만들어진 함수로, version 2에서는 처음 main에서 호출이 되었을 때 consumer monitoring이 시작되었음을 알리고, consumer에서 받은 item을 검사할 때까지 pthread\_mutex\_lock(&mtx\_cons)로 인하여 block이 된다. Monitoring 결과 해당 item이 조건에 맞다면 item 값을 그대도 전달해주고, 조건에 맞지 않다면 item 값을 반으로 나눠서 전달해준다. 이 때, monitoring동안 consumer thread가 block 되기 때문에, pthread\_mutex\_unlock(&mtx\_mon)을 통하여 consumer가 다시 동작하도록 한다. 이후 이 thread는 pthread\_mutex\_lock(&mtx\_cons)로 인해 다음 검사를 실행할 때까지 block 상태에 놓이게 된다.

5) producer 함수

version 1의 producer 함수와 가장 큰 차이점은 monitoring을 통하여 item을 검사 받는 방법에 있다. Version 2에서는 item을 생성한 후 pthread\_mutex\_lock(&mtx)를 통하여 다른 producer나 consumer가 접근을 못하게 하고, item을 monitoring하기 위해 존재하는 변수에 담은 후에, monitoring을 하는 thread를 pthread\_mutex\_unlock(&mtx\_proc)을 통하여 동작되도록 하였다. pthread\_mutex\_lock(&mtx\_mon)으로 자신을 block 시킨다. 그 후 monitoring결과 값이 나오면 다시 동작하게 되며 그 결과를 통해 얻어진 monitoring\_item의 값으로 item을 어떻게 처리할지를 결정한다. 끝으로 다른 producer와 consumer가 item을 처리할 수 있게 pthread\_mutex\_unlock(&mtx)를 호출한다.

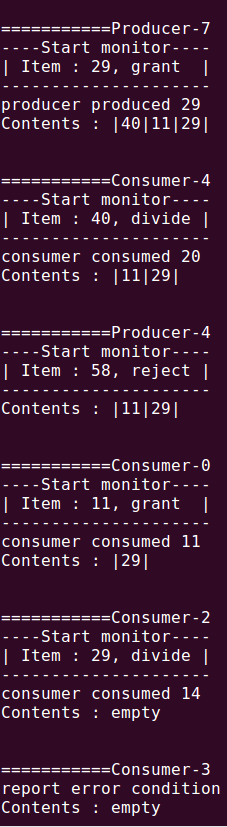
6) consumer 함수

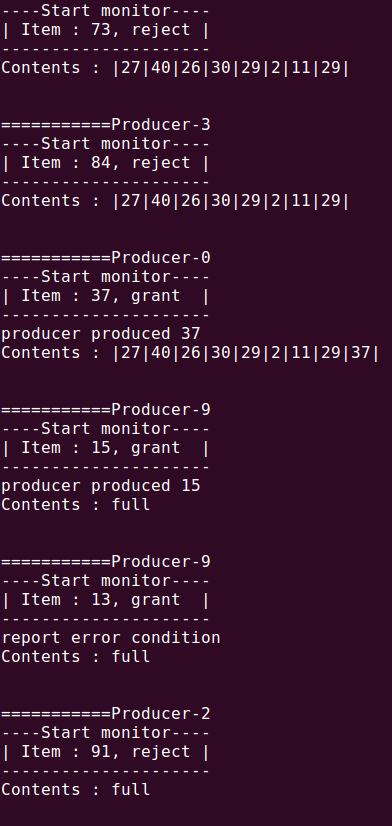
Version 1의 consumer 함수와의 차이점은 앞선 producer 함수와 마찬가지로 monitoring을 통하여 item을 검사 받는 방법에 있다. 변화가 else절에서 나타나는데, 이 때도 역시 monitoring\_item에 검사 받을 item을 전달하고, pthread\_mutex\_unlock(&mtx\_cons)를 통해서 monitoring이 동작되도록 하고 ptrhead\_mutex\_lock(&mtx\_mon)을 통해서 자신을 block한다. 이후 monitoring으로부터 결과를 전달받으면 다시 진행하며, monitoring으로부터 받은 변화된 혹은 그대로인 값을 소비하게 된다. 끝으로 다른 producer와 consumer가 item을 처리할 수 있게 pthread\_mutex\_unlock(&mtx)를 호출한다.

4. 실행화면

Version 1.

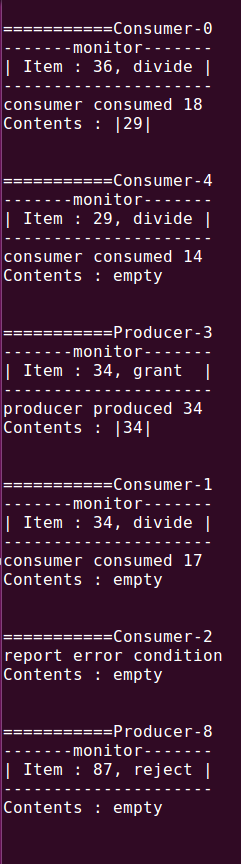
첫번째 실행결과는 producer 10개와 consumer 5개로 이루어진 출력 결과로 producer가 생산한 값이 monitoring에 의해여 적절한 값이라고 판단 된다면 grant가 되며 값의 저장이 이루어지고, 그렇지 않다면 reject가 되어 저장이 이루어지지 않는다는 것을 보여준다. Consumer의 경우 monitoring에 의하여 적절한 값이라고 판단이 된다면 grant되고, 그렇지 않다면 divide를 출력하면서 2로 나누어진 값이 소비되는 것을 보여주었다. 그리고 만약 buffer가 비어 있다면 “report error condition”을 출력하면서 contents가 empty임을 보여준다.

 두번째 실행결과는 producer 10개와 consumer 0개로 이루어진 출력결과로 monitoring에 의해 item이 grant됐지만, buffer가 가득 찼을 경우 “report error condition”을 출력하는 것을 보여준다.



C:\Users\Woojinee\AppData\Local\Microsoft\Windows\INetCache\Content.Word\실행결과_3.pngVersion 2.

version 1과의 가장 큰 차이점은 main에서 처음 monitoring을 생성하였을 때, 그것을 표시하기 위하여 monitoring이 시작되었음을 알리는 것이다.

아래의 두 실행결과는 version 1과 같은 조건에서 프로그램을 실행한 화면으로 첫번째 화면은 producer 10개와 consumer 5개로 이루어진 출력 결과이고, 두번째 화면은 producer 10개와 consumer 0개로 이루어진 출력결과이다. Monitoring에 의한 동작들이 같다는 것을 알 수 있다. Monitor가 먼저 미리 시작되었기 때문에 version 1의 start monitor 부분을 monitor로 출력 되도록 변화를 주었다.

5. 결과

수업 시간에 배운 producer-consumer 문제를 직접 구현하는 시간을 갖게 되어 매우 유익하였다. 구현을 하다 보니 세련되게 작성하고 싶은 마음이 커져서 계속해서 코드를 다듬게 되었다. 완성하고 나서는 조금이라도 결점이 없는지 찾아보았는데, 그러던 도중 앞서 구현에 필요한 개념에서 언급하였지만 producer-consumer 문제는 fixed-size buffer used as a queue를 사용한다는 것을 확인하였고, 그에 따라 기존의 stack방식이었던 코드를 바꾸게 되었다.

처음 문제의 조건에 있는 monitoring 실행 시에 producer나 consumer를 block시키라는 부분을 보고, monitoring thread를 만드는 부분을 mutex로 둘러주었다. 다시 한번 코드를 살피던 도중 pthread\_mutex\_unlock(&mtx)가 있기 때문에 불필요한 부분이라고 판단이 되어 지우게 되었다.

monitoring thread를 item을 검사할 때마다 생성하는 것이 아니라, producer와 consumer의 검사 용도로 각각 하나씩 생성하고, 프로그램 종료 시까지 동작하면서 item을 검사하는 방법도 있다는 것을 알게 되었다. 그에 따라 version 2를 작성하였고, thread 동작 방식에 대해서 좀 더 깊이 있는 이해를 할 수 있게 되었다.

강의자료의 경우 empty semaphore와 fill semaphore로 producer-consumer문제를 구현해본 부분이 있어서, 이러한 구현 방식으로 작성해 보았으나, 이러한 방식으로 구현을 하였을 시에 producer와 consumer 부분의 skeleton 코드에 있는 “report error condition”의 출력을 적절하게 이용하기가 어려워서 다른 방식을 이용하게 되었다.