OS 2차과제

(Producer Consumer Problem 모니터링)

12114497 컴퓨터공학과 한정

Producer Consumer Problem 해결

주로 사용되는 함수

```
pthread_mutex_init(&mutex,NULL);
/*
 * 함수명 : pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mute
x_attr *attr);
 * 설명 : mutex 값을 초기화 합니다. NULL을 넣을 경우 binary mutex를
 *      사용하게 됩니다.
 */
pthread_mutex_lock(&mutex);
/*
 * 함수명 : pthread_mutex_lock(pthread_mutex_t *mutex);
 * 설명 : 해당 mutex를 잠궈주게 됩니다. mutex값이 0 이하일 경우 wait상태가
 *      되게 됩니다.
 */
pthread_mutex_unlock(&mutex);
/*
 * 함수명 : pthread_mutex_unlock(pthread_mutex_t *mutex);
 * 설명 : mutex 값을 증가시킵니다.
 */
sem_init(&sema, 0, 5);
/*
 * 함수명 : sem_init(sem_t *sem, int pshared, unsigned int value)
 * 설명 : semaphore 값을 초기화 합니다.
 *      pshared값이 0일 경우 해당 프로세스 내에서만 사용합니다.
 *      value 값은 semaphore값의 초기값을 설정 해 줍니다.
 */
sem_post(&sema);
/*
 * 함수명 : sem_post(sem_t *sem)
 * 설명 : semaphore 값을 증가시켜 줍니다.
 */
sem_wait(&sema);
/*
 * 함수명 : sem_wait(sem_t *sem)
 * 설명 : semaphore 값을 감소 시킵니다.
 *      semaphore 값이 0이하일 경우 증가 할 때까지 기다리게 됩니다.
 */
```

내용

- Producer Consumer Problem 문제 설명
  1. 조건
     - Producer 쓰레드는 아이템을 생산하여 buffer에 넣어준다.
     - Consumer 쓰레드는 아이템을 buffer에서 소비한다.
     - 각 쓰레드 들은 buffer에 동시에 접근이 되어서는 안된다.
  2. 모니터링
     - Producer Monitoring

1. 1~100 사이의 숫자를 생성한다.
2. 하지만 1~50 사이의 숫자만을 buffer에 넣어준다. 51~100에 해당하는 숫자는 reject한다.

- Consumer Monitoring
  1. 버퍼 내의 숫자 중 1~25 사이의 숫자를 소비한다.
  2. 26~50 사이의 숫자는 2로 나누어 나머지를 버린 값을 소비한다.

결과화면

```
hanjungv:~/workspace $ gcc procon.c -o procon -lpthread
hanjungv:~/workspace $ ./procon 10 10 10
produce : 12
생성모니터링 : 68
consume : 12
생성모니터링 : 63
생성모니터링 : 68
produce : 3
consume : 3
생성모니터링 : 94
produce : 12
consume : 12
생성모니터링 : 85
생성모니터링 : 99
produce : 14
생성모니터링 : 92
생성모니터링 : 57
consume : 14
생성모니터링 : 85
produce : 6
consume : 6
produce : 30
소비모니터링 : 30
consume : 15
produce : 46
생성모니터링 : 68
produce : 35
생성모니터링 : 51
소비모니터링 : 46
consume : 23
생성모니터링 : 77
소비모니터링 : 35
consume : 17
생성모니터링 : 55
생성모니터링 : 61
생성모니터링 : 69
생성모니터링 : 87
생성모니터링 : 71
```

```
* 생성, 소비모니터링, 소비 결과 검증
produce : 12
consume : 12
--
produce : 3
consume : 3
--
produce : 12
consume : 12
--
produce : 14
consume : 14
--
produce : 6
consume : 6
--
produce : 30
소비모니터링 : 30
consume : 15
--
produce : 46
소비모니터링 : 46
consume : 23
--
produce : 35
소비모니터링 : 35
consume : 17

* 생성 모니터링에 걸린 case
생성모니터링 : 68
생성모니터링 : 63
생성모니터링 : 68
생성모니터링 : 94
생성모니터링 : 85
생성모니터링 : 99
생성모니터링 : 92
생성모니터링 : 57
생성모니터링 : 85
생성모니터링 : 68
생성모니터링 : 51
생성모니터링 : 77
생성모니터링 : 55
생성모니터링 : 61
생성모니터링 : 69
생성모니터링 : 87
생성모니터링 : 71
```

- 50 이상의 숫자가 생성된 경우 모니터링에서 reject가 되는 것을 확인했습니다.
- consume을 할 때 26~50 일 경우 2로 나누어 consume을 합니다.

코드 및 설명

```c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <stdlib.h>
#include <semaphore.h>
/*
 * 강의 자료 내의 buffer.h 내용을 한 파일로 합쳤습니다.
 */
typedef int buffer_item;
#define BUFFER_SIZE 10
/*
 * 변수 선언부분입니다.
 * front, rear : buffer를 환형 배열로 사용하기 위해 선언했습니다.
 * i : 반복문을 사용할 때 사용합니다.
 * isProduceItemFair, isConsumeItemFair : 해당 값을 모니터링에서
 *      판단하여 값을 변경하여 줍니다. 1일 경우 모니터링을 통과
 *      한 것이고 -1일 경우 모니터링에 걸린 것입니다.
 * producerCheckItem, consumerCheckItem : 이 값을 기준으로 모니터링을
 *      하게 됩니다.
 * mutex, empty, full : Producer, Consumer가 진입을 할 수 있게 사용하는
 *      mutex값과 semaphore값입니다. 생산이 될 경우 empty값이 1감소하고
 *      full값이 1 증가하게 됩니다.
 */
buffer_item *buffer;
int front = 0, rear = 0, i = 0;
int isProduceItemFair = 0, isConsumeItemFair = 0;
int producerCheckItem = 0, consumerCheckItem = 0;
pthread_mutex_t mutex;
sem_t empty, full;
/*
 *  mu1, mu2, mu3, mu4, conmu, promu : 새로 사용되는 mutex입니다.
 *      mu1, mu2는 producer가 모니터링을 할 때 block하는 용도로,
 *      mu3, mu4는 consumer가 모니터링을 할 때 block하는 용도로
 *      사용되게 됩니다.
 *      conmu와 promu는 consumer와 promu는 하나의 변수를 생성하고
 *      공유변수를 접근할 때 가리키는 방향이 달라질 수 있어 mutex값을
 *      이용하여 lock을 걸게 됩니다.
 */
pthread_mutex_t mu1, mu2, mu3, mu4, conmu, promu;
/*
 * 함수명 : void *producerMonitoring()
 * 설명 : 먼저 mu2가 lock이 되어있던 것이 풀리게 되면 모니터링이
 *      시작되게 됩니다. 전역 변수인 isProduceItemFair가 1일 경우
 *      monitoring에 걸리지 않는 아이템입니다. -1일 경우 50보다 큰
 *      값이 생성 된 것이므로 Insert가 이뤄지면 안됩니다.
 *      모니터링이 끝날 경우 mu1을 풀어주어 Producer에서
 *      다음 함수를 진행할 수 있게 해 줍니다.
 */
void *producerMonitoring(){
    while(1){
        pthread_mutex_lock(&mu2);
        isProduceItemFair = 1;
        if (producerCheckItem > 50){
            printf("생성모니터링 : %d\n", producerCheckItem);
```

```
            isProduceItemFair = -1;
        }
        pthread_mutex_unlock(&mu1);
    }
}
/*
 * 함수명 : void *consumerMonitoring()
 * 설명 : mu4의 lock이 풀리게 될 경우 모니터링을 시작합니다.
 *      producerMonitoring과 같이 전역변수 isConsumeItemFair를
 *      사용하게 됩니다. 25를 넘어가는 수 일경우 모니터링 결과에
 *      결리게 되고 이후에 2를 나눠야 한다는 것을 consume에
 *      알려주게 됩니다. 마치게 될 경우 mu3을 unlock하여
 *      consumer에서 소비를 할 수 있게 해 줍니다.
 */
void *consumerMonitoring(){
    while(1){
        pthread_mutex_lock(&mu4);
        isConsumeItemFair = 1;
        if(consumerCheckItem > 25){
            printf("소비모니터링 : %d\n", consumerCheckItem);
            isConsumeItemFair = -1;
        }
        pthread_mutex_unlock(&mu3);
    }
}
/*
 * 함수명 : int insert_item(buffer_item *item)
 * 설명 : item이 생성 되었을 경우 empty값을 감소시키고
 *      값을 버퍼에 넣어 주는 동안 mutex를 걸게 됩니다.
 *      이때 empty가 0보다 작다면 full한 상태이므로 wait상태로
 *      머물게 됩니다. buffer의 rear부분에 값을 넣어주고
 *      produce한 값을 출력시켜줍니다.
 *      환형 배열을 사용하고 있으므로 다음 생성이 이뤄지면
 *      값이 추가되어야 하는 부분을 rear증가로 알려줍니다.
 *      이후에 mutex를 풀게 되고 full값을 증가시켜 consumer에서
 *      값을 소비 할 수 있게 해 줍니다.
 */
int insert_item(buffer_item *item){
    if(sem_wait(&empty)!=0) return -1;
    if(pthread_mutex_lock(&mutex) != 0) return -1;

    buffer[rear]=*item;
    printf("produce : %d\n",buffer[rear]);
    rear=(rear+1)%BUFFER_SIZE;

    if(pthread_mutex_unlock(&mutex)!=0) return -1;
    if(sem_post(&full)!=0) return -1;
    return 0;
}
/*
 * 함수명 : remove_item(buffer_item *item)
 * 설명 : full값이 0일 경우 생성된 아이템이 없다는 뜻입니다.
 *      이때는 wait상태로 기다리다 Producer에서 생성이 되었을
 *      경우 실행하게 됩니다. 환형 배열이므로 front값을 consume
 *      하여 줍니다. 소비를 한 후 버퍼의 값을 0으로 바꿔준 후
```

```
 *      front값을 증가 시켜 줍니다.
 *      소비를 한 후 empty값을 1증가 시켜줍니다. full이었던 경우
 *      Producer는 다시 생산을 시작하게됩니다.
 */
int remove_item(buffer_item *item){
    if(sem_wait(&full)!=0) return -1;
    if(pthread_mutex_lock(&mutex)!=0) return -1;
    printf("consume : %d\n", buffer[front]);
    buffer[front] = 0;
    front= (front+1)%BUFFER_SIZE;
    if(pthread_mutex_unlock(&mutex)!=0) return -1;
    if(sem_post(&empty)!=0) return -1;
    return 0;
}
/*
 * 함수명 : void *producer(void *param)
 * 설명 : 먼저 생성된 producer 스레드는 1~5 사이의 sleep
 *      time을 갖게 됩니다. 이 sleep이 먼저 풀린 쓰레드
 *      부터 생산을 시작하게 됩니다. 생산을 시작하게 되는
 *      경우 전역변수 producerCheckItem을 이용하여 모니터를
 *      수행하게 되므로 promu라는 mutex를 이용하여 lock
 *      하였습니다. 또한 mu1값을 trylock을 통해 lock을 걸어
 *      모니터가 끝나지 않고 insert 하는 경우를 방지했습니다.
 *      1~100사이의 값이 생성이 되면 producerCheckItem에 값을
 *      넣어주고 mutex를 unlock 하여 모니터링을 실행해줍니다.
 *      만약 -1이 isProduceItemFair에 들어가게 된다면 버퍼에
 *      값을 추가시키지 않게 됩니다.
 */
void *producer(void *param){
    buffer_item item;
    while(1){
        int sleepTime = rand() % 5 + 1;
        sleep(sleepTime);

        pthread_mutex_lock(&promu);
        pthread_mutex_trylock(&mu1);
        /* generate a random number between 1 and 100 */
        item = rand() % 100 + 1;
        producerCheckItem = item;

        pthread_mutex_unlock(&mu2);
        pthread_mutex_lock(&mu1);
        if(isProduceItemFair == 1){
            if(insert_item(&item))
                printf("report error condition");
        }
        pthread_mutex_unlock(&promu);
    }
}
/*
 * 함수명 : void *consumer(void *param)
 * 설명 : 먼저 생성된 consumer 스레드는 1~5 사이의 sleep
 *      time을 갖게 됩니다. 이 sleep이 먼저 풀린 쓰레드
 *      부터 소비를 시작하게 됩니다. 소비를 하는 동안 다른
 *      쓰레드가 소비를 하는 것을 막았습니다. 모니터링 후
```

```
 *      소비를 해야 했기 때문입니다. consumerCheckItem에
 *      체크를 해야할 아이템을 넣고 모니터링을 수행합니다.
 *      isConsumeItemFair이 1이 아닐경우 모니터링에 걸린
 *      것이므로 buffer[front]값을 2로 나눠주고 remove_item을
 *      수행하여 소비하게 됩니다.
 */
void *consumer(void *param){
    buffer_item item;
    while(1){
        int sleepTime = rand() % 5 + 1;
        sleep(sleepTime);

        pthread_mutex_lock(&conmu);
        pthread_mutex_trylock(&mu3);
        consumerCheckItem = buffer[front];
        if(consumerCheckItem == 0){
            pthread_mutex_unlock(&conmu);
            continue;
        }

        pthread_mutex_unlock(&mu4);
        pthread_mutex_lock(&mu3);
        if(isConsumeItemFair != 1){
            buffer[front] = buffer[front] / 2;
        }
        if(remove_item(&item))
            printf("report error condition");
        pthread_mutex_unlock(&conmu);
    }
}

int main(int argc, char *argv[]){
    /*
     * 1. Get command line arguments argv[1], argv[2], argv[3]
     *    arg[1] :terminating 전에 얼마나 sleep 할지
     *    arg[2] : producer thread number
     *    arg[3] : consumer thread number
     */
    int sleepTime = atoi(argv[1]);
    int numOfProducer = atoi(argv[2]);
    int numOfConsumer = atoi(argv[3]);
    /*
     * 2. Initialize
     * Producer, Consumer 쓰레드와 모니터링 쓰레드의 아이디입니다.
     * 버퍼의 크기를 10으로 초기화 해줬습니다.
     * empty semaphore 값을 버퍼의 크기로 초기화 해주고 full을 0으로
     * 초기화 해줍니다.
     * 그리고 사용되는 모든 mutex는 binary mutex이므로 NULL로 초기화
     * 해 줍니다.
     */
    pthread_t Ctid[100], Ptid[100], Mtid1, Mtid2;
    buffer=(buffer_item*)malloc(sizeof(buffer_item)*BUFFER_SIZE);
    sem_init(&empty,0,BUFFER_SIZE);
    sem_init(&full,0,0);
    pthread_mutex_init(&mu1,NULL);
```

```
    pthread_mutex_init(&mu2,NULL);
    pthread_mutex_init(&mu3,NULL);
    pthread_mutex_init(&mu4,NULL);
    pthread_mutex_init(&mutex,NULL);
    pthread_mutex_init(&promu,NULL);
    pthread_mutex_init(&conmu,NULL);
    /*
     * 3. Create producer thread(s)
     */
    for(i=0; i<numOfProducer; i++){
        pthread_create(&Ptid[i],NULL,producer,NULL);
    }
    /*
     * 4. Create consumer thread(s)
     */
    for(i=0;i<numOfConsumer;i++){
        pthread_create(&Ctid[i],NULL,consumer,NULL);
    }

    pthread_mutex_lock(&mu2);
    pthread_mutex_lock(&mu4);
    /*
     * 모니터링 쓰레드를 생성해줍니다.
     */
    pthread_create(&Mtid1, NULL,producerMonitoring,NULL);
    pthread_create(&Mtid2, NULL,consumerMonitoring,NULL);
    /* 5. Sleep */
    sleep(sleepTime);
    /* 6. Exit */
    return 0;
}
```

테스트환경 및 어려웠던 점

- 클라우드 환경
  - testSite : https://c9.io
  - gcc --version : 4.8.4

```
hanjungv:~/workspace $ gcc --version
gcc (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4
Copyright (C) 2013 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

- 어려웠던 점
  1. producer과 consumer 내에 sleep을 다르게 줘 생성과 소비시점을 각 스레드에 따라 다르게 하여 처음에 어떻게 작동하는지 이해를 하기 힘들었던 점이 있었습니다.
  2. 또한 여러 Producer들이 생산하여 버퍼에 값을 넣으려 하고 Consumer 또한 여러 스레드들이 소비를 하려하여 어느 시점에 만들어진 스레드가 어떤 것을 생성하고 어떤 것을 소비하려는 지 알기 힘들었습니다.
  3. 이러한 문제를 단순화 하여 monitoring에 걸리는 Item과 생성, 소비 되는 아이템을 출력하여 제대로 작동하는지 확인하여 해결했습니다.

느낀점

- 이렇게 공유되는 변수에 동시에 여러 스레드들이 접근을 할 때 어떻게 문제를 해결할 지에 대해 실제적으로 고민해 볼 수 있는 시간이었습니다.
- 현재 캡스톤 프로젝트에서 하나의 데이터 베이스에 여러 request들이 동시에 접근이 되면 안되는 상황이 있었는데 이러한 부분을 공부하여 해결을 할 수 있는 방안을 생각 할 수 있게 되었습니다.