운영체제 lab2 보고서

32181854 박준영

1. 문제정의

- 생산자와 소비자 5명이 공유하는 concurrent queue 생성

- Structure: Linked List = Buffer

생산자가 RR 방식으로 Buffer에 데이터를 넣으면 소비자가 Buffer에 접근하여 데이터를 get한다.

2. 설계/구현

〈lab2_sync_types.h〉

```
typedef struct Node {
        int car_num;
        struct Node *next;
}Node;

typedef struct car_queue {      //queue for RR
        int balance;
        Node *front, *rear;
        pthread_mutex_t head_lock, tail_lock;
}CQ;

typedef struct workload {
        int p_time[5];          // start producing time
        int p_num[5];           // the number of cars
        int current[5];         // current number of cars (initial: 0)
}WL;

void QueueInit(CQ* q);
void put(CQ* q, int car_num);
int get(CQ *q);
void *producer(void *arg);
void *consumer(void *arg);
```

Lock 변수 두개와 workload를 저장하기 위한 구조체를 추가했습니다. (current 배열은 삭제했습니다.)

〈lab2_sync.c〉

```
/*
 *   DKU Operating System Lab
 *           Lab2 (Vehicle production Problem)
 *           Student id : 32181854
 *           Student name : park jun-yeong
 *
 *   lab2_sync.c :
 *       - lab2 main file.
 *       - must contains Vehicle production Problem function's declations.
 *
 */

#include <aio.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <time.h>
#include <sys/time.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <assert.h>
#include <pthread.h>
#include <asm/unistd.h>

#include "lab2_sync_types.h"

```

헤더파일은 수정하지 않았습니다.

```
//int count;
pthread_cond_t empty, fill;

CQ buffer;
Node car[5];
WL wl;

void QueueInit(CQ* q) {
        q->balance = 0;
        q->front = NULL;
        q->rear = NULL;
}

void put(CQ *q, int car_num) {                  //buffer insert
        Node* tmp = (Node*)malloc(sizeof(Node));
        tmp->car_num = car_num;
        tmp->next = NULL;

        q->rear->next = tmp;
        q->rear= tmp;

        q->balance++;
        wl.p_num[car_num]--;
}

int get(CQ* q) {                            //buffer pop
        int tmp = q->front->car_num;
        q->front = q->front->next;
        q->balance--;
        return tmp;
}
```

전역변수로 condition variable인 empty, fill을 선언, buffer와 car 노트 5개, workload 구조체의 객체를 선언한다.

1) QueueInit

buffer의 초기화를 위한 함수입니다.

2) put, get

Data를 buffer에 넣고 빼기 위한 함수입니다.

Put은 출고하고자 하는 자동차의 number만 받아와 새로운 노드를 생성 후 뒷 노드에 연결합니다.

Put은 출고, get은 구매에 대응할 수 있습니다. 따라서 각각의 수행마다 balance를 +1, -1 해줍니다.

```
void *producer(void* arg) {
        int* time_quantum = (int*)arg;
        //RR
        int timer = 0;
        int i, j;
        Node* p = buffer.front;
        Node* q;

        while ( wl.p_num[0] | wl.p_num[1] | wl.p_num[2] | wl.p_num[3] | wl.p_num[4] )
        {
                for (i = 0; i < 5; i++)
                        if ((wl.p_time[i] <= timer) & wl.p_num[i]) {
                                //while (buffer.balance == MAX_SIZE)
                                //      pthread_cond_wait(&empty, &(buffer.tail_lock));

                                for(j=0; j<*time_quantum; j++)
                                        put(&buffer, car[i].car_num);          //critical
section
                        }
                //pthread_cond_signal(&fill);

                while (p)
                {
                        if (wl.p_num[p->car_num] <= *time_quantum)  //parameter timequantum
                                wl.p_num[p->car_num] = 0;       // end producing
                        else {
                                timer += *time_quantum;
                                wl.p_num[p->car_num] -= *time_quantum;
                                wl.p_time[p->car_num] += *time_quantum;
                        }

                        q = p;
                        p = p->next;
                }
                p = q;
        }
}
```

3) producer

(producer의 lock과 condition variable은 실험을 위해 위치를 변경하여서 주석 처리 돼 있습니다. 기본적으로는 함께 동작되는 코드입니다.)

RR 스케줄링 방식으로 자동차를 출고합니다. Buffer에 자동차의 node를 넣습니다.

만약 balance가 MAX_SIZE와 같다면 empty signal을 받을 때 까지 wait 합니다. 만약 아니라면 현재 timer보다 생산시간이 작은 (그러니까, 이미 생산이 시작된) 자동차들을 time quantum 개수 만큼 buffer에 넣어줍니다.

While(p) 문 부터는 남은 자동차의 생산량, timer 등을 조정합니다. Time_quantum 보다 남은 자동차의 수가 더 적다면 자동차의 수를 0으로 만들어 생산을 종료시킨다.

```
void *consumer(void *arg) {
        int i;
        while (buffer.balance)
        {
                /*pthread_mutex_lock(&(buffer.head_lock));
                while (buffer.balance == 0)
                        pthread_cond_wait(&fill, &(buffer.head_lock));*/
                int tmp = get(&buffer);
                //pthread_cond_signal(&empty);
                //pthread_mutex_unlock(&(buffer.head_lock));
                printf("%d\n", tmp);
        }

}
```

4) consumer

버퍼에서 값을 가져온다. (여기서도 lock과 condition variable은 기본적으로 함께 동작하는 코드입니다.)

3. 실험결과

1) lock 여부

-NoLock

```
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/oslab/Desktop/2021_DKU_OS/lab2_sync/lab2_sync -c=100 -q=1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
==== Vehicle Production Problem ====
(1) No Lock Experiment
[New Thread 0x7ffff7d9d700 (LWP 26810)]
[New Thread 0x7ffff759c700 (LWP 26811)]

Thread 2 "lab2_sync" received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7ffff7d9d700 (LWP 26810)]
0x000055555555529c in put (q=0x555555558060 <buffer>, car_num=0) at lab2_sync.c:50
50              q->rear->next = tmp;
```

-Lock

```
oslab@oslab-DKU:~/Desktop/2021_DKU_OS/lab2_sync$ ./lab2_sync -c=100 -q=1
==== Vehicle Production Problem ====
(1) No Lock Experiment
==== Vehicle Production Problem ====
(1) Lock/no-Lock ExperimentExperiment Info
        Total Produce Number = 100
        Final Balance Value = 0
        Execution time = 0.000000
```

2) fine-grained/coarse-grained lock

Coarse-grained lock에서는 put 함수에 들어가기 전에 lock을 잡아 put 동작 자체를 lock하고, fine-grained lock은 put 함수 내에서 노드를 link하는 부분에 한해서만 lock을 잡습니다.

-coarse-grained lock

```
oslab@oslab-DKU:~/Desktop/2021_DKU_OS/lab2_sync$ ./lab2_sync -c=100 -q=1
==== Vehicle Production Problem ====
(1) No Lock Experiment
==== Vehicle Production Problem ====
(1) Lock/no-Lock ExperimentExperiment Info
        Total Produce Number = 100
        Final Balance Value = 0
        Execution time = 0.000000
```

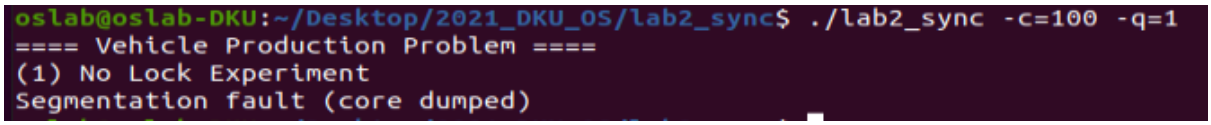
-fine-grained lock

Lock을 변경하면 또 segmentation fault가 발생합니다...

3) thread 개수

스레드 개수를 producer 1개, consumer 5개로 잡고 진행해서 어떻게 바꿔야 할지 감이 오지 않습니다...

4. 논의



Gdb를 사용하지 않고 돌라면 segmentation fault가 발생한다.. thread 끼리 꼬인 것 같은데 하나씩 구조적으로 이해해서 풀어내려면 지각제출이 될 것 같아서 우선은 실험을 진행했다... 당연히 실험 진행이 원활하게 이루어지지 않았다. 교수님께서 시간을 일주일이나 더 주셨는데 시험기간과 중간 대체 과제들과 함께 하다 보니 깊게 생각하면서 코드를 수정하지 못했다. 그래도 며칠동안 밤을 새면서 코드를 만들고 수정해보았는데 갑자기 스레드가 죽거나 실행되지 않는 오류가 매우 빈번하게 발생했다. 스레드를 다루는 실습은 스케줄링과도 연관이 있어 정말 어렵다는 것을 알았다.

자료구조 큐에 데이터를 enqueue, dequeue 하는 것은 어렵지 않다고 생각해서 슬로피 카운터 대신에 큐를 선택하고, 지난번 lab1에서 약간은 엉성하게 구현한 RR을 이번에는 잘 구현해야겠다는 다짐으로 큐와 RR을 그대로 진행했는데 결과가 잘 나오지 않았다. 인터넷에서 RR을 구현하는 소스코드를 찾아 공부도 해 보았지만, 큐에 직접적으로 노드를 넣어 구현하는 방식이 많지가 않았다. 다른 중간고사 과제들을 하면서 중간에 조금씩 시간을 냈었다면 결과가 달랐을까 후회도 했다. 다음 LAB3에서는 좀 더 여유를 가지고 과제를 진행해서 더 좋은 결과를 얻고 싶다.