REPORT

제목 : Thread 스케줄링과 multi-level queue 구현

과 목 명: 오퍼레이팅시스템

제출일자: 2020.05.16

학 과: 컴퓨터공학과

학 번: 12161597

이 름: 부성준

목차

1. 프로그램 목적

Multi-Thread Programming과 Thread Synchronization, Scheduler simulation을 이해하고 적용한다.

Multi-Thread Programming은 시스템 자원을 효율적으로 사용하고 프로그램의 응답 속도를 높일 수 있다는 장점이 있다. 따라서 본 보고서에서는 Multi-Threaded program을 직접 구현하고 동작 시킨다.

Thread들을 병렬적으로 수행하기 위해서는 Thread Synchronization(스레드 동기화)이 필요하다. Thread가 공유 자원을 사용할 때 문제가 발생할 수 있기 때문이다. 따라서 Semaphore를 이용하여 Thread의 실행 순서를 동기화 한다.

여러 Thread들의 순서를 정하고 정한 순서에 따라 수행하기 위해서는 Scheduler가 필요하다. 본 보고서에서는 4개의 Queue를 가지고 있는 Multi-Level Queue를 구현하고 각 Queue들을 우선순위에 따라 순서를 정할 것이다. Queue들은 각각 SJF(Shortest Job First), Priority(우선순위), Round-Robin 스케줄링, Priority-Round-Robin 스케줄링을 이용하여 thread를 수행시킨다. 마지막 'Priority-Round-Robin' 스케줄링은 필자가 만든 것으로 Priority 스케줄링와 Round-Roubin 스케줄링을 합친 것이다. Priority-Round-Robin 스케줄링의 특징은 프로세스를 골고루 수행하면서 우선순위가 높은 프로세스를 먼저 수행하는 경향을 보인다는 것이다. 우선순위가 높지만 매우 긴 수행시간을 가진 프로세스가 주어지더라도 time quantum 만큼 수행하고 다음 프로세스로 순서가 넘어가기 때문에 프로세스 수행이 정체되지 않는다. 추가로 Queue들간의 스케줄링을 담당하는 thread가 존재하고, 각 Queue 안의 스케줄링을 담당하는 thread가 존재하도록 구현한다.

2. 프로그램 설명

본 프로그램은 여러 개의 프로세스 정보를 입력 받았을 때, 프로세서들을 multi-level queue에 할당하고, 각 queue에서는 각각 SJF(Shortest Job First), 우선순위(Priority), Round-robin, Priority-Round-Robin 기법으로 프로세스들을 스케줄링 한 뒤 순서에 맞춰 프로세스를 수행한다. 이때 큐 사이에는 우선순위가 존재한다. 프로그램은 실행될 때 단위 시간마다 프로세스 id를 출력한다. 출력은 띄어쓰기로 구분된다. 예를 들어 id가 3인 프로세스의 수행 시간이 5라면 '3 3 3 3 3'를 출력하게 된다. 이 과정을 각각의 프로세스마다 실행하며 서로 다른 프로세스는 서로 다른 줄에 출력한다.

(1) 입력 정보

각 프로세스 마다 큐의 클래스 번호, 프로세스 id, 우선순위, 수행 시간을 직접 입력해야 한다. 하나의 프로세스 정보는 띄어쓰기로 구분되며 서로 다른 프로세스 정보는 서로 다른 줄에 입력된다. 마지막 줄에는 '0 0 0 0'을 입력하여 입력 종료를 알린다.

큐의 클래스 정보는 1, 2, 3, 4 중 하나의 값이다. 1번 큐에서는 SJF(Shortest Job First), 2번 큐에서는 우선순위, 3번 큐에서는 Round-Robin 스케줄링, 4번 큐에서는 Priority-Round-Robin 스케줄링을 이용한다. 그리고 큐 간의 스케줄링은 Priority 스케줄링을 사용하며 큐의 클래스 번호가 낮을수록 우선순위가 높다. 즉, 1번 큐 > 2번 큐 > 3번 > 4번 큐이다.

프로세스 id로는 100 이하의 자연수이며 각각의 프로세스는 고유한 id를 가진다. 따라서 프로세스 id는 모두 다르게 주어진다.

우선순위(Priority)는 0 또는 100 이하 양의 정수이다. 양의 정수인 경우 숫자가 작을수록 우선순위가 높다. 큐의 클래스 번호가 1 또는 3인 프로세스의 경우 우선순위가 무시되므로 0으로 주어진다.

수행 시간은 100 이하의 자연수로 주어지며 시간 단위는 ms이다. 만약 프로세스의 수행 시간이 3으로 입력되었다면 프로세스는 3ms동안 수행하게 된다.

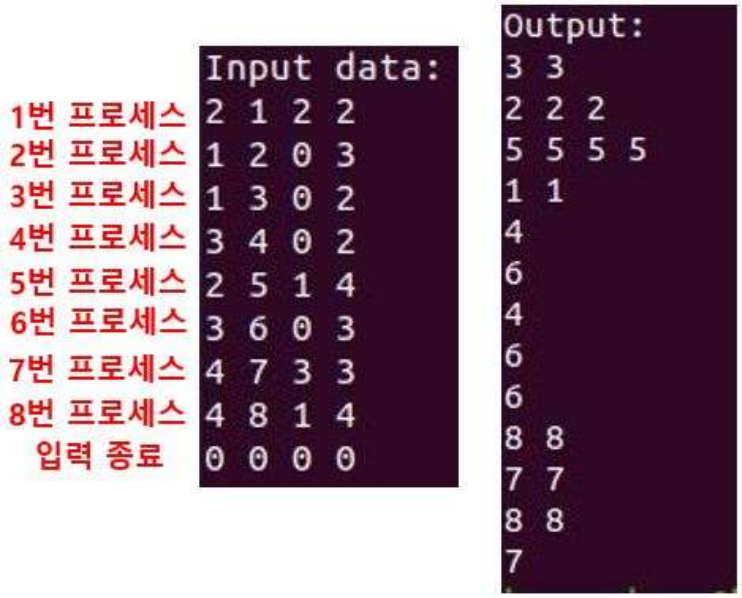
(2) 출력 정보

프로세스가 실행되면 수행 시간만큼 id를 출력한다. 예를 들어 수행 시간이 5인 프로세스는 id를 5번 출력한다. 이 때 출력한 id는 띄어쓰기로 구분이 되고 서로 다른 프로세스는 서로 다른 줄에 출력이 된다.

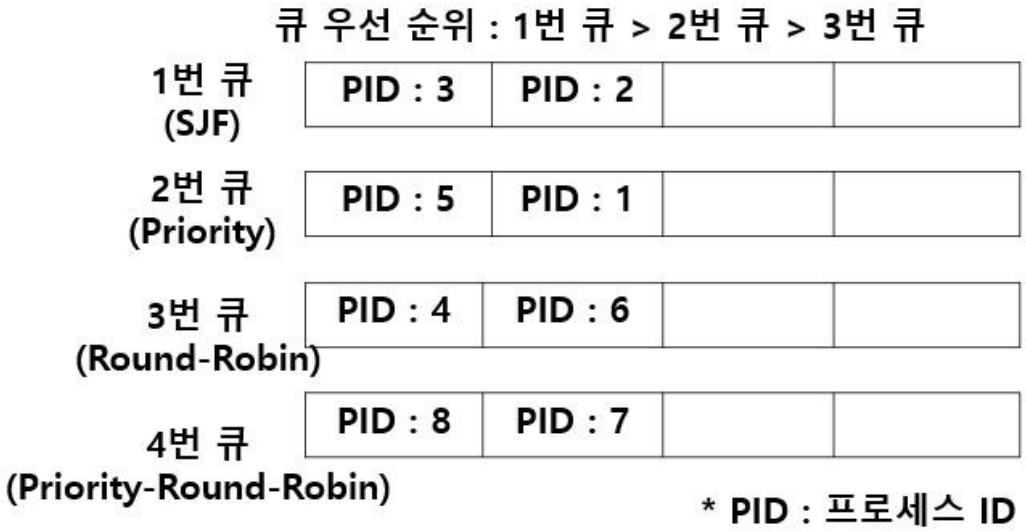
3번과 4번 큐의 프로세스가 수행될 때에는 각 프로세스의 수행 시간이 끝나기 전에 다른 프로세스로 번갈아 수행될 수 있기 때문에 같은 프로세스 id가 서로 다른 줄에 있을 수 있다. Round-

Robin 스케줄링에서 time quantum은 1ms이다. 따라서 큐 내에 있는 프로세스들은 1ms씩 번갈아 가며 수행된다. 그리고 Priority-Round-Robin 스케줄링에서 time quantum은 2ms이다.

(3) 입출력 예시



위의 예시에서 Input에서는 8개의 프로세스 정보를 서로 다른 줄에 입력하고 마지막 줄에는 '0 0 0 0'을 입력하여 입력 종료를 알리고 있다. 이에 따른 프로그램 수행 결과는 Output과 같다.



프로그램 수행 과정을 살펴보면 1번 큐에는 2번, 3번 프로세스가 들어간다. 2번 큐에는 1번, 5번 프로세스가 들어간다. 3번 큐에는 4번, 6번 프로세스가 들어간다. 4번 큐에는 7번, 8번 프로세스가 들어간다. 1번 큐는 SJF로 스케줄링을 이용하므로 3번, 2번 프로세스 순으로 진행된다. 2번 큐는

Priority 스케줄링을 이용하므로 5번 1번 프로세스 순으로 진행된다. 3번 큐는 Round-Robin 스케줄링을 이용하므로 4번 6번 프로세스가 time quantum을 두고 각 프로세스의 수행 시간만큼 번갈아 가며 수행된다. 4번 큐는 Priority-Round-Robin 스케줄링을 이용하므로 우선 순위대로 8번 7번 프로세스 순서로 time quantum을 두고 번갈아 가며 수행된다.

1번 큐 수행 결과

3 3
2 2 2

큐 우선 순위는 1번 큐가 가장 높으므로 1번 큐 내의 프로세스들이 먼저 수행된다. 1번 큐에 들어있는 3번 프로세스의 수행 시간은 2이고, 2번 프로세스의 수행 시간은 3이다. 3번 프로세스의 수행 시간이 가장 짧기 때문에 3번 프로세스가 먼저 수행되고 그 다음에 2번 프로세스가 수행된다. 따라서 1번 큐의 수행 결과는 다음과 같다.

2번 큐 수행 결과

5 5 5 5
1 1

1번 큐가 끝나면 그 다음으로 우선 순위가 높은 2번 큐가 수행된다. 2번 큐 내에 있는 5번 프로세스의 Priority는 1이고 1번 프로세스의 Priority는 2이다. 따라서 5번 프로세스가 먼저 수행되고 다음으로 1번 프로세스가 수행된다. 2번 큐 수행 결과는 다음과 같다.

3번 큐 수행 결과

4
6
4
6
6

2번 큐가 끝나면 3번 큐가 수행된다. 여기서 사용되는 Round-Robin의 time quantum은 1ms이다. 그리고 프로세스 id가 출력될 때마다 1ms가 소요된다. 따라서 프로세스가 id를 1번 출력하면 다음 프로세스로 순서가 넘어간다. 이렇게 각 프로세스가 수행되다가 큐의 마지막 프로세스가 1ms만큼 수행되면 다시 큐 내의 수행이 아직 끝나지 않은 처음 프로세스로 순서가 넘어간다. 해당 프로세스가 1ms만큼 수행되면 아직 수행이 끝나지 않은 다음 프로세스로 순서가 넘어간다. 이러한 과정이 큐 내의 모든 프로세스의 수행이 끝날 때까지 반복된다. 3번 큐의 수행 결과는 다음과

같다.

4번 큐 수행 결과

8 8
7 7
8 8
7

마지막으로 4번 큐가 수행된다. 여기서 사용되는 Priority-Round-Robin은 Priority 스케줄링과 Round-Robin을 합친 것이고 time quantum은 2ms이다. 따라서 우선 순위가 높은 8번 프로세스가 먼저 수행된다. 8번 프로세스가 2ms만큼 수행되어 id를 두 번 출력하면 그 다음으로 우선순위가 높은 프로세서로 순서가 넘어간다. 다음 프로세서인 7번 프로세스도 2ms만큼 수행되고 나면 다시 8번 프로세스로 순서가 넘어간다. 이런 식의 과정을 각 프로세스의 수행이 모두 끝날 때까지 반복한다.

3. 프로그램 구현

본 프로그램은 Ubuntu (64-bit) 운영 체제에서 c언어로 구현하고 실행하였으며 gcc 컴파일러를 이용하여 컴파일 하였다.

(1) 사용된 헤더 파일

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <semaphore.h>
#include <unistd.h>
```

POSIX Thread와 Semaphore를 이용하기 위해 'pthread.h'와 'semaphore.h' 헤더 파일을 사용한다. 'unistd.h'는 sleep()함수를 이용하기 위해 사용하였고 'stdio.h'와 'stdlib.h'는 나머지 c언어 사용을 위해 사용하였다.

(2) 전역 변수와 함수 선언

```
sem_t mutexq; //semaphore for queues
sem_t mutexp; //semaphore for threads

void *runner(void *arg); //thread that print process id
void *q1Shed(void *args); //scheduler for 1st queue
void *q2Shed(void *args); //scheduler for 2nd queue
void *q3Shed(void *args); //scheduler for 3rd queue
void *q4Shed(void *args); //scheduler for 4th queue
void *qSched(void *args); //scheduler for queues
```

전역 변수 'mutexq'는 multi-queue level에서 queue들 간의 동기화를 위해 사용하는 semaphore 변수이다. 'mutexp'는 queue 내에서 thread들 간의 동기화를 위해 사용되는 semaphore 변수이다.

선언된 함수인 'runner'는 프로세스 id를 출력하는 thread이다. 여기서 프로세스 id를 한 번 출력하며, 한 번 출력할 때 1ms가 소요된다. 'q1Shed'는 1번 큐의 스케줄링을 담당한다. 'q2Shed'는 2번 큐의 스케줄링을 담당한다. 'q3Shed는 3번 큐의 스케줄링을 담당한다. 'q4Shed는 4번 큐의 스케줄링을 담당한다. 마지막 'qSched'는 큐들 간의 스케줄링을 담당한다.

(3) main함수

```
int main(){
    //array that saves informations of processes
    int arr[101][4] = {0, };

    printf("Input data: \n");

    //get informatinos of processes
    int a, b, c, d;
    int i = 0;
    while(scanf("%d %d %d %d", &a, &b, &c, &d)){
        if(a == 0 && b == 0 && c == 0 && d == 0)
            break;
        arr[i][0] = a;
        arr[i][1] = b;
        arr[i][2] = c;
        arr[i][3] = d;
        i++;
    }

    printf("Output: \n");
    //thread identifier for queue scheduling
    pthread_t pqs;
    //cteate thread for queue scheduling
    pthread_create(&pqs, NULL, qSched, arr);
    //wait for the thread to exit
    pthread_join(pqs, NULL);

    return 0;
}
```

메인 함수에서는 우선 크기가 101 x 4 인 'arr' 배열을 만든 뒤 프로세스 정보들을 입력 받아서 이곳에 저장한다. 각 프로세스의 정보는 scanf 함수를 통해 큐의 클래스 번호, 프로세스 id, 우선순위, 수행 시간 순서로 입력 받는다. 프로세스 정보를 다 입력한 후에는 '0 0 0 0'을 입력하여 입력 종료를 알려준다. 'arr' 배열의 크기가 101 x 4 이므로 입력 받을 수 있는 프로세스의 개수는 최대 100개이다.

그 다음 thread의 identifier 'pqs'를 생성하고 'pthread_create'를 통하여 queue들 간의 스케줄링을 담당하는 thread인 'qSched'를 생성한다. 여기에 프로세스들의 정보가 담긴 'arr'를 전달 인자로 넘겨준다. 이후 main함수는 'pthread_join'으로 인하여 'qSched' thread의 수행이 끝날 때까지 기다린다.

]

(4) 프로세스 id를 출력하는 thread

```
//thread that prints process id
//arg : process id
void *runner(void *arg){
    //wait
    sem_wait(&mutexp);

    //get process id
    int id = *(int *)arg;

    //print process id once
    printf("%d ", id);
    //halt for 1ms
    usleep(1000);

    //signal
    sem_post(&mutexp);
}
```

'runner'는 프로세스 id를 출력하는 thread이다. 출력하려는 프로세스 id를 인자로 전달받는다.

'sem_wait(&mutexp)'를 통하여 다른 프로세스의 id를 출력하는 thread가 수행되지 않도록 동기화를 한다. 그 다음 프로세스 id를 한 번 출력한 뒤 'usleep(1000)' 함수를 통하여 1ms동안 멈춘다. 마지막으로 semaphore를 signal한다. 'runner'에서 'usleep(1000)' 외의 소요되는 시간은 없다고 가정한다. 따라서 'runner'의 총 수행 시간은 1ms이다.

(5) Queue들 간의 스케줄을 담당하는 thread

```
//scheduler for queues
//args : informations of processes
void *qSched(void *args){
    //get informations of processes
    int (*arr)[101][4] = args;

    int q1[101][4] = {0, }; //1st queue
    int q2[101][4] = {0, }; //2nd queue
    int q3[101][4] = {0, }; //3rd queue
    int q4[101][4] = {0, }; //4th queue

    int qCnt[5] = {0, 0, 0, 0, 0}; //count each queues
```

Queue들 간의 스케줄을 담당하는 thread인 'qSched'는 프로세스들의 정보가 담긴 배열을 인자로 전달받은 뒤, 이 정보들을 queue의 class 번호에 따라 분류할 배열 q1, q2, q3, q4를 생성한다. 4개 배열은 각각 1번째 queue, 2번째 queue, 3번째 queue, 4번째 queue에 들어갈 프로세스의 정

보들을 저장한다. 'qCnt' 배열은 현재 각 큐에 몇 개의 프로세스가 들어있는지 저장한다. qCnt[i]에는 i번 큐에 들어 있는 프로세스의 개수가 저장된다.

```
    //classify processes according to queue class number
    int i = 0;
    while(1){
        //get class no., process id, priority, runtime in line
        int temp[4];
        for(int j = 0; j < 4; j++){
            temp[j] = (*arr)[i][j];
        }
        //break when temp gets 0
        if(temp[0] == 0 && temp[1] == 0 && temp[2] == 0 && temp[3] == 0)
            break;

        //categorized as 1st queue
        if(temp[0] == 1){
            int idx = qCnt[1]++;
            for(int j = 0; j < 4; j++){
                q1[idx][j] = temp[j];
            }
        }
        //categorized as 2nd queue
        else if(temp[0] == 2){
            int idx = qCnt[2]++;
            for(int j = 0; j < 4; j++){
                q2[idx][j] = temp[j];
            }
        }
        //categorized as 3rd queue
        else if(temp[0] == 3){
            int idx = qCnt[3]++;
            for(int j = 0; j < 4; j++){
                q3[idx][j] = temp[j];

            }
        }
        //categorized as 4th queue
        else if(temp[0] == 4){
            int idx = qCnt[4]++;
            for(int j = 0; j < 4; j++){
                q4[idx][j] = temp[j];

            }
        }

        i++;
    }
```

그 다음 while 루프를 통하여 프로세스들을 queue class 번호에 따라 분류하여 해당 배열에 정보를 저장한다. 'temp' 배열을 생성하여 하나의 프로세스의 정보를 저장한 뒤 프로세스가 해당하는 큐에 프로세스 정보를 넣어준다. 큐에 프로세스 정보를 넣을 때, 현재 큐에 몇 개의 프로세스가 들어있는지 정보를 'qCnt'를 통해 알아낸 뒤, 배열의 알맞은 인덱스에 프로세스 정보를 넣는다. 이 과정을 모든 프로세스에 반복하며 'temp'의 모든 값이 0인 경우 마지막 프로세스까지 모두 분류

했다는 의미이므로 while 루프를 빠져나온다.

```
    //semaphore for queues
    sem_init(&mutexq, 0, 1);

    //thread identifier for scheduler of 1st, 2nd, 3rd queue
    pthread_t ps1, ps2, ps3, ps4;

    //create thread for scheduler of 1st queue
    pthread_create(&ps1, NULL, q1Shed, q1);
    //wait for the thread to exit
    pthread_join(ps1, NULL);

    //create thread for scheduler of 2nd queue
    pthread_create(&ps2, NULL, q2Shed, q2);
    //wait for the thread to exit
    pthread_join(ps2, NULL);

    //create thread for scheduler of 3rd queue
    pthread_create(&ps3, NULL, q3Shed, q3);
    //wait for the thread to exit
    pthread_join(ps3, NULL);

    //create thread for scheduler of 4th queue
    pthread_create(&ps4, NULL, q4Shed, q4);
    //wait for the thread to exit
    pthread_join(ps4, NULL);

    sem_destroy(&mutexq);

}
```

그 다음 queue들 간의 동기화를 위해 사용하는 semaphore를 생성하고 1번 큐, 2번 큐, 3번 큐, 4번 큐의 스케줄링을 담당하는 thread identifier 'ps1', 'ps2', 'ps3', 'ps4'를 생성한다. 큐들 간의 우선순위는 1번 큐가 가장 높으므로 1번 큐의 스케줄링을 담당하는 thread인 'q1Sched'를 생성한 뒤 'pthread_join'을 통해 기다린다. 'q1Sched' thread가 수행을 모두 마치면, 그 다음으로 우선 순위가 높은 2번 큐의 스케줄링을 담당하는 thread인 'q2Sched'를 생성하고 마칠 때까지 기다린다. 'q2Sched'가 수행을 마치면 3번 큐의 스케줄링을 담당하는 thread인 'q3Sched'를 생성하고 마칠 때까지 기다린다. 'q3Sched'가 수행을 마치면 4번 큐의 스케줄링을 담당하는 thread인 'q4Sched'를 생성하고 마칠 때까지 기다린다. 'qeSched'가 수행을 마치면 마지막으로 semaphore를 삭제한다.

(6) SFJ scheduler (1번 queue scheduler)

```
//SJF Scheduler(scheduler for 1st queue)
//args : informations of processes in 1st queue
void *q1Shed(void *args){
    //wait
    sem_wait(&mutexq);

    //get informations of processes in 1st queue
    int (*q1)[101][4] = args;

    //number of processes in 1st queue
    int cnt = 0;
    //array that saves ids and runtimes of each process
    int id_rt[101][2];

    //save ids and runtimes of each process
    //  and get number of processes in 1st queue
    for(int i = 0; (*q1)[i][0] != 0; i++){
        id_rt[i][0] = (*q1)[i][1];
        id_rt[i][1] = (*q1)[i][3];
        cnt++;
    }
```

'q1Sched'는1번 queue에서 SJF 스케줄링을 담당하는 Thread이다. 1번 큐에 들어있는 프로세스 정보가 담긴 배열을 인자로 전달받는다. 이 Thread는 수행 시간이 짧은 프로세스 순서대로 수행시킨다.

'sem_wait(&mutexq)'를 통하여 다른 큐의 스케줄링을 담당하는 thread가 shared resource에 접근하지 못하도록 막아 동기화를 한다.

큐 안의 프로세스 개수를 저장하는 'cnt' 변수와 각각의 프로세스 정보 중 id와 수행시간만 저장하는 'id_rt'배열을 생성한다. id_rt[i][0]은 i번째 프로세스의 id를 저장하고 id_rt[i][1]은 i번째 프로세스의 수행시간을 저장한다. for 루프를 통하여 프로세스 개수와 각 프로세스의 id와 수행시간을 저장한다.

```
//bubble sort
//sort id_rt array by runtime
for (int i = 0; i < cnt; i++){
    for (int j = 0; j < cnt - (i + 1); j++){
        if(id_rt[j][1] > id_rt[j + 1][1]){
            int temp[2] = {id_rt[j + 1][0], id_rt[j + 1][1]};
            id_rt[j + 1][0] = id_rt[j][0];
            id_rt[j + 1][1] = id_rt[j][1];
            id_rt[j][0] = temp[0];
            id_rt[j][1] = temp[1];
        }
    }
}
```

'id_rt'배열을 수행 시간이 짧은 순으로 정렬한다. 여기서는 버블 소트가 사용되었다.

```
    //semaphore for threads
    sem_init(&mutexp, 0, 1);

    //create threads that print id for each processes
    for(int i = 0; i < cnt; i++){
        for (int j = 0; j < id_rt[i][1]; j++){
            pthread_t t1;

            int pid = id_rt[i][0];
            int *p = &pid;

            pthread_create(&t1, NULL, runner, p);
            pthread_join(t1, NULL);
        }
        printf("\n");
    }


    sem_destroy(&mutexp);

    //signal
    sem_post(&mutexq);
}
```

다음으로 id를 출력하는 thread들 간의 동기화를 위해 사용하는 semaphore를 생성한다. 이 semaphore는 큐들 간의 동기화를 위해 사용되는 semaphore와는 다르다. 전자는 'mutexp' 변수를 이용하고 후자는 'mutexq' 변수를 이용한다.

그 다음 2중 for 루프에서는 큐 안의 프로세스들이 수행 시간만큼 'runner' thread를 생성하고 종료하는 것을 반복한다. 'runner' thread는 한 번 생성될 때마다 프로세스 id를 1ms동안 1번 출력한

다. 따라서 프로세스의 수행 시간이 3이라면 'runner' thread를 3번 생성하고 종료한다.

큐 안의 모든 프로세스들이 수행을 마치면 'mutexp' 변수를 사용하는 semaphore를 제거한 뒤 'mutexq' 변수를 사용하는 semaphore를 signal한다.

(7) Priority scheduler (2번 queue scheduler)

```
//Priority Scheduler(scheduler for 2nd queue)
//args : informations of processes in 2nd queue
void *q2Shed(void *args){
    //wait
    sem_wait(&mutexq);

    //get informations of processes in 2nd queue
    int (*q2)[101][4] = args;

    //maximum priority of processes in queue
    int cnt = 0;
    for(int i = 0; (*q2)[i][0] != 0; i++){
        cnt++;
    }
```

'q2Sched'는 2번 queue에서 Priority 스케줄링을 담당하는 Thread이다. 2번 큐에 들어있는 프로세스 정보가 담긴 배열을 인자로 전달받는다. 이 Thread는 우선 순위가 높은 프로세스부터 수행시킨다.

'sem_wait(&mutexq)'를 통하여 다른 큐의 스케줄링을 담당하는 thread가 shared resource에 접근하지 못하도록 막아 동기화를 한다.

cnt 변수는 큐 안의 들어있는 프로세스의 개수를 저장하는 변수이다. for문을 통해 큐를 탐색하면서 프로세스의 개수를 구한다.

```
//bubble sort
//sort id_rt array by priority
for (int i = 0; i < cnt; i++){
    for (int j = 0; j < cnt - (i + 1); j++){
        if((*q2)[j][2] > (*q2)[j + 1][2]){
            int temp[4] = {(*q2)[j + 1][0], (*q2)[j + 1][1],
            (*q2)[j + 1][2], (*q2)[j + 1][3]};

            (*q2)[j + 1][0] = (*q2)[j][0];
            (*q2)[j + 1][1] = (*q2)[j][1];
            (*q2)[j + 1][2] = (*q2)[j][2];
            (*q2)[j + 1][3] = (*q2)[j][3];
            (*q2)[j][0] = temp[0];
            (*q2)[j][1] = temp[1];
            (*q2)[j][2] = temp[2];
            (*q2)[j][3] = temp[3];
        }
    }
}

//semaphore for runner threads
sem_init(&mutexp, 0, 1);
```

그 다음 버블 소트를 통해 배열 안의 프로세스들을 우선 순위 순으로 정렬한다. 그리고 id를 출력하는 thread들 간의 동기화를 위해 사용하는 semaphore를 생성한다. 이 semaphore는 큐들 간의 동기화를 위해 사용되는 semaphore와는 다르다.

```
    //create threads that print id for each processes
    for(int i = 0; i < cnt; i++){
        for (int j = 0; j < (*q2)[i][3]; j++){
            pthread_t t2;

            int pid = (*q2)[i][1];
            int *p = &pid;

            pthread_create(&t2, NULL, runner, p);
            pthread_join(t2, NULL);
        }
        printf("\n");
    }

    sem_destroy(&mutexp);

    //signal
    sem_post(&mutexq);
}
```

그 다음 2중 for 루프를 이용하여 큐 안의 프로세스들이 우선순위가 높은 순서대로 수행 시간만

큼 'runner' thread를 생성하고 종료하는 것을 반복한다.

큐 안의 모든 프로세스들이 수행을 마치면 'mutexp' 변수를 사용하는 semaphore를 제거한 뒤 'mutexq' 변수를 사용하는 semaphore를 signal한다.

(8) Round-Robin scheduler (3번 queue scheduler)

```
//Round-Robin Scheduler(scheduler for 3rd queue)
//args : informations of processes in 3rd queue
void *q3Shed(void *args){
    //wait
    sem_wait(&mutexq);

    //get informations of processes in 3rd queue
    int (*q3)[101][4] = args;

    //array that saves progress of each processes
    int progress[101] = {0, };
    //number of processes in 3rd queue
    int pCnt = 0;
    //number of processes finished
    int finished = 0;

    //get number of processes in 3rd queue
    for(int i = 0; (*q3)[i][0] != 0; i++){
        pCnt++;
    }

    //semaphore for runner threads
    sem_init(&mutexp, 0, 1);
```

'q3Sched'는 3번 queue에서 Round-Robin 스케줄링을 담당하는 Thread이다. 3번 큐에 들어있는 프로세스 정보가 담긴 배열을 인자로 전달받는다. Round-Robin에서 사용되는 time quantum은 1ms이다. 'runner' thread에서 프로세스 id를 한 번 출력하는데 1ms가 걸리므로 스케줄러는 하나의 프로세스가 id를 출력하면 다음 프로세스로 순서를 넘긴다.

'sem_wait(&mutexq)'를 통하여 다른 큐의 스케줄링을 담당하는 thread가 shared resource에 접근하지 못하도록 막아 동기화를 한다.

'progress' 배열은 각 프로세스의 진행 정도를 저장한다. 만약 i번 프로세스가 2번 id를 출력하였다면 progress[i]의 값은 2가 된다. 'pCnt' 변수는 큐 안의 있는 프로세스의 개수를 저장한다. 'finished' 변수는 수행이 완료된 프로세스의 개수를 저장한다.

for 루프를 통해 큐 안의 프로세스가 몇 개 들어 있는지 알아낸다.

다음으로 id를 출력하는 thread들 간의 동기화를 위해 사용하는 semaphore를 생성한다. 이

semaphore는 큐들 간의 동기화를 위해 사용되는 semaphore와는 다르다.

```
    //create threads that print id for each processes
    while(finished < pCnt){
        for(int i = 0; i < pCnt; i++){
            if ((*q3)[i][3] > progress[i]){
                pthread_t t3;

                int pid = (*q3)[i][1];
                int *p = &pid;

                pthread_create(&t3, NULL, runner, p);
                pthread_join(t3, NULL);

                progress[i]++;
                if(progress[i] == (*q3)[i][3]){
                    finished++;
                }
                printf("\n");
            }
        }
    }

    sem_destroy(&mutexp);

    //signal
    sem_post(&mutexq);
}
```

while 루프에서는 큐 안의 프로세스가 한 번씩 번갈아 가며 'runner' thread를 생성하고 끝내는 것을 반복한다. 프로세스가 수행 시간만큼 'runner' thread를 생성하고 끝내는 것을 반복했다면 'finished' 변수의 값이 1 증가하고 해당 프로세스는 그 다음부터 수행되지 않는다.

큐 안의 모든 프로세스들이 수행을 마치면 'mutexp' 변수를 사용하는 semaphore를 제거한 뒤 'mutexq' 변수를 사용하는 semaphore를 signal한다.

(9) Priority-Round-Robin scheduler (4번 queue scheduler)

```
//Priority-Round-Robin Scheduler(scheduler for 4th queue)
//args : informations of processes in 4th queue
void *q4Shed(void *args){
    //wait
    sem_wait(&mutexq);

    //get informations of processes in 3rd queue
    int (*q4)[101][4] = args;

    //array that saves progress of each processes
    int progress[101] = {0, };
    //number of processes in 4th queue
    int pCnt = 0;
    //number of processes finished
    int finished = 0;

    //get number of processes in 4th queue
    for(int i = 0; (*q4)[i][0] != 0; i++){
        pCnt++;
    }
```

'q4Sched'는 4번 queue에서 Priority-Round-Robin 스케줄링을 담당하는 Thread이다. 4번 큐에 들어있는 프로세스 정보가 담긴 배열을 인자로 전달받는다. Priority-Round-Robin에서 사용되는 time quantum은 2ms이다. 'runner' thread에서 프로세스 id를 한 번 출력하는데 1ms가 걸리므로 스케줄러는 하나의 프로세스가 id를 두 번 출력하면 다음 프로세스로 순서를 넘긴다.

'sem_wait(&mutexq)'를 통하여 다른 큐의 스케줄링을 담당하는 thread가 shared resource에 접근하지 못하도록 막아 동기화를 한다.

'progress' 배열은 각 프로세스의 진행 정도를 저장한다. 만약 i번 프로세스가 2번 id를 출력하였다면 progress[i]의 값은 2가 된다. 'pCnt' 변수는 큐 안의 있는 프로세스의 개수를 저장한다. 'finished' 변수는 수행이 완료된 프로세스의 개수를 저장한다.

for 루프를 통해 큐 안의 프로세스가 몇 개 들어 있는지 알아낸다.

```
//bubble sort
//sort id_rt array by priority
for (int i = 0; i < pCnt; i++){
    for (int j = 0; j < pCnt - (i + 1); j++){
        if((*q4)[j][2] > (*q4)[j + 1][2]){
            int temp[4] = {(*q4)[j + 1][0], (*q4)[j + 1][1],
            (*q4)[j + 1][2], (*q4)[j + 1][3]};

            (*q4)[j + 1][0] = (*q4)[j][0];
            (*q4)[j + 1][1] = (*q4)[j][1];
            (*q4)[j + 1][2] = (*q4)[j][2];
            (*q4)[j + 1][3] = (*q4)[j][3];
            (*q4)[j][0] = temp[0];
            (*q4)[j][1] = temp[1];
            (*q4)[j][2] = temp[2];
            (*q4)[j][3] = temp[3];
        }
    }
}

//semaphore for runner threads
sem_init(&mutexp, 0, 1);
```

그 다음 버블 소트를 통해 배열 안의 프로세스들을 우선 순위 순으로 정렬한다. 그리고 id를 출력하는 thread들 간의 동기화를 위해 사용하는 semaphore를 생성한다. 이 semaphore는 큐들 간의 동기화를 위해 사용되는 semaphore와는 다르다.

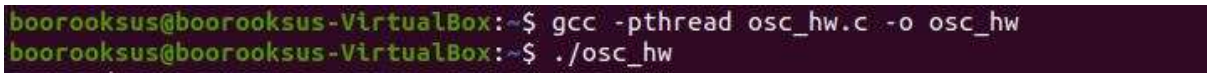
```
    //create threads that print id for each processes
    while(finished < pCnt){
        for(int i = 0; i < pCnt; i++){
            int isChanged = 0;
            for(int j = 0; j < 2; j++){
                if ((*q4)[i][3] > progress[i]){
                    isChanged = 1;
                    pthread_t t4;

                    int pid = (*q4)[i][1];
                    int *p = &pid;

                    pthread_create(&t4, NULL, runner, p);
                    pthread_join(t4, NULL);

                    progress[i]++;
                    if(progress[i] == (*q4)[i][3]){
                        finished++;
                    }
                }
            }
            if (isChanged == 1) printf("\n");
        }
    }

    sem_destroy(&mutexp);

    //signal
    sem_post(&mutexq);
}
```

while 루프에서는 큐 안의 프로세스가 두 번씩 번갈아 가며 'runner' thread를 생성하고 끝내는 것을 반복한다. 프로세스가 수행 시간만큼 'runner' thread를 생성하고 끝내는 것을 반복했다면 'finished' 변수의 값이 1 증가하고 해당 프로세스는 그 다음부터 수행되지 않는다.

큐 안의 모든 프로세스들이 수행을 마치면 'mutexp' 변수를 사용하는 semaphore를 제거한 뒤 'mutexq' 변수를 사용하는 semaphore를 signal한다.

4. 프로그램 실행

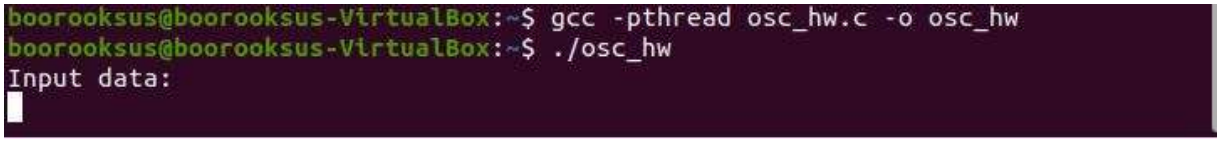
Ubuntu(64-bit) 환경에서 gcc 컴파일러를 이용하여 컴파일 하였다.



gcc -pthread [파일 이름] -o [프로그램 이름]

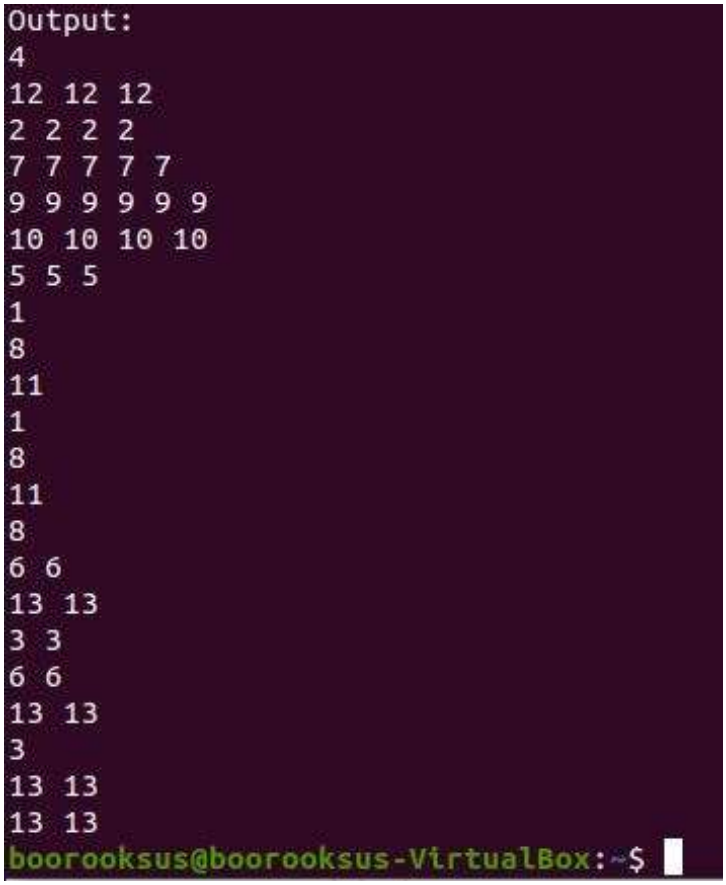
./[프로그램 이름]

위와 같이 입력하여 실행시킨다.



위와 같이 'input data:' 메시지가 뜨면 프로세스 정보들을 직접 입력한다.

프로세스 정보들을 모두 입력하면 마지막에 '0 0 0 0'을 입력하고 엔터키를 누른다.



그 다음엔 'Output:' 메시지와 함께 위와 같이 프로세스 id를 출력한다. 출력된 결과는 각 프로세스의 클래스 번호, 우선순위에 맞게 출력이 된 것을 확인할 수 있다. 여기서 주목할 점은 큐의 클래스 번호가 4인 프로세스인 3번, 6번, 13번이다. 각 프로세스의 우선순위는 6번 > 13번 > 3번이다. 6번, 13번의 실행시간은 각각 5, 8로 상대적으로 긴 편에 속한다. 만약 Priority scheduling을 하였다면 3번 프로세스는 한참 뒤에 6번, 13번 프로세스가 모두 수행되고 나서 수행을 시작한다. 하지만 Priority-Round-Robin Scheduler로 인하여 2ms씩 번갈아 가며 수행되므로 우선순위가 낮은 3번 프로세스도 큐 안에서의 대기 시간이 줄어들었다.

5. 결론

본 보고서에서는 Multi-thread programming을 이용한 프로그램을 만들고 실행해 보았다. 그리고 thread synchronization을 위한 Semaphore를 이용하였다. 또한 Multi-Level Queue, SJF Scheduler, Priority Scheduler, Round-Robin Scheduler를 직접 구현하였다. 그리고 Priority scheduling과 Round-Robin scheduling을 합한 Priority-Round-Robin Scheduler를 개발하여 적용하였다. Priority-Round-Robin Schedule 방식은 프로세스를 골고루 수행하면서 우선 순위가 높은 프로세스를 먼저 수행하는 경향을 보인다. 따라서 우선순위가 높지만 매우 긴 수행시간을 가진 프로세스가 주어지더라도 time quantum 만큼 수행하고 다음 프로세스로 순서가 넘어가기 때문에 프로세스 수행이 정체되지 않는다는 장점을 가진다.

프로그램의 실행 결과는 예상했던 결과대로 각 프로세스의 클래스 번호, 우선 순위, 수행 시간에 따라 순서대로 프로세스의 id가 출력된 것을 확인할 수 있었다.