운영체제 과제 3

201921162 송민기

1. 스케줄링 알고리즘

1.1 스케줄링 알고리즘이란?

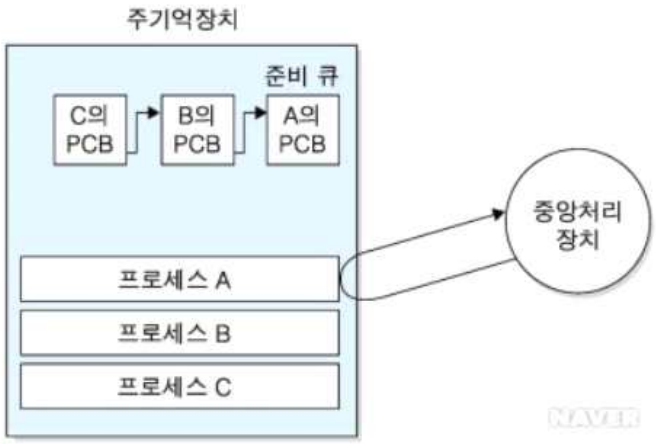
스케줄링이란 메모리에 있는 준비 상태의 프로세스 중 하나를 선택하여 CPU 자원을 할당하는 것을 말한다. CPU는 쉬고 싶어 하지 않기 때문에 최대로 효율을 발휘하고 싶어하는 특징이 있다. CPU 가 일어나는 시점은 다음과 같다.

1. 실행 상태에서 대기 상태로 전환 될 때 - 비선점
2. 실행 상태에서 준비 상태로 전환될 때 - 선점
3. 대기 상태에서 준비 상태로 전환될 때 - 선점
4. 종료될 때 - 비선점

선점 스케줄링이란 한 프로세스가 CPU를 차지 하고 있을 때, 다른 프로세스가 현재의 프로세스를 중단 시키고, CPU를 차지할 수 있는 방법이다. 비선점 스케줄링이란 CPU가 한 프로세스에 할당되면, 프로세스가 종료한다던가, 또는 대기 상태로 전환해 CPU를 해제할 때까지 CPU를 점유하는 방법이다.

1.2 FCFS 스케줄링 알고리즘

FCFS 스케줄링 알고리즘은 비선점형 알고리즘으로 먼저 CPU를 요청하는 프로세스를 먼저 처리하는 방식이다. 프로세스의 요청이 있으면 도착하는 순서대로 CPU에 할당이 된다.



위의 그림을 보면 프로세스가 A,B,C 순서대로 생성이되면 운영체제는 가장 먼저 생성된 프로세스 A에게 CPU를 배정하고 실행되도록 한다. 그리고 프로세스 B와 C는 프로세스 A가 종료될 때 까지 준비 큐에서 기다린다. 프로세스 A의 실행이 종료 되면 다음 프로세스인 B가 실

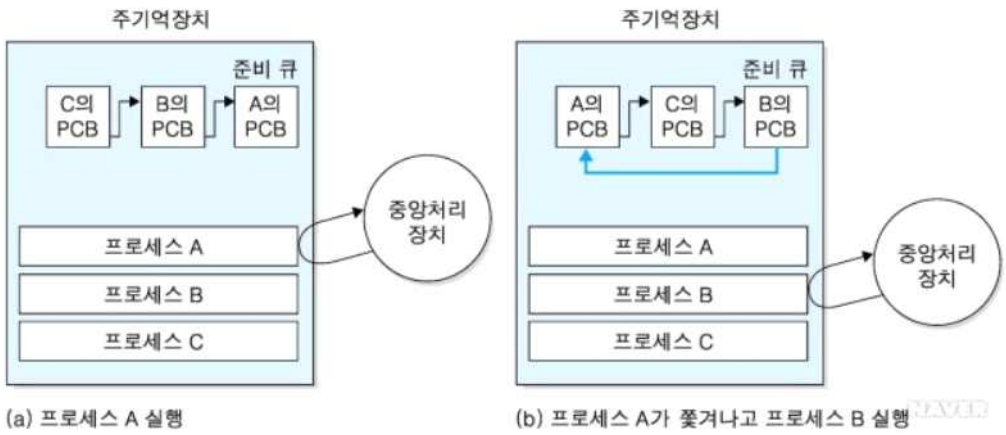
행이된다. FCFS 알고리즘을 이용할 경우 평균 대기 시간을 계산하는 방법은 다음과 같다.

프로세스	중앙처리장치 시간
프로세스 A	20ms
프로세스 B	5ms
프로세스 C	2ms

프로세스 A의 대기시간은 0ms 이고 종료 시간은 20ms 이다.프로세스의 대기시간은 20ms 이고 종료시간은 25ms 이다. 프로세스 C의 대기시간은 25ms 이고 종료 시간은 27 ms 이다. 그러므로 대기 시간의 평균은 (0+20+25)/3 = 15ms 가 된다.

1.3 RR 스케줄링 알고리즘

라운드 로빈 스케줄링은 하나의 cpu를 임의의 프로세스가 종료될 때까지 차지하는 것이 아니라, 여러 프로세스들이 중앙 처리 장치를 조금씩 돌아가며 할당 받아 실행되는 방식이다. 프로세스들은 시간 할당량 동안 cpu를 할당 받아 실행되고, 이 실행 시간 동안 실행을 종료하지 못하면 운영체제에 의해 준비상태로 쫓겨나고, 준비 큐의 다음 프로세스가 중앙처리 장치를 할당 받아 실행된다.



(a) 프로세스 A 실행 (b) 프로세스 A가 쫓겨나고 프로세스 B 실행

cpu를 할당 받은 프로세스가 A에서 B로 바뀔 때 문맥 전환이 이루어지는데, 이런 문맥 전환이 진행되는 동안 CPU가 아무런 일을 하지 못하므로 CPU의 이용률을 저하시킨다.
평균대기시간은 다음과 같이 구할 수 있다.

프로세스	중앙처리장치 시간
프로세스 A	20ms
프로세스 B	5ms
프로세스 C	2ms

1. 프로세스 A가 CPU를 할당받아 실행된다.
2. 프로세스 B와 C가 도착하면 B와 C의 프로세스 제어 블록이 준비 큐에 연결된다.
3. 프로세스 A가 실행을 시작한지 4ms(시간할당량) 가 되면 A가 쫓겨나고 B가 할당을 받게 된다.
4. 또 4ms 가 지나면 B가 쫓겨나고 C가 CPU에 할당을 받게 된다.
5. C는 2ms 이므로 시간할당량이 끝나기 전에 종료되고 A가 실행이 된다.
6. A가 쫓겨나고 B가 실행이되고
7. B의 수행이 끝나고 작업이 종료 될 때까지 A가 실행된다.

A의 대기 시간은 0+6+1= 7 B의 대기시간은 4+6 = 10 C의 대기시간은 8이 되므로 평균 대기시간은 8.3ms 가 된다.

1.4 HRN 스케줄링 알고리즘

짧은 작업에 유리한 SJF의 단점을 개선한 기법으로 각 작업의 우선순위로 서비스를 해주는 스케줄링이다. 오랫동안 대기하는 프로세스의 우선순위를 증가시키는 방법을 이용하는데, (대기시간+서비스시간)/서비스시간 을 통해서 우선순위를 지정한다. 따라서 처음 프로세스가 할당이 종료 되면 레디큐에 준비된 프로세스들끼리의 response ratio를 구하여 이것이 가장 큰 값을 가지는 것이 실행이 된다. 하나의 프로세스가 종료될 때마다 새롭게 갱신한다.

작업	대기시간	서비스시간
A	5	20
B	40	20
C	15	45
D	20	20

A의 우선순위 : (5+20)/20=1.25
B의 우선순위 : (40+20)/20=3
C의 우선순위 ; (15+45)/45 =1.3333
D의 우선순위 : (20+20)/20 =2

우선순위가 작은 프로세스가 실행되고 이후에 다시 우선순위를 정하여 response ratio 가 큰 프로세스를 실행한다.

2. 코드설명 및 실행

2.1 FCFS 스케줄링 알고리즘

FCFS 스케줄링의 주요 코드를 간략하게 구조화하면 다음과 같다.

```
for (time = 0; time < 100; time++)//cpu 할당 시간이 1000ms 라고 가정
    {

        if () 레디 큐에 아무것도 없는데
               존재하는 프로세스 개수 == 종료 프로세스 개수 일 때 종료


        if (만약 실행중인 프로세스가 있을 경우)
        {
            if (만료 시간과 현재시간이 일치할  경우)
             {
               cpu에서 프로세스 해제
            }
        }


        if (만약 실행중인 프로세스가 없을 때)
        {

            if (레디 큐에 프로세스가 있을 경우)
            {

                  if (레디 큐에 두 개 이상 있을 경우)
                  {
                      CPU에 프로세스 할당, 레디 큐 SHIFTING
                  }

                  else if (레디 큐에 한 개 있을 경우)
                  {
                        cpu에 프로세스 할당
                  }
            }

            else
            {
               레디 큐에 아무것도 없을 경우
            }
        }
    }
```

전체적인 코드를 보면 다음과 같다.

```c
typedef struct Process // 구조체 정의
{
    int burstTime;
    int arrivalTime;
    int exitTime; //끝나는 시간
    int pid;      //프로세스 아이디

} Process;

Process *readyQueue[100] = {0};  //
Process *finishQueue[100] = {0}; //
Process *pc[100] = {0};          //
```

```c
//큐 선언
int queueFront = 0;
int queueRear = 0;

void pushToReadyQueue(Process *process)
{
    readyQueue[queueRear++] = process;
}

Process *popFromReadyQueue()
{

    return readyQueue[queueFront++];
}

int isReadyQueueEmpty()
{
    return queueFront == queueRear ? 1 : 0;
}

void readyQueueInit()
{
    int queueFront = 0;
    int queueRear = 0;
}
```

우선 구조체를 선언한다. 구조체의 변수는 실행시간, 도착시간, 종료시간, 프로세스 id를 포함한다. 또한 process 구조체 형식을 가지는 큐인 readyQueue, 마지막에 대기시간을 계산하기 위해 종료 프로세스들을 저장하는 finishQueue, 그리고 input.txt를 읽고 저장하는 pc 배열을 선언하고 초기화 한다. 세 개의 배열은 복사하는 수고를 덜기 위해 포인터를 사용해 참조시켰다. 큐도 선언하였다. readyQueue에 push, pop 하는 기능과 초기화 기능, 큐가 비었는지 안 비었는지 확인하는 기능을 만들었지만, 사실 FCFS 코드에서는 큐에 내용물이 있는지 확인하는 isReadyQuereEmpty 밖에 사용되지 않는다.

```c
FILE *f;

f = fopen("input.txt", "r"); // 파일 읽기

int arrivalTime;
int burstTime;

while (fscanf(f, "%d %d", &burstTime, &arrivalTime) != -1) //input scanf해서 다음에 넣어주기
{
    readyQueue[count] = malloc(sizeof(Process)); //배열들 동적 할당
    finishQueue[count] = malloc(sizeof(Process));
    pc[count] = malloc(sizeof(Process));
    pc[count]->burstTime = burstTime; //구조체를 이용하여 bursttime,arrivaltiem,pid 저장
    pc[count]->pid = count;
    pc[count++]->arrivalTime = arrivalTime;
}

fclose(f); //파일입출력 종료
```

파일 입출력을 통해서 input.txt를 fscanf 하여 burstTime과 arrivalTime을 차례대로 읽어들이고 pc[]에 저장한다. 위에서 선언한 세 개의 배열을 동적할당을 한다. 한 사이클이 끝나면 count+1을 한다. count는 프로세스의 개수를 세는 역할을 한다.

```
int time = 0;
int finishPC = 0;           //종료된 프로세스의 개수
int readyPC = 0;            //레디큐에 존재하는 프로세스의 개수
Process *executePC = NULL; //실행중인 프로세스를 담는 구조체

for (time = 0; time < 1000; time++) //최대의 cpu 동작시간이 1000ms 라고 가정
{

    if (isReadyQueueEmpty() && finishPC == count) //cpu 종료 조건 : 레디큐에
    {
        printf("cpu 동작 종료\n\n");
        break;
    }
```

finishPc는 종료한 프로세스의 개수를 카운트하고 이것은 for문의 종료조건을 확인할 때 이용된다. readyPc는 준비큐에서 대기하고 있는 프로세스의 개수를 카운트하고 스케줄링을 진행할 때 필요한 정보이다. for문을 통해 스케줄링이 진행되고 최대 cpu 동작시간을 1000ms 라고 가정하였다. 만약 레디큐가 비어있는 상태에서 종료된 프로세스 개수와 전체 프로세스의 개수가 일치할 경우 for문이 종료된다.

```
if (pc[0] != NULL) //만약 실행할 프로세스가 아직 남아있을 때
{
    if (time == pc[0]->arrivalTime) //그 프로세스의 도착시간이 현재
    {

        //input.txt의 프로세스들의 정보가 arrivalTime이 현재 시간일
        if (readyPC == 0) //만약 레디큐에 존재하는 프로세스개수가 0일
        {
            readyQueue[0] = pc[0];
        }
        else //레디큐에 존재하는 프로세스가 있을 때, 레디큐의 가능한
        {
            readyQueue[readyPC] = pc[0];
        }
        readyPC++; //레디큐에 프로세스가 증가하므로 +1

        for (int i = 0; i < count; i++)
        {
            if (pc[i + 1] == NULL || i == count - 1) //만약 pc[i+
            {

                pc[i] = NULL;
            }
            else //그렇지 않다면 pc의 값을 왼쪽으로 쉬프팅 해준다.
            {
                pc[i] = pc[i + 1];
            }
        }
    }
}
```

만약 실행할 프로세스가 아직 남아있고 남아있는 프로세스의 도착시간이 현재 시간과 같을 때(=현재 도착한다면) 레디큐에 pc[]에 있는 프로세스를 push 해주고, pc[] 앞부분을 비우게 되므로 왼쪽으로 shifting해준다.

```
if (executePC != NULL) //만약 실행중인 프로세스가 있을 경우
{
    //만료 시간과 현재시간이 일치할 경우 cpu에서 프로세스 해제

    if (executePC->exitTime == time)
    {
        printf("time : %d\n프로세스%d 종료\n\n", time, executePC->pid);
        finishQueue[finishPC] = executePC;
        executePC = NULL;
        finishPC++;
    }
}
```

만약 실행중인 프로세스가 있을 경우, 프로세스의 종료시간이 현재시간과 같아질 때(=종료될 때) 프로세스를 cpu에서 해제하고 finishQueue로 이동시킨다.

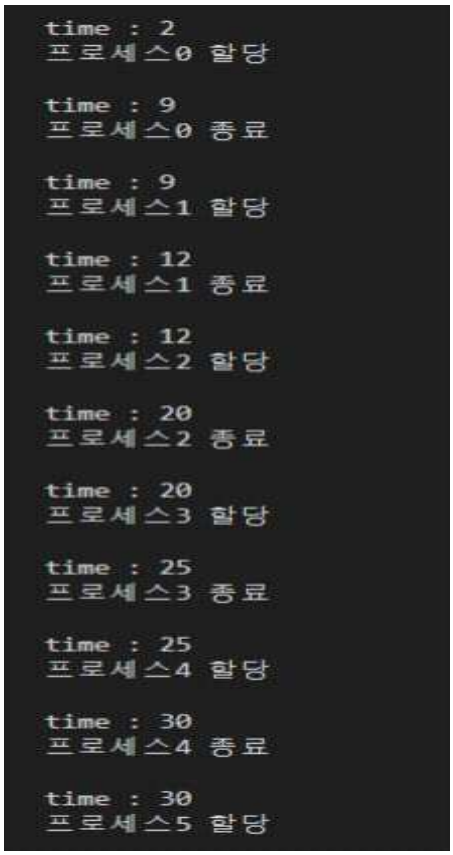
```
if (executePC == NULL)
{
    //레디큐에 뭐가 있을 경우
    if (readyPC != 0)
    {

        if (readyPC > 1) //레디큐에 존재하는 프로세스가 한개 이상일  경우
        {

            executePC = readyQueue[0]; //프로세스를 실행시킨다.

            for (int i = 0; i < readyPC; i++) //레디큐의 항목들을 쉬프팅해준다.
            {
                if (i == readyPC - 1)
                {
                    continue;
                }
                else
                {
                    readyQueue[i] = readyQueue[i + 1];
                }
            }
            readyPC--;          //레디큐 개수 -1
            executePC->exitTime = executePC->burstTime + time; //프로세스의 종
            printf("time : %d\n프로세스%d 할당\n\n", time, executePC->pid);
        }

        else if (readyPC == 1) //만약 레디큐에 존재하는 프로세스가 한 개일 경우
        {                       //레디큐에 하나 있을경우 그냥 cpu 에 할당
            executePC = readyQueue[0];
            readyQueue[0] == NULL;
            readyPC--;
            executePC->exitTime = executePC->burstTime + time;
            printf("time : %d\n프로세스%d 할당\n\n", time, executePC->pid);
        }
    }
```

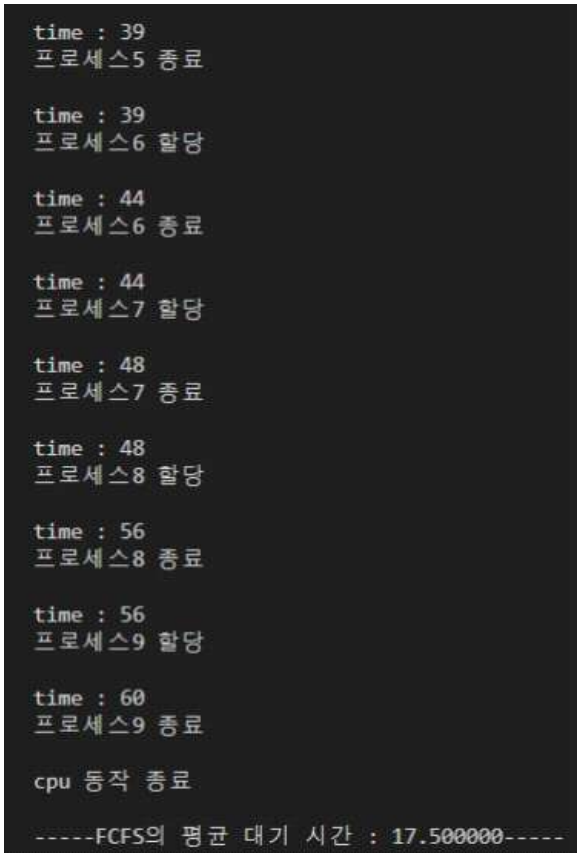
그 이후 만약 실행프로세스가 없고, 레디 큐에는 프로세스가 있고 없는 경우로 나눌 수 있다. 만약 레디 큐에 프로세스가 있으면 프로세스가 1개 있는 경우와 1개 이상 있는 경우로 나눌 수 있다. 이렇게 케이스를 나누는 이유는 레디큐에 1개 이상 프로세스가 있는 경우 내용물들을 shifting 해줄 필요가 있고, 1개만 있으면 해줄 필요가 없기 때문이다. 레디 큐에 프로세스가 1개 이상 있는 경우 레디큐 제일 앞에 있는 프로세스를 실행시키고 레디 큐의 0번째가 비게 되므로 왼쪽으로 shifting 해준다. 그리고 exitTime = 실행시간 + 현재 시간으로 지정해준다. 프로세스가 1개인 경우는 그냥 프로세스를 실행시키기만 하면 된다.

만약 레디 큐에 프로세스가 없으면 그대로 다음 for문을 진행하면 된다.

```
for (int i = 0; i < count; i++) //각 프로세스의 종료시간 - 실행시간- 도착시간으로 대기시간을 구함.
{
    sum = sum + (finishQueue[i]->exitTime - (finishQueue[i]->burstTime) - (finishQueue[i]->arrivalTime));

    // printf("%d\n", finishQueue[i]->exitTime);
}

printf("-----FCFS의 평균 대기 시간 : %f-----\n\n", (float)(sum) / (float)(count));

free(*pc);
free(*finishQueue);
free(*readyQueue);
```

각 프로세스의 대기시간은 종료시간-실행시간-도착시간으로 나타낼 수 있다. 이 부분의 코드의 간소화를 위하여 구조체에서 종료시간, 실행시간, 도착시간 변수를 선언하였다. finishQueue에 존재하는 프로세스들의 정보에 접근하여 쉽고 빠르게 불러올 수 있다.





./fcfs를 통해서 프로그램을 실행시킨 결과이다. 프로세스가 할당, 종료 되는 시간과 프로세스의 pid를 같이 출력하도록 하였다. FCFS 답게 차례대로 프로세스들이 할당되는 것을 확인할 수 있다. 결과적으로 FCFS의 평균 대기 시간은 17.5ms 가 나온다.

2.2 RR 스케줄링 알고리즘

RR 스케줄링의 주요 코드를 간략하게 구조화하면 다음과 같다.

```
for (time = 0; time < 1000; time++) //cpu 할당 시간이 1000ms 라고 가정
    {

        if () 레디 큐에 아무것도 없는데
              존재하는 프로세스 개수 == 종료 프로세스 개수 일 때 종료


        if (실행중인 cpu가 없는 경우)
        {
            if (만약 레디 큐가 비어있지 않은 경우)
            {
               cpu에 프로세스 할당
            }
            else 만약 레디 큐가 비어있으면 continue;
            {

                continue;
            }
        }

        else 실행중인 cpu가 있는 경우
        {
            if (만약 실행 프로세스가 시간할당량을 채웠을 경우)
            {
                if (실행시간이 안남은 경우//완료)
                {
                      프로세스로 종료

                    if (레디 큐 한 개 이상 있으면)
                    {
                       프로세스로 실행
                    }
                }
                else  시간 할당량을 채웠는데, 실행시간이 남은 경우
                {
                    다시 레디 큐로 PUSH
                    if (레디 큐 한 개 이상 있으면)
                    {
                      프로세스로 실행
                    }
                }
            }
            else    만약 시간 할당량을 채우지 못했을 경우
            {

                if (실행시간을 채운 경우)
                {
                    프로세스로 종료

                    if (레디 큐 한 개 이상 있으면)
                    {
                      프로세스로 실행
                    }
                }
                else 시간 할당량 못 채우고 실행시간도 못 채운경우 실행 중,,,
                {
                  실행 중
                }
            }
        }
    }
```

FCFS와 구조는 비슷하므로 달라진 부분만 보자.

```
typedef struct Process //구조체 선언
{
    int burstTime;
    int arrivalTime;
    int exitTime;
    int pid;
    int timer;          //시간 할당량
    int serviceTime;    // 현재까지의

} Process;

Process *readyQueue[100] = {0};
Process *finishQueue[100] = {0};
Process *pc[100] = {0};
```

구조체를 선언한다. FCFS와 다른점은 timer, serviceTime이 추가된 것이다. timer는 시간할당량이 어느 정도 할당되었는지, servicetime은 실행시간을 만족했는지를 확인하기 위해 사용한다.

```
if (executePC == NULL) //실행중인 cpu가 없는경우
{
    if (!isReadyQueueEmpty()) //만약 레디큐가 비어있지 않은 경우
    {
        executePC = popFromReadyQueue();    //프로세스 실행
        printf("time : %d\n프로세스%d 할당\n\n", time, executePC->pid);

        executePC->timer += 1;  //실행중인 프로세스 타이머 +1
        executePC->serviceTime += 1; //실행 프로세스 현재까지의 실행시간 +1
    }
    else    //만약 레디큐가 비어있으면 continue;
    {

        continue;
    }
}
```

스케줄링 알고리즘 부분이다. 만약 실행프로세스가 없고 레디 큐에 프로세스가 있으면 프로세스를 할당한다. 타이머의 시간+1, 해당 프로세스가 현재까지 진행한 실행시간 serviceTime+1을 한다. 레디큐가 비어있으면 실행할 프로세스가 없기 때문에 CONTINUE를 하여 다음 사이클을 돌린다.

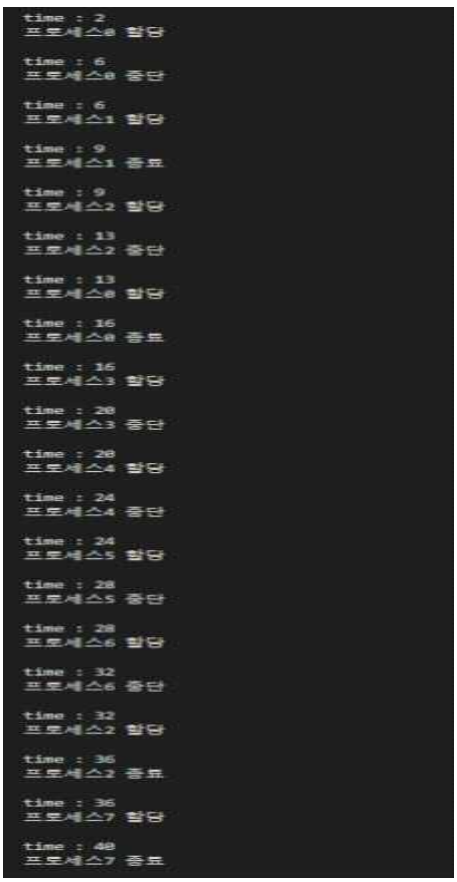
```
else //실행중인 cpu가 있는경우
{
    if (executePC->timer == 4)  //만약 실행 프로세스가 시간할당량을 채웠을 경우
    {
        if (executePC->burstTime - executePC->serviceTime == 0) //실행시간-현재까
        {
            printf("time : %d\n프로세스%d 종료\n\n", time, executePC->pid);
            executePC->timer = 0;   //타이머 초기화
            executePC->exitTime = time; //프로세스 종료시간 지정
            finishQueue[finishPC] = executePC; //끝난 프로세스로 저장

            executePC = NULL;   //실행 프로세스 없음
            finishPC++;         //끝난 프로세스 개수 +1
            if (!isReadyQueueEmpty())
            {                                   //레디큐 있으면 실행
                executePC = popFromReadyQueue(); //기다리는 프로세스 실행
                executePC->timer += 1;          //타이머 +1
                printf("time : %d\n프로세스%d 할당\n\n", time, executePC->pid);
                executePC->serviceTime += 1;     //현재까지의 실행시간 +1
            }
        }
        else  //시간 할당량을 채웠는데, 실행시간이 남은 경우
        {
            printf("time : %d\n프로세스%d 중단\n\n", time, executePC->pid);
            executePC->timer = 0;   //타이머 초기화
            pushToReadyQueue(executePC);    //다시 레디큐로 PUSH
            if (!isReadyQueueEmpty())
            {                                   //레디큐 있으면 실행
                executePC = popFromReadyQueue(); //기다리는 프로세스 실행
                executePC->timer += 1;
                printf("time : %d\n프로세스%d 할당\n\n", time, executePC->pid);
                executePC->serviceTime += 1;
            }
        }
    }
```

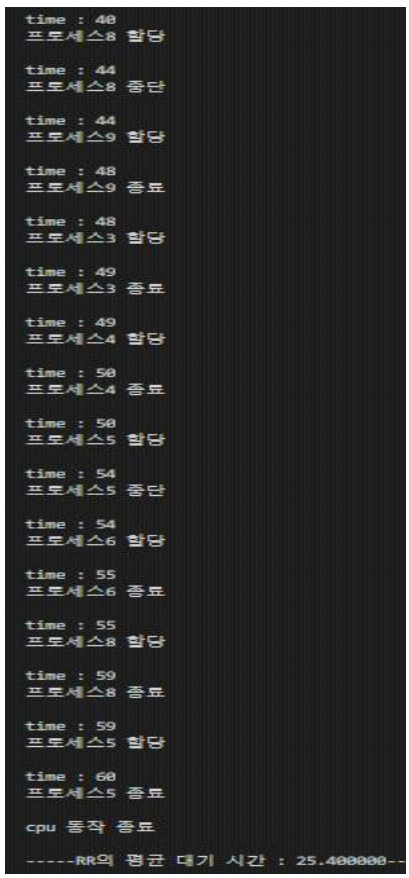
만약 실행중인 프로세스가 있을 경우, 시간할당량 4초 동안 실행했고, 실행시간이 만료되었을 경우 프로세스를 해제하고 finishQueue에 삽입한다. 레디 큐가 비어있지 않을 경우 다음 프로세스를 실행한다. 만약 실행시간이 아직 남았다면 잠시 프로세스를 중단하고 레디 큐에서 대기하는 다음 프로세스를 실행한다.

```
        else    //만약 시간 할당량을 채우지 못했을 경우
        {

            if (executePC->burstTime - executePC->serviceTime == 0) //실행시간을 채운
            {
                printf("time : %d\n프로세스%d 종료\n\n", time, executePC->pid);

                executePC->timer = 0;   //타이머 초기화
                executePC->exitTime = time; //종료시간 지정
                finishQueue[finishPC] = executePC;  //종료 프로세스로 저장

                executePC = NULL;   //실행중인 프로세스 없음
                finishPC++; //종료 프로세스 개수 +1

                if (!isReadyQueueEmpty())
                {                                   //레디큐 있으면 실행
                    executePC = popFromReadyQueue(); //기다리는 프로세스 실행
                    executePC->timer += 1;
                    printf("time : %d\n프로세스%d 할당\n\n", time, executePC->pid);
                    executePC->serviceTime += 1;
                }
            }
            else //시간 할당량 못채우고 실행시간도 못채운경우 실행중,,,
            {
                executePC->timer += 1;
                executePC->serviceTime += 1;
            }
        }
    }
}
```

만약 시간 할당량 4초를 채우지 못했는데, 실행시간이 만료되면 프로세스를 종료하고 finishQueue에 삽입하고 레디 큐에서 대기 중인 프로세스를 실행한다. 만약 그렇지 않다면 그대로 프로세스 실행을 진행한다.

코드를 실행한 결과 평균대기 시간은 25.4가 나왔다.





2.3 HRN 스케줄링 알고리즘

HRN 스케줄링의 주요 코드를 간략하게 구조화하면 다음과 같다.

```
for (time = 0; time < 100; time++)//cpu 할당 시간이 1000ms 라고 가정
    {

        if () 레디 큐에 아무것도 없는데
               존재하는 프로세스 개수 == 종료 프로세스 개수 일 때 종료


        if (만약 실행중인 프로세스가 있을 경우)
        {
            if (만료 시간과 현재시간이 일치할  경우)
            {
               cpu에서 프로세스 해제
            }
        }


        if (만약 실행중인 프로세스가 없을 때)
        {

            if (레디 큐에 뭐가 있을 경우)
            {

                  if (레디 큐에 두 개 이상 있을 경우)
                  {
                        우선순위를 비교해서 우선순위가 높은 순서대로 해서
                        제일 처음의 프로세스를 실행.
                  }

                  else if (레디 큐에 한 개 있을 경우)
                  {
                        cpu에 그냥 할당
                  }
            }

            else
            {
               레디 큐에 아무것도 없을 경우
            }
        }
    }
```

FCFS와 구조는 비슷하므로 달라진 부분만 보자.

```
typedef struct Process //구조체 선언
{
    int burstTime;
    int arrivalTime;
    int exitTime;
    int pid;
    float timeRatio;    //우선순위를

} Process;
```

우선 구조체에서 우선순위를 결정하는 Response Ratio를 저장하는 변수가 추가되었다.

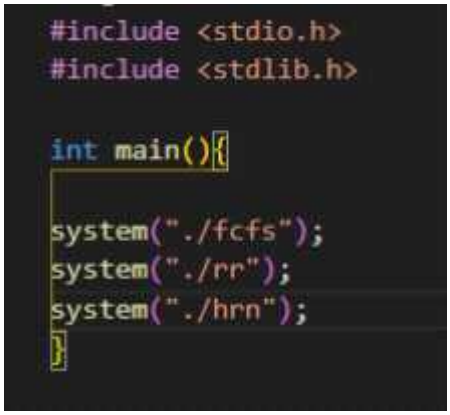
```
if (executePC == NULL)
{
    //레디큐에 뭐가 있을 경우
    if (readyPC != 0)
    {

        if (readyPC > 1)
        { // 레디큐에 두개이상 있을경우 우선순위를 비교해서 우선순위가 높은 순서대로 해서 제일 처음의 것을 뽑아낸다.

            for (int i = 0; i < readyPC; i++) // 비율 구하기  // hrn 우선 순위 = (대기시간 + 서비스 시간)/서비스 시간
            {

                readyQueue[i]->timeRatio = (float)(time - readyQueue[i]->arrivalTime + readyQueue[i]->burstTime) / (float)(readyQueue[i]->burstTime);
            }

            for (int i = 0; i < readyPC; i++) //비율 높은 순으로 정렬
            {
                for (int j = 0; j < readyPC - 1; j++)
                {

                    if (readyQueue[j]->timeRatio < readyQueue[j + 1]->timeRatio)
                    {
                        temp = readyQueue[j];
                        readyQueue[j] = readyQueue[j + 1];
                        readyQueue[j + 1] = temp;
                    }
                }
            }

            executePC = readyQueue[0]; //제일 높은 값을 가진, 제일앞에 위치하는 프로세스 실행
```

FCFS 과 다르게 HRN은 우선순위가 있기 때문에 우선순위를 결정하고 정렬하는 코드가 추가되었다. 만약 실행중인 프로세스가 없는데, 레디 큐에 대기하는 프로세스가 있으면 그 프로세스들끼리의 R.R을 구하여 가장 높은 R.R을 가지는 프로세스를 배열의 제일 앞으로 오도록 정리한 후 0번째 프로세스를 실행한다.

코드를 실행한 결과 평균 시간은 15.1이 나왔다.

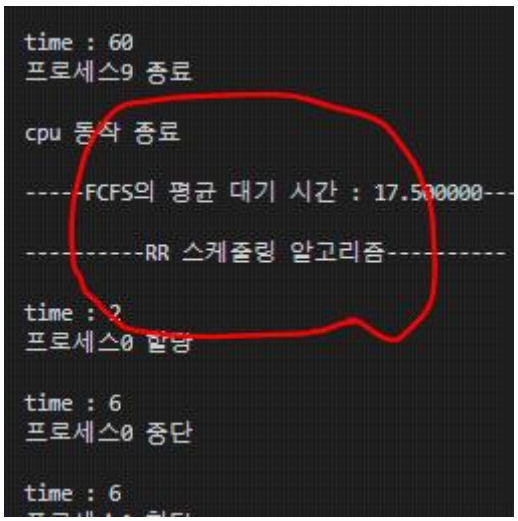
```
time : 2
프로세스0 할당

time : 9
프로세스0 할당

time : 9
프로세스1 할당

time : 12
프로세스1 할당

time : 12
프로세스3 할당

time : 17
프로세스3 할당

time : 17
프로세스4 할당

time : 22
프로세스4 할당

time : 22
프로세스2 할당

time : 30
프로세스2 할당

time : 30
프로세스6 할당

time : 35
프로세스6 할당

time : 35
프로세스7 할당

time : 39
프로세스7 할당

time : 39
프로세스9 할당

time : 43
프로세스9 할당

time : 43
프로세스5 할당

time : 52
프로세스5 할당

time : 52
프로세스8 할당

time : 60
프로세스8 할당

cpu 동작 종료

-----HRN의 평균 대기 시간 : 15.100000-----
```

2.4 scheduling.c

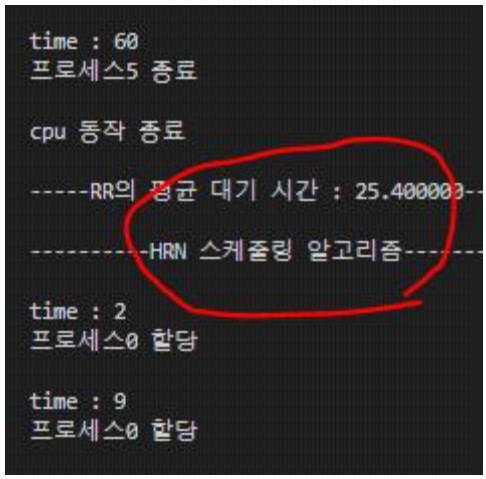


```c
#include <stdio.h>
#include <stdlib.h>

int main(){

system("./fcfs");
system("./rr");
system("./hrn");
}
```

각 파일을 실행시킬 때 마다 ./(파일명) 을 입력해야하는 번거로움이 있다. scheduling.c 는 따로 작성된 fcfs.c, rr.c, hrn.c 파일을 묶어서 한 번에 실행시켜주는 c파일이다. system("./")을 이용하기 때문에 make를 통해 만들어진 scheduling을 ./scheduling을 통해 실행시키면 여러 번 커맨드 창에 ./ 해 줄 필요 없이 시작하자마자 세 개의 파일이 차례대로 한 번에 실행이 된다.



```
time : 60
프로세스9 종료

cpu 동작 종료

-----FCFS의 평균 대기 시간 : 17.500000---

----------RR 스케줄링 알고리즘----------

time : 2
프로세스0 할당

time : 6
프로세스0 중단

time : 6
```

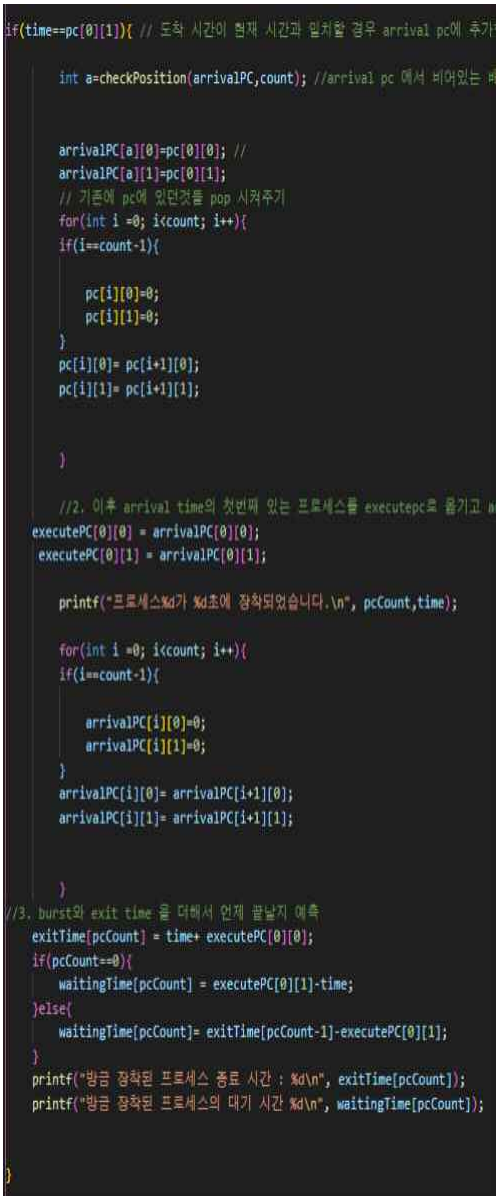


```
time : 60
프로세스5 종료

cpu 동작 종료

-----RR의 평균 대기 시간 : 25.400000--

----------HRN 스케줄링 알고리즘---------

time : 2
프로세스0 할당

time : 9
프로세스0 할당
```

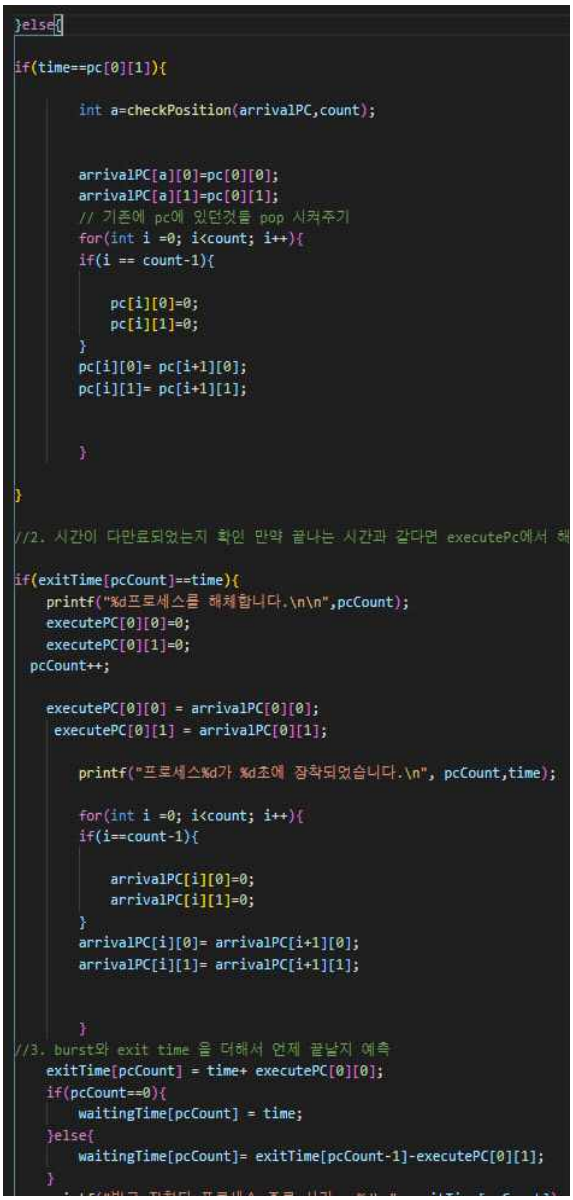
3. 개발 HISTORY 및 소감

이번 과제는 대학에서 한 프로그래밍 과제 중(팀플 과제 제외) 가장 오랜 시간을 할애한 과제였다. 그 이유는 처음 코드를 설계할 때, 세밀하게 계획하지 않고 코드를 작성했고, 자료구조를 이용하지 않았기 때문이다. 하지만 그만큼 과제를 하면서 얻은 것은 많은 것 같다.

초기에는 FCFS 코드를 큐나, 구조체를 이용하지 않고 단지 배열만을 이용하여 코드를 작성하였다.



```
if(time==pc[0][1]){ // 도착 시간이 현재 시간과 일치할 경우 arrival pc에 추가

        int a=checkPosition(arrivalPC,count); //arrival pc 에서 비어있는 

        arrivalPC[a][0]=pc[0][0]; //
        arrivalPC[a][1]=pc[0][1];
        // 기존에 pc에 있던것들 pop 시켜주기
        for(int i =0; i<count; i++){
        if(i==count-1){

            pc[i][0]=0;
            pc[i][1]=0;
        }
        pc[i][0]= pc[i+1][0];
        pc[i][1]= pc[i+1][1];


        }

        //2. 이후 arrival time의 첫번째 있는 프로세스를 executepc로 옮기고 
    executePC[0][0] = arrivalPC[0][0];
     executePC[0][1] = arrivalPC[0][1];

        printf("프로세스%d가 %d초에 장착되었습니다.\n", pcCount,time);

        for(int i =0; i<count; i++){
        if(i==count-1){

            arrivalPC[i][0]=0;
            arrivalPC[i][1]=0;
        }
        arrivalPC[i][0]= arrivalPC[i+1][0];
        arrivalPC[i][1]= arrivalPC[i+1][1];


        }
//3. burst와 exit time 을 더해서 언제 끝날지 예측
    exitTime[pcCount] = time+ executePC[0][0];
    if(pcCount==0){
        waitingTime[pcCount] = executePC[0][1]-time;
    }else{
        waitingTime[pcCount]= exitTime[pcCount-1]-executePC[0][1];
    }
    printf("방금 장착된 프로세스 종료 시간 : %d\n", exitTime[pcCount]);
    printf("방금 장착된 프로세스의 대기 시간 %d\n", waitingTime[pcCount]);


}
```



```
}else{

if(time==pc[0][1]){

        int a=checkPosition(arrivalPC,count);


        arrivalPC[a][0]=pc[0][0];
        arrivalPC[a][1]=pc[0][1];
        // 기존에 pc에 있던것들 pop 시켜주기
        for(int i =0; i<count; i++){
        if(i == count-1){

            pc[i][0]=0;
            pc[i][1]=0;
        }
        pc[i][0]= pc[i+1][0];
        pc[i][1]= pc[i+1][1];


        }

}

//2. 시간이 다만료되었는지 확인 만약 끝나는 시간과 같다면 executePc에서 해

if(exitTime[pcCount]==time){
    printf("%d프로세스를 해체합니다.\n\n",pcCount);
    executePC[0][0]=0;
    executePC[0][1]=0;
  pcCount++;

    executePC[0][0] = arrivalPC[0][0];
     executePC[0][1] = arrivalPC[0][1];

        printf("프로세스%d가 %d초에 장착되었습니다.\n", pcCount,time);

        for(int i =0; i<count; i++){
        if(i==count-1){

            arrivalPC[i][0]=0;
            arrivalPC[i][1]=0;
        }
        arrivalPC[i][0]= arrivalPC[i+1][0];
        arrivalPC[i][1]= arrivalPC[i+1][1];


        }
//3. burst와 exit time 을 더해서 언제 끝날지 예측
    exitTime[pcCount] = time+ executePC[0][0];
    if(pcCount==0){
        waitingTime[pcCount] = time;
    }else{
        waitingTime[pcCount]= exitTime[pcCount-1]-executePC[0][1];
    }
```

처음에 작성했었던 FCFS들의 주요 코드이다. 최종 코드는 구조체를 이용하여 burstTime, waitingTime 등 프로세스의 정보를 담는 반면에 초기 코드는 burstTime 과 waitingTime 등을 배열로 선언하여 저장하였다. 또한 프로세스를 이차원 배열로 선언하여 burstTime 과 waitingTime을 담도록 하였다. 이 방법으로 FCFS의 구현을 성공하긴 했지만, 같은 프로세스의 정보들이 다른 배열에 담겨서 따로따로 놀기 때문에 대기시간을 구하거나 프로세스를 쉬프팅, 할당, 종료 해주는 코드를 작성하는 데에 큰 어려움이 있었다. 또한 라운드 로빈 스케줄링 알고리즘은 burstTime, arrivalTime 말고도 serviceTime, timequatam 같은 정보를 프로세스가 내포하고 있기 때문에 배열을 더 선언하기에는 코드의 작성 효율이 매우 떨어진다고 생각하여 코드를 처음부터 작성하였다. 우선 구조체를 선언하고 구조체의 변수로 burstTime, arrivalTime, 종료시간 변수를 선언하였다. 따라서 마지막에 대기시간을 구할 때 편하게 구할 수 있도록 하였다. 또한 프로세스를 담는 배열들은 모두 참조 처리해두었기 때문에 변수에 접근하기 편했다. 원래 포인터가 어려워서 잘 사용하지 않는 편인데, 이번 기회에 포인터와 구조체의 편리함에 대해서 크게 깨달았다. 여러 자료구조를 이용하니 최종적으로 최종코드가 초

기 코드에 2배가 가량 줄어든 것을 확인할 수 있었다.
그리고 이번 과제의 중요한 핵심은 조건문이었다고 생각한다. 왜냐하면 스케줄링 알고리즘은 프로세스의 할당과 종료를 관리해주는 알고리즘이므로, 할당과 종료 조건을 만족할 경우 실행이 진행되기 때문이다. 그리고 if 문 때문에 가장 많은 시간을 할애하였다. RR 알고리즘은

if(cpu가 비었을 때) {프로세스할당}
else(cpu에 프로세스가 있을 때)

로 나누어 조건문을 작성하였다. 처음에는 HRN 도 그렇게 작성하였는데, 후에는

if(cpu에 프로세스가 있을 때) {프로세스 할당}
if(cpu가 비었을 때)

같이 코드를 작성하였다. 이렇게 하니 실행중인 프로세스가 끝이 나면 새 프로세스를 할당해줘야 하는 여러 번 진행해야할 작업을 앞에서 한 번에 해결해주면서, 코드가 간소화 되는 효과를 얻을 수 있었다. 그리고 FCFS에서 Response Ratio를 구하고 높은 순서대로 정렬하는 알고리즘만 추가하면 HRN이라는 사실을 알고 HRN을 모두 작성한 후에 R.R 구하는 코드와, 정렬 코드만 삭제하여 초기 FCFS 알고리즘에서 새 FCFS 알고리즘으로 수정할 수 있었다.

유독 이번 과제에서 디버깅 작업을 많이 하였는데, 계속 segment fault 에러가 뜨거나 종료된 0프로세스만 계속 할당되거나 대기 시간이 - 가 뜨는 일이 발생했기 때문이다. 우선 segment fault는 프로그램이 허용되지 않은 메모리 영역에 접근을 시도하거나, 허용되지 않은 방법으로 메모리 영역에 접근을 시도할 경우 발생한다고 한다. 코드에서 포인터를 이용하고 배열을 이용하는 과정에서 오류가 발생하는 것이라 생각하여 그 부분을 집중적으로 디버깅하였다. 대부분이 null인 배열의 부분이 언급 될 때 에러가 발생한다는 것을 확인할 수 있었다. 또한 한 프로세스가 계속해서 할당되는 이유는 배열에서 제일 앞에 있는 프로세스가 pop 될 때마다 shifting이 이루어지지 않았기 때문이었다. 따라서 shifting 코드를 이용하여 문제를 해결할 수 있었다. 또한 프로세스가 차례로 도착하는 기존의 예제와는 프로세스들이 띄엄띄엄 도착해 도착시간에 갭이 있어 RR 알고리즘을 작성할 때, 어려움이 있었다. 기존 예제들의 답을 보고 P0->P1->P2->P3->P0->P1->P2->P3 이런 식으로 실행될 것이라고 착각하고 있었기 때문에, P0->P1->P0->P2->P3->P1->P2의 순서 진행이 가능하다는 것을 간과하고 있었다. 깨달은 후에서야 제대로 된 코드를 작성할 수 있었다.

작성된 코드들을 보면 FCFS와 HRN은 비슷한데, RR은 전체적으로 다른 것을 확인할 수 있다. 이상하게 RR에서는 에러가 나지 않는데, FCFS와 HRN에서는 오류가 발생하여서 코드를 다르게 작성하였다. 나는 이것이 처음에 알고리즘을 설계할 때 꼼꼼하게 안하였기 때문이라고 생각하였다. 만약 기본인 FCFS를 완벽하게 설계했더라면 RR, HRN 모두 조금씩만 코드를 변형, 추가해서 쉽고 빠르게 코드를 작성할 수 있었을 것이라고 생각한다. FCFS와 HRN 방식과 같이 간단화 시키고 싶었지만, 기간 내에 내가 할 수 있는 최선이라 생각하고 제출하였다. 다음 의 세 가지 알고리즘이 기본적으로 같은 FORM을 갖고 응용된다는 것을 알았기 때문에 후에 스케줄링 알고리즘을 구성할 때는 쉽게 할 수 있을 것 같다!