0. 목표

- 4개의 Ready queue를 갖는 MFQ 스케줄링 기법 구현

(Q0 : time quntum 2인 RR 스케줄링 기법, Q1 : time quntum 4인 RR 스케줄링 기법,

Q2 : time quntum 8인 RR 스케줄링 기법, Q3 : FCFS 스케줄링 기법)

1. 규칙

- 각 프로세스는 최초에 지정된 queue로 진입함
- Q(i)에서 스케줄 받아 실행하고 해당 queue의 time quntum을 소모한 경우에는 Q(i+1)로 진입
- Q(i)에서 온 프로세스가 IO burst를 마치고 wake up 되는 경우에는 Q(i-1)로 진입
- Q(3)에서 온 프로세스는 항상 Q(3)으로 진입
- 우선순위: Q(0) > Q(1) > Q(2) > Q(3)

2. 설계

- Process의 정보를 가지는 구조체 생성
- Ready queue의 자료구조 queue구현 (push, top, pop, empty)
- wait queue와 IO queue의 자료구조 priority queue 구현 (push, top, pop)
- 알고리즘

1) input.txt 파일에서 입력을 받아서 프로세스를 만든 후, wait queue로 push

2) wait queue는 프로세스의 arrival time을 기준으로 하는 MIN HEAP 구조

3) while loop를 돌면서 1 loop가 1 time을 의미

4) wait queue에서 arrival time이 현재 time과 같은 프로세스를 ready queue로 push

5) ready queue 0부터 3까지 차례대로 대기중인 프로세스가 있는지 확인 후, 있다면 pop후 해당 ready queue의 time quntum만큼 cpu 사용

6) cpu를 사용하고 있는 프로세스의 cpu burst time을 loop 돌때마다 1씩 감소

7) 0이 되면 sleep이므로 IO queue로 push

8) IO queue에 있는 프로세스들의IO time을 loop 돌때마다 1씩 감소

9) IO time이 0이 되면 wake up이므로 원래 있던 ready queue의 -1 번째로 push

10) 모든 프로세스의 실행이 끝날 때까지 4~9 반복

3. 구현에 앞선 가정

- 입력으로 주어지는 프로세스의 cycle, BT, IO는 0이 아닌 값을 갖는다.
- 입력으로 주어지는 프로세스의 cycle은 최대 15의 값을 가질 수 있다.
- 입력으로 주어지는 프로세스의 PID는 1부터 시작하며 연속적인 값을 갖는다.

  (ex. #P: 3, PID : 3, 1, 2 가능 / #P: 3, PID : 1, 3, 4 불가능)
- 입력으로 주어지는 프로세스의 한 cycle의 burst time은 100000 이하의 값을 갖는다.
- 입력으로 주어지는 프로세스의 최대 개수는 1000개로 제한한다.
- 모든 프로세스가 스케줄링 받아 실행을 마칠 때까지 걸리는 시간은 100000으로 제한한다.
- ready queue 3에서 IO로 sleep한 경우 wake up하면 다시 ready queue 3으로 push한다.
- IO queue에서 남은 IO burst time이 같은 프로세스들이 있다면 wake up 되는 순서는 MIN HEAP 구조에서 index가 낮은 순서대로 한다.

4. 구현

```
typedef struct process {
    int PID;
    int AT;
    int QUEUE_NUM;
    int index;
    int max_index;
    int BT[31];
    int IO[30];
    int SUM;
}process;
```

PID : 프로세스 ID

AT : wait queue에서 ready queue로 가게 될 시간

QUEUE_NUM : 최초 진입 ready queue

index : 현재 몇 번째 cycle인지 확인용

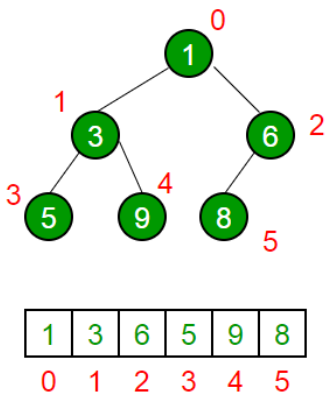
max_index : 마지막 cycle 확인용

BT, IO : Cpu Burst time , IO Burst time

SUM : BT와 IO의 총합

Ready queue는 자료 구조 queue로 구현했는데 사진은 생략.

Wait queue와 IO queue는 MIN HEAP 구조의 priority queue로 구현.



- 완전 이진트리

- 기본적인 array에서 index 0가 최상단 노드를 의미.

- i번째 노드의 자식 노드는 i * 2 + 1 번째 노드와 i * 2 + 2 번째 노드가 된다.

- i번째 노드의 부모 노드는 (i – 1) / 2 번째 노드가 된다.

- 부모 노드는 항상 자식 노드보다 작거나 같다.

이러한 자료 구조를 c코드로 작성하면 아래와 같다.

Wait queue의 비교 기준은 프로세스의 AT이고, IO queue의 비교 기준은 프로세스의 현재 스케줄링 될 cycle의 IO time 기준이다. 비교 기준만 다르고 메커니즘은 같기 때문에 IO queue는 생략하도록 하겠다.

```
void wait_q_push(process p) {
    int sz = wait_q.size;
    int idx = p.index;
    wait_q.size++;
    wait_q.arr[sz] = p;
    while (sz > 0 && p.AT < wait_q.arr[(sz - 1) / 2].AT) {
        swap(&(wait_q.arr[sz]), &(wait_q.arr[(sz - 1) / 2]));
        sz = (sz - 1) / 2;
    }
}
```

wait queue에 프로세스를 push하는 함수이다.

가장 마지막 노드에 프로세스를 넣고 부모 노드와 비교해서 작으면 자리를 바꿔가며 적절한 위치를 찾는다.

```
process wait_q_top() {
    return wait_q.arr[0];
}
```

Wait queue의 최상단 노드의 값, 즉 queue에서 최소값을 갖는 노드를 반환한다.

```
process wait_q_pop() {
    process target = wait_q_top(); // 제일 위에 있는 거 반환
    int sz = --wait_q.size;
    if (sz == 0) return target; // queue에 하나 밖에 없었다면 밑의 과정 불필요
    wait_q.arr[0] = wait_q.arr[sz]; // 제일 끝에 있는 거 제일 위로 올림

    int idx = 0;
    while (idx * 2 + 2 < sz) { // 자식이 둘다 있다
        process now = wait_q.arr[idx]; // 현재 프로세스
        process c1 = wait_q.arr[idx * 2 + 1]; // 자식 1;
        process c2 = wait_q.arr[idx * 2 + 2]; // 자식 2;
        if (now.AT < c1.AT && now.AT < c2.AT) break; //자식이 둘다 더 큰 값이면 끝
        else {//자식중에 더 작은 값이 있으면 둘 중에 더 작은 값이랑 바꿈
            if (c1.AT < c2.AT) {
                swap(&wait_q.arr[idx], &wait_q.arr[idx * 2 + 1]);
                idx = idx * 2 + 1;
            }
            else {
                swap(&wait_q.arr[idx], &wait_q.arr[idx * 2 + 2]);
                idx = idx * 2 + 2;
            }
            //자식 두개가 같은 값이면 뭐랑 바꿔야 하나 생각해봤는데 상관없음
        }
    }
    if (idx * 2 + 2 == sz) { // 자식이 하나있다
        process now = wait_q.arr[idx]; // 현재 프로세스
        process c1 = wait_q.arr[idx * 2 + 1]; // 자식 1;
        if (now.AT > c1.AT) { // 자식이 더 작은 값이면 바꿈
            swap(&wait_q.arr[idx], &wait_q.arr[idx * 2 + 1]);
        }
    }

    return target;
}
```

Wait queue에서 프로세스를 pop하는 함수이다.

최상단 노드를 target에 보관 후, 제일 마지막 index에 있는 노드를 최상단으로 올리고, MIN HEAP 구조를 만족하도록 자식 노드들과 비교해가며 적절한 위치를 찾고 target을 반환한다.

```
int check_ready_q() { // return 값은 time quntum, 다 비어있다면 1 return
    if (is_empty(&q[0]) == 0) return 2;
    else if (is_empty(&q[1]) == 0) return 4;
    else if (is_empty(&q[2]) == 0) return 8;
    else if (is_empty(&q[3]) == 0) return 1000000;
    else return 1;
}
```

Ready queue에 대기중인 프로세스가 있는지 확인하는 함수이다.

return하는 값은 해당 ready queue의 time quntum이고 1을 return하게 되면 ready queue에서 대기중인 프로세스가 없음을 의미한다.

그리고 ready queue 3은 RR 스케줄링 방식이 아니라 FCFS 스케줄링 방식인데 이는 RR 스케줄링 방식의 time quntum이 충분히 커진다면 FCFS 스케줄링 방식과 같은 효과를 기대할 수 있기 때문에 1000000이라는 값을 임의로 부여하여 구현했다.

```
process to_cpu(int i) {
    process p;
    if (i == 2) p = pop(&q[0]);
    else if (i == 4) p = pop(&q[1]);
    else if (i == 8) p = pop(&q[2]);
    else p = pop(&q[3]);

    return p;
}
```

매개변수로 time quntum 값을 받으며 이 값에 따라 해당 ready queue에서 프로세스를 pop 한다.

```
void check_IO() {//우선순위 큐로 만들어야함
    int i;
    int sz = IO_q.size;
    if (sz == 0) return;

    for (i = 0;i < sz;i++) { //IO_q에 있는 process IO time 1씩 감소
        int idx = IO_q.arr[i].index;
        IO_q.arr[i].IO[idx]--;
    }
    while (IO_q.size != 0) { //process IO time이 0이 되면 wake up
        process top = IO_q_top();
        if (top.IO[top.index] == 0) {
            top.index++;
            int qn = top.QUEUE_NUM;
            // 원래 Q(i)에서 sleep했다면 Q(i-1)로 wake up
            if (qn > 0 && qn < 3) { // RQ(3)에서 RQ(2)로 이동 불가능
            //if(qn != 0){ // RQ(3)에서 RQ(2)로 이동가능
                qn--;
                (top.QUEUE_NUM)--;
            }
            push(&q[qn], top);
            IO_q_pop();
        }
        else break;
    }
}
```

IO queue에서 sleep 상태로 IO Burst 중인 프로세스들을 관리하는 함수이다.

매 time마다 IO queue에 있는 프로세스들의 각각의 cycle에 해당하는 IO burst time을 1만큼 감소시킨다. 이 때 IO queue의 top의 값이 0이 되면 sleep 상태에서 wake up함을 의미하므로 해당 프로세스의 원래 있었던 ready queue의 우선순위보다 한 단계 높은 ready queue로 push한다.

여기에서 프로세스가 ready queue 3에서 왔다면 다시 ready queue 3으로 push하도록 하였다.

그리고 IO queue에서 같은 시간에 wake up하는 프로세스가 여럿 존재한다면 IO queue의 MIN HEAP구조에서 프로세스가 위치하는 index가 낮은 순서로 wake up하도록 하였다.

```
int num_process;
fscanf(f, "%d", &num_process);

for (i = 0;i < num_process;i++) {
    //입력값으로 process p를 세팅
    process p;
    p.index = 0;
    fscanf(f, "%d %d %d %d", &p.PID, &p.AT, &p.QUEUE_NUM, &p.max_index);
    for (j = 0; j < p.max_index - 1; j++) {
        fscanf(f, "%d %d", &p.BT[j], &p.IO[j]);
    }fscanf(f, "%d", &p.BT[j]);
    p.SUM = sum_BT_IO(p);

    wait_q_push(p); // wait_q에서 AT가 될때까지 대기
}
fclose(f);
```

input.txt 파일에서 프로세스에 대한 정보를 입력 받는다.

```
while(wait_q.size != 0 && wait_q_top().AT == time) {
    process p = wait_q_pop();
    int qn = p.QUEUE_NUM;
    push(&q[qn], p);
}
```

매 루프마다 time이 1씩 증가하는 while문 안에서 wait queue에 있는 프로세스의 AT가 현재 time과 같다면 ready queue로 push한다.

```
time++;

int nothing = 0; // 기본값으로 0을 준다, 1이면 cpu가 할 스케줄링이 없음
if (flag == 0) { // cpu가 노는중이면
    flag = check_ready_q(); //RQ 0~3 차례대로 ready상태의 process 있는지 확인
    // 스케줄링 받게 될 process의 RQ의 time quntum
    //  RQ(0)면 time quntum 2 반환, RQ(1)면 time quntum 4 반환, RQ(2)면 time quntum 8 반환
    if (flag == 1) { // RQ에 대기중인 process가 없음
        nothing = 1;
    }
    else cpu = to_cpu(flag); // 해당 process의 RQ를 pop함
    // cpu 일하는 중 flag가 다시 0 될때까지, flag는 루프마다 1씩 감소예정
}
```

우선순위에 맞게 Ready queue 0부터 3까지 차례대로 확인하는데 프로세스가 있으면 그 프로세스를 cpu로 pop한다. 이는 프로세스가 스케줄링 받아 실행 중을 의미한다.

이 후 해당 프로세스의 time quntum만큼 이 과정은 불필요하기 때문에 flag라는 값을 두어 time quntum만큼 건너뛸 수 있도록 한다.

Flag에 1이 반환 되었다는 것은 ready queue에서 기다리는 프로세스가 없음을 의미하므로 cpu가 실행할 프로세스가 없다는 의미로 nothing을 1로 한다.

```
int is_quntum = 1; // time slice를 모두 소모했음을 기본값으로 함
if (nothing != 1) {
    int idx = cpu.index;
    int qn = cpu.QUEUE_NUM;

    cpu.BT[idx]--; // 루프 돌때마다 cpu가 스케줄링중인 process의 burst time 1씩 감소
    gantt_chart[time - 1][cpu.PID] = '#';

    if (cpu.BT[idx] == 0) { // cpu가 스케줄링중인 process의 burst time이 0이 되면
        if (cpu.index == cpu.max_index - 1) { // index와 max_index가 같으면 process의 종료
            int TT = time - cpu.AT;
            int WT = TT - cpu.SUM;
            done++;
            p_time[cpu.PID][0] = TT;
            p_time[cpu.PID][1] = WT;
            flag = 1;
            is_quntum = 0;
        }
        else {
            cpu.IO[cpu.index]++; // 여기서 시간 소모 했는데 밑에서 또 소모하기 때문
            IO_q_push(cpu); // sleep, 즉 IO_q로 보냄
            flag = 1; // 남은 time quntum에 상관없이 밑에서 flag--되면서 cpu가 놀고있다는
            is_quntum = 0; // 주어진 time quntum을 다 소모해서 cpu preemption 된 것이 아닐
        }
    }
}
```

Cpu가 실행하는 프로세스의 처리 과정이다. 해당 프로세스의 현재 cycle의 CPU burst time을 1만큼 감소시킨다. 이 때 이 값이 0이 되면 프로세스의 종료 또는 IO burst를 위한 sleep 이 두가지의 의미를 갖기 때문에 나눠서 처리한다. 종료라면 TT, WT를 기록하고, sleep이라면 프로세스를 IO queue로 push 한다.

```
check_IO(); // IO_q 관리 : IO time 1씩 감소, 0되면 wake up

flag--; // cpu 스케줄링 시간을 1만큼 소모

if (nothing == 0 && flag == 0 && is_quntum == 1) { //time quntum 소모
    cpu.QUEUE_NUM++;
    push(&q[cpu.QUEUE_NUM], cpu);
}
```

루프마다 IO queue의 프로세스들을 처리해준다. Time quntum을 모두 소모했지만 burst time을 끝내지 못하고 preemption된 경우에는 현재 ready queue에서 우선순위가 한 단계 낮은 ready queue로 push한다.

```
float avg_TT = 0, avg_WT = 0;
printf("₩n- RESULT -₩n");
for (i = 1;i <= num_process;i++) {
    printf("PID: %d , TT: %d, WT: %d₩n", i, p_time[i][0], p_time[i][1]);
    avg_TT += p_time[i][0];
    avg_WT += p_time[i][1];
}
avg_TT /= num_process;
avg_WT /= num_process;
printf("Average TT: %.02f, Average WT: %.02f₩n", avg_TT, avg_WT);
printf("₩n₩n- GANTT CHART -₩nTIME     ");
for (i = 1;i <= num_process;i++) printf("P%d ", i);
printf("₩n――――――――――₩n");
for (i = 0;i < time;i++) {
    printf("  %d   :  ", i);
    for (j = 1;j <= num_process;j++) {
        printf("%c  ", gantt_chart[i][j]);
    }
    printf("₩n");
}

return 0;
```

모든 프로세스가 스케줄링 받아 종료되었다면 결과를 출력한다.

5. 예제 테스트

```
6
1 20 2 2 20 25 30
2 10 2 2 35 40 45
3 13 2 2 50 55 60
4 150 3 3 65 66 70 80 90
5 328 1 5 13 25 36 16 21 78 26 100 1
6 0 0 3 65 66 70 80 90
```

```
- RESULT -
PID: 1 , TT: 372, WT: 297
PID: 2 , TT: 340, WT: 220
PID: 3 , TT: 439, WT: 274
PID: 4 , TT: 589, WT: 218
PID: 5 , TT: 637, WT: 321
PID: 6 , TT: 613, WT: 242
Average TT: 498.33, Average WT: 262.00
```

```
4
1 20 2 2 20 25 30
2 10 2 2 35 40 45
3 13 2 2 50 55 60
4 150 3 3 65 66 70 80 90
```

```
- RESULT -
PID: 1 , TT: 170, WT: 95
PID: 2 , TT: 150, WT: 30
PID: 3 , TT: 302, WT: 137
PID: 4 , TT: 411, WT: 40
Average TT: 258.25, Average WT: 75.50
```

```
4
1 0 3 2 5 5 10
2 5 2 2 10 5 1
3 10 1 2 20 20 50
4 15 0 3 30 10 10 1 20
```

```
- RESULT -
PID: 1 , TT: 49, WT: 29
PID: 2 , TT: 71, WT: 55
PID: 3 , TT: 119, WT: 29
PID: 4 , TT: 145, WT: 74
Average TT: 96.00, Average WT: 46.75
```

```
10
1 0 0 2 50 10 30
2 5 1 2 10 5 5
3 30 2 2 20 100 10
4 100 3 2 40 20 100
5 0 0 2 10 10 10
6 50 1 2 5 200 5
7 200 2 2 30 50 5
8 200 3 2 1 1 1
9 0 0 2 50 50 50
10 10 1 2 100 100 100
```

```
- RESULT -
PID: 1 , TT: 343, WT: 253
PID: 2 , TT: 82, WT: 62
PID: 3 , TT: 597, WT: 467
PID: 4 , TT: 416, WT: 256
PID: 5 , TT: 86, WT: 56
PID: 6 , TT: 263, WT: 53
PID: 7 , TT: 432, WT: 347
PID: 8 , TT: 317, WT: 314
PID: 9 , TT: 394, WT: 244
PID: 10 , TT: 607, WT: 307
Average TT: 353.70, Average WT: 235.90
```

```
8
1 0 0 3 50 10 30 50 10
2 5 1 2 10 5 5
3 30 2 2 20 100 10
4 100 3 3 40 20 100 40 20
5 0 0 2 10 10 10
6 50 1 2 5 200 5
7 200 2 5 30 50 5 30 50 5 30 50 5
8 200 3 2 1 1 1
```

```
- RESULT -
PID: 1 , TT: 314, WT: 164
PID: 2 , TT: 56, WT: 36
PID: 3 , TT: 274, WT: 144
PID: 4 , TT: 257, WT: 37
PID: 5 , TT: 60, WT: 30
PID: 6 , TT: 243, WT: 33
PID: 7 , TT: 361, WT: 106
PID: 8 , TT: 137, WT: 134
Average TT: 212.75, Average WT: 85.50
```

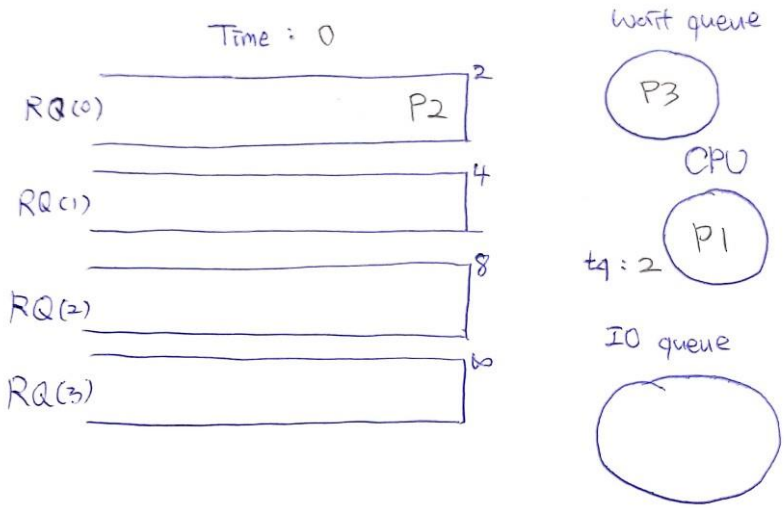
간단한 예제를 시뮬레이션하여 올바른 값을 출력하는지 확인해 보았다.

```
3
1 0 0 2 10 10 10
2 0 0 3 5 5 20 5 5
3 5 1 2 15 5 1
```

```
- RESULT -
PID: 1 , TT: 51, WT: 21
PID: 2 , TT: 70, WT: 30
PID: 3 , TT: 56, WT: 35
Average TT: 59.00, Average WT: 28.67
```

GANTT CHART는 아래와 같이 출력된다.

```
- GANTT CHART -
TIME     P1 P2 P3
------------------
  0   :  #  -  -
  1   :  #  -  -
  2   :  -  #  -
  3   :  -  #  -
  4   :  #  -  -
  5   :  #  -  -
  6   :  #  -  -
  7   :  #  -  -
  8   :  -  #  -
  9   :  -  #  -
  10  :  -  #  -
  11  :  -  -  #
  12  :  -  -  #
  13  :  -  -  #
  14  :  -  -  #
  15  :  #  -  -
  16  :  #  -  -
  17  :  #  -  -
  18  :  #  -  -
  19  :  -  #  -
  20  :  -  #  -
  21  :  -  #  -
  22  :  -  #  -
  23  :  -  #  -
  24  :  -  #  -
  25  :  -  -  #
  26  :  -  -  #
  27  :  -  -  #
  28  :  -  -  #
  29  :  -  -  #
  30  :  -  -  #
  31  :  -  -  #
  32  :  -  -  #
  33  :  #  -  -
  34  :  #  -  -
  35  :  #  -  -
  36  :  #  -  -
  37  :  -  #  -
  38  :  -  #  -
  39  :  -  #  -
  40  :  -  #  -
  41  :  -  #  -
  42  :  -  #  -
  43  :  -  #  -
  44  :  -  #  -
  45  :  #  -  -
  46  :  #  -  -
  47  :  #  -  -
  48  :  #  -  -
  49  :  #  -  -
  50  :  #  -  -
  51  :  -  -  #
  52  :  -  -  #
  53  :  -  -  #
  54  :  -  #  -
  55  :  -  #  -
  56  :  -  #  -
  57  :  -  #  -
  58  :  -  #  -
  59  :  -  #  -
  60  :  -  -  #
  61  :  -  -  -
  62  :  -  -  -
  63  :  -  -  -
  64  :  -  -  -
  65  :  -  #  -
  66  :  -  #  -
  67  :  -  #  -
  68  :  -  #  -
  69  :  -  #  -
```



위와 같은 time 0 상태에서 시작하여 time별로 스케줄링 과정을 진행해보면 위의 GANTT CHART와 같은 결과를 얻을 수 있다.