



학과 : 지능정보공학부 휴먼지능정보공학과

학번 : 201810776

조명 : 천지창조

이름 : 소재휘

과목 : 운영체제

과제명 : 라운드 로빈 알고리즘 구현

I. 라운드 로빈이란?

운영체제에서는 여러 개의 프로세스의 실행을 효율적으로 처리하기 위해서 프로세스가 실행되는 순서를 정하는, 즉 프로세스가 중앙처리장치(CPU)를 점유하는 순서를 정해주는 스케줄링을 사용한다. 스케줄링 기법으로는 크게 하나의 프로세스의 작업이 완료될 때 까지 CPU를 독점하는 비선점형 스케줄링 방식과 한 프로세스가 CPU를 점유하고 있을 때 다른 프로세스가 CPU를 빼앗을 수 있는 선점형 스케줄링 방식이 있다.

라운드 로빈(RR)이란 대표적인 선점형 스케줄링 방식이다. 라운드 로빈은 프로세스가 작업을 할 수 있는 시간, 즉 타임 슬라이스를 정하고 그 시간동안 작업을 하는 시분할 방식에 적당한 알고리즘이다. 이는 시분할 시간 동안만 작업을 하고 작업을 마치지 못했어도 다른 프로세스에게 CPU를 넘겨주고 다음 차례를 기다리므로 앞의 프로세스가 끝날 때 까지 무작정 기다려야 하는 콘베이 효과를 최소화시킬 수 있다.

한 프로세스가 CPU를 점유할 때 타임 슬라이스가 끝나 다른 프로세스에게 CPU 점유권을 넘겨주는 것을 문맥 교환이라고 한다. 선점형 스케줄링 방식과 달리 라운드 로빈 방식에서는 이 문맥 교환 시간 또한 작업 시간에 포함이 된다. 스케줄링이 효율적으로 이루어지려면 이 문맥 교환 시간을 고려하여 타임 슬라이스의 크기를 지정할 필요성이 있다.

타임 슬라이스 시간이 너무 크다면 문맥 교환이 잘 일어나지 않게 된다. 즉 비선점형 스케줄링 방식의 단점인 긴 프로세스가 실행 될 경우 뒤에 있는 프로세스들은 무한정 기다리기만 해야 하므로 대화식 시스템에 적합하지 않으며, 평균 응답 시간이 길어지는 단점이 있다.

반대로 타임 슬라이스 시간이 너무 작다면 여러 프로그램이 동시에 실행되는 것처럼 보이지만 빈번한 문맥 교환이 일어나게 되므로 평균 작업 시간이 길어지게 되며 성능이 떨어지게 된다.

따라서 라운드 로빈 알고리즘을 설계할 시 타임 슬라이스의 크기를 고려하는 것은 중요하다.

II. 라운드 로빈 알고리즘

시분할 시스템에서 사용되는 대표적인 스케줄링 방식인 라운드 로빈 알고리즘을 설계할 시 생각해아 할 사항들은 다음과 같았다.

1. 라운드 로빈을 설계할 시 적합한 컴퓨터 언어
2. timeout()과 dispatch(), exit()의 조건을 어떻게 구현할 것인가?
3. Arrival time 고려하기.
 - 3-1. 큐에 프로세스가 없을 경우 고려(큐에 프로세스가 들어오지 않은 상태(없는 상태)에서는 WT와 TT를 증가시키면 안 되며 프로세스가 큐에 도착할 때 까지 기다려야 한다.)
 - 3-2. 라운드 로빈 알고리즘이 작동 도중 프로세스가 도착했을 경우 어떻게 할 것인가?
4. 프로세스들이 작업할 때 Wait time과 Total time을 어떻게 계산할 것인가?

다음은 고려하면서 **직접!** 라운드 로빈 알고리즘을 자바를 통해 구현하였다. 우선 자바를 선택한 이유는 여러 가지의 프로세스들과 스케줄러 간의 상호 작용을 표현하기에는 객체지향 언어가 적절하다고 판단하였으며 스케줄러가 작동하는 모습을 코드만 보고도 알아볼 수 있도록 표현하기에 적절하다고 판단하였다.

```

package roundrobin;

public class Process implements Comparable<Object>{

    protected int pid;
    protected int AT;           //Arrival time 준비 큐에 프로세스가 도착하는 시간. 도달 시간
    protected int RT;           //Burst time 버스트 시간. 프로세스가 작업을 완료하는 데에 필요한 CPU를 점유시간
    protected int WT;           //Wait time 준비 상태에서 dispatch를 기다리는 시간
    protected int TT;           //Total time 모든 프로세스의 작업을 처리하는 데에 걸리는 시간 Response time+Wait time
    protected int ATcheck;      //진행중인 시간과 AT를 비교하기 위해 만든 변수
    protected int RTcheck;      //진행중인 시간과 RT를 비교하기 위해 만든 변수
    protected int index;        //AT 오름차순으로 정렬 시의 순서 index
    protected boolean flag=false; //AT가 지나고 준비 큐에 도달했는지에 대한 여부
    @Override //Sort시 AT가 작은 순서로 정렬
    public int compareTo(Object o) {
        Process p = (Process)o;
        return Integer.compare(this.AT, p.AT);
    }
}

```

다음은 프로세스에 해당하는 클래스이다. 해당 클래스에서 라운드 로빈 알고리즘에서 필요한 큐까지 프로세스가 도달하는 시간인 AT, 작업을 완료하기 위해 필요한 CPU 점유시간인 RT, 큐에서 프로세스가 대기하는 시간인 WT, 프로세스가 작업을 완료할 때 까지 걸리는 시간(Response time+Wait time)TT를 속성으로 넣었으며 프로세스를 구분하기 위해서 PID, 그 외에 알고리즘 계산 과정에서 필요한 일부 변수들을 넣었다.

```

public class Queue {
    int MAX_QUEUE_SIZE;
    Process data[]; //프로세스를 넣을 수 있는 큐 배열
    int front;
    int rear;
    public void init_queue(Queue q1)//큐 초기화
    public boolean is_empty(Queue q1)//공백 상태 검출
    public boolean is_full(Queue q1)//포화 상태 검출
    public void en_queue(Queue q1, Process p1)//enqueue
    public Process de_queue(Queue q1)//dequeue
}

```

다음은 준비 큐를 구현하기 위해서 만든 클래스이다. 매서드의 내용은 전형적인 원형 큐이므로 생략하도록 하겠다. 이곳에는 생략했지만 생성자를 통해 MAX_QUEUE_SIZE를 받는다.

```

public class Roundrobinmain {
    public static void main(String[] args) {
        Scheduler s1 = new Scheduler(5,4); //스케줄러 생성. (타임 슬라이스/프로세스 개수)
        s1.createProcess(); //프로세스 생성
        s1.getReadTime(); //RT 설정
        s1.readyQueueing(); //라운드 로빈 준비
        s1.roundRobin(); //라운드 로빈 스케줄링
        s1.showResult(); //결과 출력
    }
}

```

라운드 로빈 알고리즘을 작동시키는 메인 함수이다. 스케줄러를 생성하고 절차에 따라서 라운드 로빈 알고리즘을 수행한다.

아래의 전체 코드는 스케줄러에 해당하는 클래스이다.

```

public class Scheduler {
    int timeSlice; //타임 슬라이스
    int processNum; //프로세스 개수
    int totalTime; //모든 프로세스의 RT의 합, 총 CPU 점유시간
    Queue queue; //대기 큐
    Process selected; //선택된 프로세스
    ArrayList<Process> processArr = new ArrayList<Process>();
    Scheduler(int time, int number){ //타임 슬라이스와 프로세스의 총 개수를 생성자로 받음
        this.timeSlice=time;
        this.processNum=number;
    }
}

```

스케줄러를 생성할 때 사용자에게 타임 슬라이스와 프로세스의 개수를 생성자로 넘겨 받는다.

```

void createProcess(){
    //프로세스 개수만큼 프로세스를 생성하고 배열리스트에 추가
    for(int i=0; i<this.processNum; i++) {
        processArr.add(new Process());
        processArr.get(i).pid=i+1;
        processArr.get(i).AT = (int)(Math.random()*20)+1;
    }
}

```

다음은 프로세스를 프로세스의 개수만큼 생성하는 함수이다. 이 때 pid번호와 Arrival time을 부여한다.

```
//프로세스들의 작업시간 입력
void getReadTime() {
    Scanner sc = new Scanner(System.in);
    for(int i=0; i < this.processNum; i++) {
        System.out.printf(i+1 + "번째 프로세스 작업시간 입력: ");
        processArr.get(i).RT = sc.nextInt();
        this.totalTime += processArr.get(i).RT;    //작업시간들의 총 시간
    }
    sc.close();
}
```

프로세스의 작업 시간(RT)를 사용자에게 입력받는 기능을 한다.

```
//현재 시간과 프로세스의 RT비교와 대기 큐에 있는 WT계산, 총 TT를 계산한다.
void timeCalculate(int sequence) {
    processArr.get(sequence).RTcheck++; //현재 dispatch된 프로세스의 RT와 진행시간을 비교하고 TT를 계산한다.
    processArr.get(sequence).TT++;
    for(int t=0; t<processNum; t++) { //현재 dispatch되지 않고 준비 큐에 있는 프로세스들의 wait time과 turn around time 계산
        if(t!=sequence && processArr.get(t).RT != processArr.get(t).RTcheck && processArr.get(t).flag==true) {
            processArr.get(t).WT++;
            processArr.get(t).TT++;
        }
    }
    //프로세스들의 Arrival time을 계산함과 동시에 Arrival time이 지난 프로세스를 큐에 집어 넣는다.
}
```

현재 CPU를 점유하는 프로세스(selected)의 작업 시간(RT)과 준비 큐에서 대기 상태인 프로세스들의 대기 시간(WT), 그리고 위의 프로세스들의 총 작업 시간(TT)을 조작하거나 비교하는 함수이다. 이를 구현하기 위해서 현재 점유중인 프로세스의 index번호를 받아 해당 프로세스의 시간값을 조작하는 형태를 채택하였다.

```
void timeCalculateAT() { //프로세스들의 Arrival time을 계산함과 동시에 Arrival time이 지난 프로세스를 큐에 집어 넣는다.
    for(int i=0; i<processNum; i++) {
        if(processArr.get(i).flag==false) {
            processArr.get(i).ATcheck++; //각 프로세스들마다 AT와 비교하기 위한 ATcheck를 증가시킨다.
        }
        if(processArr.get(i).ATcheck==processArr.get(i).AT && processArr.get(i).flag==false) {
            queue.en_queue(queue, processArr.get(i)); //AT에 도달하였고 준비 큐에 들어오지 않은 프로세스들을 enqueue한다.
            processArr.get(i).flag=true; //준비 큐에 도달했다는 flag를 true로 만든다.
            System.out.printf("%darrive)", processArr.get(i).pid);
        }
    }
}
```

위의 고려사항 중 Arrival time을 고려하기 위해서 만든 AT 계산 함수이다. 이 함수가 실행될 때 마다 flag가 false인 Arrival time이 계산이 되고(flag는 준비 큐에 프로세스가 도착했는지의 여부를 판단하는 boolean), AT가 충족이 되었는 flag가 false인 경우 프로세스가 도착한 것으로 판단하고 해당 프로세스를 준비 큐에 enqueue시킨다.

```
//라운드 로빈 준비, 큐를 생성하고 큐에 첫번째 프로세스가 들어올 때까지 대기
void readyQueueing() {
    Collections.sort(processArr); //프로세스들을 AT에 따라 오름차순으로 정렬
    for(int i=0; i<this.processNum; i++) { //현재 정렬된 프로세스들의 정보 출력
        processArr.get(i).index=i;
        System.out.printf("pid : %d, AT : %d RT : %d\n", processArr.get(i).pid, processArr.get(i).AT, processArr.get(i).RT );
        this.queue = new Queue(processNum+1); //큐 생성
        queue.init_queue(queue); //큐 초기화
    }
}
```

라운드 로빈 알고리즘을 시작하기 전의 준비 단계로 큐를 생성하며 먼저 도착한 프로세스들을 판단하기 위해서 AT에 따라 정렬한 후 index를 부여해준다.

```
//준비 큐에 프로세스가 있는지를 확인하는 작업
void is_have_process() {
    //만약 준비 큐에 프로세스가 없다면 Arrival time을 계산하며 큐에 프로세스가 도착할 때 까지 while을 통해 대기한다.
    while(queue.is_empty(queue)==true) {
        timeCalculateAT();
        if(queue.is_empty(queue)==true) {
            System.out.printf("*");
        }
    }
}
```

Arrival time을 생각할 때 고려해야 할 점에는 시작할 때, 혹은 라운드 로빈 과정 도중 큐에 어떠한 프로세스도 도착하지 않은 경우가 있을 수 있다. 이를 위해서 만든 함수로 queue가 비어있는 동안은 timeCalculateAT()를 반복하여 프로세스가 도착하는 것을 기다린다.

```
void roundRobin() { //라운드 로빈 스케줄링
    int time = 1; //현재 진행 시간
    int allocate = 0; //현재 프로세스의 타임 슬라이스 할당 시간

    //라운드 로빈 초기 시작*
    is_have_process(); //큐에 프로세스가 있는가의 여부
    selected=queue.de_queue(queue); //CPU에서 작업할 프로세스를 dequeue한다.
    //진행 시간에 따른 AT, RT, WT, TT를 계산해준다.
    timeCalculateAT();
    timeCalculate(selected.index);
    System.out.printf("%d", selected.pid);
    allocate++; //현재 프로세스의 타임 슬라이스 할당 시간을 계산해준다.
    //*****라운드 로빈 알고리즘*****

    while(time<totalTime) {
        timeCalculateAT(); //AT 계산
    }
}
```

```

//해당 프로세스가 타임 슬라이스를 모두 사용했거나(timeout), 작업이 끝난 경우(exit) 문맥 교환을 해준다.
if(allocate>=timeSlice||processArr.get(selected.index).RTcheck>=processArr.get(selected.index).RT) {
//타임 슬라이스를 모두 사용했지만 작업이 끝나지 않은 경우(timeout), 다시 준비 큐에 해당 프로세스를 넣어준다.
if(allocate>=timeSlice&&processArr.get(selected.index).RTcheck<processArr.get(selected.index).RT) {
queue.en_queue(queue, processArr.get(selected.index));
}
is_have_process();//큐에 프로세스가 있는지를 검사. 없다면 프로세스가 도착할 때 까지 대기
selected=queue.de_queue(queue); // dispatch, 큐에서 프로세스를 dequeue한다.
System.out.printf("%d", selected.pid);
allocate=1; //타임 슬라이스 할당 시간 초기화
}
else {
System.out.printf("-");
allocate++; //타임 슬라이스 할당 시간 증가
}

time++; //작업 진행 시간 증가
timeCalculate(selected.index);
}

//*****라운드 로빈 알고리즘 종료*****

System.out.println("완료");

```

라운드 로빈의 핵심 알고리즘을 수행하고 있는 함수이다. 현재 작업 시간 time과 현재 프로세스의 타임 슬라이스 점유시간 allocate를 설정한다. 그리고 초기 시작 단계를 하는데 큐에 프로세스가 도착했을 때부터 time과 allocate를 조작하며 스케줄링을 시작한다. time이 총 작업해야 하는 양인 totalTime에 도달하기 전 증가할 때 마다 현재 CPU를 점유하는 프로세스(selected)와 준비 큐의 프로세스들의 RT,TT,WT,AT를 확인하고 계산한다. 타임 슬라이스를 모두 사용하였거나 작업이 끝난 경우 문맥 교환이 이루어지게 되는데 이 과정에서의 dispatch, exit, timeout의 조건을 설정하였다. 타임 슬라이스를 모두 사용하였지만 작업이 끝나지 않은 경우에는 queue에 해당 프로세스를 다시 넣어준다. 이 과정 후 문맥 교환 과정에서 준비 큐에 프로세스가 있는지를 검사한 후 없다면 프로세스가 도착할 때 까지 대기한 후 큐에서 다음 교환해줄 프로세스를 enqueue시킨다. 그리고 현재 CPU를 점유하는 프로세스(selected)를 갱신시키며 타임 슬라이스 시간을 초기화시킨다.

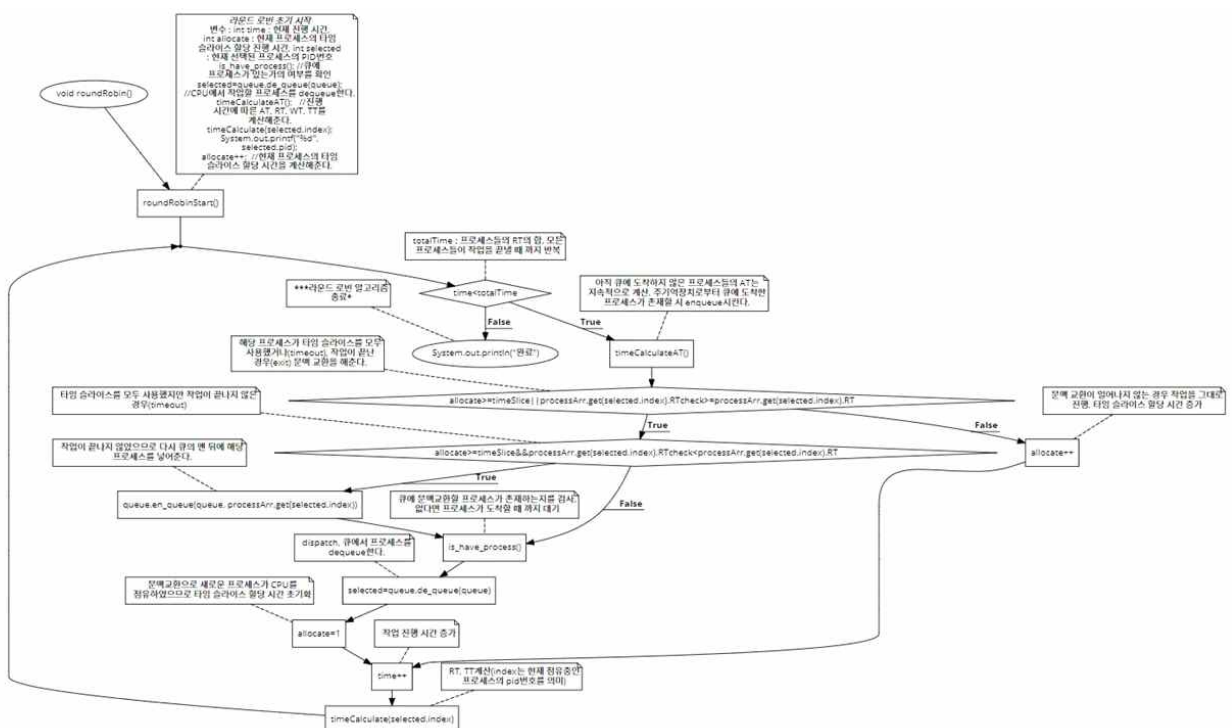
```

void showResult(){
System.out.println("(PIDarrive):큐에 해당 프로세스 arrive, 숫자(PID) : 해당 pid가 문맥 교환과 동시에
타임 슬라이스 점유, * : 큐에 프로세스가 도착하지 않음");
System.out.printf("프로세스 개수 : %d, Time slice : %d\n", processNum, timeSlice);
for(int i=0; i<this.processNum; i++) {
System.out.printf("Process[PID : %d] : AT : %d, RT : %d, WT : %d TT %d\n",
processArr.get(i).pid,processArr.get(i).AT,processArr.get(i).RT,processArr.get(i).WT,processArr.get(i).TT);
}
}

라운드 로빈을 진행한 후 계산된 AT,RT,WT,TT의 결과를 출력해주는 함수이다.

```

III. 알고리즘 순서도(Flow chart)



위의 코드를 순서대로 표현하여 라운드 로빈 스케줄링을 구현한 알고리즘의 흐름을 한 눈에 파악할 수 있도록 만들어 보았다.

알고리즘 설계 시 고려 사항들이 있었는데 그것들을 중점으로 순서도를 설명하고자 한다.

1. 기본적으로 모든 프로세스들이 작업을 마칠 때 까지(exit()할 때 까지) while 루프가 반복되는 형태이다.
2. 시간들의 계산은 while문 안에서 time이 증가할 때 마다 timeCalculate()를 통하여 주기적으로 계산하였다.
3. dispatch()와 timeout(), 그리고 exit()의 조건은 각각 처음 라운드 로빈을 처음 시작할 때나 문맥 교환이 일어날 경우 큐에 프로세스가 남아있을 경우 dispatch(), 배정받은 타임 슬라이스가 끝났으며 작업할 RT가 남아있는 경우 timeout(), 모든 작업이 끝나 RT와 RTcheck가 일치하는 경우 exit()하는 것으로 설계하였다.
4. Arrival time을 고려하기 위해서 time이 증가할 때 마다 ATcheck()를 통하여 Arrival time을 계산해주었다. 또한 enqueue(dispatch())전에 is_have_process()를 통하여 큐에 프로세스가 있는지를 확인하였으며 프로세스가 없다면 time을 증가시키지 않고 ATcheck()를 반복하며 큐에 프로세스가 도착할 때 까지 대기하는 것으로 알고리즘을 설계하였다.

IV. 결과 분석

```
pid : 3, AT : 4 RT : 3
pid : 2, AT : 10 RT : 7
pid : 4, AT : 11 RT : 8
pid : 1, AT : 14 RT : 12
*** (3arrive)3--*(2arrive)2(4arrive)---(1arrive)-4----1----2-4--1----1-완료
(PIDarrive):큐에 해당 프로세스 arrive, 숫자(PID) : 해당 pid가 문맥 교환과 동시에 타임 슬라이스 점유, - : 타임 슬라이스 점유, * : 큐에 프로세스가 도착하지 않음
프로세스 개수 : 4, Time slice : 5
Process[PID : 3] : AT : 4, RT : 3, WT : 0 TT : 3
Process[PID : 2] : AT : 10, RT : 7, WT : 10 TT : 17
Process[PID : 4] : AT : 11, RT : 8, WT : 11 TT : 19
Process[PID : 1] : AT : 14, RT : 12, WT : 11 TT : 23
```

(프로세스 개수 4, 타임 슬라이스 5 : Scheduler s1 = new Scheduler(5,4);)

시작할 때 큐에 프로세스가 도착하지 않았을 경우 PID 3에 해당하는 프로세스가 도착할 때 까지((3arrive)로 출력되는 부분) *로 결과가 출력되어 큐에 프로세스가 도착할 때 까지 대기하는 것을 확인 할 수 있다. 지정된 타임 슬라이스를 다 사용하거나 작업을 마치게 되면 문맥 교환도 잘 일어난다. 또한 라운드 로빈 도중 PID:3와 PID:2에서의 문맥교환을 보면 문맥교환 할 프로세스가 큐에 없을 시 *을 통해 대기하는 것 또한 확인이 되었다. 따라서 라운드 로빈 스케줄링이 잘 구현되었음을 확인할 수 있다.