Report

CPU Scheduler

2016320266 컴퓨터학과 박준성

목차

1. CPU 스케줄러의 개념

CPU 스케줄링은 다중프로그램 운영체제에서 필수적인 요소이다. 단일 처리기 시스템에서는 한 순간에 오직 하나의 프로세스만이 실행 될 수 있는데, 이러한 단일처리기로 메모리상의 여러개의 프로세스를 순차적으로 실행하기 위해서는 스케줄링 기법이 필요하다. 프로세스실행은 CPU실행과 입/출력 대기 사이클로 구성된다. 입/출력 중심의 프로그램은 전형적으로 짧은 CPU Burst를 가진다. CPU중심의 프로그램은 다수의 긴 CPU Burst를 가진다. 입/출력과 CPU 사이클이 적절하게 조화를 이루어야 한다. CPU 스케줄러는 메모리내의 프로세스들중에 하나를 선택하여 CPU 할당하고, CPU가 유휴 상태가 될때마다 OS는 준비큐에 있는 프로세스들 중에 하나를 선택하여 실행한다.

2. 다른 CPU 스케줄링 시뮬레이터에 대한 소개

< A CPU Scheduling Algorithm Simulator(Sukanya Suranauwarat / School of Applied Statistics, National Institute of Development Administration) >

이 시뮬레이터는 많은 유일한 특징들을 가지고 있다. 먼저, 사용자에 의해 쉽게 구성 될 수 있는 사실적인 프로세스 모델을 사용한다. 둘째, 시간에 따른 프로세스의 진행 상황을 시각적으로 묘사한다. 이 표현을 통해 시스템 내부의 상황을 쉽게 알 수 있고 왜 다른 시간에 다른 프로세스의 집합이 CPU 할당의 후보가 되는지 알 수 있다. 마지막 특징은 그래픽을 사용한 사용자 인터페이스를 통해 사용자가 스케줄링을 직접 결정함으로써 자신이 이해한 개념을 테스트하고 이해도를 높일 수 있다.

이 시뮬레이터는 자바 5.0을 사용하여 작성되었고, 두 실행 모드를 가지고 있다. 먼저 시뮬레이션 모드에서는 사용자가 알고리즘이 어떻게 작동하는지 단계별로 관찰하거나, 처음부터 끝까지 한 번에 실행되는 것을 볼 수도 있다. 연습 모드에서는 사용자가 자신의 스케줄링 정책을 만듦으로써 스케줄링에 대한 이해도를 높일 수 있다. 즉 언제 프로세스가 실행되는지, 얼마나 오래 실행되는지를 사용자가 직접 결정하는 것이다.

3. 시뮬레이터의 시스템 구성도

이 시뮬레이터는 프로세스에 대한 정보를 pro라는 구조체에 저장하고, 구조체 안에는 process id, cpu burst time, arrival time, priority, state가 저장된다. 또한 I/O operation을 위해서 io_list라는 2차원 배열을 사용했다. 각 스케줄링 알고리즘은 fcfs, non_sjf, non_pri, pre_sjf, pre_pri, round라는 함수로 구현되어 있다. 또한 insert_sjf, delete_sjf는 sjf 스케줄링에서 ready queue를 구현하기 위하여 사용되는 함수이고 insert_pri, delete_pri는 priority 스케줄링에서 ready queue를 구현하기 위해 사용되는 함수이다.

4. 모듈에 관한 설명

```
fcfs() {
        time=0;
        while(1) {
                모든 프로세스의 상태 검사(모두 terminal 상태라면 break);
                현재 run 상태인 프로세스의 cpuburst--;
                현재 io 작업 중인 프로세스의 io length--;
                간트 차트에 현재 run 상태인 process id 삽입;
                time++;
                if(arrival time==time) ready queue에 삽입;
                if(io_length == 0) io가 완료된 프로세스를 ready queue에 삽입;
                if(io 발생 시점이라면)
                        run상태인 프로세스를 waiting queue에 삽입;
                        새로운 프로세스를 run으로 전환;
                if(io 실행 중인 프로세스가 없다면) waiting queue에서 추출
                if(cpuburst==0)
                        현 프로세스의 상태를 종료로 바꾸고 ready queue에서 새
                        프로세스를 run상태로 전환
        }
        간트차트 출력;
}

non_sjf() {
        time=0;
        while(1) {
                모든 프로세스의 상태 검사(모두 terminal 상태라면 break);
                현재 run 상태인 프로세스의 cpuburst--;
                현재 io 작업 중인 프로세스의 io length--;
                간트 차트에 현재 run 상태인 process id 삽입;
                time++;
                if(arrival time==time) ready queue에 삽입;
                if(io 실행 중인 프로세스의 io_length == 0)
                        io가 완료된 프로세스를 ready queue에 삽입;
                if(run 상태의 프로세스의 io 발생 시점이라면)
                        run상태인 프로세스를 waiting queue에 삽입; 새로운 프로세스를
                        run으로 전환;
                if(io 실행 중인 프로세스가 없다면) waiting queue에서 추출;
                if(cpuburst==0)
                        현 프로세스의 상태를 종료로 바꾸고 ready queue에서 새
                        프로세스를 run상태로 전환;
```

```
        }
        간트 차트 출력;
}

non_pri() {
        time=0;
        while(1) {
                모든 프로세스의 상태 검사(모두 terminal 상태라면 break);
                현재 run 상태인 프로세스의 cpuburst--;
                현재 io 작업 중인 프로세스의 io length--;
                간트 차트에 현재 run 상태인 process id 삽입;
                time++;
                if(arrival time==time) ready queue에 삽입;
                if(io 실행 중인 프로세스의 io_length == 0)
                        io가 완료된 프로세스를 ready queue에 삽입;
                if(run 상태의 프로세스의 io 발생 시점이라면)
                        run상태인 프로세스를 waiting queue에 삽입; 새로운 프로세스를
                        run으로 전환;
                if(io 실행 중인 프로세스가 없다면) waiting queue에서 추출;
                if(cpuburst==0)
                        현 프로세스의 상태를 종료로 바꾸고 ready queue에서 새 프로세
                        스를 run상태로 전환;
        }
        간트 차트 출력;
}

pre_sjf() {
        time=0;
        while(1) {
                모든 프로세스의 상태 검사(모두 terminal 상태라면 break);
                현재 run 상태인 프로세스의 cpuburst--;
                현재 io 작업 중인 프로세스의 io length--;
                간트 차트에 현재 run 상태인 process id 삽입;
                time++;
                if(arrival time==time) ready queue에 삽입;
                if(io 실행 중인 프로세스의 io_length == 0)
                        io가 완료된 프로세스를 ready queue에 삽입;
                if(run 상태의 프로세스의 io 발생 시점이라면)
                        run상태인 프로세스를 waiting queue에 삽입;
                        새로운 프로세스를 run으로 전환;
                if(io 실행 중인 프로세스가 없다면) waiting queue에서 추출;
```

```
            if(cpuburst==0)
                현 프로세스의 상태를 종료로 바꾸고 ready queue에서 새
                프로세스를 run상태로 전환;
            else if(ready queue의 첫 번째 process의 cpuburst time < run상태의
            process의 cpuburst time)
                ready queue의 첫 번째 process를 run 상태로 전환;
                이전의 프로세스는 ready queue에 삽입;
        }
        간트 차트 출력;
}

pre_pri() {
        time=0;
        while(1) {
            모든 프로세스의 상태 검사(모두 terminal 상태라면 break);
            현재 run 상태인 프로세스의 cpuburst--;
            현재 io 작업 중인 프로세스의 io length--;
            간트 차트에 현재 run 상태인 process id 삽입;
            time++;
            if(arrival time==time) ready queue에 삽입;
            if(io 실행 중인 프로세스의 io_length == 0)
                io가 완료된 프로세스를 ready queue에 삽입;
            if(run 상태의 프로세스의 io 발생 시점이라면)
                run상태인 프로세스를 waiting queue에 삽입; 새로운 프로세스를
                run으로 전환;
            if(io 실행 중인 프로세스가 없다면) waiting queue에서 추출;
            if(cpuburst==0)
                현 프로세스의 상태를 종료로 바꾸고 ready queue에서 새
                프로세스를 run상태로 전환;
            else if(ready queue의 첫 번째 process의 priority < run상태의 process
            의 priority)
                ready queue의 첫 번째 process를 run 상태로 전환;
                이전의 프로세스는 ready queue에 삽입;
        }
        간트 차트 출력;
}
```

```
round() {
        time=0; time quantum 랜덤 생성;
        remain 배열에 각 프로세스의 time quantum 저장;
        while(1) {
                모든 프로세스의 상태 검사(모두 terminal 상태라면 break);
                현재 run 상태인 프로세스의 cpuburst--, time quantum--;
                현재 io 작업 중인 프로세스의 io length--;
                간트 차트에 현재 run 상태인 process id 삽입;
                time++;
                if(arrival time==time) ready queue에 삽입;
                if(io 실행 중인 프로세스의 io_length == 0)
                        io가 완료된 프로세스를 ready queue에 삽입;
                if(run 상태의 프로세스의 io 발생 시점이라면)
                        run상태인 프로세스를 waiting queue에 삽입;
                        새로운 프로세스를 run으로 전환;
                if(io 실행 중인 프로세스가 없다면) waiting queue에서 추출;
                if(cpuburst==0)
                        현 프로세스의 상태를 종료로 바꾸고 ready queue에서 새
                        프로세스를 run상태로 전환;
                else if(running process의 남은 time quantum이 0이라면)
                        ready queue의 첫 번째 process를 run 상태로 전환;
                        이전의 프로세스는 ready queue에 삽입;
        }
        간트 차트 출력;
}

insert_sjf(pro ready[], pro item, int* size) {
        size = size+1;
        i=size;
        ready[i]=item;
        while(i!=1 && ready[i].cpuburst time < ready[i/2].cpuburst time)
                ready[i]=ready[i/2];
                i=i/2;
}

pro delete_sjf(pro ready[], int *size) {
        ready[1]을 temp에 저장;
        ready[size]를 ready[1]에 삽입;
        parent =1; child=2;
        while {
                if(ready[parent].cpuburst <= ready[child].cpuburst) break;
```

```
                child와 parent를 교체;
                parent = child;
                child = child*2;
        }

}

insert_pri(pro ready[], pro item, int* size) {
        size = size+1;
        i=size;
        ready[i]=item;
        while(i!=1 && ready[i].priority > ready[i/2].priority)
                ready[i]=ready[i/2];
                i=i/2;
}

pro delete_pri(pro ready[], int *size) {
        ready[1]을 temp에 저장;
        ready[size]를 ready[1]에 삽입;
        parent =1; child=2;
        while {
                if(ready[parent].priority >= ready[child].priority) break;
                child와 parent를 교체;
                parent = child;
                child = child*2;
        }

}

evaluation(int gantt[], int ganttc) {
        waiting time = 프로세스의 종료시간 - cpu burst time - arrival time;
        turnaround time = 프로세스의 종료시간 - arrival time;
        간트 차트 출력;
}

display(int gantt[], int ganttc) {
        현재의 간트 차트 출력;
}
display_plist(void) {
        생성된 process list 목록 출력;
}
```

5. 실행 화면

```
park@park-VirtualBox:~$ ./simul
number of processes : 5
process id: 1, arrival time: 0, cpu burst time: 3, priority: 1
process id: 2, arrival time: 5, cpu burst time: 3, priority: 1
process id: 3, arrival time: 2, cpu burst time: 2, priority: 4
process id: 4, arrival time: 0, cpu burst time: 2, priority: 6
process id: 5, arrival time: 5, cpu burst time: 3, priority: 6
number of io processes: 1
process 1 number of io : 1
1. timing : 2
   length : 3
```

프로세스의 수와 io process의 횟수와 timing, length를 설정하는 화면이다.

```
<fcfs>
process id: 1, arrival time: 0, cpu burst time: 3, priority: 1
process id: 2, arrival time: 5, cpu burst time: 3, priority: 1
process id: 3, arrival time: 2, cpu burst time: 2, priority: 4
process id: 4, arrival time: 0, cpu burst time: 2, priority: 6
process id: 5, arrival time: 5, cpu burst time: 3, priority: 6
    1   1   4   4   3   3   2   2   2   5   5   5   1
  0   1   2   3   4   5   6   7   8   9  10  11  12  13
process1 waiting time : 10, turnaround time : 13
process2 waiting time :  1, turnaround time :  4
process3 waiting time :  2, turnaround time :  4
process4 waiting time :  2, turnaround time :  4
process5 waiting time :  4, turnaround time :  7

average waiting time : 3.800000
average turnaround time : 6.400000
---------------------------------
```

생성된 프로세스를 기반으로 fcfs를 수행한 화면이다. 아래에 프로세스 별로 waiting time과 turnaround time이 출력되고 맨 아래쪽에 average waiting time, average turnaround time이 출력된다.

```
<non_sjf>
process id: 1, arrival time: 0, cpu burst time: 3, priority: 1
process id: 2, arrival time: 5, cpu burst time: 3, priority: 1
process id: 3, arrival time: 2, cpu burst time: 2, priority: 4
process id: 4, arrival time: 0, cpu burst time: 2, priority: 6
process id: 5, arrival time: 5, cpu burst time: 3, priority: 6
    4   4   3   3   1   1   2   2   2   1   5   5   5
  0   1   2   3   4   5   6   7   8   9  10  11  12  13
process1 waiting time :  7, turnaround time : 10
process2 waiting time :  1, turnaround time :  4
process3 waiting time :  0, turnaround time :  2
process4 waiting time :  0, turnaround time :  2
process5 waiting time :  5, turnaround time :  8

average waiting time : 2.600000
average turnaround time : 5.200000
---------------------------------
```

생성된 프로세스를 기반으로 nonpreemtive-sjf를 수행한 화면이다. 아래에 프로세스 별로 waiting time과 turnaround time이 출력되고 맨 아래쪽에 average waiting time, average turnaround time이 출력된다.

```
<non_pri>
process id: 1, arrival time: 0, cpu burst time: 3, priority: 1
process id: 2, arrival time: 5, cpu burst time: 3, priority: 1
process id: 3, arrival time: 2, cpu burst time: 2, priority: 4
process id: 4, arrival time: 0, cpu burst time: 2, priority: 6
process id: 5, arrival time: 5, cpu burst time: 3, priority: 6
    4   4   3   3   1   1   5   5   5   2   2   2   1
  0   1   2   3   4   5   6   7   8   9  10  11  12  13
process1 waiting time : 10, turnaround time : 13
process2 waiting time :  4, turnaround time :  7
process3 waiting time :  0, turnaround time :  2
process4 waiting time :  0, turnaround time :  2
process5 waiting time :  1, turnaround time :  4

average waiting time : 3.000000
average turnaround time : 5.600000
---------------------------------
```

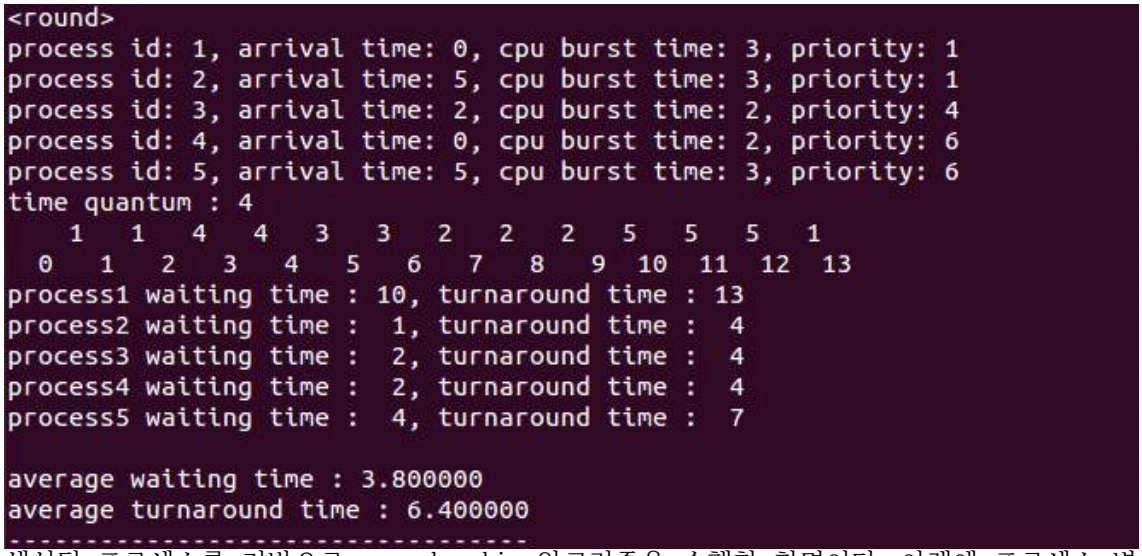
생성된 프로세스를 기반으로 nonpreemtive-priority 알고리즘을 수행한 화면이다. 아래에 프로세스 별로 waiting time과 turnaround time이 출력되고 맨 아래쪽에 average waiting time, average turnaround time이 출력된다.

```
<pre_sjf>
process id: 1, arrival time: 0, cpu burst time: 3, priority: 1
process id: 2, arrival time: 5, cpu burst time: 3, priority: 1
process id: 3, arrival time: 2, cpu burst time: 2, priority: 4
process id: 4, arrival time: 0, cpu burst time: 2, priority: 6
process id: 5, arrival time: 5, cpu burst time: 3, priority: 6
    4   4   3   3   1   1   2   2   2   1   5   5   5
  0   1   2   3   4   5   6   7   8   9  10  11  12  13
process1 waiting time :  7, turnaround time : 10
process2 waiting time :  1, turnaround time :  4
process3 waiting time :  0, turnaround time :  2
process4 waiting time :  0, turnaround time :  2
process5 waiting time :  5, turnaround time :  8

average waiting time : 2.600000
average turnaround time : 5.200000
---------------------------------
```

생성된 프로세스를 기반으로 preemtive-sjf 알고리즘을 수행한 화면이다. 아래에 프로세스 별로 waiting time과 turnaround time이 출력되고 맨 아래쪽에 average waiting time, average turnaround time이 출력된다.

```
<pre_pri>
process id: 1, arrival time: 0, cpu burst time: 3, priority: 1
process id: 2, arrival time: 5, cpu burst time: 3, priority: 1
process id: 3, arrival time: 2, cpu burst time: 2, priority: 4
process id: 4, arrival time: 0, cpu burst time: 2, priority: 6
process id: 5, arrival time: 5, cpu burst time: 3, priority: 6
    4   4   3   3   1   5   5   5   2   2   2   1   0   0   0   1
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
process1 waiting time : 13, turnaround time : 16
process2 waiting time :  3, turnaround time :  6
process3 waiting time :  0, turnaround time :  2
process4 waiting time :  0, turnaround time :  2
process5 waiting time :  0, turnaround time :  3

average waiting time : 3.200000
average turnaround time : 5.800000
---------------------------------
```

생성된 프로세스를 기반으로 preemtive-priority 알고리즘을 수행한 화면이다. 아래에 프로세스 별로 waiting time과 turnaround time이 출력되고 맨 아래쪽에 average waiting time, average turnaround time이 출력된다.



생성된 프로세스를 기반으로 round robin 알고리즘을 수행한 화면이다. 아래에 프로세스 별로 waiting time과 turnaround time이 출력되고 맨 아래쪽에 average waiting time, average turnaround time이 출력된다.

6. 성능 비교

fcfs - average waiting time : 3.8 / average turnaround time : 6.4
non-sjf - average waiting time : 2.6 / average turnaround time : 5.2
non-pri - average waiting time : 3 / average turnaround time : 5.6
pre-sjf - average waiting time : 2.6 / average turnaround time : 5.2
pre-pri - average waiting time : 3.2 / average turnaround time : 5.8
round - average waiting time : 3.8 / average turnaround time : 6.4
위의 결과를 보면 nonpreemtive shortest job first 알고리즘과 preemtive shortest job first 알고리즘의 성능이 가장 뛰어난 것을 알 수 있다.

7. 프로젝트 수행 소감

cpu scheduling 알고리즘을 이론으로만 접했을 때는 약간 난해하고 이해가 잘 안 된 부분이 있었는데 직접 스케줄러를 구현하면서 스케줄링의 원리와 순서를 더 잘 이해할 수 있게 되었다. 또한 코딩을 하면서 예상하지 못한 에러도 많이 발생했는데, 이런 에러들을 해결하면서 프로그래밍 실력 또한 향상되었다고 느꼈다. 앞으로도 수업시간에 배운 내용들을 실습을 통해 더 확실히 익혀야겠다는 생각이 들었다.