COSE341-01 운영체제 Term Project

CPU Scheduling Simulator

2016320128 컴퓨터학과 이수민



목차

* CPU Scheduling
* 기존 시뮬레이터 소개
* 본인이 설계한 시뮬레이터 설명
* 실행 결과
* 알고리즘 간 비교분석
* 프로젝트 소감 및 추후 발전 계획

1. **CPU Scheduling**
   1. **CPU Scheduling의 정의**

어떤 프로세스가 실행을 하다가 interrupt가 발생했을 때, multiprogramming을 하지 않는 시스템에서는 CPU가 해당 프로세스의 interrupt completion을 기다리며 idle상태에 있다. 이런 방식은 CPU 연산의 효율성을 떨어뜨리기 때문에 CPU가 idle 상태가 될 때마다 실행 준비중인 프로세스를 CPU에 할당하여 CPU 이용 효율을 극대화 하고자 하는 방식이 CPU scheduling이다.

CPU 스케줄링 결정은 다음의 네 가지 상황에서 발생 가능하다.

1. 프로세스가 종료했을 때
2. 프로세스에 I/O interrupt가 발생하였을 때
3. I/O가 완료되었을 때
4. 새로운 프로세스가 ready 상태로 진입했을 때

만약 스케줄링 알고리즘이 1,2의 상황에서만 context switch를 한다면 그 알고리즘은 비선점형 스케줄링 알고리즘이다. 비선점형 스케줄링 알고리즘은 context switch 횟수가 상대적으로 적기 때문에 context switch overhead가 작은 편이다.

반면, 1,2,3,4 모든 상황에서 context switch가 발생한다면 그 알고리즘은 선점형 스케줄링 알고리즘이다. 선점형 스케줄링 알고리즘에서는 어떤 프로세스가 현재 실행중인 프로세스를 쫓아내고 CPU를 차지할 수 있다. 이 방식은 비선점형 방식보다 응답 속도가 빠를 수 있다.

* 1. **CPU Scheduling Algorithm을 평가하는 기준**

다양한 CPU Scheduling 알고리즘이 존재하기 때문에 어떤 알고리즘을 선택할 지 결정하기 위해서는 평가기준이 필요하다.

1. **CPU utilization**

CPU utilization은 전체 수행 시간 중 CPU가 idle하지 않고 실제로 프로세스를 수행 한 시간의 비율을 말한다. 비율이 높을 수록 높은 효율을 가지고있는 알고리즘이다.

1. **Throughput**

Throughput은 단위시간당 종료된 프로세스의 개수를 이야기한다. 프로세스의 길이가 길수록 throughput이 낮아진다. 높은 throughput은 많은 프로세스가 실행되고있음을 의미한다.

1. **Waiting Time**

Waiting Time은 프로세스가 ready queue에 넘겨진 순간부터 종료될 때까지 ready queue에서 기다린 시간이다. 이 값이 작을 수록 CPU scheduler가 긍정적으로 평가된다.

1. **Turnaround Time**

Turnaround Time은 프로세스가 도착한 순간부터 종료한 순간까지 걸린 시간을 의미한다. 이때의 시간은 ready queue에서의 대기시간, CPU에서의 수행 시간, I/O 수행 시간을 모두 포함한다.

1. **Response Time**

Response Time은 프로세스가 도착한 순간부터 처음 CPU에 할당되어 실행하기 까지 ready queue에서 기다린 시간을 의미한다. 사용자와의 상호작용이 중요한 시스템일 수록 response time이 작은 것이 중요하다.

위와 같이 다양한 평가기준을 하나의 스케줄링 알고리즘이 모두 완벽하게 만족하는 것은 불가능하다. 따라서 시스템의 특성에 따라 어떤 평가기준에 가중치를 두어 알고리즘을 선택하기도 하고, 여러 알고리즘을 조합해서 사용하여 더 높은 효율을 끌어내기도 한다.

* 1. **CPU Scheduling Algorithms**

스케줄링 알고리즘은 ready queue에 있는 프로세스 중 어떤 프로세스를 먼저 CPU에 할당할 것인지 특정 기준에 따라 우선순위를 부여하여 결정한다.

1. **First-Come, First-Served Scheduling(FCFS)**

FCFS 알고리즘은 ready queue에 먼저 도착한 순서대로, 즉 CPU 사용 요청을 먼저 한 순서대로 프로세스를 CPU에 할당하는 알고리즘이다. Non-preemptive 방식이기 때문에 프로세스가 종료되거나 I/O interruption이 발생하지 않으면 다른 프로세스가 CPU를 강탈할 수 없다. FCFS 알고리즘은 가장 간단한 알고리즘 중 하나로 구현이 쉽고, 요청 순서대로 실행을 하기 때문에 starvation 현상이 일어나지 않는다는 장점이 있다. 하지만 앞 순서에 긴 실행시간을 가진 프로세스가 실행을 하면 뒷 순서의 짧은 실행시간을 가진 프로세스들이 불필요하게 대기를 오래하는 convoy effect가 일어날 수 있다는 단점이 있다. Convoy effect는 CPU utilization을 낮춘다.

1. **Shortest Job First Scheduling**

SJF 알고리즘은 가장 CPU burst time이 짧은 프로세스부터 CPU에 할당하는 알고리즘이다.

1. **Non-preemptive**

비선점 SJF는 먼저 수행시간이 긴 프로세스가 CPU할당을 요청하고, 곧 이어 수행시간이 짧은 프로세스가 CPU할당을 요청해도, CPU를 바꿔줄 수 없으므로 SJF 방식의 이점이 크게 작용하지 못한다.

1. **Preemptive**

1)의 문제 때문에 SJF 알고리즘이 preemptive 특성을 가지게 설계를 하면, 각 context switch 상황마다 가장 CPU burst remain time이 작은 프로세스가 CPU에 할당이 된다. 이는 average waiting time 측면에서 최소를 보장해주는 최적의 해결방안이 된다. 다만 실제 시스템 환경에서는 CPU burst time을 알 수 없고, 예측을 하는 것도 어렵기 때문에 실제 시스템 환경에서 쓰이지 못한다. 또한, 짧은 수행 시간을 가진 프로세스만 계속 우선권을 부여받기 때문에 수행시간이 짧은 프로세스들이 지속적으로 CPU할당을 요청하는 한, 긴 수행 시간을 가진 프로세스는 CPU 할당을 받지 못하고 무한히 순서에서 밀려나며 starvation이 발생한다.

1. **Priority Scheduling**

Priority 알고리즘은 프로세스 생성 때 프로세스의 특성에 따라 우선순위를 부여하고, 우선순위가 높은 프로세스부터 CPU에 할당하는 알고리즘이다.

1. **Non-preemptive**

비선점 priority scheduling은 우선순위가 낮은 프로세스가 먼저 CPU할당을 요청하고, 곧 이어 우선순위가 높은 프로세스가 CPU할당을 요청해도, CPU를 바꿔줄 수 없으므로 SJF 방식의 이점이 크게 작용하지 못한다.

1. **Preemptive**

1)의 문제 때문에 priority scheduling은 선점형으로 많이 쓰인다. 프로세스 특성에 따라 우선순위가 부여되기 때문에 중요한 프로세스부터 실행시킬 수 있다는 장점이 있지만 preemptive SJF과 마찬가지로 starvation문제가 발생한다. 이를 해결하기 위한 방안으로 aging 기법을 도입한다. Aging기법은 프로세스의 대기시간이 일정 값을 넘으면 우선순위를 높여주어 우선순위가 낮은 프로세스도 유한 시간 내에 CPU 할당을 받을 수 있도록 하는 방식이다.

1. **Round Robin Scheduling**

Round Robin scheduling은 FCFS처럼 ready queue에 도착한 순서대로 프로세스를 실행하지만, time quantum이라고 정해진 시간 동안만 수행하고 다음 프로세스로 preemption이 일어나는 스케줄링 알고리즘이다. 이 방식은 특히 한 컴퓨터의 자원을 여러 유저 프로그램이 동시에 사용하고자 하는 시분할 시스템에서 유용하게 쓰인다. Time quantum을 작게 설정하면 동시에 실행되는 것처럼 느낄 수 있기 때문이다. 하지만 여러 프로세스들을 번갈아 수행하기 위해 context swtich가 자주 일어나기 때문에 overhead가 커져서 효율이 떨어질 수 있다는 단점이 있다.

1. **Multilevel Queue Scheduling**

MQS는 서로 다른 유형에 속하는 프로세스들을 나눠서 유형 별로 다른 ready queue에 넣고, 각 ready queue에는 우선순위를 부여하여 우선순위가 높은 queue의 프로세스들부터 실행을 하는 알고리즘이다. 이 알고리즘에서 특징적인 점은 각 queue마다 서로 다른 스케줄링 알고리즘을 적용할 수 있다는 점이다. 예를 들면 foreground queue에는 round robin 알고리즘을 적용하고, background queue에는 FCFS 알고리즘을 적용하는 식으로 구현이 가능하다.

또한 알고리즘에서는 우선순위가 높은 queue의 프로세스의 수행이 전부 완료되어야 그 다음 우선순위를 가진 queue의 프로세스들이 수행을 할 수 있다. 따라서 낮은 우선순위 queue 에 있는 프로세스들이 starvation을 경험할 수 있다.

1. **Multilevel Feedback Queue Scheduling**

MQS 에서 일어나는 starvation문제를 해결하기 위해 고안된 알고리즘이 MFQS 알고리즘이다. 너무 오랫동안 실행되지 못한 프로세스를 우선순위가 한 단계 높은 queue로 이동시키는 것으로 aging 기법을 구현하여 starvation이 일어나는 것을 막는다.

1. **기존 시뮬레이터 소개**

기존의 시뮬레이터[1]에서 사용한 알고리즘은 총 다섯 가지로, First-Come, First-Serve(FCFS), Round-Robin(RR), Shortest-Job-First(SJF), Shortest-Remaining-Time-First(SRTF), Multilevel Feedback Queues(MLFQ) 였다. 사용자는 프로세스 세부사항(CPU burst time, I/O burst time등)을 직접 설정하거나 랜덤으로 생성되게 할 수 있었다. 특별한 이벤트가 일어날때마다 이벤트 메세지가 생성되었다. 여러가지 다른 CPU Scheduling 알고리즘의 성능을 나타내기 위해 각 프로세스별로 해당 시간에 CPU에서 실행이 되었는지, ready queue에서 대기하고 있었는지, I/O 완료를 기다리며 waiting queue에 있었는지를 나누어 간트 차트에 표현했다.

1. **본인이 설계한 시뮬레이터 설명**

본인이 설계한 시뮬레이터의 모듈은 크게 세가지의 종류로 나누어볼 수 있다. 첫번째는 전역 변수로 선언된 큐를 관리하기 위한 함수들, 큐를 정렬하기 위한 mergesort 함수, 프로세스를 랜덤 생성하기 위한 create\_processes() 함수 등의 기본적인 기능을 구현하기 위한 모듈들이다. 두번째는 실제로 알고리즘이 실행되는데 필요한 함수들이다. FCFS\_alg(), SJF\_alg(), PRI\_alg(), PRESJF\_alg(), PREPRI\_alg(), RR\_alg(), MULTI\_Q() 에서 각 알고리즘이 실행되며 0부터 시간을 재면서 어떤 프로세스가 CPU에서 실행중인지를 출력한다. Wait() 함수는 한 프로세스를 실행하는 동안 ready queue에서 대기 중인 프로세스들의 waiting time을 1씩 증가시키는 함수이다. Waiting() 함수는 waiting queue가 비어있지 않다면 매 시간 I/O burst 시간을 1씩 감소시키다가, I/O burst time remain이 0이 되는 순간 해당 프로세스를 ready queue로 옮겨주는 함수이다. 세번째 종류는 평가시간을 출력하는 함수이다. 알고리즘 함수에서 각 프로세스에 평가 시간이 저장되고, 해당 알고리즘의 cpu utilization도 계산이 된다. Evalutation() 함수는 각 프로세스마다의 평가 시간을 출력하고, TotalEval() 함수는 각 알고리즘별로 평가 시간의 평균과 cpu utilization을 출력한다.

1. FCFS\_alg의 pseudo code

runP = NULL

for nowTime = 0; check < num\_process; nowTime++

If process arrived from job queue

Then poll process from job queue to ready queue

If ready queue is not empty

Then poll process from ready queue to runP

If runP == NULL and waiting queue is empty

Then print “bb ”

Else if runP == NULL and waiting queue is not empty

Then print “bb ”

Call waiting(ARRIVAL)

Else if runP is not NULL

Print process ID of runP

Call wait(runP’s pid) and waiting(ARRIVAL)

If cpu burst remain time of runP is 0

Then terminate runP

Check++

I/O happens for 5% chance

Pseudo code이므로 알고리즘과 직접 관련된 부분만 표기했다. 기본 구조는 모든

프로세스가 완료될 때까지 for문을 반복하는 것이다. 프로세스가 도착하면 ready queue에 들어가고, ready queue가 비어있지 않으면 NULL로 초기화 되어있던 runP 변수로 프로세스가 저장된다. runP 변수는 현재 CPU에서 실행되고 있는 프로세스를 나타낸다. 따라서 runP 가 NULL이고, waiting queue도 비어있으면, 아직 아무 프로세스도 ready queue 에 도착하지 않은 상태로, “bb “만 출력해준다. CPU가 비어있고, waiting queue가 차있는 상태는 모든 프로세스가 ready queue로 옮겨졌지만, 모두 실행을 마치거나 I/O가 발생한 것이다. 따라서 CPU가 idle하다는 표시로 “bb “를 출력해주고, waiting queue에 있는 I/O burst가 진행될 수 있도록 waiting()함수를 불러준다. CPU에 프로세스가 할당되어있다면 해당 프로세스의 id를 간트 차트에 출력해주고, ready queue, waiting queue가 진행이 될 수 있도록 wait(), waiting() 함수를 호출한다. 해당 프로세스의 cpu burst remain 시간이 0이 되면 해당 프로세스를 종료시키고 종료 표시로 check++을 해준다. 이후 5% 확률로 I/O 발생이 일어나도록 한다.

이후 나올 알고리즘 함수들의 구조도 FCFS의 구조와 흡사하다. 알고리즘에 따라 preemption이 추가가 되거나 ready queue를 정렬하는 방법이 달라질 뿐이다.

1. SJF\_alg의 pseudo code

runP = NULL

for nowTime = 0; check < num\_process; nowTime++

If process arrived from job queue

Then poll process from job queue to ready queue

If ready queue is not empty

Then sort ready queue by CPU burst time

poll process from ready queue to runP

If runP == NULL and waiting queue is empty

Then print “bb ”

Else if runP == NULL and waiting queue is not empty

Then print “bb ”

Call waiting(CPU REMAIN)

Else if runP is not NULL

Print process ID of runP

Call wait(runP’s pid) and waiting(CPU REMAIN)

If cpu burst remain time of runP is 0

Then terminate runP

Check++

I/O happens for 5% chance

Shortest Job First 알고리즘은 CPU burst time을 기준으로 선택을 하는 알고리즘이므로 FCFS과 구조는 동일하되, queue를 CPU burst time으로 오름차순 정렬한다. Non-preemptive algorithm 이므로 달리 바뀌는 점은 없다.

1. PRI\_alg의 pseudo code

runP = NULL

for nowTime = 0; check < num\_process; nowTime++

If process arrived from job queue

Then poll process from job queue to ready queue

If ready queue is not empty

Then sort ready queue by priority

poll process from ready queue to runP

If runP == NULL and waiting queue is empty

Then print “bb ”

Else if runP == NULL and waiting queue is not empty

Then print “bb ”

Call waiting(PRIORITY)

Else if runP is not NULL

Print process ID of runP

Call wait(runP’s pid) and waiting(PRIORITY)

If cpu burst remain time of runP is 0

Then terminate runP

Check++

I/O happens for 5% chance

PRI\_alg()의 알고리즘은 비선점형 우선순위 알고리즘이므로 SJF과 크게 다르지 않다. 숫자가 작을 수록 우선순위가 높으므로 우선순위 오름차순으로 ready queue를 정렬하고 프로세스를 선택한다.

1. PRESJF\_alg의 pseudo code (preemptive SJF alg)

runP = NULL

for nowTime = 0; check < num\_process; nowTime++

If process arrived from job queue

Then poll process from job queue to ready queue

If new process has shorter CPU burst time than current process,

Then context switch between new and current processes

If ready queue is not empty

Then sort ready queue by CPU burst time

poll process from ready queue to runP

If runP == NULL and waiting queue is empty

Then print “bb ”

Else if runP == NULL and waiting queue is not empty

Then print “bb ”

Call waiting(CPU REMAIN)

Else if runP is not NULL

Print process ID of runP

Call wait(runP’s pid) and waiting(CPU REMAIN)

If process returned from waiting queue has shorter cpu burst time than current process, context swtich occurs.

If cpu burst remain time of runP is 0

Then terminate runP

Check++

I/O happens for 5% chance

Non-preemptive SJF와 형광색으로 표기한 부분이 다르다. Preemptive scheduling 알고리즘에서는 새로운 프로세스가 도착했을 때, I/O를 완료한 프로세스가 ready queue로 돌아왔을 때 preemption이 일어난다.

1. PREPRI\_alg의 pseudo code(preemptive priority alg)

runP = NULL

for nowTime = 0; check < num\_process; nowTime++

If process arrived from job queue

Then poll process from job queue to ready queue

If new process has HIGHER(smaller number) priority than current process, Then context switch between new and current processes

If ready queue is not empty

Then sort ready queue by priority

poll process from ready queue to runP

If runP == NULL and waiting queue is empty

Then print “bb ”

Else if runP == NULL and waiting queue is not empty

Then print “bb ”

Call waiting(PRIORITY)

Else if runP is not NULL

Print process ID of runP

Call wait(runP’s pid) and waiting(PRIORIRY)

If process returned from waiting queue has HIGHER(smaller number) priority than current process, context swtich occurs.

If cpu burst remain time of runP is 0

Then terminate runP

Check++

I/O happens for 5% chance

Preemptive SJF과 같은 코드이나, 우선순위를 기준으로 ready queue를 정렬한다.

1. RR\_alg의 pseudo code

runP = NULL

for nowTime = 0; check < num\_process; nowTime++

If process arrived from job queue

Then poll process from job queue to ready queue

If ready queue is not empty

Then sort ready queue by CPU burst time

poll process from ready queue to runP

If runP == NULL and waiting queue is empty

Then print “bb ”

Else if runP == NULL and waiting queue is not empty

Then print “bb ”

Call waiting(CPU REMAIN)

Else if runP is not NULL

Print process ID of runP

Call wait(runP’s pid) and waiting(CPU REMAIN)

If cpu burst remain time of runP is 0

Then terminate runP

Check++

If runP’s time quantum expired

Context switch occurs

I/O happens for 5% chance

Round robin 알고리즘은 FCFS와 동일하나 프로세스가 실행할 때, time quantum이 주어진다. Time quantum을 다 쓰면 다음 프로세스에게 순서가 넘어간다.

1. MULTI\_Q의 pseudo code

runP = NULL

for nowTime = 0; check < num\_process; nowTime++

If process arrived from job queue

If process has upper 50% priority, add process to foreground queue

Else add process to background queue

For every process in background queue

If process’s waiting time > 20 then move process to foreground queue

If foreground queue is not empty

Then sort foreground queue by arrival time

poll process from foreground queue to runP

else if background queue is not empty

then sort background queue by arrival time

poll process from background queue to runP

If runP == NULL and waiting queue is empty

Then print “bb ”

Else if runP == NULL and waiting queue is not empty

Then print “bb ”

Call waiting(CPU REMAIN)

Else if runP is not NULL

Print process ID of runP

Call wait(runP’s pid) and waiting(CPU REMAIN)

If cpu burst remain time of runP is 0

Then terminate runP

Check++

If runP’s time quantum expired

Context switch occurs

I/O happens for 5% chance

Multilevel feedback queue에서는 process가 여러 개의 queue로 나누어져 들어간다. 여기서는 우선순위가 높은 프로세스를 foreground queue로 넣고, 나머지를 background queue로 넣었다. Foreground queue의 프로세스가 모두 실행되어야 background queue의 프로세스들이 실행될 수 있어 starvation의 우려가 있기 때문에 background queue에 있는 프로세스 중 waiting time이 20을 넘은 프로세스는 foreground queue로 이동되도록 aging 기법을 적용했다.

1. **실행결과**
   1. **프로세스 생성**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

6개의 프로세스를 생성했다. I/O는 프로세스 실행 중에 CPU burst time과 비례하여 랜덤으로 발생한다. Time quantum은 3으로 설정했다.

* 1. **각 알고리즘 별 실행결과**

시작 시간이 0일때부터 모든 프로세스가 종료될 때까지 해당 시간에 어떤 프로세스가 실행 중인지를 출력하여 간트 차트를 만든다. 실행중에 I/O interruption이 발생하면 프로세스의 id와 I/O burst time을 표기하고, I/O가 끝나면 프로세스의 id와 CPU burst remain time을 표기해준다. 프로세스가 모두 완료된 후에는 각 프로세스의 waiting time, turnaround time, response time을 표시해주고, 해당 스케줄링 알고리즘에서 각 평가 시간의 평균과 CPU utilization을 계산하여 출력한다.

* + 1. **FCFS**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

프로세스가 도착하는 순서대로 수행했다. 프로세스가 아직 도착하지 않았을때만 idle하기 때문에 CPU utilization이 높다. 중간에 I/O가 발생한 프로세스들이 turnaournd time이 높음을 확인할 수 있다.

* + 1. **SJF**

**텍스트이(가) 표시된 사진

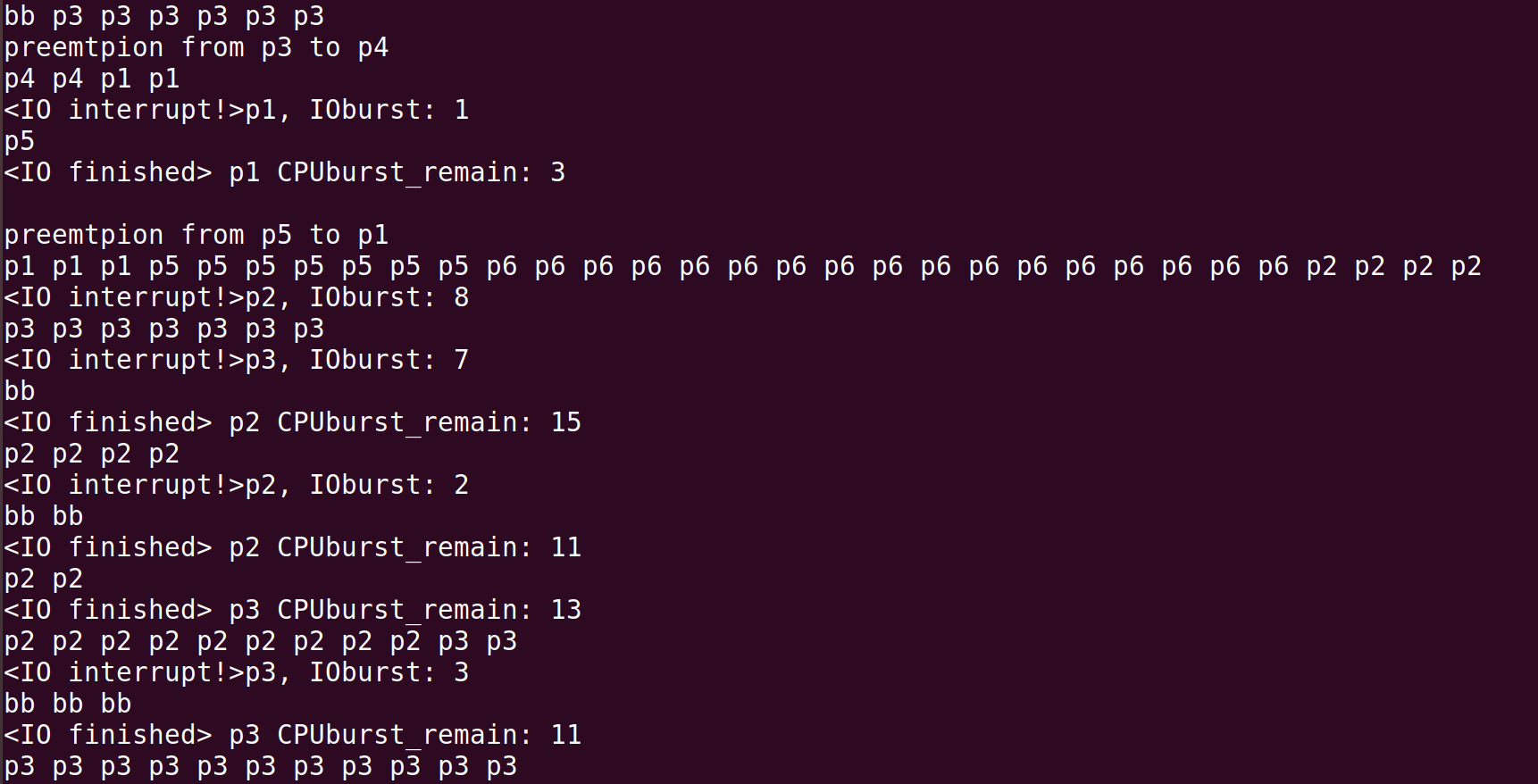
자동 생성된 설명**

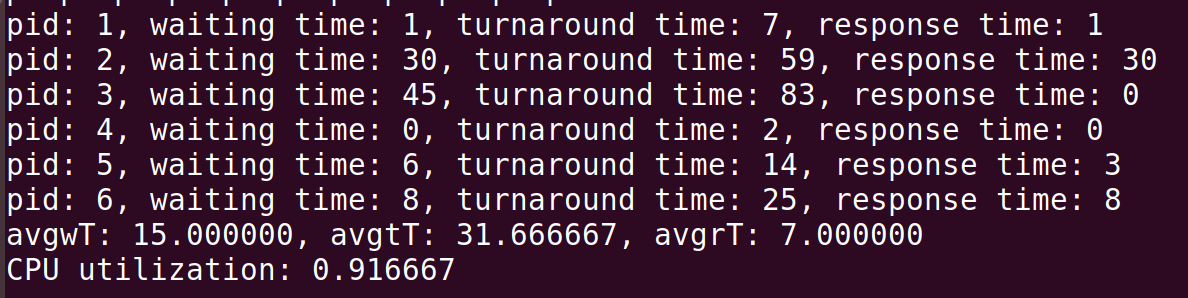
Non-preemptive Shortest Job First 알고리즘이기 때문에 CPU burst time이 작은 순서대로 실행되는 것이 원칙이다. 다만 도착 시간이 차이가 나기 때문에 도착 시간이 가장 이른 프로세스가 가장 먼저 수행된다. 모든 프로세스가 도착한 이후에 프로세스들은 CPU burst time이 작은 순서대로 CPU에 할당된다. Ready queue가 비어있는 상태에서 I/O가 발생했기 때문에 CPU가 idle 상태가 된다. 이는 CPU utilization을 낮추는 원인이 된다.

* + 1. **PRIORITY텍스트이(가) 표시된 사진

       자동 생성된 설명**

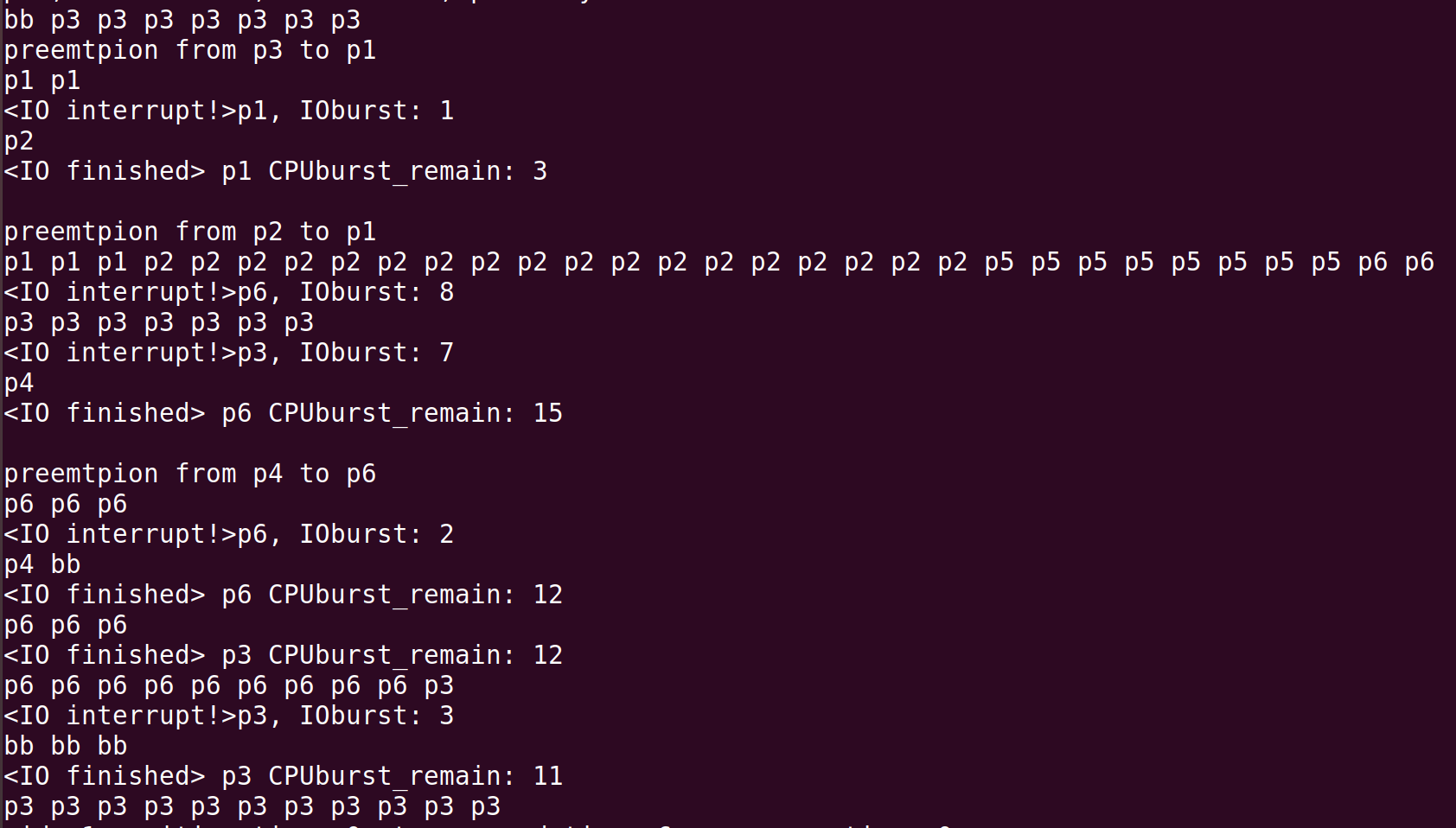
Non-preemptive priority 알고리즘이기 때문에 우선순위가 높은 순서대로 실행되는 것이 원칙이다. 다만 도착 시간이 차이가 나기 때문에 도착 시간이 가장 이른 프로세스부터 실행이 되고, 모든 프로세스가 도착한 이후부터는 우선순위가 높은 프로세스부터 실행이 된다. 우선순위가 같은 프로세스들은 CPU burst time, CPU burst time도 같으면 arrival time, arrival time도 같으면 process id순서로 비교하여 프로세스를 선택하여tie-breaking을 한다.

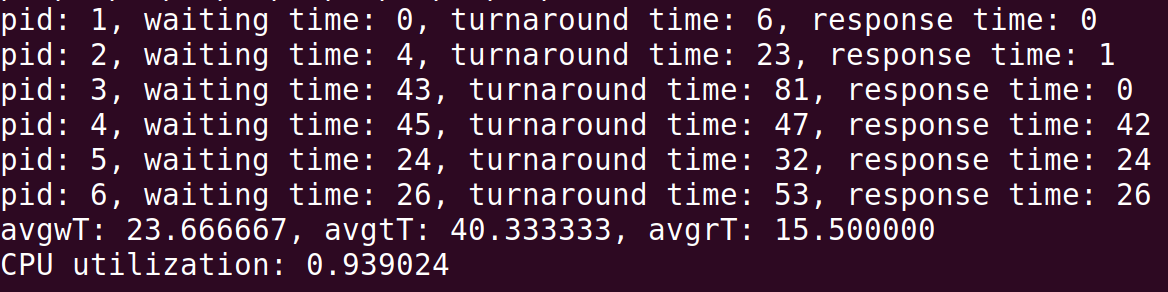
* + 1. **PREEMPTIVE SJF**



Preemptive Shortest Job Fisrt 알고리즘이기 때문에 preemption 상황마다 CPU burst remain time이 가장 작은 프로세스를 선택한다. Preemptive 알고리즘이기 때문에 도착시간이 제일 이르고 수행시간이 긴 p3가 수행을 전부 마칠 때까지 기다리지 않고 CPU burst remain time이 더 짧은 p4가 도착했을 때 CPU 할당을 바꿔준다. 이후로도 기준에 따라 CPU burst remain time이 짧은 프로세스 순서대로 실행이 된다. Ready queue가 비어있는 상태에서 I/O가 발생했기 때문에 CPU가 idle 상태가 된다. 이는 CPU utilization을 낮추는 원인이 된다.

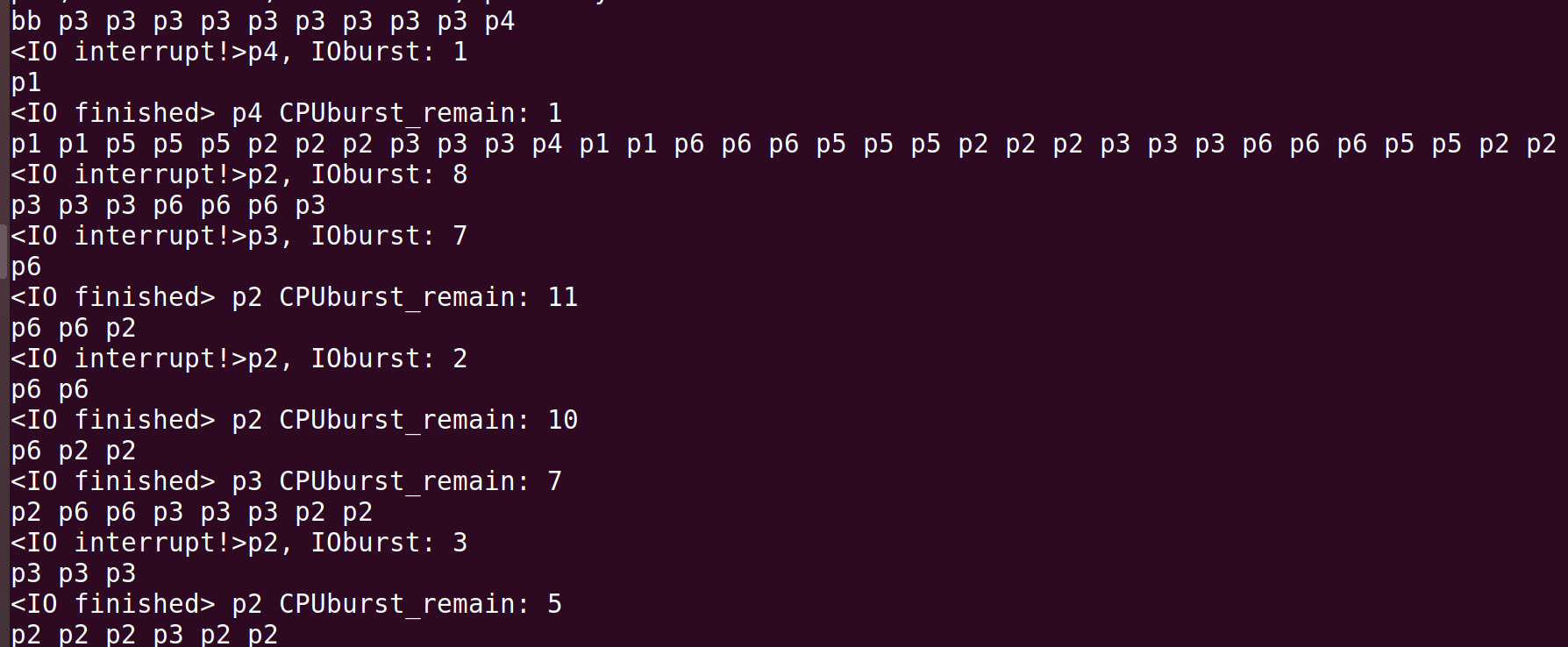
* + 1. **PREEMPTIVE PRIORITY**

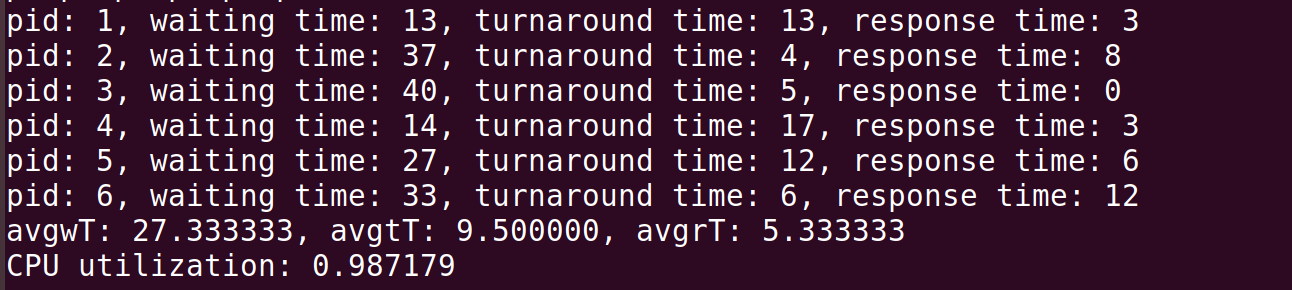
****

****

Preemptive Priority 알고리즘이기 때문에 preemption 상황마다 우선순위가 가장 높은 프로세스를 선택한다. 우선순위가 같다면 CPU burst remain time이 가장 작은 것을 골라 tie-breaking을 해준다. 따라서 가장 먼저 도착한 프로세스인 p3가 실행되다가 더 우선순위가 높은 p1이 도착하자 CPU 할당을 넘겨준다. 이후로도 같은 방식으로 프로세스를 실행한다. Ready queue가 비어있는 상태에서 I/O가 발생했기 때문에 CPU가 idle 상태가 된다. 이는 CPU utilization을 낮추는 원인이 된다.

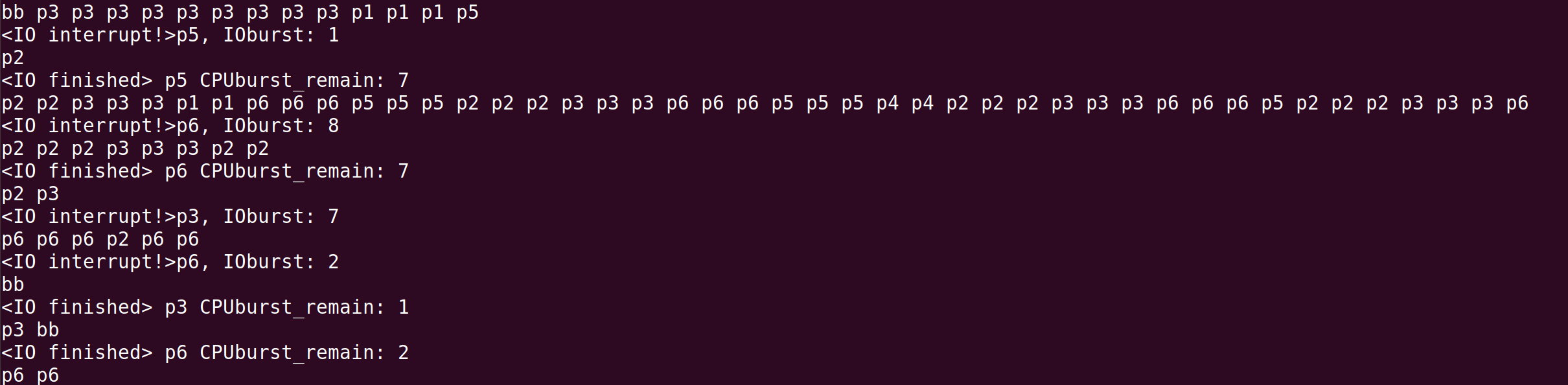
* + 1. **ROUND ROBIN**

****

****

Round Robin은 FCFS처럼 ready queue에 도착한 순서대로 실행이 되지만 time quantum 이 모두 소진되면 다름 프로세스로 CPU 자원을 넘겨준다. 프로세스 생성 과정에서 time quantum을 3으로 정했기 때문에 3번씩 실행이 될때마다 프로세스가 바뀐다. 프로세스가 도착하지 않은 시기만 CPU가 idle상태이다.

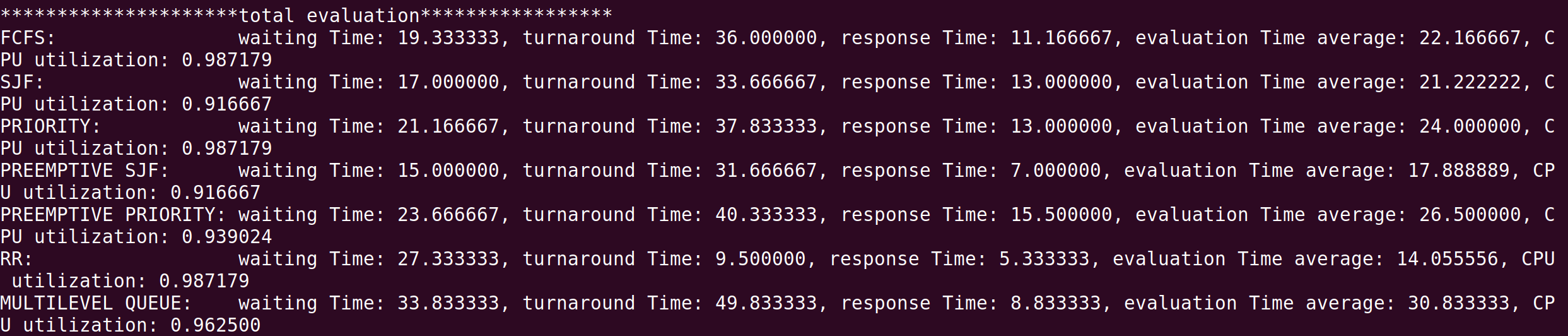
* + 1. **MULTILEVEL FEEDBACK QUEUE**

****

**텍스트, 실내이(가) 표시된 사진

자동 생성된 설명**

Multilevel Feedback queue 스케줄링은 우선순위에 따라서 프로세스를 두 개의 queue에 나눠서 넣는다. 가능한 우선순위 중 상위 50%의 번호를 가진 프로세스는 foreground queue에, 하위 50% 번호를 가진 프로세스는 background queue에 넣는다. 따라서 이 예제에서는 p4를 제외한 모든 프로세스가 foreground queue에 위치한다. Background queue에 위치한 프로세스들은 foreground queue가 모두 비어야 실행이 가능하기 때문에 starvation의 위험이 있다. 따라서 waiting time이 20을 넘었을 때 p4를 foreground queue로 옮겨준다. 두 queue에 서로 다른 스케줄링 알고리즘을 적용할 수도 있으나 여기서는 스케줄링 알고리즘은 동일하게 time quantum이 3인 round robin 스케줄링 알고리즘을 적용했다. 프로세스가 도착하지 않은 시기만 CPU가 idle상태이다.

* 1. **총 실행결과**

각 평가 시간 별로 스케줄링 알고리즘을 비교해볼 수 있다. Waiting time은 optimal solution인 preemptive SJF가 15로 가장 작게 나왔고, time quantum이 끝날때마다 context switch를 해줘야하는 round robin이 33.8333으로 가장 길게 나왔다. Turnaround time의 경우, round robin 알고리즘이 9.5로 특별히 작게 나왔고, 나머지 알고리즘들은 빠르면 31, 느리면 49의 시간을 보였다. Round robin 알고리즘은 프로세스들이 돌아가면서 실행을 하기 때문에 더 빠른 시간에 종료가 가능한 것으로 보인다. Response time은 preemptive SJF가 7, round-robin이 5.3333로 좋은 성능을 보였다. Preemptive SJF는 수행 시간이 짧은 프로세스부터 실행하기 때문에 상대적으로 좋은 response time을 보이고, round robin은 프로세스들이 서로 돌아가면서 수행하기 때문에 응답시간이 짧은 것으로 보인다. 모든 평가시간을 고려해보았을 때, Round robin이 고르게 좋은 평가시간을 보인다.

1. **알고리즘 간 비교분석**

Non-preemptive SJF, Priority 알고리즘은 FCFS와 크게 다르지 않은 성능을 보이는 것을 확인할 수 있었다. 이는 프로세스 선택 기준에 더 부합하는 프로세스가 ready queue에 도착해도 이미 수행중인 프로세스에게서 CPU를 강탈 할 수 없기 때문이다. 각 알고리즘의 프로세스 선택기준보다 ready queue 도착시간이 성능을 좌지우지할 수 있다.

Preemptive SJF와 preemptive Priority 스케줄링 알고리즘은 성능이 비슷할 것으로 예상했으나 preemptive SJF의 성능이 preemptive priority보다 월등히 좋았다. 이는 priority가 프로세스 수행 시간을 고려하지 않고 정해지기 때문에 상대적으로 waiting time과 turnaround time이 더 길어지기 때문이다.

Round Robin의 성능은 preemptive SJF과 비슷했으나 여러 시뮬레이션을 해본 결과 waiting time은 preemptive SJF보다 안 좋지만 turnaround time, response time 측면에서 성능이 월등했다.

1. **프로젝트 소감 및 추후 발전 계획**
   1. **프로젝트 소감**

CPU Scheduler Simulator 라는 큰 프로그램을 구현하면서 프로그래밍 실력이 향상되었다. 항상 코딩 테스트 수준의 짧은 프로그램만 구현해보다가 1000줄 이상의 긴 코드를 어떻게 구현해야 할 지 구조를 궁리하고 고민하는 값진 경험이 되었다. CPU 스케줄링 알고리즘을 직접 구현해보면서 여러 스케줄링 알고리즘에 대한 이해도도 깊어졌다. 직접 구현한 시뮬레이터로 여러 스케줄링 알고리즘의 성능을 비교해보니 활자로만 이해했던 선점형과 비선점형 알고리즘의 성능 차이도 한눈에 볼 수 있었다.

* 1. **추후 발전 계획**

현재는 모든 queue를 전역 변수로 설정해서 queue가 여러 개인 multilevel queue scheduling을 구현할 때 어려움이 있었다. Queue를 지역 변수로 설정하는 것으로 구조를 수정한다면 multilevel queue scheduling이나 multi-processor 환경을 구성하기에 용이할 것이다.

현재는 single processor 환경만 지원하지만 이후 보완을 통해 multi-processor 환경을 지원하도록 할 것이다.

Rate Monotonic Scheduling, Earliest Deadline First Scheduling과 같은 real-time 시스템 환경을 위한 스케줄링 알고리즘도 시뮬레이션 해볼 수 있도록 추가할 것이다.

참고 문헌

[1] A CPU Scheduling Algorithm Simulator(Sukanya Suranauwarat)