CPU Scheduler Analysis

22311947 김서현
1254 운영체제

Ⅰ. CPU scheduling

일반적인 컴퓨터 프로그램은 CPU burst와 I/O burst를 반복하며 실행된다. 이때 I/O burst가 발생하면 때 CPU는 사용되지 않고 idle 상태에 놓이게 된다. 운영체제의 가장 핵심적인 역할 중 하나는 이러한 idle 시간을 최소화하여 CPU 자원을 최대한 활용하는 것이며 이를 위한 기술이 바로 CPU 스케줄링이다.

CPU 스케줄링은 다중 프로그래밍 환경에서 운영체제가 CPU를 다양한 프로세스에 적절히 할당함으로써 시스템의 성능을 개선하는 메커니즘이다. CPU 스케줄링에 대해 알아보기 위해서는 우선적으로 CPU 스케줄러, Dispatcher(문맥 교환), 스케줄링 판단 기준의 세 가지 요소에 대해 함께 살펴볼 필요가 있다.

1. CPU 스케줄러

CPU 스케줄러는 운영체제의 핵심 구성요소로, CPU 스케줄링을 관리하기 위한 일종의 모듈이다. 기본적으로 CPU 스케줄러는 메모리 내에서 실행 준비가 된 프로세스들 중 어떤 프로세스에게 CPU를 할당할지 결정하고, 현재 실행 중인 프로세스로부터 CPU를 회수하여 다른 프로세스에게 전환할지를 제어한다.

CPU 스케줄러가 작동하는 4가지 경우는 다음과 같다. Running 상태에서 Waiting 상태가 될 때, Running 상태에서 Ready 상태가 될 때, Waiting 또는 new 상태의 프로세스가 Ready가 될 때, 프로세스가 끝났을 때이다. 스케줄링 정책은 하나의 프로세스가 CPU를 사용하고 있을 때 다른 프로세스가 이를 빼앗을 수 있는 선점형과 그렇지 못한 비선점형으로 나뉜다. 기본적으로 스케줄링 정책은 비선점형 작업이지만 모든 스케줄러는 선점형이 될 수 있다.

2. Dispatcher

문맥 교환은 하나의 프로세스가 CPU를 사용 중인 상태에서 다른 프로세스가 CPU를 사용할 수 있도록 하기 위해 현재 프로세스의 상태를 저장하고 새로운 프로세스의 상태를 불러오는 과정을 의미한다. 문맥 교환 시 오버헤드가 발생할 수 있으며 이는 스케줄링 정책에 악영향을 끼친다. 그러나 이번 보고서의 모든 스케줄러 분석에서는 문맥 교환으로 인한 오버헤드를 고려하지 않을 것이다.

3. 판단 기준

어떠한 스케줄링 정책이 얼마나 효율적인지를 판단할 때는 기준이 필요하다. 스케

줄링 정책을 판단하는 판단 기준은 Throughput, Turnaround time, Waiting time, Response time 4가지가 존재한다.

Throughput은 단위 시간 당 처리되는 프로세스의 개수를 의미한다. Throughput을 최대로 높이기 위해서는 처리 시간이 짧은 프로세스를 먼저 처리해야 한다. Turnaround time 프로세스를 기준으로 생각해, 새롭게 만들어진 프로세스가 완전히 완료될 때까지의 시간을 의미한다. Waiting time은 어떠한 프로세스가 레디큐에서 얼마나 많은 시간을 기다렸는지를 의미한다. Waiting time을 줄이기 위해서는 실행시간이 짧은 작업을 우선적으로 처리해야 한다. Response time은 레디큐에서부터 첫 번째 응답까지 걸리는 시간을 의미한다.

좋은 스케줄링 정책은 CPU의 활용도와 Throughput을 최대로 하며 Turnaround time, Waiting time, Response time을 줄여야 한다. 하지만 Response time을 줄이면 문맥 교환이 증가하고 이는 Throughput의 감소로 이어지는 등 모든 판단 기준을 맞추는 것은 사실상 불가능하다. 따라서 자신이 사용하려는 환경이 어떤 판단 기준을 요구하는지를 생각하고 이에 맞는 스케줄링을 사용하는 것이 옳다.

본 보고서는 위의 3가지 요소 중 CPU 스케줄러에 초점을 맞춰 분석하고자 한다. CPU 스케줄링은 비선점형과 선점형, 단일 CPU 환경과 다중 CPU 환경, 하나의 레디큐를 사용하는 경우와 다중 큐를 사용하는 경우 등 다양한 상황에 따라 방식이 달라지며, 이들 상황에 맞는 적절한 스케줄링 알고리즘 선택이 매우 중요하다. 이제 본문에서는 주요 CPU 스케줄링 알고리즘들을 하나씩 살펴보며, 각각의 동작 사례와 특성, 장단점에 대해 분석해보도록 하겠다.

Ⅱ. FCFS

FCFS는 First-Come, First-Served 스케줄링으로 레디큐에 도착한 순서대로 서비스를 제공한다. FIFO 스케줄링 방식을 사용해 간단하고 공정하다는 특징이 있다. 오직 비선점 방식으로 다른 프로세스들에 자리를 내어주지 않는다. 단순한 형태의 스케줄링 알고리즘으로 혼자만 사용되는 경우는 거의 존재하지 않는다.

1. 스케줄링 알고리즘

```
class Process
    int id
    int burstTime

class FCFS
    Queue<Process> readyQueue
    Process currentProcess
    boolean systemRunning
```

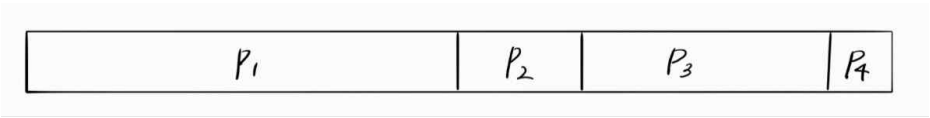
```
int systemTime

run() 함수
    while (systemRunning)
        while (레디큐에 넣을 수 있는 새 프로세스 존재)
        readyQueue.add(process)

        if (CPU가 idle이고 레디큐에 프로세스 존재)
            currentProcess = readyQueue.poll()

        if (currentProcess != null)
            currentProcess.burstTime -= 1
            if (currentProcess.burstTime == 0)
                currentProcess = null

        systemTime++
```

2. 동작 사례

예를 들어 아래 차트와 같은 Burst time을 가진 프로세스 4개의 프로세스가 0초의 순간 순서대로 도착했다고 가정 해보자.

Process	priority number
P1	8
P2	3
P3	6
P4	1

FCFS 알고리즘은 도착 순서 외에 다른 요소를 보지 않으므로 P1, P2, P3, P4 순으로 프로세스가 작동되며 선점은 없다. 이때 아래와 같은 모양의 간트 차트를 그릴 수 있다.

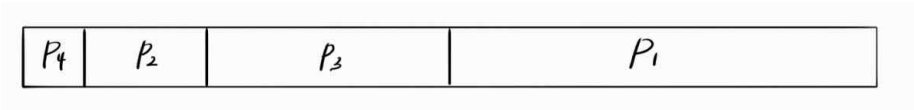


따라서 각 프로세스의 대기 시간은 0, 8, 11, 17초가 되고 (0+8+11+17)/4로 계산해 평균 대기 시간은 9초가 된다.

3. 특징

FCFS의 가장 큰 특징은 convoy effect가 발생한다는 것이다. convoy effect는 호위 효과로 긴 프로세스 뒤의 짧은 프로세스가 받게 된다. 예를 들어 아까와 똑같은 프로세스들이 P4, P2, P3, P1의 순서대로 도착했다고 가정한다면 간트 차트의 모양

은 아래와 같이 달라진다.



이때의 평균 대기 시간은 (0+1+3+6)/4로 계산되어 2.5초가 된다. 프로세스의 burst time이 같음에도 처음의 동작 사례에 비해 7.5초가 단축되었음을 알 수 있다. 따라서 FCFS의 경우 각 프로세스가 도착하는 순서에 따라 평균 대기 시간이 길어질 수 있다는 단점이 존재한다.

4. 장단점

FCFS 스케줄러의 장점은 구현이 매우 간단하고 단순하다는 것이다. FIFO 방식의 비전섬형 알고리즘이므로 공정성이 매우 높다. 우선순위가 낮은 프로세스가 서비스를 받지 못하는 Starvation 현상 역시 발생하지 않는다. 프로세스의 문맥 교환이 일어나지 않으므로 오버헤드가 적다는 장점도 있다.

반면 convoy effect의 발생으로 인해 프로세스의 평균 대기 시간이 길어질 수 있다는 단점이 존재한다. CPU bound 프로세스와 I/O bound 프로세스의 균형을 고려하지 않는다. 비선점형 방식으로 한 번 CPU를 할당받으면 완료될 때까지 CPU를 놓지 않는다는 단점 역시 존재한다.

Ⅲ. SJF

SJF는 Shortest Job Fist 스케줄링으로 프로세스가 처리되는데 걸리는 시간(next CPU burst time)이 짧은 순서대로 서비스를 제공한다. 비선점형, 선점형으로 2가지의 방식이 존재한다.

비선점형의 경우 일단 CPU에 프로세스가 주어지면 해당 프로세스를 끝낼 때까지 선점이 일어나지 않는다. 선점형의 경우 새로 레디큐에 들어온 프로세스의 burst time이 현재 CPU에서 돌아가는 프로세스의 남은 시간보다 짧으면 선점된다. SRTF(Shortest remaining time first)라 불린다.

1. 스케줄링 알고리즘

① SJF

```
class Process
        int id
        int burstTime

class SJF
        Queue<Process> readyQueue
         Process currentProcess
```

```
        boolean systemRunning
        int systemTime

        run() 함수
            while (systemRunning)
                while (레디큐에 넣을 수 있는 새 프로세스 존재)
                    readyQueue.add(process)

                if (CPU가 idle이고 레디큐에 프로세스 존재)
                    currentProcess = 선택 (레디큐를 돌며 burstTime이
                                            가장 짧은 프로세스를 선택)
                    readyQueue에서 해당 process 제거

                if (currentProcess != null)
                    currentProcess.burstTime -= 1
                    if (currentProcess.burstTime == 0)
                        currentProcess = null

                systemTime++
```

② SRTF

```
class Process
        int id
        int burstTime
        int remainingTime

class SRTF
        Queue<Process> readyQueue
         Process currentProcess
         boolean systemRunning
         int systemTime

        run() 함수
            while (systemRunning)
                while (레디큐에 넣을 수 있는 새 프로세스 존재)
                    readyQueue.add(process)
```

```
if (CPU가 idle이고 레디큐에 프로세스 존재)
    currentProcess = 선택 (레디큐를 돌며 remainingTime이
                           가장 짧은 프로세스를 선택)

if (레디큐에 currentProcess보다 더 짧은 프로세스 존재)
    currentProcess = 선택 (레디큐를 돌며 remainingTime이
                           가장 짧은 프로세스를 선택)

if (currentProcess != null)
    currentProcess.remainingTime -= 1
    if (currentProcess.remainingTime == 0)
        readyQueue에서 해당 process 제거
        currentProcess = null

systemTime++
```

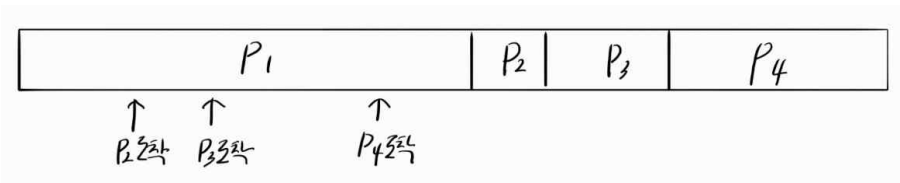
2. 동작 사례

아래 차트와 같은 도착 시간과 Burst time을 가진 프로세스 4개가 존재한다고 가정하자.

Process	Arrival time	Burst time
P1	0.0	7
P2	2.0	2
P3	3.0	3
P4	5.0	4

① SJF

가장 먼저 0초의 순간 도착한 프로세스는 P1밖에 없으므로 P1 프로세스가 실행된다. 이후 P1 프로세스가 종료되는 7초의 순간 P2, P3, P4는 모두 도착돼 있다. 간트 차트는 아래와 같이 그려진다.

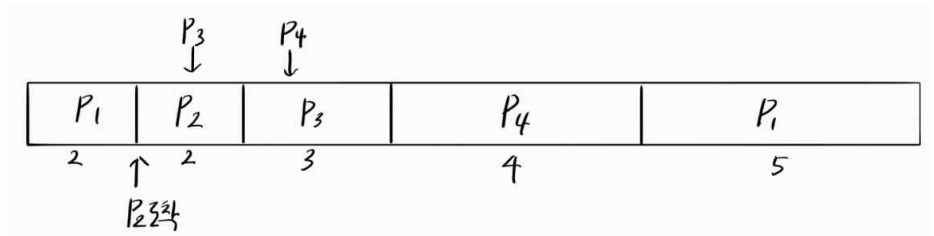


각 프로세스의 대기 시간은 0초, 7초, 9초, 12초이며 평균 대기 시간은 (0+7+9+12)/4 = 7초이다.

② SRTF

가장 먼저 0초의 순간 도착한 프로세스는 P1밖에 없으므로 P1 프로세스가 실행된

다. 이후 2초의 순간 P2 프로세스가 도착하며 P1의 남은 시간인 5초보다 짧으므로 선점이 일어난다. P2 프로세스가 실행 중인 3초의 순간 P3가 도착하지만 P2의 남은 시간이 더 짧으므로 선점되지 않는다. 이러한 방식으로 모든 프로세스가 끝날 때까지 스케줄링을 실행하면 아래와 같은 간트 차트가 그려진다.



각 프로세스의 대기 시간은 9, 0, 1, 2초이며 평균 대기 시간은 (9+0+1+2)/4 = 3초이다. SJF 스케줄링 알고리즘에 비해 4초가 단축된 것을 확인할 수 있다.

3. 특징

SJF 스케줄링 알고리즘은 주어진 프로세스 집합에 대해 이론적으로 최소 평균 대기 시간을 제공한다. 해당 알고리즘을 사용하기 위해선 다음 프로세스의 CPU burst를 알아야 한다. 그러나 실제 시스템을 사용하는 상황에서 다음 프로세스의 정확한 CPU burst를 예측하는 것은 불가능에 가깝다. 즉, 실제 구현이 거의 불가능하다는 것이다.

때문에 해당 알고리즘을 사용하기 위해서는 지수 평균(exponential averaging)과 같이 이전 CPU burst의 값을 통해 Next CPU burst를 예측하는 과정이 반드시 필요하다. 또 SJF는 실제 시스템 구현보다는 이론적인 기준점으로 더 자주 활용되며 다른 스케줄링 알고리즘들의 성능을 평가하는 데 참조 모델로써 활용된다.

4. 장단점

SJF 스케줄링의 가장 큰 장점은 평균 대기 시간이 모든 알고리즘 중 가장 짧다는 것이다. 짧은 작업을 선호하므로 프로세스 처리량이 높다. 시스템의 효율성을 크게 높일 수 있다. 다만 긴 프로세스의 실행 기회가 계속 뒤로 밀리는 현상이 발생할 수 있으며 각 프로세스의 실행시간을 정확히 예측하는 것이 어렵다. 비선점형 방식으로 구현할 경우, 선점형 방식에 비해 응답 시간이 길어질 수 있다.

SJF의 선점형 버전인 SRTF는 SJF에 비교해서도 평균 대기 시간이 더 짧다. 또한 새로운 프로세스가 도착하면 즉시 이를 반영할 수 있으며 이는 응답 시간을 개선하는 효과를 낸다. 다만 실행시간이 긴 프로세스의 실행이 밀리는 현상이 SJF에 비해 더 심해질 수 있으며 문맥 교환이 빈번히 발생하며 오버헤드가 증가할 수 있다. 또한 매 CPU 사이클마다 프로세스의 남은 실행시간을 지속적으로 추적해야 한다.

Ⅳ. Priority Scheduling

모든 스케줄링 알고리즘은 우선순위 알고리즘이다. 예를 들어 SJF 스케줄링의 경

우, next CPU burst time이 가장 짧은 것을 우선순위로 두고 알고리즘을 실행한다. 이 파트에서 설명하게 될 우선순위 알고리즘은 시스템이 정한 정수값의 priority number를 기준으로 두고 알고리즘을 수행한다. 위의 다른 알고리즘들과 마찬가지로 선점형과 비선점형으로 나뉜다.

1. 스케줄링 알고리즘

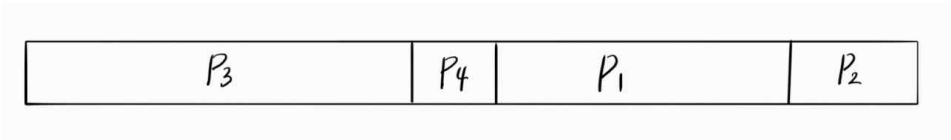
```
class Process
    int id
    int burstTime
    int priority

class PriorityScheduler
    Queue<Process> readyQueue
    Process currentProcess
    boolean systemRunning
    int systemTime

    run() 함수
        while (systemRunning)
            while (레디큐에 넣을 수 있는 새 프로세스 존재)
                readyQueue.add(process)

            if (CPU가 idle이고 레디큐에 프로세스 존재)
                currentProcess = 선택 (레디큐를 돌며 priority가
                                        가장 큰 프로세스를 선택)
                readyQueue에서 currentProcess 제거

            if (currentProcess != null)
                currentProcess.burstTime -= 1
                if (currentProcess.burstTime == 0)
                    currentProcess = null

            systemTime++
```

2. 동작 사례

아래 차트와 같은 priority number와 Burst time을 가진 프로세스 4개가 존재한다고 가정하자.

Process	priority number	Burst time
P1	5	7
P2	3	2
P3	10	8
P4	6	1

우선순위 스케줄링은 priority number가 큰 순으로 실행되므로 4개의 프로세스가 동시에 도착한 경우 P3, P4, P1, P2 순으로 프로세스가 작동된다. 이때의 간트 차트는 아래와 같이 그려진다.



이때 각 P1, P2, P3, P4 프로세스의 각 대기 시간은 9, 16, 0, 2초가 되며 평균 대기 시간은 (9+16+0+2)/4 = 6.75초가 된다.

3. 특징

우선순위 스케줄링의 가장 큰 문제점은 Starvaion이 발생할 수 있다는 것이다. 높은 우선순위의 프로세스들을 실행하다 보면 낮은 우선순위의 어떤 프로세스는 영원히 서비스를 받지 못할 가능성이 있다. 효율성과 공평성이 가장 중시되는 운영체제에서 이러한 문제를 해결하는 것은 필수적이다.

Starvaion 현상을 해결하기 위한 가장 쉬운 방법은 Aging 정책을 도입하는 것이다. 시스템의 우선순위는 그대로 두되, 우선순위 스케줄링을 위한 우선순위를 도입하는 것이다. Aging 정책을 사용하면 대기 시간이 길어짐에 따라 프로세스의 우선순위가 상승한다. 아무리 낮은 우선순위를 가진 프로세스더라도 시간이 지나기만 하면 반드시 서비스를 받을 수 있을 것이다.

4. 장단점

우선순위 스케줄링은 시스템이 정한 우선순위를 기준으로 작동함으로써 중요한 프로세스에 우선적으로 처리될 기회를 제공한다. 또한 Aging 등의 추가 정책을 통해 시스템의 요구사항에 따라 유연하게 우선순위를 조정할 수 있다. 선점형과 비선점형 모두로 구현이 가능하다.

RR의 단점은 낮은 우선순위를 가진 프로세스에서 Starvaion 현상이 발생할 수 있다는 것이다. 이를 위해 Aging과 같은 추가적인 정책이 필요하다. 또한 우선순위 결정 매커니즘이 추가적인 복잡성을 가져온다. 우선순위가 고정되면 시스템 상황 변화에 적응이 어려우며, 우선순위를 조정하는 경우에 따라서는 우선순위 역전(Priority Inversion) 문제가 발생할 수 있다.

Ⅴ. RR

Round Robin은 FIFO를 베이스로 한 스케줄링 알고리즘이다. RR에서 프로세는 오직 한정된 시간 동안만 실행된다. 이때 제한되는 단위 시간을 time slice 또는 time quantum이라고 부른다. FCFS와 달리 FIFO를 사용함에도 선점형으로 작동된다. RR은 모든 프로세스에 공정한 CPU 시간을 제공해 Response time을 낮추는 것을 목표로 한다. 때문에 다른 복잡한 알고리즘의 일부로써 사용되는 경우가 많다.

1. 스케줄링 알고리즘

```
class Process
    int id
    int burstTime

class RR
    Queue<Process> readyQueue
    Process currentProcess
    boolean systemRunning
    int systemTime
    int timeQuantum
    int RunningTime

    run() 함수
        while (systemRunning)
            while (레디큐에 넣을 수 있는 새 프로세스 존재)
                readyQueue.add(process)

            if (CPU가 idle이고 레디큐에 프로세스 존재)
                currentProcess = readyQueue.poll()
                RunningTime = 0

            if (currentProcess != null)
                currentProcess.remainingTime -= 1
                RunningTime += 1

                if (currentProcess.remainingTime == 0)
                    currentProcess = null
                else if (timeSlice == timeQuantum)
                    readyQueue.add(currentProcess)
```

currentProcess = null

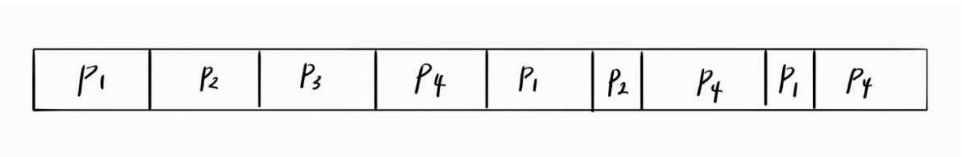
systemTime++

2. 동작 사례

RR의 time quantum=2, 아래 차트와 같은 Burst time을 가진 4개의 프로세스가 0초의 순간 순서대로 들어왔다고 가정하자.

Process	Burst time
P1	5
P2	3
P3	2
P4	6

프로세스는 레디큐에 들어온 순서대로 실행되며 time quantum을 넘기면 다음 프로세스에게 선점된다. 따라서 아래와 같은 간트 차트를 그릴 수 있다.



이때 각 프로세스의 대기 시간은 9, 8, 4, 10초이며 평균 대기 시간은 (9+8+4+10)/4 = 7.75초이다. 이 간트 차트를 살펴보면 CPU Burst time에 비해 평균 대기 시간이 상대적으로 길다는 것을 알 수 있다. 대신 어떤 프로세스든 최소 6초에 한 번은 서비스를 받을 수 있다. 이를 통해 RR은 다른 스케줄링 알고리즘에 비해 평균 Waiting time이 높은 대신 평균 Response time은 낮다는 것을 확인할 수 있다.

3. 특징

RR 알고리즘에서 가장 중요한 것은 time quantum의 적절한 사이즈를 지정하는 것이다. 예를 들어 time quantum의 사이즈가 지나치게 커질 경우, 프로세스 실행 내내 단 한 번의 문맥 교환도 일어나지 않아 RR이 FIFO와 똑같은 방식으로 운영되게 된다. 반대로 time quantum의 사이즈가 지나치게 작아질 경우, 문맥 교환 오버헤드가 이 실제 프로세스의 실행시간보다 짧아지게 되어 CPU의 효율이 크게 낮아질 수 있다. 따라서 time quantum의 사이즈는 최소 프로세스들의 문맥 교환 시간보다는 길어야 한다는 것을 알 수 있다.

또한 위에서 설명했듯 RR은 많은 다른 스케줄링 알고리즘의 일부로써 사용되는 경향이 있다. 예를 들어 MS-Windows 운영체제의 경우 프로세스들을 관리하기 위해 RR과 우선순위 스케줄링을 함께 사용한다. Linux 운영제체는 CFS(Completely Fair Scheduler)를 기본 스케줄러로 사용하지만, 실시간 프로세스를 위해 SCHED_RR이라는 Round Robin 정책을 제공한다. 또한 Unix 및 Unix 계열 시스템들은 Multi-level Feedback Queue와 Round Robin을 결합하여 사용한다.

4. 장단점

RR은 모든 프로세스에 공정하게 CPU 시간을 할당한다. SJF와 같은 스케줄링 알고리즘들과 달리 실행 시간을 미리 알 필요가 없으며 선점형 방식으로 프로세스가 CPU를 독점할 수 없다. Response time은 낮아 대화형 시스템에 적합하다.

다만 time quantum 설정이 성능에 큰 영향을 미치게 된다는 단점도 존재한다. time quantum이 너무 커지면 RR은 FCFS와 유사해진다. 반대로 time quantum이 너무 작아지면 문맥 교환 오버헤드가 증가해 성능을 떨어트린다. 또한 실행 시간이 다양한 프로세스들 간에 평균 대기 시간이 길어질 수 있으며, 개별로 사용될 시 프로세스들의 우선순위를 고려하지 않는다

VI. Multilevel Feedback Queue

프로세스 관리를 위한 레디큐는 여러 개로 나뉠 수 있다. 분리된 큐는 크게 foreground와 background로 나뉘는데 foreground는 주로 상호작용이 필요하며 빠르게 처리되어야 하는 프로세스가 담기고 background는 상대적으로 foreground에 비해 느리게 처리되어도 성능에 큰 지장을 주지 않는 프로세스들이 담기게 된다.

각 큐는 다른 큐들과 상관없이 자신의 독자적인 스케줄링 알고리즘을 가진다. 또한 큐들 간의 스케줄링 정책 역시 필요하다. 큐들 간의 스케줄링은 fixed priority scheduling과 time slice로 나뉜다. fixed priority scheduling은 우선순위가 높은 상위 큐부터 저리되는 방식이며 starvation이 발생할 가능성이 존재한다. time slice는 각 큐에 특정 CPU time을 할당하여 모든 큐가 일정 시간 동안 프로세스를 실행할 수 있도록 보장한다.

일반적인 Multilevel Queue 스케줄링은 각 프로세스가 처음 시스템에 진입할 때 정해진 큐에 영구적으로 할당된다. 반면 Multilevel Feedback Queue의 경우, 프로세스들이 정해진 규칙에 따라 여러 큐 사이를 이동할 수 있다. 이를 위해 프로세스가 최초에 들어가게 될 큐를 결정하는 메소드 외에 프로세스를 상위 큐로 옮기는 upgrade 메소드와 프로세스를 하위 큐로 되돌리는 demote 메소드가 필요하다. 이러한 피드백 메커니즘을 통해 시스템은 프로세스의 특성과 행동에 따라 동적으로 프로세스의 우선순위를 조정할 수 있게 된다.

1. 스케줄링 알고리즘

```
class Process
    int id
    int burstTime
    int queueLevel
    int runningTime
```

```
class MLFQ
    Queue<Process> queue0  // RR
    Queue<Process> queue1  // RR
    Queue<Process> queue2  // FCFS
    Process currentProcess
    boolean systemRunning
    int systemTime
    int timeQuantum0
    int timeQuantum1

    run() 함수
        while (systemRunning)
            while (레디큐에 넣을 수 있는 새 프로세스 존재)
                process.queueLevel = 0
                queue0.add(process)

            if (CPU가 idle이고)
                if (0번 큐에 프로세스 존재)
                    currentProcess = queue0.poll()
                    currentProcess.runningTime = 0
                else if (1번 큐에 프로세스 존재)
                    currentProcess = queue1.poll()
                    currentProcess.runningTime = 0
                else if (2번 큐에 프로세스 존재)
                    currentProcess = queue2.poll()
                    currentProcess.runningTime = 0

            if (currentProcess != null)
                currentProcess.burstTime -= 1
                currentProcess.runningTime += 1

                if (currentProcess.burstTime == 0)
                    currentProcess = null
                else if (현재 프로세스의 queueLevel==0이고
                                    runningTime == timeQuantum0라면)
                    currentProcess.queueLevel = 1
                    queue1.add(currentProcess)
```

```
            currentProcess = null
        else if (현재 프로세스의 queueLevel==1이고
                                runningTime == timeQuantum1라면)
            currentProcess.queueLevel = 2
            queue2.add(currentProcess)
            currentProcess = null

    systemTime++
```

2. 동작 사례 및 특징

위의 코드는 가장 간단한 형태의 Multilevel Feedback Queue를 구현한 것이다. Q0, Q1, Q2라는 세 개의 큐가 있다고 가정했을 때 Q0는 4의 time quantum을 가진 RR 큐, Q1은 8의 time quantum을 가진 RR 큐, Q2는 FCFS 큐이다. 새로 도착한 프로세스는 가장 먼저 Q0에 배치된다. 그곳에서 CPU burst 4초를 초과하였음에도 작업을 완료하지 못했다면 Q1로 이동한다. 만약 Q1에서 CPU burst 4초를 초과하였음에도 작업을 완료하지 못했다면 FCFS 큐인 Q2로 이동하게 된다.

해당 동작 사례와 그에 맞는 슈도 코드는 스케줄링 알고리즘의 흐름을 파악하기 위해 가장 간단한 형태로 작성되었다. 실제 프로그램 구현 시에는 새로 도착한 프로세스가 반드시 Q0로 들어갈 필요가 없으며 마지막 큐인 Q2에서 Aging 등의 추가 정책을 통해 Starvation 현상을 막는 것도 필요하다.

Multilevel Feedback Queue는 시간의 흐름에 따라 프로세스를 점차 낮은 우선순위로 이동시키는 구조를 가지며, 이로 인해 짧은 작업은 빠르게 처리되고 긴 작업은 점진적으로 FCFS 큐로 내려가게 된다. 또한 프로세스가 큐를 이동하는 정책과는 별개로 큐들 간의 우선순위를 고려한 스케줄링 정책(fixed priority scheduling, time slice 등) 역시 구현되어야 한다.

3. 장단점

Multilevel Feedback Queue의 장점은 반응성이 우수하다는 점이다. 짧고 자주 CPU를 요청하는 인터랙티브 작업을 빠르게 처리할 수 있어, 사용자 반응성이 중요한 GUI 기반 시스템에서 자주 활용된다. 또한, 프로세스의 특성에 따라 자동으로 적절한 큐로 분류되므로 처리 유연성이 높으며 CPU-bound 작업과 I/O-bound 작업을 효과적으로 분리함으로써 CPU 이용률을 향상시킬 수 있다. 이러한 특성 덕분에, 짧은 작업과 긴 작업이 혼재된 혼합형 환경에서 특히 적합하다.

반면 큐 이동 로직, 우선순위 관리, aging 정책 등을 구현해야 하므로 다른 스케줄링 알고리즘에 비해 구현이 복잡하다. 또한 aging 정책이 적용되지 않을 경우, 낮은 우선순위 큐에 있는 프로세스가 장시간 대기하게 되는 starvation 현상이 발생할 수 있다. time quantum, 큐의 개수, 이동 조건 등의 파라미터 설정이 적절하지 않으면

오히려 성능이 저하될 수 있으며 큐가 동적으로 변하기 때문에 전체적인 성능을 예측하기 어려운 단점도 존재한다. 이러한 이유로 실시간 시스템에서는 이러한 예측 불가능성이 더 큰 문제가 될 수 있다.

Ⅶ. 기타 스케줄링 알고리즘

위에서 설명한 FCFS, SJF, SRTF, Priority Scheduling, RR, Multilevel Feedback Queue 외에도 운영체제에는 다음과 같은 다양한 스케줄링 알고리즘이 존재한다.

Fair Share Scheduling (FSS)는 사용자나 그룹 단위로 CPU 자원을 공정하게 분배하여 시스템 자원이 특정 사용자에게 과도하게 집중되지 않도록 한다. 또한 상대적으로 우선순위가 높은 그룹에 더 많은 자원을 할당할 수 있으며 할당받은 CPU 자원은 그룹 내에서 각자의 스케줄링 정책에 따라 관리된다.

Thread Scheduling은 프로세스 내부의 다중 쓰레드에 대해 CPU를 할당하며 쓰레드 간 우선순위나 동기화를 고려한 스케줄링을 수행한다. 쓰레드는 일반 프로세스보다 가볍고 생성 비용이 적은 Light-Weight Process로, 웹 서버처럼 다수의 클라이언트 요청을 처리해야 하는 환경에서 효과적으로 사용된다.

Multiprocessor Scheduling은 두 개 이상의 CPU를 사용하는 시스템에서 작업을 효율적으로 분산시켜 CPU 부하를 균등하게 조절하는 스케줄링 방식이다. 이 과정에서 프로세스를 가능한 한 동일한 CPU에 유지하려는 Processor Affinity 문제와 CPU 간 작업 분산의 효율을 높이기 위한 Load Balancing 문제가 함께 고려된다.

Real-Time Scheduling은 주어진 시간 제약 내에 작업이 반드시 완료되도록 보장하는 것을 목표로 한다. Hard real-time과 Soft real-time으로 나뉜다. Hard real-time은 시간 내 작업 완료가 절대적으로 요구되며, Soft real-time은 일정 수준의 기한 준수를 목표로 하되 일정한 유연성을 허용한다.

Deadline-based Scheduling은 각 프로세스에 설정된 Deadline을 기반으로 작업의 우선순위를 정하여 주어진 기한 내에 작업을 완료하는 것을 중점으로 둔다.

이처럼 운영체제에서는 다양한 요구사항과 시스템 환경에 맞춰 여러 스케줄링 알고리즘이 존재하며 각각의 알고리즘은 상황에 따라 장단점이 뚜렷하게 나타난다. 다음으로는 이러한 스케줄링 알고리즘 분석을 바탕으로 실제 상황에 맞춰 설계한 나만의 스케줄러를 제안하고 그 설계 철학과 기대 효과에 대해 설명하고자 한다.

Ⅷ. 나만의 스케줄러

여러 스케줄링 알고리즘을 분석하는 과정에서 '사용자의 만족도를 최우선으로 고려하는 스케줄링 방식'을 구상해보고 싶다는 생각이 들었다. 고민을 거듭한 끝에 사용자 만족도를 높이는 가장 효과적인 방법은 Response Time을 줄이는 것이라는 결론에 도달하였다.

응답 시간을 줄이기 위해서는 프로세스가 신속하게 CPU를 할당받아 초기 작업을 시작할 수 있어야 하며, 동시에 CPU burst가 짧은 작업을 우선적으로 처리하여 전체 시스템의 Throughput을 향상시킬 필요가 있다.

이를 위해 제안한 스케줄러는 RR 방식으로 공정하게 프로세스를 처리하다가, 일정 시간이 지난 후에는 남은 작업 시간이 짧은 순서로 프로세스를 처리하는 RR-SJF 스케줄링을 설계하였다.

이러한 전환을 통해, 초반에는 모든 프로세스에게 빠르게 CPU를 할당하여 응답 속도를 보장하고, 후반에는 짧은 작업부터 먼저 처리함으로써 전체적인 처리 속도와 자원 효율성을 함께 향상시킬 수 있다. 자세한 작동 방식은 아래와 같다.

1. 스케줄링 초기에는 모든 프로세스가 동일한 time quantum을 가지며 RR 방식으로 공정하게 처리된다. 이 단계에서는 I/O-bound 작업이나 CPU burst time이 짧은 작업들이 대부분 빠르게 종료된다.
2. 이후 시스템 시간이 일정 기준을 초과하거나, 레디큐에 누적된 프로세스 개수가 임계치를 초과할 경우, 레디큐의 프로세스들은 남은 CPU burst time을 기준으로 정렬되어 SJF 방식으로 처리된다. 이때의 SJF는 비선점으로 동작한다.
3. SJF 구간에 머무를 수 있는 제한 시간이 끝나거나, 특정 프로세스의 처리가 과도하게 늦어지거나, 레디큐에 누적된 프로세스의 수가 일정 수 미만으로 감소할 경우, 다시 RR로 복귀한다.

RR-SJF 스케줄링 알고리즘의 가장 큰 장점은 응답 시간을 줄여 사용자의 체감 성능과 서비스 반응성을 향상시킬 수 있다는 점이다. 또한 RR과 SJF를 조합함으로써 기존 RR 방식보다 Throughput이 향상된다. 레디큐에 과도한 프로세스가 누적될 경우, SJF 방식으로 전환되어 처리 효율과 큐 길이를 동시에 개선할 수 있다. 반대로 SJF에서 특정 프로세스의 지연이 발생할 경우, RR로 복귀하여 Starvation을 완화하고 전체적인 처리 속도를 높일 수 있다.

RR-SJF 스케줄링 알고리즘의 단점은 전환 및 복귀 조건이 복잡하여 스케줄러 구현 난이도가 높다는 점이다. 또한 이 알고리즘은 CPU의 실제 효율보다는 사용자 만족도를 우선시하므로 복잡한 정책에 따른 성능 저하 가능성을 감안해야 한다. 더불어, 잦은 정책 전환으로 인한 문맥 교환 비용 증가도 고려해야 할 요소이다.

IX. 결론

이번 프로젝트를 통해 다양한 CPU 스케줄링 알고리즘을 분석하고, 직접 알고리즘을 설계해보며 각 방식의 특징과 활용 방식에 대해 깊이 이해할 수 있었다.

모든 스케줄링은 각자의 스케줄링 알고리즘마다 고유한 장점과 단점이 있으며, 특정 알고리즘이 모든 상황에 이상적으로 적용되는 것은 아니다. 중요한 것은 운영체제가 자주 마주하는 상황과 시스템의 요구사항을 정확히 파악한 뒤, 그에 적합한 스케

줄러를 선택하거나 조합하는 것이다.

실제로 많은 현대 운영체제는 하나의 알고리즘만을 고정적으로 사용하지 않고 여러 알고리즘을 혼합하거나 변형하여 보다 효율적인 스케줄링 정책을 실현하고 있다. 이번 보고서를 통해 상황에 맞는 유연한 스케줄링 전략의 필요성과 사용자 만족도와 시스템 성능 간의 균형을 고려한 설계의 중요성을 다시 한 번 느낄 수 있었다.