운영체제: 강의노트 06

A. Silberschatz, P.B. Galvin, G. Gagne
Operating System Concepts,
Sixth Edition, John Wiley & Sons, 2003.

Part II. Process Management

6 CPU 스케줄링

6.1 기본 개념

- 다중 프로그래밍의 목적은 항상 실행할 수 있는 프로세스가 있도록 하여 CPU 사용 효율을 극대화하는데 있다.

6.1.1 CPU-I/O 버스트 주기

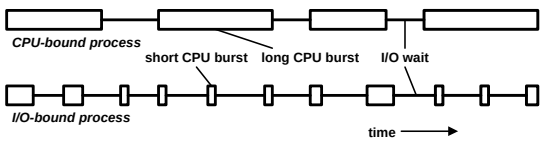
- 프로세스는 실행되는 동안 CPU 실행과 입출력 대기 두 주기를 반복한다.
- 계산 중심 프로세스는 적은 수의 매우 긴 CPU 버스트를 가지며, 입출력 중심 프로세스는 많은 수의 짧은 CPU 버스트를 가진다. 그림 6.1 참조.
- 프로세스의 CPU 버스트는 프로세스의 특성과 시스템마다 다양하지만 보통 그림 6.2와 같은 빈도수를 나타낸다. 물론 시스템마다 차이는 있을 수 있지만 대체로 유사한 빈도수를 나타낸다. 즉, 짧은 CPU 버스트의 빈도가 많고, 8 ms 이상의 CPU 버스트는 거의 없다.
- CPU 성능이 발달할 수록 프로세스는 입출력 중심 프로세스가 되는 경향이 높아진다.

6.1.2 CPU 스케줄러

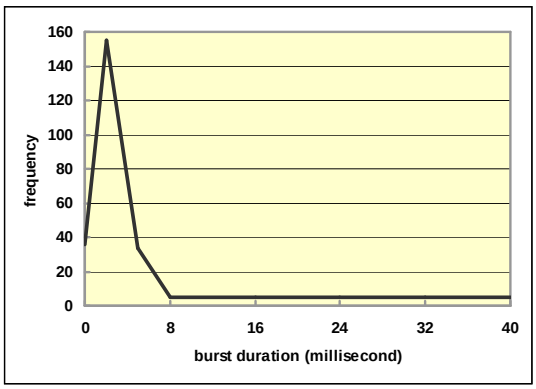
- CPU가 유휴 상태가 되면 준비완료 큐에 있는 프로세스를 하나 선택해서 실행한다. 이 선택은 CPU 스케줄러(또는 단기 스케줄러)가 한다.

6.1.3 선점 스케줄링

- CPU 스케줄링 결정은 네 가지 상황에서 일어난다.



<그림 6.1> 계산 중심과 입출력 중심 프로세스



<그림 6.2> CPU 버스트 시간의 도표

  - 상황 1. 프로세스가 실행중 상태에서 대기중 상태로 전환될 때(예: 입출력 요청)
  - 상황 2. 프로세스가 실행중 상태에서 준비완료 상태로 전환될 때(예: 인트럽트 발생)
  - 상황 3. 프로세스가 대기중 상태에서 준비완료 상태로 전환될 때(예: 입출력 완료)
  - 상황 4. 프로세스가 종료될 때
- **비선점(nonpreemptive) 스케줄링 방식**: 이 방식에서는 프로세스가 CPU에 할당되면 그 프로세스가 종료되거나 대기중 상태로 전환될 때까지 CPU를 점유한다. 상황 1과 상황 4에서만 스케줄링이 일어나면 비선점 방식이다.
- **선점(preemptive) 방식**: 어떤 조건이 만족되면 현재 실행중인 프로세스의 의사와 상관없이 그것의 실행을 중단하고 다른 프로세스를 CPU에 할당한다. 실행중인 프로세스를 중단하면 여러 가지 문제가 발생할 수 있으므로 선점 방식은 이런 것을 대처할 수 있어야 한다.

6.1.4 Dispatcher

- 스케줄러가 선택한 프로세스를 CPU에 할당해주는 요소를 **디스패처(dispatcher)**라 한다.
- 디스패처의 임무
  - 문맥 전환
  - 모드 전환
- 디스패처가 하나의 프로세스를 중단하고 다른 프로세스를 실행하기까지 소요되는 시간을 디스패치 지연(dispatch lantency)이라 한다.

6.2 스케줄링 기준

- CPU 스케줄링 알고리즘을 선택할 때 고려되는 기준
  - **CPU 사용 효율(CPU utilization)**: 가능한 CPU가 계속 유용한 작업을 하도록 해야 한다. 만약 어떤 정해진 시간 동안 계속 CPU가 유용한 작업을 하였으면 CPU 사용 효율은 100%가 된다.

  - **처리율(throughput)**: 시간당 완료되는 프로세스의 수
  - **반환시간(turnaround time)**: 하나의 프로세스가 제시된 시점에서 그 프로세스가 종료될 때까지 걸린 시간
  - **대기시간(waiting time)**: 한 프로세스가 준비완료 큐에서 대기한 총 시간
  - **응답시간(response time)**: 서비스를 요청한 후에 그 서비스에 대한 첫 반응이 나오기까지 걸린 시간
- 위 기준은 사용자 관점(반환시간, 응답시간 등)과 시스템 관점(CPU 사용 효율, 처리율)으로 나눌 수 있다.
- CPU 사용 효율과 처리율은 최대화하고 반환시간, 대기시간, 응답시간은 최소화하는 알고리즘이 가장 이상적이다.
- 응답시간의 경우에는 평균 응답시간을 최소화하는 것보다 편차를 최소화하는 것이 더 우수한 알고리즘이라고 주장하는 이도 있다.

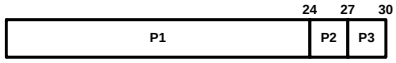
6.3 스케줄링 알고리즘

6.3.1 FCFS 스케줄링

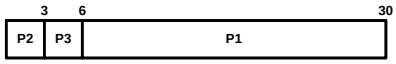
- **FCFS(First-Come First-Served)** 스케줄링 알고리즘은 먼저 요청한 프로세스 순으로 스케줄링한다.
- 이 알고리즘은 비선점 방식이다.
- FCFS 알고리즘은 FIFO 큐를 이용하여 쉽게 구현할 수 있다.
- 예) 다음과 같은 프로세스가 있을 때 스케줄링 결과를 관찰하여 보자.

프로세스	도착시간	버스트 시간
P_1	0 ms	24 ms
P_2	1 ms	3 ms
P_3	2 ms	3 ms

  프로세스의 도착 순서가 P_1, P_2, P_3이면 스케줄링 결과의 Gantt 도표는 다음과 같다.



  P_1의 대기시간은 0 ms, P_2는 $24-1=23$ ms, P_3는 $27-2=25$ ms이므로 평균 대기시간은 16 ms이다. 만약 도착 순서가 P_2, P_3, P_1이면 이 때 Gantt 도표는 다음과 같으며,



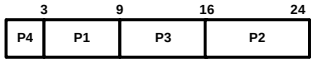
  평균 대기 시간은 2 ms이다.
- FCFS의 단점은 호위 효과(convoy effect)가 발생할 수 있다는 것이다. 호위 효과란 하나의 큰 프로세스가 CPU를 양보할 때까지 다른 모든 프로세스가 기다리는 현상을 말한다.

6.3.2 SJF 스케줄링

- **SJF(Shortest-Job-First)** 스케줄링 알고리즘은 다음 CPU 버스트 시간이 가장 짧은 프로세스 순으로 스케줄링한다.
- 만약 프로세스가 같은 CPU 버스트 시간을 가지면 FCFS 정책을 따른다.
- 예) 다음과 같은 프로세스가 있을 때 스케줄링 결과를 관찰하여 보자.

프로세스	버스트 시간
P_1	6 ms
P_2	8 ms
P_3	7 ms
P_4	3 ms

  위 프로세스가 모두 현재 준비완료 큐에 있다고 가정하면(모두 도착시간이 0 ms) 스케줄링 결과는 다음과 같다.



  평균 대기 시간은 $(3+16+9+0)/4=7$ ms이다. 만약 P_1, P_2, P_3, P_4 순서로 FCFS 알고리즘으로 스케줄링 하였다면 평균 대기 시간은 10.25 ms이다.
- SJF 알고리즘은 평균 대기 시간 측면에서는 최적에 가장 가까울 것이다.
- SJF 알고리즘의 가장 큰 어려움은 프로세스의 다음 CPU 버스트 시간을 예측하는 것이다.
- 보통 다음 CPU 버스트 시간은 이전의 CPU 버스트들의 길이를 지수 평균(exponential average)하여 구한다.
- 지수 평균은 다음과 같이 정의된다.

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

  여기서 t_n은 n번째 CPU 버스트 시간이고, τ_n은 예측한 n번째 CPU 버스트 시간이다. α는 0과 1사이의 수이며, 가중치를 결정한다. $\alpha = 0.5$를 가장 널리 사용한다.
- 어떤 고정된 값 또는 전체 시스템 평균을 τ_0로 사용한다.
- 예) $\alpha = 0.5$이고 $\tau_0 = 10$ ms라고 가정할 때 예측 시간과 실제 시간을 비교하면 다음과 같다.

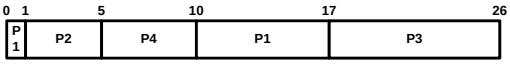
CPU 버스 시간		6	4	6	4	13	13
예측 시간	10	8	6	6	5	9	11

- 선점 SJF 알고리즘에서 새 프로세스가 준비완료 큐에 도착하면 이 프로세스의 다음 CPU 버스트 시간과 현재 수행중인 프로세스의 남은 CPU 버스트 시간을 비교한다. 이 때 새 프로세스의 다음 시간이 수행 중 프로세스의 남은 시간보다 적으면 기존 프로세스를 강제로 종료하고 새 프로세스를 할당한다.

- 선점 SJF 알고리즘은 다른 말로 **SRTF(Shortest-Remaining-Time-First)** 알고리즘이라 한다.
- 예) 다음과 같은 프로세스가 있을 때 스케줄링 결과를 관찰하여 보자.

프로세스	도착 시간	버스트 시간
P_1	0 ms	8 ms
P_2	1 ms	4 ms
P_3	2 ms	9 ms
P_4	3 ms	5 ms

스케줄링 결과는 다음과 같다.



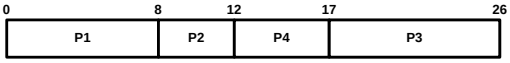
평균 대기 시간은 다음과 같으며,

$((10-1)+(1-1)+(17-2)+(5-3))/4 = 6.5$ ms

평균 반환 시간은 다음과 같다.

$((17-0)+(5-1)+(26-2)+(10-3))/4 = 13$ ms

비선점 방식의 SJF 알고리즘으로 스케줄링 하였을 경우에 스케줄링 결과는 다음과 같다.



평균 대기 시간은 다음과 같으며,

$(0+(8-1)+(17-2)+(12-3))/4 = 7.75$ ms

평균 반환 시간은 다음과 같다.

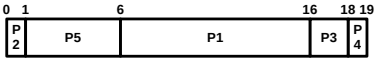
$((8-0)+(12-1)+(26-2)+(17-3))/4 = 14.25$ ms

6.3.3 우선순위 스케줄링

- **우선순위** 스케줄링은 우선순위가 높은 프로세스에게 먼저 CPU를 할당한다. 우선순위가 같으면 FCFS 정책으로 할당한다.
- SJF도 우선순위 스케줄링 중 하나이다.
- 우선순위는 보통 정수값으로 나타내며, 0에서 7까지와 같은 고정된 범위를 가진다. 이 때 보통 낮은 수일수록 높은 우선순위를 나타낸다.
- 예) 다음과 같은 프로세스가 있을 때 스케줄링 결과를 관찰하여 보자.

프로세스	버스트 시간	우선 순위
P_1	10 ms	3
P_2	1 ms	1
P_3	2 ms	4
P_4	1 ms	5
P_5	5 ms	2

5개의 프로세스가 모두 0 ms에 도착했다고 하자. 스케줄링 결과는 다음과 같다.



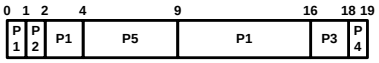
평균 대기 시간은 다음과 같다.

$$(6+0+16+18+1)/5 = 8.2 \text{ ms}$$

- 우선순위는 내부요인에 의해 결정(운영체제가 판단)될 수 있고 외부요인에 의해 결정(사용자가 지정)될 수 있다.
- 선점 우선 순위 방식에서는 선점 SJF 알고리즘과 마찬가지로 도착한 프로세스가 현재 수행중인 프로세스보다 우선순위가 높으면 현재 프로세스는 중단되고 새 프로세스가 CPU에 할당되어 실행된다.
- 예) 다음과 같은 프로세스가 있을 때 스케줄링 결과를 관찰하여 보자.

프로세스	도착 시간	버스트 시간	우선 순위
P_1	0 ms	10 ms	3
P_2	1 ms	1 ms	1
P_3	2 ms	2 ms	4
P_4	3 ms	1 ms	4
P_5	4 ms	5 ms	2

5개의 프로세스가 같은 시간에 도착했다고 하자. 스케줄링 결과는 다음과 같다.



평균 대기 시간은 다음과 같다.

$$(6+0+(16-2)+(18-3)+0)/5 = 7 \text{ ms}$$

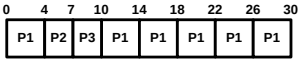
- 우선순위 알고리즘의 문제점은 영구 대기(indefinite blocking) 또는 굶주림(starvation) 현상이 발생할 수 있다는 것이다. 우선순위가 낮은 프로세스는 평생 실행되지 않을 수 있다.
- 이것을 극복하기 위해 aging 기법을 사용한다. aging 기법은 대기하는 프로세스의 우선순위를 점진적으로 증가시켜 준다.

6.3.4 라운드 로빈 스케줄링

- **라운드 로빈(RR, Round-Robin)** 스케줄링 알고리즘은 시분할 시스템에서 사용하는 알고리즘이다.
- 이 알고리즘에서는 **시간 조각(time quantum, time slice)**이라고 하는 작은 시간을 정의하여 이 시간이 경과될 때마다 현재 프로세스를 중단하고 다음 프로세스를 실행한다.
- 보통 준비완료 큐를 순환 버퍼로 만들어 구현한다.
- 예) 다음과 같은 프로세스가 있을 때 스케줄링 결과를 관찰하여 보자.

프로세스	버스트 시간
P_1	24 ms
P_2	3 ms
P_3	3 ms

위 프로세스가 모두 같은 시간에 도착했고 시간 조각이 4 ms이라고 가정하면 스케줄링 결과는 다음과 같다.

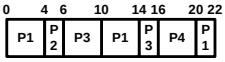


평균 대기 시간은 $(6 + 4 + 7)/3 = 5.66$ ms이다.

- 예) 다음과 같은 프로세스가 있을 때 스케줄링 결과를 관찰하여 보자.

프로세스	도착 시간	버스트 시간
P_1	0 ms	10 ms
P_2	1 ms	2 ms
P_3	2 ms	6 ms
P_4	13 ms	4 ms

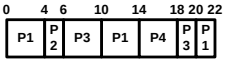
스케줄링 결과는 다음과 같다.



평균 대기 시간은 다음과 같다.

$$(12 + 3 + 8 + 3)/4 = 6.5 \text{ ms}$$

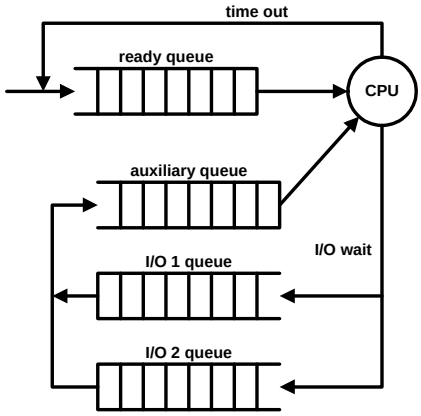
만약 P_4의 도착 시간이 9 ms이면 스케줄링 결과는 다음과 같다.



평균 대기 시간은 다음과 같다.

$$(12 + 3 + 12 + 1)/4 = 7 \text{ ms}$$

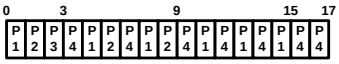
- RR 알고리즘은 기본적으로 선점 방식이다.
- RR 알고리즘의 성능은 시간 조각에 크게 의존한다. 이 크기가 매우 크면 RR 알고리즘은 FCFS 알고리즘과 큰 차이가 없다. 반대로 매우 작으면 문맥 전환이 많이 발생한다.
- RR 알고리즘의 문제점은 계산 중심 프로세스가 입출력 중심 프로세스보다 많은 시간을 할당받게 된다는 것이다(공정하지 못함).
  - 계산 중심 프로세스는 자신의 시간 조각을 다 사용한 다음에 바로 다시 준비완료 큐에서 대기한다. 반면에 입출력 중심 프로세스는 자신의 시간 조작을 보통 다 사용하지 못하고 입출력 큐에서 대기하게 된다. 이런 대기 사이에 계산 중심 프로세스는 계속 CPU 시간을 할당받게 된다.
  - 이 문제를 해결하기 위해 입출력을 완료한 프로세스는 준비완료 큐로 옮기지 않고 별도의 큐로 옮긴다. 현재 수행중인 프로세스의 시간 조각이 끝나면 준비완료 큐보다 먼저 이 큐에 있는 프로세스에게 시간 조각을 할당해준다. 그림 6.3 참조.
- 반환시간도 시간 조각의 크기에 의존한다.
- 예) 다음과 같은 프로세스가 있을 때 시간 조각의 증가에 따른 반환시간을 관찰하여 보자.



<그림 6.3> 가상 RR 스케줄러

프로세스	버스트 시간
P_1	6 ms
P_2	3 ms
P_3	1 ms
P_4	7 ms

시간 조각이 1 ms일 때는 다음과 같다.

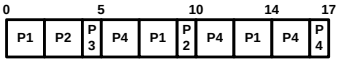


이 때 평균 반환시간은 다음과 같다.

$$(15 + 9 + 3 + 17)/4 = 11 \text{ ms}$$

반환시간 대신에 정규화된 반환시간(normalized turnaround time)을 측정하는 경우도 있다. 정규환된 반환시간은 시스템에 머문 전체 시간과 실행 시간의 비율로 측정된다. $P_1 = 15/6 = 2.5$, $P_2 = 9/3 = 3$, $P_3 = 3/1 = 3$, $P_4 = 17/7 = 2.4$이므로 평균 정규화된 반환시간은 2.725이다.

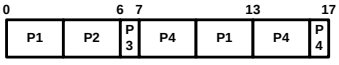
시간 조각이 2 ms일 때는 다음과 같다.



이 때 평균 반환시간은 다음과 같다.

$$(14 + 10 + 5 + 17)/4 = 11.5 \text{ ms}$$

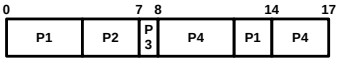
시간 조각이 3 ms일 때는 다음과 같다.



이 때 평균 반환시간은 다음과 같다.

$$(13 + 6 + 7 + 17)/4 = 10.75 \text{ ms}$$

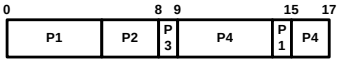
시간 조각이 4 ms일 때는 다음과 같다.



이 때 평균 반환시간은 다음과 같다.

$$(14 + 7 + 8 + 17)/4 = 11.5 \text{ ms}$$

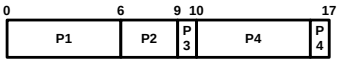
시간 조각이 5 ms일 때는 다음과 같다.



이 때 평균 반환시간은 다음과 같다.

$$(15 + 8 + 9 + 17)/4 = 12.25 \text{ ms}$$

시간 조각이 6 ms일 때는 다음과 같다.



이 때 평균 반환시간은 다음과 같다.

$$(6 + 9 + 10 + 17)/4 = 10.5 \text{ ms}$$

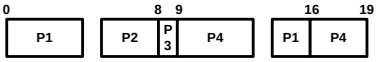
이 이상 시간 조각을 증가하여도 반환시간은 변하지 않는다.

- 위 예에서는 문맥전환에 소요되는 시간은 고려하지 않았다. 문맥전환에 1 ms가 소요된다면 시간 조각이 2 ms일 때와 4 ms일 때 반환시간을 계산하여 보자. 먼저 시간 조각이 2 ms일 때는 다음과 같다.

이 때 평균 반환시간은 다음과 같다.

$$(19 + 14 + 7 + 22)/4 = 15.5 \text{ ms}$$

시간 조각이 4 ms일 때는 다음과 같다.



이 때 평균 반환시간은 다음과 같다.

$$(16 + 8 + 9 + 19)/4 = 13 \text{ ms}$$

- 일반적으로 CPU 버스트 시간의 80%는 시간 조각보다 적어야 가장 바람직하다.

6.3.5 다중레벨 큐 스케줄링

- 다중레벨 큐 스케줄링에서는 준비완료 큐를 여러 개의 큐로 나누어 사용한다.
- 프로세스는 그것의 특성에 따라 특정 큐에 할당된다.
- 각 큐는 독자적인 스케줄링 알고리즘을 사용한다.
- 큐 간에는 별도 스케줄링 알고리즘을 사용한다.
  - 방법 1. 상위 우선순위 큐는 하위 우선순위 큐보다 절대적 우선순위를 가지도록 할 수 있다. 이 경우에는 하위 큐에 있는 프로세스는 상위에 있는 모든 큐가 비어있을 경우에만 스케줄링된다.
  - 방법 2. 큐 간에 일정 비율로 CPU 시간을 할당해 줄 수 있다.

6.3.6 다중레벨 피드백 큐 스케줄링

- 일반 다중레벨 큐 스케줄링은 유연성이 떨어진다.
- 다중레벨 피드백 큐 스케줄링은 프로세스가 큐 간에 이동할 수 있도록 해준다.
- 이 스케줄링의 주 목표는 CPU 버스트 시간 특성이 다른 프로세스들을 분리하여 굶주림 현상과 호위 효과 현상을 제거하는 것이다.
- 대화식 프로세스와 입출력 중심 프로세스는 상위 큐에 할당되고 CPU 중심 프로세스는 하위 큐에 할당된다.
- 다중레벨 피드백 큐를 결정하는 파라미터
  - 큐의 개수
  - 각 큐의 스케줄링 알고리즘
  - 프로세스를 높은 우선순위 큐로 올려주는 시기를 결정하는 방법
  - 프로세스를 낮은 우선순위 큐로 내려주는 시기를 결정하는 방법
  - 프로세스가 준비완료 큐에 들어올때 어떤 큐에 할당할지를 결정하는 방법

6.4 다중 프로세서 스케줄링

- 다증 프로세스에서 스케줄링할 때 고려사항
  - 가정: 프로세스는 동일하다.
  - 만약 특정 입출력 장치가 하나의 프로세서의 시스템 버스에만 연결되어 있다면 그 장치를 사용해야 하는 프로세스는 그 프로세서에 할당되어야 한다.
  - 각 프로세서마다 별도의 준비완료 큐를 두면 한 큐는 비어 있고 다른 큐에는 많은 프로세스가 대기중일 수 있다. 따라서 보통은 공통된 준비완료 큐를 사용한다.
  - 각 프로세서가 스스로 큐에서 선택하면 두 프로세서가 같은 프로세스를 선택하는 등의 문제가 발생할 수 있다. 이것을 해결하기 위해 하나의 프로세서는 스케줄링만 담당하도록 할 수 있다.

6.5 실시간 스케줄링

- 실시간 시스템 중에 엄격한 실시간 시스템은 반드시 정해진 시간 내에 프로세스의 수행을 완료해야 한다.
- 보통 프로세스는 제출할 때 끝내야 하는 시간을 함께 제출한다. 스케줄러는 이 시간내에 이 프로세스의 수행을 완료할 수 없으면 거부한다.
- 일반 범용 컴퓨터에서 각 작업의 최대 소요 시간을 예측하는 것이 어렵다. 따라서 엄격한 실시간 시스템은 특수 소프트웨어와 실시간 작업에 맞게 제작된 특수 하드웨어를 사용하여 구현된다.

- 완화된 실시간 시스템은 덜 제한적이므로 실시간 프로세스에게 높은 우선순위를 주어 실행하면 된다.
- 완화된 실시간 시스템에서 스케줄러는 다음과 같은 특성을 가지고 있어야 한다.
  - 우선순위 스케줄링 알고리즘을 사용해야 하며, 실시간 프로세스에게는 가장 높은 우선순위를 부여해 주어야 한다.
  - 실시간 프로세스는 우선순위가 낮아져서는 안 된다.
  - 디스패치 지연이 작어야 한다.
- 디스패치 지연을 최소화하는 방법
  - 시스템 호출도 중간에 중단시킨다. 이를 위해 중단 지점(preemption point)을 사용한다. 즉, 긴 시스템 호출 중간에 안전하게 중단할 수 있는 지점을 지정하여 이 지점에서는 중단시킬 수 있도록 한다.
  - 낮은 우선순위 프로세스가 실시간 프로세스가 필요로 하는 자원을 사용하고 있을 수 있다. 이 경우 우선순위 상속 프로토콜(priority-inheritance protocol)을 이용하여 낮은 우선순위도 높은 우선순위를 갖도록 하여 빨리 완료하도록 할 수 있다.

6.6 알고리즘 평가

- 스케줄링 알고리즘을 선택하기에 앞서 시스템의 요구사항을 분석하여야 한다. 이를 통해 알고리즘을 비교할 기준을 정해야 한다.
- 기준의 예
  - 최대 응답시간이 1초라는 제약 조건 하에서 CPU 사용 효율을 최대화하는 알고리즘
  - 한 프로세스의 반환시간이 그 프로세스의 전체 실행 시간에 선형적으로 비례하도록 처리율을 극대화하는 알고리즘
- 기준이 정해졌으면 여러 분석 방법을 이용하여 그 기준에 가장 적합한 알고리즘을 선택한다.
- 분석 방법의 종류
  - 결정적 모델: 주어진 데이터를 가지고 실제 계산하여 비교하는 방법이다. 이 방법은 계산 결과가 사용한 데이터에만 적용된다는 문제점이 있다.
  - 큐잉 모델: 수학적으로 분석하는 방법이다. 하지만 수학적으로 분석하기 위해서는 여러 가정을 해야 하며, 가정된 데이터가 현실과 거리가 있을 수 있다.
  - 시뮬레이션: 가장 정확한 분석 방법이지만 많은 시간이 소요되며, 시뮬레이터를 설계하고 구현하는 것이 쉽지 않을 수 있다.

6.7 프로세스 스케줄링 모델

6.7.1 솔라리스 2

- 우선순위 스케줄링 알고리즘을 사용한다.
- 우선순위를 실시간, 시스템, 시분할, 대화식 네 가지 유형으로 분류한다.
- 프로세스는 기본적으로 시분할 유형으로 분류된다. 시분할 유형에 대한 스케줄링 알고리즘은 다중레벨 피드백 큐를 사용하며, 각 큐마다 다른 길이의 시간 조각을 사용한다.
  - 우선순위와 시간 조각은 역관계가 성립한다. 우선순위가 높을 수록 시간 조각은 작다.
  - 상호작용 프로세스는 우선순위가 높고, 계산 중심 프로세스는 우선순위가 낮다. 따라서 상호작용 프로세스에게는 높은 응답성을 제공하며, 계산 중심 프로세스는 높은 처리율을 제공한다.
- 솔라리스는 시스템 유형을 이용하여 스케줄러와 같은 커널 프로세스를 실행한다. 시스템 유형에 할당된 프로세스는 시분할 방식을 사용하지 않으며, 대기 상태가 되거나 상위 우선순위 프로세스가 선점할 때까지 실행된다.

6.7.2 윈도우즈 2000

- 선점방식의 우선순위 스케줄링 알고리즘을 사용한다.
- CPU에 할당된 스레드는 다음 상황이 발생할 때까지 계속 실행된다.
  - 자신보다 높은 우선순위 프로세스에 의해 선점된 경우
  - 대기해야 하는 시스템 호출을 하는 경우
  - 시간 조각이 끝난 경우
- 윈도우즈 2000은 총 32레벨의 우선순위를 사용한다. 이 우선순위 레벨은 크게 가변 클래스(1에서 15)와 실시간 클래스(16에서 31)로 구분된다.
- 각 우선순위 레벨마다 큐가 있으며, 만약 모든 큐가 비어 있으면 스케줄러는 “idle thread”을 실행한다.
- 가변 클래스에 속한 스레드의 실행이 시간 조각의 경과로 인트럽트되면 이 스레드의 우선순위를 한 레벨 낮춘다. 또한 대기중이던 스레드가 다시 준비완료 상태로 바뀌면 스케줄러는 스레드가 대기하던 사건에 따라 스레드의 우선순위를 높여준다. 이를 통해 대화식 스레드는 높은 우선순위로 실행될 수 있다.

6.7.3 리눅스

- 리눅스는 공정한 선점 방식의 시분할 알고리즘과 절대적 우선순위를 사용하는 실시간 프로세스를 위한 알고리즘 두 가지를 사용한다.
- 리눅스에서는 실시간 프로세스라 하더라도 커널 프로세스를 선점할 수 없다.
- 리눅스는 시분할을 위해 우선순위를 가지는 credit 기반 알고리즘을 사용한다.
  - 각 프로세스는 특정한 수의 스케줄링 credit을 가지고 있다.
  - 가장 많은 스케줄링 credit을 가지고 있는 프로세스 순으로 스케줄링 한다.
  - 타이머 인터럽트가 발생할 때마다 현재 실행중인 프로세스는 하나의 credit을 잃게된다. Credit이 0이 되면 현재 프로세스를 중단하고 다음 프로세스를 실행한다.
  - 실행 가능한 모든 프로세스의 credit이 전부 0이면 시스템에 있는 모든 프로세스에게 credit을 다음 규칙에 따라 재할당한다.

    $$\text{credits} = \frac{\text{credits}}{2} + \text{priority}$$

    즉, 프로세스의 역사와 그것의 우선순위를 함께 고려한다.
- 리눅스는 두 개의 실시간 스케줄링 클래스를 제공한다. 하나는 FCFS 방식이고, 다른 하나는 RR 방식이다. 이 두 방식의 차이는 FCFS는 선점되지 않으나 RR 방식에서는 같은 우선순위를 가지는 프로세스가 CPU 시간을 공정하게 나누어 사용하게 된다.