CH05 CPU Scheduling

Objectives

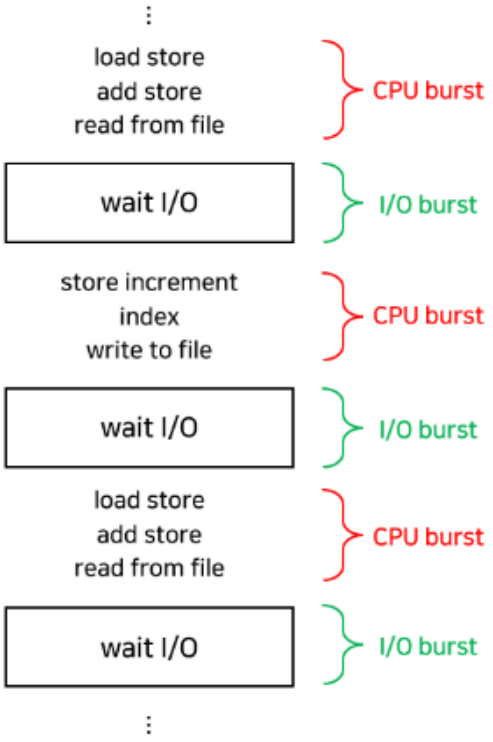
- 다양한 CPU 스케줄링 알고리즘을 설명한다.
- 스케줄링 기준에 따라 CPU 스케줄링 알고리즘을 평가한다.
- 다중 처리기 및 다중 코어 스케줄링과 관련된 쟁점을 설명한다.
- 다양한 실시간 스케줄링 알고리즘을 설명한다.

Basic Concepts

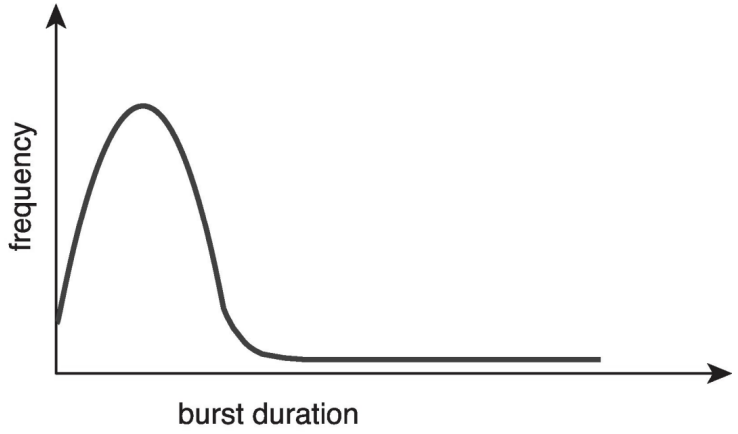
> 💡 다중 프로그래밍의 목적은 CPU 이용률을 최대화하기 위해 항상 실행 중인 프로세스를 가지게 하는 데 있다.

CPU-I/O Burst Cycle

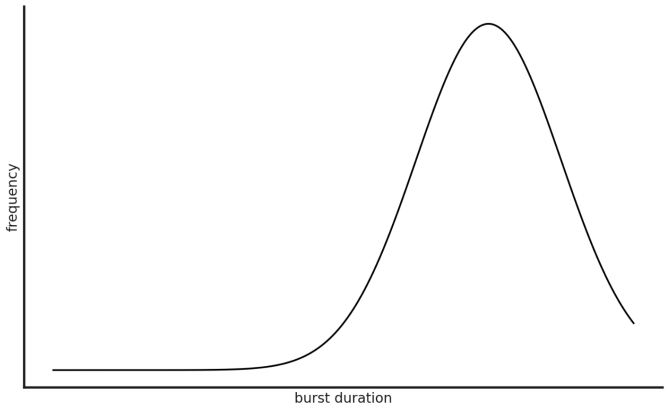
- CPU Burst : 명령어를 실행하는 시간
- I/O Burst : 입출력을 실행하는 시간



- 프로세스 실행은 CPU Burst로 시작된다. 뒤이어 I/O Burst가 발생하고, 그 뒤를 이어 또다른 CPU Burst가 발생한다. 이를 반복한다.
- CPU 지향 프로그램은 다수의 긴 CPU Burst를 가지며, I/O 지향 프로그램은 다수의 짧은 CPU Burst를 가진다.



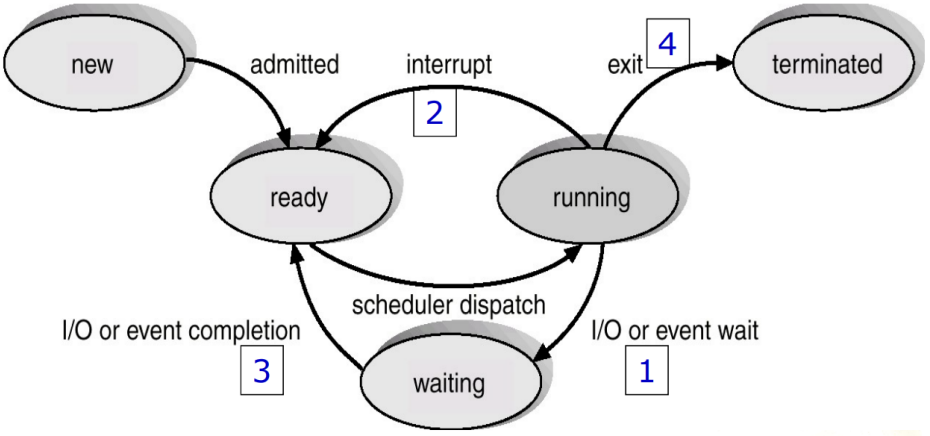
- 이 그림과 같이, 짧은 Cpu burst duration의 빈도수가 많으면 해당 프로세스는 I/O Bound Job로 여긴다.



- 이 그림과 같이, 긴 Cpu burst duration의 빈도수가 많으면 해당 프로세스는 CPU Bound Job으로 여긴다.

CPU Scheduler

- CPU가 유휴 상태가 될 때마다, 운영체제는 준비 큐에 있는 프로세스 중에서 하나를 선택해 실행해야 한다. 선택 절차는 CPU 스케줄러에 의해 수행된다. 스케줄러는 실행 준비가 되어 있는 메모리 내의 프로세스 중에서 선택하여, 이들 중 하나에게 CPU를 할당한다.
- CPU 스케줄링 결정은 다음의 네 가지 상황에서 발생할 수 있다.



1. 한 프로세스가 실행 상태에서 대기 상태로 전환될 때(ex. I/O 요청이나 자식 프로세스가 종료되기를 기다리기 위해 wait()를 호출할 때)
2. 프로세스가 실행 상태에서 준비 완료 상태로 전환될 때(ex. 인터럽트가 발생할 때)
3. 프로세스가 대기 상태에서 준비 완료 상태로 전환될 때(ex. I/O의 종료)
4. 프로세스가 종료할 때

→ 상황 1과 4의 경우 이러한 스케줄링 방법을 비선점이라고 한다. (스케줄링 면에서 선택의 여지가 없다.) 비선점 스케줄링하에서는, 일단 CPU가 한 프로세스에 할당되면 프로세스가 종료하든지, 또는 대기 상태로 전환해 CPU를 방출할 때까지 점유한다.

비선점(nonpreemptive)와 선점(preemptive)

💬 **[비선점 Scheduling]** 프로세스에 CPU 제어권이 주어지면 프로세스가 실행을 완료하거나 CPU를 자발적으로 포기할 때까지 실행

💬 **[선점 Scheduling]** 프로세스에 CPU 제어권이 주어지더라도 OS가 실행을 멈추고 다른 프로세스에 제어권을 넘길 수 있음 *Windows, macOS, Linux 및 UNIX를 포함한 거의 모든 최신 운영체제들은 선점 스케줄링 알고리즘을 사용한다.

Dispatcher

> 디스패처는 CPU 코어의 제어를 CPU 스케줄러가 선택한 프로세스에 주는 모듈이다.

- 한 프로세스에서 다른 프로세스로 문맥을 교환하는 일
- 사용자 모드로 전환하는 일
- 프로그램을 다시 시작하기 위해 사용자 프로그램의 적절한 위치로 이동(jump)하는 일

→ 디스패처는 모든 프로세스의 문맥 교환 시 호출되므로, 가능한 최고로 빨리 수행되어야 한다.

✔️ **디스패치 지연(dispatch latency)** : 디스패처가 하나의 프로세스를 정지하고 다른 프로세스의 수행을 시작하는 데까지 소요되는 시간

스케줄링 기준(Scheduling Criteria)

[1] CPU 이용률(utilization)

- 가능한 한 CPU를 최대한 바쁘게 유지하기를 원한다.

[2] 처리량(Throughput)

- 단위 시간당 완료된 프로세스의 개수

[3] 총처리 시간(Turnaround Time)

- `프로세스를 실행하는 데 소요된 시간 = 프로세스의 완료 시간 - 프로세스의 제출 시간`
- 준비 큐에서 대기한 시간, CPU에서 실행하는 시간, I/O시간의 합

[4] 대기 시간(Waiting Time)

- 준비 큐에서 대기하면서 보낸 시간의 합

[5] 응답 시간(Response Time)

- 하나의 요구를 제출한 후 첫 번째 응답이 시작되는 데까지 걸리는 시간(그 응답을 출력하는 데 걸리는 시간은 아니다.)

🌟 CPU 이용률과 처리량은 **최대화**하고, 총처리 시간, 대기시간, 응답 시간은 **최소화**하는 것이 바람직하다.

스케줄링 알고리즘(Scheduling Algorithm)

선입 선처리 스케줄링(First come, First served Scheduling)

> 💬 CPU를 **먼저 요청하는 프로세스**가 CPU를 **먼저 할당**받는다.

- 선입 선처리 정책은 선입선출(FIFO) 큐로 쉽게 관리할 수 있다. 프로세스가 준비 큐에 진입하면, 이 프로세스의 프로세스 제어 블록(PCB)을 큐의 끝에 연결한다. CPU가 가용 상태가 되면, 준비 큐의 앞부분에 있는 프로세스에 할당된다. 이 실행 상태의 프로세스는 이어 준비 큐에서 제거된다.
- 선입 선처리 스케줄링 알고리즘은 `비선점형` 이다. 즉, 일단 CPU가 한 프로세스에 할당되면, 그 프로세스가 종료하든지 또는 I/O 처리를 요구하든지 하여 CPU를 방출할 때까지 CPU를 점유한다.

▼ Gantt Chart

Process	Burst Time
P_1	24
P_2	3
P_3	3

프로세스들이 P1, P2, P3 순으로 도착하고, 선입 선처리 순으로 서비스 받는다고 가정하자.

평균 대기 시간은 다음과 같다.

- (P1의 대기 시간(0) + P2의 대기 시간(24) + P3의 대기 시간(27))/3 = 17밀리초

만약 프로세스가 P2, P3, P1 순으로 도착한다면 평균 대기 시간은 다음과 같다.

- (P2의 대기 시간(0) + P3의 대기 시간(3) + P1의 대기 시간(6))/3 = 3밀리초

→ 선입 선처리 정책 하에서 평균 대기 시간은 일반적으로 최소가 아니며, 프로세스 CPU 버스트 시간이 크게 변할 경우 변화할 수 있다.

- 모든 다른 프로세스들이 하나의 긴 프로세스가 CPU를 양도하기를 기다리는 것을 호위 효과(convoy effect)라고 한다. 이 효과는 짧은 프로세스들이 먼저 처리되도록 허용될 때보다 CPU와 장치 이용률이 저하되는 결과를 낳는다.

최단 작업 우선 스케줄링(Shortest-Job-First Scheduling)

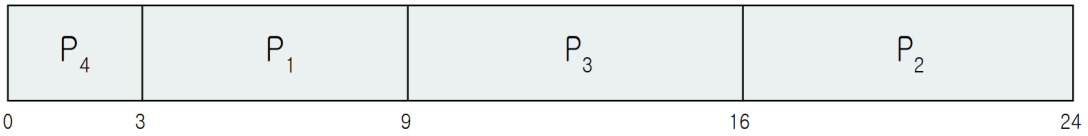
> 💬 각 작업의 프로세서 **실행 시간**을 이용하여 프로세서가 사용 가능할 때 **실행 시간이 가장 짧은 작업에 할당하는 방법**

- 동일한 길이의 다음 CPU 버스트를 가지면, 순서를 정하기 위해 선입 선처리 스케줄링을 적용한다.
- 이 스케줄링은 프로세스의 전체 길이가 아니라 다음 CPU 버스트의 길이에 의해 스케줄링이 되기 때문에, 최단 다음 CPU 버스트(shortes-next-CPU-burst) 알고리즘이라는 용어가 더 적합하다.

▼ Gantt Chart

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

프로세스들이 P1, P2, P3,P4 순으로 도착한다고 가정하자.



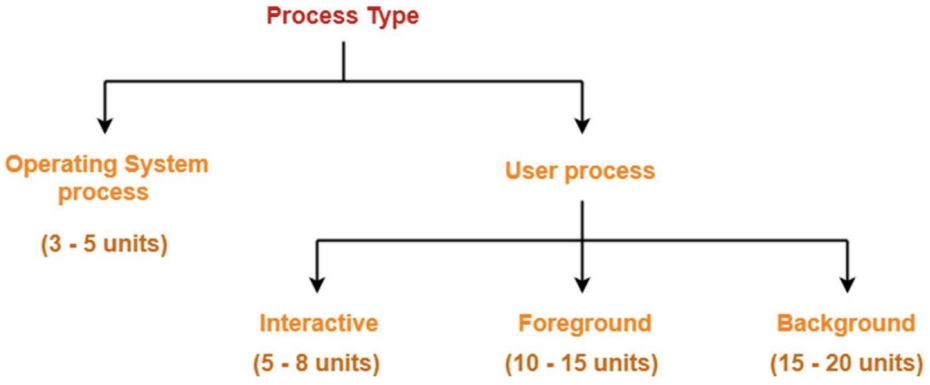
평균 대기 시간은 다음과 같다.

- (P4의 대기 시간(0) + P1의 대기 시간(3) + P3의 대기 시간(9) + P2의 대기 시간(16))/4 = 7밀리초

▼ SJF 알고리즘은 다음 CPU 버스트의 길이를 알 방법이 없기 때문에 CPU 스케줄링 수준에서 구현할 수 없다. 그러므로, CPU 버스트 길이의 근삿값을 계산해 이를 이용한다.

다음 CPU 버스트의 길이를 예측하는 방법

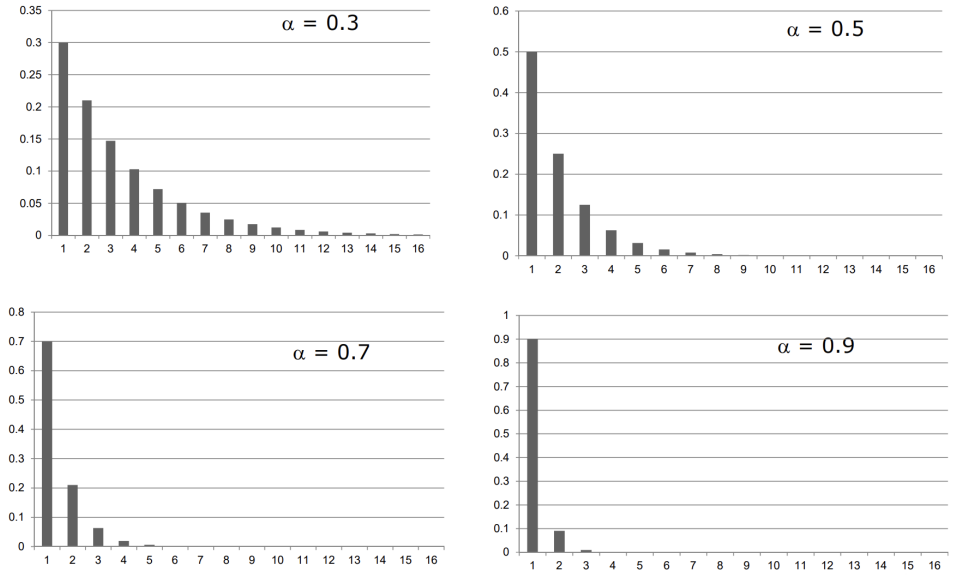
1) static 방법 : 프로세스 타입을 통해 길이를 예측한다.



2) dynamic 방법 : 지수 평균(exponential averaging)

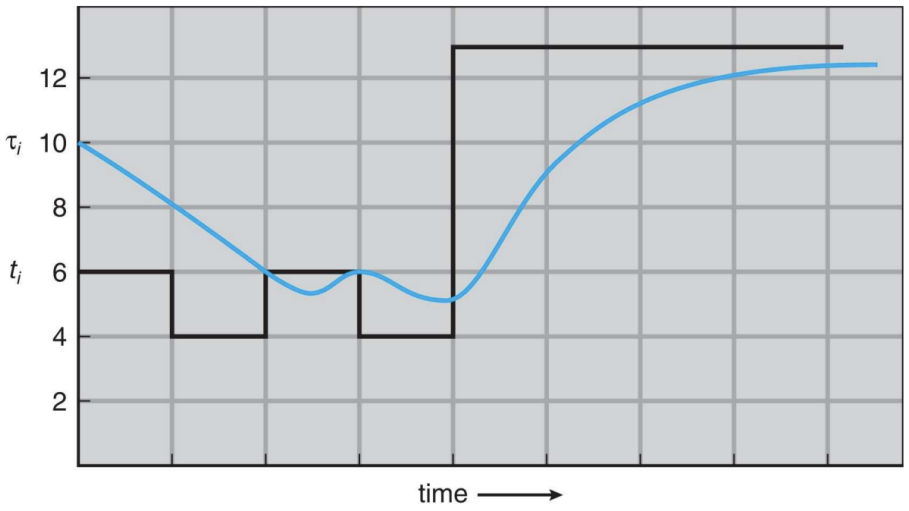
$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

- Tau(n+1) = 다음 CPU Burst에 대한 예측값
- Tau(n) = 과거의 역사
- t(n) = 최근의 정보
- α = 우리의 예측에서 최근의 값과 이전 값의 상대적인 무게를 제어



매개변수 값에 따른 CPU burst 가중치 변화(1에 가까울수록 최근 CPU Burst)

- 만약 이 매개변수 값이 0이면, 최근의 역사는 아무 영향이 없는 것이다.
- 만약 이 매개변수 값이 1이면, 가장 최근의 CPU burst만 중요시 된다.
- 만약 이 매개변수 값이 1/2이면, 최근의 역사와 과거의 역사가 같은 무게를 가진다.)



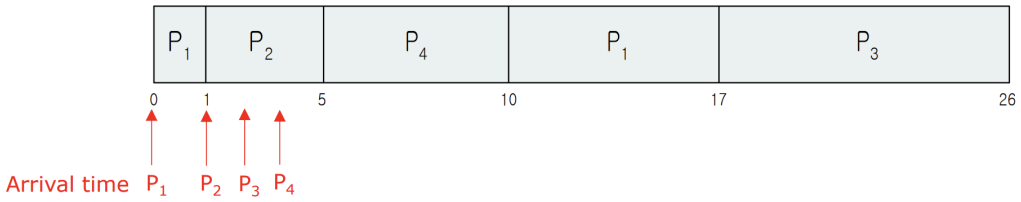
CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i) 10	8	6	6	5	9	11	12	...

다음 CPU 버스트의 길이 추정

- SJF 알고리즘은 선점형이거나 비선점형일 수 있다. 앞의 프로세스가 실행되는 동안 새로운 프로세스가 준비 큐에 도착하면 선택이 발생한다.
  - 비선점형일 경우, 현재 실행하고 있는 프로세스가 자신의 CPU 버스트를 끝내도록 허용한다. (만약 도착한 프로세스의 CPU 버스트가 더 짧을지라도)
  - 선점형일 경우, 때때로 최소 잔여 시간 우선(Shortest Remaining Time First-SRTF) 라고 불린다.

    ▼ SRTF Gantt

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



본 상황에서, 프로세스 P1은 큐에 있는 유일한 프로세스이므로 시간 0에 시작된다. 프로세스 P2는 시간 1에 도착한다. 프로세스 P1의 남은 시간(7밀리초) 프로세스 P2가 요구하는 시간(4밀리초)보다 크기 때문에 프로세스 P1이 선점되고 프로세스 P2가 스케줄 된다. 이 예에서 평균 대기 시간은 `[(10-1) + (1-1) + (17-2) + (5-3)]/4 = 6.5 밀리초`이다.

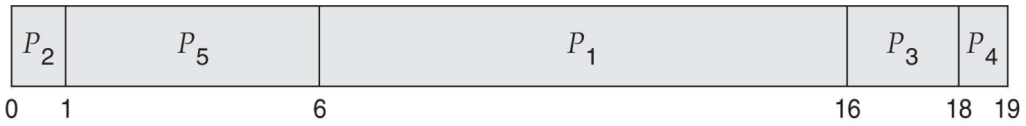
우선순위 스케줄링(Priority Scheduling)

> 💬 **준비 큐에 프로세스가 도착하면**, 도착한 프로세스의 **우선순위와** 현재 실행 중인 프로세스의 **우선순위**를 비교하여 우선순위가 **가장 높은 프로세스에** 프로세서를 할당하는 방식

- 우선순위가 각 프로세스들에 연관되어 있으며, CPU는 가장 높은 우선순위를 가진 프로세스에 할당된다. 우선순위가 같은 프로세스들은 선입 선처리(FCFS) 순서로 스케줄된다.
- 0이 최상위 또는 최하위 우선순위라는 점에 대한 명확한 기준은 없다. 해당 책에선 낮은 수가 높은 우선순위를 나타낸다고 가정하였다.
- 우선순위
  - 내부적 우선순위 : 시간 제한, 메모리 요구, 열린 파일의 수, 평균 I/O 버스트의 평균 CPU 버스트에 대한 비율 등이 계산에 사용
  - 외부적 우선순위 : 프로세스의 중요성, 컴퓨터 사용을 위해 지불되는 비용의 유형과 양, 그 작업을 후원하는 부서 그리고 정치적인 요인 등이 계산에 포함

▼ Gantt

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2



평균 대기 시간 = (P2(0) + P5(1) + P1(6) + P3(16) + P4(18))/5 = 8.2밀리초

- **문제**[기아 상태(starvation)]와 **해결 방안**[노화(aging)]
  - 기아 상태(starvation) : 실행 준비는 되어 있으나 CPU를 사용하지 못하는 프로세스는 CPU를 기다리면서 봉쇄된 것으로 간주할 수 있다. 우선순위 스케줄링 알고리즘을 사용할 경우 낮은 우선순위 프로세스들이 CPU를 무한히 대기하는 경우가 발생한다.
  - 노화(aging) : 오랫동안 시스템에서 대기하는 프로세스들의 우선순위를 점진적으로 증가시킨다.

라운드 로빈 스케줄링(RR, Round-Robin Scheduling)

> 💬 시분할 시스템을 위해 설계된 선점형 스케줄링의 하나로서, 프로세스들 사이에 우선순위를 두지 않고 순서대로 시간단위(Time Quantum/Slice)로 CPU를 할당하는 방식

- 라운드 로빈 스케줄링 알고리즘은 선입 선처리 스케줄링과 유사하지만 시스템이 프로세스들 사이를 옮겨 다닐 수 있도록 선점이 추가된다.
- 시간 할당량(time quantum), 또는 시간 슬라이스(Time Slice)라고 하는 작은 단위의 시간을 정의한다. 시간 할당량은 일반적으로 10에서 100밀리초 동안이다.
- 라운드로빈 스케줄링 알고리즘에서는, 유일하게 실행 가능한 프로세스가 아니라면 연속적으로 두 번 이상의 시간 할당량을 할당받는 프로세스는 없다. 만일 프로세스의 CPU 버스트가 한 번의 시간 할당량을 초과하면, 프로세스는 선점되고 준비 큐로 되돌아간다. 따라서 RR 스케줄링 알고리즘은 선점형 이다.

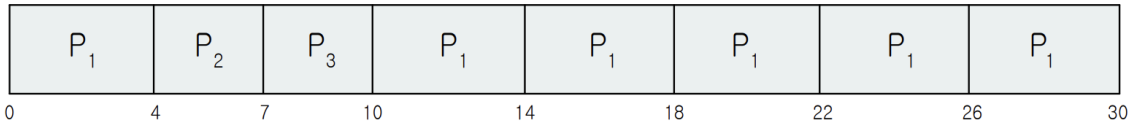
- 준비 큐에 n개의 프로세스가 있고 시간 할당량이 q이면, 각 프로세스는 최대 q시간 단위의 덩어리로 CPU 시간의 1/n을 얻는다. 각 프로세스는 자신의 다음 시간 할당량이 할당될 때까지 (n-1)xq 시간 이상을 기다리지는 않는다.
- 새로운 프로세스들은 준비 큐의 꼬리에 추가된다. CPU 스케줄러는 준비 큐에서 첫 번째 프로세스를 선택해 한 번의 시간 할당량 이후에 인터럽트를 걸도록 타이머(timer)를 설정한 후, 프로세스를 디스패치(dispatch) 한다.

  1) 프로세스의 CPU 버스트가 한 번의 시간 할당량보다 작은 경우 프로세스 자신이 CPU를 자발적으로 방출하고 스케줄러는 그 후 준비 큐에 있는 다음 프로세스로 진행한다.

  2) 현재 실행 중인 프로세스의 CPU 버스트가 한 번의 시간 할당량보다 긴 경우, 타이머가 끝나고 운영체제에 인터럽트가 발생할 것이다.

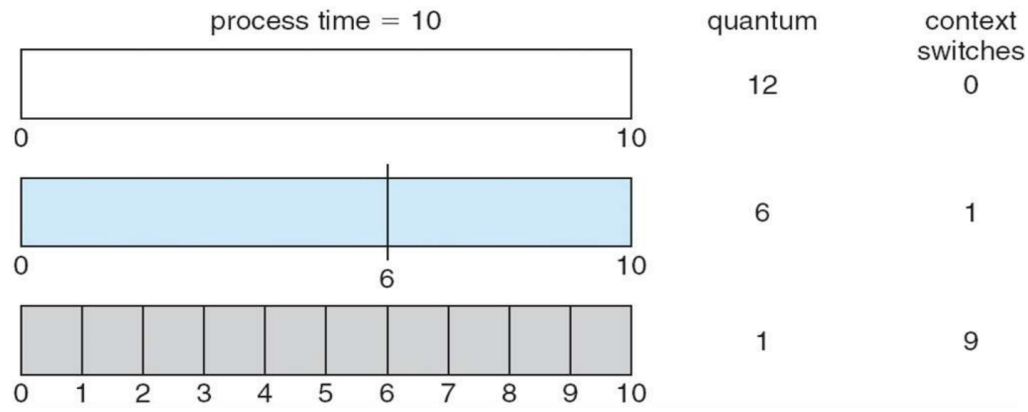
▼ Gantt

Process	Burst Time
P_1	24
P_2	3
P_3	3



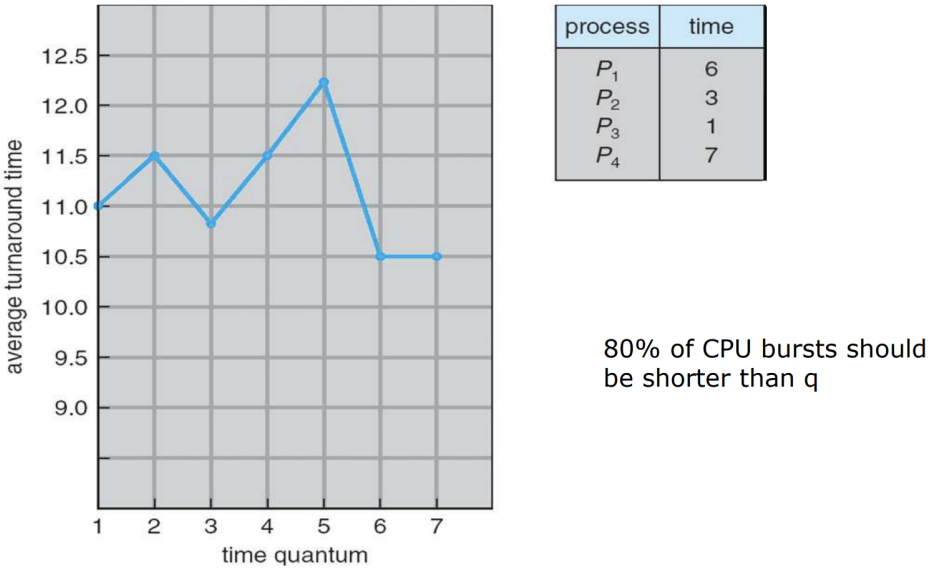
평균 대기시간 = (P1(10-4=6) + P2(4) + P3(7))/3 = 5.66밀리초

✔ RR 알고리즘의 성능과 시간 할당량



시간 할당량이 적을수록 문맥 교환 횟수가 늘어나는 모습

- RR 알고리즘의 성능은 시간 할당량의 크기에 매우 많은 영향을 받는다. 극단적인 경우, 시간 할당량이 매우 크면 RR 정책은 선입 선처리 정책과 같다. 이와 반대로 시간 할당량이 매우 적다면(예를 들면 1마이크로초) RR 정책은 매우 많은 문맥 교환을 야기한다. 그러므로 우리는 시간할당량이 문맥 교환 시간과 비교해 더 클 것을 원한다.

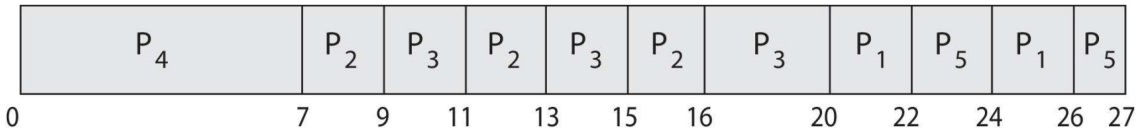


총처리 시간이 시간 할당량에 따라 변하는 모습

- 총처리 시간 또한 시간 할당량의 크기에 좌우된다.

+) Priority Scheduling With Round-Robin

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

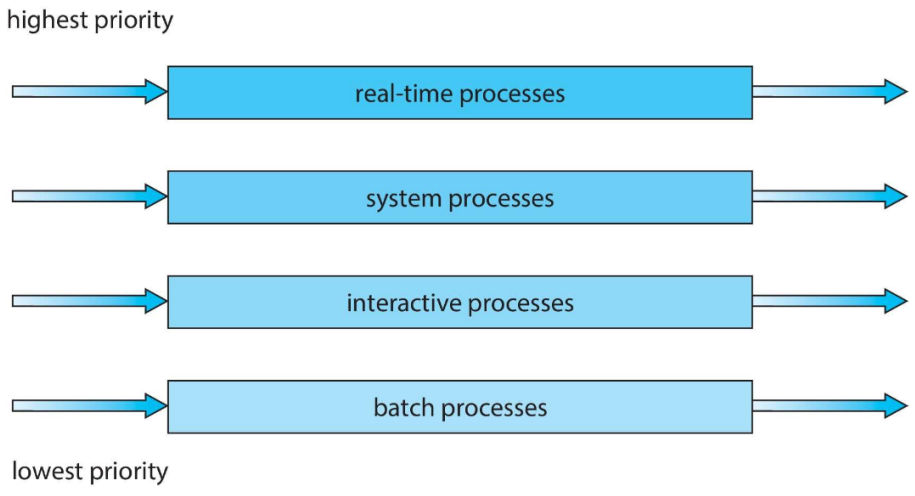


- 우선순위를 기준으로 스케줄링하되, 동일한 우선순위를 가질 경우 라운드 로빈으로 실행된다.

다단계 큐 스케줄링(Multilevel Queue Scheduling)

> 💬 **우선순위마다의 별도의 준비 큐를 형성**하여 스케줄링하는 방법

- 프로세스들은 프로세스의 특성(대화형, 배치(background) 등)에 따라 **우선순위가 부여되어 한 개의 큐에 영구적으로 할당되는데** 각 큐에는 그의 성격에 맞는 스케줄링 알고리즘을 별도로 적용할 수 있다. (FCFS, RR 등)
- 항상 가장 높은 **우선순위 큐의 프로세스에게 CPU를 먼저 할당해준다.**



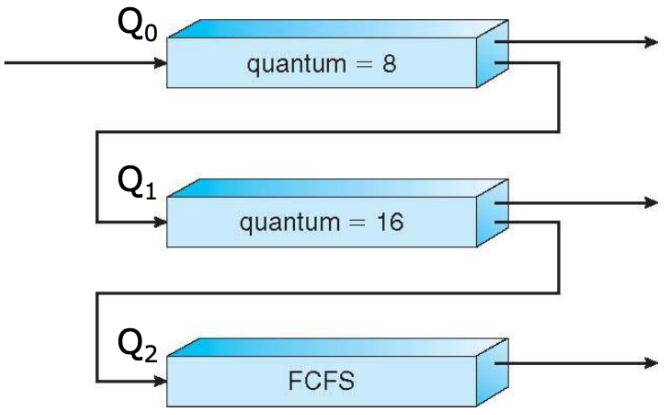
- 한 프로세스가 실행하고 있던 도중에 인터럽트가 발생, 비자발적으로 문맥 교환이 일어나 준비 큐로 돌아왔다고 하자. 인터럽트를 처리한 후 다시 스케줄링을 하려고 할 때 **기존 수행 중이었던 프로세스가 있는 큐보다 높은 단계의 큐에 새로운 프로세스가 하나라도 있다면 바로 그 프로세스에 CPU를 할당해주어야 한다.** 즉, 다단계 큐 스케줄링은 **선점형 스케줄링**의 일종이다.
- 각 큐는 CPU 시간의 일정량을 받아서 자기 큐에 있는 다양한 프로세스들을 스케줄 할 수 있다. 예를 들어, 포그라운드-백그라운드 큐 예에서 포그라운드 큐는 자신의 프로세스들 사이에서 라운드 로빈 스케줄링을 위해 CPU 시간의 80%가 주어지고, 백그라운드 큐는 CPU 시간의 20%를 받아 자신의 프로세스들에 선입 선처리 방식으로 줄 수 있다.

다단계 피드백 큐 스케줄링(Multilevel Feedback Queue Scheduling)

✔️ **다단계 큐 스케줄링 + 동적인 프로세스 우선순위 변화**

> 💬 다단계 큐 스케줄링에서는 프로세스들이 영구적으로 하나의 준비 큐에 할당되었다. 이와는 다르게, 다단계 피드백 큐 스케줄링에서는 **프로세스가 큐들 사이를 이동하는 것을 허용**한다.

- 프로세스들은 CPU 버스트 성격에 따라서 구분한다. 어떤 프로세스가 CPU 시간을 너무 많이 사용하면, 낮은 우선순위의 큐로 이동된다. 이 방법에서는 I/O 중심의 프로세스와 대화형 프로세스들을 높은 우선순위의 큐에 넣는다. 마찬가지로 낮은 우선순위의 큐에서 너무 오래 대기하는 프로세스는 높은 우선순위의 큐로 이동할 수 있다. 이러한 노화 형태는 기아 상태를 예방한다.



- 예를 들어, 번호가 0에서 2까지인 세 개의 큐를 가진 다단계 피드백 큐 스케줄러를 생각해 보자. 스케줄러는 처음에 큐 0의 모든 프로세스를 실행시킨다. 큐 0이 비어있을 때만 큐 1에 있는 프로세스들을 실행시킨다.마찬가지로 큐 0과 1이 비어있을 때만 큐 2에 있

는 프로세스들이 실행된다. 큐 1에 도착한 프로세스는 큐 2에 있는 프로세스를 선점한다. 큐 1에 있는 프로세스는 큐 0에 도착한 프로세스에 의해 선점될 것이다.

- 새로 진입하는 프로세스는 큐 0에 넣어진다. 큐 0에 있는 프로세스는 8 밀리초의 시간 할당량이 주어진다. 만약 프로세스가 이 시간 안에 끝나지 않는다면 큐 1의 꼬리로 이동된다. 큐 0이 비어 있다면, 큐 1의 머리에 있는 프로세스에 16밀리초의 시간 할당량이 주어진다. 이 프로세스가 완료되지 않는다면, 선점되어 큐 2에 넣어진다. 큐 0과 큐 1이 비어있을 때만 큐 2에 있는 프로세스들이 선입 선처리 방식으로 실행된다. 기아를 방지하기 위해 우선순위가 낮은 큐에서 너무 오래 대기하는 프로세스가 점차 우선순위가 높은 프로세스로 이동될 수 있다.
- 일반적으로, 다단계 피드백 큐 스케줄러는 다음의 매개변수에 의해 정의된다.
    - 큐의 개수
    - 각 큐를 위한 스케줄링 알고리즘
    - 한 프로세스를 높은 우선순위 큐로 올려주는 시기를 결정하는 방법
    - 한 프로세스를 낮은 우선순위 큐로 강등시키는 시기를 결정하는 방법
    - 프로세스에 서비스가 필요할 때 프로세스가 들어갈 큐를 결정하는 방법

스레드 스케줄링(Thread Scheduling)

1. 프로세스-경쟁 범위 (Process-Contention Scope, PCS)

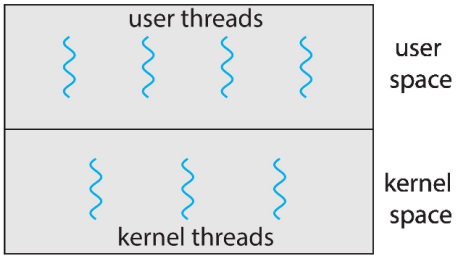
- **동일 프로세스에 속한 스레드들 사이에서** CPU를 할당받기 위한 경쟁을 말한다.
- 다대일 또는 다대다 모델을 구현하는 시스템에서 스레드 라이브러리는 **LWP 상에서 스케줄을 실행**한다.
- 스케줄러는 가장 높은 우선순위를 가진 실행 가능한 프로세스를 선택하며, 현재 실행 중인 스레드보다 더 높은 우선순위의 스레드가 나타난다면 해당 스레드는 선점된다.

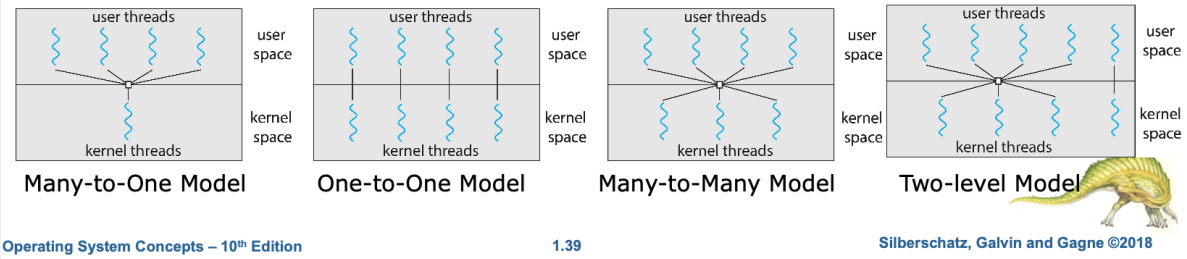
2. 시스템-경쟁 범위 (System-Contention Scope, SCS)

- **어느 커널 스레드에게** CPU를 할당해줄 것인지를 결정하는 것을 말한다.
- SCS 스케줄링에서의 CPU에 대한 경쟁은 **시스템 상의 모든 스레드 사이에서 일어난다.**
- 일대일 모델을 사용하는 시스템은 오직 SCS만을 사용하여 스케줄을 한다.

Thread Scheduling

- Distinction between **user-level** and **kernel-level threads**
- **User threads**
  - **management** done by **user-level threads library**
  - Three primary thread libraries:
    - **POSIX Pthreads**
    - **Windows threads**
    - **Java threads**
- **Kernel threads** - **Supported by the Kernel**





Thread Scheduling

- When threads supported, **threads scheduled**, **not processes scheduled by OS**
- **User-level Thread Scheduling**
  - **User-level threads** are **managed by a thread library**, and the **kernel is unaware of them**
  - Many-to-one and many-to-many models, **thread library schedules user-level threads to run on LWP**
  - Do **not** mean that the thread is **actually running on a CPU**
  - A scheme known as **process-contention scope (PCS)**, since **scheduling competition is within the process**
  - Typically done via **priority set by programmer** and **not adjusted by the thread library** - typically **preemptive scheduling**

Thread Scheduling

- **Kernel-level Thread Scheduling**
  - **Kernel thread scheduled onto available CPU is system-contention scope (SCS)** – competition **among all threads** in system
  - OSs using the one-to-one model, such as Linux, schedule threads using only SCS