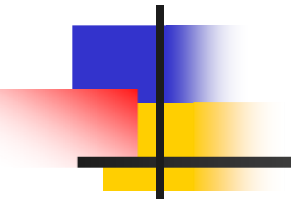
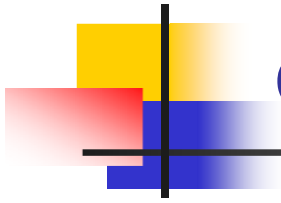


Operating Systems
(CPU Scheduling)

Chapter 5

These lecture materials are modified from the lecture notes written by A. Silberschatz, P. Galvin and G. Gagne.

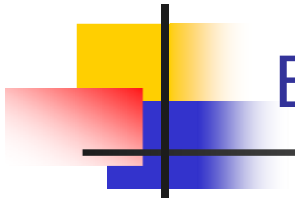
August, 2022



Outline

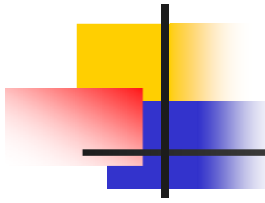
- Background
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Thread Scheduling

Background



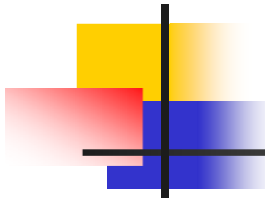
Background

- CPU는 주요 리소스 이므로, CPU 스케줄링은 OS의 핵심
- 태스크 간에 CPU를 전환함으로써 생산성을 높일 수 있음.
- *"프로세스 스케줄링"*과 *"스레드 스케줄링"*이라는 용어는 종종 같은 의미로 사용됨.
- 여기서는 일반적인 스케줄링 개념을 논의할 때 *"프로세스 스케줄링"* 용어를 사용



CPU Scheduler

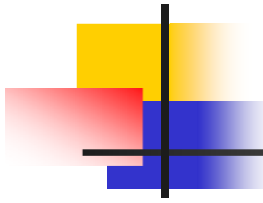
- CPU가 idle 상태가 될 때마다 OS는 ready queue에 있는 프로세스 중 하나를 선택해야 함.
- 큐의 record는 일반적으로 프로세스의 PCB임.
- 스케쥴러는 실행할 준비가 된 프로세스 중에서 실행할 것을 선택하고 해당 프로세스에 CPU를 할당함.



CPU Scheduler

- CPU 스케줄링 결정은 다음 네 가지 상황에서 이루어짐.

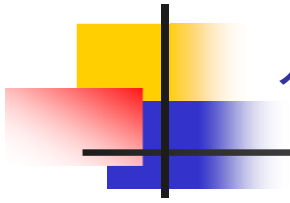
- 1. 프로세스가 running 상태에서 waiting 상태로 전환되는 경우
  (예: 자식 프로세스의 종료에 대한 I/O 요청 또는 wait() 호출의 결과)
- 2. 프로세스가 running 상태에서 ready 상태로 전환되는 경우
  (예: 인터럽트 발생 시)
- 3. 프로세스가 waiting 상태에서 ready 상태로 전환되는 경우
  (예: I/O 완료 시)
- 4. 프로세스가 종료되는 경우



CPU Scheduler

- 1과 4의 경우 반드시 ready queue의 새 프로세스를 선택해야 함. 그러나 2,3은 선택 할 수도, 아닐 수도 있음.

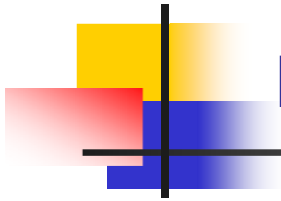
- 스케줄링이 상황 1과 4에서만 발생하는 경우, 스케줄링 방식은 비선점(Non-Preemptive)형이라고 함.
- 상황 2, 3에서도 발생하는 경우, 선점(Preemptive)형이라고 함.



선점(Preemptive)

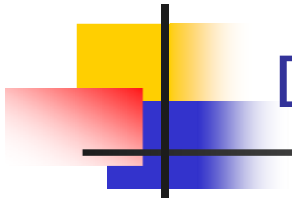
- 프로세스가 CPU에 할당되어 있을 때도 OS가 개입해 다른 프로세스에게 CPU를 할당하는 것

- 최신 OS는 선점형 스케줄링 알고리즘을 사용함.



Dispatcher

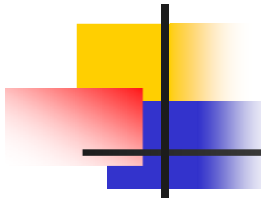
- CPU 스케줄링 기능과 관련된 또 다른 구성 요소
- 디스패처는 CPU 스케줄러가 선택한 프로세스에 CPU 코어의 제어를 제공하는 모듈
- 디스패처에 포함된 기능
  - 한 프로세스에서 다른 프로세스로 context switch
  - 유저 모드로 전환
  - 프로그램의 적절한 위치로 점프하여 해당 프로세스를 재개함
- 디스패처는 모든 context switch 중에 호출되기 때문에 가능한 한 빨라야함.
- 디스패처가 한 프로세스를 중지하고 다른 실행을 시작하는 데 걸리는 시간을 dispatch latency이라고 함.



Dispatcher

- 디스패처는 모든 context switch 중에 호출되기 때문에 가능한 한 빨라야함.
- 디스패처가 한 프로세스를 중지하고 다른 실행을 시작하는 데 걸리는 시간을 dispatch latency이라고 함.

Scheduling Criteria



Scheduling Criteria

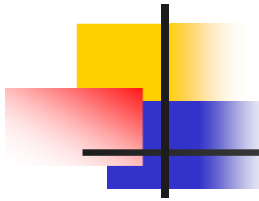
- CPU 스케줄링 알고리즘마다 속성이 다름.
- 특정 상황에서 사용할 알고리즘을 선택할 때 다양한 알고리즘의 속성을 고려해야함.
- 상황에 따라 어떤 기준으로 알고리즘의 성능을 평가할 것인가는 매우 중요함.

스케쥴링 알고리즘의 평가 기준

- **CPU utilization:** 사용자는 CPU를 가능한 한 바쁘게 유지하기를 원함.
  CPU 사용률은 0%에서 100% 사이이며, 실제 시스템에서는 40%에서 90%범위여야 함.

- **Throughput:** 시간 단위당 실행이 완료된 프로세스의 수.

- **Turnaround time:** 프로세스의 submission 시간부터 completion 시간까지의 간격.
  ready queue에서 대기하고 CPU에서 실행하고 I/O를 수행하는 데 소요된 시간의 합계.

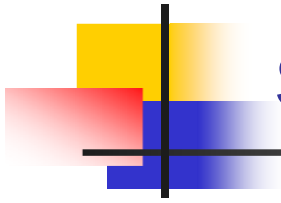


스케줄링 알고리즘의 평가 기준

- **Waiting time:** Ready queue에서 대기한 기간의 합계.
  - CPU 스케줄링 알고리즘은 프로세스 실행, I/O 수행에 걸리는 시간에 영향을 미치지 않음

- **Response time:** 사용자의 request submission부터, 프로세스의 첫 번째 response가 생성될 때까지의 시간.

Scheduling Algorithms



Scheduling Algorithms

- CPU 스케줄링은 Ready queue에 있는 프로세스 중 CPU의 코어를 할당할 프로세스를 결정하는 문제를 다룸.
- 여기서는 대부분의 최신 CPU 아키텍처에는 여러 개의 core가 있지만, 여기서는 사용 가능한 core가 하나일 때를 기준으로 설명함.
- 즉, 예시들은 single core인 single CPU 환경이므로, 한 번에 하나의 프로세스만 실행 가능함.

First-Come, First-Served Scheduling(FCFS)

- 먼저 들어온 프로세스를 먼저 프로세서에 할당하는 방식
- 먼저 온 프로세스가 끝날 때까지 운영체제가 개입하지 않는 비선점 스케줄링.

- **장점**
  - FIFO(First-In First-Out) Queue로 손쉽게 관리됨.
  - 구현이 쉬워서 간단한 시스템에 자주 사용됨.
- **단점**
  - 프로세스 처리 순서에 따라 성능이 크게 달라질 수 있음.
  - Convoy effect가 발생
    - Convoy effect: 수행 시간이 큰 프로세스가 먼저 들어오면, 그 뒤에 들어온 프로세스들이 불필요하게 오랜 시간을 기다리게 되는 효과

Scheduling Algorithms

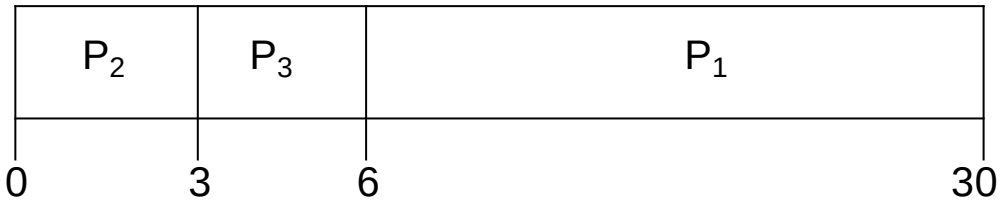
- First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

  - Suppose that the processes request the CPU in the order: P_1 , P_2 , P_3
    The Gantt Chart for the schedule is:

  - Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
  - Average waiting time: (0 + 24 + 27)/3 = 17

- First-Come, First-Served (FCFS) Scheduling (cont'd)
  - Process arrival order
    - P_2 , P_3 , P_1
  - The Gantt chart for the schedule is:



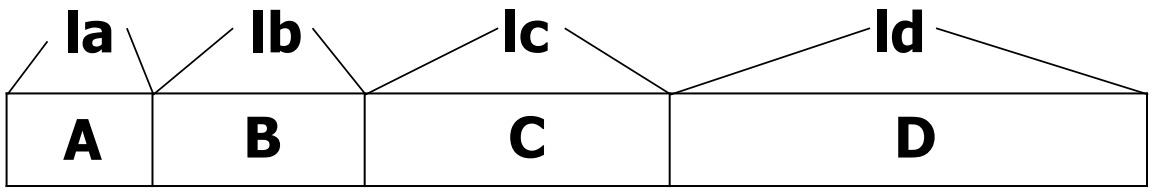
  - Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
  - Average waiting time: (6 + 0 + 3)/3 = 3
  - Much better than previous case
  - *Convoy effect*
    - short process behind long process
    - e.g. 1 cpu-bound process, many I/O bound processes
    - This increases average waiting time

Shortest-Job-First Scheduling(SJF)

- 프로세스의 수행 시간이 짧은 순서에 따라 프로세서에 할당함.
- 비선점, 선점 둘 다 가능함.
  - SRTF(Shortest-Remaining-Time-First) : 선점형 SJF. 현재 실행 중인 프로세스의 남은 시간보다 짧은 새 프로세스가 도착하면 선점함.

- **장점**
  - FCFS에서 발생하는 Convoy effect를 해결할 수 있음
  - 최적 알고리즘.
- **단점**
  - 프로세스의 수행 시간을 정확히 알 수 없음. 앞서 처리한 해당 프로그램 기반의 프로세스들의 기록을 보고 추측해야 함
  - 실행 시간이 큰 프로세스가 계속 뒤로 밀려나는 Starvation(기아)이 발생.

- Why is it optimal?



Average waiting time

A: 0 B: la C: la+lb D: la+lb+lc **(3la+2lb+lc)/4**

Average turnaround time

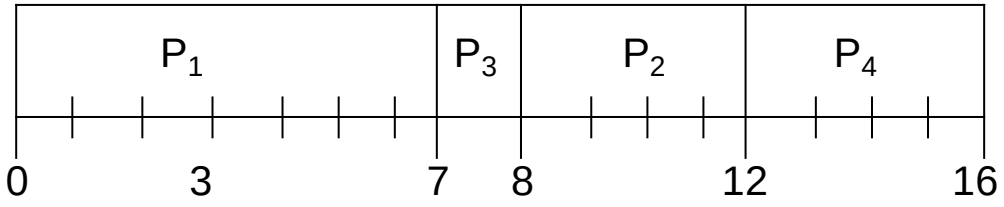
A: la B: la+lb C: la+lb+lc D: la+lb+lc+ld

(4la+3lb+2lc+ld)/4

- SJF (non-preemptive)

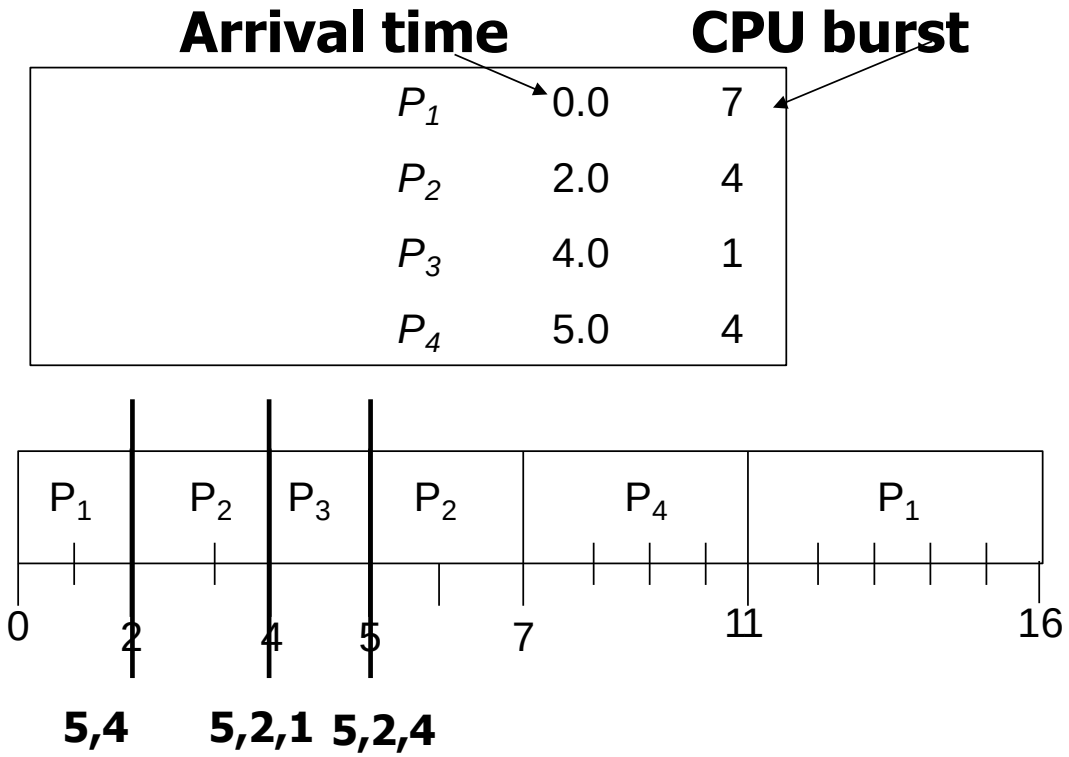
	Arrival time	CPU burst
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)



Average waiting time = (0 + 6 + 3 + 7)/4 = 4

- SJF (preemptive)



Average waiting time = (0 + 6 + 3 + 7)/4 = 4

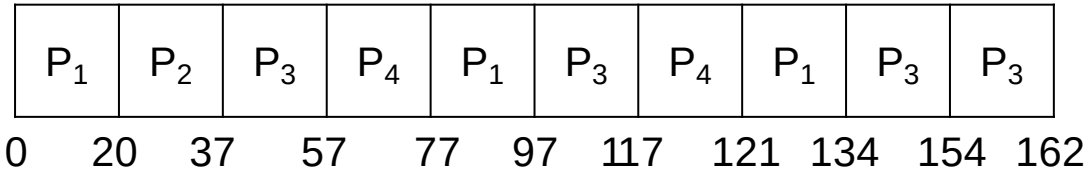
Round-Robin Scheduling(RR)

- 각 프로세스마다 순서대로 일정 시간 할당량(Time quantum)만큼 CPU에 할당.
- Time quantum에 따라 운영체제가 계속 개입하는 선점 스케줄링 방식

- **장점**
  - ready queue에 n개의 프로세스가 있고 Time quantum이 q일 경우, (n-1)q 이상 대기하는 프로세스는 없음.
- **단점**
  - q가 크면 FIFO와 같아짐.
  - q가 너무 작으면 context switching이 너무 늘어나서 오버헤드가 커짐.

- Round-robin scheduling

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24



Ready queue

1-2-3-4 2-3-4-1 3-4-1

Typically, higher average turnaround than SJF, but better *response*

- Context-switch overhead

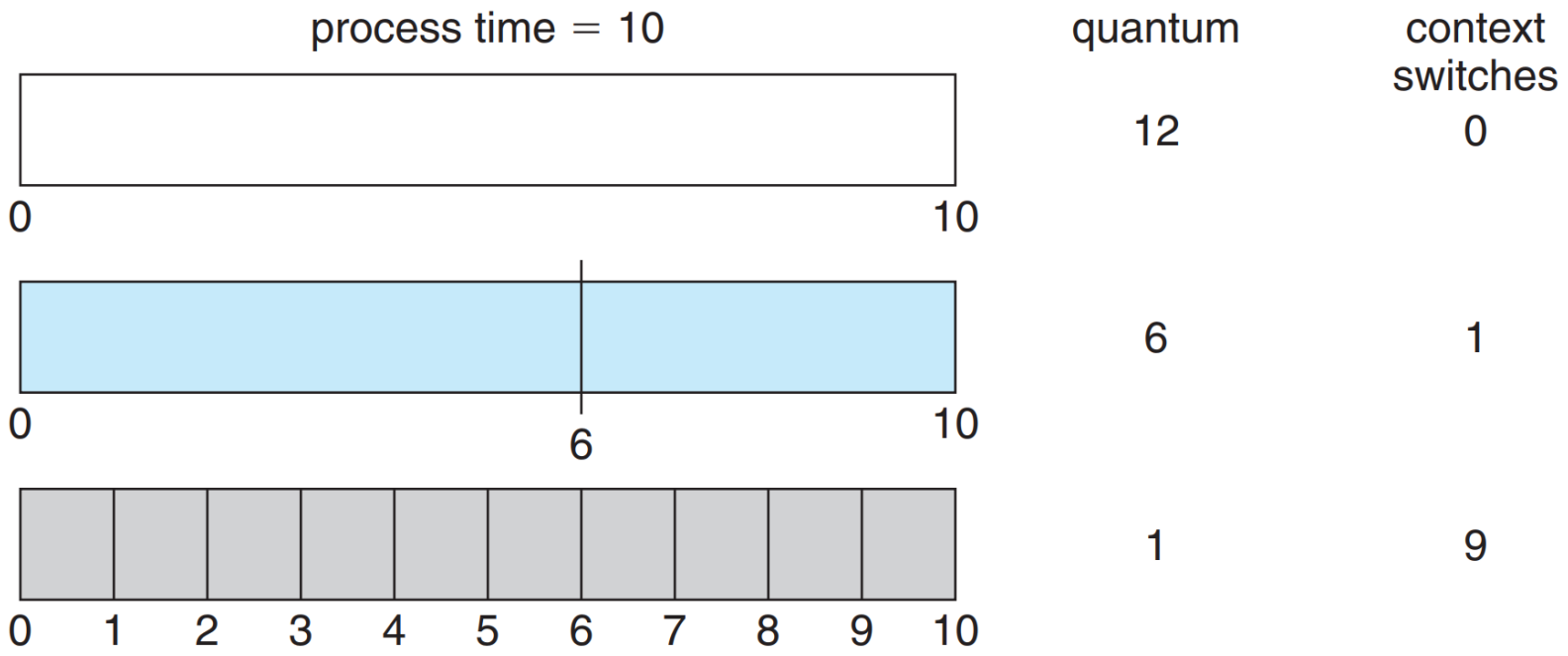
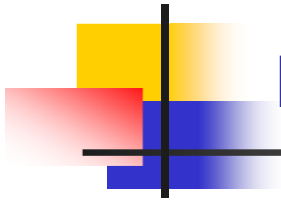


Figure 5.5 How a smaller time quantum increases context switches.



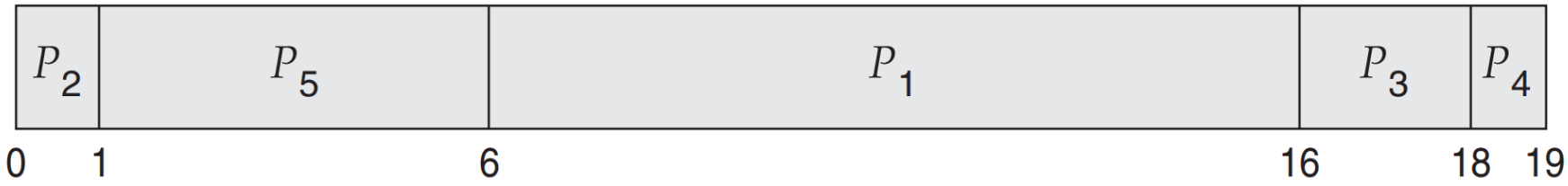
Priority Scheduling

- 먼저 프로세스의 우선 순위(Priority)를 계산하고 우선순위가 높은 프로세스를 선택함.
  - SJF 알고리즘은 Priority Scheduling에서 우선순위가 프로세스의 실행 시간인 경우.
- 선점형일수도, 비선점형일 수 있음.

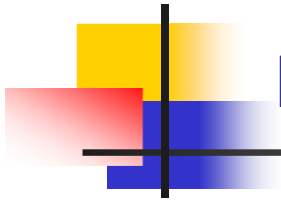
- **Static priority:** 한번 결정한 우선 순위를 끝까지 유지하는 것.
- **Dynamic priority:** 우선순위를 지속적으로 변경하는 것.
  - 우선 순위가 낮은 프로세스가 무한 대기하게 되는 것을 막기 위해 aging을 사용.
  - aging: 시스템에서 오랫동안 대기하는 프로세스의 우선 순위를 점진적으로 높이는 것

- Priority scheduling

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



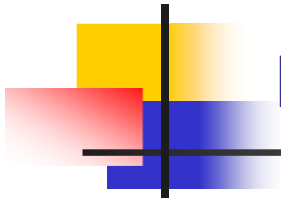
The average waiting time is 8.2 milliseconds.



Multilevel Queue Scheduling

- 각 프로세스의 우선순위에 따라 별도의 queue를 가지는 방식.
- 각 queue는 자신만의 스케쥴링 기법을 따로 가짐.

- **장점**
  - 프로세스들의 특성을 잘 살릴 수 있음.
- **단점**
  - 프로세스들의 priority가 고정되어, 한번 들어간 queue에서 다른 queue로 옮겨지지 않음.
  - 스케줄링 오버헤드는 낮지만 융통성이 없음.



Multilevel Queue Scheduling

- 구현 방식은 여러 방법이 있으며, 보통 선점 방식

- **Fixed priority scheduling:** 우선순위가 높은 큐가 빌 때만 우선순위가 낮은 큐의 프로세스가 선택됨. starvation(기아)가 발생 가능.
- **Time slice:** 각 queue마다 시간 사용 비율을 지정함. 우선순위가 높을 수록 시간 사용 비율이 높음.

- Multilevel Queue Scheduling

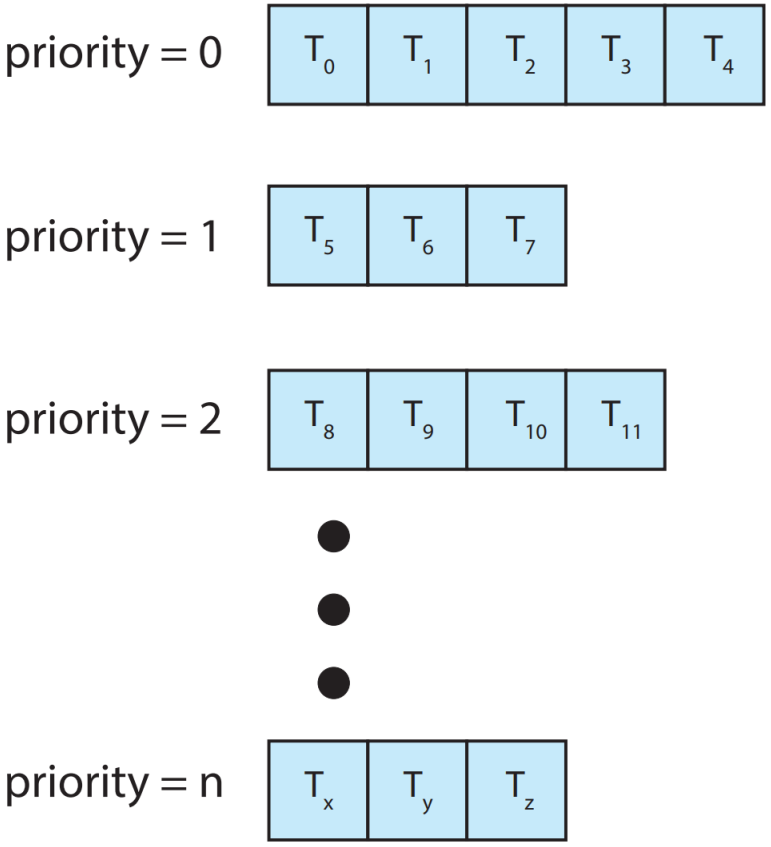


Figure 5.7 Separate queues for each priority.

- Multilevel Queue Scheduling

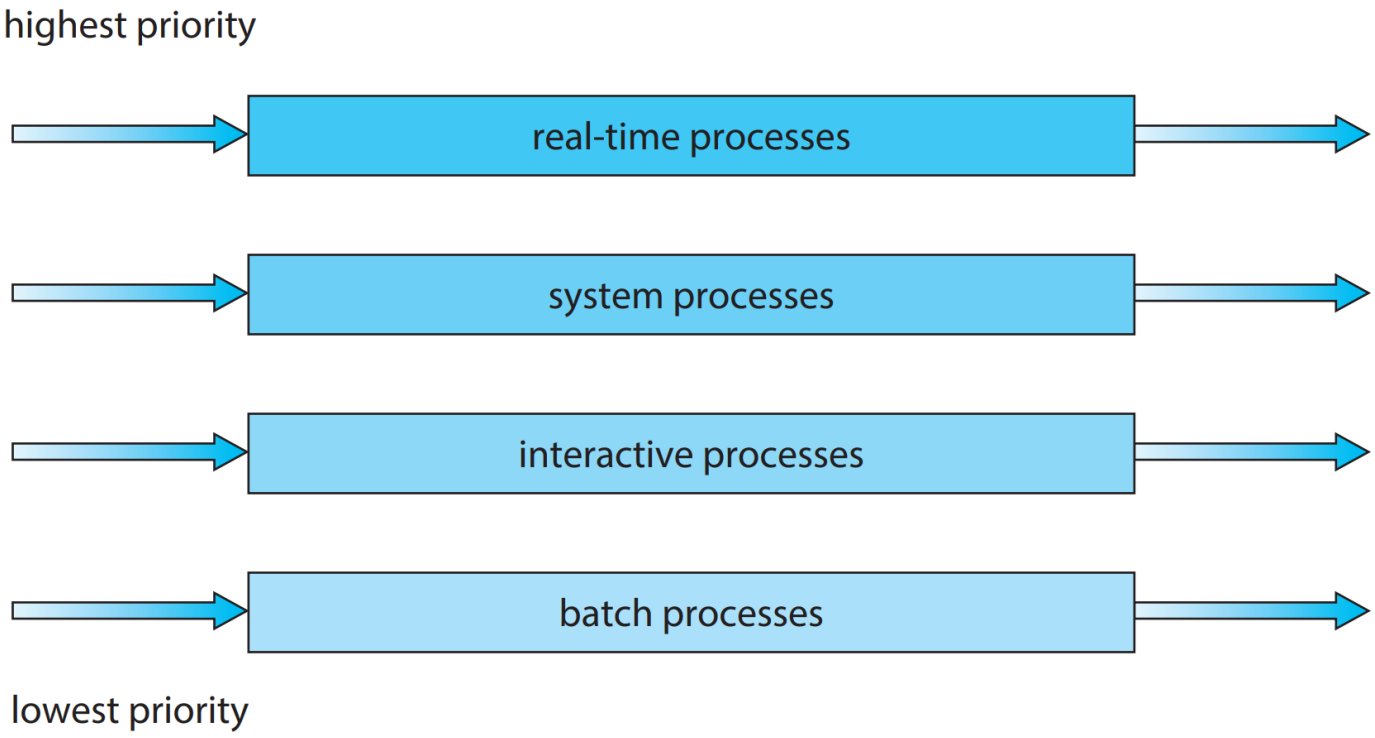
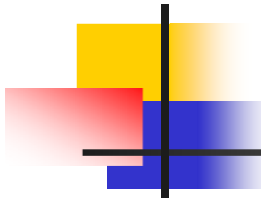


Figure 5.8 Multilevel queue scheduling.



Multilevel Feedback Queue Scheduling

- Multilevel Queue Scheduling의 단점을 해결하기 위해, 각 프로세스가 queue 사이를 이동할 수 있도록 함.
- 프로세스가 CPU 시간을 너무 많이 사용하면 우선 순위가 낮은 큐로 이동함.
  - 일반적으로 짧은 CPU 사용시간을 특징으로 하는 I/O bound 및 interactive 프로세스를 더 높은 우선 순위 큐에 남김.
  - 낮은 우선 순위 큐에서 너무 오래 대기하는 프로세스는 높은 우선순위 큐로 이동되는 aging을 사용하여 starvation을 예방.

Multilevel Feedback Queue Scheduling

- 다음 parameter들로 정의
  - queue 수
  - 각 queue에 대한 스케줄링 알고리즘
  - 프로세스를 우선 순위가 더 높은 queue로 업그레이드하는 조건
  - 프로세스를 우선 순위가 더 낮은 대기열로 내리는 조건
  - 해당 프로세스에 서비스가 필요할 때 프로세스가 들어갈 queue을 결정하는 조건

- 가장 일반적인 CPU 스케줄링 알고리즘
- 가장 복잡한 알고리즘
  - 최상의 스케줄러를 정의하려면 모든 parameter에 대한 값을 선택해야 하기 때문임.

- Multilevel Feedback Queue Scheduling

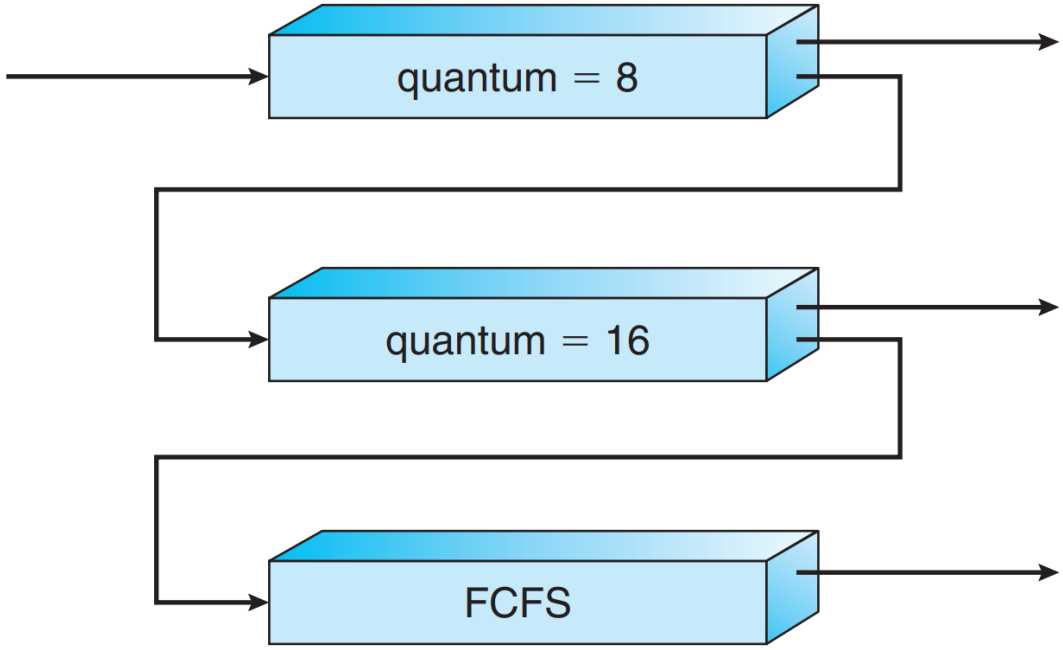
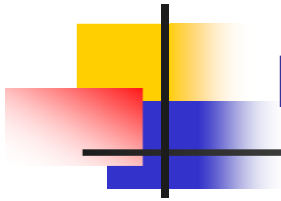


Figure 5.9 Multilevel feedback queues.

Multiple-Processor Scheduling

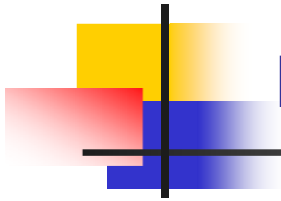


Multiple-Processor Scheduling

- 여러 CPU를 사용할 수 있는 경우 CPU 스케줄링이 복잡함.

- **SMP(Symmetric Multiprocessing) :**
  각 프로세서가 자체적으로 스케줄링 결정을 내림.

- **ASMP(Asymmetric multiprocessing) :**
  하나의 프로세서만 시스템 데이터 구조에 액세스하여 데이터 공유의 필요성을 줄임.
  - Master-slave relationship



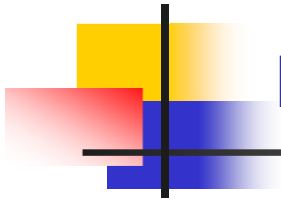
Multiple-Processor Scheduling

- SMP를 사용할 경우의 CPU 스케줄링 문제를 해결하는 방법
  - 1. Process affinity
  - 2. Load balancing

Process affinity

- 각 프로세스는 현재 실행 중인 프로세서에 대한 선호도를 가짐

- **Soft affinity:** 동일한 프로세서에서 프로세스를 계속 실행하려고 시도하는 정책이 있지만 그렇게 할 것이라고 보장하지는 않음.
- **hard affinity:** OS는 프로세스가 다른 프로세서로 migration 되지 않도록 지정하는 system call을 제공함.



Load balancing

- SMP 시스템에서 CPU 효율량을 높이려면, 한 CPU에 프로세스가 몰려 있으면 안되고, 균등하게 배분되어 있어야 함.
- 각 프로세서마다 각자의 ready queue를 갖고 있을때 필요한 기법.
  - 공통으로 갖고 있다면 이 문제는 저절로 해결

- **Push migration**
  - 1. 특정 작업이 주기적으로 프로세서의 부하를 확인
  - 2. 부하 불균형이 발견되면 프로세스를 이동하여 부하를 고르게 분산
- **Pull migration**
  - idle 프로세서가 사용 중인 프로세서에서 대기 중인 작업을 가져옴

Thread Scheduling

Thread Scheduling

- 사실 현대의 OS는 프로세스를 스케쥴링 하는 것이 아님.
- 프로세스의 유저 스레드를 커널 스레드에 매핑한 후, 이것을 스케쥴링 하는 형태임.

- **모든 유저 스레드는 궁극적으로 커널 스레드에 매핑되어야함.**
- 유저 스레드들을 lightweight process (LWP)에 매핑하고, LWP를 커널 스레드에 매핑할 수도 있음.

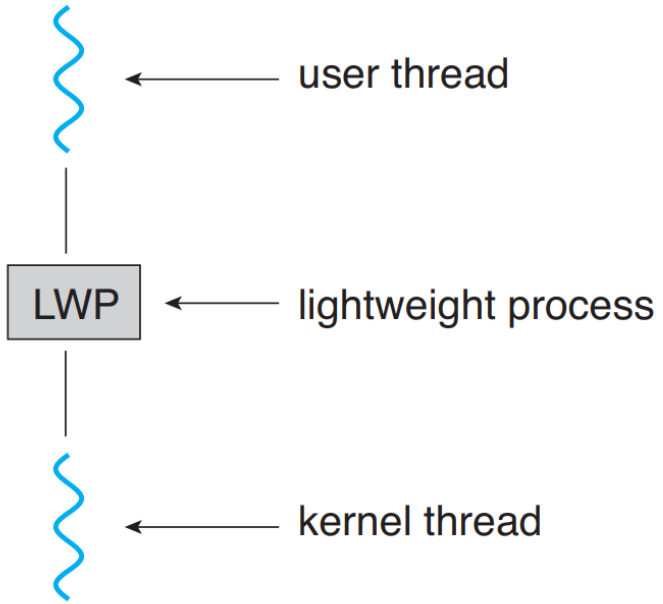
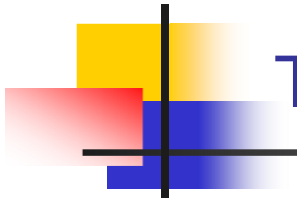


Figure 4.20 Lightweight process (LWP).



Thread Scheduling

- **Local Scheduling**
  - 스레드 라이브러리가 LWP에 매핑할 유저 스레드를 결정함.
  - 동일한 프로세스 내의 스레드 끼리 CPU를 경쟁함.
  - Process-contention scope

- **Global Scheduling**
  - 커널이 다음에 실행할 커널 스레드를 결정하는 방법
  - System-contention scope

Chapter 5
Finish