5. CPU Scheduling

▼ 목차

1. CPU 스케줄링

여러 개의 프로그램이 실행될 때 다음과 같은 이슈가 발생

1. 여러 Process가 CPU를 점유하려고 할 때, 누구에게 줄 것인가?
2. 하나의 Process에게 얼마나 오랫동안 CPU를 할당할 것인가?

이러한 이슈들을 해결하는 작업이 바로 CPU 스케줄링이다.
즉, 어떤 프로세스에게 얼마 동안 CPU를 할당할지 결정하는 작업이라고 할 수 있다.

---

2. CPU burst & I/O burst

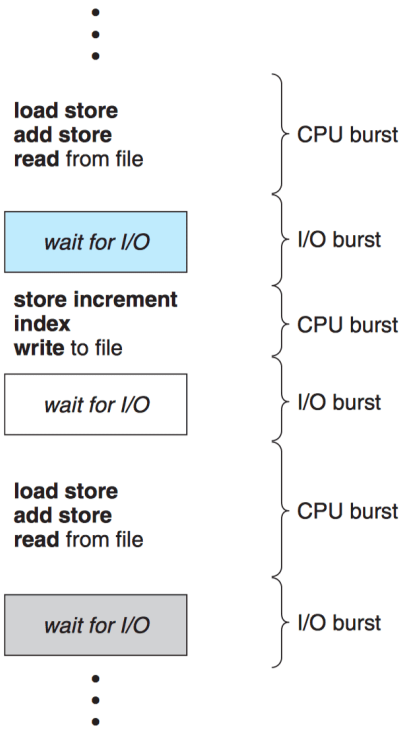
하나의 프로세스가 실행될 때 CPU burst와 I/O burst가 교대로 실행

CPU burst

CPU가 프로세스를 실행하는 동안 소요되는 시간

I/O burst

입출력 장치와의 상호 작용으로 인해 발생하는 시간
프로그램 실행중에 I/O작업이 끝날때까지 block되는 구간이다.



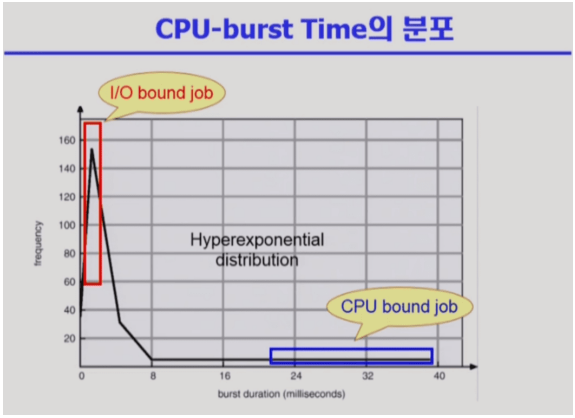


Figure 6.1 Alternating sequence of CPU and I/O bursts.

CPU burst / I/O burst의 빈도와 길이에 따라 프로세스를 다음의 두 가지로 나눔

I/O bound process

- CPU를 잡고 계산하는 시간보다 I/O에 많이 시간이 필요한 프로세스
- 사용자와의 인터랙션이 많아서 I/O burst가 빈번하게 발생하는 경우가 여기에 해당
- **many short CPU bursts**

CPU bound process

- 계산 위주의 프로세스
- CPU를 진득하게 쓰는 프로그램
- **few very long CPU bursts**

CPU Scheduler와 Dispatcher는 운영체제 커널 코드의 한 부분

- **CPU Scheduler**
  - ready 상태의 프로세스 중 CPU를 할당할 프로세스를 선택한다.

- **Dispatcher**
  - CPU Scheduler에 의해 선택된 프로세스에게 CPU를 할당한다.
  - 이 과정을 Context switch(문맥 교환)이라고 한다.

`nonpreemptive`(비선점형) — 자진 반납

`preemptive`(선점형) — 강제로 빼앗음

---

3. CPU 스케줄링 성능 척도(Scheduling Criteria)

스케줄링 알고리즘을 평가하는 성능 척도

3-1. 시스템 입장에서의 성능 척도 (주인 입장)

하나의 CPU를 가지고 최대한 많은 일을 진행
→ 제한된 시간 내에 최대한 많은 프로세스를 처리 ('작업량'이 관건)

- **CPU utilization**(이용률)
  전체 시간 중에서 CPU가 프로세스들을 실행한 시간의 비율 (0 ~ 100%)
  ⇒ 고용한 직원이 놀지 않고 일을 하는 시간

- **Throughput**(처리량)

  단위 시간당 완료한 프로세스들의 개수
  네트워크에서는 단위 시간당 전송한 양을 의미 (e.g. Mbps, Kbps)
  ⇒ 얼마나 많은 손님들을 뺄 수 있는지

3-2. 프로세스 입장에서의 성능 척도 (고객 입장

CPU를 최대한 빨리 얻어서 처리
→ CPU를 최대한 빨리 얻어서 빨리 실행 ('시간'이 관건)

- **Turnaround time**(소요시간, 반환시간)
  특정 프로세스에 대해 실행을 요청한 시점부터 완전히 종료된 시점까지의 소요 시간
  (실제 실행 시간 외에 ready queue에서의 대기시간과 device queue에서의 대기시간도 포함)
  ⇒ 주문과 식사 그리고 퇴실까지 걸리는 시간

- **Waiting time**(대기시간)
  프로세스가 ready queue에서 기다린 시간의 총 합
  ⇒ 식사 시간을 제외한 기다리는 시간

- **Response time**(응답시간)
  실행이 요청된 시점부터 최초로 CPU를 얻기까지 소요된 시간
  ⇒ 주문을하고 첫번째 음식이 나올 때까지 기다리는 시간

---

4. 스케줄링 알고리즘

4-1. FCFS (first come fitst out)

들어온 순서에 따라 프로세스를 실행

- nonpreemptive (비선점형)
  일단 CPU를 얻으면 자진 반납하기 전까지 빼앗기지 않음
- 프로세스를 ready queue에 들어온 순서대로 실행
  FCFS 알고리즘의 평가 기준은 평균적인 waiting time이고, burst time과 ready queue에 추가된 순서를 고려하여 계산한다.

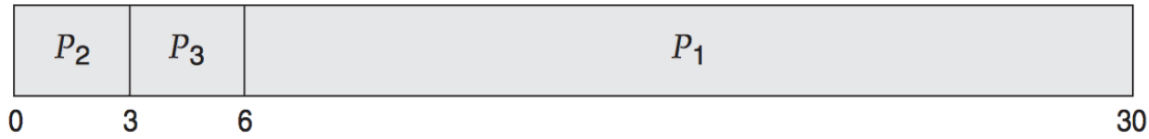
Process	Burst Time
P1	24
P2	3
P3	3

- Order: P1 → P2 → P3
- Gantt chart:

- Average waiting time = (0 + 24 + 27) / 3 = 17ms

→ Average waiting time = 17

- Order: P2 → P3 → P1
- Gantt chart:



- Average waiting time = (0 + 3 + 6) / 3 = 9ms

→ Average waiting time = 9

이처럼 먼저 들어오는 프로세스의 처리 속도에 따라 average waiting time이 달라짐

`convoy effect`
burst time이 긴 프로세스가 ready queue에 먼저 들어오게 되면 비교적 실행 시간이 짧은 프로세스들이 지나치게 오래 기다려야 한다

→ 이러한 문제점을 해결하려면 burst time이 짧은 프로세스를 먼저 처리하면 된다.

4-2. SJF (Shortest-Job-First)

실행 시간이 가장 짧은 프로세스를 먼저 처리

- nonpreemptive (비선점형) & preemptive (선점형) 두 가지 방식으로 구현할 수 있다.
- burst time이 짧은 프로세스를 먼저 처리하기 때문에
  평균적인 waiting time을 평가 기준으로 했을 때 가장 최적의 성능을 보장 (preemptive 방식일때)

---

nonpreemptive (비선점형)

하나의 프로세스가 CPU를 잡으면 실행이 완료될 때까지 CPU를 빼앗기지 않는다.
(하나의 프로세스가 완료될 때 마다 burst time을 비교하여 실행할 프로세스를 선정)

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

- Average waiting time = (0 + (8 - 1) + (8 + 4 + 5 - 2) + (8 + 4 - 3)) / 4 = 7.75ms

가장 먼저 도착한 P1부터 실행한다. P1의 실행이 완료되면 (P2, P3, P4가 모두 도착해있는 상태) burst time이 짧은 P2 → P4 → P3 순으로 실행한다.

Average waiting time = (0 + (8 - 1) + (8 + 4 + 5 - 2) + (8 + 4 - 3)) / 4 = 7.75ms

가장 먼저 도착한 P1부터 실행
P1의 실행이 완료되면 (P2, P3, P4가 모두 도착해있는 상태)
burst time이 짧은 P2 → P4 → P3 순으로 실행

CPU 스케줄링은 하나의 프로세스가 온전히 완료 되고 나서 다시 진행

---

preemptive (선점형)

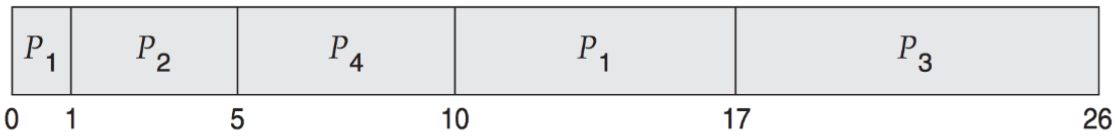
현재 수행중인 프로세스의 남은 burst time보다 더 짧은 burst time을 가지는 새로운 프로세스가 도착하면 CPU 제어권을 빼앗아서 새로운 프로세스에게 할당
→ ready queue에 새로운 프로세스가 도착할때마다 burst time을 비교하여 실행할 프로세스를 선정

이 방법을 SRTF (Shortest-Remaining-Time-First) 라고도 한다.

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

> ※ Arrival Time: ready queue에 들어온 시점

- Gantt chart:



- Average waiting time = [(10 - 1) + 0 + (17 - 2) + (5 - 3)] / 4 = 26 / 4 = 6.5 ms

preemtive 방식에서 계산된 average waiting time은 최적화된 값이므로, 위와 같은 예제에서는 6.5ms 보다 짧은 average waiting time을 얻을 수 없다.

Average waiting time = [(10 - 1) + 0 + (17 - 2) + (5 - 3)] / 4 = 26 / 4 = 6.5 ms

preemtive 방식에서 계산된 average waiting time은 최적화된 값이므로, 위와 같은 예제에서는 6.5ms 보다 짧은 average waiting time을 얻을 수 없다.

CPU 스케줄링은 하나의 프로세스가 모두 실행되지 않았더라도 남은 수행 시간 보다 더 짧은 프로세스가 들어온다면 순서 바꿈

SJF 의 두 가지 문제

1. Starvation (Indefinite Blocking)

SRTF(preemtive) 방식의 경우 계속해서 실행 시간이 짧은 프로세스들이 들어올 경우 우선순위가 낮은 프로세스(long burst time)는 마지막까지 실행되지 않을 수도 있다.

해결방법

Aging 기법

사람이 나이를 먹는 것처럼 프로세스도 시간이 지나면 priority가 1씩 높아지게 된다. 따라서 아무리 낮은 priority를 가지고 있는 프로세스도 언젠가는 높은 priority를 갖게 되기 때문에 CPU에 의해 실행될 수 있게 된다.

---

2. CPU burst time을 미리 알 수 없다.

해결방법

exponential average 사용

과거 burst time 패턴을 통해 '추정'할 수 있다.

exponential average

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n, \quad 0 \le \alpha \le 1.$$
$$= \alpha t_n + (1-\alpha)\alpha t_{n-1} + \cdots + (1-\alpha)^j \alpha t_{n-j} + \cdots + (1-\alpha)^{n+1}\tau_0.$$

==========

4-3. Priority Scheduling

우선순위가 제일 높은 프로세스에게 CPU를 준다

- 프로세스마다 적절한 우선순위를 부여하여 우선순위가 높은 프로세스부터 실행하는 알고리즘
  프로세스별로 **priority number** (integer)를 할당하여 우선순위를 나타낼 수 있으며 일반적으로 priority number가 '작을수록' 높은 우선순위를 뜻한다.
- 우선순위는 프로세스 생성 시에 사용자에 의해 지정되기도 하고
  운영체제에서 내부적으로 메모리 용량이나 실행 시간 등을 기준으로 하여 결정하기도 한다.
- SJF, SRTF는 Priority Scheduling의 일종이다
  예를 들어 SJF는 우선순위를 CPU 실행 시간으로 둔 것
- nonpreemtive & preemtive 두 가지 방식으로 구현할 수 있다.

→ 이 알고리즘 역시 우선순위가 높은 프로세스가 계속해서 ready queue에 들어올 경우 우선순위가 낮은 프로세스는 영원히 실행되지 않을 수 있는 문제가 있다 Starving 문제

→ 이러한 문제를 해결하기 위해 Aging 이라는 기법을 사용한다
아무리 우선순위가 낮더라도 오래 기다렸다고 하면 우선순위를 조금씩 높여주는 것

4-4. Round-Robin (RR)

현대적인 컴퓨터 시스템에서 사용하는 알고리즘
ready queue의 프로세스들을 일정한 할당 시간 단위(quantum)로 돌아가면서 실행

- preemtive
- ready queue의 프로세스들을 일정한 할당 시간 단위(quantum)로 돌아가면서 실행한다. 선택된 프로세스는 실행을 위해 할당된 시간을 가지며 이 시간이 만료되면 CPU를 빼앗기고 강제로 ready queue에 삽입된다.
- CPU burst time과 대기시간이 비례한다는 특징이 있다. CPU 작업량이 많을 수록 더 빈번하게 기다려야 하기 때문.

- 할당 시간 단위(q)가 길어질수록 FCFS에 가까워진다.
- 할당 시간 단위(q)가 짧아질수록 Response Time이 짧아지는 대신 빈번한 문맥 교환으로 인한 오버헤드가 늘어난다.

Round-Robin (RR)의 장점

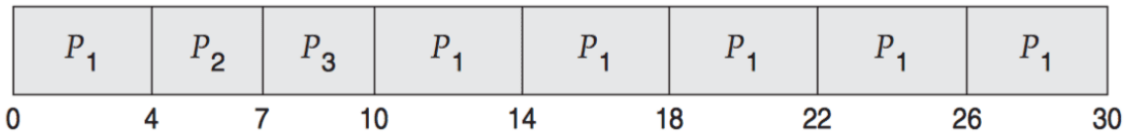
1. 할당 시간 만큼 CPU를 줬다가 뺐는 작업을 반복하기 때문에 **응답시간이 짧아지는 효과**
   짧은 계산 작업과 입출력 작업이 반복적으로 발생하는 대화형 프로세스에 적합
2. CPU burst time을 예측할 필요가 없고 **모든 프로세스가 CPU를 얻을 수 있다**.

   어떤 프로세스도 할당 시간 단위(q) x (프로세스 개수(n) - 1)만큼의 시간 이상 기다릴 필요가 없다. 이 시간 내에 반드시 실행할 기회가 주어진다.

   → 다양한 burst time을 가지는 프로세스들이 뒤섞여 있을 때 효과적
   (burst time이 길어도 CPU를 얻을 수 있기 때문에)

Process	Burst Time
P1	24
P2	3
P3	3

- Gantt chart:



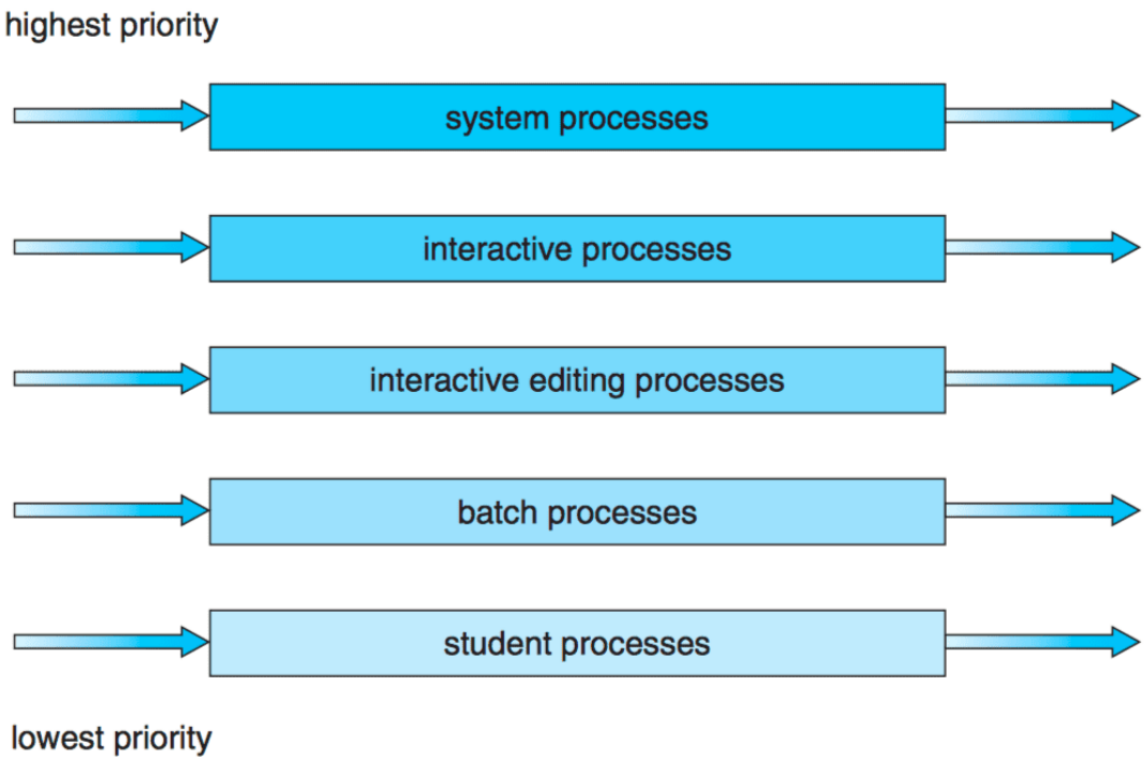
Average waiting time = [(10 - 4) + 4 + 7] / 3 = 5.66 ms

- 일반적으로 SJF보다 Turnaround time, Waiting time은 길지만 Response Time은 더 짧다.
- 모든 프로세스가 동일한 실행시간을 가지는 경우
  → FCFS: 먼저 도착한 프로세스를 먼저 처리하여 종료시킬 수 있지만
  → Round-Robin: 모든 프로세스를 일정 단위로 잘게 쪼개서 번갈아가며 처리하기 때문에 Turnaround time, Waiting time이 길어지고 모든 프로세스가 마지막까지 메모리에 남아있다가 거의 동시다발적으로 종료된다.

---

5. Multilevel Queue

ready queue를 여러 개로 분할 하는 것



- ready queue를 프로세스들의 특성에 따라 여러 레벨로 분리
  - 높은 우선순위 queue에 프로세스가 없다면 그 다음 레벨의 queue에 있는 프로세스를 실행
  - 각 레벨 별로 별도의 스케줄링 알고리즘을 적용할 수도 있음

  foreground(대화형) _ (interavtive)
  background _ (batch-no human interaction) _ 사람과 상호작용 없이 CPU만 오래 동안 쓰는 것
  이러한 작업은 주로 컴퓨터 시스템이나 소프트웨어 환경에서 자동으로 실행되는 작업을 지칭

- 각 queue는 각자에 맞는 독립적인 스케줄링 알고리즘을 가짐
  - foreground(대화형) - RR
    사람과 interaction 하기 때문에 RR 사용
  - background - FCFS
    어차피 CPU만 오래 사용하는 것이기 때문에 Context switching overhead 줄이기 위해 FCFS 사용

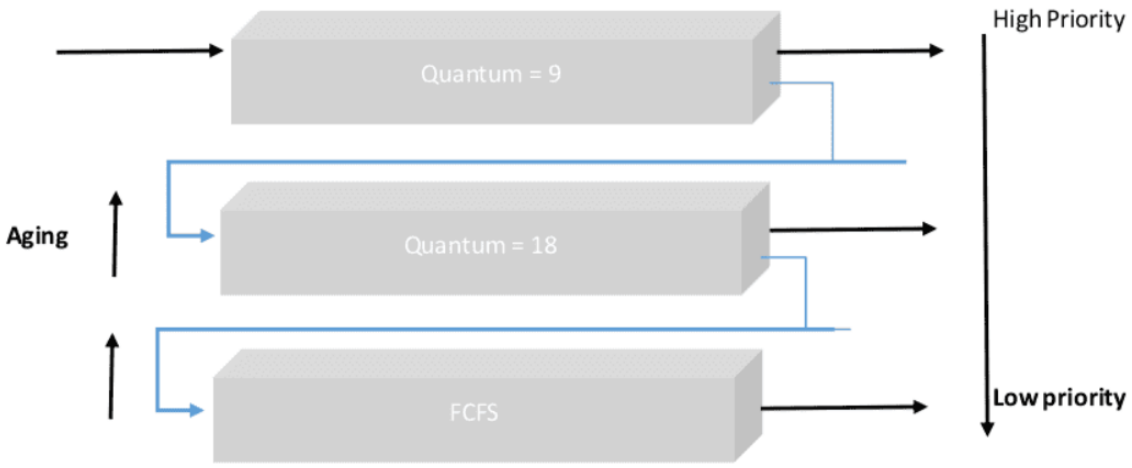
- queue에 대한 스케줄링이 필요
  - **Fixed priority scheduling**
    높은 레벨의 queue를 절대적으로 먼저 처리하는 스케줄링
    즉, 높은 레벨의 queue가 비었을 때만 낮은 레벨의 queue에 있는 프로세스를 실행함 → 낮은 레벨에 있는 queue는 영원히 실행되지 않을 수 있음 (Starvation 문제)
  - **Time slice**
    각 queue에 CPU 시간을 적절한 비율로 할당
    (ex CPU의 80%를 높은 레벨의 queue에 할당하고 나머지 20%는 낮은 레벨의 queue에 할당)

Multilevel Queue는 각 큐 별로 우선순위를 극복할 수 없는 문제가 있기 때문에 Multilevel Feedback Queue라는 스케줄링 방법도 있다

---

6. Multilevel Feedback Queue

프로세스들이 상황에 따라 queue와 queue 사이를 이동할 수 있는 스케줄링 알고리즘



- Multilevel Feedback Queue의 일반적인 운영방식
  - queue를 몇 개 둘 것인가
  - 프로세스가 queue로 들어갈 때 어떤 queue로 들어갈 것인가
    가장 먼저 도착한 프로세스를 가장 상위 queue로 배치
  - 각 queue별로 어떤 스케줄링 알고리즘을 적용할 것인가

    - 상위 queue로 갈수록 짧은 quantum 시간을 주고 RR 적용
    - 해당 큐에서 완료되지 못한 프로세스는 아래로 강등

- 가장 하위의 queue는 FCFS

- 프로세스를 상위(또는 하위) queue로 보내는 기준은 무엇인가

  상위 queue에서 할당한 시간이 만료될 경우 바로 다음 하위 queue로 강등
  상위 queue가 빌 때 까지 대기
  결국 CPU 사용 시간이 짧은 프로세스한테 우선순위를 많이 주는 방식

queue 하나에 대해 RR을 적용하는것 보다 더 **CPU burst가 짧은 프로세스를 우대**해주는 스케줄링

---

7. Multiple processor scheduling

CPU가 여러 개 있는 시스템에서의 스케줄링 방법

- Homogeneous processor인 경우
  → homogeneous processor의 각 코어는 동일한 유형의 작업을 수행하고 동일한 명령어를 실행하는 데 사용되는 설계
  - queue에 한 줄로 세워서 각각을 processor가 알아서 꺼내가게 할 수 있다.
  - 반드시 특정 프로세서에서 수행되어야 하는 job이 있는 경우,
    해당 job을 특정 프로세서에 먼저 할당하고 나머지를 할당할 수 있다.
- Load Sharing이 잘되어야 한다
  - 일부 프로세서에 job이 몰리고 나머지 프로세서는 놀고 있지 않도록 부하를 적절히 공유하는 메커니즘이 필요하다.
  - 각각의 CPU별로 별개의 queue를 두는 방법도 있고 공동 queue를 사용하는 방법도 있다

- **Symmetric Multiprocessing (SMP)**
  - 모든 CPU들이 대등한 것
  - 대등하기 때문에 각 프로세서가 각자 알아서 스케줄링 결정
- **Asymmetric multiprocessing (AMP)**
  - 여러 개의 CPU가 있는데 특정 CPU가 전체적인 컨트롤을 담당
  - 따라서 하나의 프로세서가 시스템 데이터의 접근과 공유를 책임지고 나머지 프로세서는 거기에 따른다.

---

8. Real Time scheduling

데드라인이 보장되는 방식 _ 정해진 시간 안에 반드시 실행이 되어야 함

> real-time job은 주로 특수 목적의 컴퓨터 시스템에서 적용되는데, 실행 결과가 제대로 출력되는 것 뿐만 아니라 실행되는 시점도 지정된 시간 내에 실행이 완료되어야 한다.
> (ex 미사일 방어 시스템, 자동 브레이크 시스템)real-time job들은 periodic한 특성을 가진 경우가 많다. (e.g. 10초에 한 번씩 최소한 1초 동안 CPU를 가져야 한다.)

이 방식은 real time job들을 보고 미리 스케줄링을 해서 데드라인을 보장해준다

- **Hard real-time system** – 정해진 시간 안에 반드시 끝나도록 스케줄링 해야 함
- **Soft real-time computing** – 일반 프로세스에 비해 높은 priority를 갖도록 함 (deadline 보장x)

---

9. Thread Scheduling

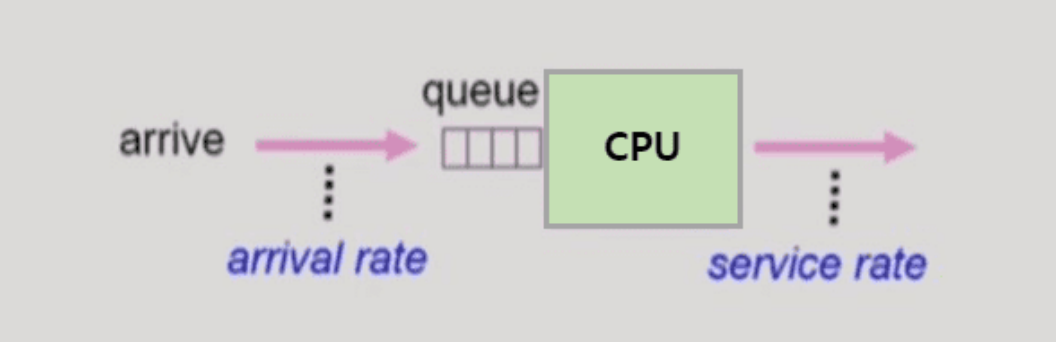
스레드 구현하는 방식 두 가지

- **Local Scheduling**
  User level thread의 경우 사용자 프로세스가 직접 thread를 스케줄링한다.
  운영체제는 thread의 존재를 모르는 상태에서 해당 프로세스에게 CPU를 줄지 말지를 결정할 뿐이다. 프로세스에 CPU가 주어지면 프로세스 내부에서 어떤 thread에게 CPU를 줄건지 결정한다.
- **Global Scheduling**
  Kernel level thread의 경우 운영체제가 thread의 존재를 알고 있다.
  프로세스와 마찬가지로 운영체제가 thread 스케줄링을 담당한다.

---

10. 알고리즘 평가 방법 _ Scheduling Algorithm Evaluation

어떤 알고리즘이 좋은 알고리즘인지 평가하는 방법에는 3가지가 있다



1. **Queueing models**

   확률 분포로 주어지는 arrival rate(프로세스의 도착률)와 service rate(CPU의 처리율) 등을 통해 각종 performance index 값을 계산해서 여러가지 성능 척도들을 도출하는 방법

   수식 계산을 통해 평가하는 방법이지만 최근에는 프로그램을 직접 돌려보는 방식이 선호

2. **Implementation & Measurement**

   실제 시스템에 알고리즘을 구현하여 실제 작업(workload)에 대해서 성능을 측정 비교

3. **Simulation**

   알고리즘을 모의 프로그램으로 작성 후 trace를 입력하여 결과 비교
