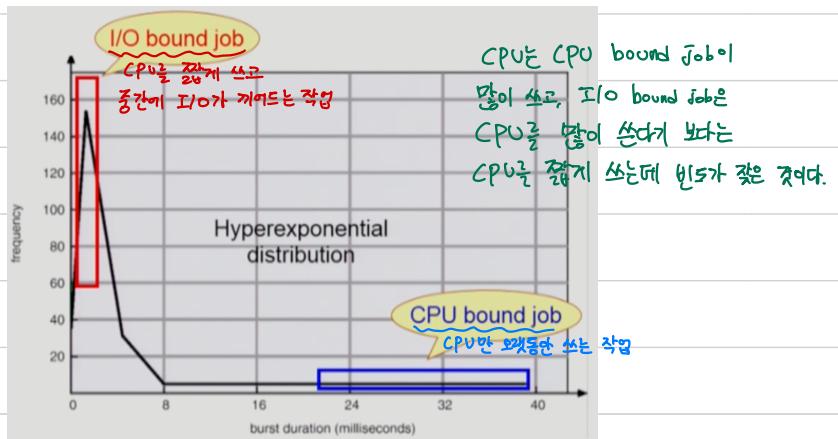


CPU and I/O Bursts in Program Execution



CPU-burst Time의 분포



** 여러 종류의 job (= process)이 섞여 있기 때문에 CPU 스케줄링이 필요하다.

- interactive job에게 적절한 response 제공 모양
- CPU와 I/O 장치 등 시스템 자원을 공유로 효율적으로 사용

프로세스의 특성 분류

- 프로세스는 그 특성에 따라 다음 두 가지로 나눌

v I/O-bound process

- CPU를 잡고 계산하는 시간보다 I/O에 많은 시간이 필요한 job
- many short CPU bursts

v CPU-bound process

- 계산 위주의 job
- few very long CPU bursts

CPU Scheduler & Dispatcher

- CPU scheduler

v Ready 상태의 프로세스 중에서 이번에 CPU를 줄 프로세스를 고른다.

- Dispatcher

v CPU의 제어권을 CPU Scheduling에 의해 선택된 프로세스에게 넘긴다.

v 이 과정을 Context Switch(문맥 교환)라고 한다.

- CPU 스케줄링이 필요한 경우는 프로세스에게 다음과 같은 상태 변화가 있는 것이다.

1. Running → Blocked (예: I/O 요청하는 시스템 풀)

2. Running → Ready (예: 할당 시간 만료로 Timer Interrupt)

3. Blocked → Ready (예: I/O 완료 후 인터럽트)

4. Terminate

* 1, 4에서의 스케줄링은 **non-preemptive** (= 강제로 빼앗지 않고 차관 반납, 비선점)

* All other scheduling is preemptive (= 강제로 빼앗음, 선점)

Scheduling criteria

performance Index (= Performance Measure, 성능 측도)

CPU utilization (이용률)

- Keep the CPU as busy as possible

⇒ 전체 시간 중에서 CPU가 놓지 않고 일한 비율

→ 가능한 바쁘게

→ 시스템 입장에서의 성능 측도

(CPU 하나 가지고 최대한 일을 많이 시키면 좋은 것)

Throughput (처리량)

- * of processes that complete their execution per time unit.

⇒ 주어진 시간 동안에 몇개의 작업을 완료 했느냐

Turnaround Time (소요시간, 반환시간)

프로세스가 시작되서 종료될 때까지의 시간의 총 합이 아니라 CPU를 쓰려 들어와서 I/O 처리 끝까지의 총시간을 의미한다.

Waiting Time (대기시간)

- amount of time to execute a particular process

⇒ CPU를 사용하려 들어와서 다 쓰고 나갈 때 까지 걸린 시간.

Response Time (응답시간)

- amount of time process has been waiting in the ready queue

⇒ 순수하게 ready queue에서 줄 서서 기다린 시간.

CPU를 얻었다 빼었다 하는 Time-sharing 환경에서는

⇒ 처음으로 CPU를 얻는 시간이 사용자 응용과 관련해서 굉장히 중요한 시간 개념.

- amount of time it takes from when a request was submitted

until the first response is produced, not output (for time-sharing environment)

⇒ ready queue에서 들어와서 처음으로 CPU를 얻기까지 걸린 시간.

* 선점형 스케줄링에서 ready queue에서 '최초'로 CPU를 얻기 까지 걸린 시간 ⇒ Response Time

CPU를 받고 뺏기고 받고 뺏기고 반복할 때

그 때마다 ready queue에서 대기한 각각의 시간 ⇒ Waiting Time

CPU 기다리는 시간, 쓴 시간, 기다리는 시간, 쓴 시간 더 합쳐서 들어와서 나갈 때 까지 ⇒ Turnaround Time

프로세스 입장에서의 성능 측도

(CPU를 빨리 얻어서 빨리 끝나면 좋은 것)

Scheduling Algorithms

- FCFS (First-come First-served)

- SJF (Shortest-Job-First)

- SRTF (Shortest-Remaining-Time-First)

- Priority scheduling

- RR (Round Robin)

- Multilevel Queue

- Multilevel feedback queue

FCFS (First-Come First-Served)

— Example

Process	Burst Time
P ₁	24
P ₂	3
P ₃	3

— 프로세스의 도착 순서 P₁, P₂, P₃

스케줄 순서를 Gantt Chart로 나타내면 다음과 같다.

P ₁	P ₂	P ₃
24	27	30

. Waiting Time for P₁=0, P₂=24, P₃=27

. Average Waiting Time: (0+24+27)/3 = 17

만약 P₂, P₃, P₁ 순으로 들어 왔다면

Waiting Time for P₁=6, P₂=0, P₃=3

Average Waiting Time: (6+0+3)/3 = 3

• 비선점형

• 썩 효율적이지 않음

• CPU를 짧게 쓰는 프로세스들이 도착 하면서도 긴 프로세스가 CPU를 놓기 까지 기다려야 함.

* Convoy effect : 긴 프로세스가 도착해서 뒤에 짧은 프로세스들이 차단하게 오래 기다리는 현상

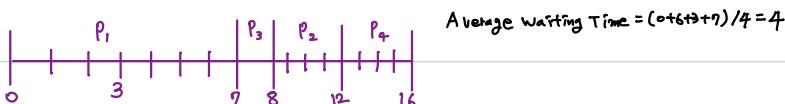
SJF (Shortest-Job-First)

- 각 프로세스의 다음번 CPU burst time을 가지고 스케줄링에 활용
- CPU burst time이 가장 짧은 프로세스를 제일 먼저 스케줄
- Two Schemes:
 - Nonpreemptive
 - 일단 CPU를 잡으면 어떤 CPU burst가 와온다 떄까지 CPU를 선점(preemption) 당하지 않음.
 - Preemptive
 - 현재 수행중인 프로세스의 남은 burst time 보다 더 짧은 CPU burst time을 가지는 새로운 프로세스가 도착하면 CPU를 빼앗김.
 - 이 방법을 Shortest-Running-Time-First(SRTF)이라고도 부른다.
- SJF is optimal
 - 주어진 프로세스들에 대해 minimum average waiting time을 보장

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P ₁	0.0	7
P ₂	2.0	4
P ₃	4.0	1
P ₄	5.0	4

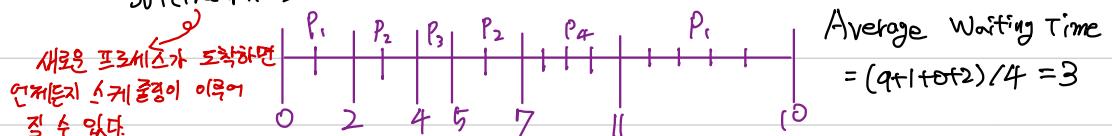
- SJF (Non-preemptive) \rightarrow CPU를 더 쓰고 나가는 시점에 Scheduling을 할지 결정



Example of preemptive SJF

Process	Arrival Time	Burst Time
P ₁	0.0	7
P ₂	2.0	4
P ₃	4.0	1
P ₄	5.0	4

• SJF(Preemptive)



* SJF의 문제점 : 1. Starvation(기아 현상)

↳ 짧은 burst-time을 가지는 프로세스들이 계속 들어오면

긴 burst-time을 가지는 프로세스는 계속 CPU에 올라가지 못하는 상황

2. CPU 사용 시간을 미리 알 수 없다.

↳ 실제 시스템에선 프로세스가 얼마나 오래 사용할지 모르기 때문에 불가능함.

물론 CPU 사용률을 정확하게 알 수 없지만 예측은 할 수 있다.

↳ 과거에 얼마나 사용했는지를 트대로

다음 CPU Burst Time의 예측

- 다음번 CPU burst time을 어떻게 알 수 있는가? (Input data, branch, user...)

- 추정(estimate)만이 가능하다

- 과거의 CPU burst time을 이용하여 추정(exponential averaging)

1. t_n = actual length of n^{th} CPU burst

$\therefore t$: 실제 CPU 사용시간

2. T_{n+1} = Predicted value for the next CPU burst

T : CPU 사용을 예측한 시간

3. $\alpha, 0 \leq \alpha \leq 1$

수정 배율

4. Define : $T_{n+1} = \alpha t_n + (1-\alpha) T_n$

Exponential Averaging

- $\alpha = 0$

v $I_{n+1} = I_n$

v Recent history does not count.

- $\alpha = 1$

v $I_{n+1} = t_n$

v Only the actual last CPU burst counts

극단적인 경우

- 식을 풀면 다음과 같다.

$$\begin{aligned} I_{n+1} &= \alpha t_n + (1-\alpha) \alpha t_{n-1} + \dots \\ &\quad + (1-\alpha)^2 \alpha t_{n-2} + \dots \\ &\quad + (1-\alpha)^{n-1} \alpha t_1 \end{aligned}$$

- α 와 $(1-\alpha)$ 가 둘다 1 이하이므로 후속 term은 전행 term보다 적은 가중치 값을 가진다.

Priority Scheduling

- A priority number (integer) is associated with each process

- highest priority를 가진 프로세스에게 CPU 할당 (smallest integer = highest priority)

v Preemptive

v Nonpreemptive

- SJF는 일종의 priority scheduling alg.

Priority = predicted next CPU burst time.

- Problem

v Starvation (기아현상) : low priority processes may never execute.

- Solution

v Aging : as time progresses increase the priority of the process.

Round Robin (RR)

- 각 프로세스는 동일한 크기의 할당 시간 (time quantum)을 가짐. (일반적으로 10~100 milliseconds)
- 할당 시간이 지나면 프로세스는 선점 (preemptive) 당하고 ready queue의 제일 뒤에 가서 다시 줄을 선다.
- n개의 프로세스가 ready queue에 있고 할당 시간이 q time unit인 경우 각 프로세스는 최대 q time unit 단위로 CPU 시간의 1/n을 얻는다.
→ 어떤 프로세스도 (n)q time unit 이상 기다리지 않는다.

- Performance

↳ v q large \Rightarrow FIFO(FCFS)

↳ v q small \Rightarrow Context Switch overhead가 커진다.

→ 적당한 크기의 time quantum을 주는게 바람직하다.

* 중요한 장점: 누구든지 짧은 시간만 기다리면 적어도

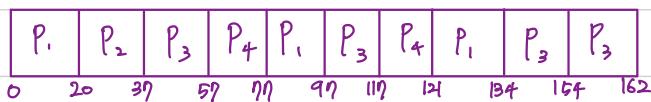
CPU를 한 번 맷 볼 수 있는 기회가 생김

그리고 누가 CPU를 오래 쓰지 모르는 상황에서
굳이 예측 할 필요가 없다.

Example : RR with Time Quantum = 20

Process	Burst Time
P ₁	53
P ₂	17
P ₃	68
P ₄	24

- The Gantt Chart is:



- 일반적으로 SJF 보다 average turnaround time이 길지만 response time은 짧다.

* 특이한 예로 프로세스의 burst time이 모두 같다면 RR에서는 끝나는 시점이 모두 늦게 끝나지만 SJF에서는 모두 늦게 끝나지 않고 순서대로 끝나기 때문에 이를면 SJF가 더 효과적일 수 있다.

RR의 장점은 turnaround time이 아니라 response time이 빨라진다는 점을 생각하자!

Multilevel Queue

- Ready Queue를 여러 개로 분할.

큐 풀개
v foreground (interactive)
v background (batch - no human interaction)

- 각 큐는 독립적인 스케줄링 알고리즘을 가짐.

v foreground - RR → 사용자 interaction 하는 프로세스들이므로 RR이 아무래도 효율적

v background - FCFS → CPU만 사용하는 프로세스들 응답시간이 빠르고 좋을게 없다. 그래서 Context Switch overhead를 줄여 위해 FCFS가 조금 더 효율적

- 큐에 대한 스케줄링이 필요

v Fixed priority scheduling

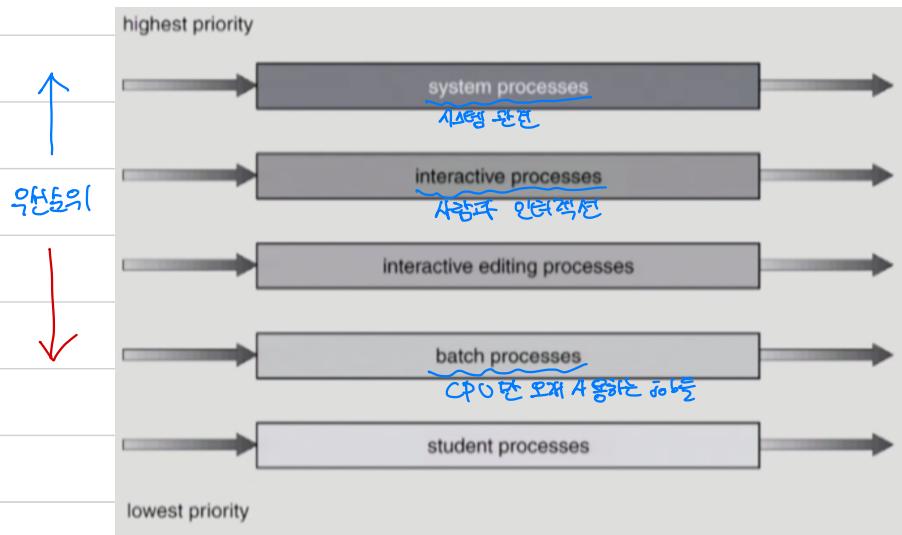
• Serve all from foreground then from background

• Possibility of Starvation

v Time Slice

• 각 큐에 CPU time slice를 조정한 비율로 할당

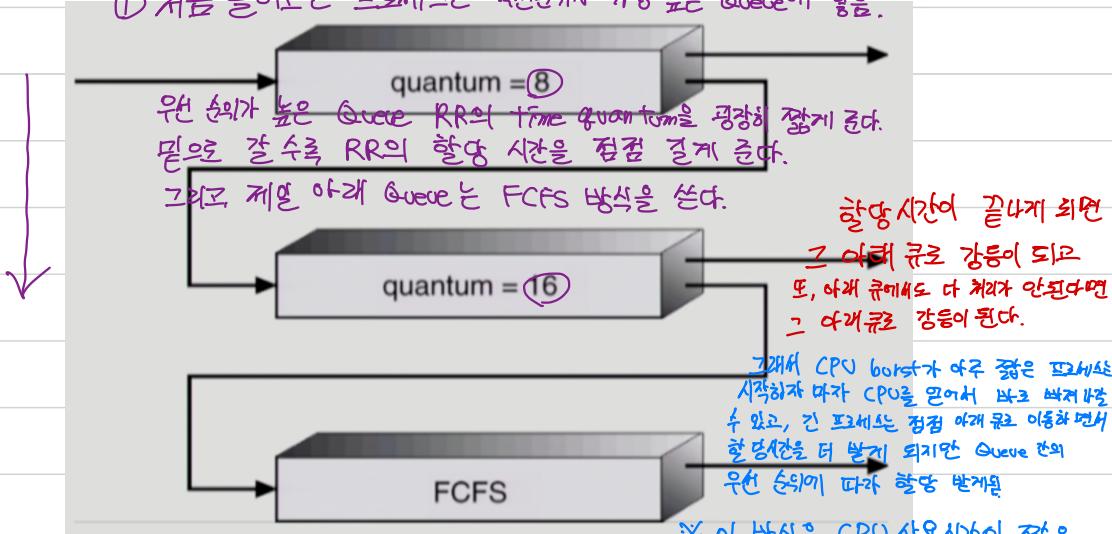
• Eg., 80% to foreground in RR, 20% to background in FCFS



Multilevel Feedback Queue

- 프로세스가 다른 큐로 이동 가능
- 에이징(Aging)을 이용해 같은 방식으로 구현할 수 있다.
- Multilevel-feedback-queue scheduler를 정의하는 파라미터들
 - v Queue의 수
 - v 각 큐의 Scheduling algorithm
 - v Process를 상위 큐로 보내는 기준
 - v Process를 하위 큐로 내쫓는 기준
 - v 프로세스가 CPU 서비스를 받으려 할 때 들어갈 큐를 결정하는 기준

① 처음 들어오는 프로세스는 우선순위가 가장 높은 Queue에 넣음.



• Three Queues :

- v Q₀ - Time quantum 8 milli seconds
- v Q₁ - Time quantum 16 milli seconds
- v Q₂ - FCFS

• Scheduling

- v New job이 Queue Q₀로 들어감.
- v CPU를 끌어서 할당 시간 8milli seconds 동안 수행됨.
- v 8 milliseconds 동안 다 끌어지 못했으면 Queue Q₁으로 내려감.
- v Q₁에 끌어서 기다렸다가 CPU를 끌어서 16ms 동안 수행됨.
- v 16ms에 끌어지 못한 경우 Queue Q₂로 쫓겨남.

Multiple - Processor Scheduling

- CPU가 여러개인 경우 스케줄링은 더 복잡해짐

- Homogeneous processor인 경우

v Queue에 한줄로 세워서 각 프로세서가 알아서 끄내가게 할 수 있다.

v 반드시 특정 프로세서에서 수행되어야 하는 프로세스가 있는 경우에는 문제가 더 복잡해짐.

- Load sharing

v 일부 프로세서에 任务이 몰리지 않도록 부하를 적절히 공유하는 메커니즘 필요

v 별개의 큐를 두는 방법 VS 공동 큐를 사용하는 방법

- Symmetric Multi processing (SMP)

v 각 프로세서가 각자 알아서 스케줄링 결정

- Asymmetric Multi processing

v 하나의 프로세서가 시스템 대역의 접근과 공유를 책임지고 나머지 프로세서는 그것에 따름
전체적인 컨트롤 많음

* real-time job > 정해진 시간안에 반드시 수행해야 하는 job

Real - Time Scheduling

- Hard real-time systems

v Hard real-time task는 정해진 시간안에 반드시 끝내도록 스케줄링 해야 함.
CPU에서 deadline 만에 처리가 끝나도록 해야함

- Soft real-time Computing

v Soft real-time task는 일반 프로세스에 비해 높은 Priority를 갖도록 해야 함.

↳ deadline을 조금 어기고 어쩔 수 없는 시스템.

그때 그때 스케줄링 하기 보다는 real-time job들이 주기적 있고, 미리 스케줄링을 해서 deadline이 오기 되도록 계획적으로 배치를 하는 방법을 사용.

↳ 그래서 반드시 deadline을 보장 해야 할 때는 오프라인에서 미리 스케줄링을 하거나 또는, real-time job들이 period는 한 생활을 가진 것들이 많이 있어서 정해진 시간 안에 아무데서나 실행되면 된다던가 하는 것을 (ex. 1초에 한번, 5초에 한번)

Thread Scheduling

↳ Process안에 여러개의 CPU 수행단위

- Local Scheduling => OS가 하는게 아니라 사용자 스레드가 직접 어느 스레드에 CPU를 줄지 결정

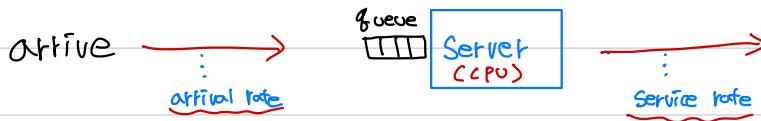
v User level thread의 경우 사용자 수준의 Thread library에 의해 어떤 Thread를 스케줄 할지 결정
↳ 사용자 프로세스가 작업 스레드를 관리하고 운영체제는 스레드의 존재를 모르는 경우

- Global Scheduling => 프로세스 스케줄링 하듯이 알고리듬에 의해 OS가 어떤 스레드에 CPU를 줄지 결정

v Kernel level thread의 경우 일반 프로세스와 마찬가지로 커널의 단기 스케줄러가 어떤 thread를 스케줄 할지 결정

↳ 운영체제가 스레드의 존재를 이미 알고 있는 상황

Algorithm Evaluation



이론적인 방법, 이론가들의 주로 사용

- Queueing models

- ✓ 확률 분포로 주제는 arrival rate와 service rate 등을 통해 각종 performance index 값을 계산

- Implementation (구현) & Measurement (성능 측정)

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

ex) Linux에 구현한 알고리즘으로 코드를 고쳐서 workload를 하고 기존 Linux 이 그대로 workload를 해서 두개의 결과를 비교

- Simulation (모의 실험)

- ✓ 알고리즘을 모의 프로그램으로 작성후 trace를 입력으로 하여 결과 비교

SST 초기 계산 예제 같은 것을

→ input data (임의로 만들 수도 있고, 프로그램을 통하여 뽑을 수도 있음)

실제 CPU burst 패턴에 근거해서 사용자에게 한대면 좀 더 신빙성을 있을 것이다.