

### CPU and I/O Bursts in Program Execution

CPU에서 인스트럭션은 수행되는 거지. →

0:48:32

- 1. CPU burst와 I/O burst가 번갈아 나온다.
- 2. CPU burst만 나오다가 I/O가 나올수도, 빈번하게 두개가 번갈아 나올 수 있다.

### Job의 종류가 섞여있다.

#### CPU-burst Time의 분포

Hyperexponential distribution

0:48:32

- 1. I/O bound job 번갈아 실행되는 빈도가 높은 프로세스
- 2. CPU bound job CPU가 전담이 실행되는 프로세스.

CPU 스케줄링시 주의점.  
공평한 것이 아닌 효율적 측면.  
Interactive 한 작업을 오래 기다리지 않게 하는 것.

### 프로세스의 특성 분류

- 프로세스는 그 특성에 따라 다음 두 가지로 나뉨
- ✓ I/O-bound process
  - CPU를 잡고 계산하는 시간보다 I/O에 많은 시간이 필요한 job
  - (many short CPU bursts)
- ✓ CPU-bound process
  - 계산 위주의 job
  - (few very long CPU bursts.)

0:58:00

- 1. CPU를 줄 것이지.
- 2. 배넷을 것이지.

### CPU Scheduler & Dispatcher

- ✓ CPU Scheduler
  - Ready 상태의 프로세스 중에서 이번에 CPU를 줄 프로세스를 고른다.
- ✓ Dispatcher
  - CPU와 제어권을 CPU scheduler에 의해 선택된 프로세스에게 넘긴다
  - 이 과정을 context switch(문맥 교환)라고 한다
- CPU 스케줄링이 필요한 경우는 프로세스에게 다음과 같은 상태 변화가 있을 경우이다.
  - Running → Blocked (예: I/O 요청하는 시스템 콜)
  - Running → Ready (예: 할당 시간만료로 timer interrupt)
  - Blocked → Ready (예: I/O 완료 후 인터럽트)
  - Terminate

모든 프로세스가 CPU를 받을 수 있는 것은 아니다. (구현에 따라)

1, 4에서의 스케줄링은 nonpreemptive (=강제로 빼앗지 않고 자진 반납)

All other scheduling is preemptive (=강제로 빼앗음)

더 이상 수행할 게 없어서 자진 반납.

0:58:00

세심하게 읽고 넘어가야 할 사항.  
1. CPU Scheduler는 하드웨어가 변조로 감당하는데.  
해당 프로그램이 변조로 인해서.  
⇒ 운영체제에 cpuScheduling하는 코드가 변조로 인해서.  
그것을 CPU scheduler라 한다.

### CPU Scheduling Algorithm.

- 1. nonpreemptive한 알고리즘.
- 2. preemptive한 알고리즘. 조금 더 빠른 이점.

# Scheduling Criteria

## CPU utilization (이용률)

keep the CPU as busy as possible

## Throughput (처리량)

# of process that complete their execution per time unit

## Turnaround time (소요시간, 변환시간)

amount of time to execute a particular process

CPU 라 쓰고 나갈 때까지 걸린 시간. \* CPU scheduling 관점이기 때문이

프로세스가 종료될 때 까지 아닌 I/O 할 때까지 총 시간임.

## Waiting time (대기시간)

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

가이드라인 시간

## Response time (응답시간)

time sharing 시 필요.

amount of time it takes from when a request was submitted until the first response

is produced, not output

ready queue에 들어와서 처음으로 받은 시간.

(for time-sharing environment)

성능 척도.

1. 시스템 입장에서 성능 척도.

2. 프로그램 입장에서 성능 척도. (프로세스)

waiting time과 response time의 차이.

프로세스 A가 CPU 변경과 밀려나 반복.

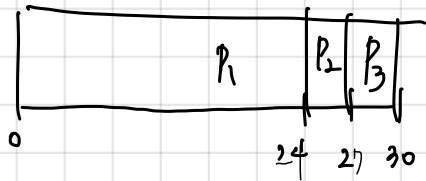
변경된 ready queue에 서서 차례 기다리는 시간 이므로.

총합! = waiting time., 첫 서비스는 response time.

# Scheduling Algorithms

## 1. FCFS (First Come First Served)

- nonpreemptive.

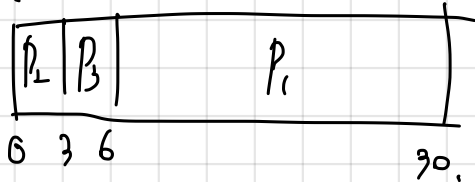


P <sub>i</sub>	Waiting Time
P <sub>1</sub>	0
P <sub>2</sub>	24
P <sub>3</sub>	27

Mean =  $\frac{0+24+27}{3}$   
= 17

P<sub>2</sub>, P<sub>3</sub>가 interactive한 Job 이었다면  
매우 비효율적.

if.



P <sub>i</sub>	Waiting time
P <sub>2</sub>	0
P <sub>3</sub>	3
P <sub>1</sub>	6

Mean =  $\frac{0+3+6}{3}$   
= 3

Much better than previous case.

**Convoy effect**: short process behind  
long process

# 2 SJF(Shortest-Job-First)

CPU burst time이 가장 짧은 프로세스를 제일 먼저 스케줄.

minimum average waiting time 보장.  
*(preemptive 한 방식?)*

두 가지 방식.

1. Nonpreemptive.

2. Preemptive. (=Shortest-Remain-Time-First)

수행중인 프로세스보다 더 짧은 Cpu burst 프로세스가 안면 이를 수행.

문제점.

1. starvation 문제.  
CPU burst가 처리x

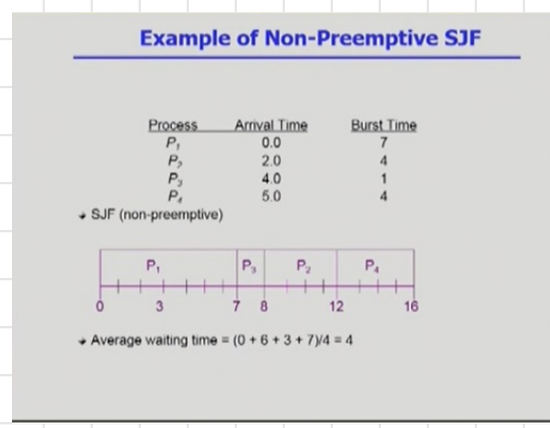
2. CPU 사용시간을 미리 알기 어렵다.

*추정 가능 (과거 CPU 사용량도 보지 추정한다.)*

*exponential averaging*

- 1.  $t_n$  = actual length of  $n^{th}$  CPU burst
- 2.  $Z_{n+1}$  = predicted value for next CPU burst.
- 3.  $\alpha, 0 \leq \alpha \leq 1$
- 4. Define  $Z_{n+1} = \alpha t_n + (1-\alpha)Z_n$   
 $\Rightarrow$  일반식,  $Z_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \dots$

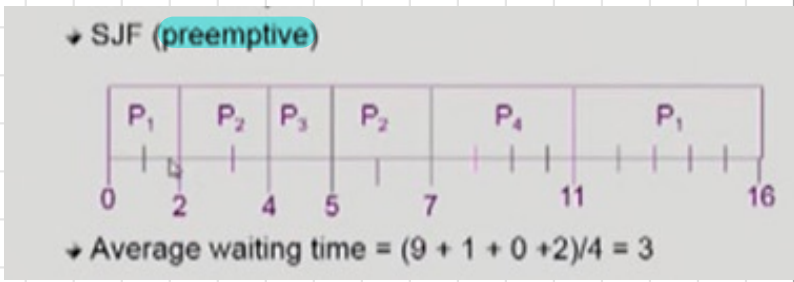
$$+ (1-\alpha)^j \alpha t_{n-j} + \dots + (1-\alpha)^{n-1} Z_0$$



0초 시점에 도착프로세스 있음.

$P_1$ 의 Burst Time이 끝난 시점에  $P_2 \sim P_4$  모두 도착.

이 중에서 CPU Burst 타임이 작은것 수행.



스케줄링을 하느니 있어서 도착시 변경가능.

## 3. Priority Scheduling

highest priority를 가진 프로세스에게 CPU 할당.  
(Smallest Integer = highest priority)

Preemptive

Non-preemptive. 두 방법 존재.

SJF: priority = predicted next CPU burst time.

Starvation Solution.

Aging: as time progresses increase the priority of the process

## 4. Round Robin (RR)

현대 컴퓨터에서  
가장 흔한 스케줄링.

각 프로세스는 동일한 크기의 할당 시간(time quantum)을 가짐.

n개의 프로세스가 ready queue에 있고, 할당 시간이 q time unit인 경우

각 프로세스는 최대 q time unit 단위로 CPU 시간의 몫을 얻는다.

⇒ 어떤 프로세스도 (n-1)q time unit 이상 기다리지 않는다.

→ 반증성

Waiting time  $\propto$  CPU burst.

Performance

q large  $\Rightarrow$  FCFS

q small  $\Rightarrow$  context switch 오버헤드가 커진다.

⇒ 적당한 크기의 q를 부여하자. (10 ~ 100 msec)

CPU 사용 시간이 모두 동일한 프로그램에서  
(ex: 100초 4개)  
4개 일 경우,

400초일 때 4프로세스 모두 종료.

1개씩 돌리면 100초마다 1개씩 빠져나가므로

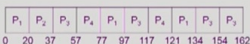
waiting time이 적다.

라운드 로빈은 무작위 CPU burst일 경우 효율적으로 작동한다.

Example: RR with Time Quantum = 20

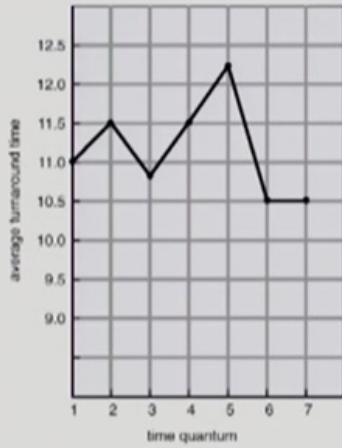
Process	Burst Time
P <sub>1</sub>	53
P <sub>2</sub>	17
P <sub>3</sub>	68
P <sub>4</sub>	24

\* The Gantt chart is:



\* 일반적으로 SJF보다 average turnaround time이 짧지만 response time이 더 길다.

## Turnaround Time Varies With Time Quantum



process	time
$P_1$	6
$P_2$	3
$P_3$	1
$P_4$	7

6  
3  
1  
7

줄이 여러 개이면 CPU는 하나기 때문에 고려해야 할 사항이 존재.

1. 프로세스를 어느 줄에 세울 것인가?

2. 낮은 우선순위에 큐의 starvation

이화여자대학교 EWHA WOMANS UNIVERSITY

### Multilevel Queue

우선순위에 따른 큐 구조:

- highest priority
- system processes
- interactive processes
- interactive editing processes
- batch processes
- student processes
- lowest priority

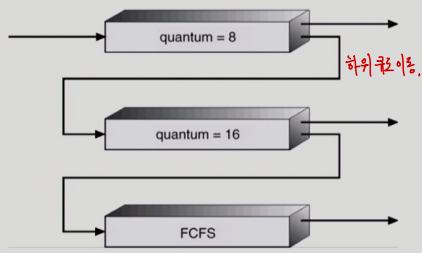
이화여자대학교 EWHA WOMANS UNIVERSITY

### Multilevel Queue

- Ready queue를 여러 개로 분할 **여기 묶어?**
  - ✓ foreground (interactive)
  - ✓ background (batch - no human interaction)
- 각 큐는 독립적인 스케줄링 알고리즘을 가짐 **큐 내 스케줄링.**
  - ✓ foreground - RR
  - ✓ background - FCFS
- 큐에 대한 스케줄링이 필요
  - ✓ Fixed priority scheduling
    - serve all from foreground then from background.
    - Possibility of starvation.
  - ✓ Time slice
    - 각 큐에 CPU time을 **적절한 비율**로 할당
    - Eg., 80% to foreground in RR, 20% to background in FCFS



## Multilevel Feedback Queue



보통

1. 첫번째 프로세스는 상위 큐,
2. 두번째 프로세스는 하위 큐...

상위 큐에 더 많은 프로세스가 있으면  
하위 큐로 이동하여 처리.

즉, CPU 사용시간이 짧은 프로세스에게 우선순위가 높다.

CPU 사용시간 예측값이 짧은 프로세스가 우선시된다.

## Multilevel Feedback Queue

- 프로세스가 다른 큐로 이동 가능
- 에이징(aging)을 이와 같은 방식으로 구현할 수 있다
- Multilevel-feedback-queue scheduler를 정의하는 파라미터들
  - Queue의 수
  - 각 큐의 scheduling algorithm
  - Process를 상위 큐로 보내는 기준 **점**
  - Process를 하위 큐로 내보내는 기준
  - 프로세스가 CPU 서비스를 받으려 할 때 들어갈 큐를 결정하는 기준

↑ 단일 프로세서

↓ 멀티 프로세서 or 시간 dead line 조건 or 다중 Thread.

## Multiple-Processor Scheduling

- CPU가 여러 개인 경우 스케줄링은 더욱 복잡해짐
- Homogeneous processor인 경우**
  - Queue에 한줄로 세워서 각 프로세서가 알아서 꺼내게 할 수 있다
  - 반드시 특정 프로세서에서 수행되어야 하는 프로세스가 있는 경우에는 문제가 더 복잡해짐 (ex) **전달 데이터**
- Load sharing**
  - 일부 프로세서에 job이 몰리지 않도록 **무한정 적절히 공유**하는 여러 이름 필요
  - 별개의 큐를 두는 방법 vs. 공통 큐를 사용하는 방법
- Symmetric Multiprocessing (SMP) 대등**
  - 각 프로세서가 각자 알아서 스케줄링 결정
- Asymmetric multiprocessing**
  - 하나의 프로세서가 시스템 데이터의 접근과 공유를 책임지고 나머지 프로세서는 거기에 따름

## Real-Time Scheduling = 정해진 시간 내에 끝내야 하는 job

- Hard real-time systems**
  - Hard real-time task는 정해진 시간 안에 반드시 끝내도록 스케줄링해야 함
- Soft real-time computing**
  - Soft real-time task는 일반 프로세스에 비해 높은 priority를 갖도록 해야 함

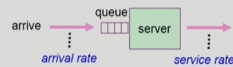
· Real-Time Job  
보통 귀기쁨 가진다.

· Soft real-time Job  
· 영화 같은 것. 데드라인은 꼭 지켜야 하지 못함.

## Thread Scheduling

- **Local Scheduling**
  - ✓ **User level thread**의 경우 사용자 수준의 **thread library**에 의해 어떤 thread를 스케줄할지 결정  
*프로세스 내부에서 결정*
- **Global Scheduling**
  - ✓ **Kernel level thread**의 경우 일반 프로세스와 마찬가지로 커널의 단가 스케줄러가 어떤 thread를 스케줄할지 결정

## Algorithm Evaluation



- **Queueing models**
  - ✓ 확률 분포로 주어지는 **arrival rate**와 **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
- **Simulation (모의 실험)**
  - ✓ 알고리즘을 모의 프로그램으로 작성 후 **trace**를 입력으로 하여 결과 비교  
*실제 프로그램에서 추출한 data*

· 알고리즘 평가 방법.

1. Queueing models

광장히 이론적인 방법.

server = cpu

2.

만약 linux CPU Scheduling과 서로 만든 CPU Scheduling을 비교한다면, linux kernel source code를 수정하여 새로운 알고리즘을 구현한 커널을 컴파일하여 bin code, 실행.

실제 프로그램을 들어 어떤 스케줄러가 좋은지 평가.

but, kernel 수정은 매우 힘든 작업.

3. Simulation.

복잡한 프로그램에 대해 모의 프로그램을 작성하여 계산.



