Chapter 13 : M/M/1 to GPS

13.1 Generalized Processor Scheduling

Generalized Processor Scheduling (GPS): generalization of a resource sharing scheme with ideal properties.

When a single job A needs to be processed, it is allocated the full processing capability of the server. (just like M/M/1) + However, as soon as a second job B arrives and requests service, instead of queuing the job, we distribute the capacity in two portion.

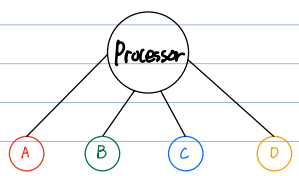
Equal distribution을 원하면 assign 50% of server capacity to A and rest 50% to B. The quantities w_B and w_A can be seen as "weight" and are such that $w_A, w_B \in [0,1]$, and that $w_A + w_B \leq 1$ 100%

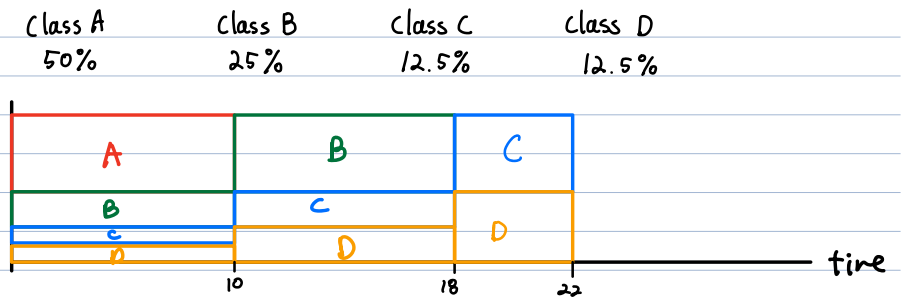
N jobs, $i \in \{1,2,\dots,N\}$, and that $w_1 + w_2 + w_3 + \cdots w_N = \sum_{i=1}^{N} w_i \leq 1$. so, if the server under consideration has capacity C_s, then the capacity C_i assigned to each job be.

server capacity

$$C_i = \frac{w_i}{\sum_{i=1}^{N} w_i} \cdot C_s$$

그냥 얼마만큼 차지하냐 이거야 / 생각해보면 직렬이 끝나는 i들이 있으니, 직렬해 있는 i들 중에서만 계산



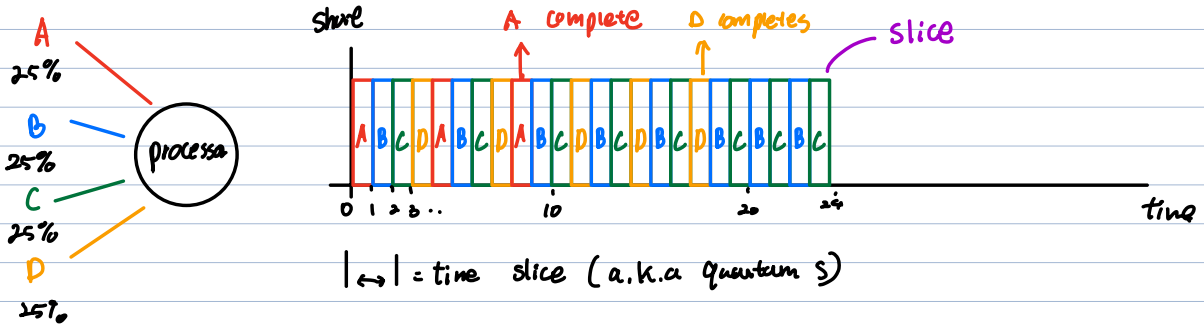


processor assignment 4:2:1:1

13.2 Round Robin (RR)

- No perfect way to split 50, 50

So we implement time quantum | 시간대를 나눠서 돌아가면서 작업을 수행



|↔| : time slice (a.k.a quantum S)

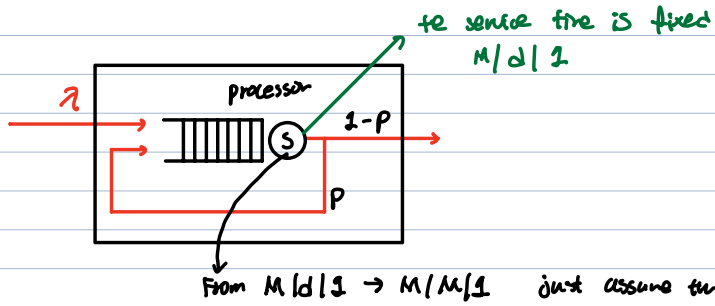
Round-Robin Scheduling

No job is picked twice until the "round" is completed

How long is each job? We don't know in principle.
→ But once it completes a slice, it may be done or not.

P | the probability of the process is not done after the slice

1 − P | " done



From M/d/1 → M/M/1 just assume that slice is exponentially distributed with mean 1

끝날 확률이 1−P, 다시들어갈 확률이 P, 베르누이 법칙을 따른다.

f(m): m 번이후에 다시 일하러 re-enter : $P^m(1-P)$ 하고 끝날 확률

그러니까, 총 일은 m+1 번 해야한다.

$t = (m+1) \cdot S$ → m 번다시들어가고 끝난 일의 처리량
↓ processor (execute process)

Geometric distribution 에서 평균 값은

$$E[m] = \frac{\rho}{1-\rho}$$

↓

이것으로 service time, T_S 구할수 있다.

$$T_S = E[t] = E[(m+1)\cdot s] = (E[m]+1)\cdot s = \left(\frac{\rho}{1-\rho}+1\right)\cdot S = \left(\frac{1}{1-\rho}\right)\cdot S = \left(\frac{S}{1-\rho}\right)$$

13.3 — System Utilization

1) 무한히 세밀하게 분할 할수 있는 시스템에서 $S \to 0$ 에 가깝게 사용하면 m수가 급격하게 늘어나, 작업을 완료하는데 엄청나게 많은 다운로드가 필요합니다.

2) ρ 는 1 (다시 명심하세요) 에 가까워 진다. $(1-\rho) \to 0$ 에 가까워지고 Geometric distribution 이 exponential 과 매우 비슷해진다. 이때 exponential 은 μ 와 $\frac{1}{\mu}$ 를 가집니다.

T_S (service Time) $= E[(m+1)\cdot S] = E[m\cdot s] + S$

when $S \to 0$, $T_S \doteq E[m\cdot s] + 0 = \frac{1}{\mu}$

그래서 결국 $T_S \doteq \frac{1}{\mu} \doteq \frac{S}{1-\rho}$

여태 말해온 M/D/1 → M/M/1 시스템으로 바꾼다. 왜? Jackson theorem 같이 다른 법칙을 사용하고싶어서

어떻게?

slice S 를 exponentially distributed 도 랜하게 작업.

새로운 도착률 λ'

$\lambda' \cdot (1-\rho) = \lambda$ In other words: $\lambda' = \frac{\lambda}{(1-\rho)}$

$\rho = \lambda' \cdot T_S$

$= \lambda' \cdot S = \frac{\lambda \cdot S}{1-\rho}$, since $\mu \cdot s = 1-\rho$ $\frac{\lambda \cdot S}{\mu \cdot s} \doteq \frac{\lambda}{\mu}$

말에 대한 보고겠어

Chapter 14 Basic Resource Scheduling

common strategies to perform Scheduling for a Single resource

14.1 Average vs Actual Case

여태까지 Average를 봤으니, process-centric 에서 보면 "resource management" 를 보자.

Resource management는 시스템에 다음과 같다

1) Resources are fairly Utilized
2) Requests are classified according to some metric of importance.

) 특히 heavily utilized 되어있으면 변화를 더 잘 보고 있다.

14.2 Resource and Scheduling

항상 최고의 성능을 가질 수 있는 Scheduling 은 "불가능"

성능측정 기준과 관리 자원에 따라 다르기 때문

단일 자원

i) Stateful & Stateless
2) preemptive & non-preemptive

- Stateless
  - Amount of time to service a given request is independent from the history request
  - no resource is truly stateless. → 전의요청이 새요청이 무시할 수 있을 정도로 영향이 적을때
- Stateful
  - the time it takes to satisfy a request does depend on previously serviced.

CPU : stateless. Cache: Stateful.

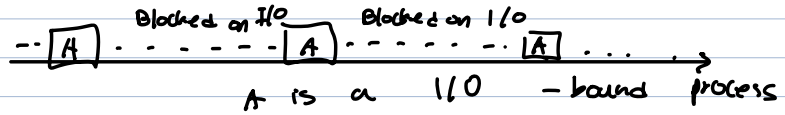
- preemptive (선점 가능한 자원): Resources that can be temporarily interrupted to serve a different request
- Non-preemptive (비선점 자원) : Resources that cannot be interrupted while servicing a request.
  ex) printer, modem GPU

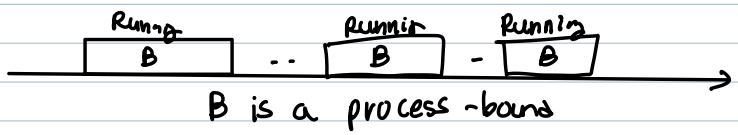
- Scheduling Overhead : 스케줄링 결정에 필요한 "Extra computation"
- Scheduling Event : scheduler가 호출 되는시점, 다음 Schedule이 시작되는것

— Scheduling policy : 어떤 작업을 다음에 실행할지 결정

Process states
1) Ready
2) blocked waiting

Bound processes — job(process)
1) Processor bound : 긴 job과 짧은 I/O 상호작용, 대부분 프로세서에서 작업을 수행 ex) 수학계산, 데이터처리
2) I/O-bound : 짧은 job과 긴 I/O 대기 시간을 갖는다. , 읽고, 쓰기, 네트워크통신



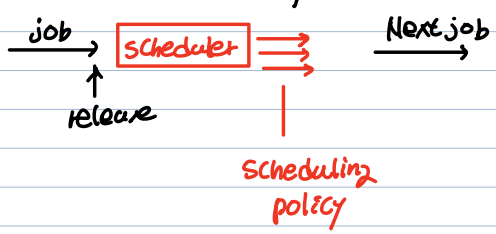


Job and Task
- Job: An interval of execution that can be entirely conducted on the processor.

- Task: a sequence of jobs

※ process-bound, i/o bound는 상대적인 것. GPU 환경에서 CPU도 상대적인것

Scheduler Anatomy



Scheduler Invocation	When it wakes up and takes a decision

Scheduling policy | How the decision is taken.

process-centric metrics

- Response time (turnaround time)

$R = t_f - t_r$

↙ job has completed 작업이 완료되었을 때 ↘ release time of job 잡이 도착했을 때

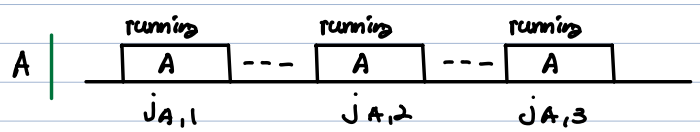
- Fairness (공정성)
  - job 간의 공정이 얼마나 지켜지는지
  - Fair scheduler would treat the two classes equally

System-centric metric

- Utilization : scheduler 에 따라 Utilization
  - 스케줄러 평가의 일반적인 방향
  - Scheduler can maximize processor
- Throughput (처리율)
  - Rate at which requests are completed
- Complexity
  - How difficult is to correctly implement and analyse a scheduling algorithm
  - Implementation complexity impacts how much memory the scheduler will consume to compute its policy. 버그나 결함 발생 증이고, 오버헤드 증임.

우선 work-conserving 에 초점을 맞춤

14.4 General purpose Scheduling policies



$j_{i,k}$ i Task k^{th} job

job	Arrival Time	length
j_1	0	4
j_2	3	6
j_3	4	9
j_4	7	7
j_5	9	2

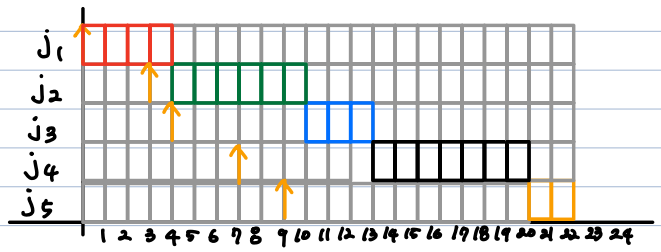
14.4.1 First come First serve (FCFS) (NP)

First come (Earliest Arrival time)

1. Invocation : No job in the system. a new job has been released NON-preemptive
2. policy : jobs are sorted in ascending order by arrival time. Tie job that has been waiting longer has selected.

↑ job Arrival

job	Arrival Time	length
j_1	0	4
j_2	3	6
j_3	4	3
j_4	7	7
j_5	9	2



Average Response time

$R_1 = 4 - 0 = 4$
$R_2 = 10 - 3 = 7$
$R_3 = 13 - 4 = 9$
$R_4 = 20 - 7 = 13$
$R_5 = 22 - 9 = 13$

$$\bar{R} = \frac{1}{5}\sum_{i=1}^{5} R_i = \frac{1}{5}(46) = 9.2$$

Fairness. ?

How fair FCFS is for the shortest job?

USE Slowdown to Reason about fairness.

$$\text{Slow down for } j_5 = \frac{R_5 \leftarrow \text{response time}}{C_5 \leftarrow \text{service time}} = \frac{13}{2} = 6.5$$

Could we Achieve a better average response time

Ex) Swap j_4 and j_5, $R'_4 = 22 - 7 = 15$, $R'_5 = 15 - 9 = 6$ $\bar{R}' = 8.2$

Short coming of FCFS:

once a long job is given control of the processor, all the other jobs need to wait a long time before having the chance to execute, even if their demand is small.

짧은 작업이 너무오래 기다릴수 있어서 Quantum [시간차] 를 측정한 이유?

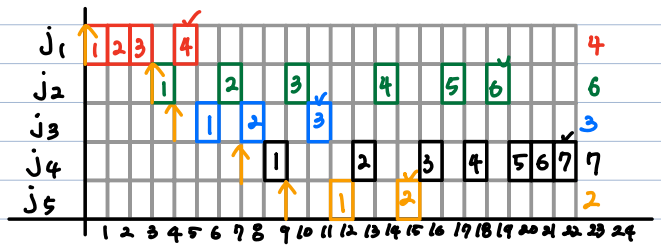
14.4.2 Round Robin (P)

- Invocation : invoked after Expiration of a fixed time interval, quantum, q.
- Policy : the job that has been waiting the longest since the last time a processor quantum was assigned to it is picked next for execution.

↑ job Arrival

q = 1 (1칸씩 돌아간다)

job	Arrival Time	length
j_1	0	4
j_2	3	6
j_3	4	3
j_4	7	7
j_5	9	2



Average Response Time

$R_1 = 5 - 0 = 5$
$R_2 = 19 - 3 = 16$
$R_3 = 11 - 4 = 7$
$R_4 = 22 - 7 = 15$
$R_5 = 15 - 9 = 6$

$\bar{R} = \frac{1}{5}\sum_{i=1}^{5} R_i = \frac{1}{5}(49) = 9.8$ → FCFS 보다 빨리완료! 왜?

Fairness

How fair RR is for the shortest job?

USE Slowdown to Reason about fairness.

Slow down for $j_5 = \frac{R_5 \text{ (response time)}}{C_5 \text{ (service time)}} = \frac{6}{2} = 3$

RR Fairness (3) < FCFS fairness(6.8)

Main Property of RR : use preemptive (NP) to attempt fair assign of processor time to ready tasks. - prevent processor-hungry jobs from taking over the processor for long period of time.

q → 0 으로 갈수록 perfect fairness. q가 커질수록 FCFS 랑 가까워진다. q가 longest job in the system 보다 크면 보통 FCFS 랑 같아진다.

Round Robin : 저부하도, 저오버헤드, 높은 implementation. job을 작은 고정길이로 나눠서 효과적

How about RR

First come first serve

$q \to \infty$ | (non-preemptive) FCFS scheduler

↓

Invent time slack for each Round Robin

$q \to 0$ | perfectly fair CPS ... but overhead?

Overhead Control

Say have measured the Overhead: Ovr

And you want no more than 1% overhead

q | $q \geq 100 \cdot Ovr$

↳ q에서 Ovr의 비율이 1% 보다 적게 유지

Short Coming of RR

- 작업이 할당되면 할당량 (q) 동안 프로세서가 실행, 끝나면 전환
- I/O bound 작업일경우 q가 끝나기전에 완료, 대기시간이 아깝다.
- short job은 inefficient.

↓

14.4.3 Virtual Round Robin → I/O bound를 조금 더 효율적으로 처리 하기 위한 알고리즘

Invocation: Invoked at the expiration of a fixed time interval (q), Also invoked if the Currently running job completes.

policy: the scheduler keeps two queues.

High-Priority Queue

- A task Enter HPQueue
- if a job belongs to that task has completed before the expiration of its assigned quantum
- if a job in HP queue is ready then do this job first

Since these jobs have a high chance to become I/O bound

Low-Priority Queue

내가 아닌 그냥 jobs. (지잡들)

↑ job Arrival

job	Arrival Time	length
j_1	0	5
j_2	0	1
j_3	0	5
j_4	0	5
j_5	0	5

Chapter 15: Length-Aware job Scheduling

Still single resource!, but take account the length of scheduled jobs.
1) estimate, 2) predict the length of a job.

예제 1) 먼저, 먼저나 직원이 오고, 긴지 몰랐다. 3), 기한도 몰랐다.
이제는 Arrival time, length of a job 등과 신청한다. FCFS로부터 시작해보자.

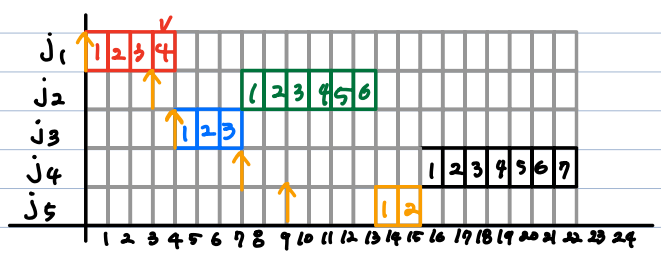
C_i	length of a generic job

15.1.1 Shortest job Next (SJN) FCFS 전략 내건 (NP)

Invocation: Completion of a job (Non-preemptive)
Policy: Read job with the Shortest length is picked next to execute on the processor

↑ job Arrival

job	Arrival Time	length
j_1	0	4
j_2	3	6
j_3	4	3
j_4	7	7
j_5	9	2



Average Response Time

R_1 : 4 - 0 = 4
R_2 : 13 - 3 = 10
R_3 : 7 - 4 = 3
R_4 : 22 - 7 = 15
R_5 : 15 - 9 = 6

$$\bar{R} = \frac{1}{5}\sum_{i=1}^{5} R_i = \frac{1}{5}(38) = 7.6$$

SJN : optimal non-preemptive strategy to minimize average response time

Fairness

Slowdown for shortest job

$j_5 = \frac{R_5}{C_5} = \frac{6}{2} = 3$

Shot Coming of SJN: Starvation:

길이가 짧은 job이 계속들어오면 길이가 긴 job은 영영 실행 못시킬수도 있다.

Blocking

j_2 : 길이가 엄청길다고하면, j_5은 짧음. 비교 대상 job 이라고 해도 j_2 에의해 blocking 되는 상황이다.

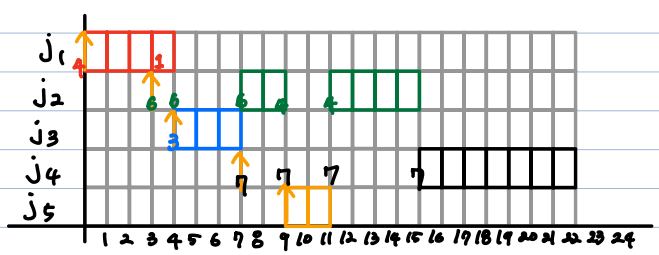
15.1.2 Shortes Remaining Time (SRT)

Invocation: at the completion of the job, or At the arrival of a new job (pre-emptive)

policy : Ready job with the shortest remaining execution time is picked next to execute on the processor

↑ job Arrival

job	Arrival Time	length
j_1	0	4
j_2	3	6
j_3	4	3
j_4	7	7
j_5	9	2



Average Response Time

R_1 : 4 - 0 = 4
R_2 : 15 - 3 = 12
R_3 : 7 - 4 = 3
R_4 : 22 - 7 = 15
R_5 : 11 - 9 = 2

$\bar{R} = \frac{1}{5}\sum_{i=1}^{5} R_i = \frac{1}{5}(36) = 7.2$

Fairness

Slowdown for shortest job

$\therefore \frac{R_5}{C_5} = \frac{2}{2} = 1$

Fairness improved.

j_5 : C_5 = 9 - 4

Blocking : 개선되었다.

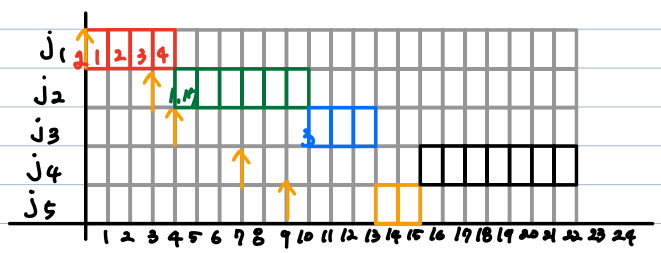
But Starvation is Not fixed.
긴 job의 많은 작업으로 긴 short job만 계속 도착하면 안고쳐진다.

15.1.3 Highest Slowdown Next (HSN) (NP)

Invocation : invoked at the completion of a job (non-preemptive) (NP)
Policy : Ready job with the largest slowdown is picked next to execute on the processor.
At a scheduling event, the slowdown for ready tasks is calculated as the slowdown that the task would have if it was executed next.
↳ 실행했을때의 slow down을 기준으로삼음

↑ job Arrival

job	Arrival Time	length
j_1	0	4
j_2	3	6
j_3	4	3
j_4	7	7
j_5	9	2



Average Response Time

R_1 : 4 - 0 = 4
R_2 : 10 - 3 = 7
R_3 : 13 - 4 = 9
R_4 : 22 - 7 = 15
R_5 : 15 - 9 = 6

$\bar{R} = \frac{1}{5}\sum_{i=1}^{5} R_i = \frac{1}{5}(41) = 8.2$

4에서
j_2 = 10 - 3 = 7 $\frac{7}{6} = 1.17$
j_3 = $\frac{7-4}{1}$ = 3 $\frac{3}{3} = 1$

10에서
j_3 = 13 - 4 = 9 $\frac{9}{3} = 3$ ✓
j_4 = 17 - 7 = 10 $\frac{10}{7} = 1.42$
j_5 = 12 - 9 = 3 $\frac{3}{2} = 1.5$

13에서
j_4 = 20 - 7 = 13 $\frac{13}{7} = 1.85$
j_5 = 15 - 9 = 6 $\frac{6}{2} = 3$.

Fairness
Slowdown for shortest job

$j_5 : \frac{5}{2} = 3$

Slowdown 계산법 — 앞으로 완료까지 걸릴 시간

$$\text{Slowdown} = \frac{t + C_i - a_i}{C_i}$$

t: Scheduling event
C_i: Task length
a_i: Arrival time

HSN 의 특성?

15.2 Estimating Job lengths. 작업 길이를 사용해서 스케줄러의 성능을 향상 시킬수있다.

It can be derived in few ways

1. Static analysis:
   Traditional computer application, job length analyzeize at compile-time or after compilation.
   Static analysis rely on hardware.

2. Offline measurements:
   If the sets of applications that will request execution on a given system/processor is known, it is possible to measure the individual execution time.

3. Online estimation:
   Usually, application produce jobs that exhibit regular patterns in their parameter.
   we can observe and predict the length of subsequent jobs

→ most practical way

15.2.1 Simple Average

Observe $C_{i,k}$ | $C_{1,1}, C_{2,2} \ldots$ for Task T_i, collected | $\bar{C}(n)$ - predicted length of n

length of the job 1
Actual

$$\bar{C}(n) = \frac{1}{n}\sum_{k=1}^{n} C_{i,k}$$

↓

try to keep the each $C_{i,k}$ value. to measure,

it a large number of samples, the equation above would require that we keep in memory the observed value of all the completed n jobs.

A more efficient way of computig the quantity is only keep track of

i) the last average value $\bar{C}(n-1)$; and n

$$\bar{C}(n) = \frac{\bar{C}(n-1) \times (n-1) + C_{i,n}}{n},$$

Problem : "long age" → have the same weight of recent observations.

15.2.2. Averages over a Sliding Window.

window of length w. The last w observations contribute to the current estimation

$$\bar{C}(n) = \frac{1}{w}\sum_{k=n-w+1}^{n} C_{i,k}$$

마지막 w 만 생각

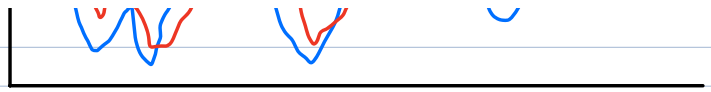
a new observation is available,

$$\bar{C}(n) = \bar{C}(n-1) - \frac{C_{i,n-w}}{w} + \frac{C_{i,n}}{w}$$

↓

새거 = 전거 → 마지막 + 새로들어온것

Sliding window Average



Actual — w=3 — w=10 — w=15

w 값이 작을수록 : 가장최근것만 크게 반영함. 전체적인 경향 비슷

15.2.3 Exponentially weighted Moving Averages (EWMA)

지수 가중 이동 평균

$\alpha \in [0, 1]$ $0 < \alpha \leq 1$ 인 weight / keep track of the system's past but recent ones

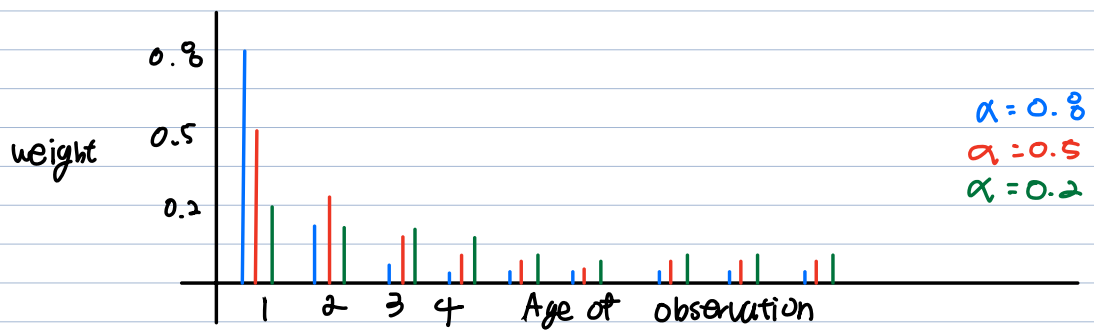
$$\bar{C}(n) = \alpha C_{i,n} + (1-\alpha)\,\bar{C}(n-1)$$

why this called exponential?

$$\begin{aligned}\bar{C}(n) &= \alpha C_{i,n} + (1-\alpha)[\alpha (C_{i,n-1} + (1-\alpha)\bar{C}(n-2)] \\ &= \alpha [C_{i,n} + (1-\alpha) C_{i,n-1}] + (1-\alpha)^2 \bar{C}(n-2)\end{aligned}$$

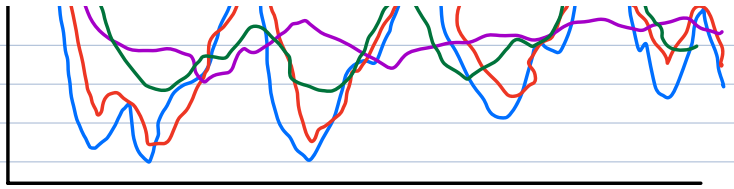
After k steps,

$$\bar{C}(n) = \alpha [C_{i,n} + (1-\alpha) C_{i,n-1} + (1-\alpha)^2 C_{i,n-2} + \cdots + (1-\alpha)^k C_{i,n-k}] + (1-\alpha)^{k+1} \bar{C}(n-(k+1))$$



Higher α follow better the observations, but are more influenced by temporary fluctuations
Smaller α yield to predictions that leg behind the real values

α 가 높으면 더 좋은 최근 observation, but 특히 outlier에 의 반응
α 가 작으면 더 오래전 과거까지 반영



Actual — $\alpha=0.8$ — $\alpha=0.5$ — $\alpha=0.2$

- Priority Based Scheduling

Priority
FCFS: Inversely proportional to arrival time, higher the arrival timestamp, the lower priority
SJN: Inversely proportional to C_i
HSN: priority of j_i is directly proportional to the $\frac{t+c_i-a_i}{C_i}$

Chapter 16 State-Aware Resource Management

resource strategis for resources are stateful.

CPU is stateless resource because the time to perform context-switch / scheduling in between jobs is typically almost constant and much smaller than the length of jobs.

Stateful : 1) Traditional magnetic disk, 2) DRAM memory subsystem

16.1.1 System-centric Metrics

In context of stateless resources, utilization depends on arrival rate & process time.
Utilization in stateless only depends on supply and demands

How about stateful Resources?

Stateless
i) fetch new request
ii) serve request, repeat

2) fetch new request
ii) recognize resource (overhead) → stateful utilization 에서 고려해야한다.
iii) serve request

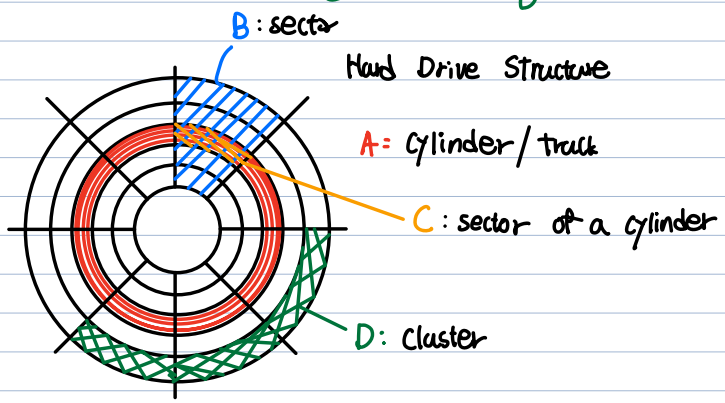
16.1.2 Process-centric Metrics

use Slowdown for fairness metrics

For a job j_i, slowdown $\frac{R_i}{C_i}$

Key: Schedule Reordering

16.2 Hard disk Scheduling



- seed delay : Find Right Cylinedor
- Rotational delay : Find Right Sector.
- Transfer time : Read / write time in the right position.

16.2.1 FCFS Scheduling

In order of arrival

Implementation ✓

Simple

Fairness ✓

No re-ordering, fairness is guaranteed

Head Movement ✗

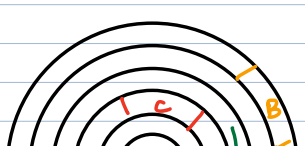
when a and b requests are coming, but they are far. this case, efficiency is low

16.2.2 Shortest Scan First - SSF (가장 짧은 스캔우선)

Requests are kept in Queue, Then the position of the head after the last processed request is considered, and from the queue SSF extracts the pending request for the disk block lints closer to the disk head position

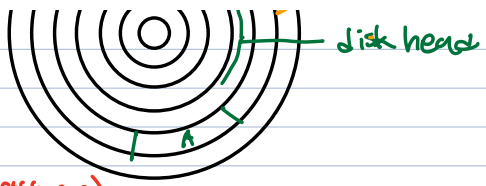
Implementation Halt

geometry has to be known., keep track / predict of



C → A → B

the position of the disk head to find closer.



Fairness ✗

Unfair, as a large burst of spatially co-located requests can delay requests for far-away blocks indefinitely.
(공간적으로 인접한 요청들이 대량으로 발생하면 Stuck, (head stickness)
다소 불공평함)

Head Movement ✓ but could be better

- Tries to minimize head movene
- But related to arrival time. 도착 시간이 차이가 크게 나면 FCFS랑 비슷해진다.

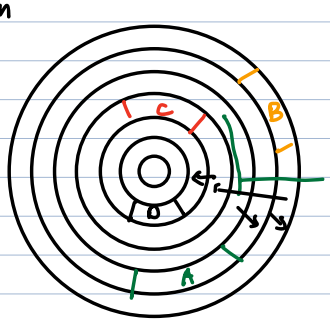
a1, b1, a2, b2

16.2.3 Scan Scheduling

Elevator Algorithm | Constraint on the direction

Pick the request that is closer in space to the position of the head, but only one direction
Ex) 29, 27, 25, 40 , At 30 up↑ → 40 → 29, 27, 25
down ↓

Implementation → Save as SSF Half
Need to keep track of current position,
predict / compute the desired head location



A → B → → C → D

A B C D →

Fairness : X Still need to fixed
Monopolize 를 막기 위해서는, 같은 위치에
갈 때에도 Head Stickness possible

16.2.4 C-SCAN scheduling

SCAN의 가장 큰 문제는 0에서 갈 때도 다음 갈 때도 0에 돌아옴. worst case
0 → N, N → 0

limiting the direction of the head always on one direction
Request 가 없으면 0으로 돌아감

Other than mitigating the worst-case, identical to SCAN

16.2.5 Complete Fair Queuing Scheduler - CFQ (Linux)

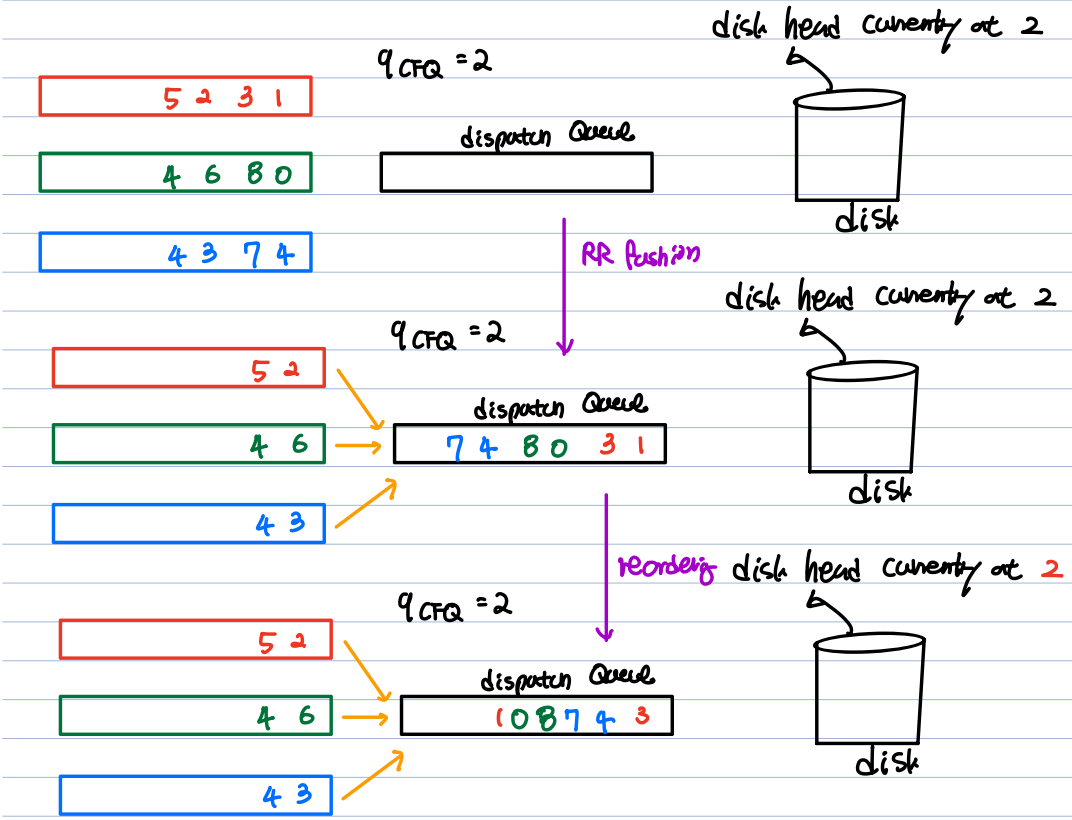
Distinguished between synchronous requests and asynchronous requests
(requests need to be satisfied) (Can be done later without impacting performance)

Synchronous requests

2 stages

1) One queue, q (Quantum) RR fashion, if empty then wait and take it.

2) Reorder, minimize disk head movement



Implementation ✗

Not Easy, maintain a set of perprocess Queue, a dispatch Queue, and perform re-ordering

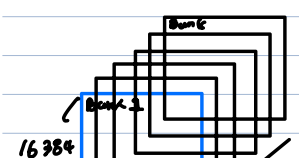
Fairness ✓

Round Robin 방식으로 monopolize 방지. reordering 할 때 fairness 가 깨지 않도록 제한 (qCFQ · N (# of tasks with pending requests)

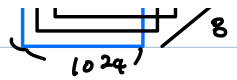
Head Movement ✓

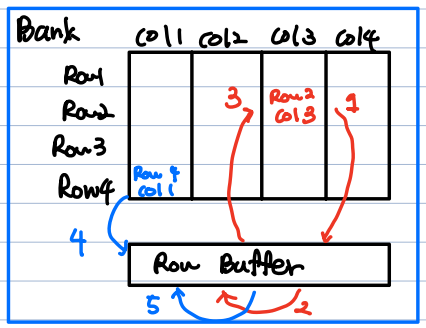
- comprises minimization and fairness.
- In real words if per-application queue has less than qCFQ then CFQ non-work-conserving

6.3 DRAM Subsystem (Dynamic Random Access Memory)



Each bank holds 16MB





At t=0, row buffer is empty

Operations:
1. Activate Row2
2. Read/write Col 3 in Row 2
3. Precharge Row 2
4. Activate Row 4
5. Read/write col 1 in Row4

Type A → chip x, bank y, row 2, col 3

FR, FC, FS

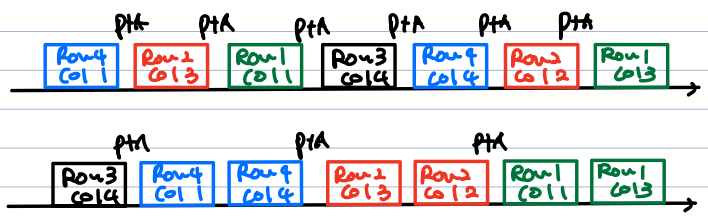
16.3.1 First Ready, First come First served - FR-FCFS

A different queue is defined for each bank. FR-FC-FS keep track of which one is the row currently in the row buffer.

Implementation (in hardware) ✓
Fair easy to implement.
keep track of currently open Row

Fairness ✗
Starvation could happen

Memory throughput ✓
maximum throughput for outstanding requests
Least Amount of PtA operations
(Precharge + Activ)



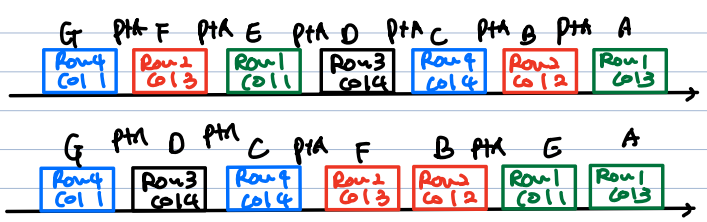
16.3.2 Capped First Ready, First come First served - FR FCFS - Capped

Cap k, on reordering. No more k younger row-hits can be re-ordered (effectively jump the line) before older requests.

k=2 # of Cut

Implementation (in hardware) Half
Trickier to implement / track of reordering and suffering k

Fairness ✓



Situation Now fixed

Memory throughput ✓

maximum throughput for outstanding requests

Least Amount of P+A operations

(Precharge + Active)

Chapter 17: Real-Time Scheduling

17.1 Cyber-Physical Systems

Computation tightly with the physical world (CPS)

ex) Thermostat in home

washing machine check temperature

Charlie card reader

Airbag system. → malefunct could lead to catatrosphic

절대로 잘못 작동하면 안된다.

17.2 Real Time Applications

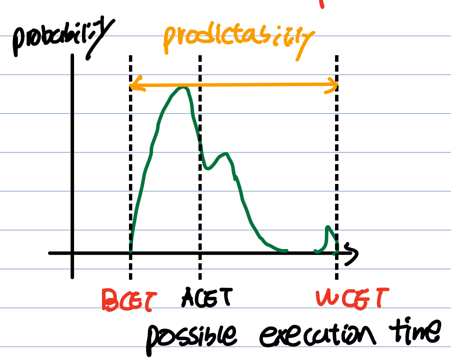
Hard-Real Time

☆Deadlines☆

Absolutely Not okay to miss a deadline for a hard real-time task.

predictability

Time difference in between (A) best possible case and (B) worst-case behavior is small



BCET : best-case Execution time

WCET : worst-case Execution Time

ACET : Average-case Execution time

It's okay to trade in some performance to bring BCET closer to the WCET.

When analyzing real-time system, always reason in terms of WCET

17.2.2 Deadlines, Tardiness, and Utility

Soft real-time system : Utility gradually decreases.
Hard real-time system : Utility immediately 0 after deadline or —

17.3 Real-Time Scheduling

In Real-time systems, it does not matter how long it will take for a job to complete as long as it is before deadlines.

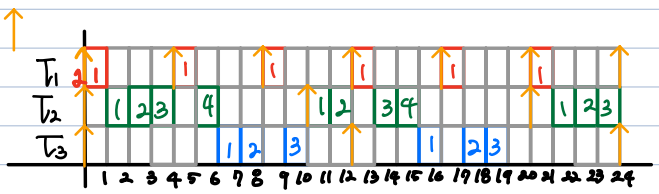
Performance: Maximum work to process not passing deadline.
Capped work: Utilization bound, Utilization이 커질수록, response time은 어차피 커진다. /cap.

T_i	New job $j_{i,k}$ release period
C_i	WCET of a job Task T_i
D_i	deadline of a job T_i, it Job is released at t, then deadline is $t + D_i$

17.4 Static Priority Scheduling

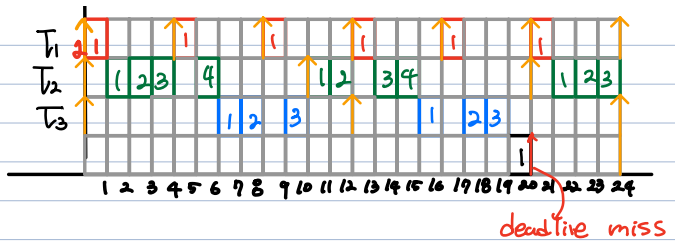
Rate Monotonic Scheduling (RM)
Priority : Inversely proportional to the length
Invocation : completion of job or when a new job is released (P)
Policy : Ready job with the shortest period (higher priority) executes next on the processor.
T_i 주기의 길이

	T_i	C_i
T_1	4	1
T_2	10	4
T_3	12	3



If we add one more task T_4 $T_i = 20$, $C_i = 2$.

	T_i	C_i
T_1	4	1
T_2	10	4
T_3	12	3
T_4	20	2



Not schedulable under RM

RM Schedulability Test

Tell whether or not a Task-set is schedulable by just looking at it.

$$\sum_{i=1}^{m} \frac{C_i}{T_i} \leq m\left(2^{\frac{1}{m}} - 1\right)$$

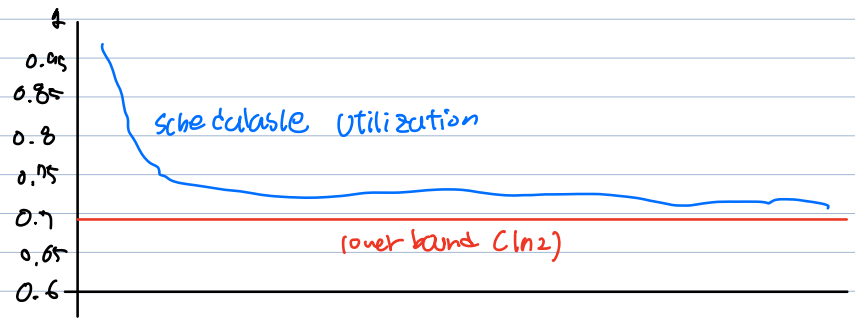
C_i : WCET

T_i : Arrival period

m : # of Tasks

3개의 Task가 있을경우, $\frac{1}{4} + \frac{4}{10} + \frac{3}{12} = \frac{1}{4} + \frac{2}{5} + \frac{1}{4} = 0.9 \geq 3(2^{\frac{1}{3}} - 1) = 0.78$ 방정식을 성립하지 않는다.
이건 충분조건이지 필요조건이 아니다. 정확한건 그림을 그려봐야 알 수 있다.

$m = \infty$ $\quad m\left(2^{\frac{1}{m}} - 2\right) \rightarrow \ln(2) = 0.69$ 이용률이 69% 이하면 정확히 RM 하에서



17.5 Dynamic Priority Scheduling

또다른 방식은 Priority 를 도착할때마다 다시계산.

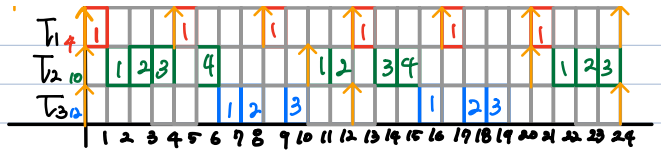
Earliest Deadline First (EDF)

How close to deadline First

Invocation : preemptive (P)

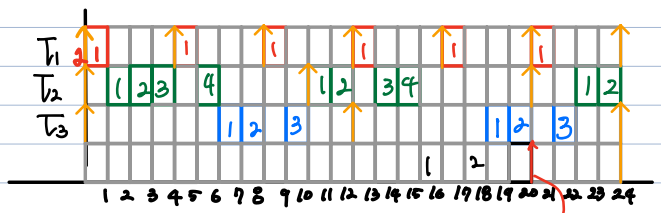
Policy : Ready job with the closest deadline (highest priority) executes next on processor.

	T_i	C_i
T_1	4	1
T_2	10	4
T_3	12	3



If we add one more task T_4 $T_i = 20$, $C_i = 2$.

	T_i	C_i
T_1	4	1
T_2	10	4
T_3	12	3
T_4	20	2



남은 deadline을 계속 계산해나간다.

deadline miss

Schedulability Test

$$\sum_{i=1}^{m} \frac{C_i}{T_i} \leq 1$$

Able to schedule a task-set with 100% utilization

17.6 Practical Scheduling

과연 EDF가 RM보다 항상 좋을까?

1. 복잡성
2. 실제 시스템의 프로세서 사용률
3. utilization이 100% 일때 WCET가 조금으로 달라진 EDF에서 더 악화된다. (RM) 도영향을 덜 받는다.

RM ≤ 69% | EDF ≤ 100%

What would you choose for your hard real-time system?

Obviously.. RM =

- complexity
- workload composition 30%로 사용하게 허용을 수 있지만, 69로 넘어오면 실제로 스케줄
- Robustness to timing, 가능한 최악의, non-critical이 사용할 수 있다.

실제로는 RM을 사용한다.

Chapter 18 : Multi-Resource Scheduling

Multiple Resources scheduled

18.1 Multiple Resources

Global multiprocessor scheduling
- job sent indistinguishably to any of the available cores

Partitioned multi-processor scheduling
- some jobs need to be done at certain processor

18.2 Multiple parallel Resources

1) Compute the priority from time to time of the ready job
2) Decide which processor each job executes

18.2.1 Partitioned Scheduling

N/M/M/1

Global Multiprocessor scheduling	Partitioned Multi-processor Scheduling
N processors/ servers	N processors / servers
1 ready queue	N Ready queues
work-conserving	Work-conserving

Partitioned

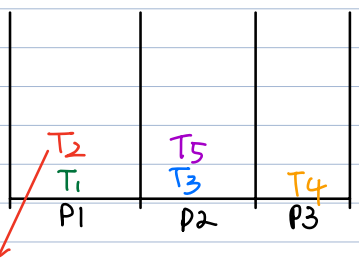
1) Offline, consider parameter and assign tasks to processors | 어려고 보내지 제작
2) Apply Single-core scheduling on each core independently | 보내면 processor가 작업

in RT, schedulable is the most important!!

RM-based partitioning strategy

1) Try to assign a Task, 2) check Schedulability

Task	T_i	C_i	U_i
T_1	10	3	0.3
T_2	15	6	0.4
T_3	20	6	0.3
T_4	32	18	0.56
T_5	8	2	0.25



$T_1 + T_2 = 0.3 + 0.4 \geq 0.69$ NO! 그치만 이건 충분조건이지 필수조건이 아니므로
$2(2^{\frac{1}{2}} - 1) = 0.83$ Yes!

RM-FF (First Fit)

For the whole set: $U = \sum_{i=1}^{m} \frac{C_i}{T_i}$

A Task-set is Schedulable under RM-ff if $U \le N \cdot (\sqrt{2}-1)$ (N ↓ # processor)

Tack	T_i	C_i	U_i
T_1	10	3	0.3
T_2	15	6	0.4
T_3	20	6	0.3
T_4	32	18	0.56
T_5	8	2	0.25

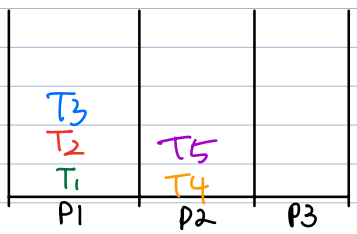
$U = 1.81 \;/\; \le 3 \cdot (\sqrt{2}-1) = 1.24$ 안되는거지.

$1.81 \le 1.24$

EDF - FF

For the whole set: $U = \sum_{i=1}^{m} \frac{C_i}{T_i}$

A Task-set is Schedulable under EDF-ff if $U \le \frac{\beta \cdot N + 1}{\beta + 1}$



$\beta = \left\lfloor \frac{1}{\max \frac{C_i}{T_i}} \right\rfloor$: floor 함수야. 정수

Tack	T_i	C_i	U_i
T_1	10	3	0.3
T_2	15	6	0.4
T_3	20	6	0.3
T_4	32	18	0.56
T_5	8	2	0.25

$U = 1.81 \;/\; \le 2$ Schedulable

EDF-FF: $\frac{2 \cdot 3 + 1}{1+2} = \frac{4}{2} = 2$

$\beta = \left\lfloor \frac{1}{0.56} \right\rfloor = \lfloor 1.78.. \rfloor = 1$

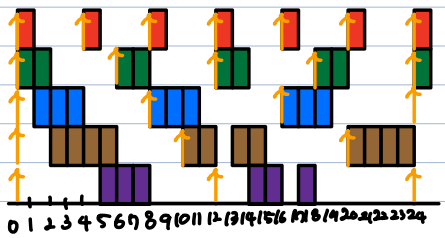
보통 partitioning scheduling 중 최고.

18.2.2 Global Scheduling

Try 해보는 수밖에 없다.

1) Offline, rank all the tasks / jobs in the shared ready queue
2) Select top N to run on the N available processors.

Task	T_i	C_i	U_i
T_1	4	1	0.25
T_2	6	2	0.33
T_3	8	3	0.38
T_4	10	4	0.4
T_5	12	3	0.25



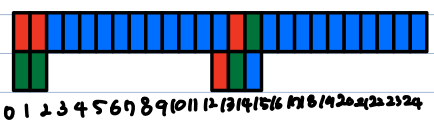
lower job ID wins

$U \approx 1.6$

Task	T_i	C_i	U_i
T_1	12	2	0.17
T_2	12	2	0.17
T_3	13	12	0.92

lower job ID wins

$U \approx 1.2$



Task-set is Not Schedulable on 2 processor under Global EDF.

Some low-Utilization task-sets are Not Schedulable, even with large N

The Dhall's Effect.

Chapter 19 : Mutual Exclusion

19.1 Data Races

Multiprogramming
No real concurrency : interleaved execution at different resources.

Multiprocessing
real concurrency AND interleaved execution

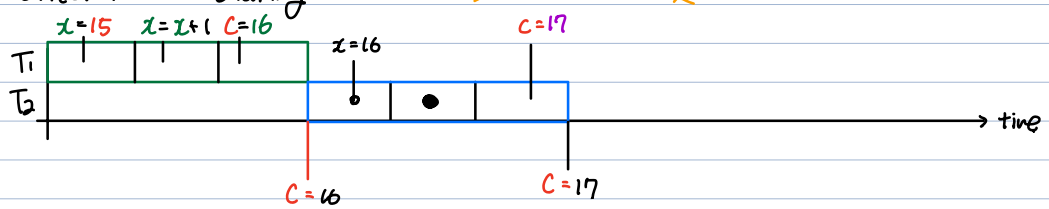
Distributed processing
All of the above, but with possibly large communication

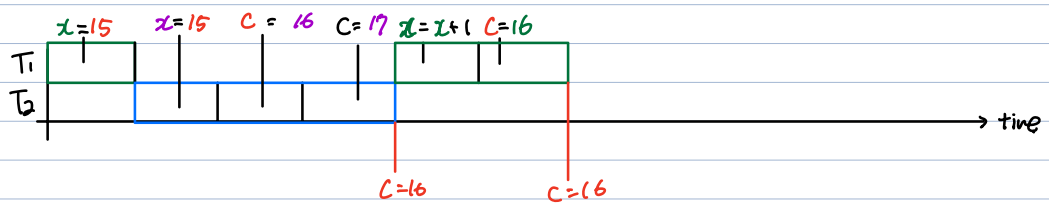
Concurrency
1) might need to share resources (ex) I/O devices, processor, interface
2) might need to share data a memory data structure, a file

```
void main() {
    int x
    x = c;
    x = x+1;
    c = x;
}
```

c is shared variable c=15

Uncontrolled sharing 내가 생각하는 것



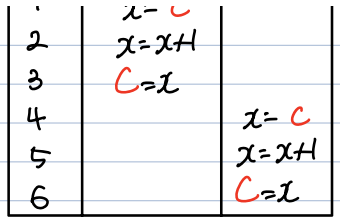


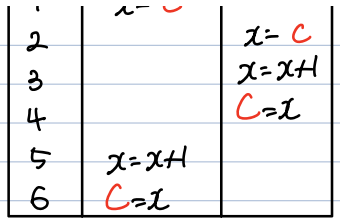
Focus on Process Interleaving :

step	Task 1	Task 2
1		

step	Task 1	Task 2
1		

"Corectness" is wrong





"Data Race" — Uncontrolled access by two or more concurrent processes to a shared variable.

19.2 Critical Sections of Code
- when a process manipulates shared resources / data, it is said to be in a Critical section

19.3 Mutual Exclusion (상호배제 문제)
Sharing Resource R is safe if at any time only one process is in a critical section that manipulates R

Assumptions about the processing Environment
- Progress : None of process stops. It moves forward.
- Unknown Rate of progress: arbitary speeds
- Concurency exists : It could be any concurency
- Unknown Interleaving : Any arbitary way of interleaving
- Serialized Memory : Allow for shared-memory.

Generic Code

```
repeat:
    reminder section
    [Entry Section]
    Critical section!
    [EXIT Section]
    reminder section
forever
```

Desirable properties

Mutual Exclusion
at most one process in a given critical section CS

Isolation
only processes tying to enter a CS get to decide which one enters

Progress (no deadlocks)
if no progress in CS and processes want to enter, then someone will

No starvation

no process waits an infinite amount of time before entering a CS

1)

2 Party Mutual Exclusion

```
Process i                    ← index of the current process
  repeat:
    reminder section
    [Entry Section]
    Critical section!
    [EXIT Section]
    reminder section
  forever
```

Taking Turns

```
Process i
  repeat:
    reminder section
    while (turn != i);       ← busy-loop until it's process i's turn
    Critical section!
    turn = j;                ← once done, give turn to process j
    reminder section
  forever
```

아니면 스킵함요

Isolation is Not satisfied

✱ Mutual Exclusion을 만족, when P_j is not in used, P_i도 진입해야 함에도 불구하고 j가 진입해서 switch 해주어야하는 문제점.

Attempt 2 : Signal Use of Critical section

Sharing Signal (flag) that I'm in the Critical section

Locked Variable

initialized to False

```
Process i
  repeat:
    while (locked);
    locked = true;
    Critical section!
    locked = false;
  forever
```

step	Task 1	Task 2
1	while (locked);	
2		while (locked);
3		locked = true;
4		Critical section!
5	locked = true;	
6	Critical section!	

Mutual Exclusion is Not satisfied

Task 1 and 2 can enter into locked at the same time since locked value is

yuuuy false.

To fix this, use one such flag per process

Each Flag for processes

```
Global variables
var flag[i] = false;
var flag[j] = false;

Process i
    repeat:
        reminder section
        while (flag[j] == true);
        flag[i] = true;
        Critical section!
        flag[i] = false;
        reminder section
forever
```

NO, Tasks can entry to Critical section after read each others flag and before change their flag

step	Task 1	Task 2
1	while (flag[j]==true);	
2		while (flag[i]==true);
3	flag[i]= true;	
4		flag[j] = True;
5		Critical section!
6	Critical section!	

Attempt 3 : Signal Intention to use Critical section

The mutual Exclusion is invalid due to check others flag before setting my flag.

Intent my flag before read

```
Global variables
var flag[i] = false;
var flag[j] = false;

Process i
    repeat:
        reminder section
        flag[i] = true;
        while (flag[j] == true);
        Critical section!
        flag[i] = false;
        reminder section
forever
```

Might Enter a DEADLOCKS

step	Task 1	Task 2
1	flag[i]= true;	
2		flag[j] = True;
3	while (flag[j]==true);	
4		while (flag[i]==true);
5	while (flag[j]==true);	
6	.	while (flag[i]==true);

Attempt 4 : Intermittently, Signal Intention to use Critical section
when process is realizing that my process is not able to proceed, then set my flag free.

set my flag free

```
Global variables
var flag[i] = false;
var flag[j] = false;

Process i
   repeat:
      reminder section
      flag[i] = true;
      while (flag[j] == true){
         flag[i] = false;
         delay();
         flag[i] = true;
      }
      Critical section!
      flag[i] = false;
      reminder section
   forever
```

Still NO DEADLOCK
NO progress

step	Task 1	Task 2
1	flag[i] = true;	
2		flag[j] = True;
3	while (flag[j]==true);	
4		while (flag[i]==true);
5	flag[i] = false;	
6		flag[j] = false;
7	delay();	
8		delay();
...	...	...
k+1	flag[i] = true;	flag[i] = true;
k+2		
k+3	while (flag[j]==true);	while (flag[j]==true);
k+4		
k+4	flag[i] = false;	flag[i] = false;
k+5		
k+6	delay();	delay();

Attempt 5 : Intermittently and Assertively, Signal Intention to use Critical section
previous was almost correct, but still possible for deadlock

one possibility is giving higher priorities. higher priority does not reset flag

Still Not fair → could lead to starvation, bound waiting correctness criterion.

Dekker's Algorithm : Intermittently but Fairly, Signal Intention to use Critical section.
대칭을 깨는 것을 선택, 한쪽에만 대칭을 깨고 싶다는 생각

Turn 과 flag

Dekker Algorithm

```
Global variables
var flag[i] = false;
```

```
var flag[i] = false;
var flag[j] = false;
var turn = i;
Process i
  repeat:
    reminder section
    flag[i] = True;
    while (flag[j] == true){
      if(turn == j){
        flag[i] = false;
        while (turn == j);
        flag[i] = True;
      }
    }
    Critical section!
    turn = j;
    flag[i] = false;
    reminder section
forever
```

Turn이 너꺼여도 flag[j]가 [illegible] 이상 [illegible]

with Turn set to j, it's Task is turn to back off!

After CS, release resorce and switch turn

Got All 4. mutual Exclusion
isolation
progress
No-starvation

How about fairness.

peterson's Algorithm

which is also fair

```
process i
repeat:
  flag[i] = true;
  turn=j;           → 우선권 양보
  while ( flag[j] && turn == j);
  Critical section!  → j가 하도록
  flag[i] = false;
forever
```

차이점: CS에 들어갈 수 있는지 확인전에, 우선권을 양보.

Mutual Exclusion: Lock / Unlock

```
process i
repeat:
  flag[i] = true;
  turn=j;           → 우선권 양보
```

Lock

```
void peterson_lock (bool (&?) flag, bool &turn)
{
  flag[i] = true;   → 내가 먼저 Signal (intent)
  turn = j;
  while ( flag[j] && turn == j ) {};
}
```

```
while ( flag[j] && turn == j);
Critical section!   ↘ j가 하는 중
flag[i] = false;
forever
```

Unlock

```
void peterson_unlock (bool [2] flag)
{
  flag[i] = false;
}
```

In reality Out of Order (OOO) for optimization

Chapter 20: Hardware Assisted Synchronization

Out of order : 실제로는 서로 다른 순서로 진행된다.

First hardware assistance

Instruction 1;	Instruction 3;
Instruction 2;	Instruction 1;
Instruction 3;	Instruction 4;
Instruction 4;	Instruction 2;
barrier ()	barrier ()
Instruction 5;	Instruction 7;
Instruction 6;	Instruction 5;
Instruction 7;	Instruction 8;
Instruction 8;	Instruction 6;

(Instructions 3, 1, 4, 2 in the right column: } out of order)

Due to Out of Orders

20.1 Hardware Assisted Mutual Exclusion

There are two task :

i) read / check of a flag
ii) write / set the flag

} → make it one Atomic) do it together at once.

```
bool test_and_set (bool & flag)   → 사용하고 있는지 확인하는것.
{
   if ( flag == 0 ) {     ↗ 사용안하고 있을때
     flag = 1;
     return true;
   } else {
     return false;
   }
```

Hardware provides

Mutual Exclusion

```
void test_set_lock (bool & flag)
{
  while (test_and_set (flag) == false);
}

void test_set_unlock ( bool & flag)
{
```

```
    flg = 0;
}
```

20.2 Spinlocks

```
void acquire (bool& flg)
{
  while (test_and_set (flg) == true);
  barrier(); → critical section에 들어가기 전 모든 일을 끝내고 들어가기
}

void require (bool& flg)
{
  barrier(); → 나가기 전에 일을 다 끝내고 나감
  flg = 0;
}
```

```
/* some process */

Repeat;
    acquire (flag);
    critical section
    release (flg)
    reminder section
forever
```

i) Using Busy wait so, it could lead to process waste.
2) For this reason, used for short period of CS
3) barrier() is critical for OOO.

20.3 Semaphores.

In order to fix processor busy wait waste from spinlock, [?] the process to put in queue

i) Attempt to acquire lock
ii) succeeds, continue with execution
iii) fails, blocked and enter a lock-specific queue

1 Count : int counter that captures the current status of semaphore;
2. queue : queue of processor blocked on the resources / critical section for which the semaphore is being used.

- Count ≥ 0 : How many more tasks are allowed for concurrent usage.
- Negative value : # of tasks waiting in the queue.
- queue : Reference to tasks blocks, Another job is released then unblock waiting Task

Definition

```
class semaphore {
  int count;
  queue queue;

  bool flag;
```

```
void wait (semaphore &sem) {
    acquire (sem.flag);
    sem.count --;
    if (sem.count < 0) {
      sem.queue.enqueue (curr_proc);
```

```
}
```

```
        # block process & unlock
        block( curr_proc, sem.flag);
    } else {
        release(sem.flag);
    }
}
```

```
void Signal(semphore & sem)
    acquire(sem.flag);
    sem.count ++;
    if (sem.count <= 0) {
        sem.queue.dequeue();
        release(sem.flag)
        onblock(curr_proc);
    } else {
        release(sem.flag);
    }
}
```

verify to use queue still blocked...

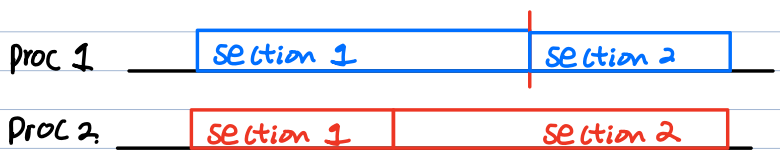
Synchronization Problem 1.

Simple waiting

Proc 1's section 2 should only be run if Proc 2's section 1 is completed.

1.

2.



Initialization:
semaphore synch := 0

Proc 1
repeat
    section 1
    wait(synch)

signal

Proc 2
repeat
    section 1
    signal(synch)

wait (synch)
section 2
forever

signal (synch)
section 1
forever

Dating

1.

2.

Initialization:
semaphore synch1 := 0
semaphore synch2 = 0

proc 1
repeat
section 1
Signal (synch 1)
wait (synch2)
section 2
forever

proc 2
repeat
section 1
Signal (synch2)
wait (synch1)
section 2
forever