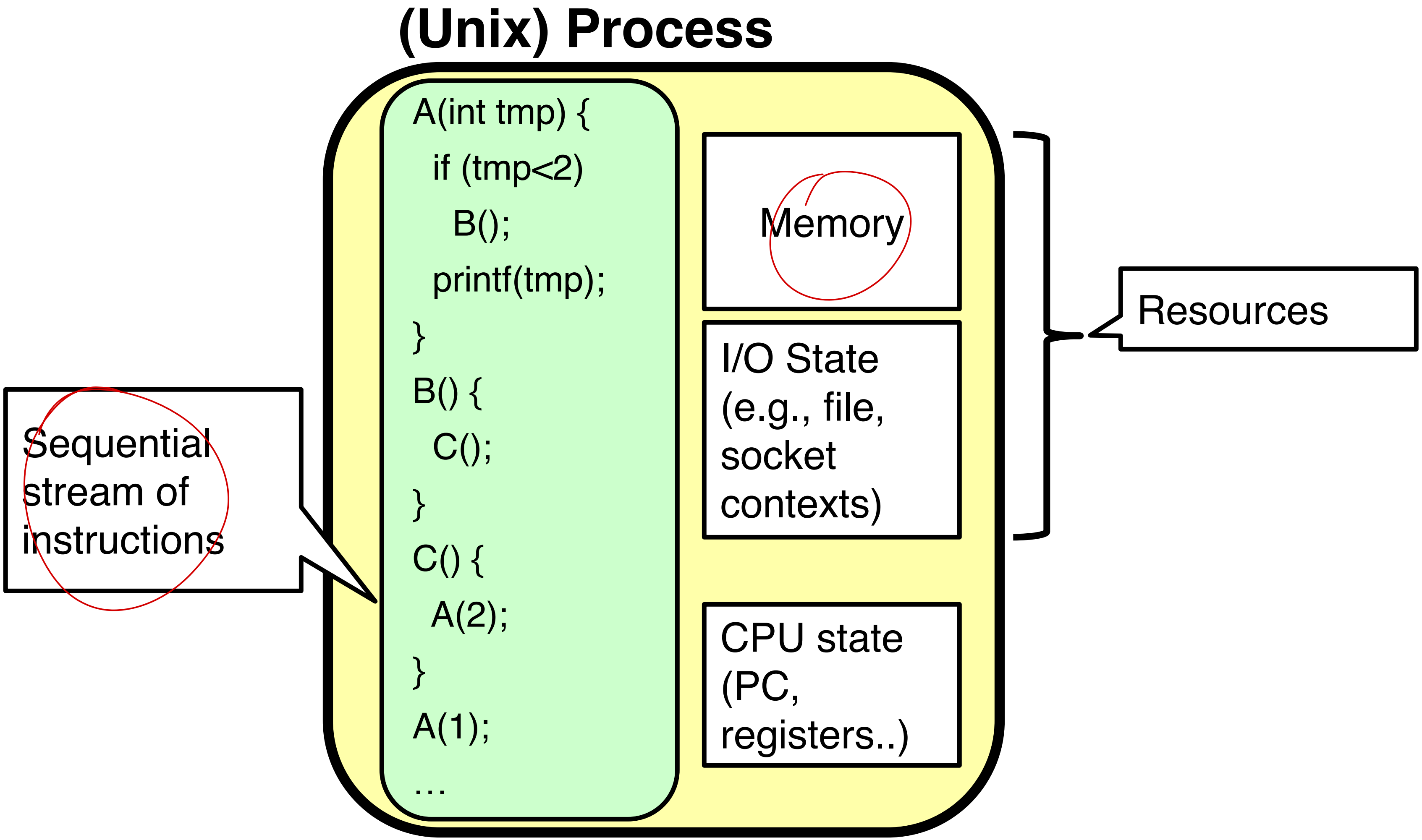
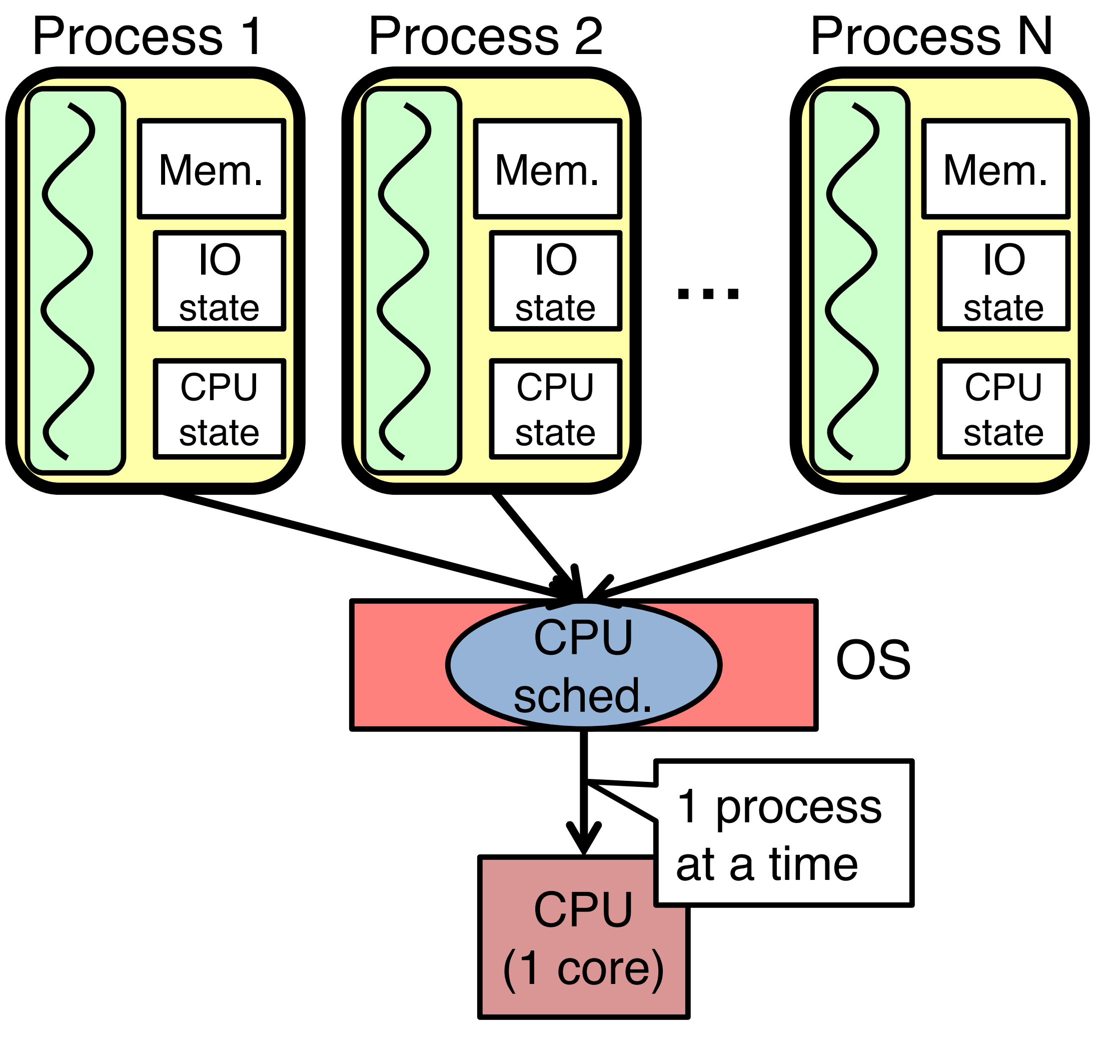
Scheduling

Instructor: Youngjin Kwon

Refine Process



Processes



Process 1
Process 2
Process N
Mem.
IO state
CPU state
Mem.
IO state
CPU state
...
Mem.
IO state
CPU state
CPU sched.
OS
1 process at a time
CPU (1 core)

Main Points

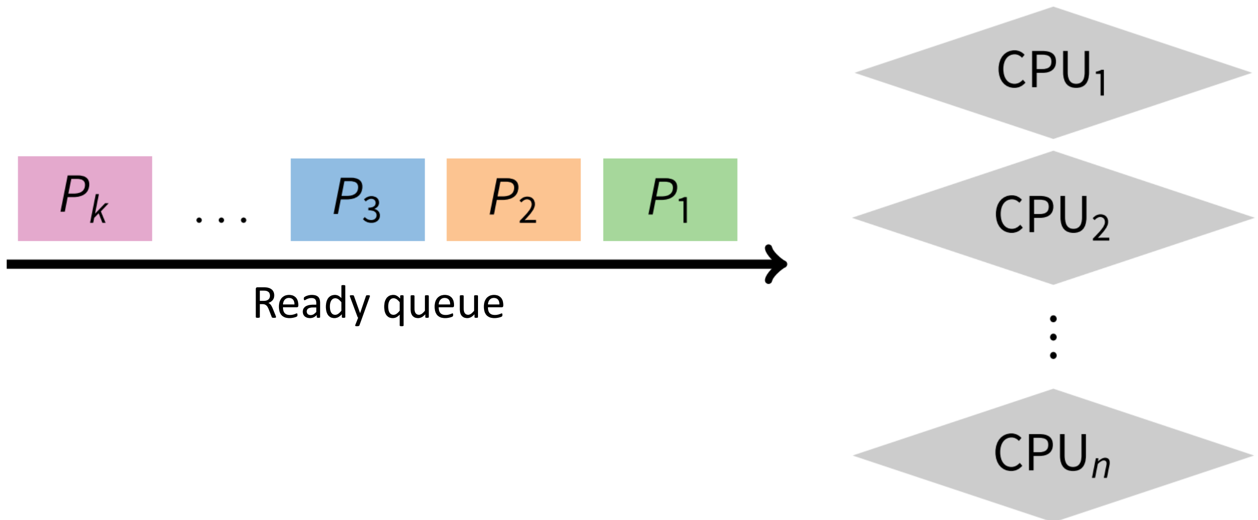
- **Scheduling policy**: what to do next, when there are multiple tasks ready to run
  - Or multiple packets to send, or web requests to serve, or …
- Uniprocessor policies
  - FIFO, round robin, optimal
  - multilevel feedback as approximation of optimal
- Multiprocessor policies
  - Affinity scheduling, gang scheduling

Example

- You manage a web site, that suddenly becomes wildly popular. Do you?
  - Buy more hardware? more processor
  - Turn away some users? Which ones? evict process
  - Implement a different scheduling policy?

efficient policy

Design: scheduling problem

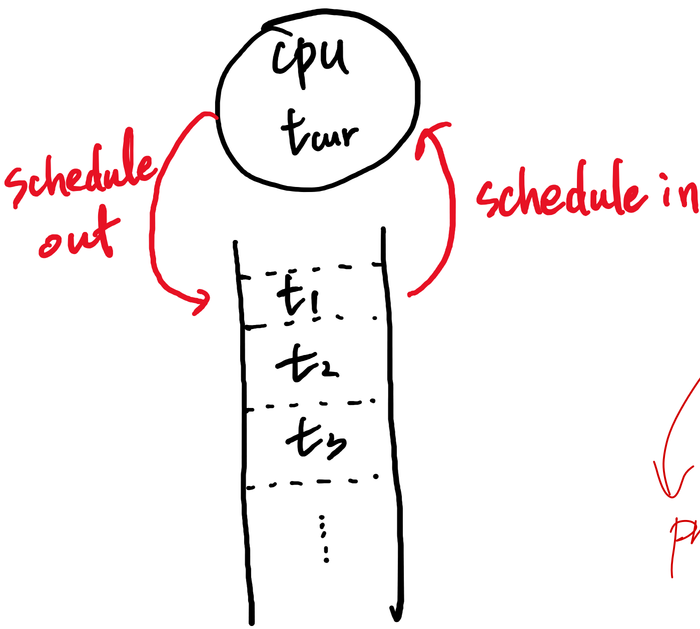


- Scheduling algorithm
  - takes a workload as input
  - decides which tasks to do first
  - Performance metric (throughput, latency) as output

Definitions

- Task/Job
  - User request: e.g., mouse click, web request, shell command, …
- Workload
  - Set of tasks for system to perform
- Overhead
  - How much extra work is done by the scheduler?
- Fairness
  - How equal is the performance received by different users?
- Predictability
  - How consistent is the performance over time?

Scheduler concept



What scheduler does?

- Pick a task from run queue according to scheduler algorithm
- Kick out the running task from CPU
- Make the selected task run in CPU

preempt

Scheduler design choice

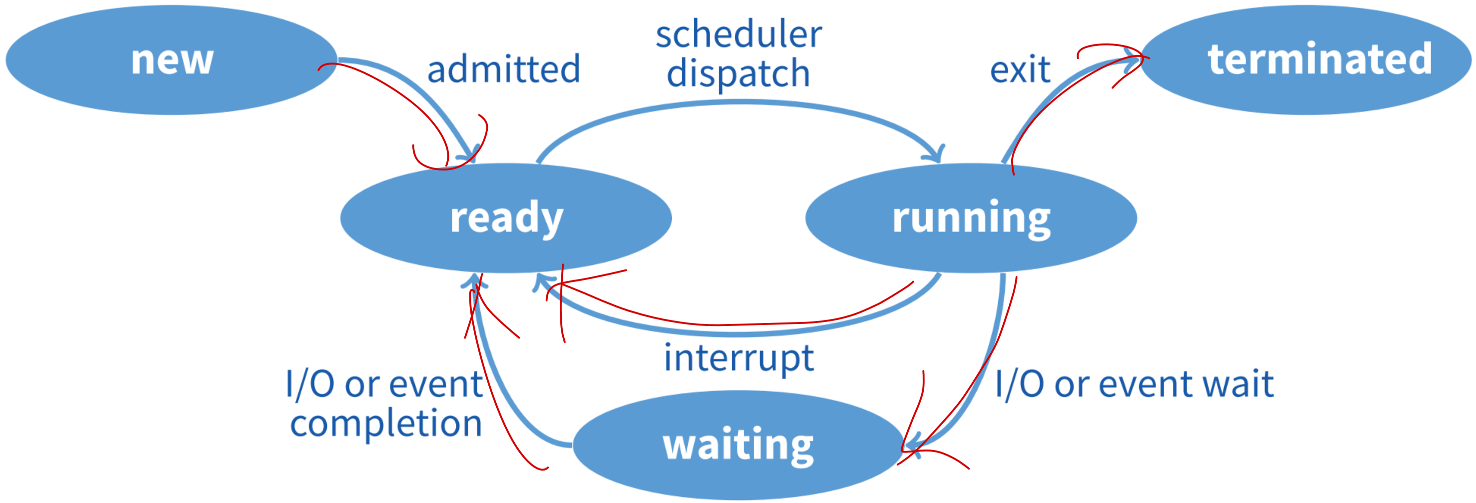
- Preemptive scheduler
  - If we can take resources away from a running task

thread가 레디상태에 있을때 CPU가 돌고 있음.

- Work-conserving
  - Resource is used whenever there is a task to run
  - When is non work-conserving scheduler useful?

→ Network system waiting for full packets to arrive

When does OS invoke scheduler?



Preemptive scheduler:

1. Waiting → Ready
2. Running → Waiting
3. Running → Ready
4. New/waiting → Ready
5. Exit

Non-preemptive scheduler:

New → Ready

exit

Scheduler performance metric

- Throughput
  - How many tasks can be done per unit of time?
  - # of jobs / time
- Turnaround time
  - How long does a task take to complete?
  - T_{finish} - $T_{arrival}$
- Response time
  - Time from request to "first" response
  - $T_{response}$ - $T_{arrival}$
- Waiting time
  - Waiting time of a task = $\sum$ Time spent in ready & wait states
  - Average waiting time = Avg. (waiting time of tasks in system)

Contents

- Uniprocessor policies
  - FIFO, round robin, optimal
  - multilevel feedback as approximation of optimal
- Multiprocessor policies
  - Affinity scheduling, gang scheduling

First In First Out (FIFO)

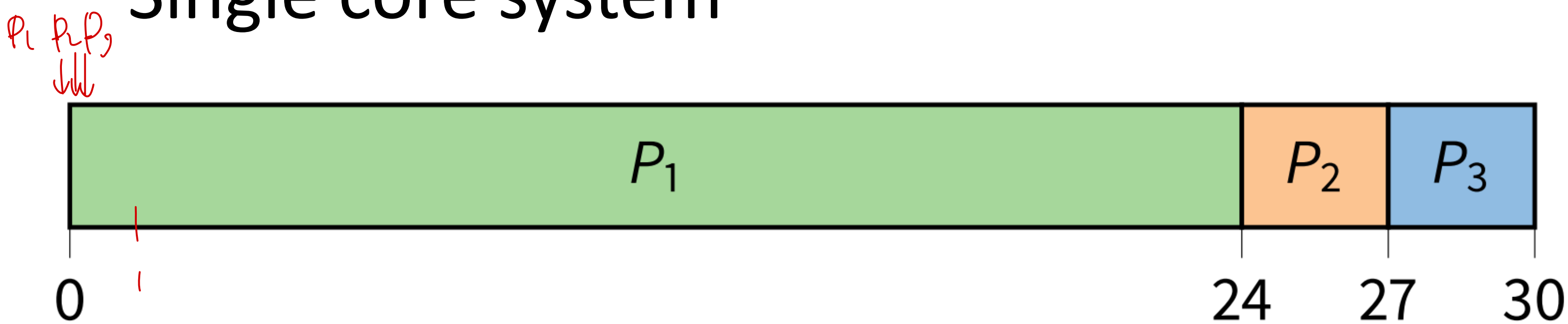
Non preemptive scheduling

- Schedule tasks in the order they arrive
  - Run tasks in order that they arrive
- Example: caching server
  - Facebook: cache of friend lists, image blobs etc

- On what workloads are FIFO particularly bad?

FIFO scheduling example

- P1 needs 24 sec, P2 and P3 needs 3 sec
- Arrival order: P1, P2, P3
- Single core system



- Avg. Throughput: 3 jobs/30sec = 0.1 jobs/sec
- Avg. Turnaround time : (24 + 27 + 30) / 3 = 27
- Avg. wait time : (0 + 24 + 27) / 3 = 17

Can we do better?

Beyond FIFO scheduling

- T1 needs 24 sec, T2 and T3 needs 3 sec
- Changing scheduler order: P2, P3, P1

- Avg. Throughput: 3 jobs/30sec = 0.1 jobs/sec
- Avg. Turnaround time: (3 + 6 + 30) / 3 = 13 <27
- Avg. wait time : (0 + 3 + 6) / 3 = 3 <17

Lesson: schedule algorithm

can reduce turnaround time and wait time

Convoy effect

Img source:
https://cs.jhu.edu/~huang/cs318/fall18/lectures/lec4_sched.pdf

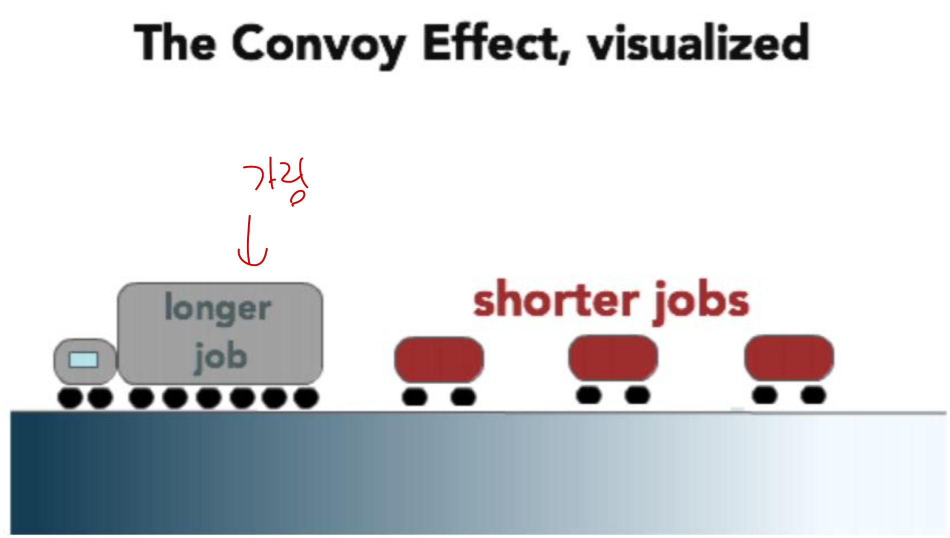
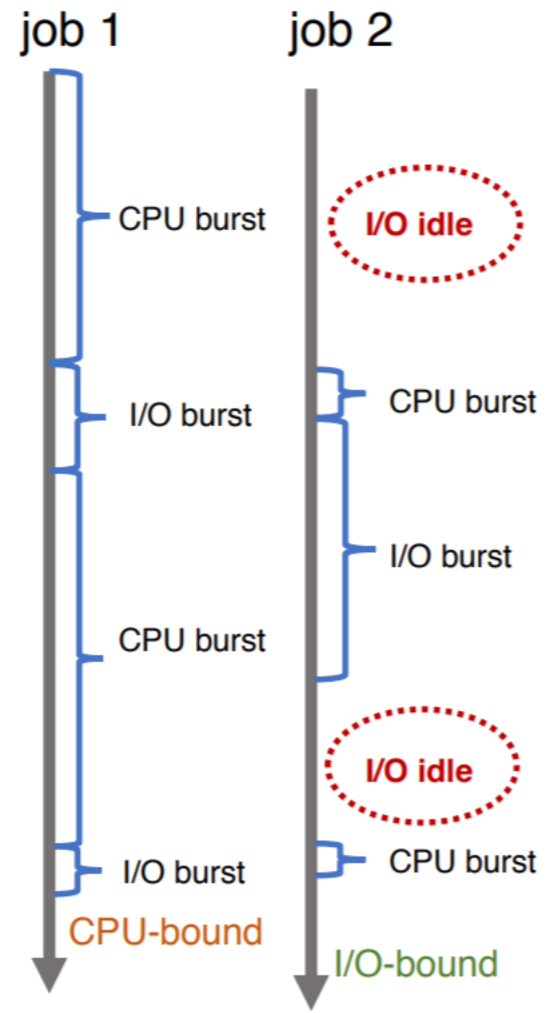


image source:
http://web.cs.ucla.edu/classes/fall14/cs111/scribe/7a/convoy_effect.png

Convoy effect in CPU-bound vs. IO-bound jobs

- CPU-bound jobs will hold CPU until exit or I/O
  - But I/O burst for CPU-bound job is small
  - Long periods where no I/O requests issued, and CPU held
  - Result: poor I/O device utilization

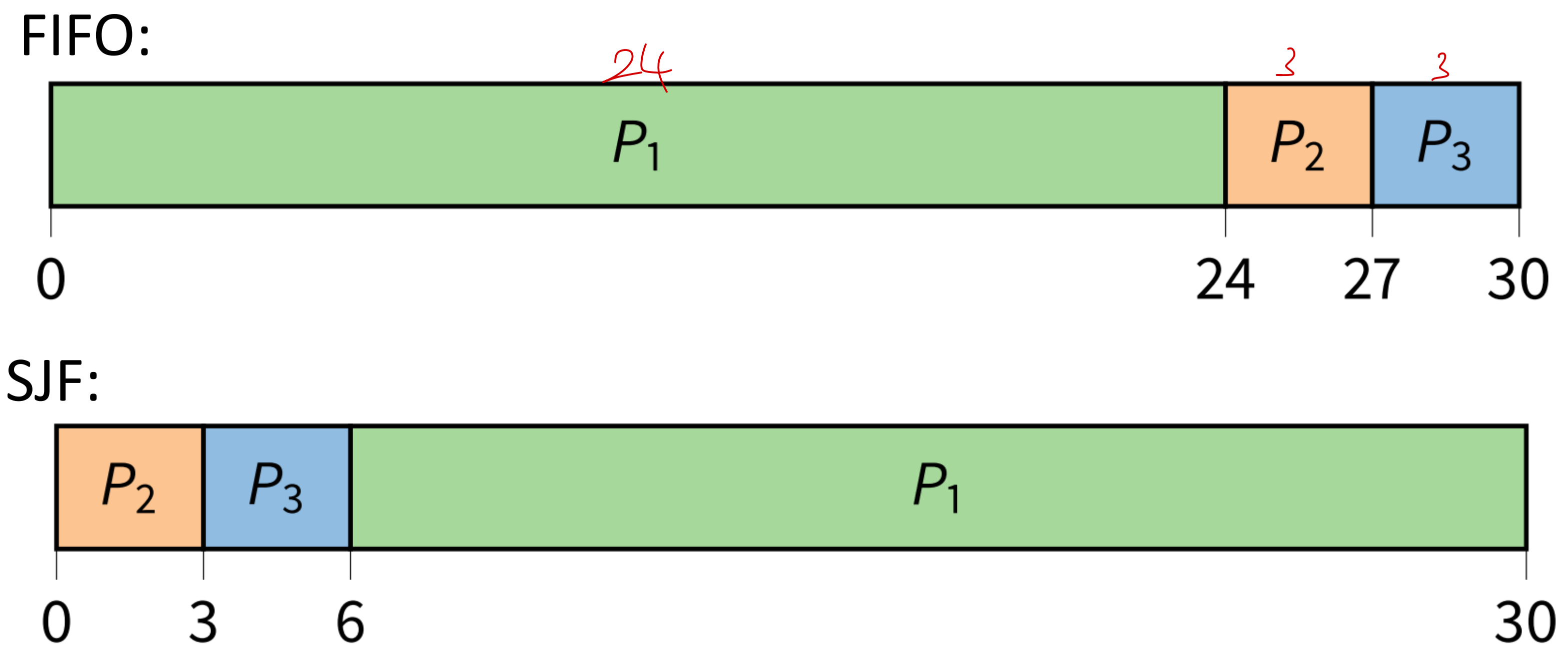


Shortest Job First (SJF)

→ min ANT!

Non preemptive vs Preemptive

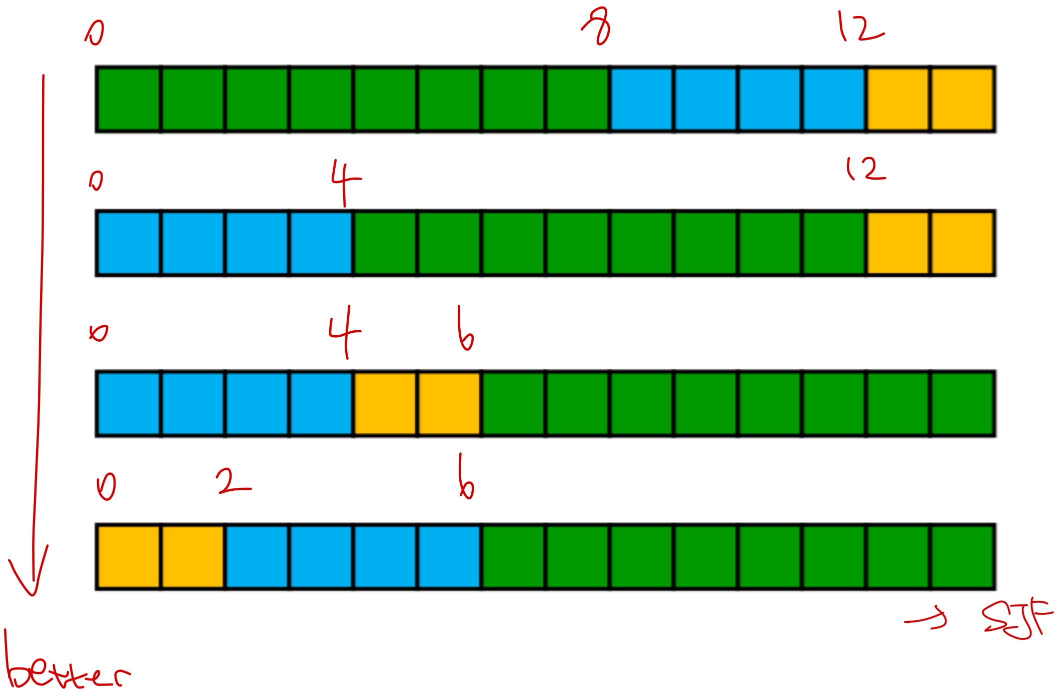
- Always do the task that has the shortest remaining amount of work to do
  - Often called Shortest Remaining Time First (SRTF)



Shortest Job First (SJF)

- Provably optimal minimum average waiting time (AWT)

Choose the job with the smallest expected CPU burst



AWT = (0+8+(8+4))/3 = 6.67

AWT = (0+4+(4+8))/3 = 5.33

AWT = (0+4+(4+2))/3 = 3.33

AWT = (0+2+(2+4))/3 = 2.67

Two schemes of SJF

- Non-preemptive SJF
  - Once CPU is given to a process, it cannot be preempted (kicked-out from currently using CPU) until it completes its work
- Preemptive SJF
  - If a new process arrives with shorter remaining time of current running process, preempt it

AWT, Response time.

What scheduling metric does SJF improve overs FIFO?

preemptive.

x ?

Non-preemptive vs preemptive.

Which has better response time? AWT?

Draw scheduling diagram

$TT_N = (7+4+10+11)/4 = 8$

$TT_P = (16+5+1+6)/4 = 7$

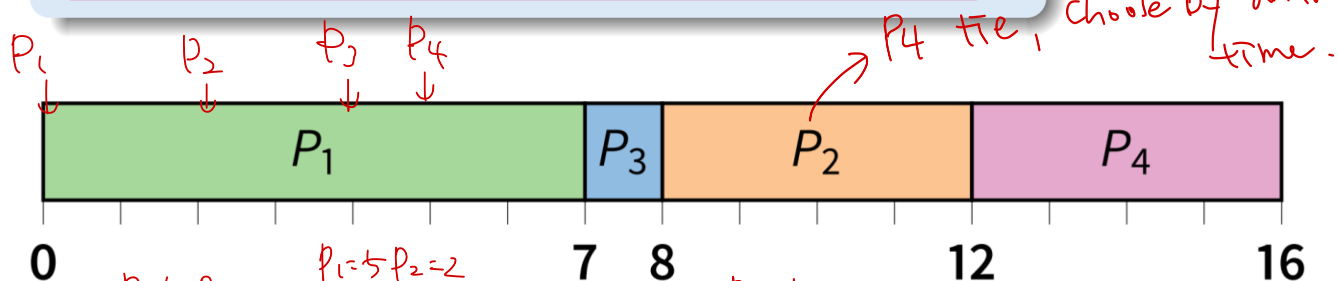
Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Response time

$P_1=0,\ P_2=6,\ P_3=3,\ P_4=7$

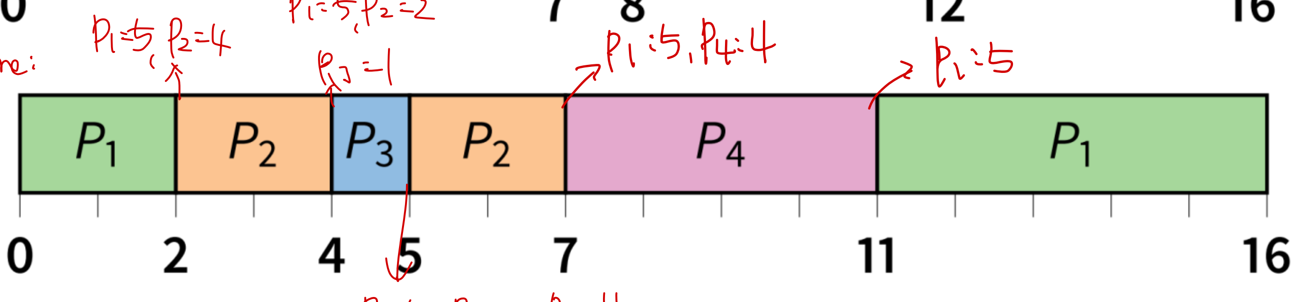
$AWT = (0+6+3+7)/4 = 16/4 = 4$

Non-preemptive:



Remaining time:

Preemptive:



Response time

$P_1:0$ $P_4:2.$

$P_2:0$

$P_3=0$

AWT of P_i

$P_2 = 0+1$

$P_1 = 0+9$ $P_3 = 0$ $P_4 = 2$

Compute and compare avg. response time and AWT

$AWT = 12/4 = 3$

→ 둘다 preemptive가 better

SJF limitations

Turnaround Time

- Doesn't always minimize average TT
  - Only minimizes waiting time
  - How to improve TT of SJF?
- Sometime, impossible to know size of CPU burst ahead of time → predicting..
  - Like choosing person in line without looking inside basket/cart

How to predict CPU burst time?

request $r_1 \rightarrow$

$r_2 \rightarrow$

$r_3 \rightarrow$

predict burst time of r_1

- Estimate CPU burst length based on the past
  - E.g., Exponentially Weighted Average
    - t_n : length of a process's n-th CPU burst
    - $t_{n+1} = \alpha t_n + (1 - \alpha) t_{n-1}$ linear combination of n-th and (n-1)-th requests
      - α is parameter (e.g., 0.5)

measured value

SJF limitations

- Doesn't always minimize average TT
  - Only minimizes waiting time
  - How to improve TT of SJF?
- Sometime, impossible to know size of CPU burst ahead of time
  - Like choosing person in line without looking inside basket/cart
- What is a critical problem of SJF?

Can potentially lead to "Starvation"

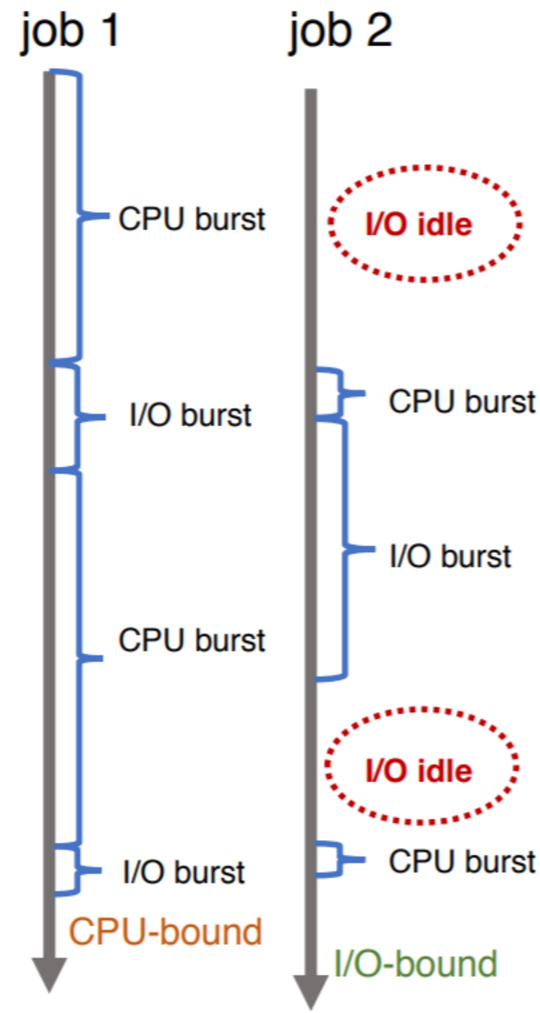
Some jobs never get CPU

ex. 1 long task

+ many short tasks coming

Revisit convoy effect in CPU-bound vs. IO-bound jobs

- CPU-bound jobs will hold CPU until exit or I/O
  - But I/O burst for CPU-bound job is small
  - Long periods where no I/O requests issued, and CPU held
  - Result: poor I/O device utilization
- **Simple solution: run process whose I/O completed**
  - What is a potential problem?



Round Robin

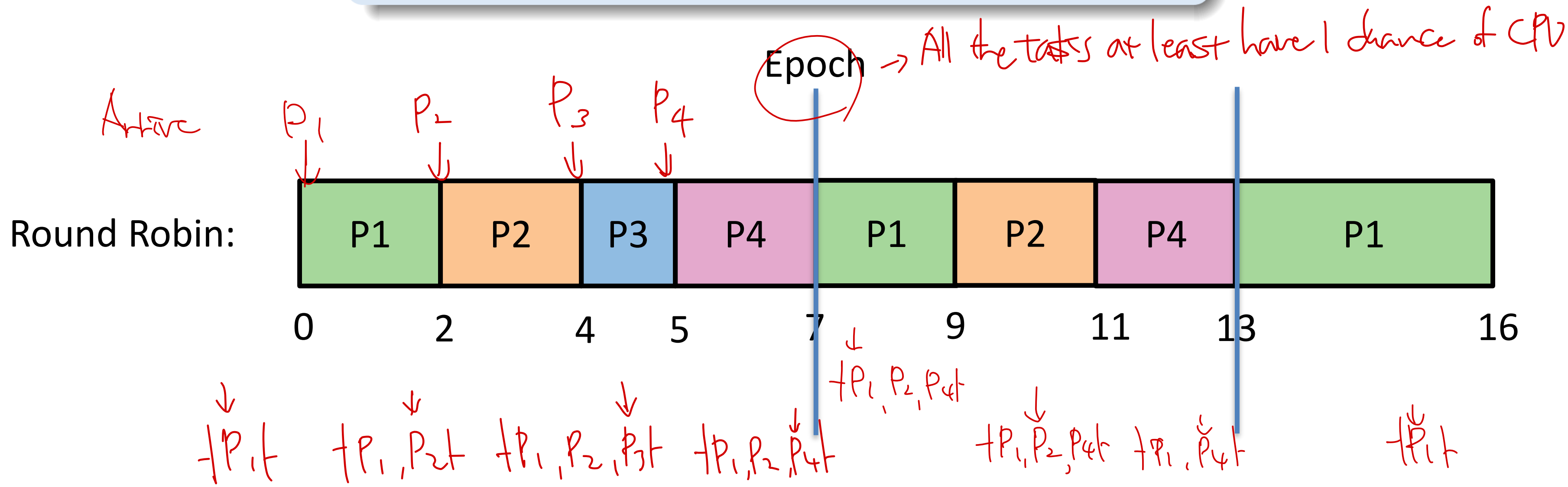
- Solution to avoid the starvation problem in SJF

- Each task gets resource for a fixed period of time (time quantum or time slice)
  - If task doesn't complete, it goes back to FIFO queue

Round Robin

Time slice: 2

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

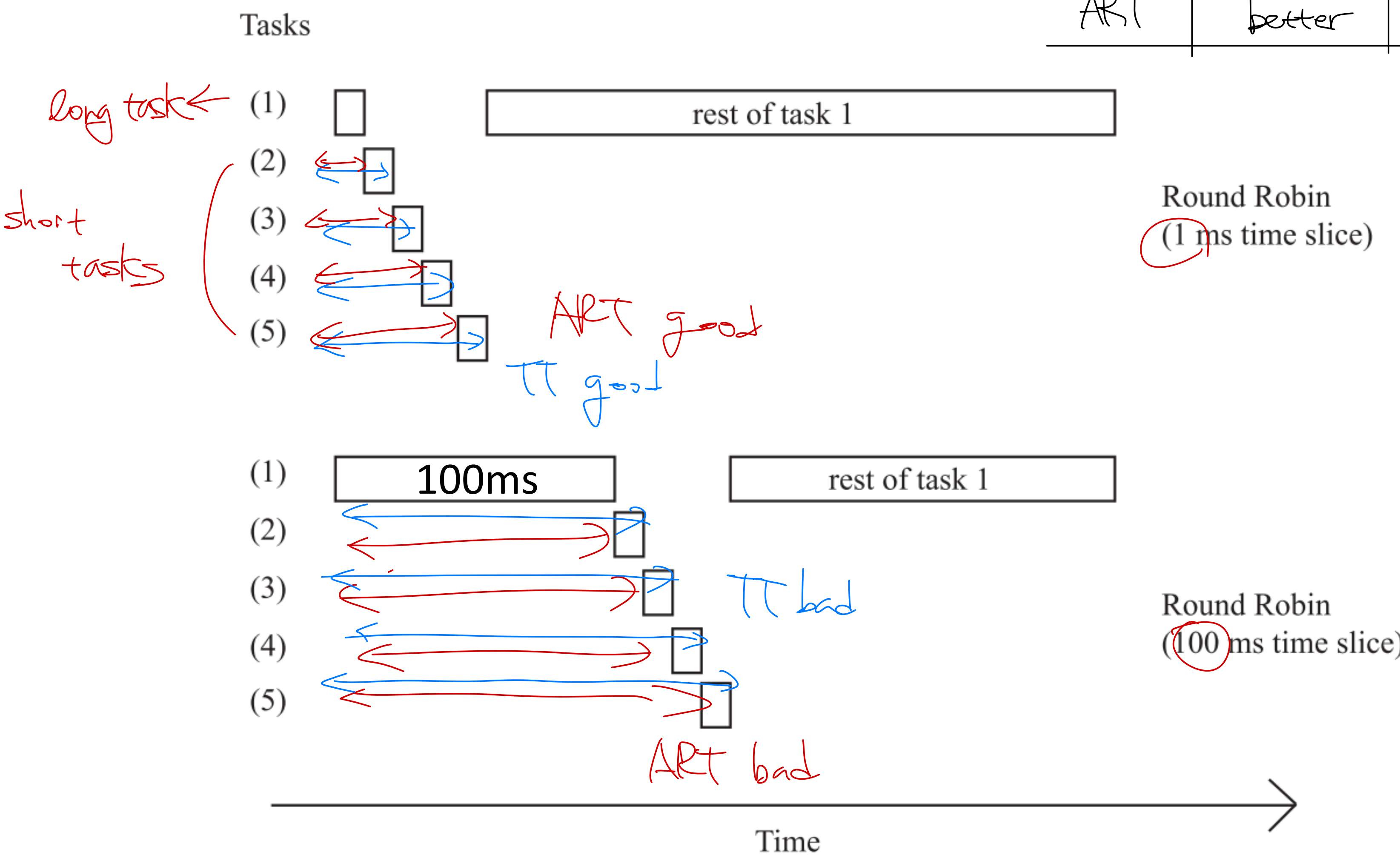


Round Robin

- Need to pick a time quantum
  - What if time quantum is too long?
    - Infinite? → Same as FIFO
  - What if time quantum is too short?
    - One instruction? → Context switch overhead gets heavy

Round Robin

Metric \ Time Slice	Short	Long
TT	better	
ART	better	



In terms of scheduling performance metric, what is different?

Time slice summary

Longer or shorter?

CPU bound task prefer (longer) time slices

long CPU burst

→ Usually server

IO bound task prefer (shorter) time slices

Short CPU burst

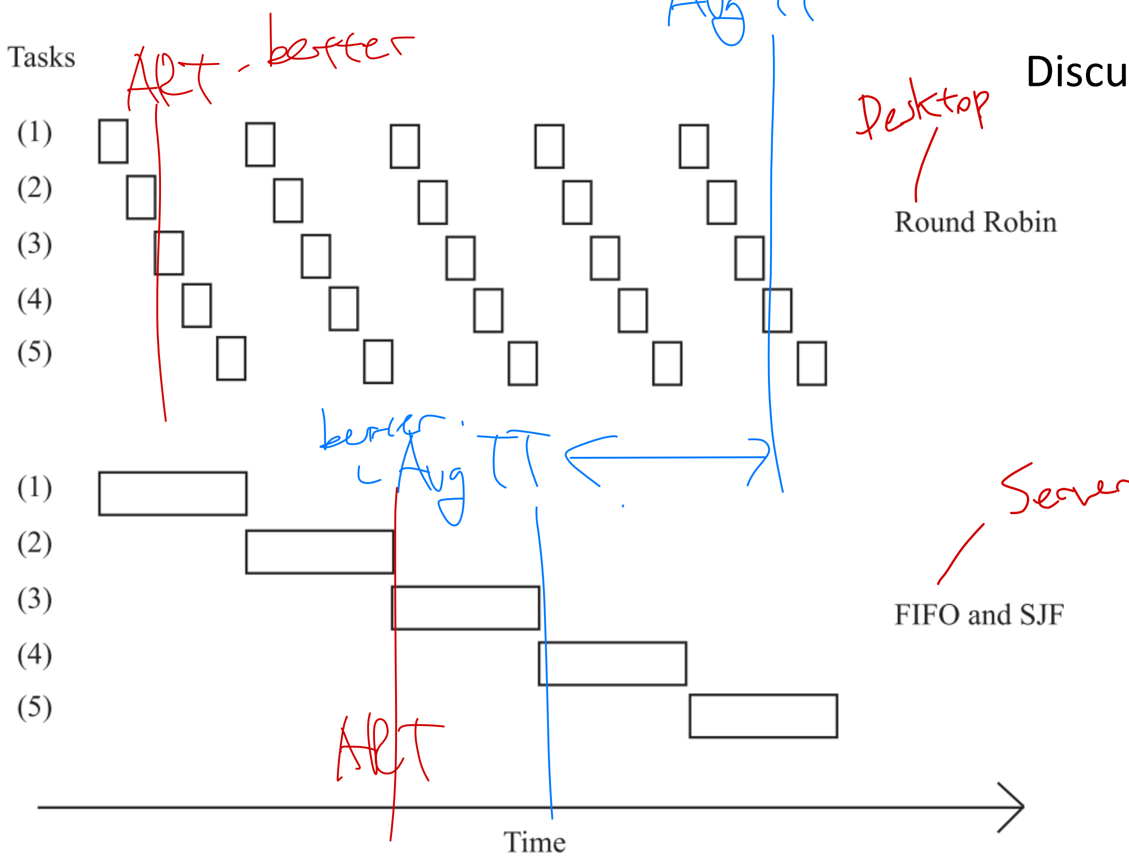
→ Usually Desktop

→ More responsive

TT → FIFO better Response Time → Round Robin

Round Robin vs. FIFO

- Assuming zero-cost time slice, is Round Robin always better than FIFO?



Discuss response and turnaround time

Good response time
Terrible turnaround time

Modest response time
Good turnaround time

Round Robin = Fairness?

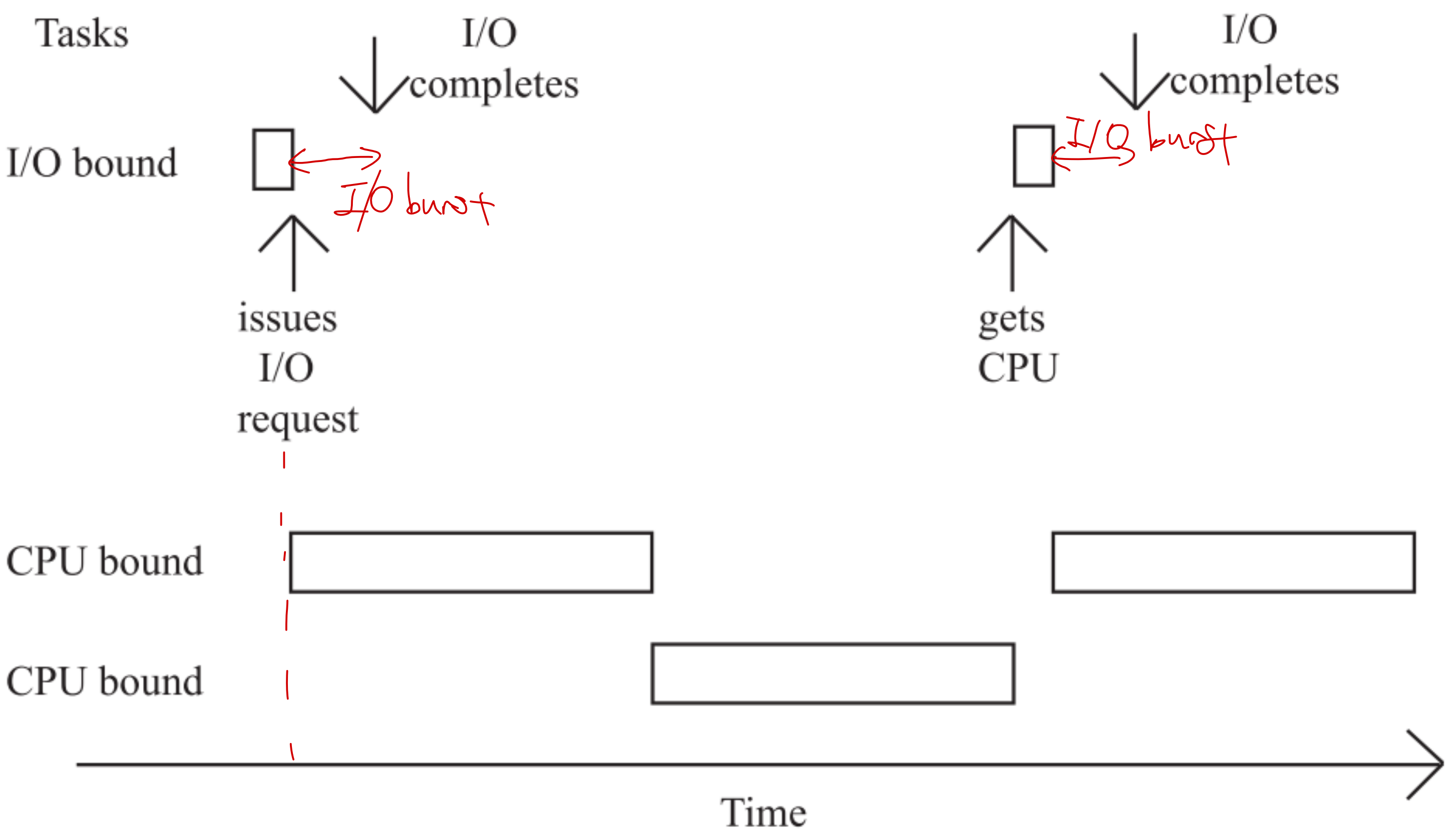
P_1 4 CPU burst

P_2 2 CPU burst

- Is Round Robin always fair?

- What is fair?
  - FIFO? →선착?
  - Equal share of the CPU? →같은 시간 사용?
  - What if some tasks don't need their full share?
  - Minimize worst case divergence?
    - Time task would take if no one else was running
    - Time task takes under scheduling algorithm

Mixed Workload



How to define fairness in mixed workload?

Max-Min Fairness

- How do we balance a mixture of repeating tasks:
  - Some I/O bound, need only a little CPU
  - Some compute bound, can use as much CPU as they are assigned
- One approach: *maximize the minimum allocation given to a task*
  - If any task needs less than an equal share, schedule the smallest of these first
  - Split the remaining time using max-min
  - If all remaining tasks need at least equal share, split evenly

Widely used in network scheduling

Max-Min Fairness example

- Demands of 4 tasks = {1.9, 2.5, 4, 5}, capacity = 10
  - Equal share = 10/4 = 2.5
  - Share: {**2.5**, 2.5, 2.5, 2.5}
  - A task only needs 1.9 → 0.6 extra
    Put first
  - Equally distribute 0.6 to the rest three tasks
  - Each of them could have 0.6/3 = 0.2 extra
  - Share: {1.9, **2.7**, 2.7, 2.7}
  - 0.2 extra: distribute 0.2/2 = 0.1 to the rest two tasks
  - Share: {1.9, 2.5, 2.8, 2.8} Done!

Priority scheduling

- Give CPU to the process with highest priority
  - Preemptively or non-preemptively
  - What is the priority in SJF? → Burst time
- A critical problem of priority scheduling

→ Starvation

Priority scheduling

- Give CPU to the process with highest priority
  - Preemptively or non-preemptively
  - What is the priority in SJF?
- Priority scheduling can cause starvation
  - What if higher priority tasks keep coming to system?
- Solution
  - Aging: increase a process’s priority as it waits

Multi-level Feedback Queue (MLFQ)

└ Linux default scheduler

- Goal #1: Optimize job turnaround time for "batch" jobs
  - Shorter jobs run first → Long time slice for batch jobs
  - Why not SJF? → Starvation?
- Goal #2: Minimize response time for "interactive" jobs

  → Short time slice for interactive jobs

- Challenge: - No a priori knowledge of what type a job is, what the next burst is, etc.

- Idea: - Change a process's priority based on how it behaves in the past ("feedback")

MLFQ

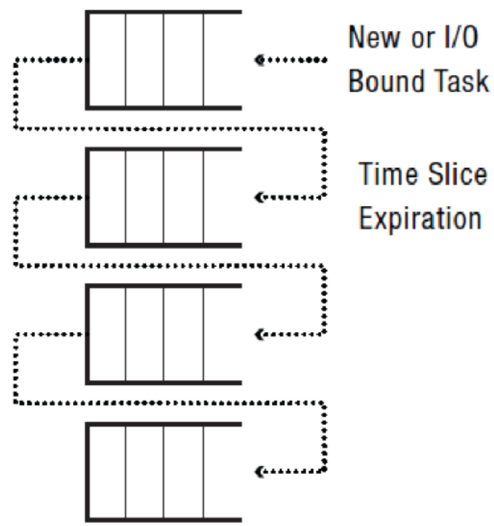
Good TT & Response Time

- Set of Round Robin queues
  - Each queue has a separate priority
- High priority queues have *short time slices* interactive
  - Low priority queues have *long time slices* — batch
- Scheduler picks the first thread in highest priority queue
- Tasks start in the highest priority queue
  - If time slice expires, task drops one level

MLFQ example

Priority	Time Slice (ms)
1	10
2	20
3	40
4	80

Round Robin Queues



CPU bound process goes down

Response time↓ for interactive jobs

Shorter time slice in higher priority queues. Why?
MLFQ ensures IO bound tasks to be schedule quickly. How?
The MLFQ algorithm is starvation-free? No.

IO bound tasks 가 계속 오면 priority 4가 안됨 → Need aging

Contents

- Uniprocessor policies
  - FIFO, round robin, optimal
  - multilevel feedback as approximation of optimal
- Multiprocessor policies
  - Affinity scheduling, gang scheduling

Multiprocessor Scheduling

- Must decide on more than which processes to run
  - Decide on which CPU to run which process

→ 어떤 process는 어떤 cpu에서 돌릴지.

- Moving between CPUs has costs
  - Cache/TLB misses, Task migration costs

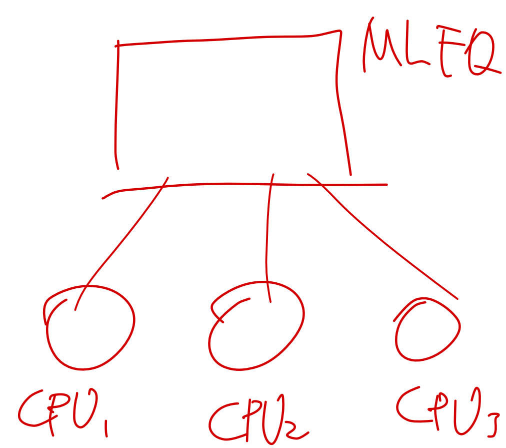
Cold miss

Multiprocessor Scheduling

L

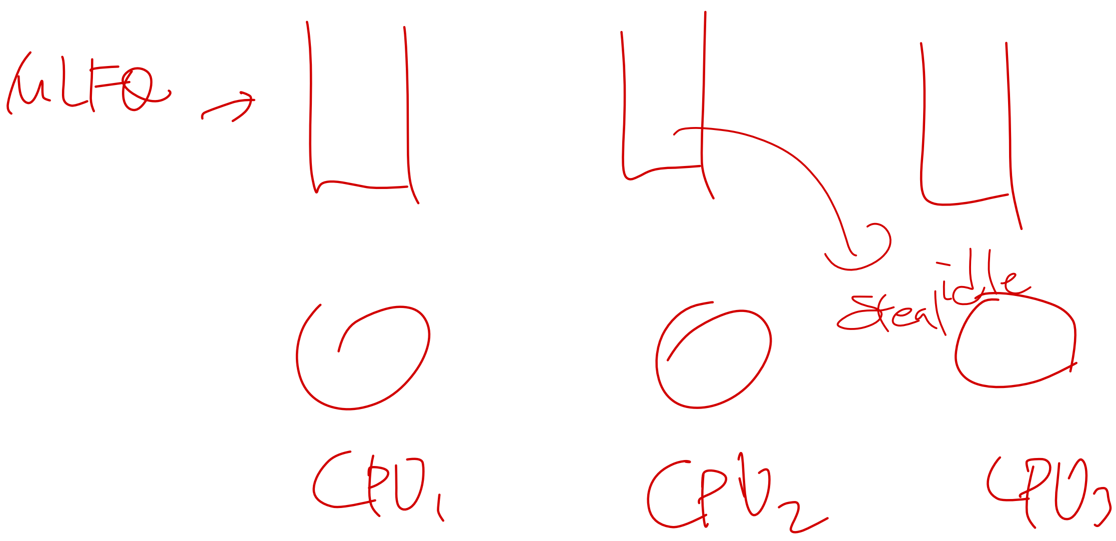
- What would happen if we used a global MFQ on a multiprocessor?
  - Contention for scheduler spinlock → different MLFQ for each core

    Solution:
  - Cache slowdown due to ready list data structure pinging from one CPU to another
  - Limited cache reuse: thread’s data from last time it ran is often still in its old cache



Per-core ready list

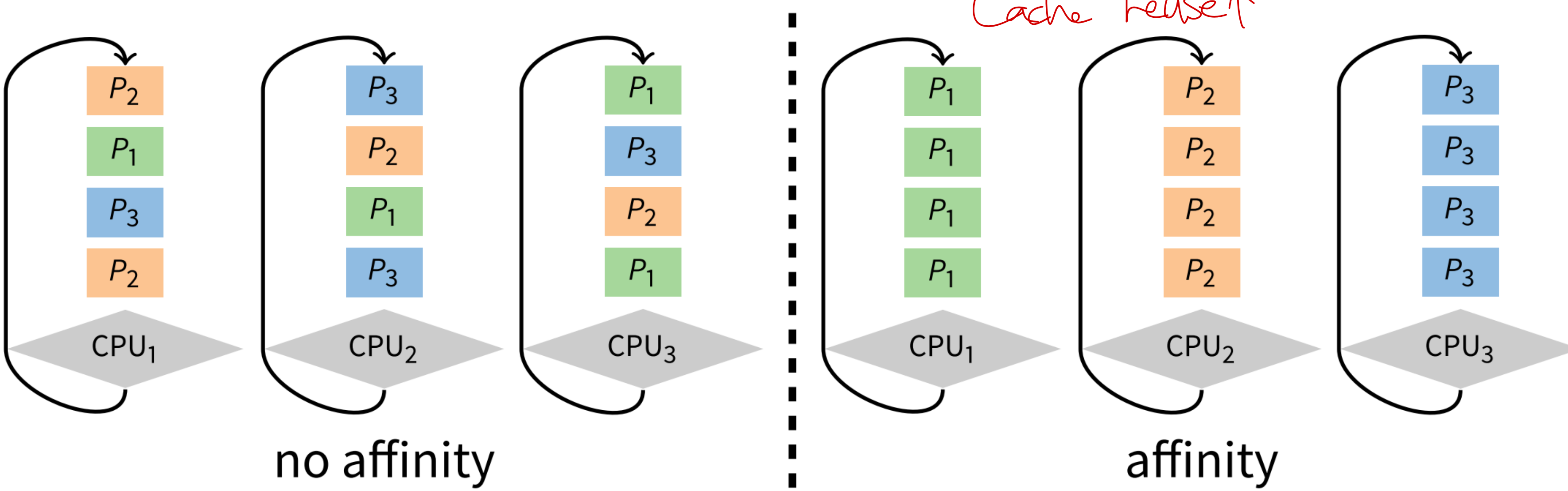
- Each CPU has a per-core ready list
- Work-conserving: Idle processors can **steal** process from other processor



Per-Processor Affinity Scheduling

- Each processor has its own ready list
  - Protected by a per-processor spinlock
- Put threads back on the ready list where it had most recently run
  - Ex: when I/O completes, or on Condition->signal

Cache reuse ↑

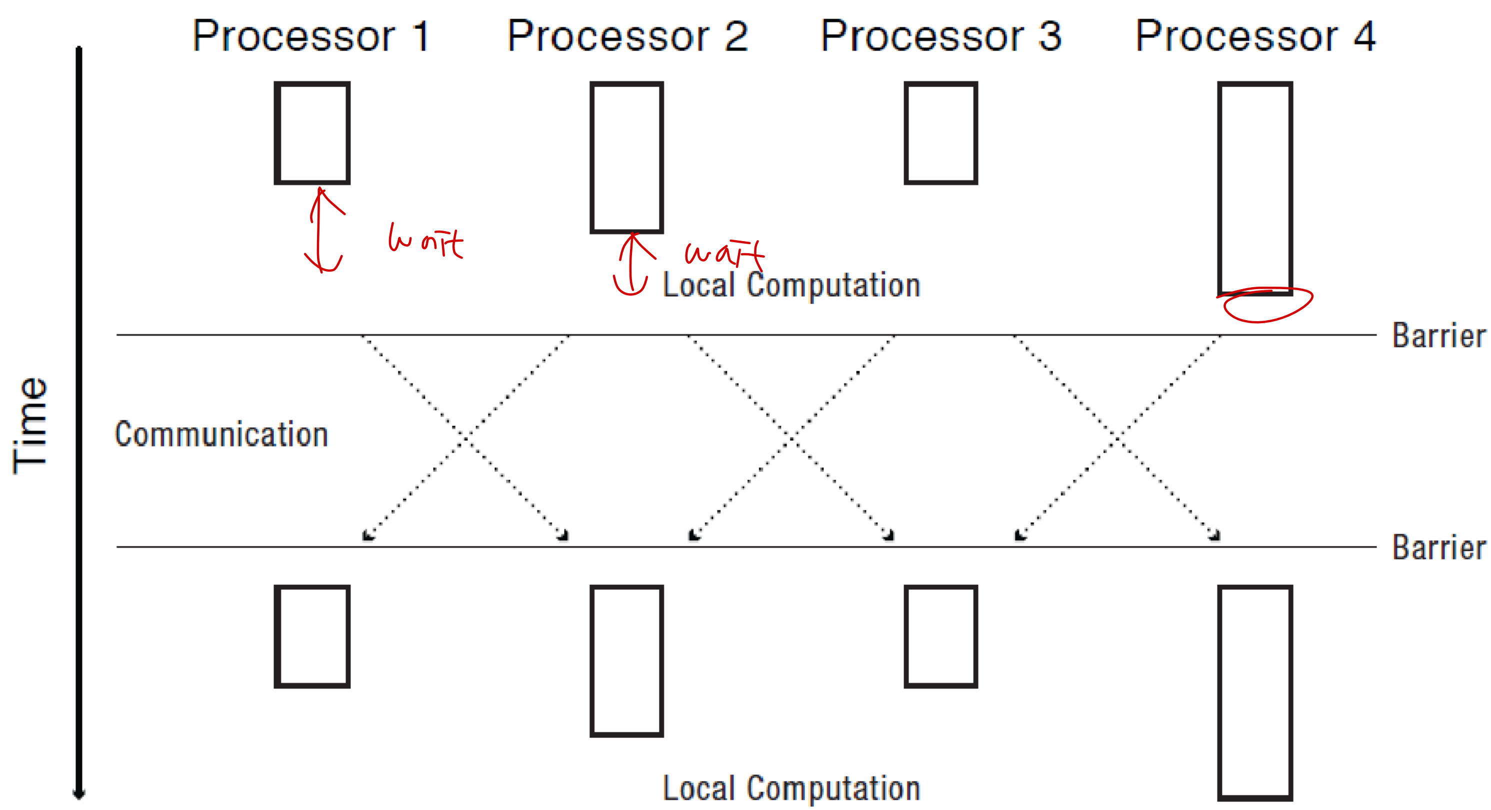


→ bad for cache reuse.

Scheduling Parallel Programs

- What happens if one thread gets a short time slice while other threads have longer time slices?
  - Assuming program uses locks and condition variables, it will still be correct
  - What about performance?

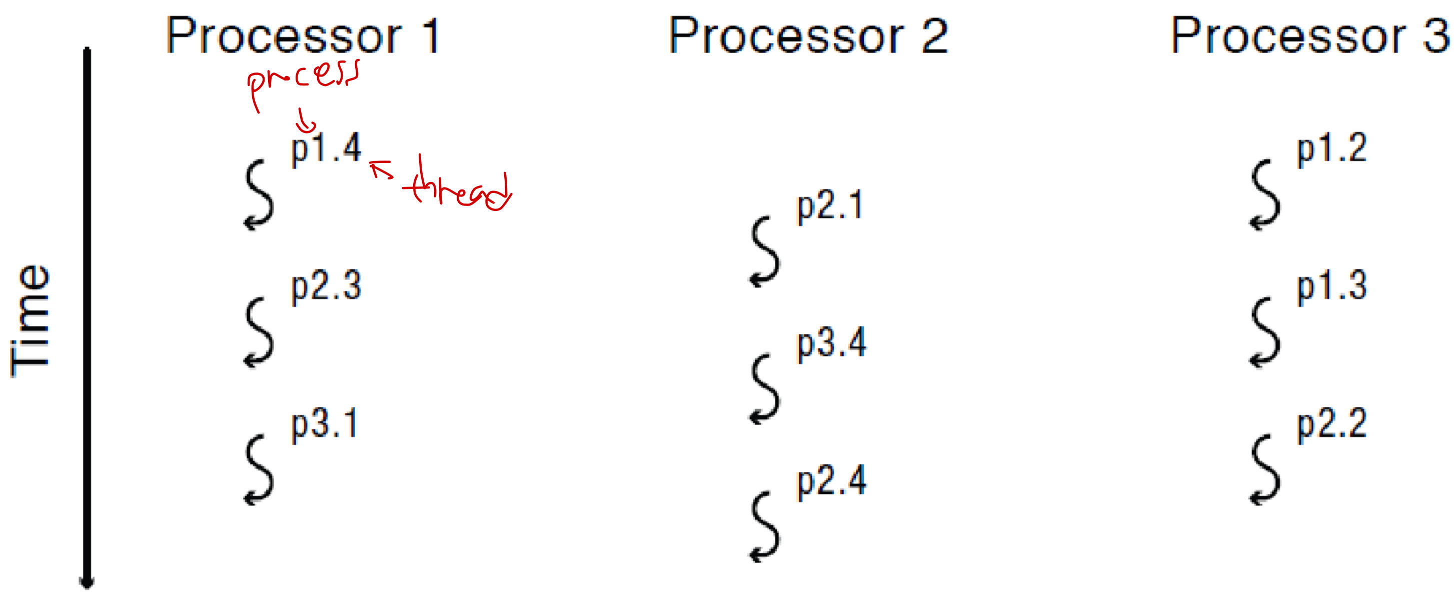
Tail Latency
by bulk synchronous pattern



Scheduling Parallel Programs

의식하지 못하는

Oblivious: each processor time-slices its ready list independently of the other processors



px.y = Thread y in process x

Gang scheduling

- Schedule related processes/threads together
  - Based on communication, data sharing pattern

- Schedule all CPUs synchronously
  - With synchronized time slice, which is easier to scheduler related processes/threads together