Scheduling → ใช้ประโยชน์ในการทำงานก่อน ? และ

Main Points

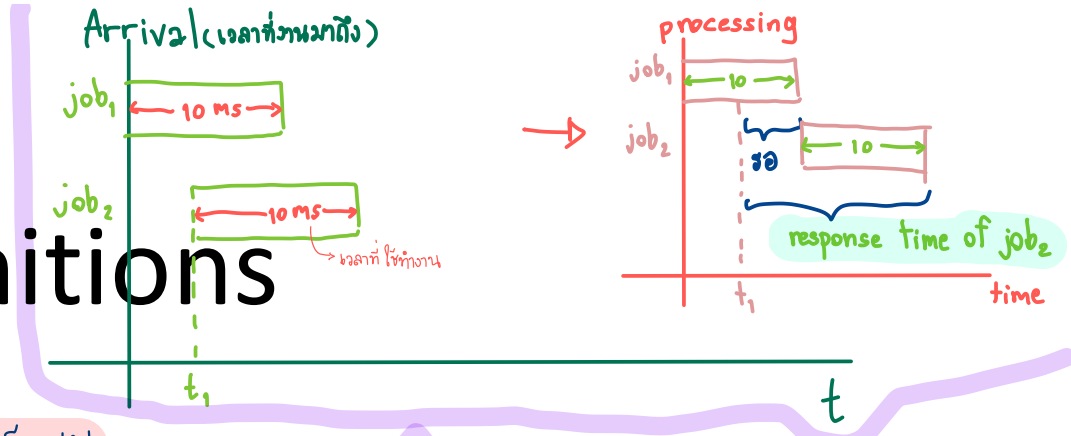
- Scheduling policy: what to do next, when there are multiple threads ready to run
  - Or multiple packets to send, or web requests to serve, or …
- Definitions
  - response time, throughput, predictability
- Uniprocessor policies
  - FIFO, round robin, optimal
  - multilevel feedback as approximation of optimal
- Multiprocessor policies → ไม่สอน
  - Affinity scheduling, gang scheduling
- Queueing theory → ไม่ออกสอบ
  - Can you predict/improve a system's response time?

Example

เทียบ processor = server ← Client req = task, job

- You manage a web site, that suddenly becomes wildly popular. Do you?
  - Buy more hardware? เทพกรู
  - Implement a different scheduling policy? → ต้องให้ software OS มาช่วย จัดตารางเวลาเพื่อลด load
  - Turn away some users? Which ones? เพดานของ ก. ให้บริการ เหมือน ชิม ช้อป ใช้ → หรือไม่ก็ปิด server ทิ้งไปเลย
- How much worse will performance get if the web site becomes even more popular?

อยู่ดีๆ popular ขึ้นมา ex วันลงทะเบียน เว็บ reg



Definitions

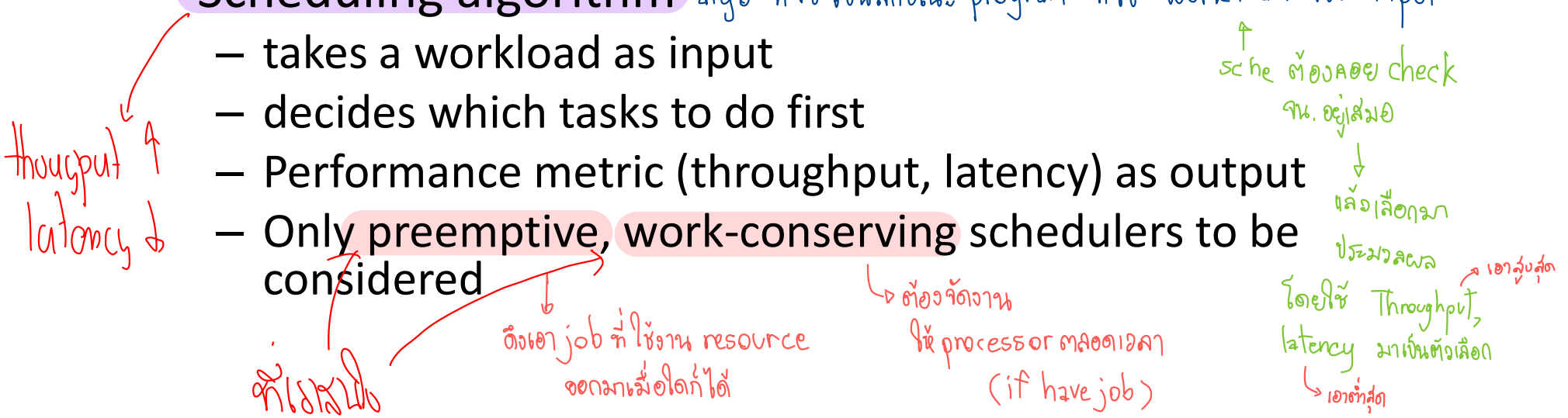
- **Task/Job** งานที่ user ป้อนเข้ามาในระบบ
  - User request: e.g., mouse click, web request, shell command, …
- **Latency/response time** เวลาที่งานใช้ประมวลผล → ยิ่งน้อยยิ่งดี
  - How long does a task take to complete?
- **Throughput** → จำนวนงานที่ทำเสร็จใน 1 unit of time
  - How many tasks can be done per unit of time?
- **Overhead** ยิ่งน้อยยิ่งดี → เน้นที่ overhead ที่เกิดกับ schedular → ถ้าใช้ algorithm ง่ายๆ overhead น้อย
  - How much extra work is done by the scheduler?
- **Fairness** จัดสรรเวลา, ทรัพยากร ให้เท่าเทียม → ตาม priority
  - How equal is the performance received by different users?
- **Predictability** ค.สามารถในการ เดา → program นั้น ต้องให้ผลลัพธ์ที่สม่ำเสมอ → เช่น เล่นเกมแล้วแลค → หงุดหงิด
  - How consistent is the performance over time?

user ไม่ได้ req แต่ต้องทำ = overhead

↳ cpu เสียเวลาคิดน้อย

More Definitions

- **Workload** กลุ่มของ job
  - Set of tasks for system to perform
- **Preemptive scheduler** ก.ทำ scheduler ที่ scheduler เข้าไปขัดจังหวะได้
  - If we can take resources away from a running task
- **Work-conserving** processor จะต้องทำงานตลอดเวลา ถ้ายังมีงานอยู่ใน queue
  - Resource is used whenever there is a task to run
- **Scheduling algorithm** algo ที่ใช้เป็นลักษณะ program ที่ใช้ workload เป็น input
  - takes a workload as input
  - decides which tasks to do first
  - Performance metric (throughput, latency) as output
  - Only preemptive, work-conserving schedulers to be considered



ง่ายสุด

First In First Out (FIFO)

เลือก job ไปใช้งานตามลำดับมาก่อน → algo ง่าย

- Schedule tasks in the order they arrive
  - Continue running them until they complete or give up the processor
- Example: memcached
  - Facebook cache of friend lists, …

- On what workloads is FIFO particularly bad?

ข้อจำกัด

↳ ให้ performance ในบางกรณีไม่ดีเท่าไร

ใช้ใน ทฤษฎี ปฏิบัติไม่ได้ → เพราะ วัดตัวที่ใช้เวลาน้อยไม่ได้
↳ตั้งแต่ตอนมันยังไม่ทำงาน ex รอนอนก่อผู้ใช้ จะป้อน 0

Shortest Job First (SJF)

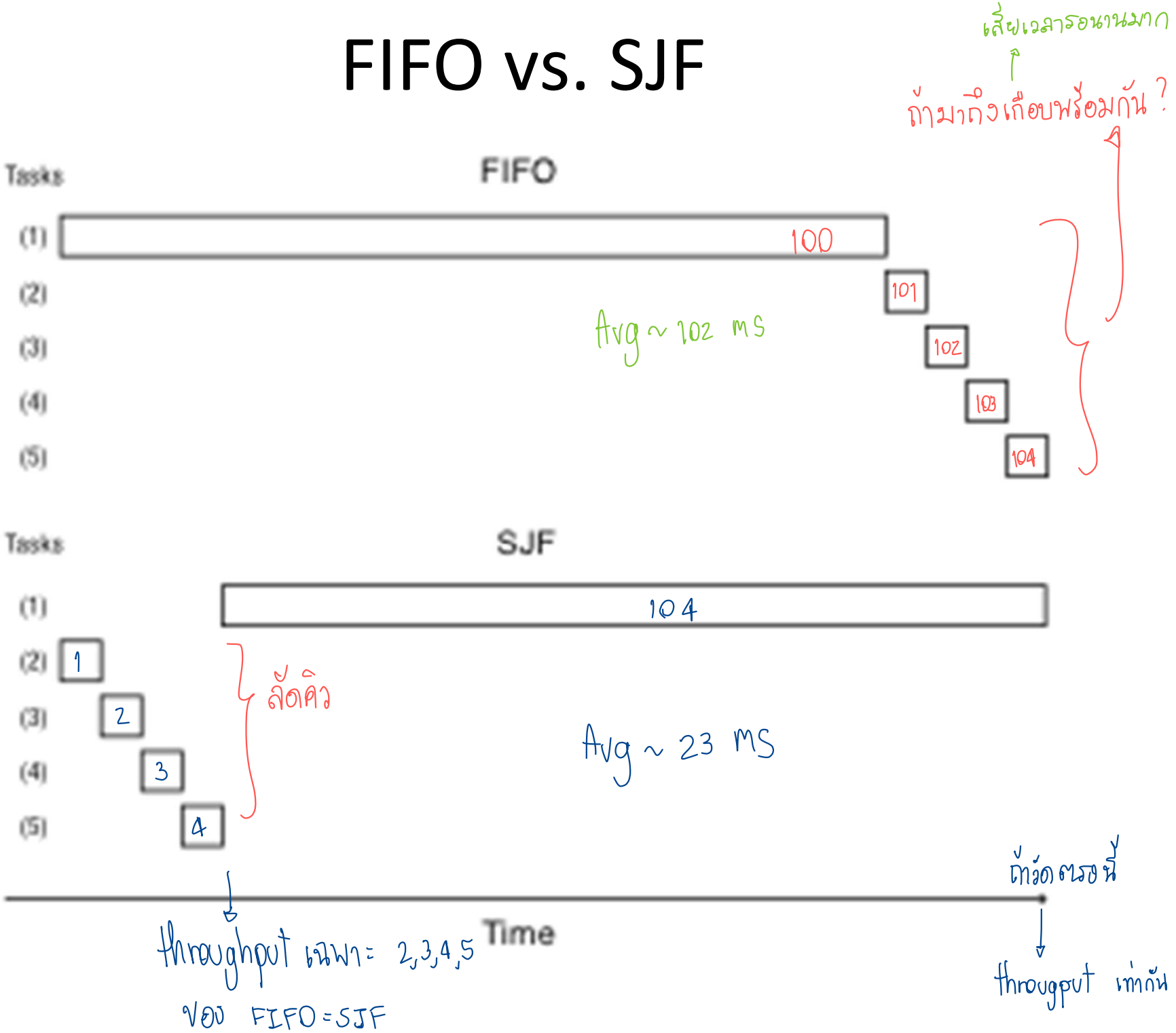
เป็นแบบ first in first out แต่สามารถสลับคิวได้
↳ตัวที่ใช้เวลาน้อยสลับคิวได้

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

แต่ถ้าทำงานไปแล้วจะแย่ง resource ไปเลยไม่ได้

- Suppose we have five tasks arrive one right after each other, but the first one is much longer than the others
  - Which completes first in FIFO? Next?
  - Which completes first in SJF? Next?

FIFO vs. SJF


เสียเวลารอนานมาก
ถ้ามาถึงเกือบพร้อมกัน ?
Tasks
FIFO
(1)
(2)
(3)
(4)
(5)
100
101
102
103
104
Avg ~ 102 ms
Tasks
SJF
(1)
(2)
(3)
(4)
(5)
104
1
2
3
4
สั้นคิว
Avg ~ 23 ms
Time
throughput เฉพาะ: 2,3,4,5
ของ FIFO=SJF
ถ้าจบตรงนี้
throughput เท่ากัน

Question

- Claim: SJF is optimal for average response time
  - Why? สรุปจากรูปด้านบน

- Does SJF have any downsides? ข้อเสีย

งานที่ใช้เวลามากจะถูกแซงไปเรื่อยๆ

↳ ไม่ได้ process สักที

Question

- Is FIFO ever optimal? เหมาะที่สุด best case ของ FIFO

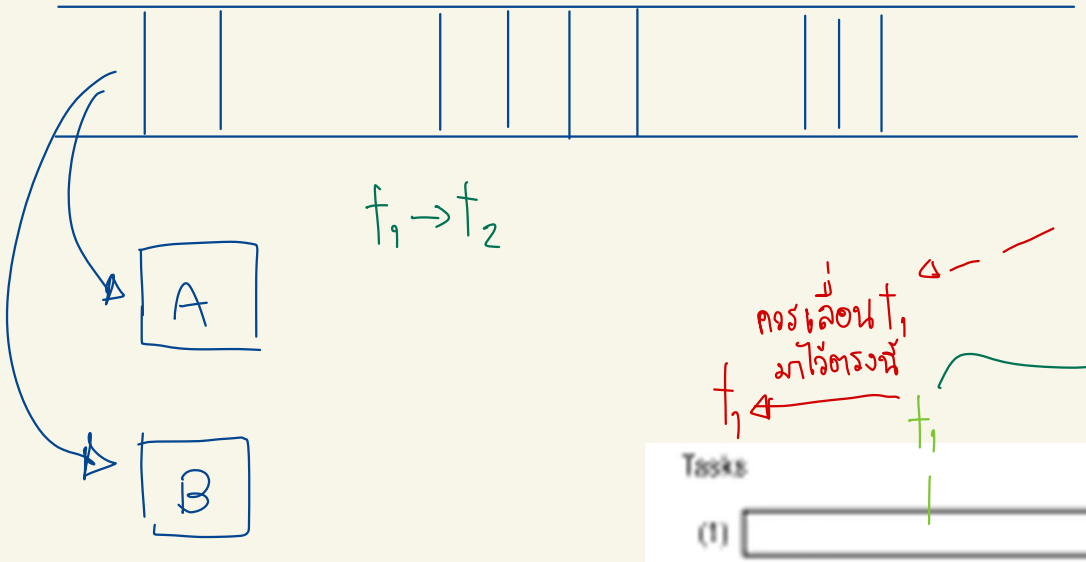
  งานที่เข้ามาใช้เวลาเท่ากันหมด → กราฟของ FIFO กับ SJF เหมือนกัน

- Pessimal? ห่วยสุด

  แบบในรูปห่วย มีตัวใช้เวลามาก มาถึงอันดับแรก

Starvation and Sample Bias

- Suppose you want to compare two scheduling algorithms
  - Create some infinite sequence of arriving tasks
  - Start measuring
  - Stop at some point
  - Compute average response time as the average for completed tasks between start and stop
- Is this valid or invalid?



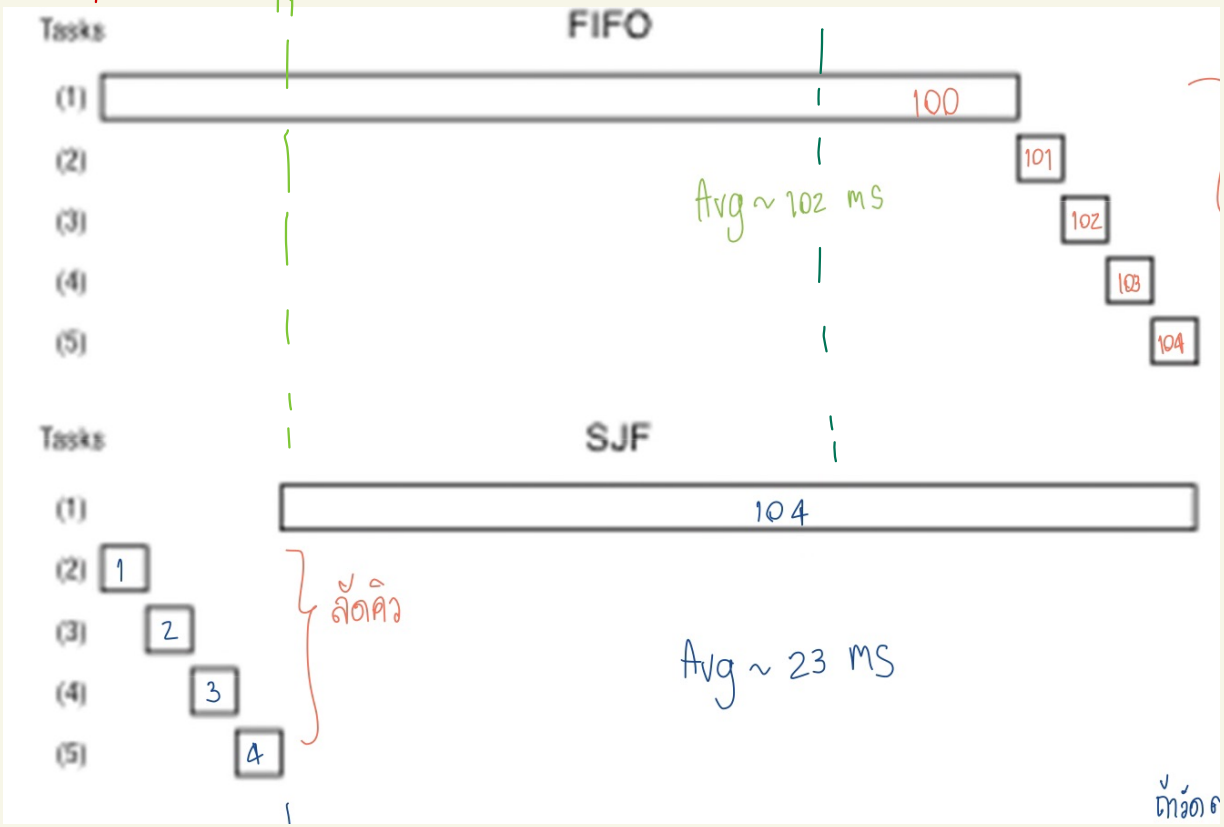
∴ อัลกอริทึมแบบนี้ไม่ได้

FIFO vs SJF

ทั้ง 2 อัน ประสิทธิภาพเท่ากัน Throughput = 0 ทั้งคู่

ถอยเลื่อน t_1 มาไว้ตรงนี้

t_1 t_2



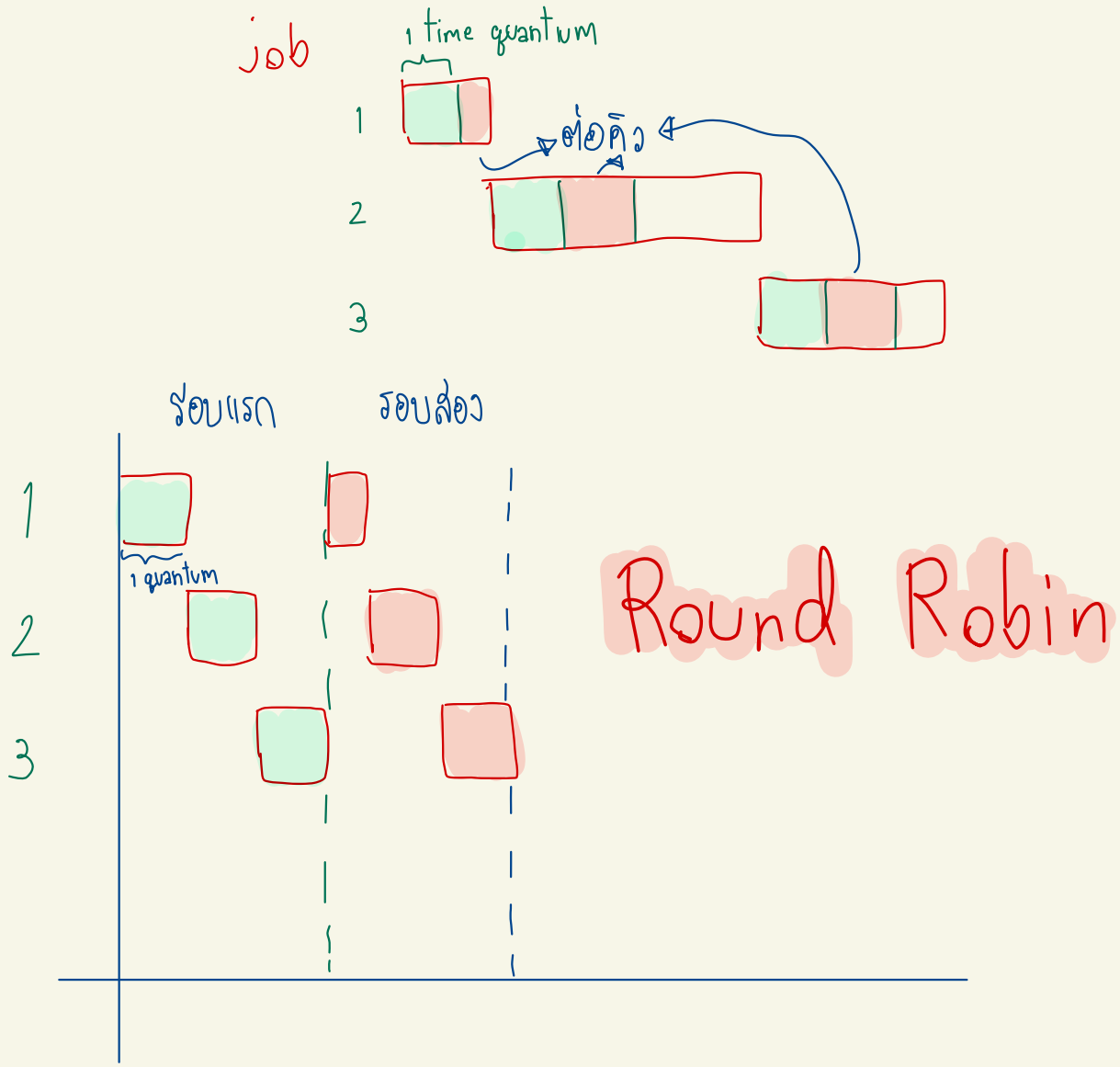
Sample Bias Solutions

- Measure for long enough that # of completed tasks >> # of uncompleted tasks
  - For both systems!
- Start and stop system in idle periods
  - Idle period: no work to do
  - If algorithms are work-conserving, both will complete the same tasks

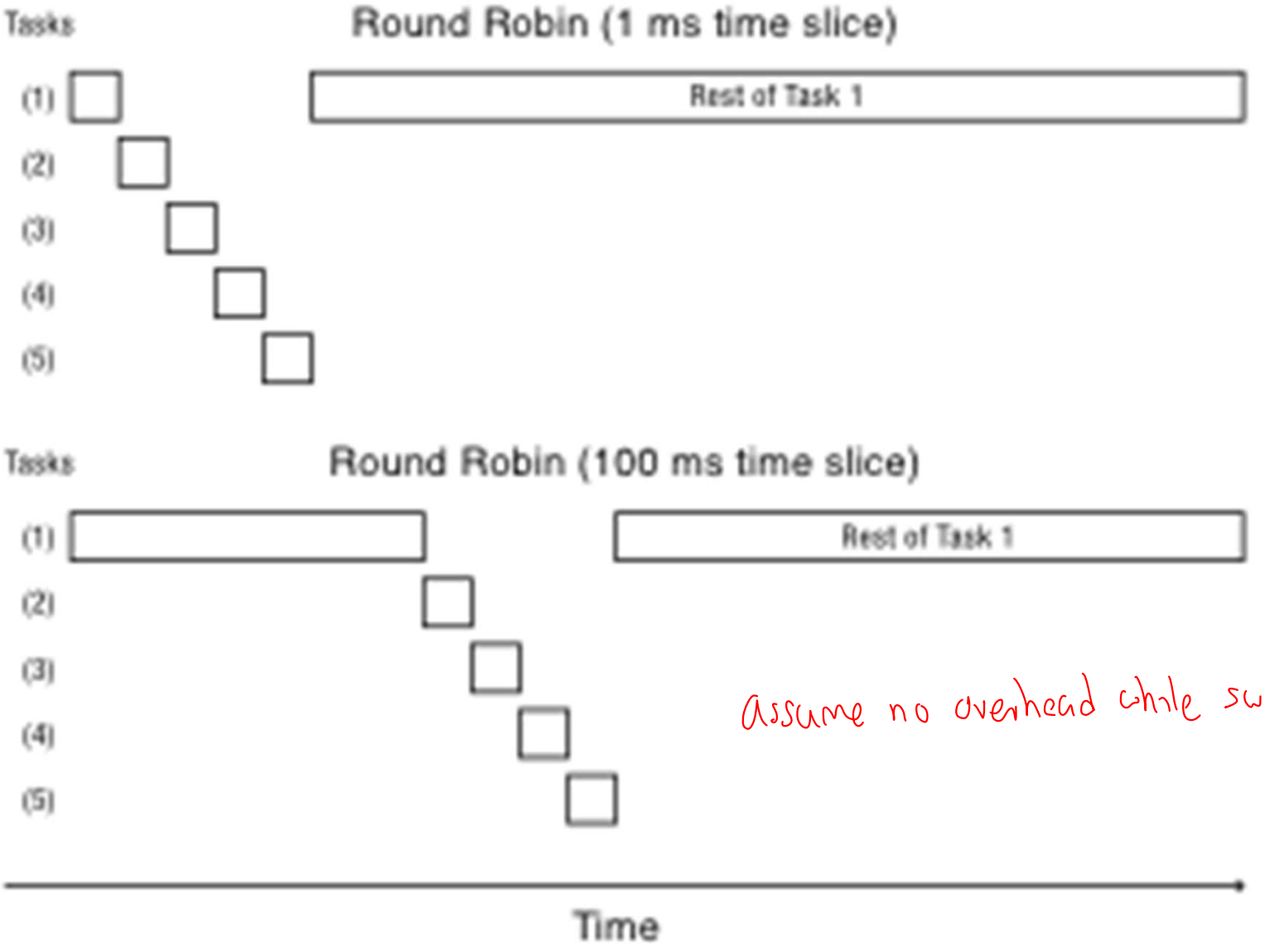
Round Robin

ทำการแบ่ง cpu time เท่ากันหมด → ช่วงเวลาที่ job ได้ = time quantum ⇝ ได้ 1 time quantum เท่ากันหมด → ทำยังไม่เสร็จ → ไปต่อคิวใหม่

- Each task gets resource for a fixed period of time (time quantum)
  - If task doesn't complete, it goes back in line
- Need to pick a time quantum
  - What if time quantum is too long? → ยิ่งใหญ่ → ใกล้ FIFO
    - Infinite? ↳ ถ้ามีขนาดใหญ่มาก $\lim_{n\to\infty}$ → 1 task จบภายใน 1 quantum → กลายเป็น FIFO
  - What if time quantum is too short?
    - One instruction? → เล็กมากๆ ก็อาจเกิดกรณีคล้าย SJF


job
1 time quantum
1
2
3
ต่อคิว
รอบแรก
รอบสอง
1
2
3
1 quantum
Round Robin

Round Robin



Assume no overhead while switching

Round Robin vs. FIFO

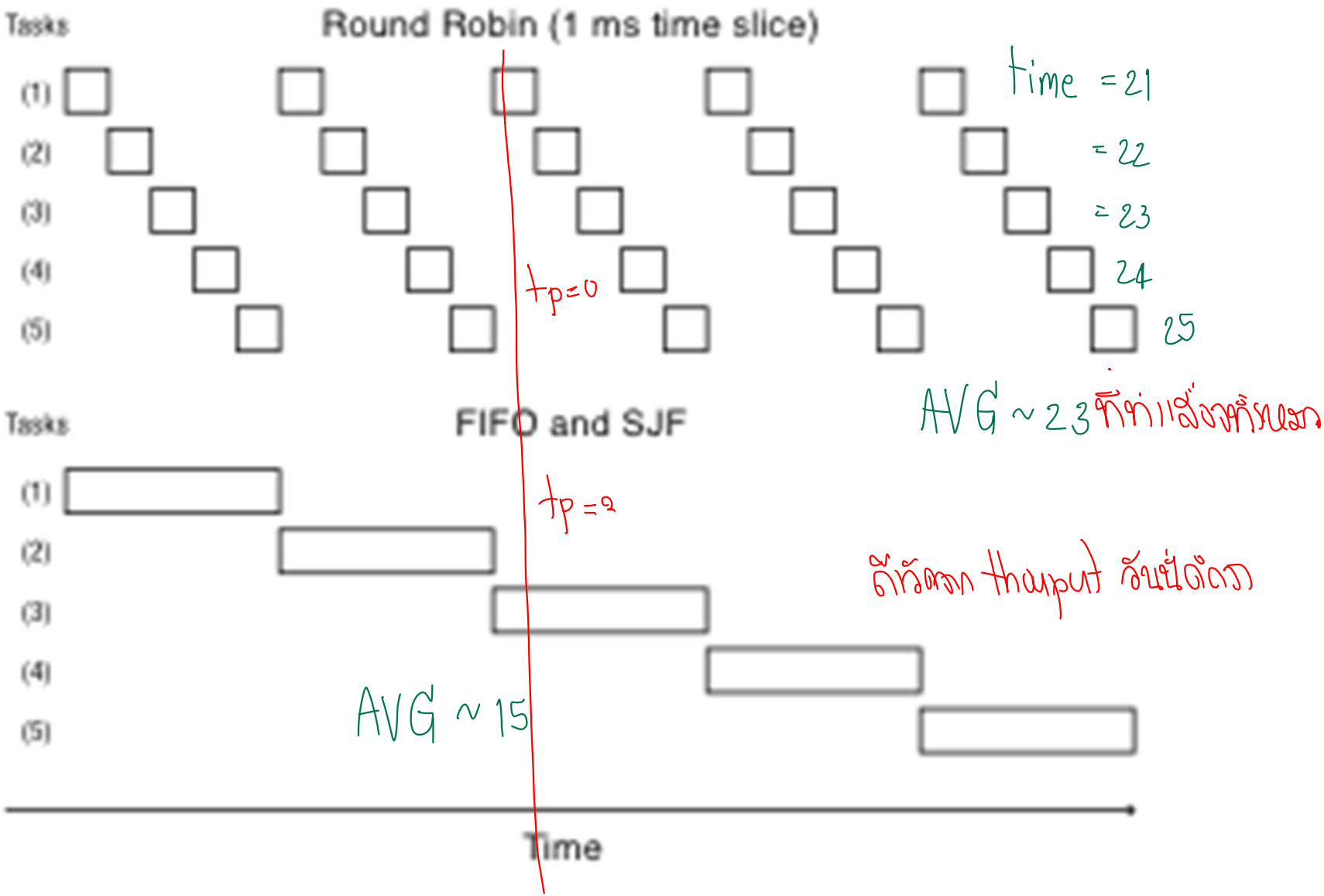
ไม่ใช้เวลา slice

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

↳ ทุกงานใช้เวลาเท่ากันหมด

↓

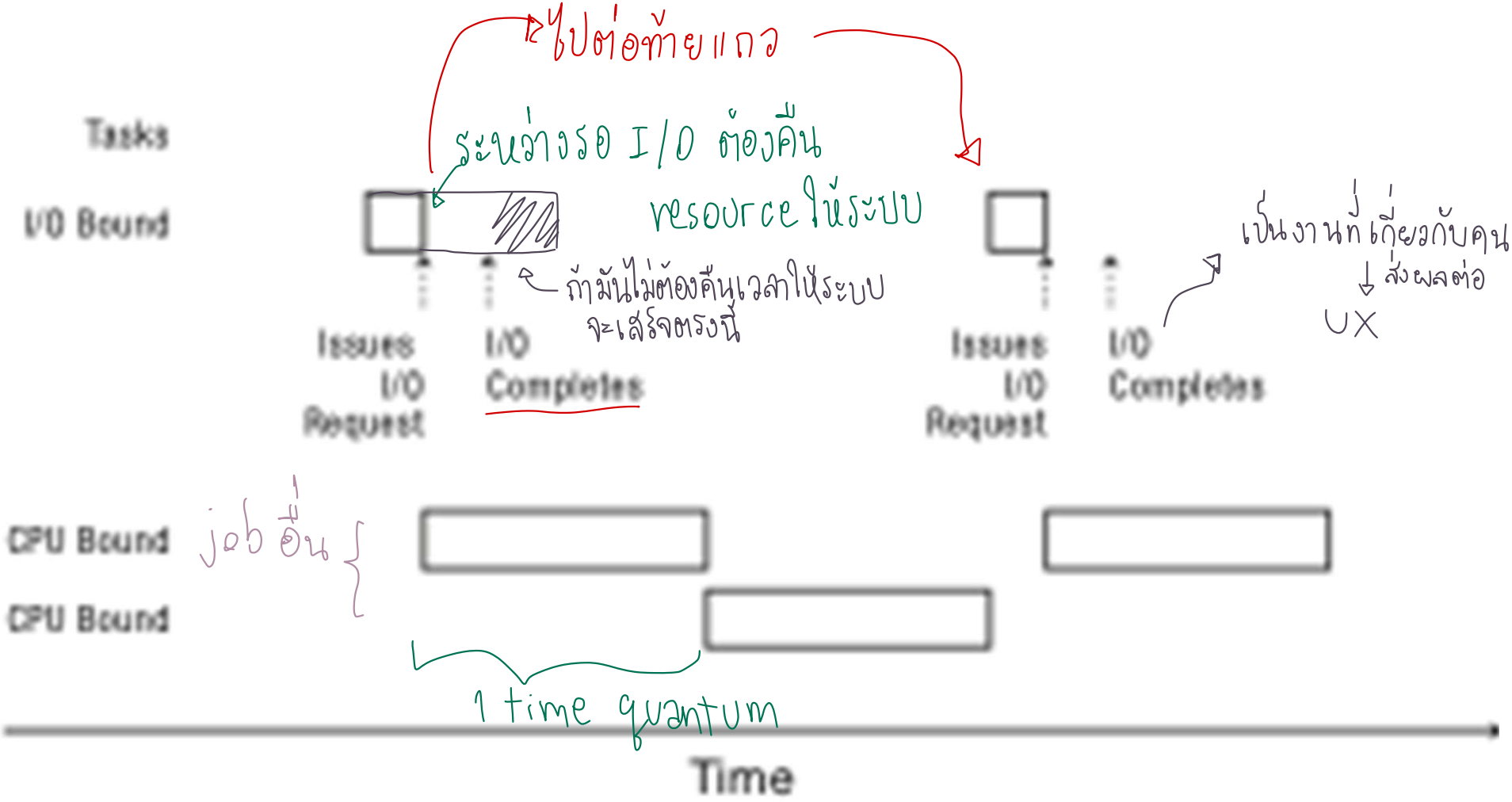
Round Robin vs. FIFO



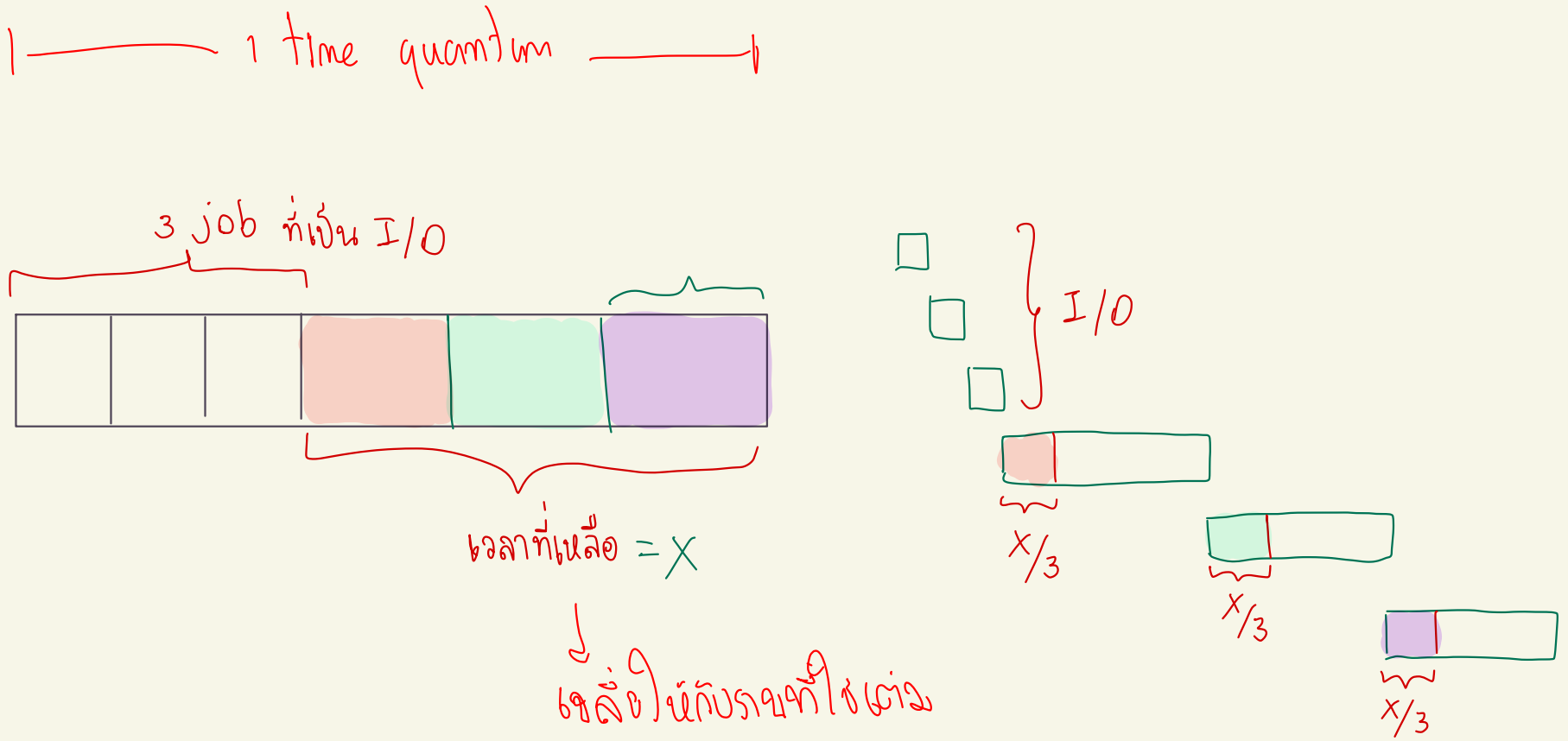
Round Robin = Fairness?

- Is Round Robin always fair?
- What is fair?
  - FIFO?
  - Equal share of the CPU?
  - 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



new cl → ให้ตัวที่ใช้เวลาน้อยกว่า 1 tq ก่อน ที่เหลือไปเฉลี่ย → ทำจริงไม่ได้ → รู้ได้ไงว่าใครน้อยกว่า 1 tq?


1 time quantum
3 job ที่เป็น I/O
เวลาที่เหลือ = X
เฉลี่ยให้กับงานที่เหลือ
I/O
X/3
X/3
X/3

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

Multi-level Feedback Queue (MFQ)

implement ได้จริง

- Goals: ประสิทธิภาพดี
  - Responsiveness ✓
  - Low overhead ✓
  - Starvation freedom ไม่มีก.แย่งคิวจน บาง job ไม่ได้ทำงาน
  - Some tasks are high/low priority มีก.กำหนด priority
  - Fairness (among equal priority tasks) ปัดๆ
- Not perfect at any of them! nobody perfect <3
  - Used in Linux (and probably Windows, MacOS)

love the way you are.

MFQ

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

MFQ

เมื่อมี job ใหม่ จะถือว่ามี priority สูงสุด

ใช้ time qt 10 ms

ถ้าเป็น I/O job จะใช้ 10 ms ไม่ครบ

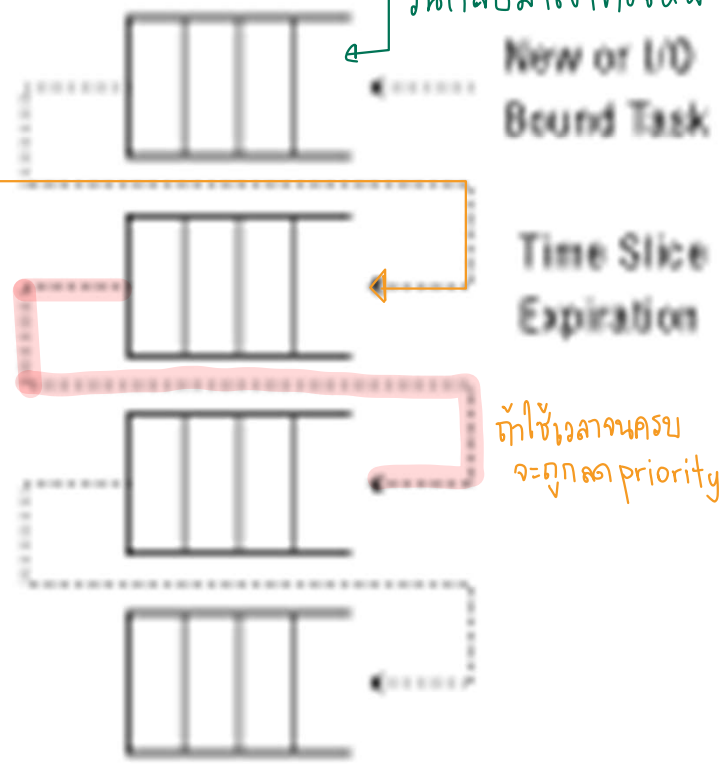
ถ้าไม่ได้เป็น I/O ใช้จนครบเวลา

time quantum

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

ใช้เวลาไม่ครบ จะถูกลด priority ลงมา

แตกต่างกัน (เลขสมมติ)



อาจเกิดปัญหาได้เช่นกัน

สามารถโกงโดยการใส่ I/O เข้าไปเพื่อให้ job ตัวเองเป็น prio 1 เสมอ

แก้ ↳ จะมี parameter ที่คอยนับเวลาที่ job เข้ามาใช้ cpu ถ้าถึงเวลาที่กำหนดไว้ จะลด priority ลงมา

↳ แต่อาจมี I/O บางตัว ที่ใช้เวลามากก็ได้ จะมี parameter อื่นมาแก้อีก

Uniprocessor Summary (1)

- FIFO is simple and minimizes overhead.
- If tasks are variable in size, then FIFO can have very poor average response time.
- If tasks are equal in size, FIFO is optimal in terms of average response time.
- Considering only the processor, SJF is optimal in terms of average response time.
- SJF is pessimal in terms of variance in response time.

Uniprocessor Summary (2)

- If tasks are variable in size, Round Robin approximates SJF.
- If tasks are equal in size, Round Robin will have very poor average response time.
- Tasks that intermix processor and I/O benefit from SJF and can do poorly under Round Robin.

Uniprocessor Summary (3)

- Max-Min fairness can improve response time for I/O-bound tasks.
- Round Robin and Max-Min fairness both avoid starvation.
- By manipulating the assignment of tasks to priority queues, an MFQ scheduler can achieve a balance between responsiveness, low overhead, and fairness.

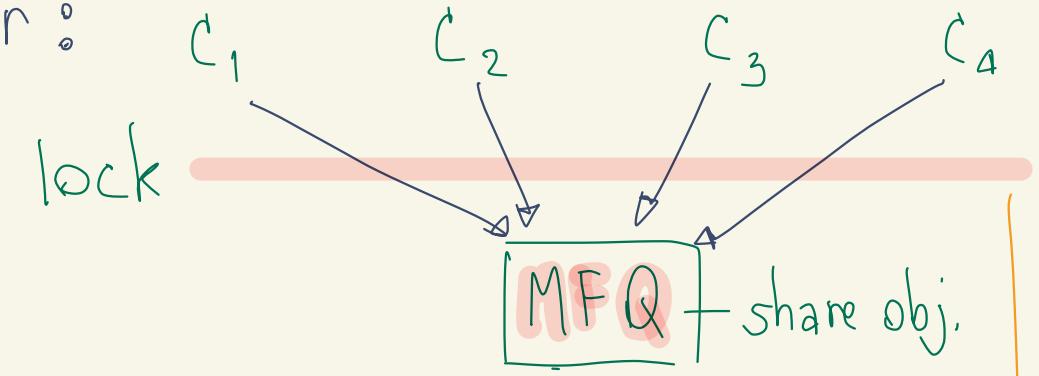
ระบบปฏิบ. ~> multi core

Multiprocessor Scheduling

- What would happen if we used MFQ on a multiprocessor? ปัญหา
  - Contention for scheduler spinlock
  - 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

switch งาน
ต้องไป load data ที่เกี่ยวข้อง

processor :



problem : 1. ดึงใช้ทีละ job

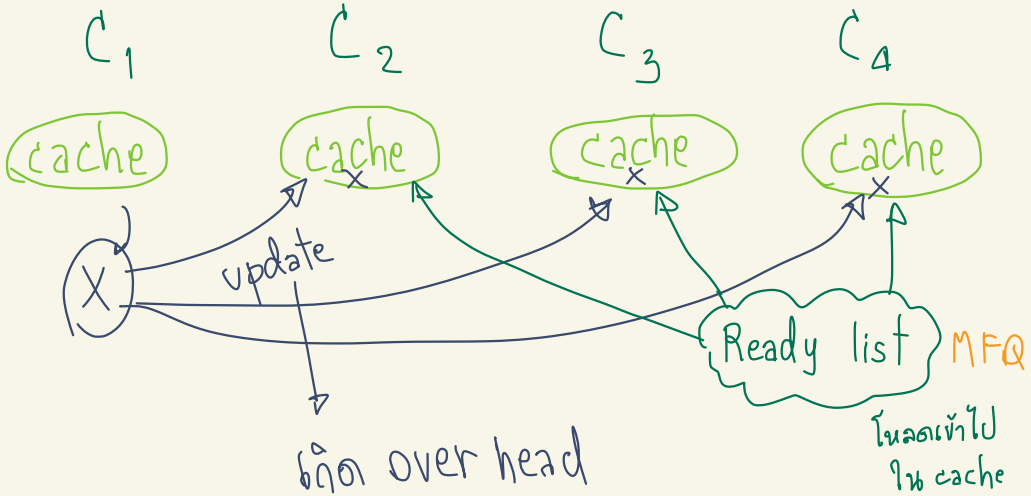
ถ้ายังงานให้ processor ทีละตัว

↓

ลักษณะเหมือน share obj.

2. cache

processor :



โหลดเข้าไปใน cache (check ค่า x ว่าเป็นปจบ.ยัง)

↳ มีผลต่อก.ทำงานของ scheduler

เกิด over head

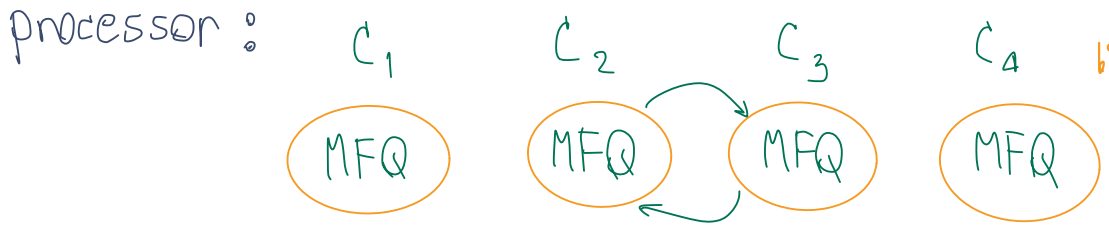
แต่ละ processor มี cache ของตัวเอง

↓

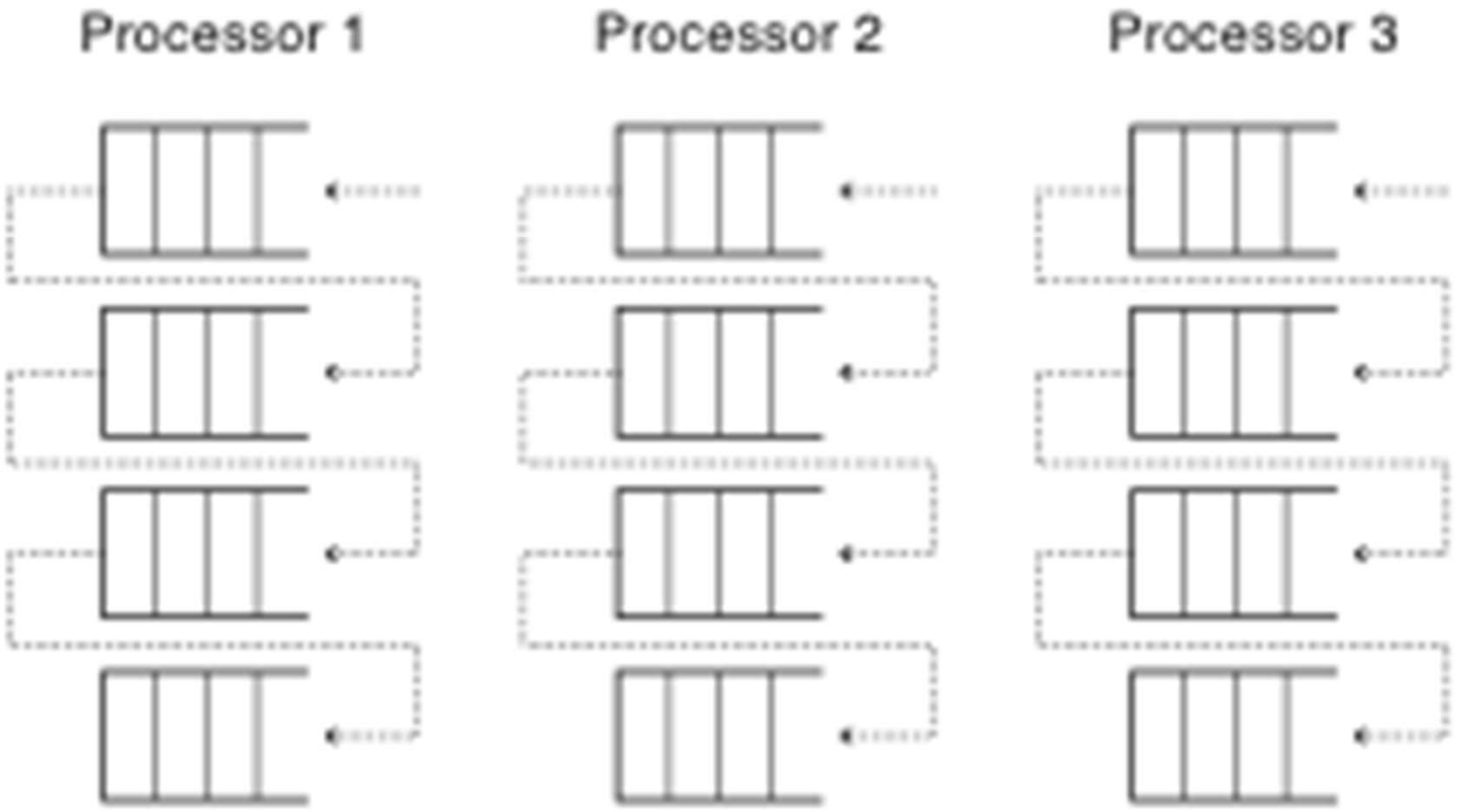
ถ้าทุกตัวต้องก. ทำงานกับ ตัวแปรเดียวกัน

Per-Processor Affinity Scheduling

- Each processor has its own ready list → แต่ละ = processor มี ready list (MFQ) เป็นของตัวเอง
  - Protected by a per-processor spinlock
- Put threads back on the ready list where it had most recently run ↓ ไม่ต้องไป check ว่า update ที่สุดยัง
  - Ex: when I/O completes, or on Condition->signal
- Idle processors can steal work from other processors กรณี processor ที่ว่างงานอยู่ สามารถไปขโมยงานจาก list ของตัวอื่นได้ ↓ เสร็จแล้ว ต้องเอาคืนไปที่ list นั้นด้วย



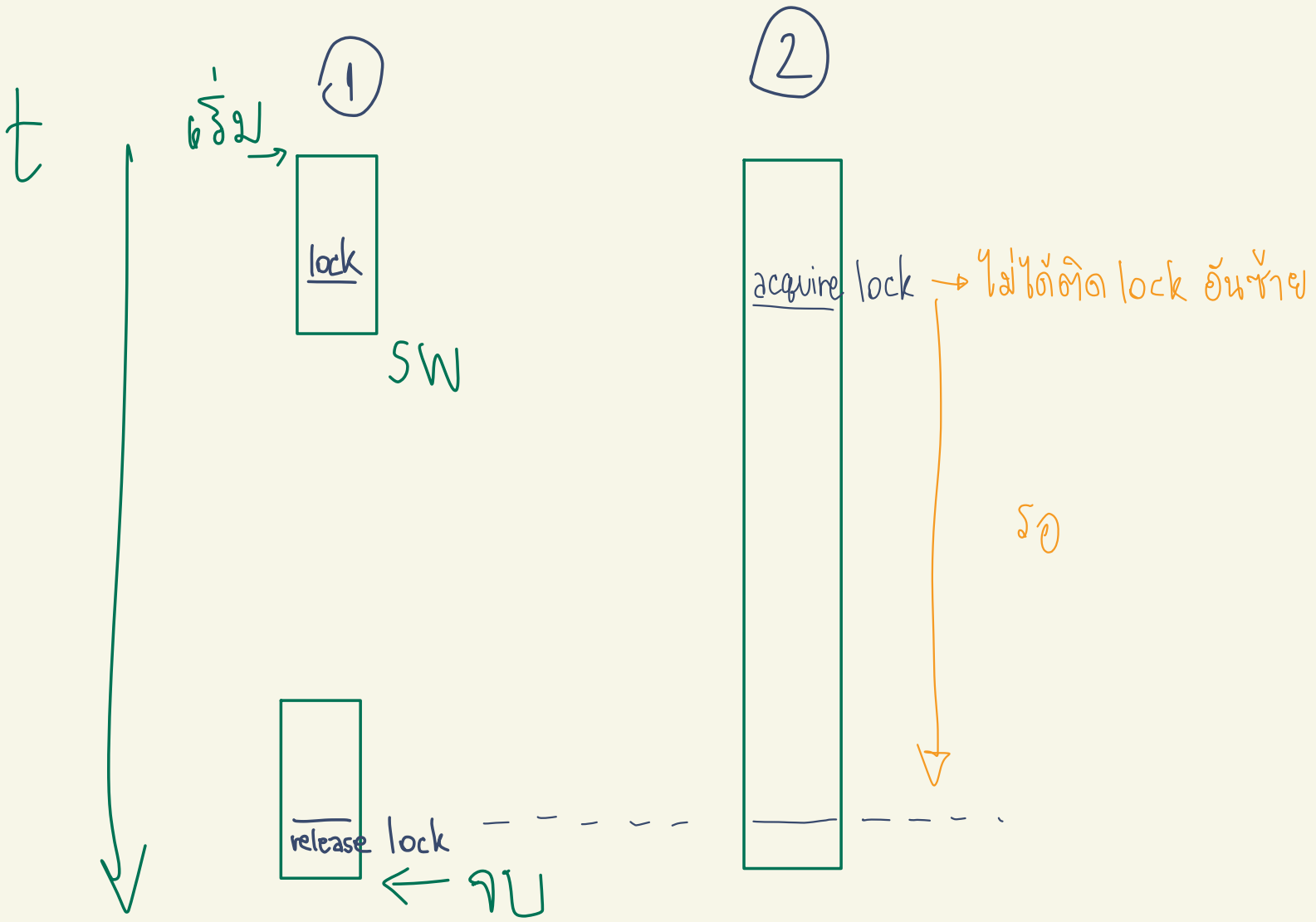
Per-Processor Multi-level Feedback with Affinity Scheduling



การแตก thread ให้ทำงาน ขนานกันให้ได้มากที่สุด

Scheduling Parallel Programs

- What happens if one thread gets time-sliced while other threads from the same program are still running?
  - Assuming program uses locks and condition variables, it will still be correct
  - What about performance?

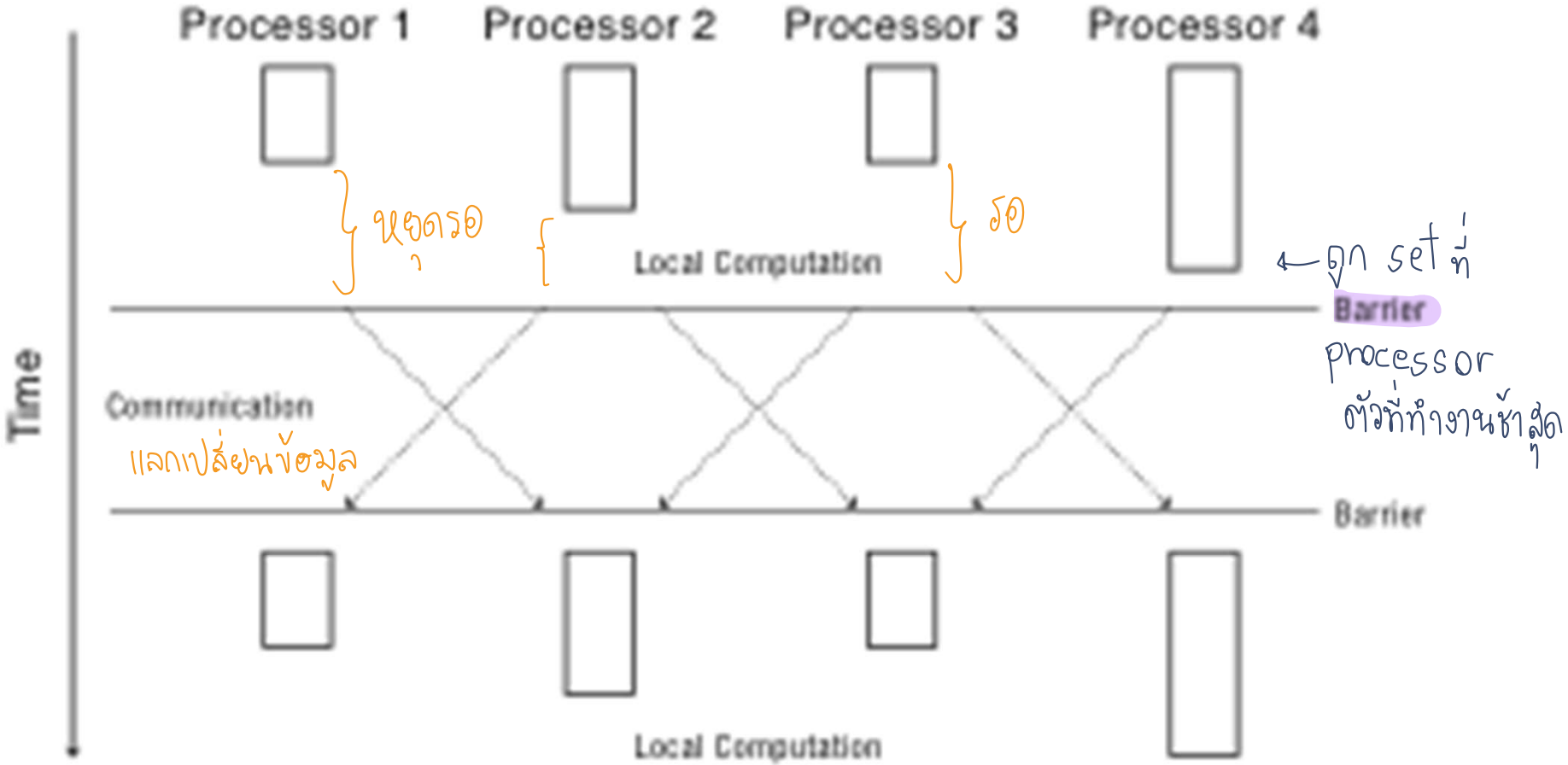

t
1
2
เริ่ม
lock
SW
acquire lock
ไม่ได้ติด lock อันซ้าย
รอ
release lock
จบ

หลักใช้เขียน program แบบ Parallel

Bulk Synchronous Parallelism

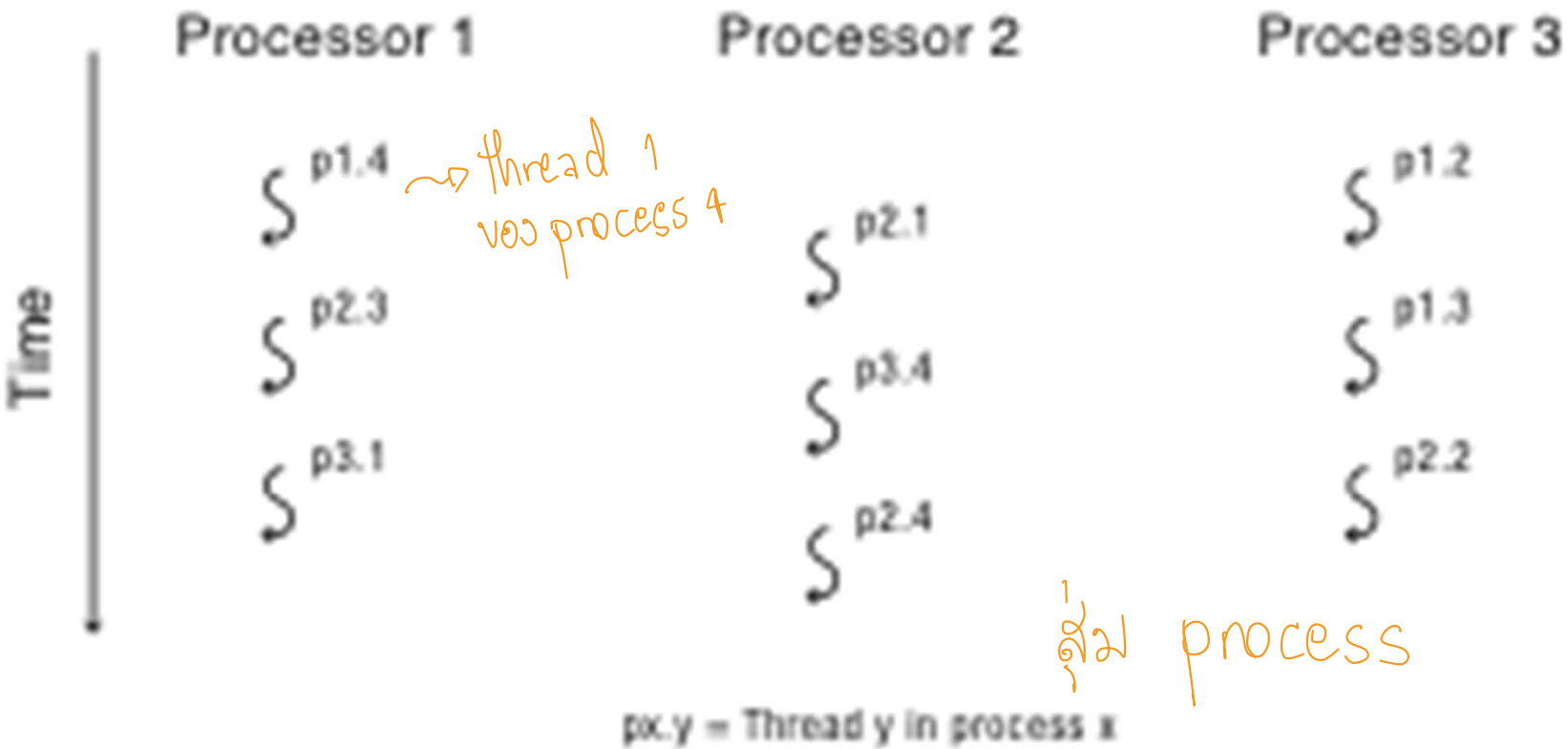
- Loop at each processor:
  - Compute on local data (in parallel) คำนวณ
  - Barrier ~> set ส่ง data ที่คำนวณแล้ว แลกเปลี่ยนข้อมูลกัน
  - Send (selected) data to other processors (in parallel)
  - Barrier ~> ปลด
- Examples:
  - MapReduce
  - Fluid flow over a wing
  - Most parallel algorithms can be recast in BSP
    - Sacrificing a small constant factor in performance

Tail Latency

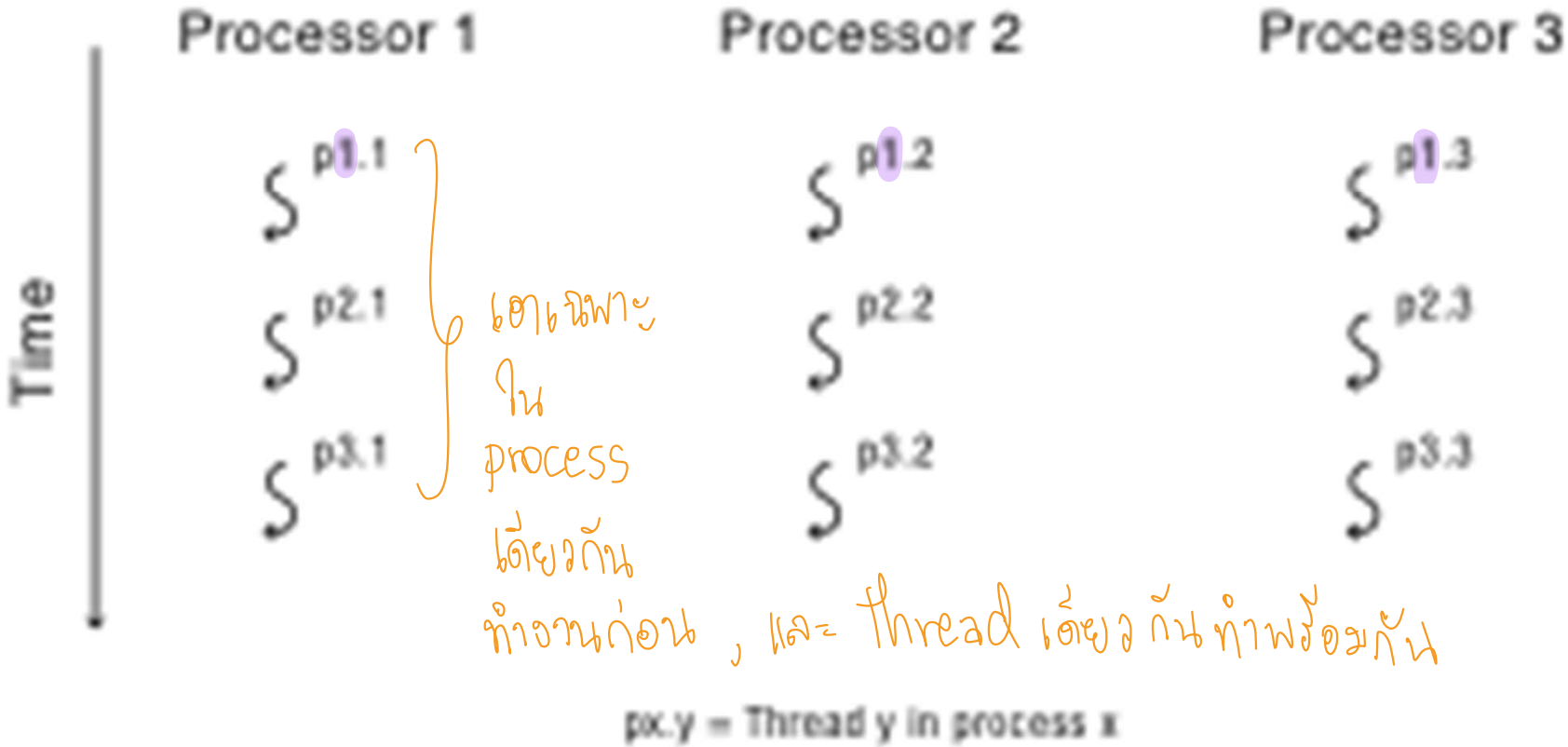


Scheduling Parallel Programs

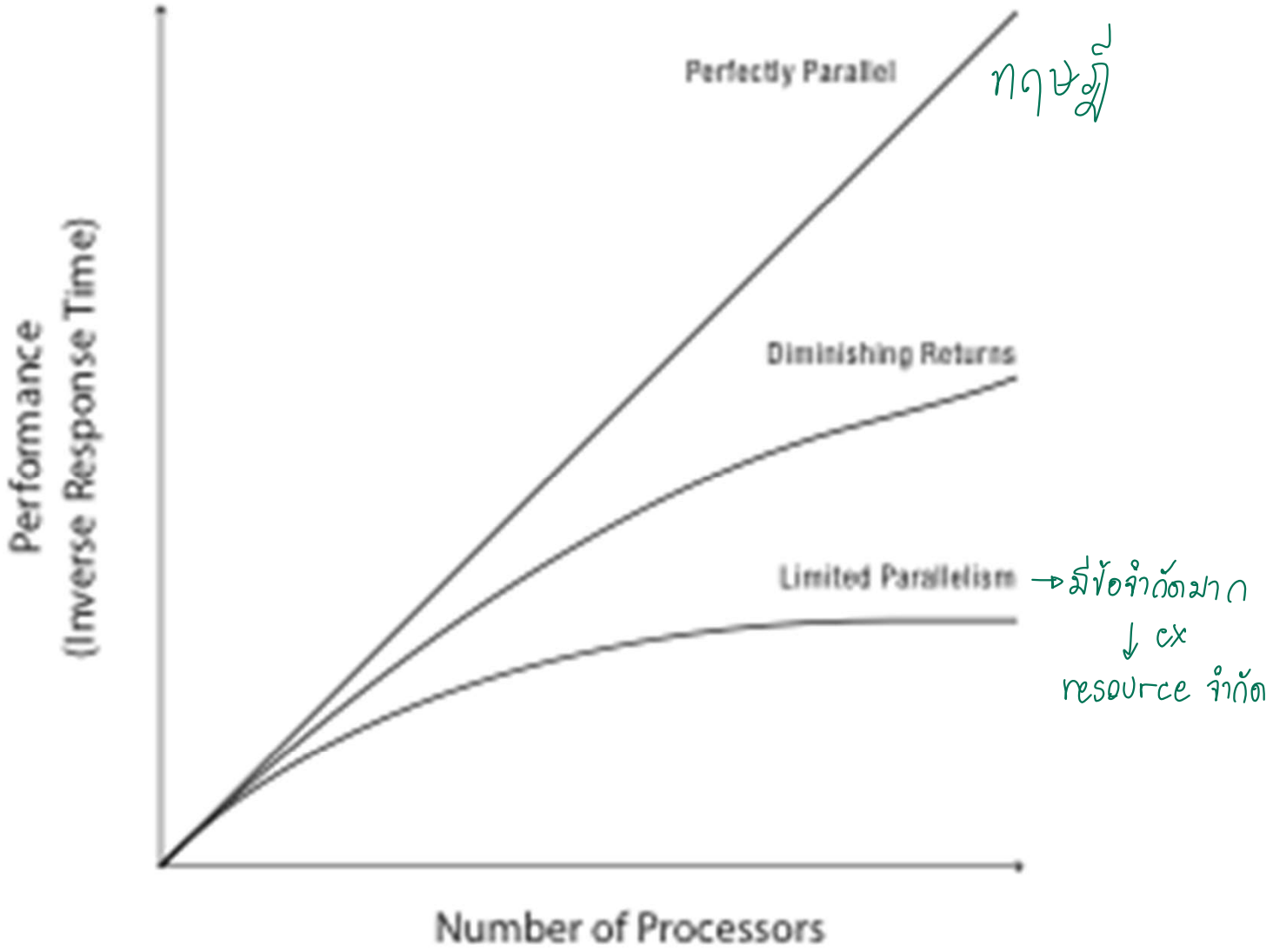
Oblivious: each processor time-slices its ready list independently of the other processors



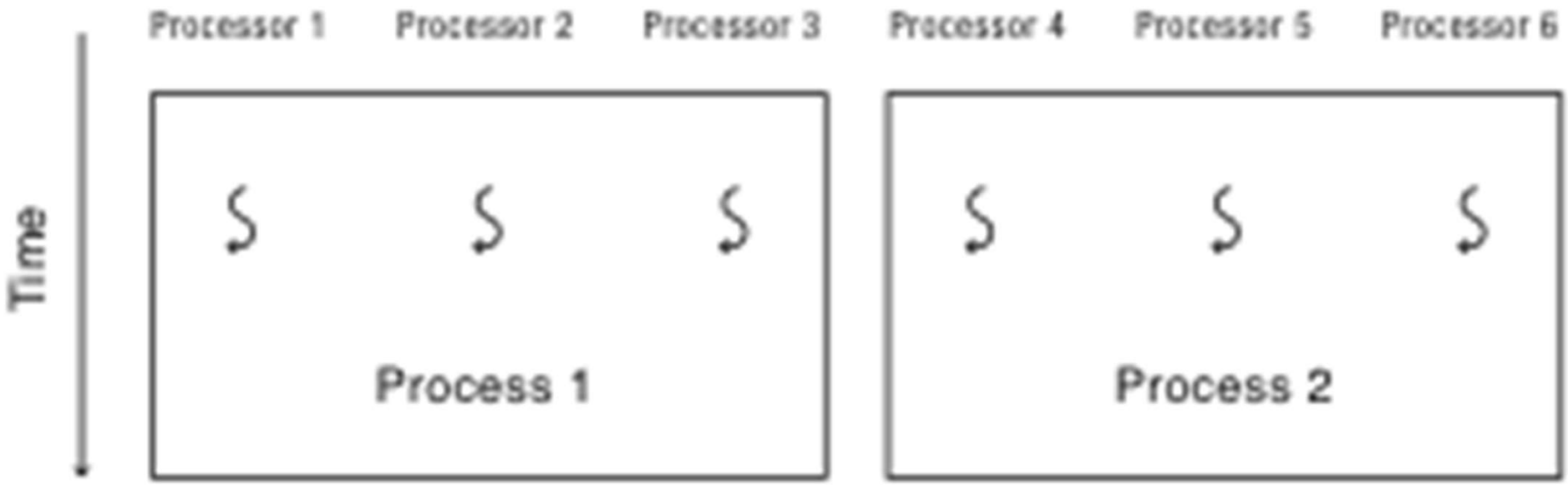
Gang Scheduling



Parallel Program Speedup

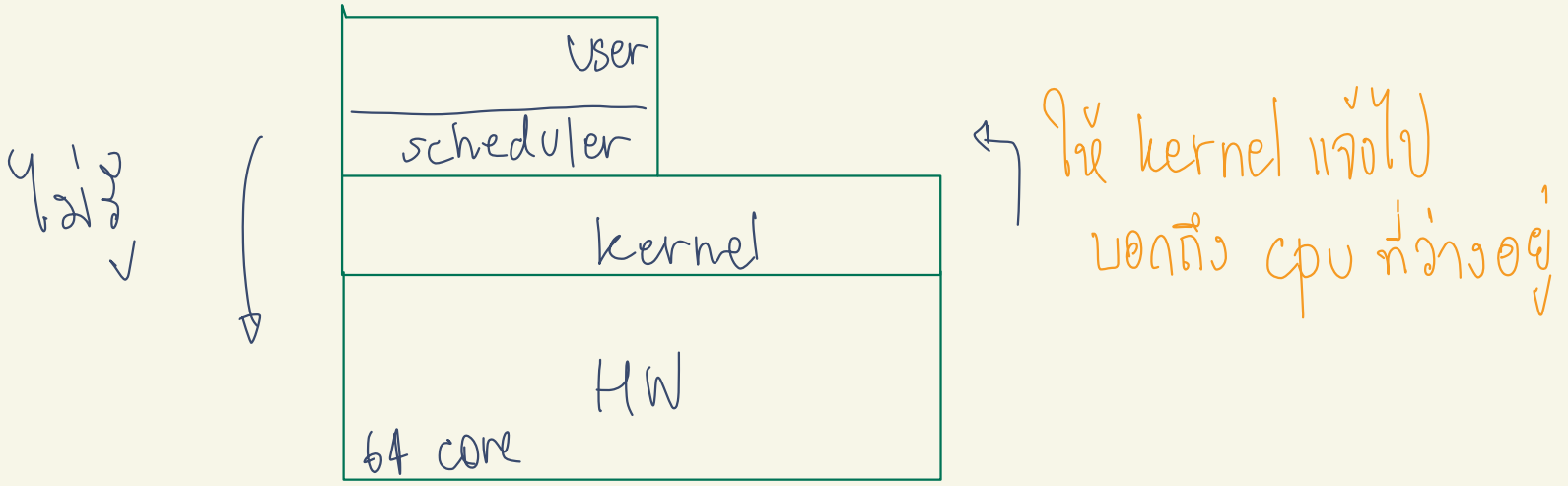


Space Sharing



Scheduler activations: kernel tells each application its # of processors with upcalls every time the assignment changes

▸ ย้ายไปอยู่บน user space → ไม่รู้จักสถานภาพของ hw


ไม่มี
User
scheduler
kernel
HW
64 core
ให้ kernel แจ้งไป
บอกถึง cpu ที่ว่างอยู่

ไม่สอน ไม่ออกสอบ

Queueing Theory

- Can we predict what will happen to user performance:
  - If a service becomes more popular?
  - If we buy more hardware?
  - If we change the implementation to provide more features?

เป็น stat math → วัดประสิทธิภาพ ของ การจัด ตาราง
คำนวณ avg. time ได้ละเอียด

Queueing Model

Assumption: average performance in a stable system, where the arrival rate (ƛ) matches the departure rate (μ)

Definitions

- Queueing delay (W): wait time
  - Number of tasks queued (Q)
- Service time (S): time to service the request
- Response time (R) = queueing delay + service time
- Utilization (U): fraction of time the server is busy
  - Service time * arrival rate (ƛ)
- Throughput (X): rate of task completions
  - If no overload, throughput = arrival rate

Little's Law

$$N = X * R$$

N: number of tasks in the system

Applies to *any* stable system – where arrivals match departures.

Question

Suppose a system has throughput (X) = 100 tasks/s, average response time (R) = 50 ms/task

- How many tasks are in the system on average?
- If the server takes 5 ms/task, what is its utilization?
- What is the average wait time?
- What is the average number of queued tasks?

Question

- From example:

  X = 100 task/sec

  R = 50 ms/task

  S = 5 ms/task

  W = 45 ms/task

  Q = 4.5 tasks

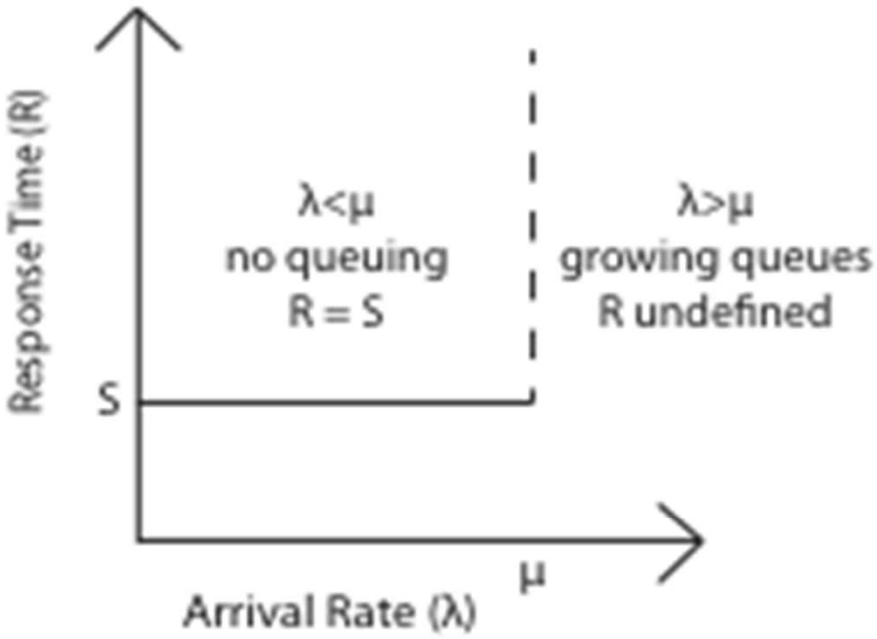
- Why is W = 45 ms and not 4.5 * 5 = 22.5 ms?
  - Hint: what if S = 10ms? S = 1ms?

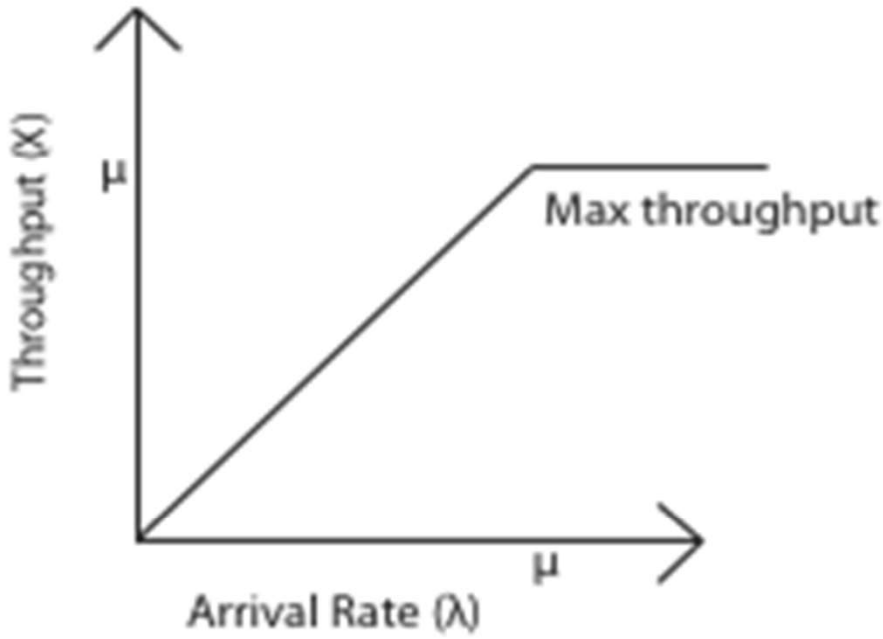
Queueing

- What is the best case scenario for minimizing queueing delay?
  - Keeping arrival rate, service time constant

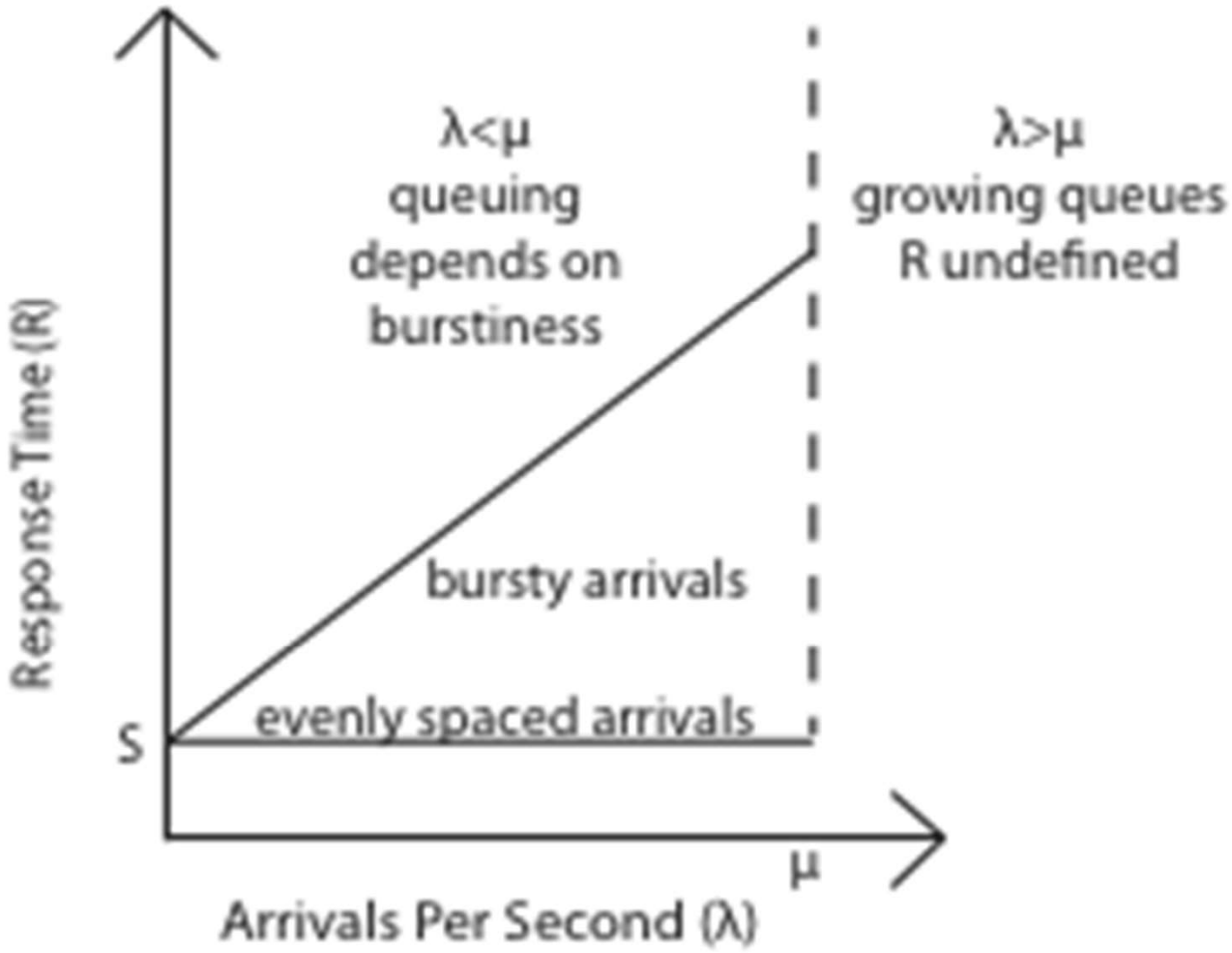
- What is the worst case scenario?

Queueing: Best Case





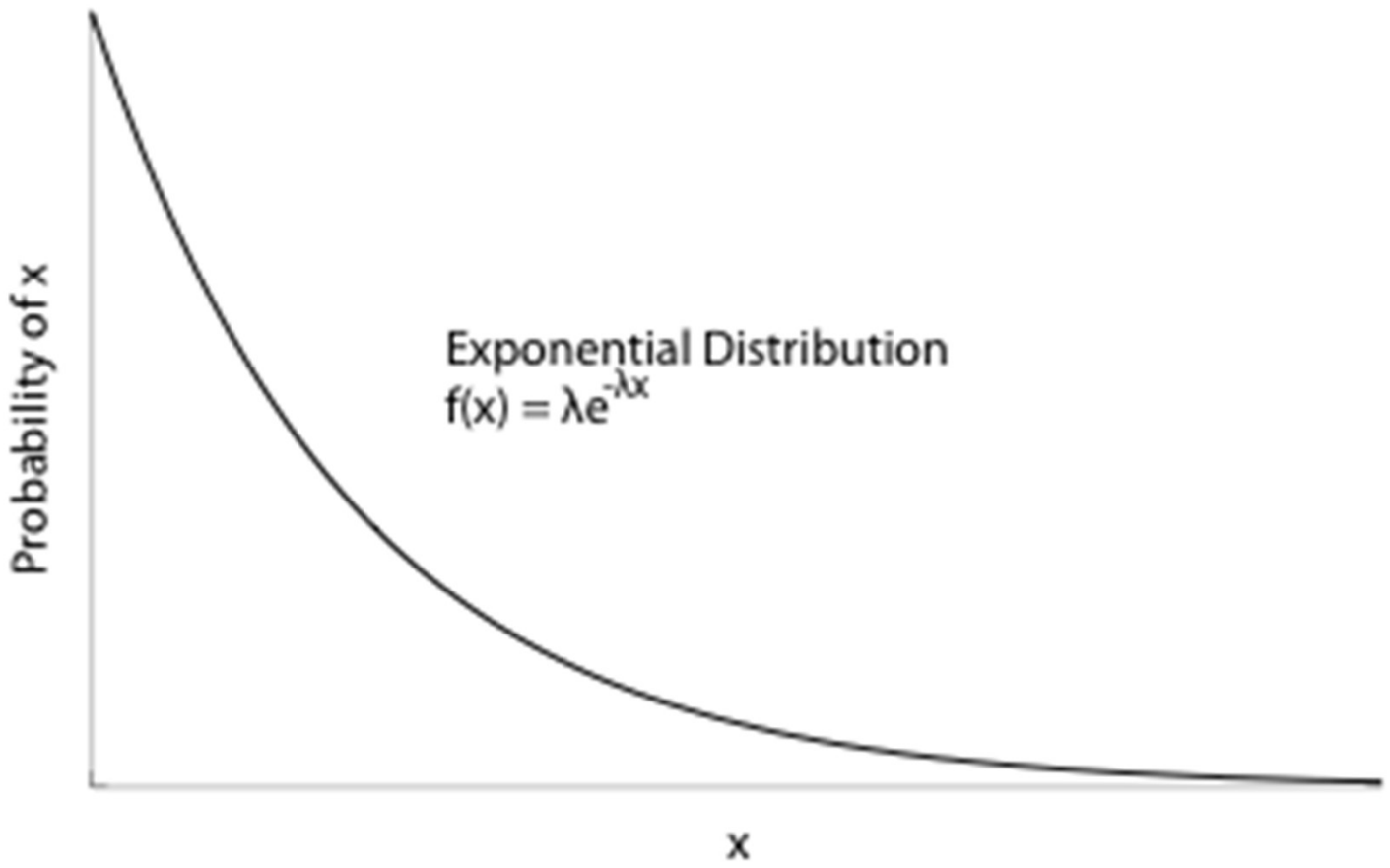
Response Time: Best vs. Worst Case



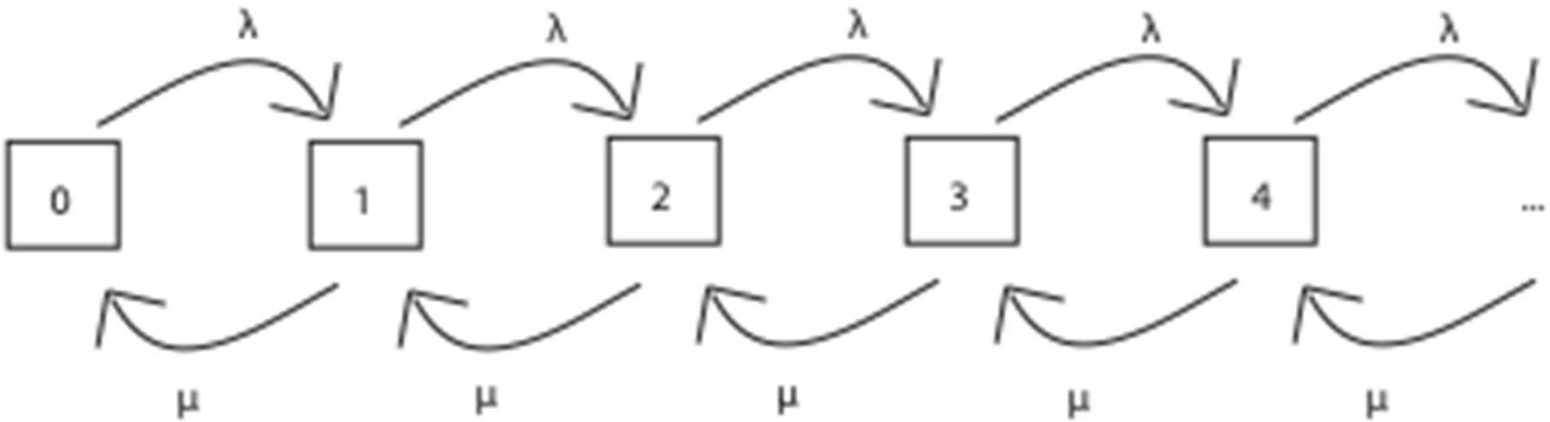
Queueing: Average Case?

- What is average?
  - Gaussian: Arrivals are spread out, around a mean value
  - Exponential: arrivals are memoryless
  - Heavy-tailed: arrivals are bursty
- Can have randomness in both arrivals and service times

Exponential Distribution

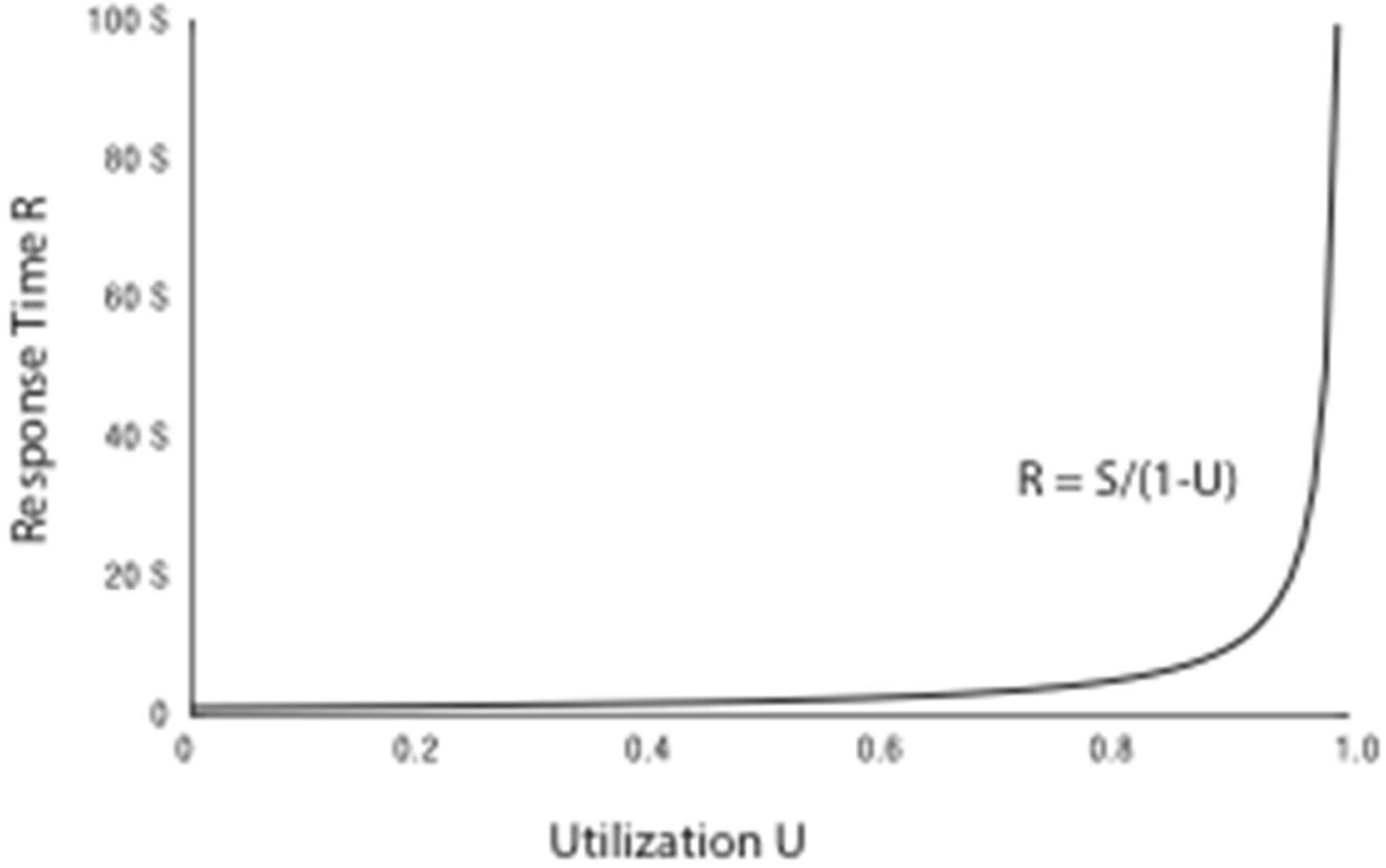


Exponential Distribution



Permits closed form solution to state probabilities, as function of arrival rate and service rate

Response Time vs. Utilization



Question

- Exponential arrivals: R = S/(1-U)
- If system is 20% utilized, and load increases by 5%, how much does response time increase?

- If system is 90% utilized, and load increases by 5%, how much does response time increase?

Variance in Response Time

- Exponential arrivals
  - Variance in R = S/(1-U)^2
- What if less bursty than exponential?
- What if more bursty than exponential?

What if Multiple Resources?

- Response time =

  Sum over all i

    Service time for resource i /

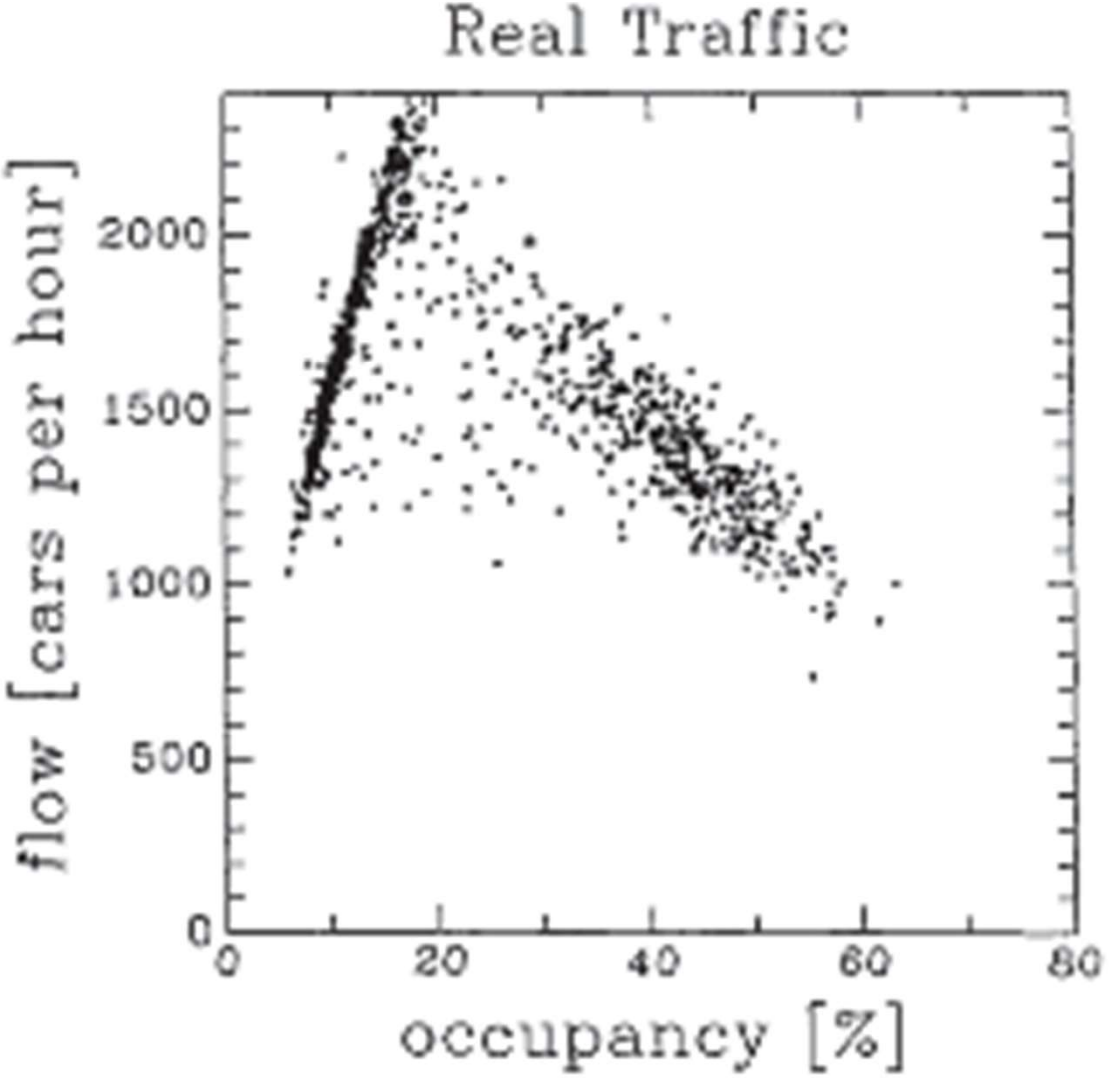
      (1 – Utilization of resource i)

- Implication
  - If you fix one bottleneck, the next highest utilized resource will limit performance

Overload Management

- What if arrivals occur faster than service can handle them
  - If do nothing, response time will become infinite
- Turn users away?
  - Which ones? Average response time is best if turn away users that have the highest service demand
  - Example: Highway congestion
- Degrade service?
  - Compute result with fewer resources
  - Example: CNN static front page on 9/11

Highway Congestion (measured)



Why Do Metro Buses Cluster?

- Suppose two Metro buses start 15 minutes apart
  - Why might they arrive at the same time?