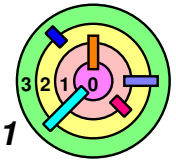
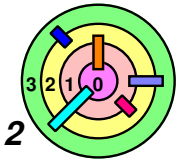
Housekeeping (Lecture 11 - 7/8/2025)

- PA4 is due at 11:45pm on Tuesday, 7/15/2025
  - if you have code from current or a previous semester, *do not look at/copy/share* any code from it
    - it's best if you just get rid of it
  - if you include files that's not part of the original "`make pa4-submit`" command, the grader will delete them
  - PA4 is probably the most difficult assignment
    - it's probably a good idea to start early
    - the last part (graph) may be time-consuming
      - unless you have found a tool that works, you might want to leave that part until the very end so you don't have to re-do the graphs

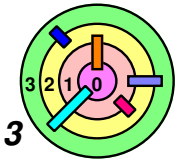


Housekeeping (Lecture 11 - 7/8/2025)

- *Grading guidelines* is the *ONLY* way we will grade and we can only grade on a *standard 32-bit Ubuntu Linux 16.04 inside VirtualBox/UTM* or on AWS Free Tier
  - although not recommended, you can do your development on a different platform
    - you must test your code on a "standard" platform since it's the *only* platform the grader is allowed to grade on
- If you make a submission, read and understand the *ticket* in the *web page* and save the web page as PDF as a record of your submission
  - make sure to *"Verify Your Ticket"* and *"Verify Your Submission"*

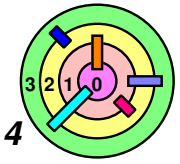


(6.5) Deadlock (cont...)



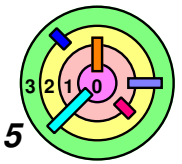
Transactions: Which Transaction To Abort?

- If a bunch of threads are in a deadlock, which thread should you abort its transaction?
  - typically, you would abort the transaction of the *youngest* thread
    - to maximize the change that some threads will finish/commit
- *Wound-Wait:*
  - if a younger transaction needs a resource held by a older transaction, just wait
  - if an older transaction needs a resource held by a younger transaction, abort the younger transaction and preempt the resource held by the younger transaction



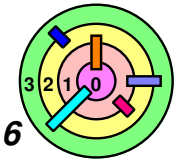
Detecting Deadlock

- Deadlock detection is difficult to implement
  - oftentimes, systems would take a more conservative approach to detect a possible deadlock instead
    - *false positive* is possible: a non-deadlocked thread is incorrectly classified as deadlocked
    - e.g., in the old telephone network, if the connection setup failed, it would abort call and ask the user to try again



Resource Allocation Graph

- If there are serveral resources and one thread can hold each resource at a time (e.g., a microphone, a webcam, a mutex), we can detect a deadlock by analyzing a simple *resource allocation graph*
  - each *thread* and each *resource* is represented by a node
  - there is a directed edge *from a resource to a thread* if the resource is *owned by* the thread
  - there is a directed edge *from a thread to a resource* if the thread is *waiting for* the resource
  - there is a deadlock *if and only if* there is a cycle in the resource allocation graph
- If there are multiple instances of some resources, then we represent a resource with K interchangeable instances as a node with K connection points
  - there is a directed edge *from a thread to a resource* if the thread is waiting for *one unit* of that resource
  - a cycle in the resource allocation graph is a *necessary but not a sufficient condition* for deadlock

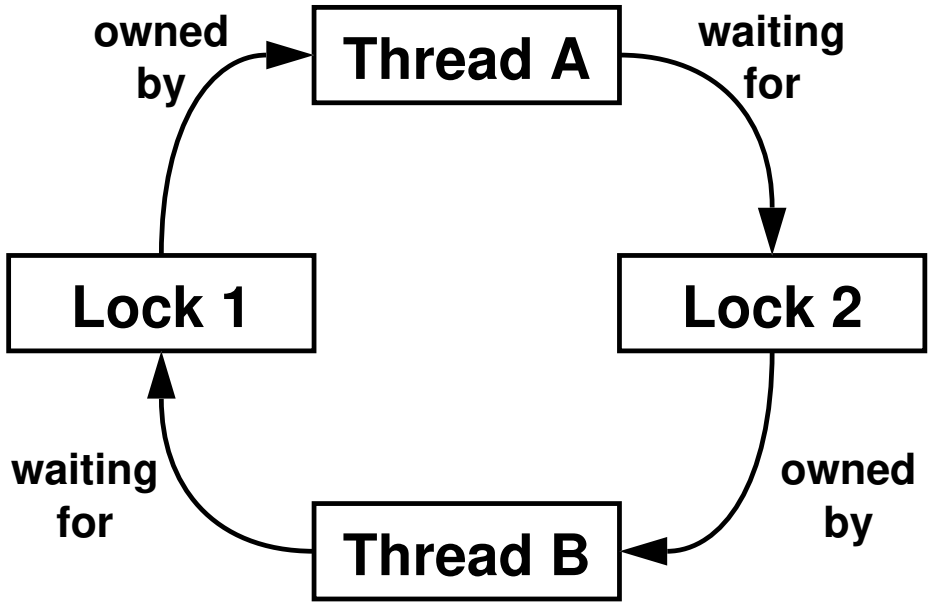


Resource Allocation Graph

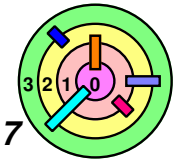
Ex: if each resource can only be held by at most one thread at a time (e.g., mutex)

```
thread A:
  lock1.acquire();
  lock2.acquire();
  ...

thread B:
  lock2.acquire();
  lock1.acquire();
  ...
```

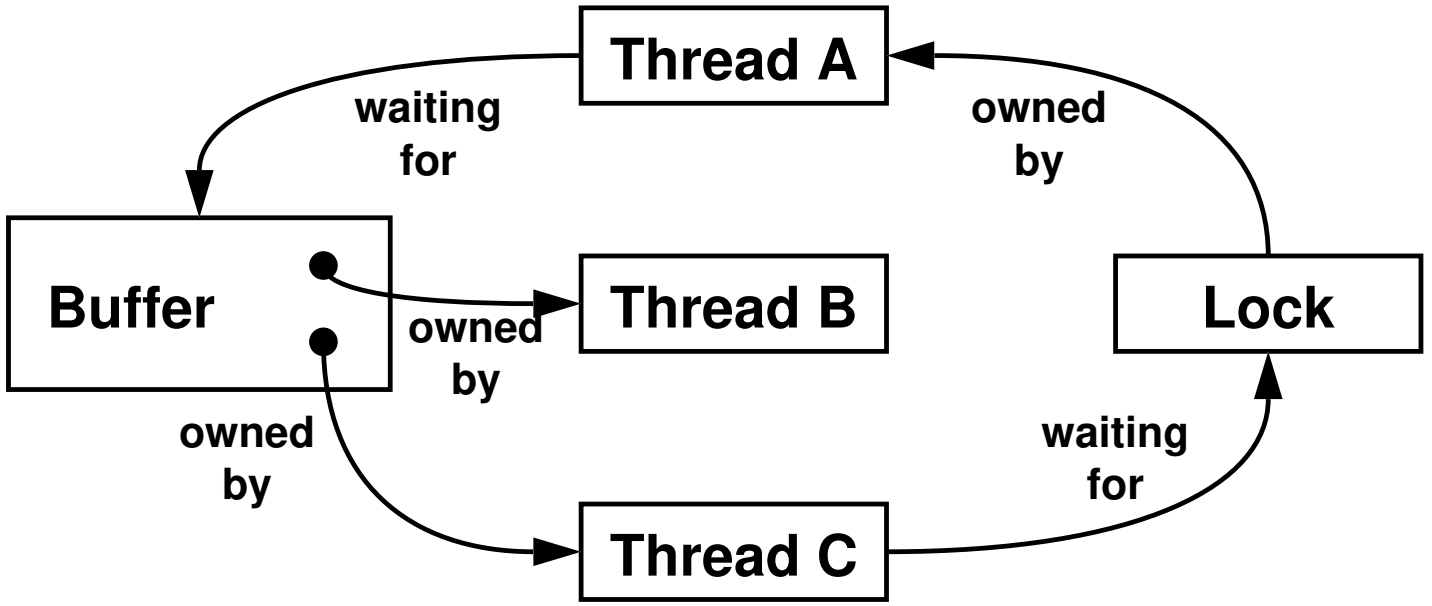


- there is a cycle in this resource allocation graph, it *may* deadlock
  - i.e., a cycle in the resource allocation graph is a *necessary but not a sufficient condition* for deadlock



Resource Allocation Graph

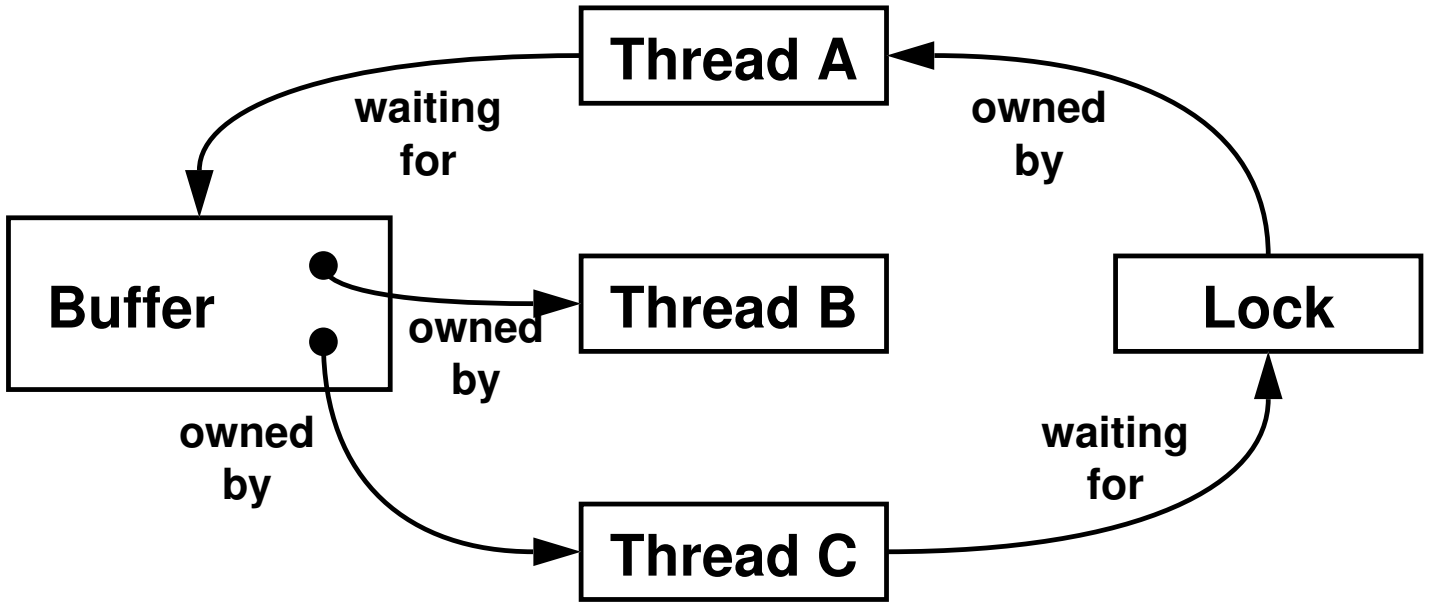
Ex: if there are multiple instances of some resources



- there is a cycle in this resource allocation graph, but it *may not* deadlock

Resource Allocation Graph

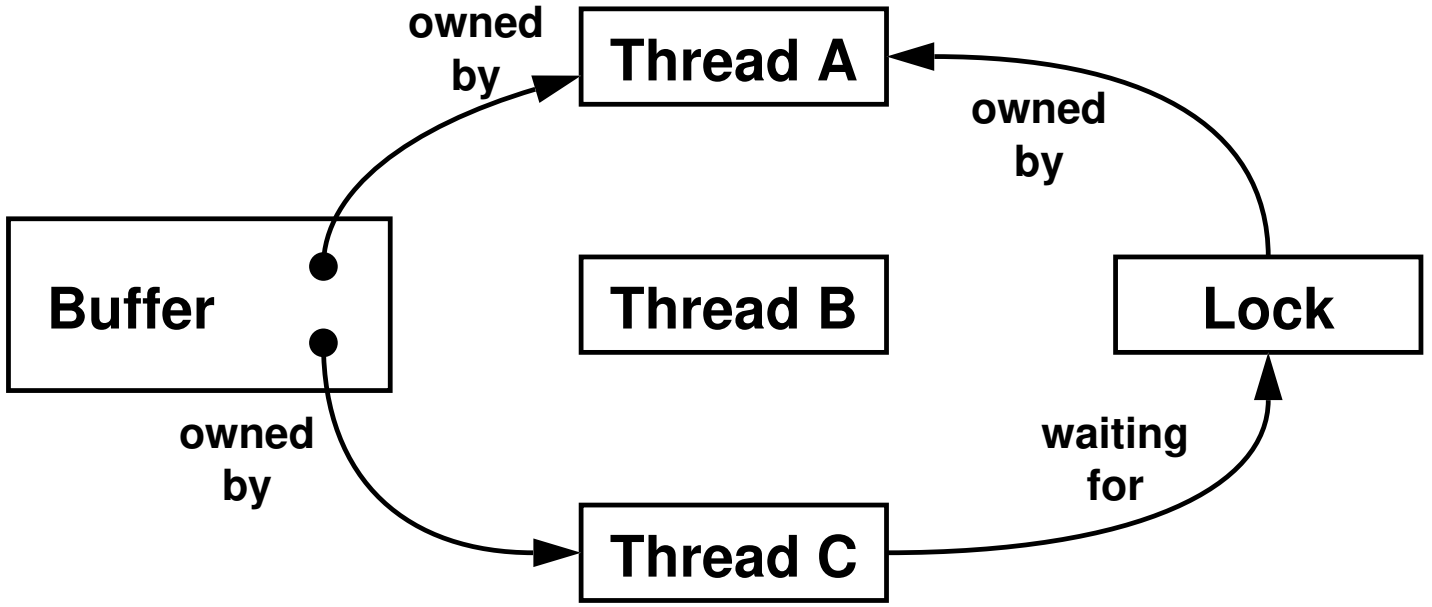
Ex: if there are multiple instances of some resources



- there is a cycle in this resource allocation graph, but it *may not* deadlock
  - if thread B releases its buffer and thread A gets the buffer, threads A and C may both complete their tasks

Resource Allocation Graph

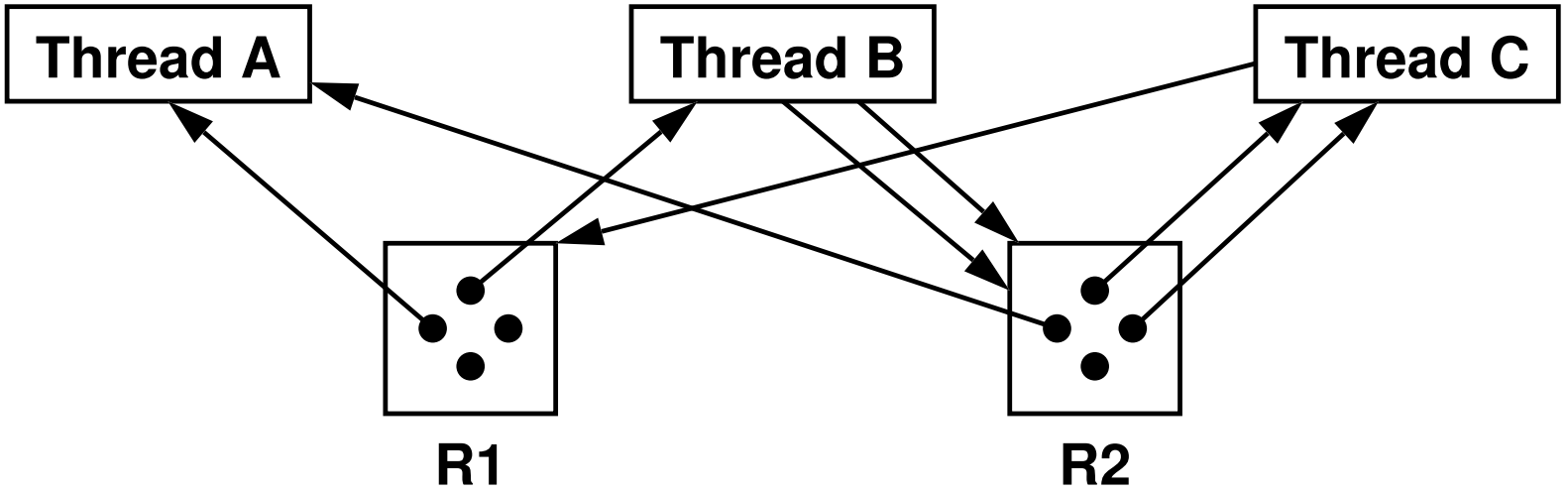
Ex: if there are multiple instances of some resources



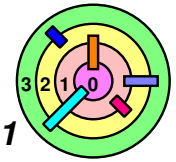
- there is a cycle in this resource allocation graph, but it *may not* deadlock
  - if thread B releases its buffer and thread A gets the buffer, threads A and C may both complete their tasks

Resource Allocation Graph

Ex: if there are multiple instances of some resources

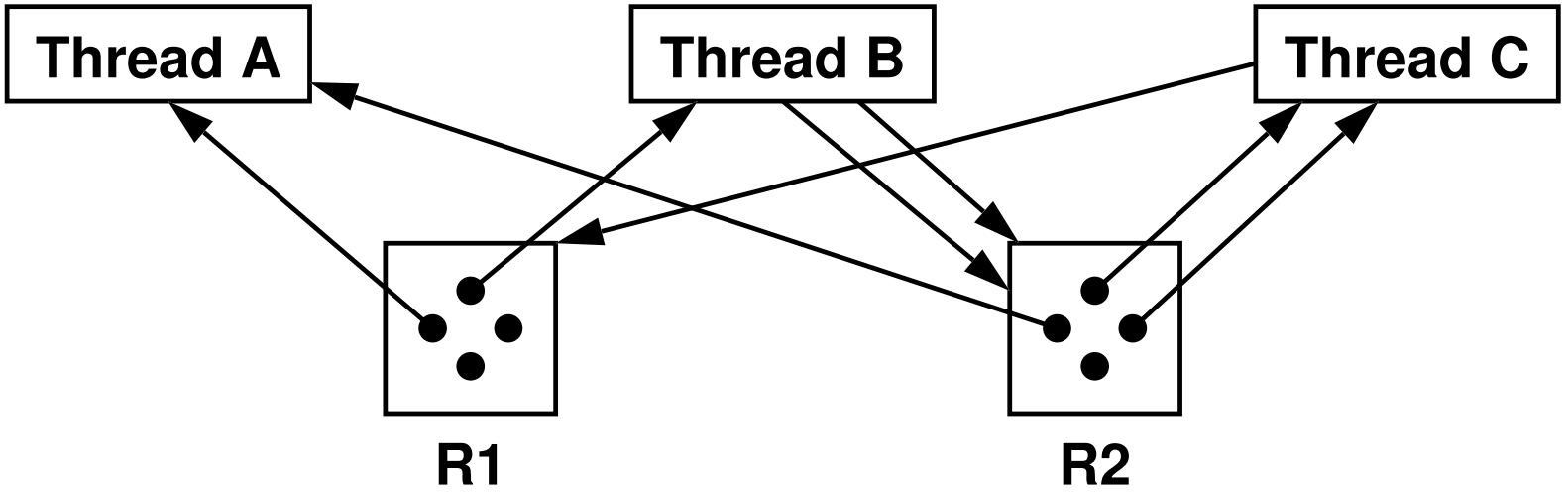


- who is blocked?
- is there a cycle?
- can this deadlock?

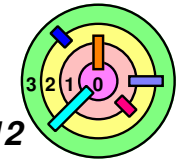


Resource Allocation Graph

Ex: if there are multiple instances of some resources

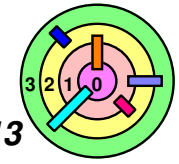


- who is blocked?
  - thread B
- is there a cycle?
  - thread B → R2 → thread C → R1 → thread B
- can this deadlock?
  - need to look at this very carefully to be sure

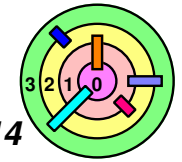


What Do Real OS Do?

- Not much
  - up to programmers to write code that doesn't deadlock
  - some might do detection, but no recovery

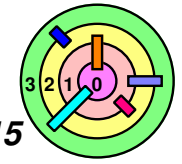


(6.6) Non-Blocking Synchronization

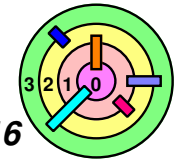


Non-Blocking Synchronization

- Goal: data structures that can be read/modified without acquiring a lock
  - no lock contention
  - no deadlock
- General approach using atomic Compare-And-Swap (similar to the atomic Test-And-Set machine instruction):
  - create a private copy of a shared data structure
  - modify the private copy
  - swap old pointer with the next pointer using `CAS()`
  - restart if pointer has changed
- This is tricky stuff and should be left to the experts!
  - e.g., google this: "non-blocking algorithm FIFO queue" to see some research papers



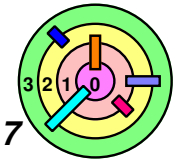
Extra Slides



Ch 7: Scheduling

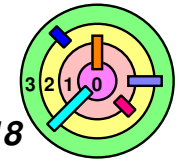
Bill Cheng

http://merlot.usc.edu/william/usc/



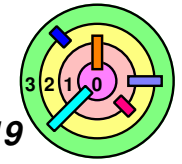
Main Points

- Scheduling policy: what to do next when there are multiple threads ready to run
- Definitions for response time, throughput, predictability
- Uniprocessor policies
  - FIFO, round robin, optimal
  - multilevel feedback as approximation of optimal
- Multiprocessor policies
  - affinity scheduling, gang scheduling
- Queueing theory
  - can you predict/improve a system's response time?



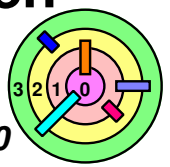
Example

- You manage a web site that suddenly become wildly popular, what should you do?
  - buy more hardware?
  - implement a different scheduling policy?
  - turn away some users and which ones (to make the rest of the users happy)?
- How much worse will performance get if the web site becomes even more popular?



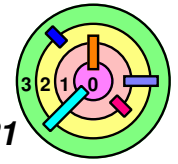
Definitions

- ***Task/Job:*** **a user request, e.g., mouse click, web request, shell command, ...**
  - **a thread can perform many tasks**
  - **this is not a very precise definition**
- ***Service time:*** **time to complete a task, *assuming no waiting***
- ***Latency/Response time:*** **time it takes for a job take to complete (including waiting time)**
- ***Throughput:*** **number of jobs that can be completed per unit of time**
- ***Overhead:*** **amount of extra work done by the scheduler to switch jobs**
- ***Fairness:*** **how equal is the performance received by different users?**
  - **starvation: lack of progress for one job, due to resource given to higher priority job**



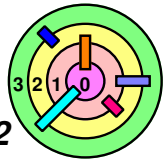
Definitions

- ***Predictability:*** **how consistent is the performance over time?**
  - **low variance means more predictable**
- ***Workload:*** **set of jobs for system to perform**
- ***Compute/CPU-bound job:*** **jobs that only (or mostly) use the processor**
- ***I/O-bound job:*** **jobs that rarely use the processor and spend most of its time waiting for I/O operations to complete**
- ***Work-conserving scheduler:*** **never leaves the processor idle if there is work to do**
  - **for non-preemptive schedulers, work-conserving is not always better**

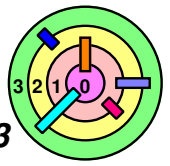


Definitions

- ***Preemptive scheduler:*** **scheduler can preempt the processor and give it to some other job**
  - **preemption can happen due to a *timer interrupt* or the *arrival of a higher priority job***
- ***Scheduling algorithm:***
  - **takes a workload as input**
  - **decides which jobs to do first**
  - **performance metric (throughput, latency) as output**
  - **only preemptive, work-conserving schedulers to be considered**

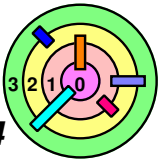


(7.1) Uniprocessor Scheduling



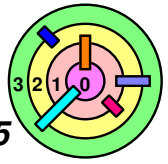
First-In-First-Out (FIFO)

- Also known as First-Come-First-Served (FCFS)
  - schedule jobs in the order they arrive and continue running them until they give up the processor voluntarily (i.e., job iscomplete)
  - minimize scheduling overhead
  - fair: every job waits its turn
- Weakness:
  - a long job can delay short jobs



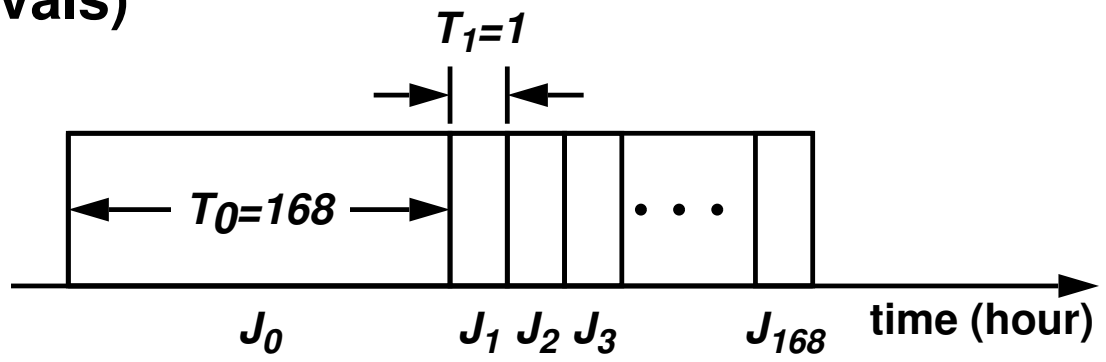
Shortest Job First (SJF)

- Always do the job that has the shortest remaining amount of work to do
  - this is a preemptive scheduler since a newly arrived job would preempt the current job if the new job's service time is less than the remaining service time of the current job
    - often called Shortest Remaining Time First (SRTF)
- Weakness:
  - unfair to long jobs, even starvation is possible if short jobs keep arriving
- Suppose we have five jobs arrive one right after another, but the first one is much longer than the others
  - which job completes first (and 2nd) in FIFO?
  - which job completes first (and 2nd) in SJF?

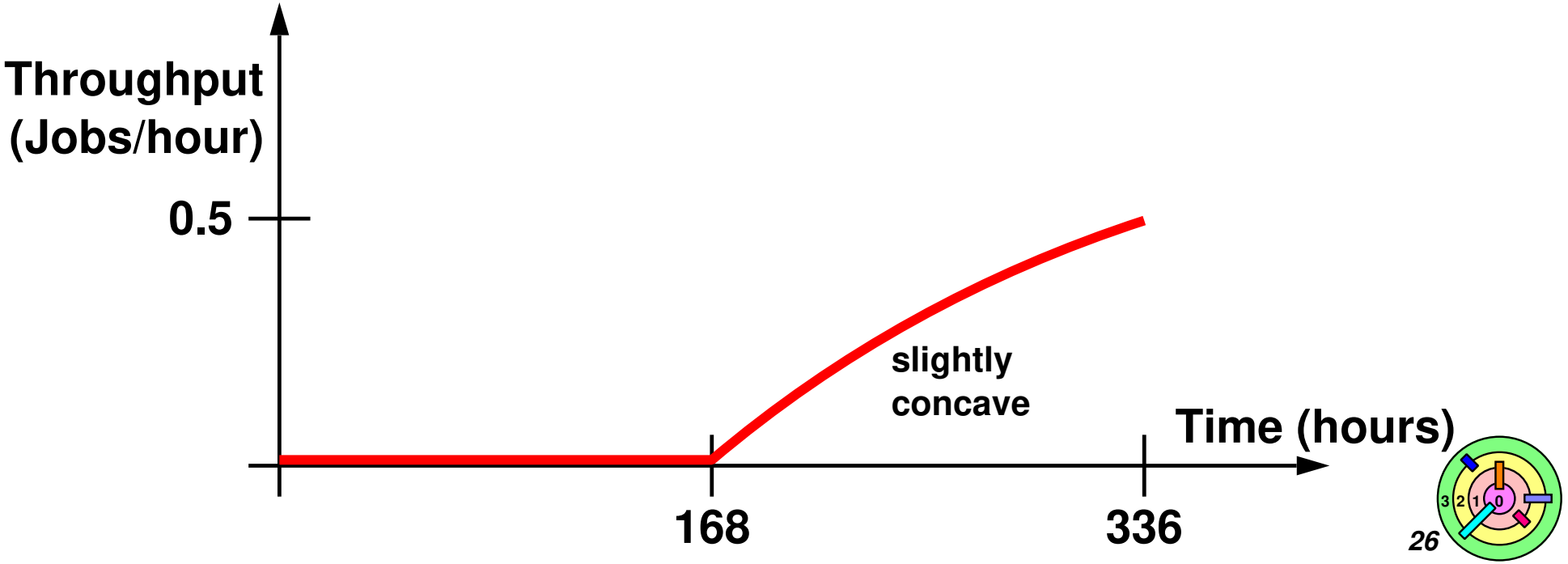


Comparing FIFO and SJF

- Workload: one *168*-hour job followed by *168* one-hour jobs (assuming no more arrivals)
- FIFO:

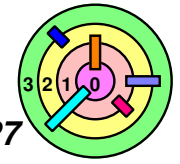


- Throughput: number of jobs completed / elapsed time



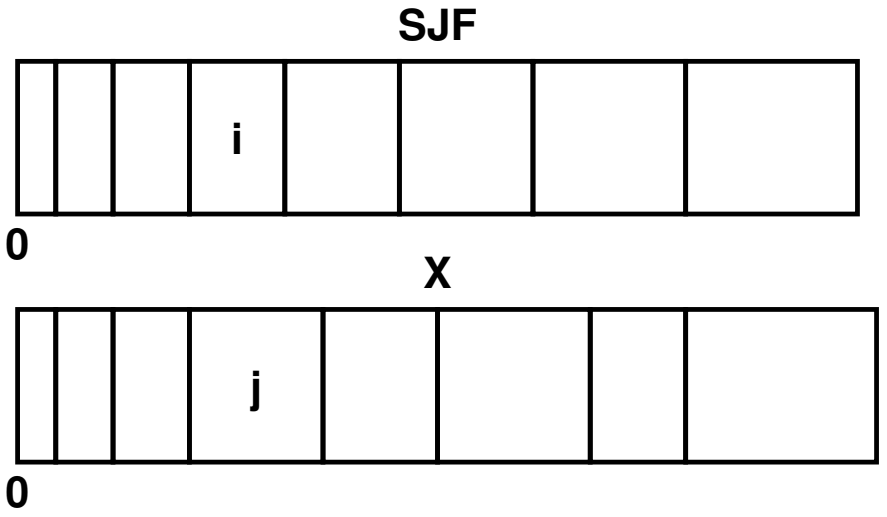
Another Performance Metric: Average Response Time

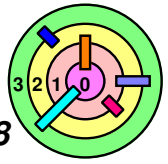
- Jobs J_i with *service time* T_i for $0 \le i < n$
- *Average Response Time (ART)* for FIFO (assuming no new arrivals)
  - J_i started at time t_i
  - $t_i = \sum_{j=0}^{i-1} T_j$ (for FIFO)
  - $RT_i = t_i + T_i$ (for non-preemptive schedulers)
  - $ART = \sum_{i=0}^{n-1} RT_i / n = \sum_{i=0}^{n-1} (t_i + T_i) / n$
- For our example (which is the *worst-case* for FIFO)
  - ART = *252* hours (with a standard deviation of *48.79* hours)
    - $Var(X) = E(X^2) - [E(X)]^2$, $StdDev(X) = sqrt(Var(X))$
    - large average and large variation (for this example)
- In general, ART for FIFO is more difficult to compute
  - need to look at all possible ordering of jobs and the probability of getting each particular order



SJF

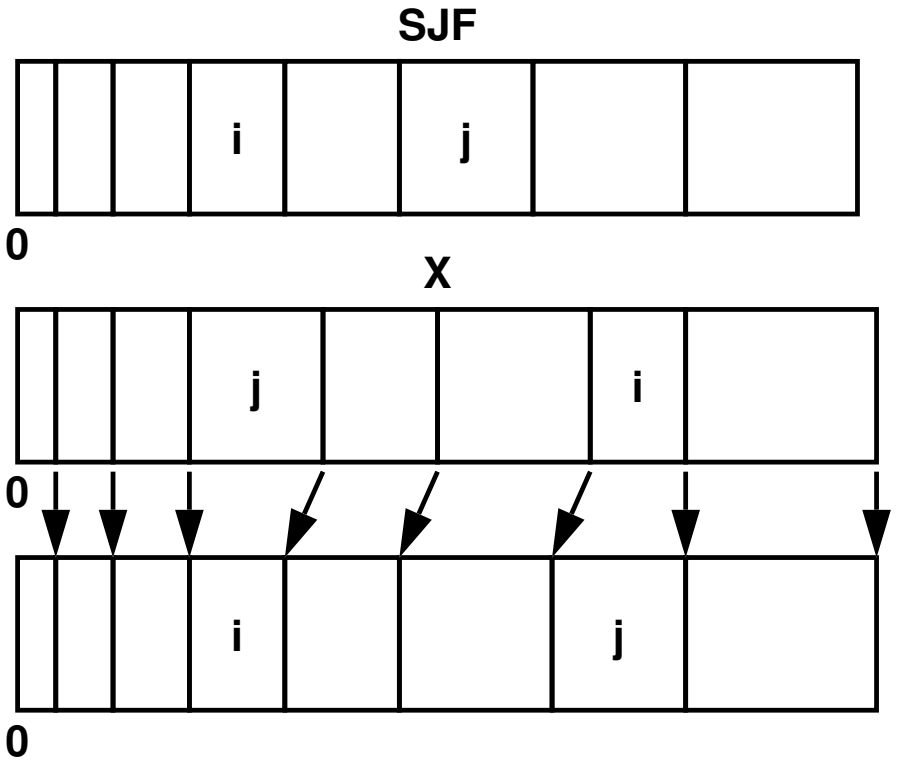
- SJF minimizes ART
  - *proof by contradiction* (CS 270):
    - assuming that scheduler X can create a schedule that has a smaller ART than SJF
    - this means that the schedule it creates is different from the SJF schedule
    - find the first scheduling decision where they do not agree, i.e., SJF runs job i while X runs job j and $T_j > T_i$

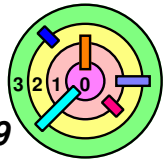




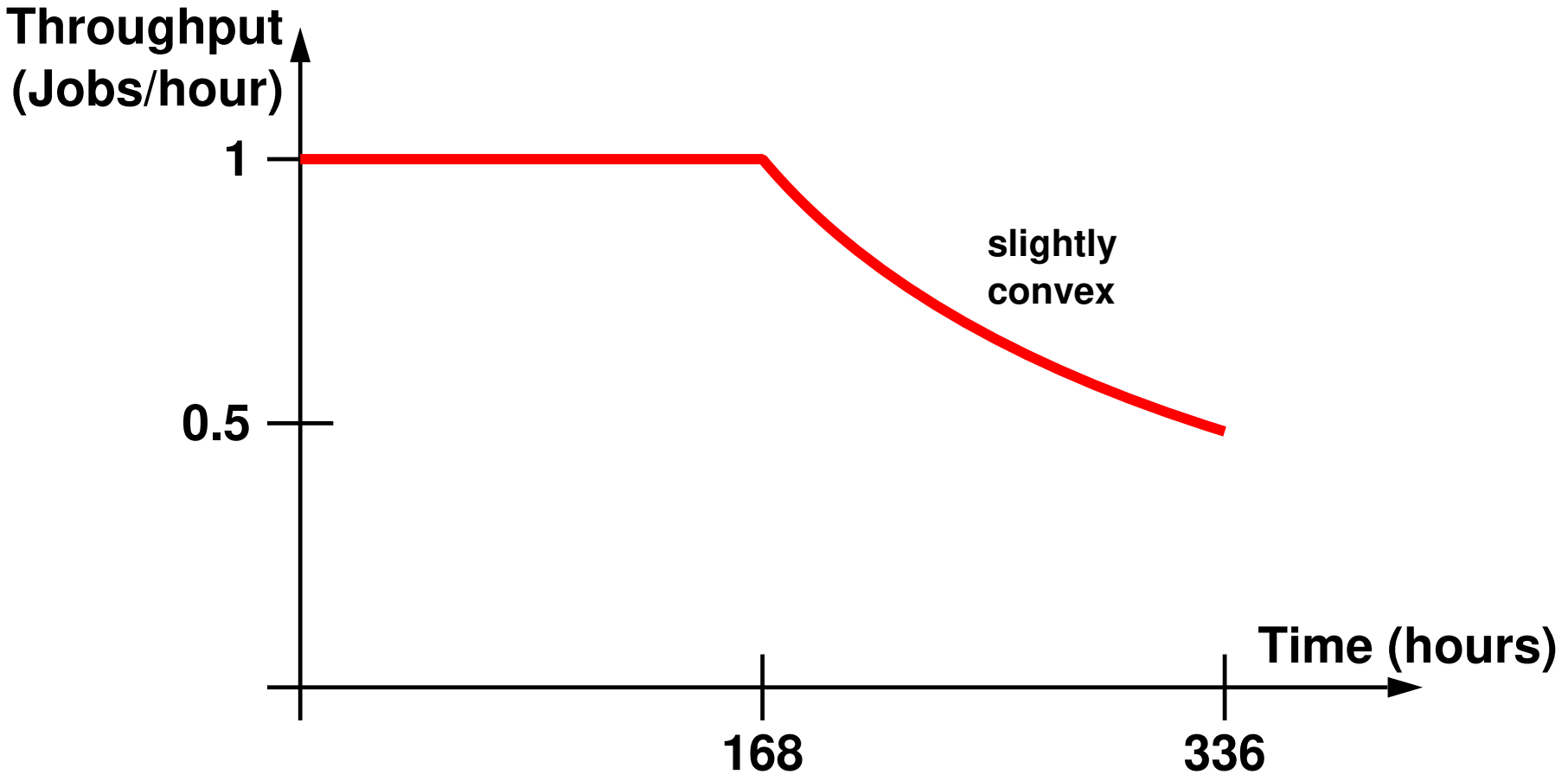
SJF

- SJF minimizes ART
  - *proof by contradiction* (CS 270):
    - assuming that scheduler X can create a schedule that has a smaller ART than SJF
    - this means that the schedule it creates is different from the SJF schedule
    - find the first scheduling decision where they do not agree, i.e., SJF runs job i while X runs job j and $T_j > T_i$
    - in the schedule created by X, find job i and swap jobs i and j and the resulting schedule has a smaller ART
      - contradiction! ¤





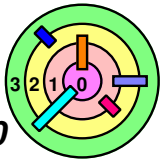
SJF



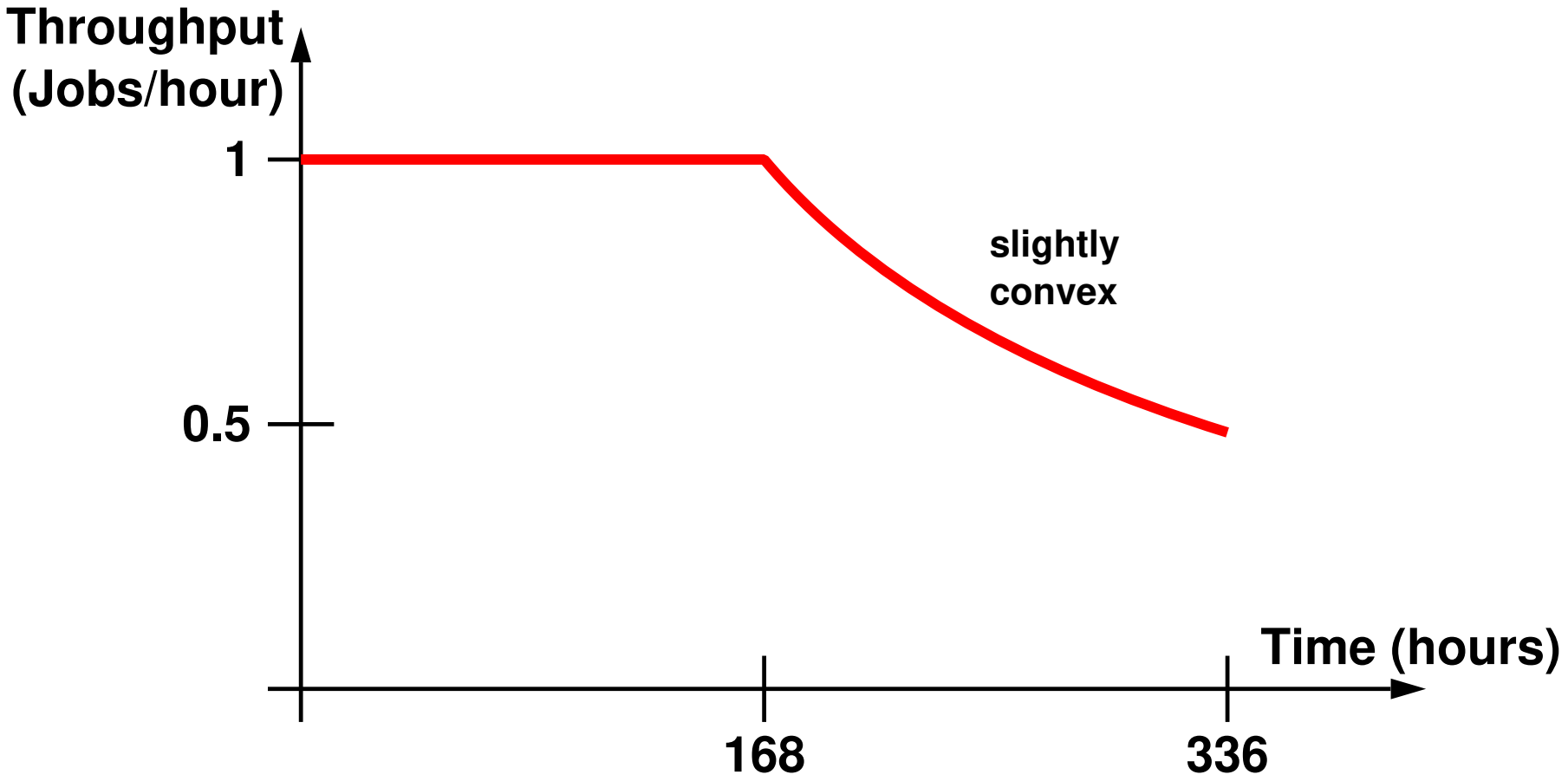
- ART = *85.99* hours (with a standard deviation of *52.06* hours)

⇨ Instantaneous throughputs can be different for different schedulers

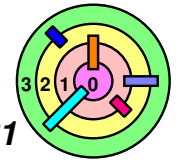
- but throughput at time = *336* is *identical for all work-conserving schedulers*



SJF

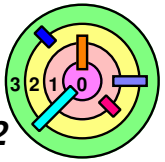


- What if short jobs keep arriving?
  - *starvation*
    - is this an unacceptable scheduling policy (too unfair)?
      - for lightly loaded web servers, may work nicely



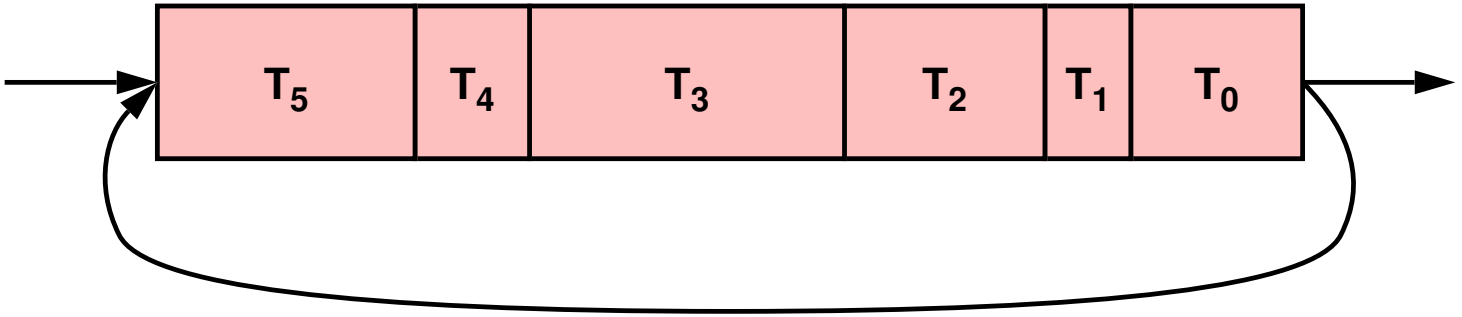
Fairness

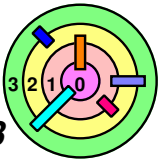
- FIFO
  - each job eventually gets processed
  - that seems fair
- SJF/SRTF
  - a long job might have to wait indefinitely
- What's a good measure of fairness?



Round Robin

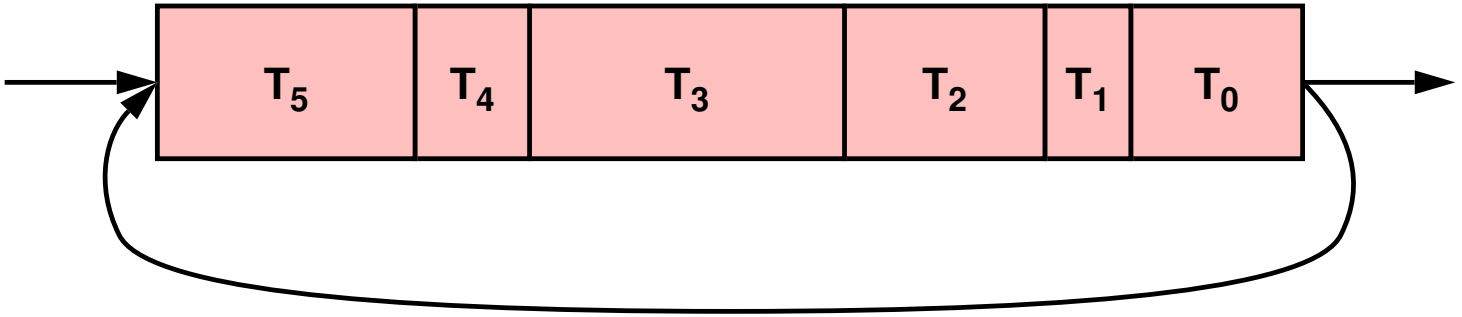
- Time-slicing
  - *q* = *time quantum* or *time slice*

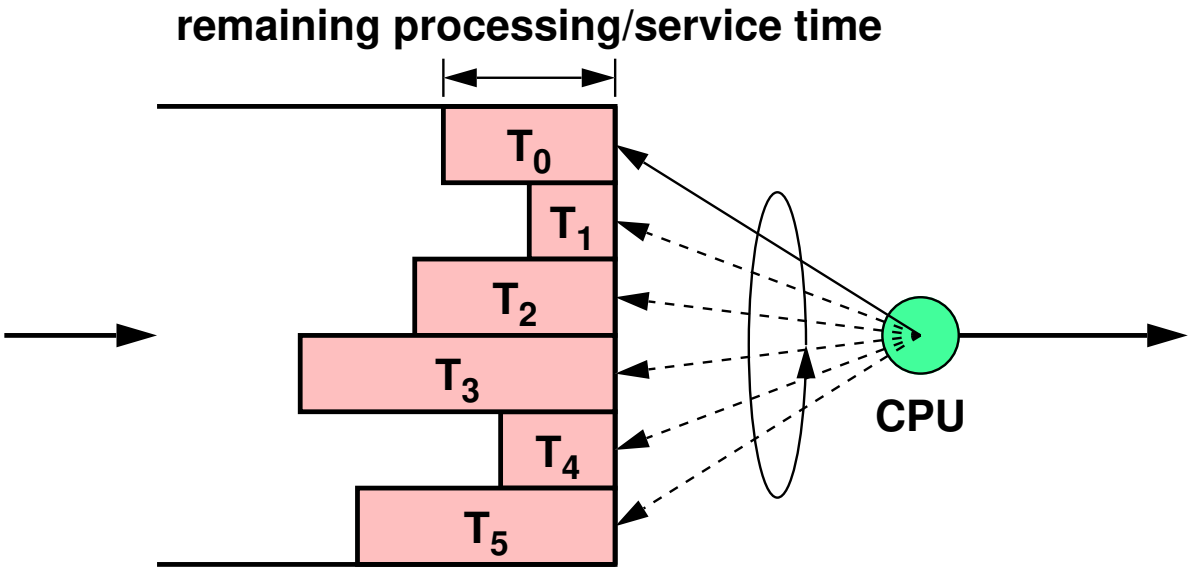


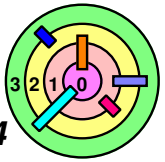


Round Robin

- Time-slicing
  - *q* = *time quantum* or *time slice*

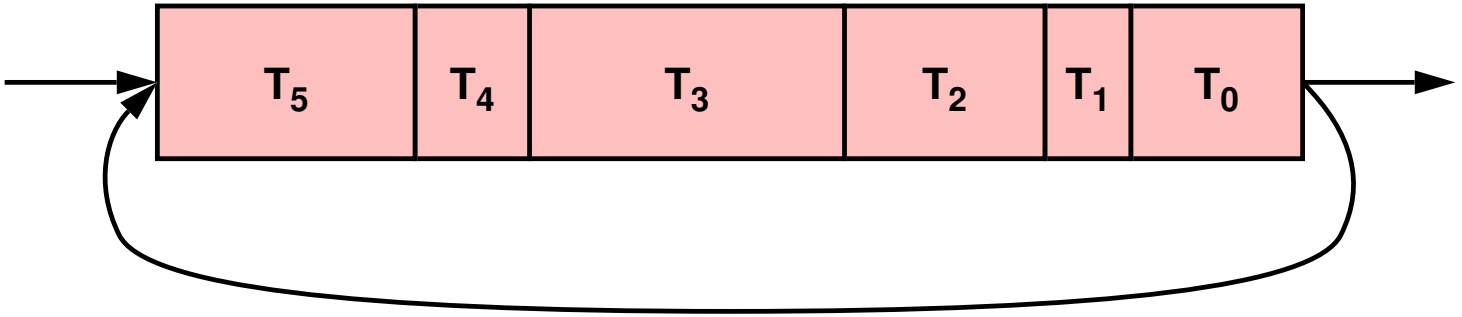




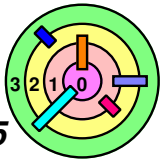


Round Robin

- Time-slicing
  - *q* = *time quantum* or *time slice*

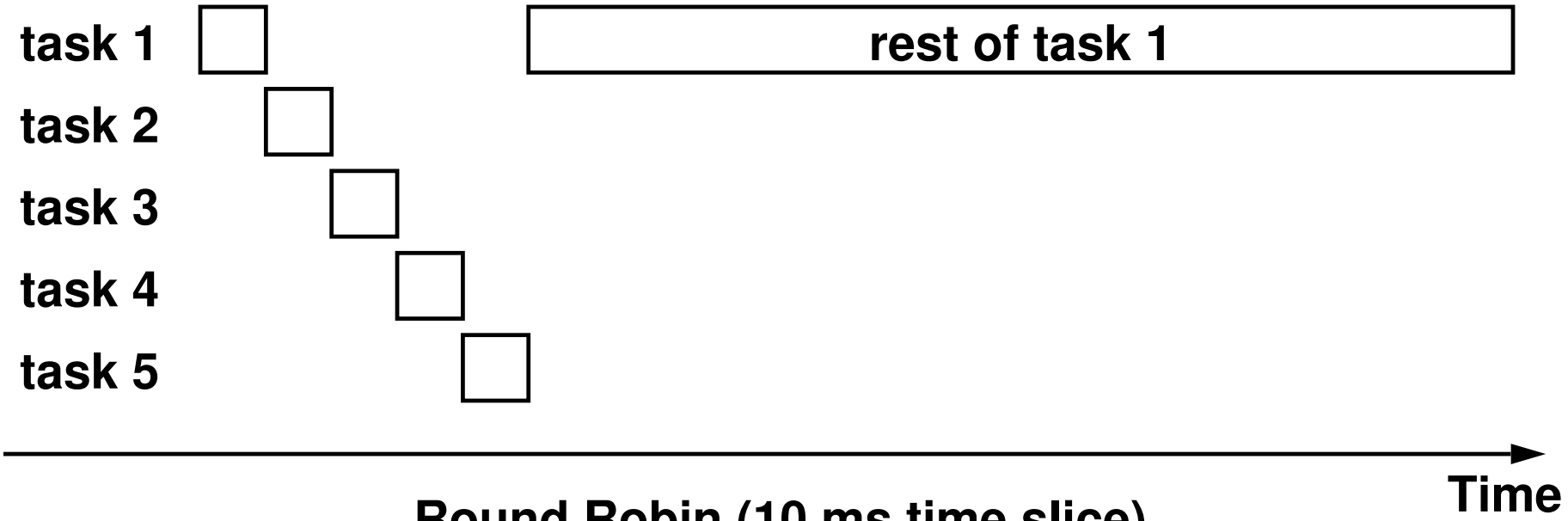


- Different values of *q*
  - *q* → *0:* processor-sharing (idealized case)
    - not realistic
    - poor cache performance
  - *q* too large: some jobs appear to be not making progress

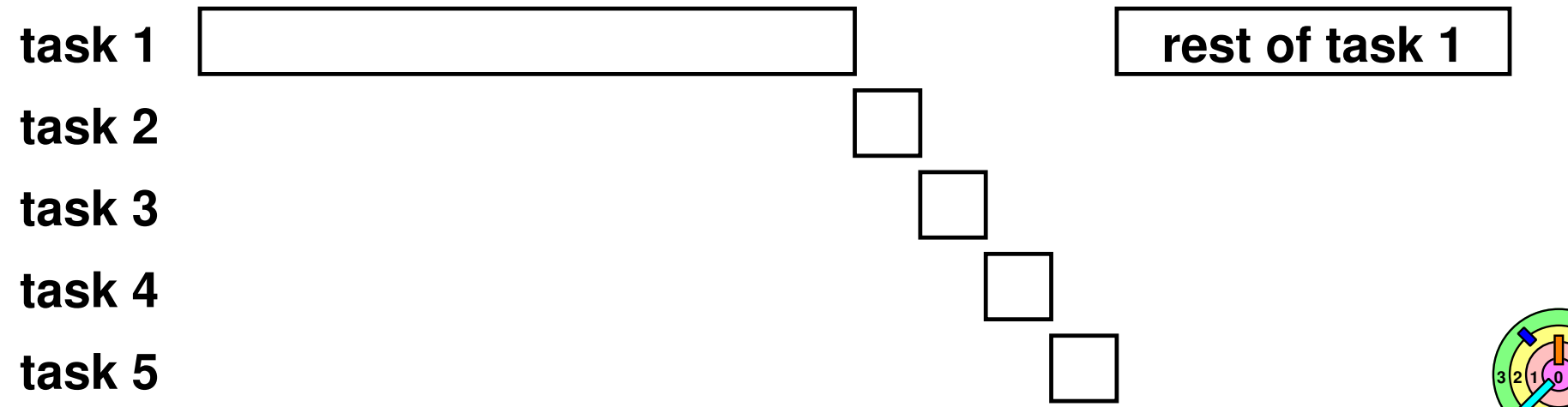


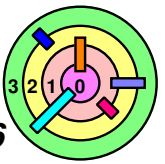
Time Quantum / Time Slice

Round Robin (1 ms time slice)



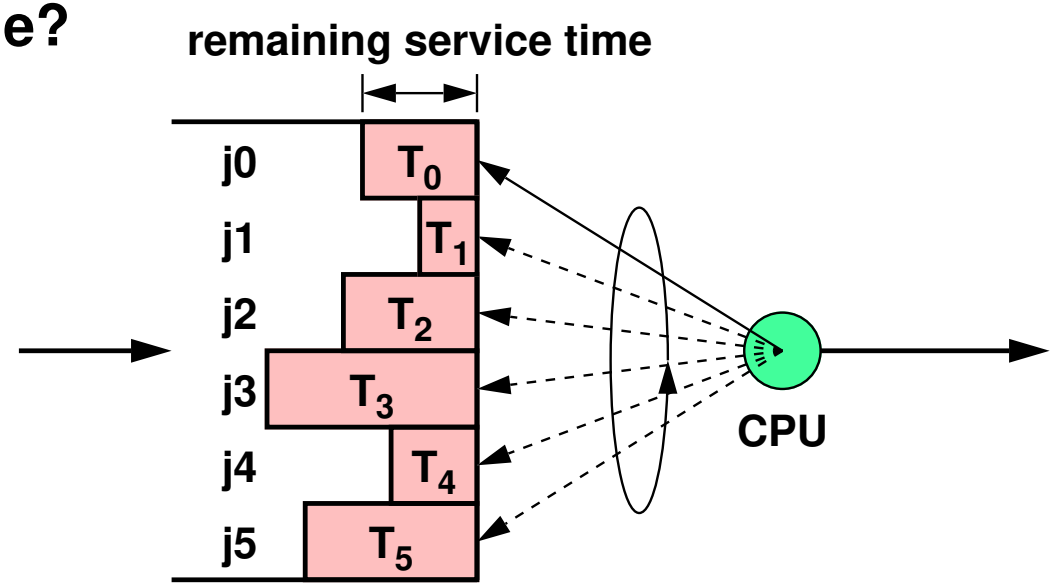
Round Robin (10 ms time slice)





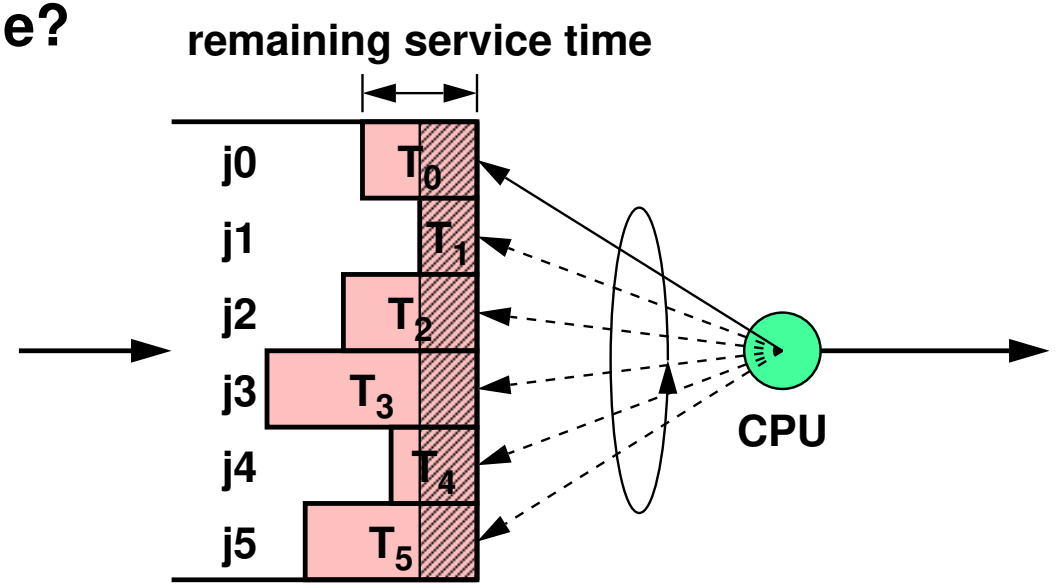
Round Robin

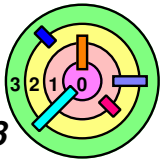
- How to calculate response time?
  - RT_1 =



Round Robin

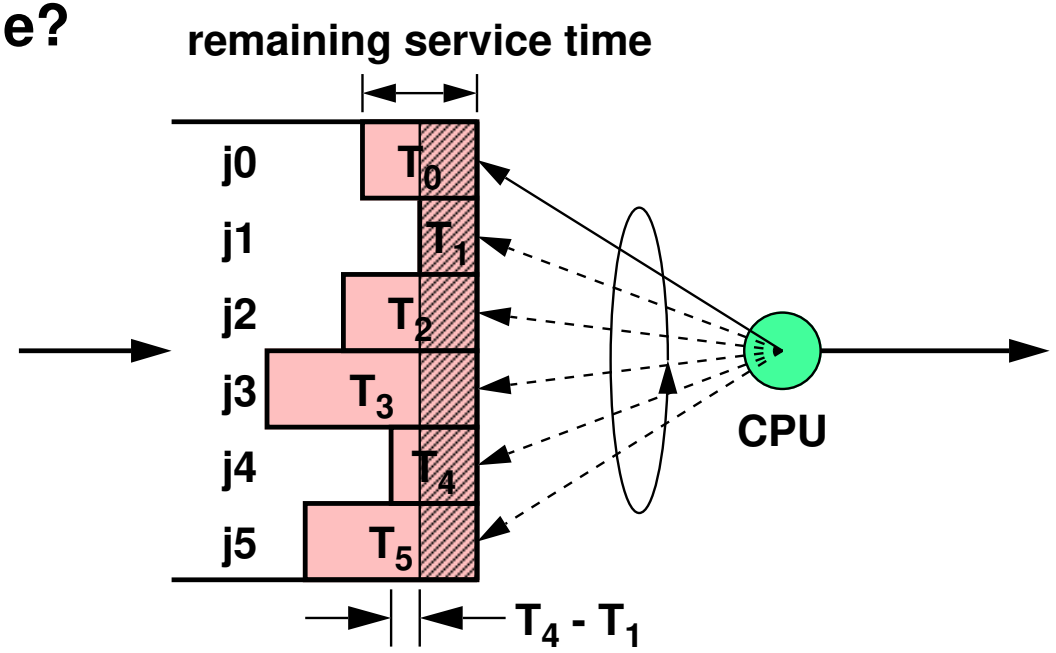
- How to calculate response time?
  - $RT_1 = 6 \times T_1$
    - why not $6 \times T_1 - 4 \times q$
    - $q \rightarrow 0$, we are doing calculus, not algebra

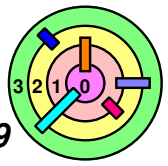




Round Robin

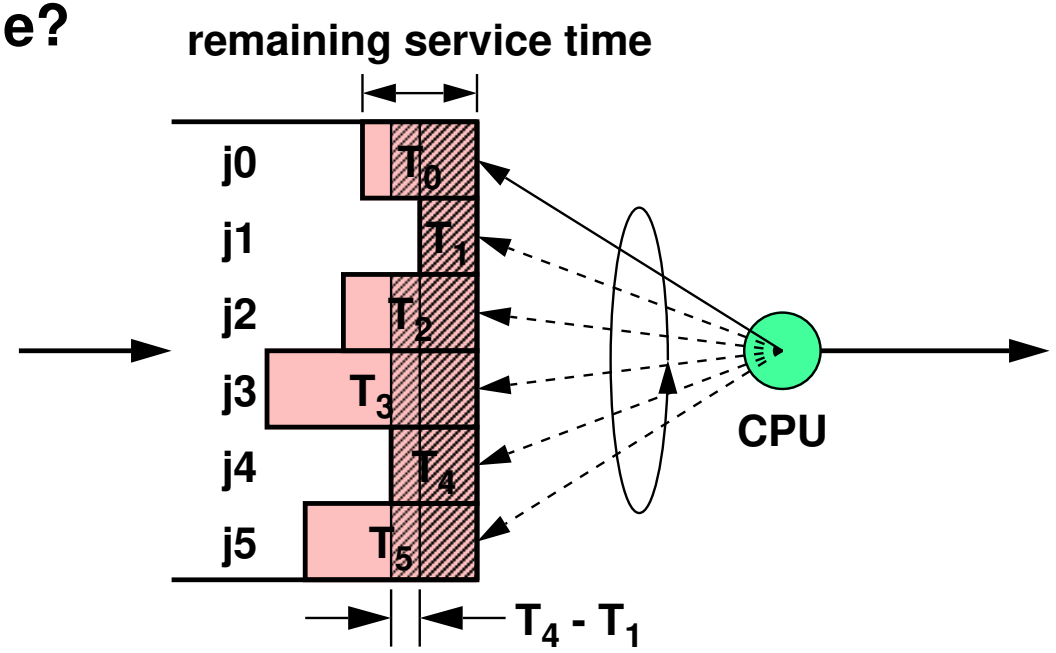
- How to calculate response time?
  - $RT_1 = 6 \times T_1$
  - $RT_4 = ?$

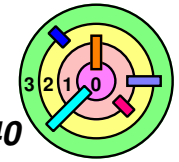




Round Robin

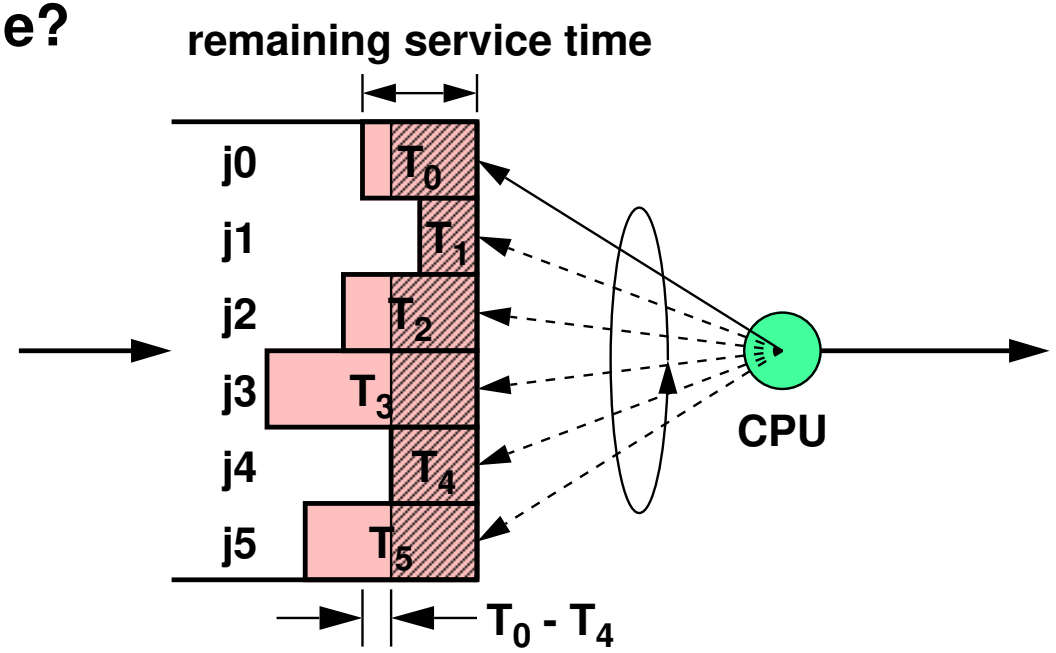
- How to calculate response time?
  - $RT_1 = 6 \times T_1$
  - $RT_4 = 5 \times (T_4 - T_1) + RT_1$

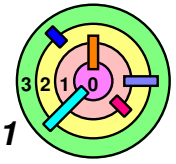




Round Robin

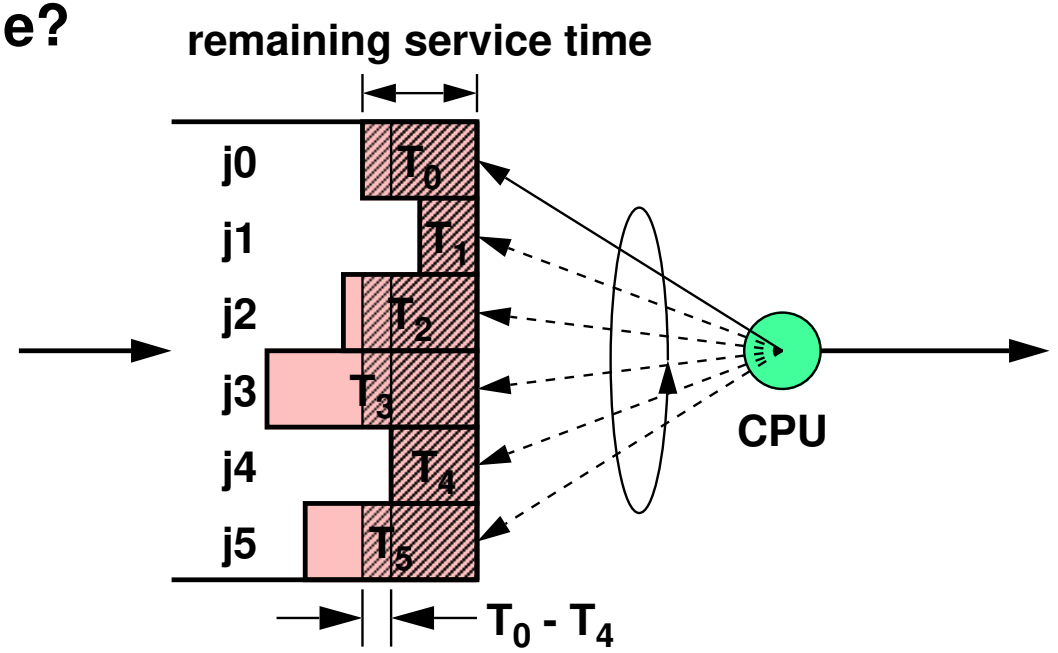
- How to calculate response time?
  - $RT_1 = 6 \times T_1$
  - $RT_4 = 5 \times (T_4 - T_1) + RT_1$
  - $RT_0 = ?$

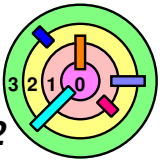




Round Robin

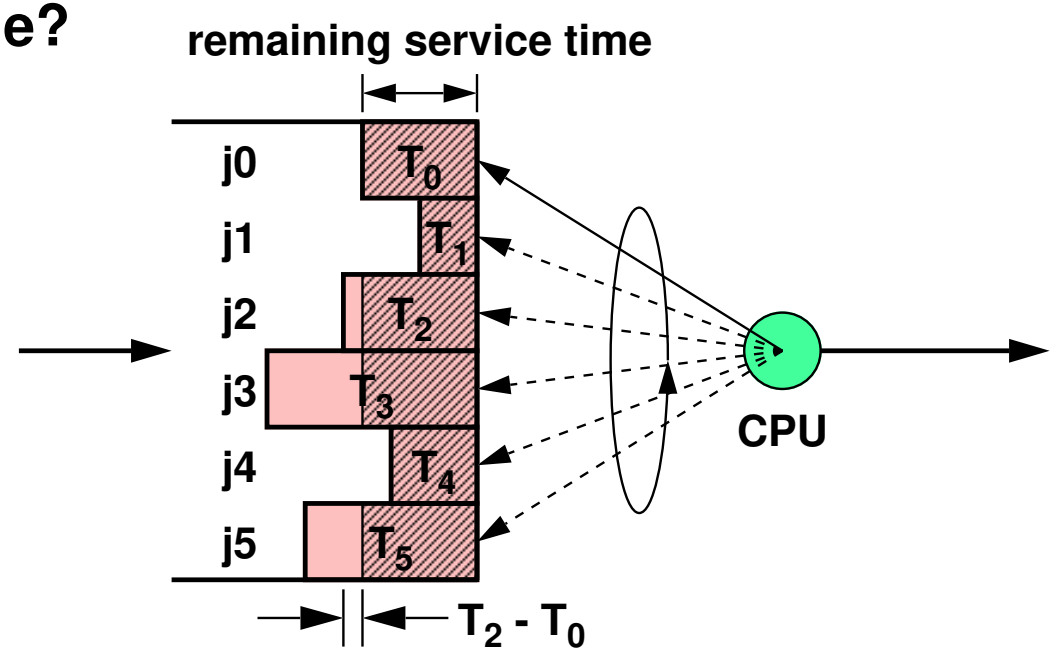
- How to calculate response time?
  - $RT_1 = 6 \times T_1$
  - $RT_4 = 5 \times (T_4 - T_1) + RT_1$
  - $RT_0 = 4 \times (T_0 - T_4) + RT_4$

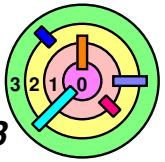




Round Robin

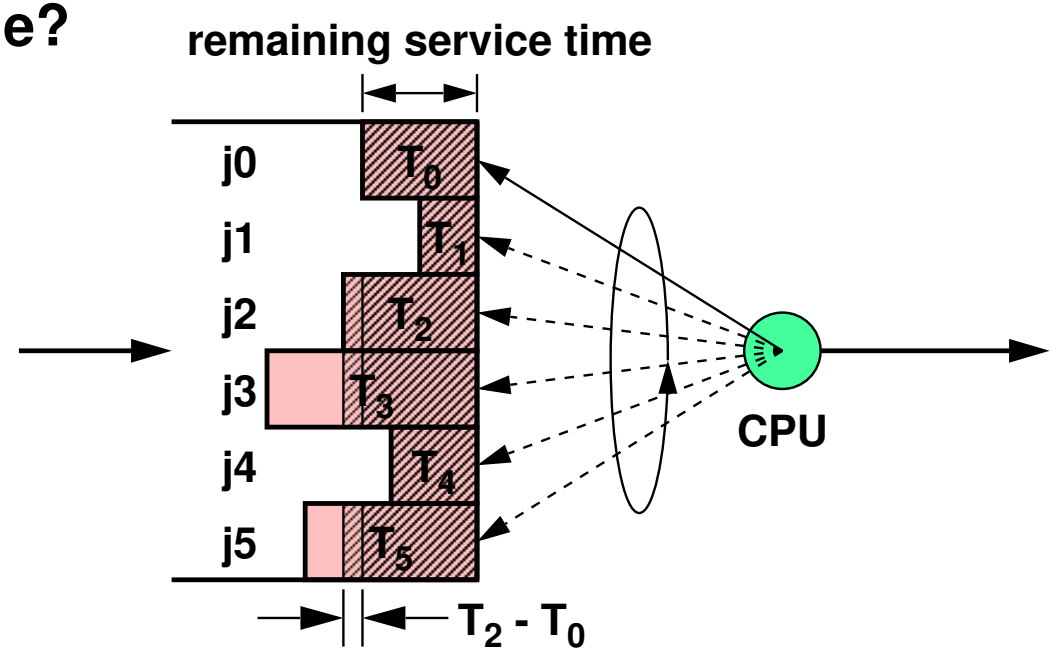
- How to calculate response time?
  - $RT_1 = 6 \times T_1$
  - $RT_4 = 5 \times (T_4 - T_1) + RT_1$
  - $RT_0 = 4 \times (T_0 - T_4) + RT_4$
  - $RT_2 = ?$

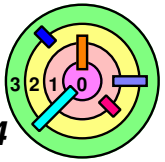




Round Robin

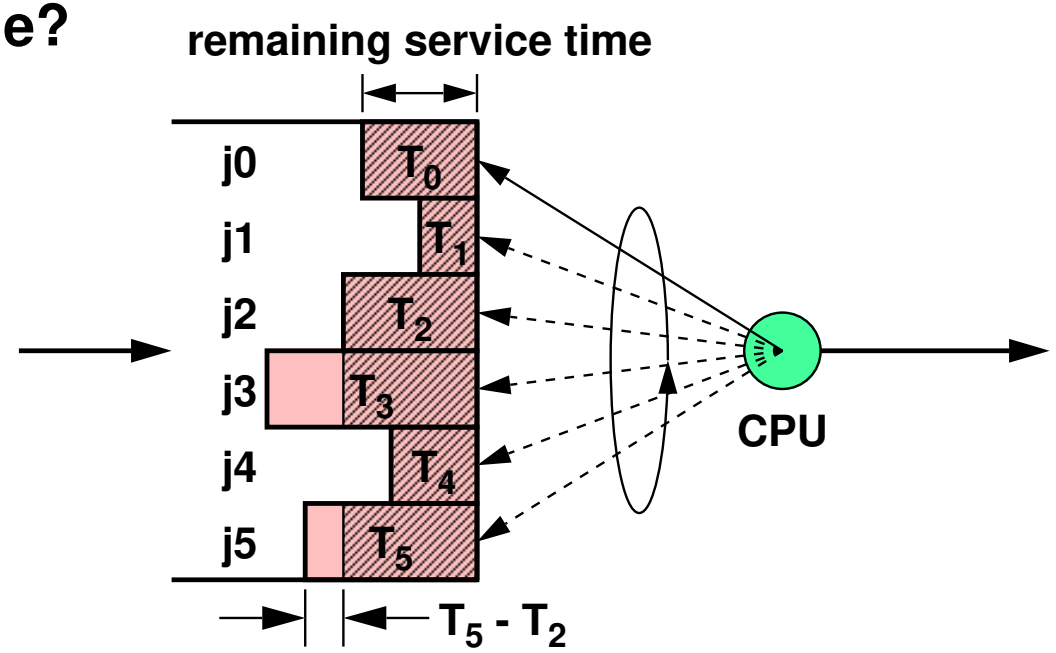
- How to calculate response time?
  - $RT_1 = 6 \times T_1$
  - $RT_4 = 5 \times (T_4 - T_1) + RT_1$
  - $RT_0 = 4 \times (T_0 - T_4) + RT_4$
  - $RT_2 = 3 \times (T_2 - T_0) + RT_0$

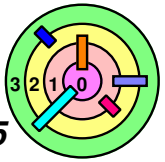




Round Robin

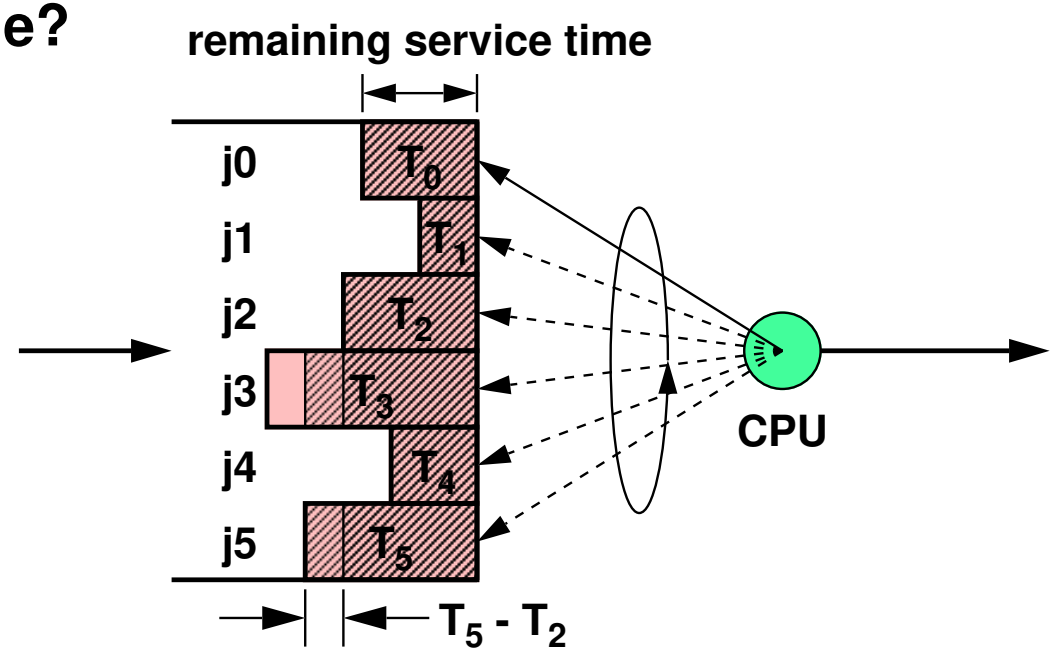
- How to calculate response time?
  - $RT_1 = 6 \times T_1$
  - $RT_4 = 5 \times (T_4 - T_1) + RT_1$
  - $RT_0 = 4 \times (T_0 - T_4) + RT_4$
  - $RT_2 = 3 \times (T_2 - T_0) + RT_0$
  - $RT_5 = ?$

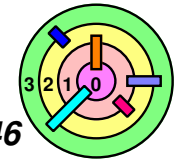




Round Robin

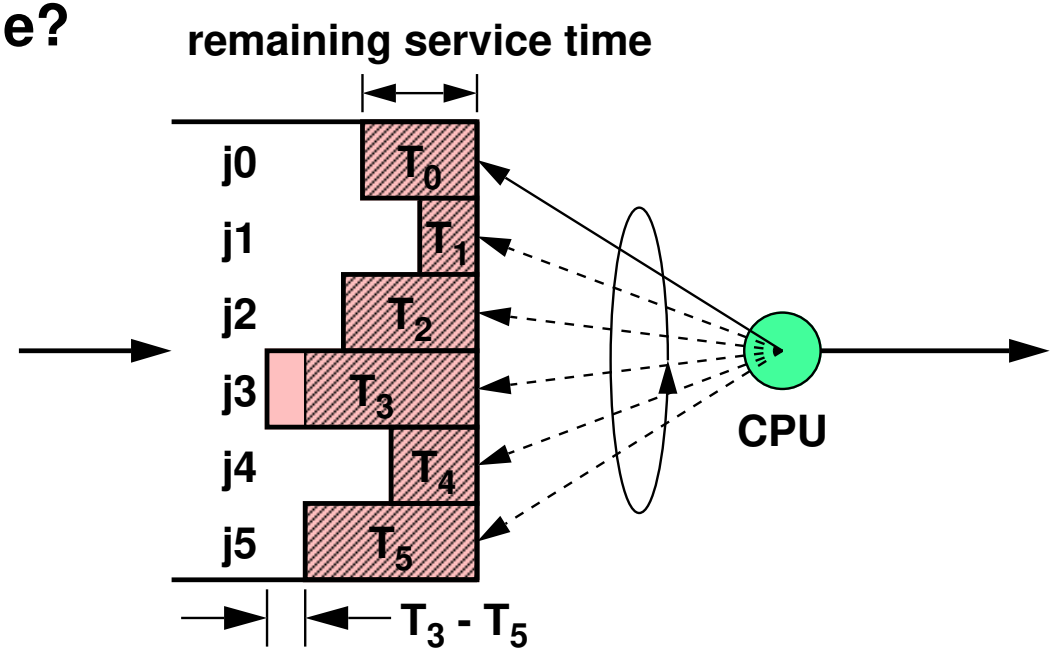
- How to calculate response time?
  - $RT_1 = 6 \times T_1$
  - $RT_4 = 5 \times (T_4 - T_1) + RT_1$
  - $RT_0 = 4 \times (T_0 - T_4) + RT_4$
  - $RT_2 = 3 \times (T_2 - T_0) + RT_0$
  - $RT_5 = 2 \times (T_5 - T_2) + RT_2$

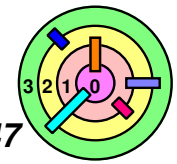




Round Robin

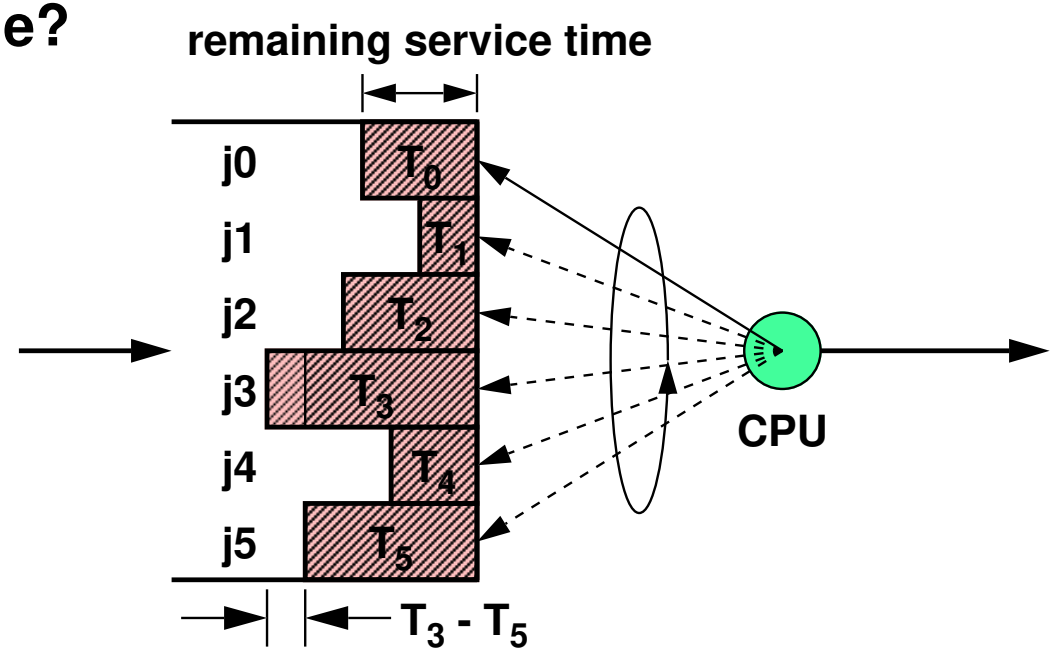
- How to calculate response time?
  - $RT_1 = 6 \times T_1$
  - $RT_4 = 5 \times (T_4 - T_1) + RT_1$
  - $RT_0 = 4 \times (T_0 - T_4) + RT_4$
  - $RT_2 = 3 \times (T_2 - T_0) + RT_0$
  - $RT_5 = 2 \times (T_5 - T_2) + RT_2$
  - $RT_3 = ?$



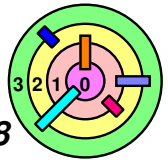


Round Robin

- How to calculate response time?
  - $RT_1 = 6 \times T_1$
  - $RT_4 = 5 \times (T_4 - T_1) + RT_1$
  - $RT_0 = 4 \times (T_0 - T_4) + RT_4$
  - $RT_2 = 3 \times (T_2 - T_0) + RT_0$
  - $RT_5 = 2 \times (T_5 - T_2) + RT_2$
  - $RT_3 = 1 \times (T_3 - T_5) + RT_5$

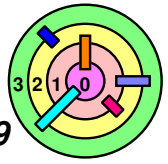


- Does it check out?
  - $RT_3 = T_0 + T_1 + T_2 + T_3 + T_4 + T_5$



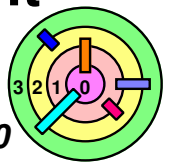
Round Robin

- ART?
  - let quantum approach *0* (and pretend that it's realistic)
    - if quantum is too small, scheduling overhead will be too large
  - *169* jobs sharing the processor
  - run at *1/169th* speed for first week
  - short jobs receive one hour of processor time in *169* hours
    - all short jobs finish at about the same time
  - long job completes in *336* hours
  - ART = *169.99* hours
    - recall that ART is *252* hours for FIFO in our example and *85.99* hours for SJF
  - *average deviation = 12.81 hours*
    - recall that average deviation = *48.79* hours for FIFO in our example and *52.06* hours for SJF



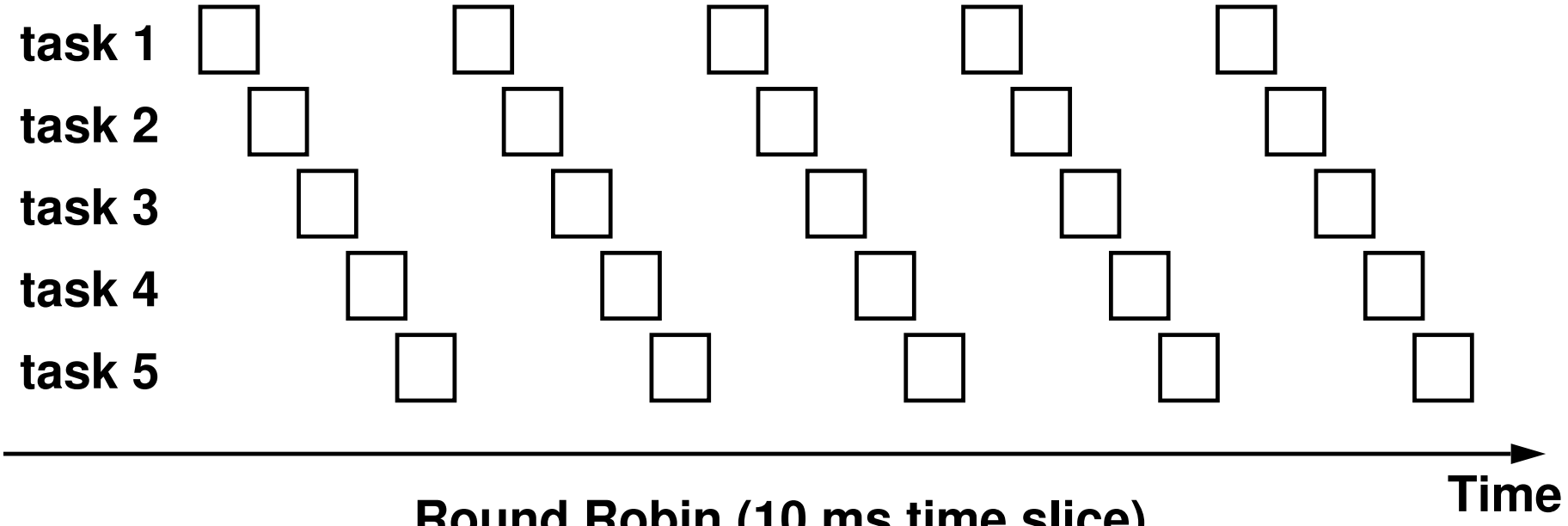
Round Robin

- Pro:
  - appears to be "fair" (no starvation)
- Con:
  - overhead for time slicing can be high
    - timer interrupt can be serviced in tens of microseconds
    - reloading caches may take longer, depending on the cache hierarchy of the system
    - it's typical to set time slice interval to be between 10 to 100 milliseconds
  - if all tasks start at the same time and they all have the same service time, RR has the worst ART for this workload
    - time slicing added overhead without any benefit!
    - for this workload, FIFO and SJF have optimal ART
    - although for streaming video, it's important to achieve a predictable and stable rate of progress and RR is just right for that (and time to complete download is unimportant)

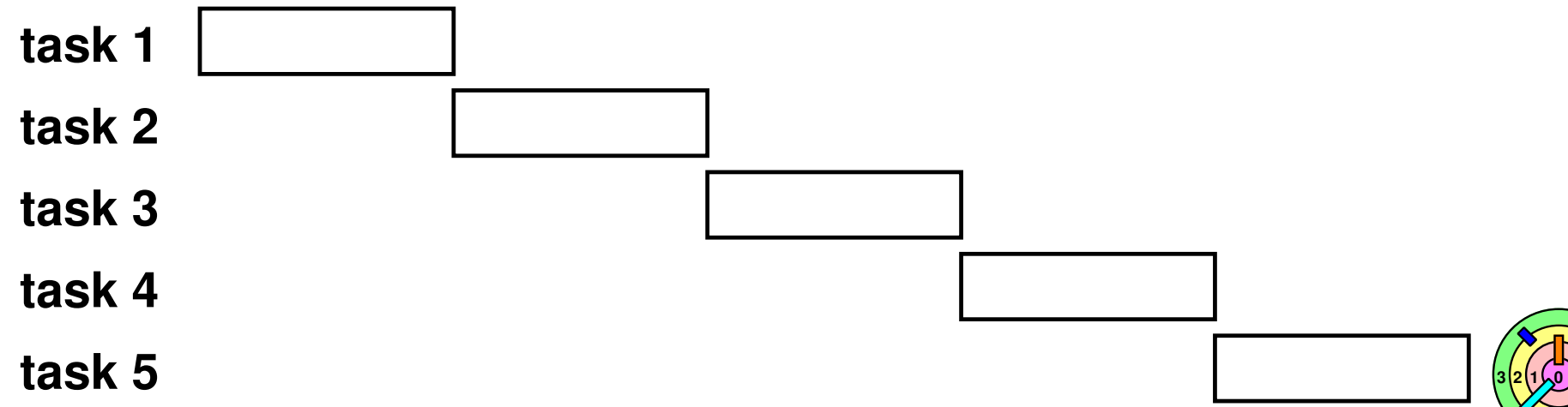


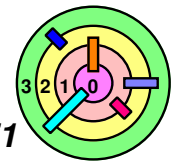
Round Robin

Round Robin (1 ms time slice)



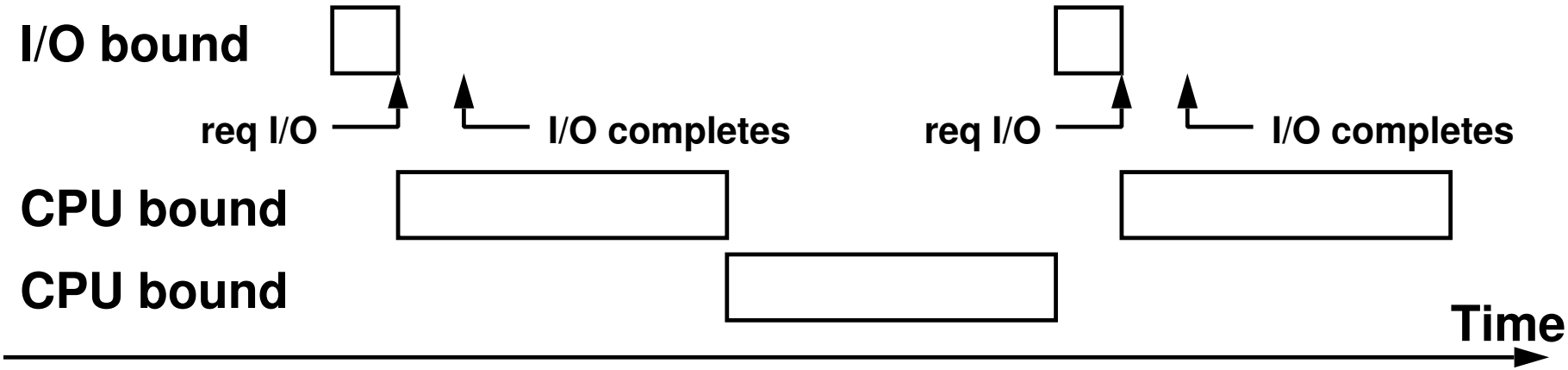
Round Robin (10 ms time slice)





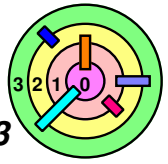
Round Robin

- Con:
  - if the workload is a mixture of I/O bound and compute bound tasks, RR may perform poorly
    - e.g., text editor is I/O bound
      - takes a few milliseconds to echo a keystroke to the screen (faster than human perception)
      - if there are lots of compute bound tasks competing for the processor and each of them take a 100ms full time slice, the text editor can appear quite sluggish to the user



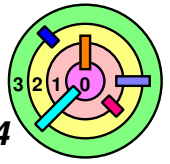
Max-Min Fairness

- How do we balance a mixture of tasks?
  - some I/O bound, need only a little CPU, some compute bound and can use as much CPU as they are assigned
- Ex: if the government has $1,000,000 to give to 1,000 citizens, what's the fair way to distribute the money?
  - is it fair to give everyone $1,000?
    - no, if someone needs less than $1,000, you are not allowed to waste resource
- Please note that ***fairness*** is difficult to define
  - RR seems fair, but the mixed worload of I/O bound and compute bound tasks shows that I/O bound tasks are treated "unfairly"
- If we want to talk about fairness, we have to first define what fairness means (or what kind of fairness are we trying to achieve)



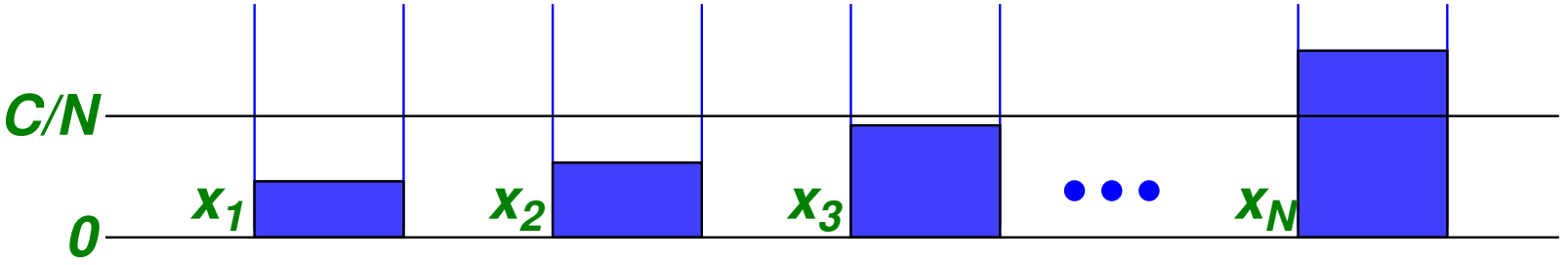
Max-Min Fairness

- ***Max-Min Fairness:*** **a fair service maximizes the allocation of the requester requesting the smallest amount of resource**
  - **no requester receives more than its request**
  - **if any requester needs less than an equal share, allocate to satisfy the smallest request first *(i.e., maximize the minimum allocation)***
    - **split the remaining resource using max-min recursively**
  - **if all remaining tasks need at least equal share, split evenly**
- **What's a resource?**
  - **anything finite that's shared**
    - **processing time**
    - **memory**
    - **network bandwidth**
    - **user, application, thread?!**
    - **etc.**
  - **this is broadly refer to as *resource allocation problems***

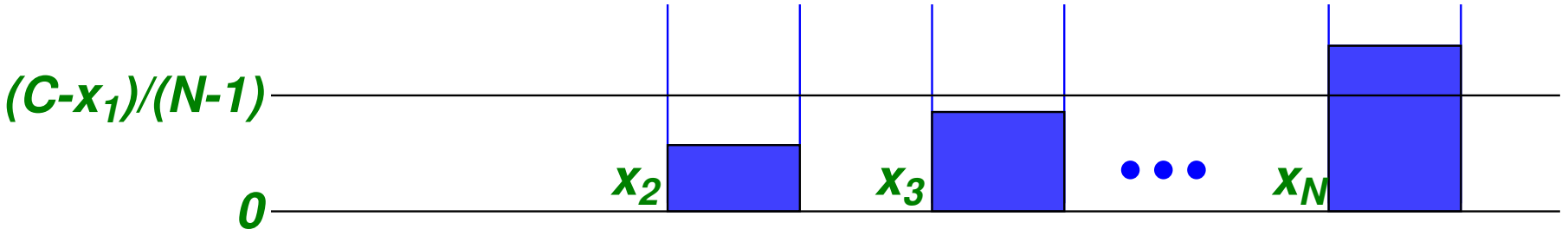


Max-Min Fairness Example

- Total capacity *C* divided among *N* jobs
  - x_i is the request of job *i*
  - sort jobs based on x_i
  - initially, assign *C/N* to each job

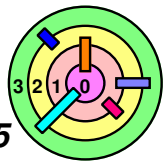


  - satisfy x_1, redistribute remaining capacity evenly



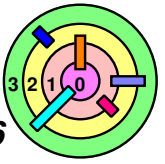
  - recursion

- This is basically *"processor sharing" (i.e., RR with $q \rightarrow 0$)*
  - although fair, but has performance issues



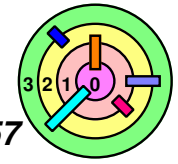
Scheduling for Interactive Systems

- In general, wants to treat interactive tasks differently from background tasks
  - don't want an interactive user feel that the system is sluggish
  - need some kind of *priority-based scheduler*
- How do you know if a task is interactive or not?
  - an application can be interactive sometimes (e.g., gathering information from the user) and non-interactive some other time
- Maybe we can use a task's behavior at the scheduler and give preferential treatment to I/O bound tasks (i.e., those who give up the processor quickly)
  - give it a time slice and see if it yield the processor voluntarily
    - if not, lower its priority



Multi-Level Feedback Queues (MFQ)

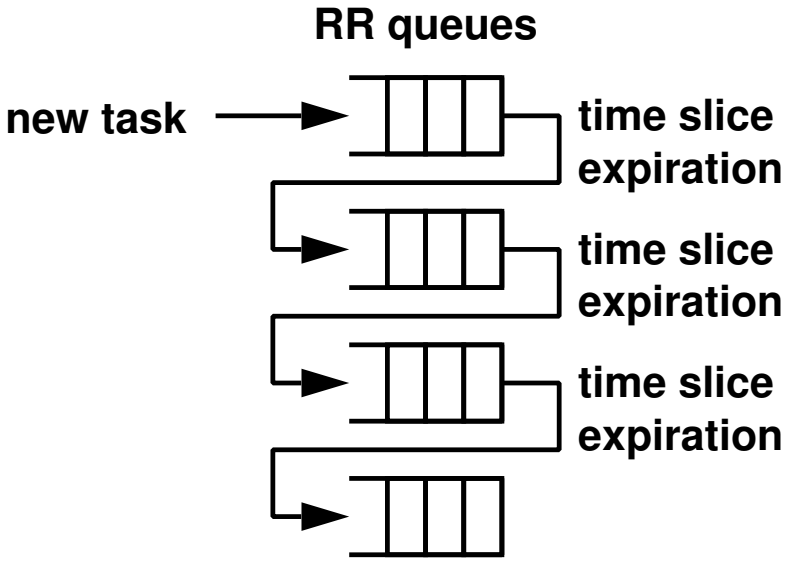
- Goals:
  - *responsiveness:* run short tasks quickly, as in SJF
  - *low overhead:* minimize the number of preemptions, as in FIFO, and minimize the time spent making scheduling decisions
  - *starvation freedom:* all tasks should make progress, as in RR
  - *background tasks:* low priority tasks should not interfere with user work
  - *fairness:* assign non-background processeses approximately their max-min fair share of the processor
- MFQ is not perfect at any of them, but a reasonable compromise in most real-world cases
  - used in Linux, Windows, and Mac OS X

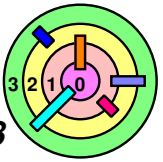


Multi-Level Feedback Queues (MFQ)

- MFQ is an extension of RR
  - instead of only a single queue, MFQ has multiple RR queues, each with a different priority level and time quantum
  - task at a higher priority level preempt lower priority tasks
  - tasks at the same level are scheduled in RR fashion
  - higher priority levels have shorter time quanta than lower levels
  - a new task enters at the top priority level
  - every time the task uses up its time quantum, it drops a level

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

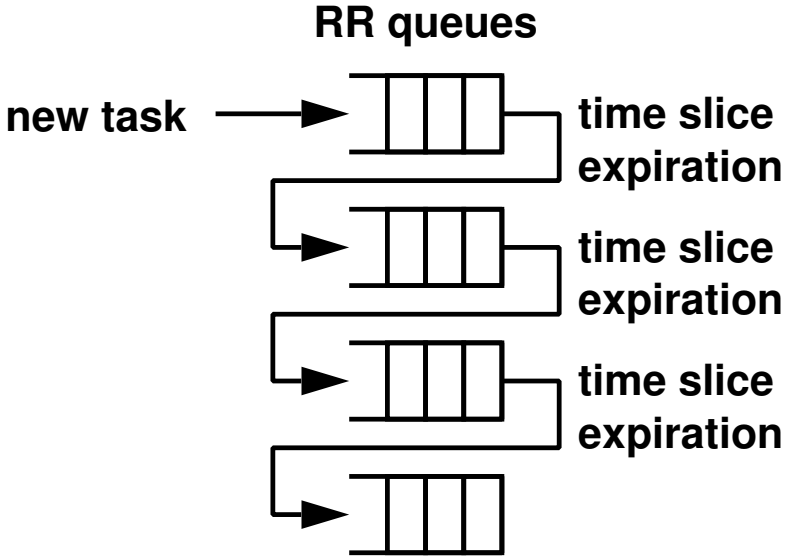


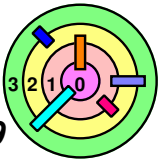


Multi-Level Feedback Queues (MFQ)

- Starvation
  - if I/O bound tasks keep arriving, a compute bound task can starve
  - aging: if a thread hasn't run for a while, increase its priority
    - different OS does this differently

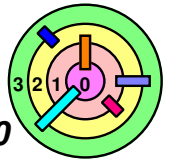
Priority	Time Slice (ms)
1 (highest)	10
2	20
3	40
4	80



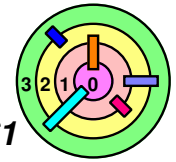


Uniprocessor Scheduling: Summary

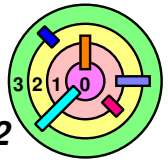
- FIFO is simple and minimizes overhead
- If tasks are variable in size, then FIFO can have very poor ART
- If tasks are equal in size, FIFO is optimal in terms of ART
- Considering only the processor, SJF is optimal in terms of ART
- SJF is pessimal in terms of variance in response time
- If tasks are variable in size, Round Robin approximates SJF
- If tasks are equal in size, Round Robin will have very poor ART
- Tasks that intermix processor and I/O benefit from SJF and can do poorly under Round Robin
- 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



(7.2) Multiprocessor Scheduling

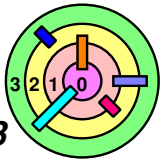


(7.3) Energey-Aware Scheduling



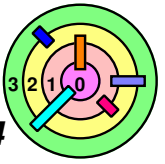
Battery

- Battery is an unusual resource
  - when a thread uses up battery, the resource is gone and is unavailable for all activities in the system
  - battery is a resource that cannot be virtualized (unlike memory or processor)



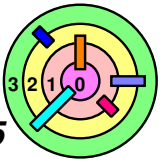
Energy vs. Performance Tradeoffs

- Modern hardware systems can trade performance for power consumption (i.e. energy)
  - increase performance (rate of instruction execution) by consuming more power
  - heterogeneous cores (some high performance high power cores and some low performance low power cores)
  - powering on or off cores and I/O devices



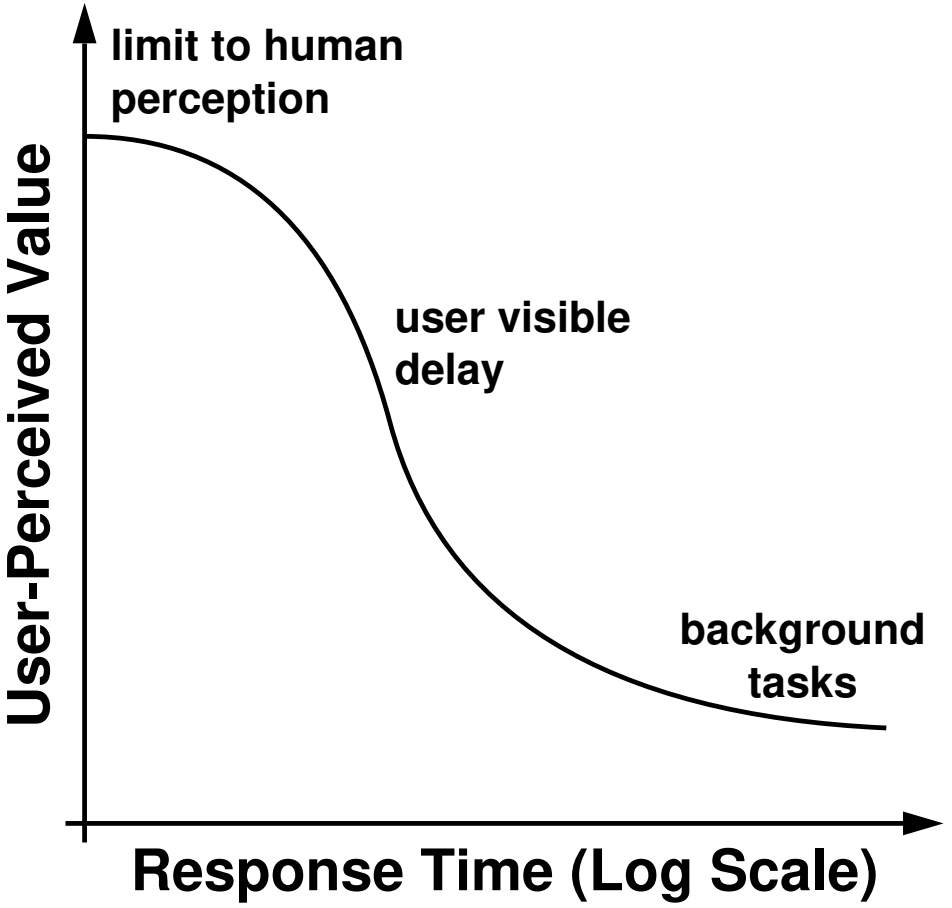
Energy Policies & Scheduling

- On battery powered devices (laptops and phones) user's can often select an energy policy
  - lower performance and greater battery life
  - better performance and lower battery life
  - or a blend
- To achieve this blend the scheduler needs to be involved
  - should I schedule this thread on the high performance, high power core?
  - would allowing threads from this program to get all the resources for a few time slices allow some I/O device to be powered down temporarily?



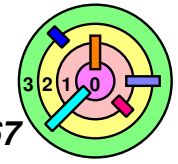
Relationship Between Response Time And User-Perceived Value

- The longer something takes, the less useful it is to the user
- Human perception is unable to tell the difference between a few tens of milliseconds
  - adding a short delay (and save some energy) will not matter much for most tasks
- Increased energy use often provides dinimishing returns in terms of the perception of improved performance
  - e.g., quickly updating the display after a user interface command is probably more important than transferring files quickly in the background

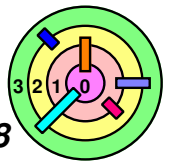


Basic Approach

- If low performance is below human perception:
  - then optimize for energy user, i.e., lower performance and save energy
- If low performance is above human perception:
  - then optimize for response time (so the user doesn't notice any slowdown)
- Long running and background tasks
  - balance energy use and responsiveness depending on the available battery resources (i.e. battery vs. plugged in)

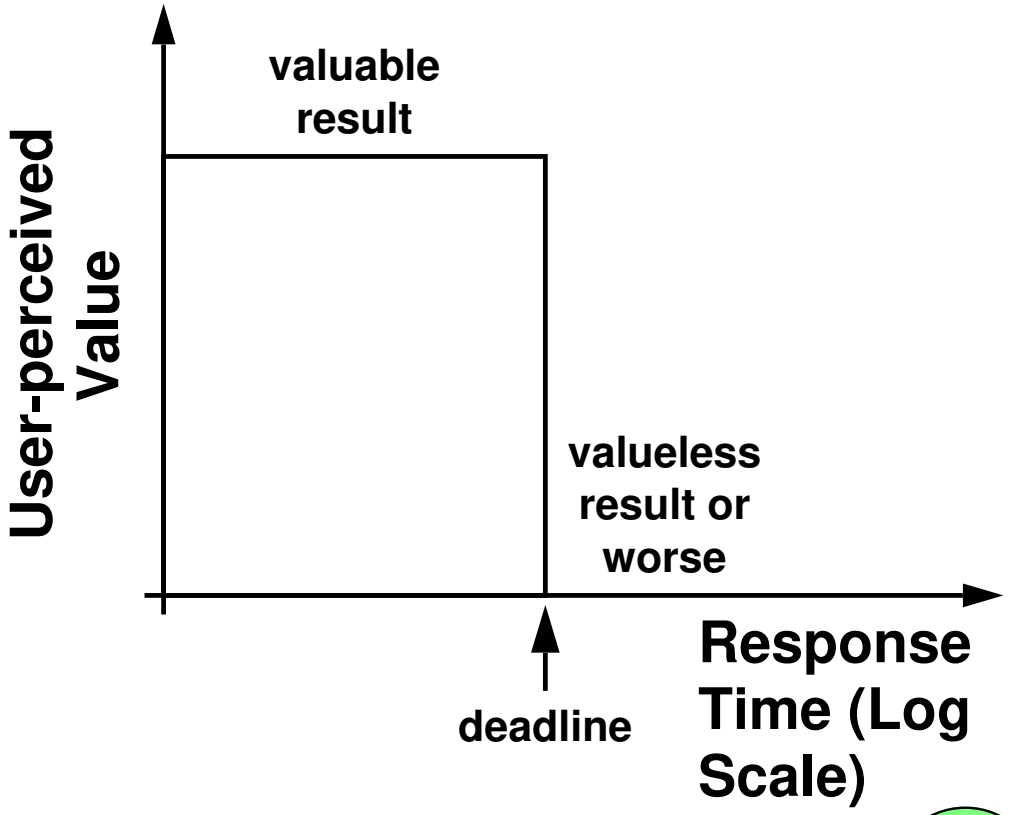


(7.4) Real-Time Scheduling



Real-Time Constraints

- Hard Real-Time vs. Soft Real-Time
  - *Hard Real-Time:* missing a deadline results in failure (i.e. no value for the computation) or catastrophe (i.e., people die)
  - *Soft Real-Time:* performance/usefulness degrades if deadlines missed
- Programs often have deadlines and scheduler must do its job trying to meet those deadlines



Real-Time Scheduling Strategies

- *Over provisioning*
  - ensure the hardware is more than needed to keep up with the software workload
  - ensure utilization (fraction of time system is busy) is never too high
- Scheduling is almost always based on priority
  - highest priority ready thread is chosen
- A more abstract scheduling strategy is Earliest Deadline First (EDF)
  - choose the next thread to run based on the earlier deadline *(time at which the job must finish)*
- *Priority donation* (also known as priority inheritance)
  - solves *priority inversion* problem by having higher priority tasks that need a resource held by a low priority task to donate its high priority to the low priority task temporarily
    - priority donation doesn't solve all priority inversion problems