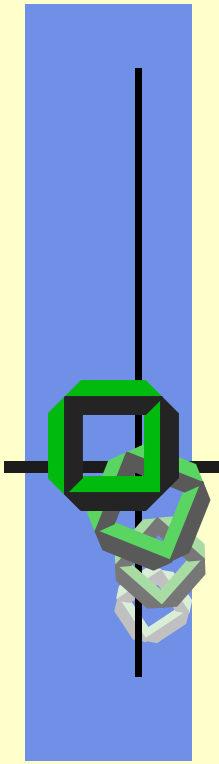
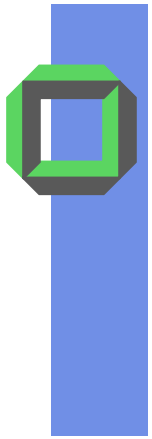


System Architecture

13 Scheduling

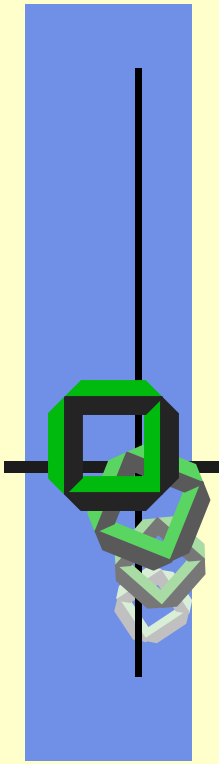
Theory & Practice
Design Parameters
Performance Measures
Scheduling Algorithms

December 8 2008
Winter Term 2008/09
Gerd Liefländer

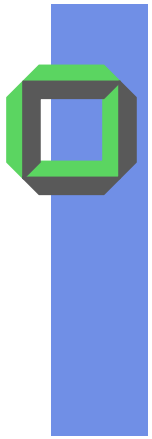


Agenda

- Motivation & Introduction
- Theory & Practice
- Scheduling Problems
- Performance Measures
- Design Parameters
- Scheduling Policies



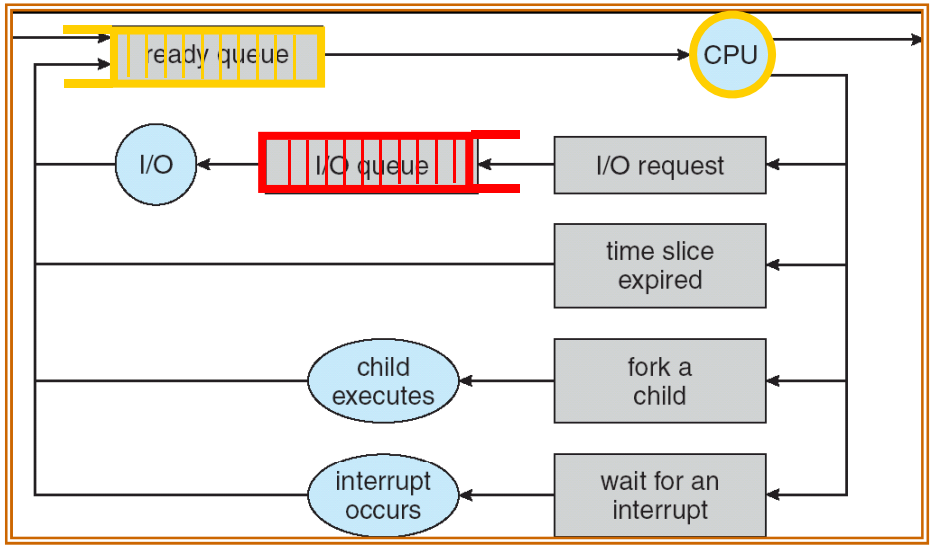
Design Parameters



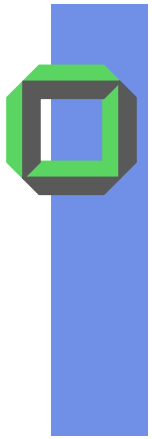
Scheduling?

- Any scheduling is correlated to some *policy*
- A scheduling policy can depend on:
  - kind of usage of the HW/SW resource
    - CPU versus I/O scheduling
  - operating mode of system
    - batch, interactive, real-time
  - instant of time when scheduling takes place
    - online versus offline
  - predictability of process/thread behavior
    - deterministic or probabilistic
  - symmetric versus asymmetric scheduling
    - With symmetric scheduling each CPU is assumed to can execute each process/KLT
  - computer architecture (UP versus MP)
  - duration of scheduling (long-, medium, short-term scheduling)

CPU Scheduling Environment



- Earlier, we talked about the life-cycle of a process/thread
  - Activated threads work their way from ***ready queue*** to ***running (queue)*** to various ***waiting queues***.
- *How does OS decide which of several threads to take off of a queue?*
  - Most interesting queue to worry about is the **ready queue**
  - However, the other queues can be scheduled as well
- **Scheduling**: deciding which activities (e.g. processes/KLTs) are given the chance to use the resources (e.g. CPU, …)



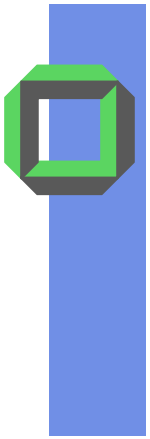
General* Scheduling Problem

When to assign executable units (processes, KLTs) to executing units (e.g. CPUs)?

∃ general criteria for scheduling?

- overall system goals
- specific constraints
  - time-critical processes/KLTs should meet their "deadlines", i.e. they must execute in time, *neither too early, nor too late*

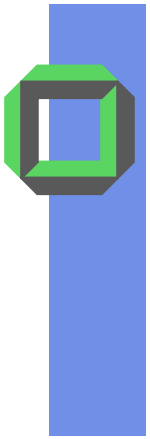
*Simplification: Focus only on the resource **CPU**



Classes of Scheduling Problems

Levels of scheduling:

- *Scheduled Unit* (SU) = *task*, i.e. application,
  e.g. which task should be *admitted next* (and for how long) and/or which task must wait until ∃ enough *system capacity*
- SU = *process/KLT* (of multi-threaded application), e.g. which admitted process/KLT should run on which CPU and/or which threads should stay in the ready state even if some CPUs are available again
- SU = *sequence of operations* of a thread on a pipelined or superscalar CPU



Concrete Scheduling Problems

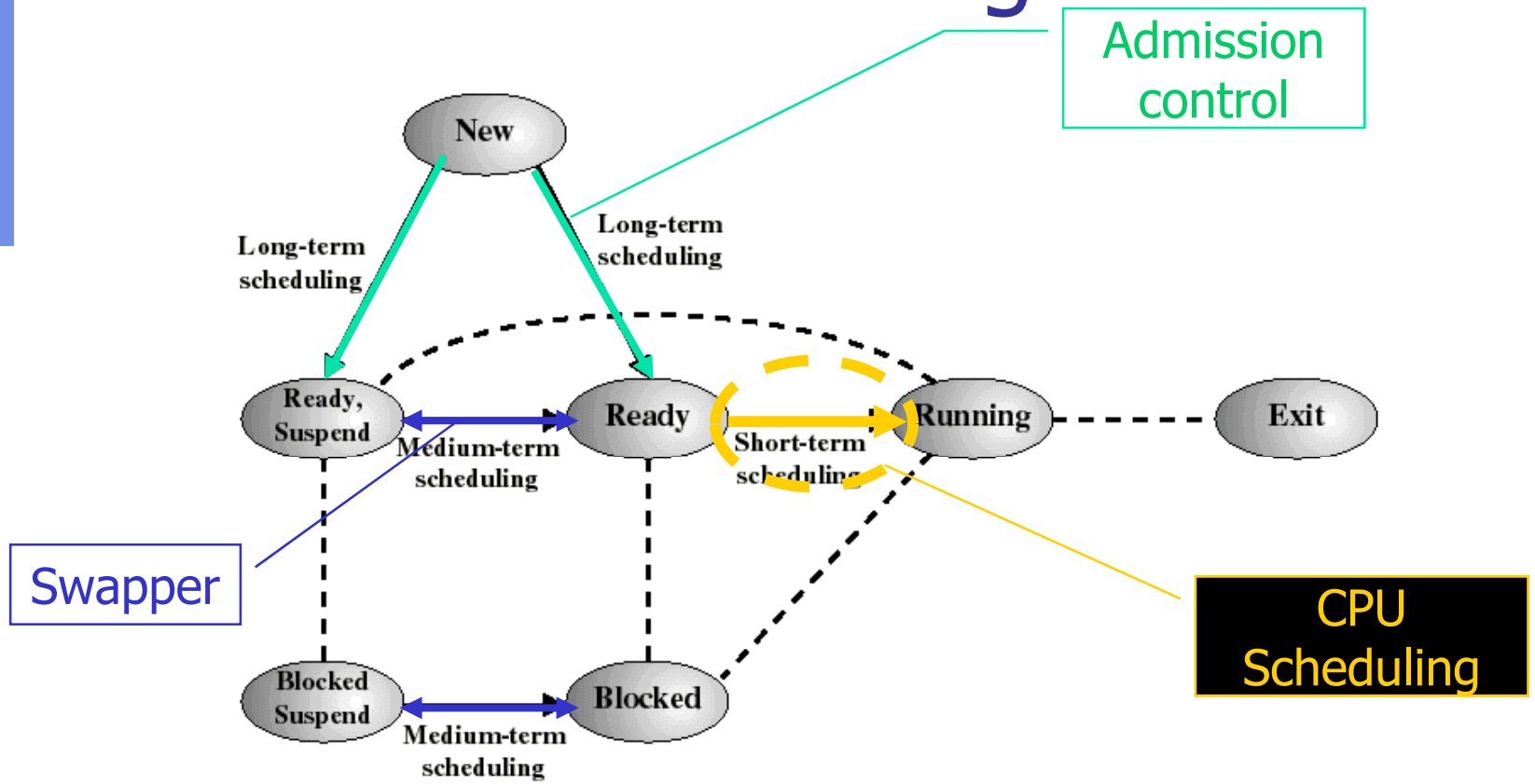
- Assume: in a multi-programming system

  n ≥ 1 KLTs are ready

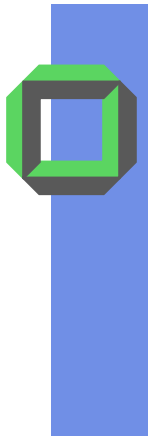
  - *Which of these ready KLTs should run next?*

- On your PC you're watching a YouTube video
  - How to manage that all the network software, decoding, output to screen and audio is well done even though in the *background* you have initiated a heavily *compute-bound process*

Duration of Scheduling



Long-term	[s – min]	which process to admit
Medium-term	[ms - s]	which process to swap in/out
Short-term	[µs –ms]	which process to run next



Goals of Long-Term Scheduling

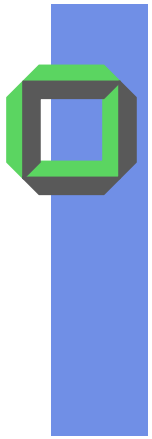
Good **mixture**[1] & high **population** of admitted tasks

- application tasks (and system tasks)
- degree of "multiprogramming" (as high as possible?)

Consequences when more processes/tasks are admitted

- It is less likely that all tasks will be blocked awaiting some event
  ⇒ better CPU usage (at least some times)
- Each task has less fraction of the CPU
  ⇒ longer turnaround/response time

[1]To find a good mixture you must be able to ***predict*** the behavior of all applications and system processes/tasks

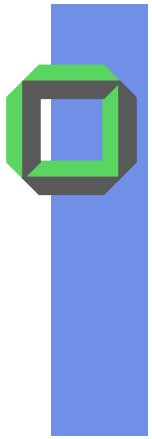


Medium-Term Scheduling

Swapping decisions based on the need to manage the multiprogramming degree

Done in context with memory management software, e.g. *swapping policy*

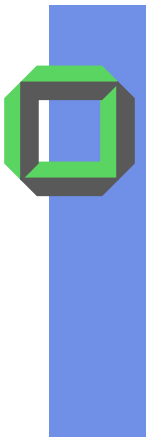
- (discussed in later chapters)
- or by some specialized regulating module
- see load control in VM



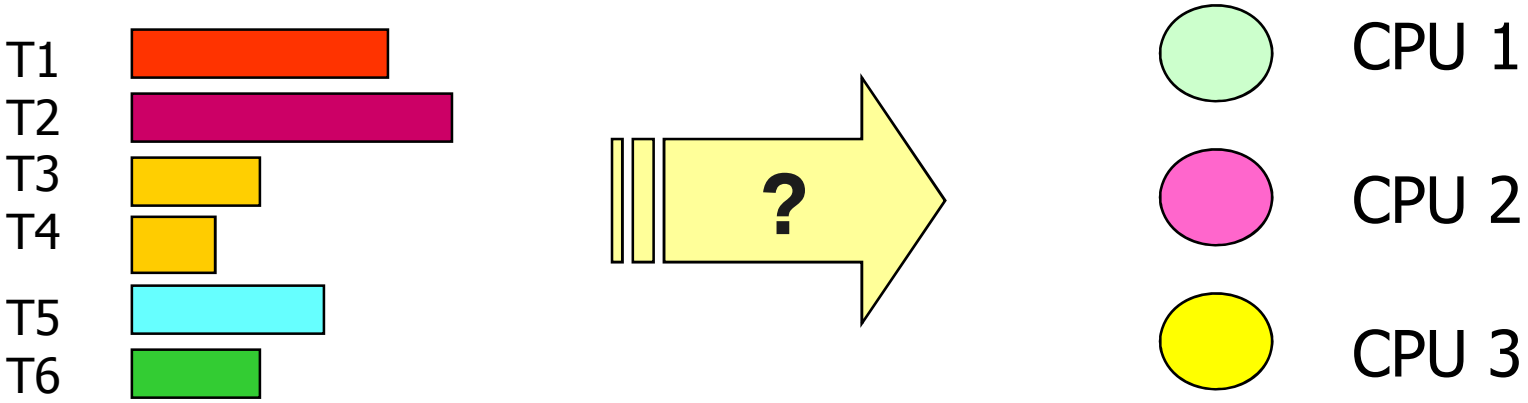
Short-Term or CPU Scheduling

The short term scheduler selecting a new KLT/process from the ready queue(s) can be invoked on one of the following *events:*

- interrupt
  - clock or I/O *or …?*
- exception
  - page fault *or …?*
- system call
  - `yield()` or `exit()` *or …?*
- notification from another KLT/process
  - signal, send_message *or …?*



Abstract Scheduling Problem

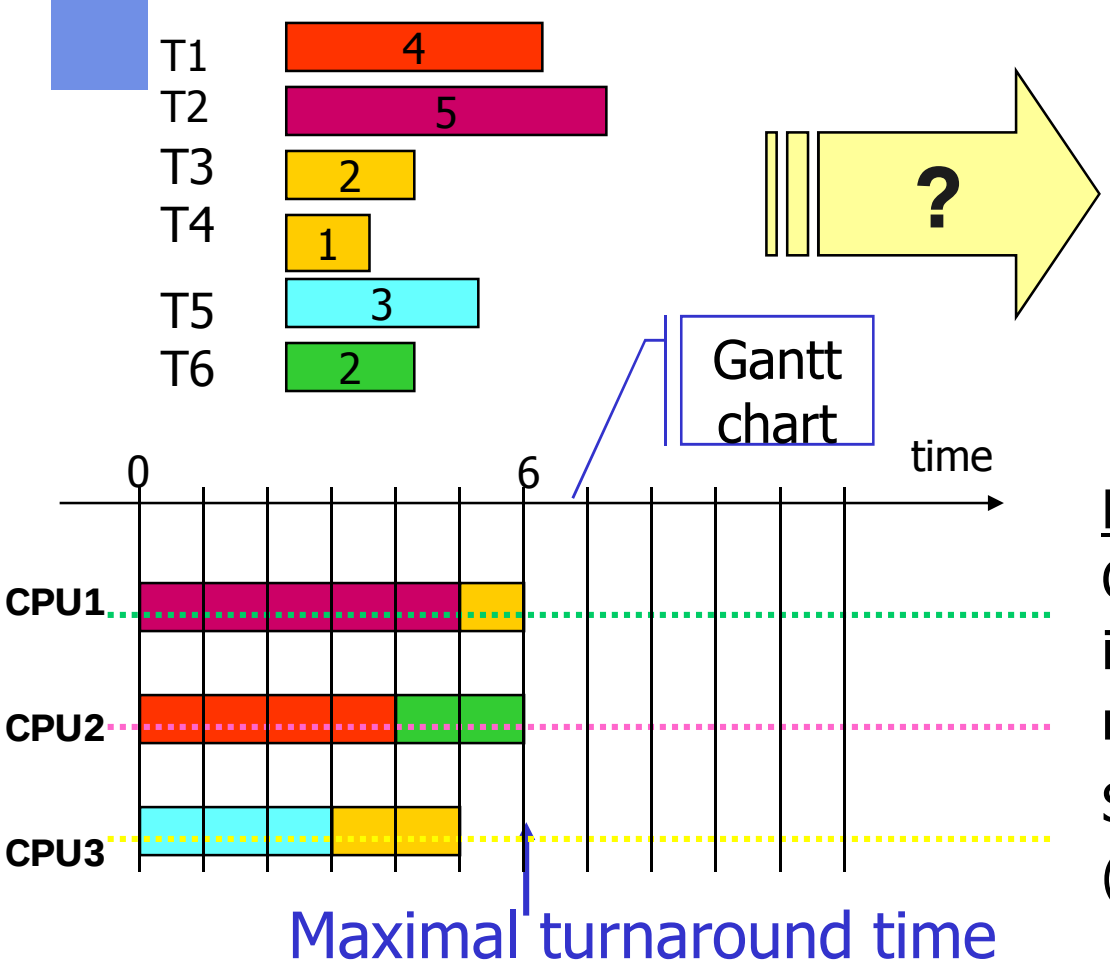


How to assign these 6 threads to 3 CPUs?

Is there an optimal schedule?

As long as there is no scheduling criterion or an accepted performance *measure*, we can neither produce a good, nor a bad schedule

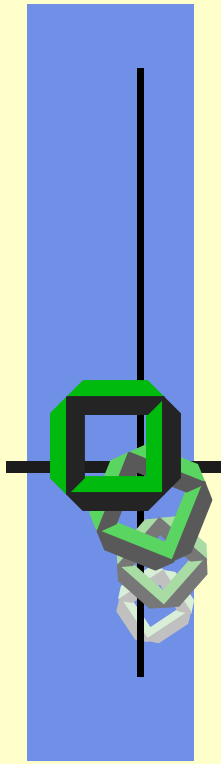
Abstract Scheduling Problem



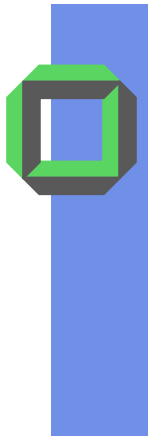
Result:

Quite a good schedule if we try to minimize max. turnaround time

Scheduling policy = LPT (Longest Processing Time First)

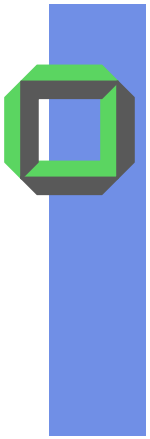


Performance Measures



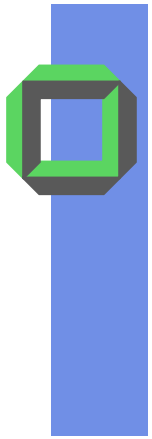
Scheduling Criterions

- User oriented
  - Response time
  - Turnaround time
  - Meeting the deadlines
  - Predictability
- System oriented
  - Throughput
  - CPU utilization
  - Fairness
  - Priorities
  - Load balance



Quantitative Metrics

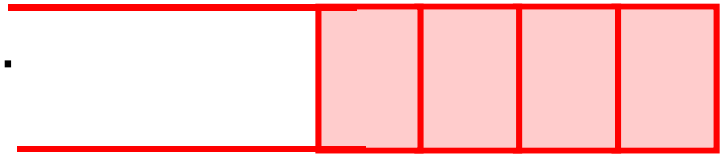
- High *processor utilization*
  - ~ percentage a processor is not idle
- High *throughput*
  - number of threads completed per unit of time
- Low *turnaround time*
  - time elapsed from the submission of a request (activation of a thread) to its *completion*
- Low *response time*
  - time elapsed from the submission of a request to the *beginning of a response activity*
- Low *waiting time*
  - Time spent in waiting and ready queues(s)



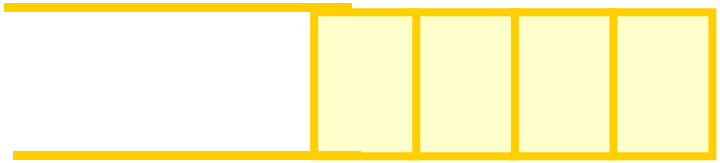
What influences Waiting Times?

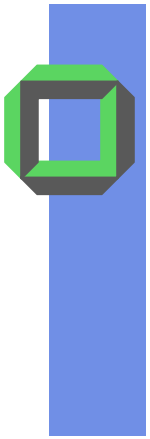
- *Contributions to the waiting time?*

1. Time a process is blocked, i.e. due to
   - a missing message
   - missing input data, …



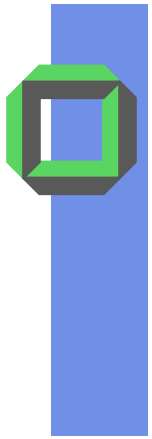
2. Time a process spends in the **ready** queue





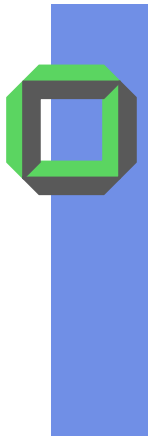
Major *Qualitative* Metrics

- No (or low) deadline violation
  - meeting all deadlines (if possible)
- High predictability
  - low variance in turnaround times and/or response times of a specific task
- High fairness
  - few starving applications
  - In MULTICS, shutting down the machine, they found a "10 year old job"
- High robustness
  - (Few) NO system crashes



Scheduling Batch Systems

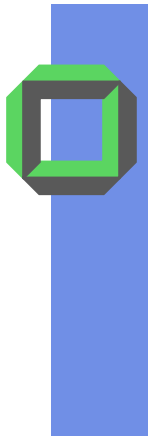
- Fairness
  - Often first come first served
- Load balancing
- Maximize throughput
  - Maximize income of computer service
- Turnaround time
  - Minimize maximal turnaround time
- CPU utilization



Scheduling Interactive Systems[1]

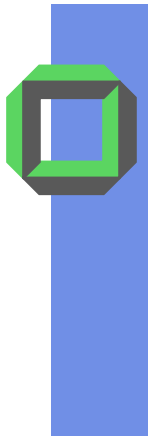
- Minimize average response time
  - Time between waiting & next I/O
  - Provide output to user as quickly as possible
  - Process input as soon as received
- Minimize *variance of response time*
  - To enhance *predictability*
  - Even a higher average can be better than a lower one, iff you can guarantee a *low variance*

[1]This & next slide from: Emery Berger, Univ. of Mass., Amherst



Scheduling Servers

- Maximize *throughput*
  - Minimize OS overhead, context switching
  - Make efficient use of CPU & I/O devices
- Minimize *waiting time*
  - Give each process same time on CPU
  - Might *increase* average response time



Priorities

Problems:

Who is allowed to determine priorities?

- Either user itself or a user-specific medium-term scheduler

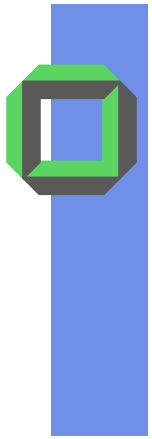
High priority value ↔ high or low priority?

- Differs from system to system (Unix = Linux it' s inverse)
- In KIT: high priority value = high priority

Scheduler always prefers threads of "*higher priority*"
⇒ low-priority threads may suffer from *starvation*

To *reduce starvation* a potentially starving thread should be able to improve its priority value based upon

- its age (see privileges of a *senior pass*) or
- its (non) execution history, i.e. its *waiting time*



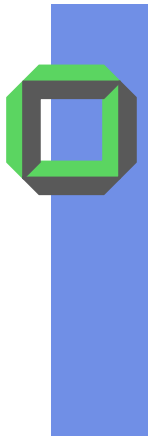
Components of Scheduling Policy

- Selection function determines which ready thread will run next, i.e.
  - it selects the next running thread amongst the ready set(s)
- Decision mode specifies the events when the selection function will be executed
  - Non preemptive

    Once a thread is running, it will continue until it terminates, or yields, or blocks (e.g. due to I/O, page fault etc.)
  - Preemptive

    Running thread is preempted (put back → ready queue)

    a) when *more urgent work* has to be done or
    b) when it has *expired its time slice*

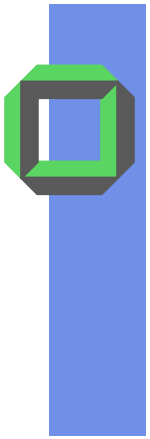


The CPU- I/O-Cycle

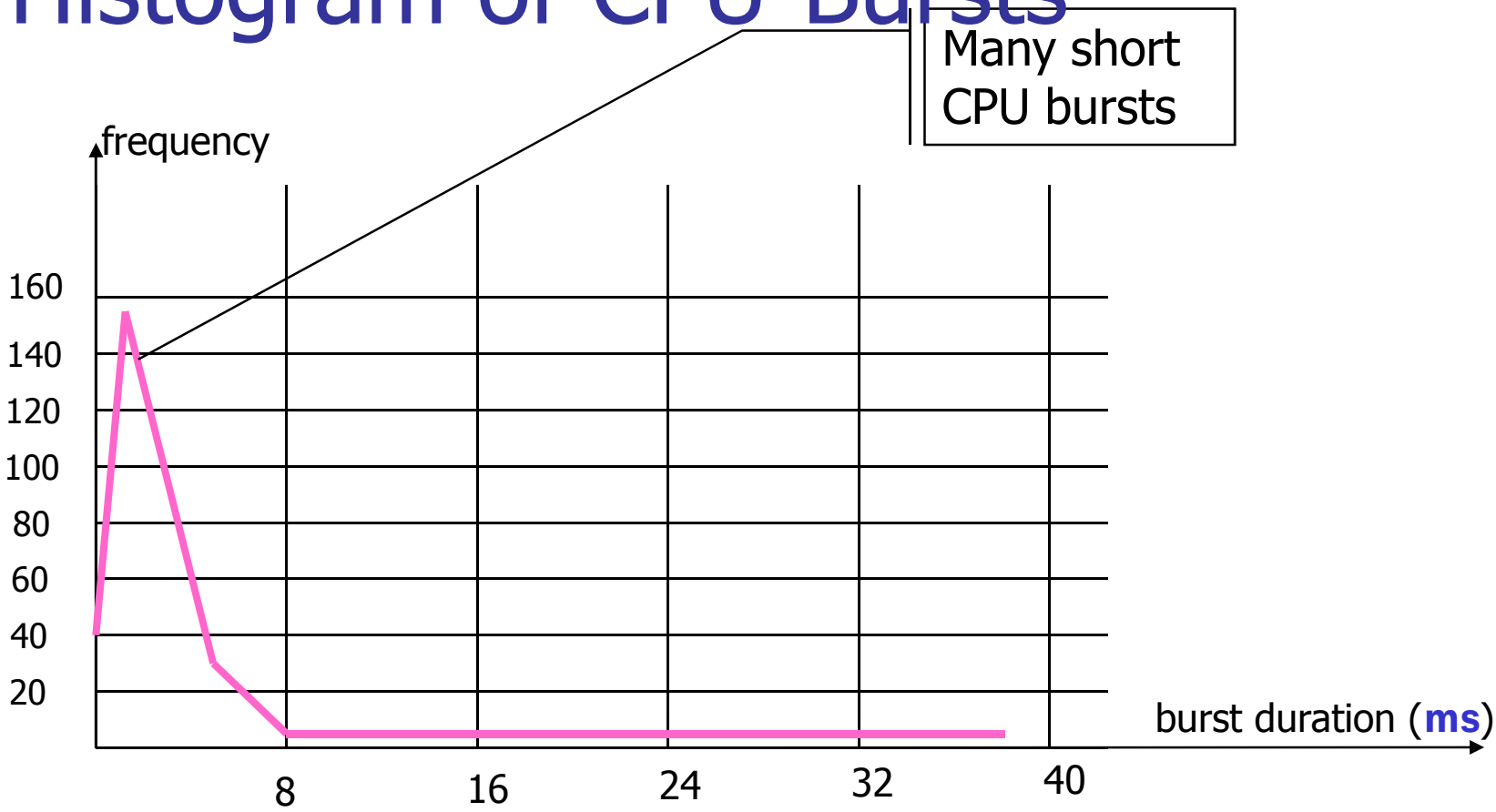
- Threads require alternate usage of CPU and synchronous I/O* and other causes for waiting times
- Each cycle: CPU burst followed by an I/O burst**
- Thread either terminates voluntarily during a CPU burst or it is preempted during an I/O or CPU burst by another thread (process)
- Typically CPU-bound threads have longer CPU bursts than I/O-bound threads

* *Any advantages with asynchronous I/O?*

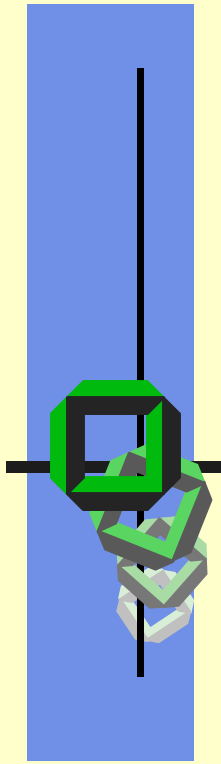
** Each I/O burst is *much longer* than a typical CPU burst



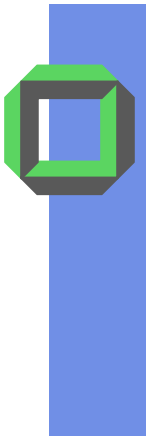
Histogram of CPU-Bursts



Remark: Above histogram is taken from Silberschatz
Don't rely anymore on the absolute values of time



Scheduling Policies



Example Scheduling Problem*

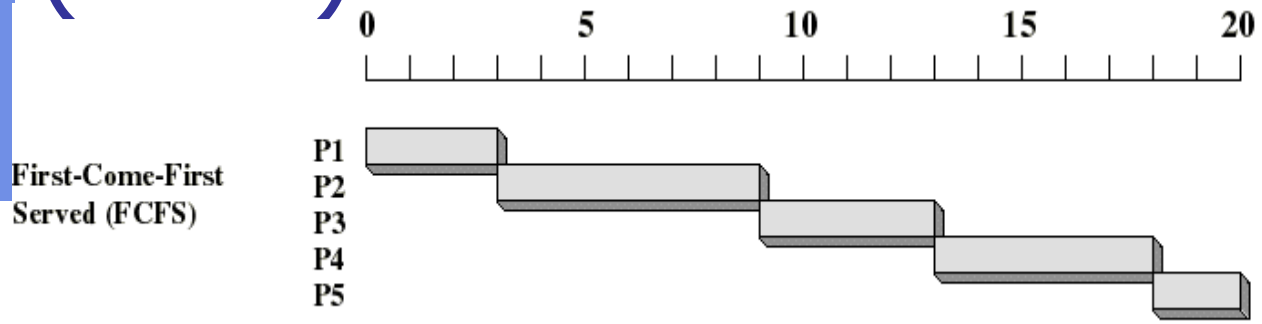
Thread	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

Service time = total processor time needed

"Jobs" with "long service time" are CPU-bound jobs
~ "long jobs" (see thread 2 above)

*Example is used to analyze effects of various scheduling policies

First Come First Served Policy (FCFS)

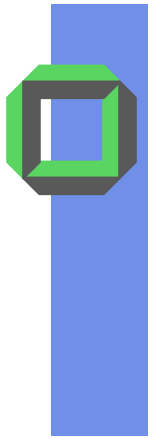


Selection function: select the **oldest** ready thread

Decision mode: non preemptive, thread runs until it

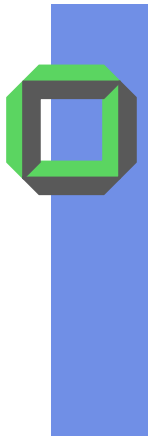
- cooperates (e.g. yields) or
- blocks itself (e.g. initiates an I/O) or
- does an exception (being killed) or
- terminates

Remark: Many things in daily life are scheduled according to FCFS. It's fair & quite simple



How to implement effective FCFS?

- Admitting a new process/KLT is quite easy, i.e. just append it to the ready queue
  - All the other ready activities must be older
  - Assume: your admission control had not swapped out
- *How to deal with a process/KLT, when it is unblocked, i.e. when it enters the ready queue again?*
  - It has been waiting for a while, ⇒ simple appending does not help, because in the ready queue might be younger ones
- Use a ***time stamp*** when the process/task has been created (or admitted for the first time)
  - *Sufficient for an effective FCFS?*



Analysis of Non Preemptive FCFS

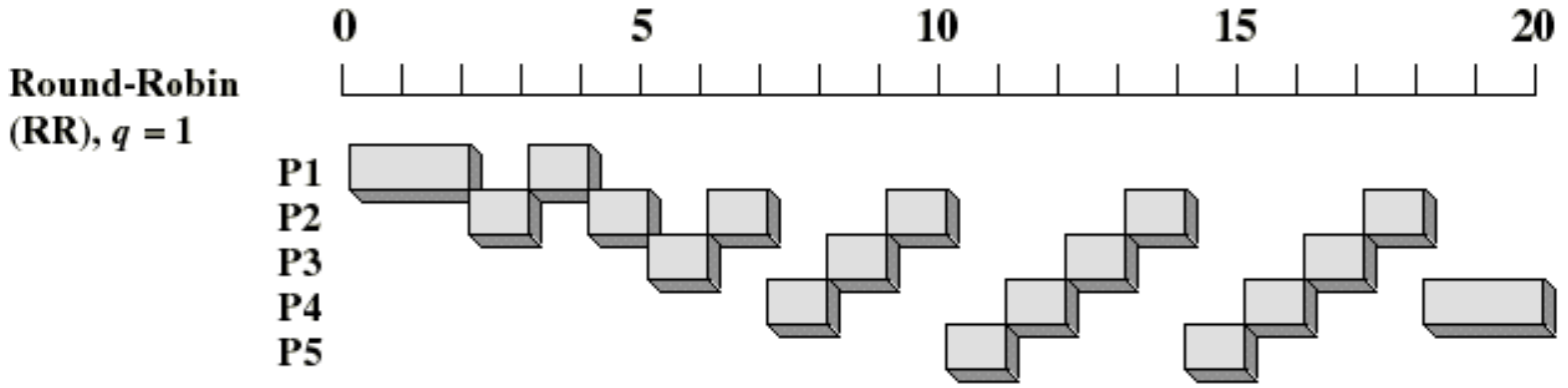
Convoy effect: a couple of short runners behind a long runner

Processes with few I/O can cause a ***tailback*** in front of the CPU, thus monopolizing the CPU

FCFS implicitly favors CPU-bound processes/KLTs:

- I/O-bound threads have to wait after each I/O until the current running thread leaves the CPU $\Rightarrow$
- Low usage of I/O devices

Round Robin (Time Slicing)



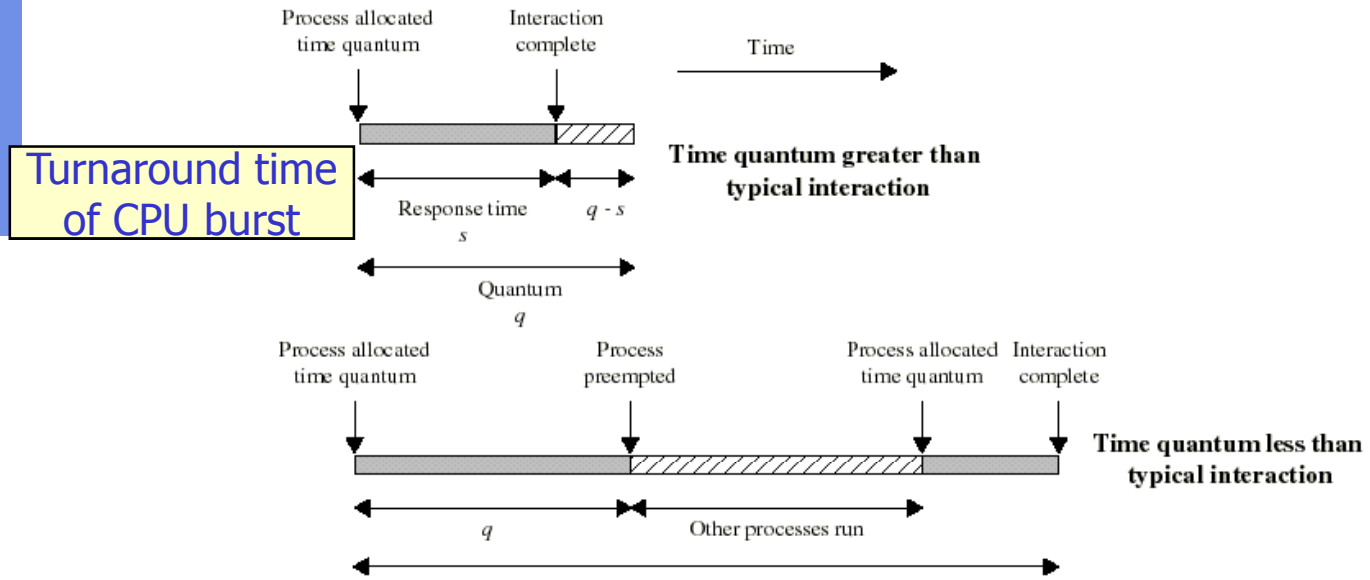
Selection function: select first thread in ready queue

Decision mode: ("time") preemptive

- A process/KLT is allowed to run until its time slice TS ends (TS ∈ [1, 100]* ms)
- When a timer interrupt occurs, the running process/KLT is *appended* to the ready queue

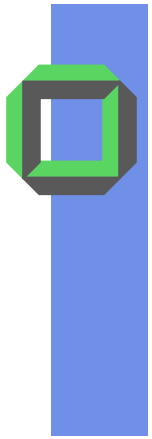
*Depends on application and HW system

Size of Time Slice



Recommendation (initially Unix TS = 1 s):

1. TS larger than time required to handle clock interrupt + dispatching, otherwise *inefficient*
2. TS larger than execution time of a *typical interaction CPU burst* (but not too large to avoid penalizing I/O bound jobs), otherwise *ineffective* and more and more like FCFS

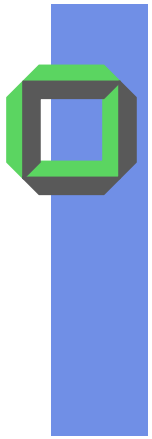


FCFS versus RR

- Assuming zero-cost switch-time, is RR always better than FCFS?
- Simple example: 10 KLTs, each take 100s of CPU time
  RR scheduler quantum of 1s
  All KLTs start at the same time
- Completion Times:

Job #	FIFO	RR
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

  - Both RR and FCFS finish at the same time, i.e max. TT is identical
  - Average turnaraound time is much worse under RR!
    - Bad when all KLTs have same pure execution time
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
  - Total time for RR much longer even for zero-cost switch!



Analysis of Round Robin

- Inherently favors CPU-bound threads

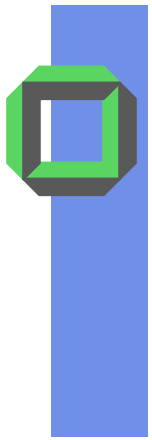
  I/O bound thread doesn't use up its TS, it is blocked waiting for I/O. CPU-bound thread uses its TS, is put back to ready queue ⇒ it can overtake an I/O-bound thread several times (failed box stop in formula 1)

- Haldar's "approach": **Virtual Round Robin**

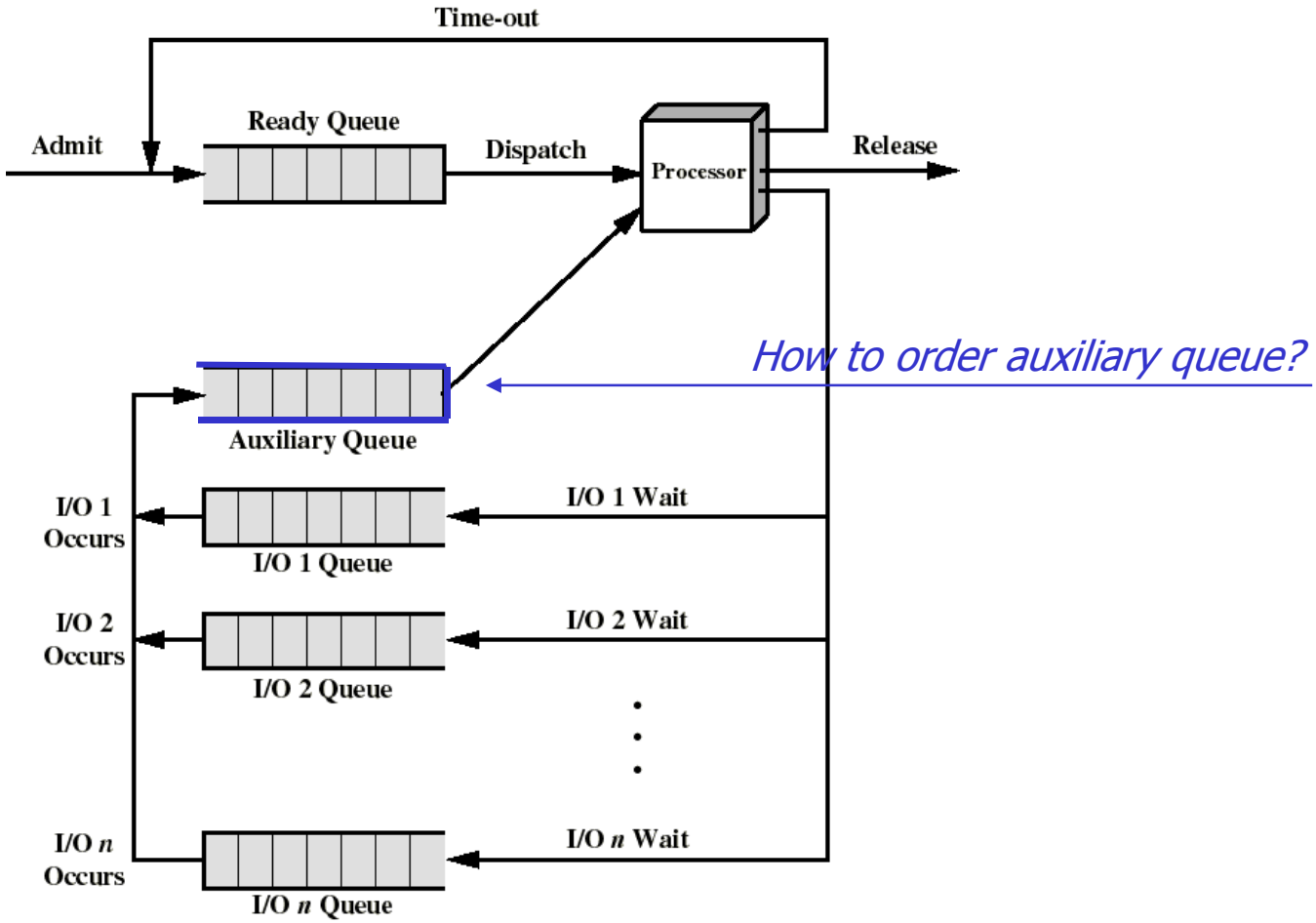
  When I/O has completed, blocked KLT is moved to an auxiliary queue (preference over main ready queue)

  "Such a thread being dispatched from the auxiliary queue, runs ***no longer*** than the basic time quantum "minus" the time it was running in its previous TS

How to improve this approach?

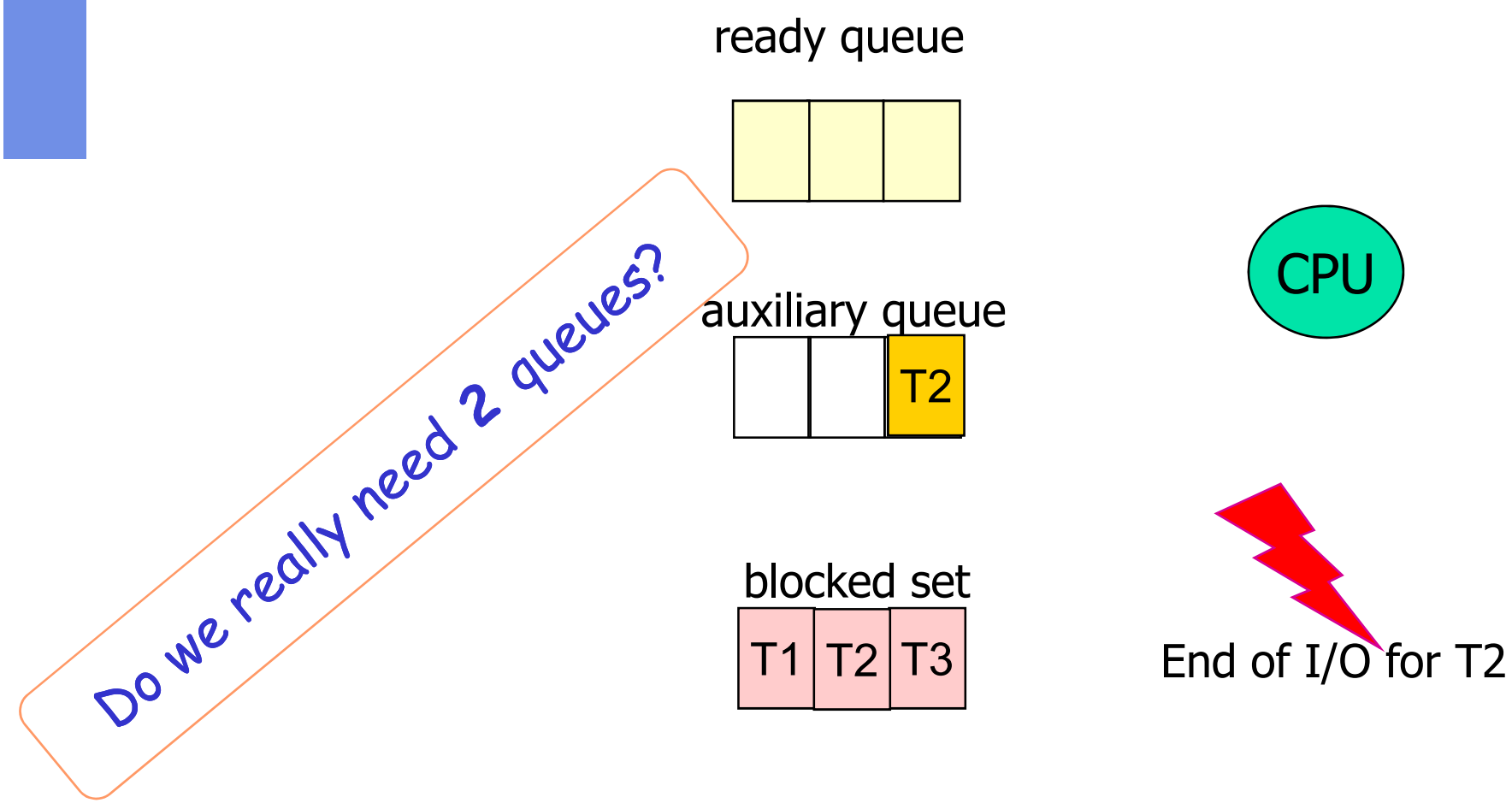


Queuing Model: Virtual RR

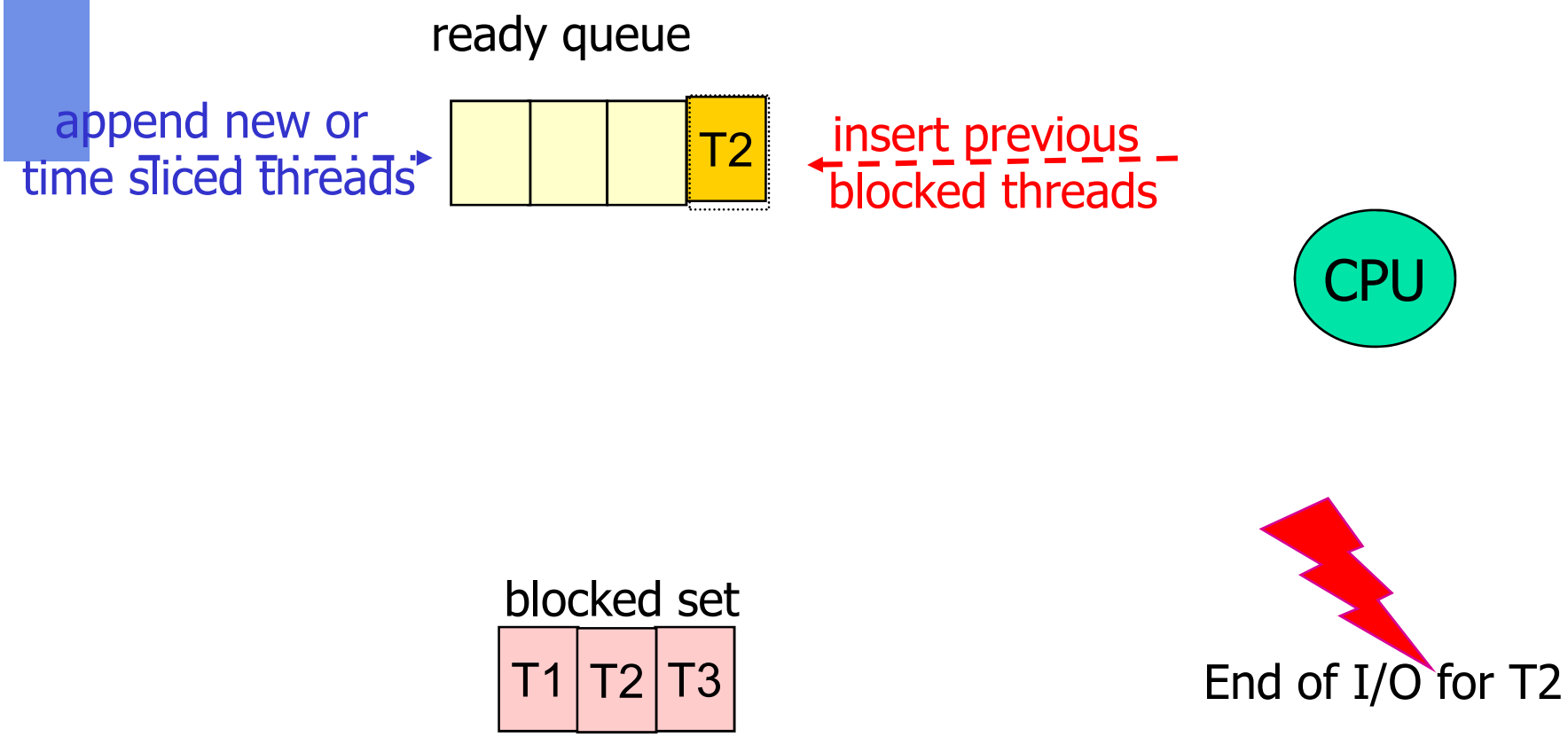


How to order auxiliary queue?

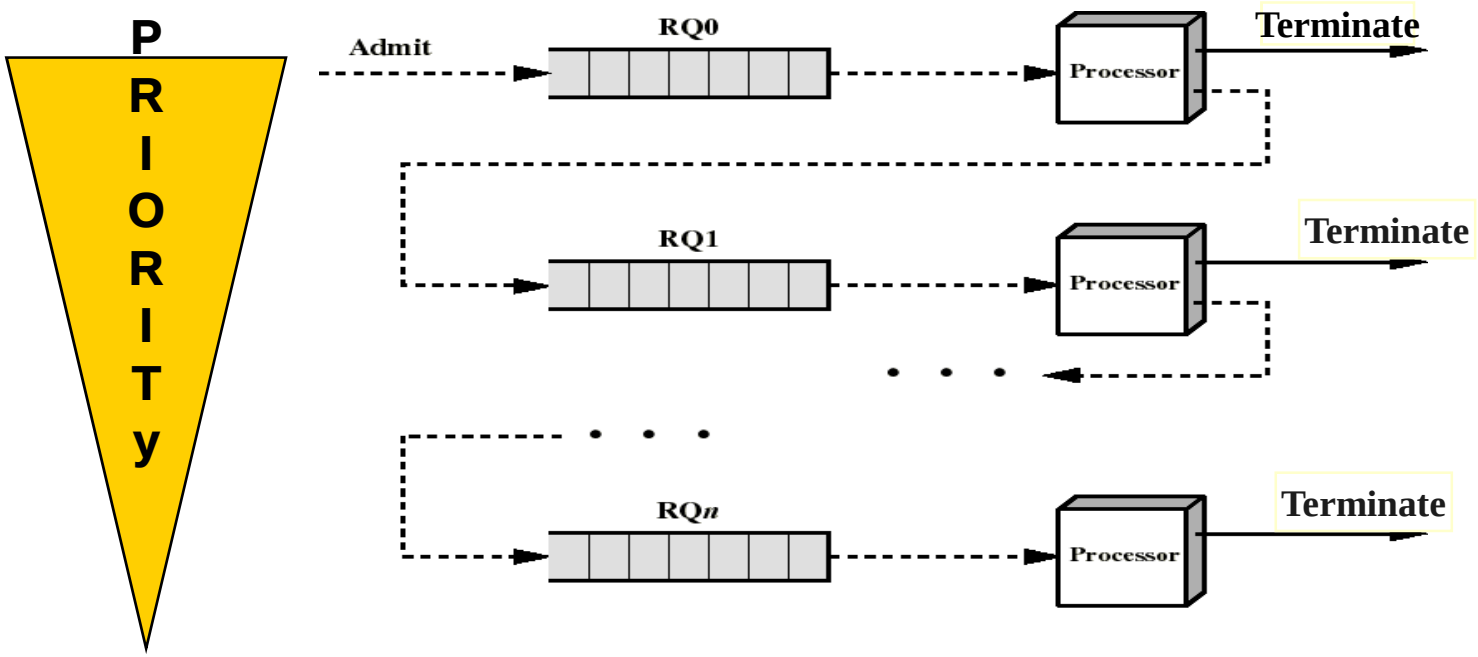
Problem with 1 or 2 Queues



Problem with 1 or 2 Queues



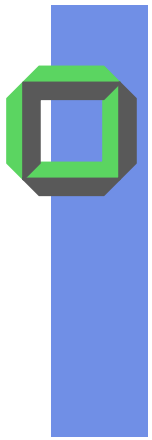
Multilevel Feedback*



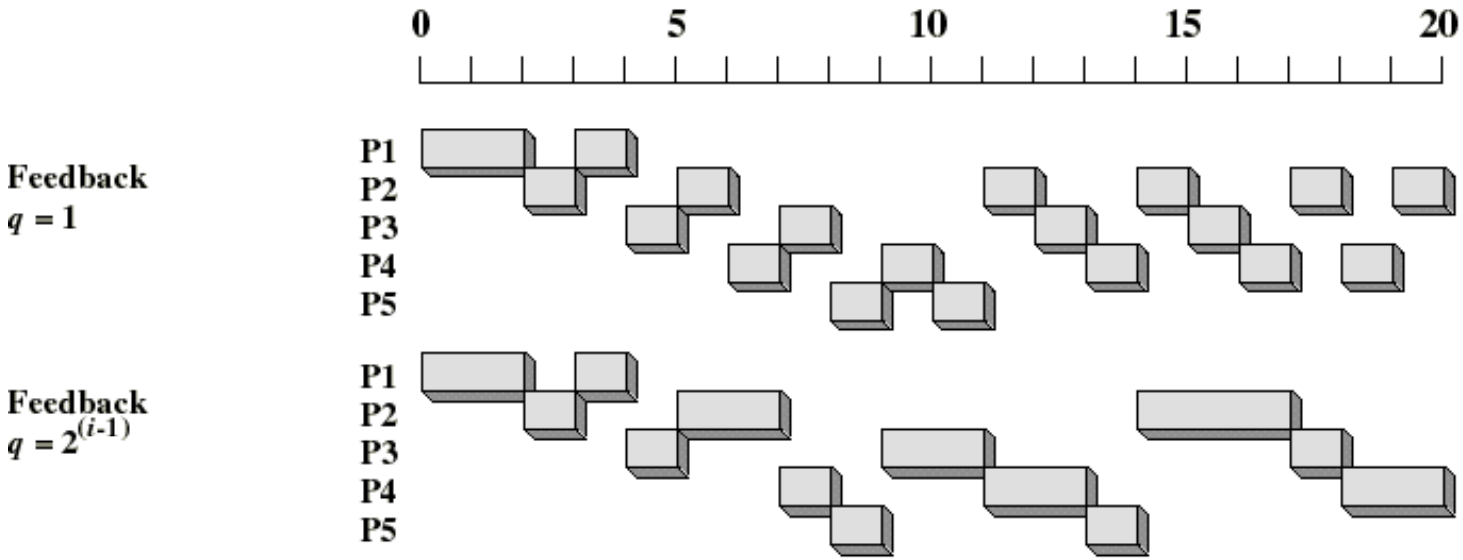
Selection function: first thread in highest ready queue

Decision mode: Preemptive (at least due to time slices)
However, you may also add priority preemption

[1]CTSS started in 1961 at MIT (reused in MULTICS)



Time Quanta: Multilevel Feedback



With fixed time quanta turnaround times of long threads can stretch out alarmingly, i.e. average turnaround time of long runners increase

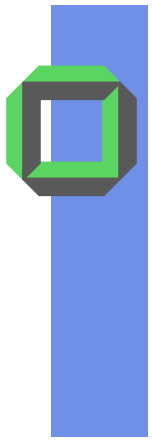
$\Rightarrow$

increase time quanta according to level of the queue

Example: time quantum of $RQ_i = 2^{i-1}$

If we would know the Future?

- Shortest Job First (SJF):
  - Run whatever job has the least amount of computation to do
  - Sometimes called "Shortest Time to Completion First" (STCF)
- Shortest Remaining Time First (SRTF):
  - Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time of the current job, immediately preempt CPU
  - Sometimes called "Shortest Remaining Time to Completion First" (SRTCF)
- These can be applied either to a whole program or to the current CPU burst of each program
  - Idea is to get short jobs out of the system
  - Big effect on short jobs, only small effect on long ones
  - Result is better average turnaround & response time



Estimating next CPU Burst

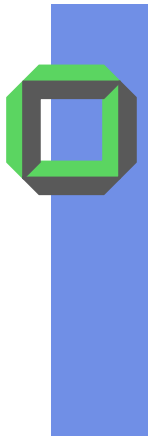
- Let T[i] be execution time for i-th instance of this thread, i.e. the actual duration of the i-th CPU burst of this thread
- Let S[i] be the predicted value for the i-th CPU burst of this thread. The simplest choice is:

$$S[n+1] = (1/n) \sum_{\{i=1 \text{ to } n\}} T[i]$$

- To avoid recalculating entire sum we can rewrite this as:

$$S[n+1] = (1/n)\, T[n] + ((n-1)/n)\, S[n]$$

This convex combination gives equal weight to each instance



Estimating next CPU Burst

Recent instances are more likely to reflect future behavior,

⇒ use exponential averaging

$S[n+1] = \alpha\, T[n] + (1-\alpha)\, S[n]$; $0 < \alpha < 1$

more weight is put on recent instances whenever $\alpha > 1/n$

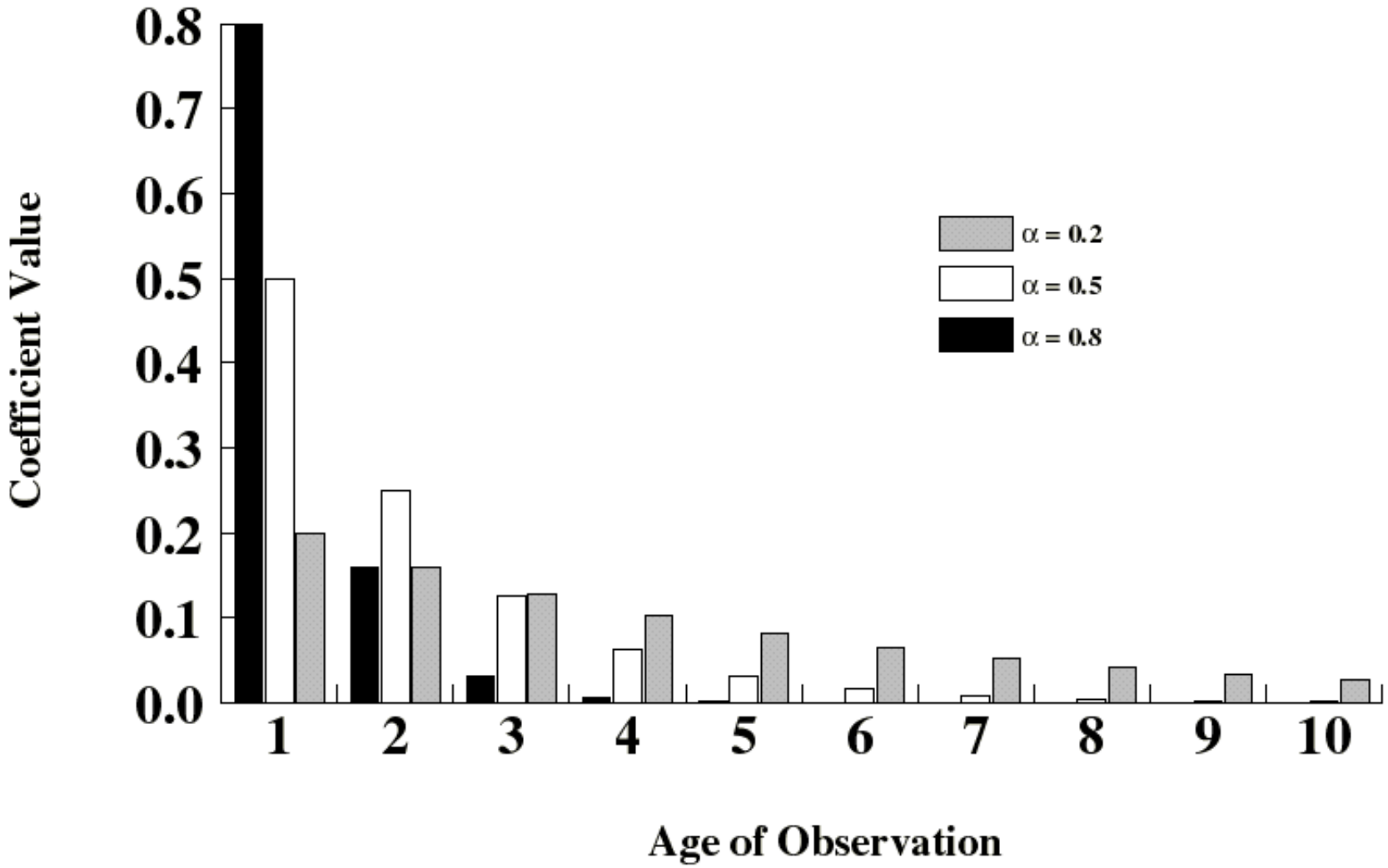
⇒ weights of past instances are decreasing exponentially

$$S[n+1] = \alpha T[n] + (1-\alpha)\alpha T[n-1] + \ldots + (1-\alpha)^{i}\alpha T[n-i] + \ldots + (1-\alpha)^{n} S[1]$$

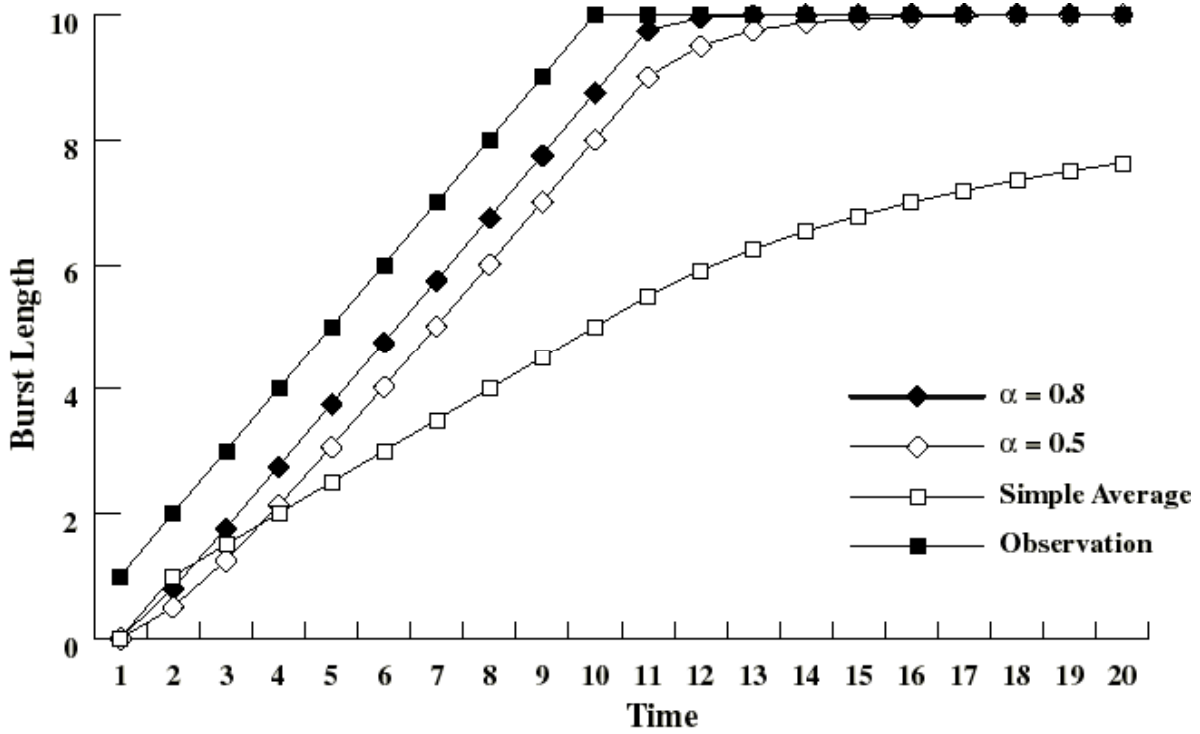
predicted value of 1st instance S[1] is not calculated;

usually set to 0 to give a standard priority to new threads

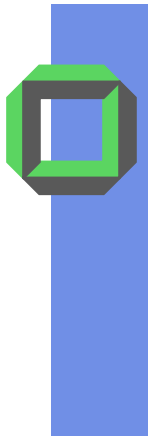
Exponentially Decreasing Averaging Coefficients



Use of Exponential Averaging

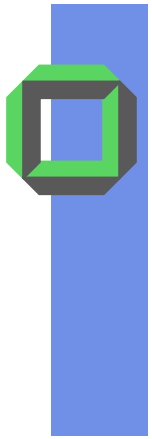


Here S[1] = 0 to give high priority to new threads. Exponential averaging tracks changes in threads behavior much faster than simple averaging



Analysis of (R)SJN

- Starvation for long processes/KLTs as long as there is a steady supply of short ones
- Without preemption ⇒ not suited in time sharing systems, CPU bound KLT gets lower preference, but a thread doing no I/O still can monopolize the CPU
- (R)SJN implicitly incorporates priorities: shorter jobs are given preferences, but **RSJN** is **not fair**
- Somehow need to predict the future
  - How can we do this?
  - Some systems ask the user
    - When you submit a job, have to say how long it will take
    - To stop cheating, system kills job if takes too long
  - But: Even non-malicious users have trouble predicting runtime of their jobs



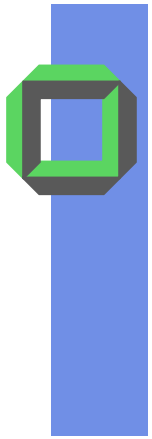
Highest Response Ratio Next*

Response Ratio (~ inherent a dynamic priority):

r := (waiting-time + processing-time) / processing-time

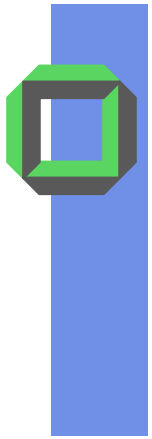
Selection function:	thread with highest response ratio
Decision mode:	non preemptive
Comment:	Shorter jobs are favored, however, longer jobs do not have to wait forever, because their response ratio increases the longer they wait.

*Tanenbaum's Guaranteed Scheduling



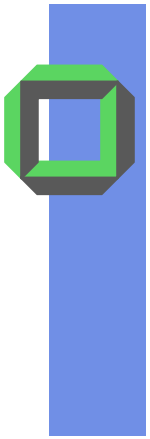
Lottery Scheduling

- Give each process/KLT some lottery tickets
- On each TS, randomly pick a ticket, ticket owner gets CPU (~lottery)
- Scheduling behavior is dependent on number of tickets a process/KLT owns
- How to assign tickets?
  - To approximate SRTF, short runners get more, long runners get fewer
  - To avoid starvation, every job gets **at least one** ticket (everyone makes progress)



Lottery Scheduling

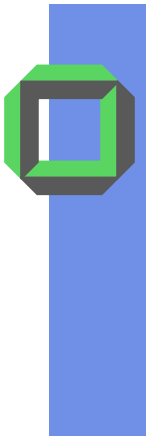
- Advantage over strict priority scheduling: behaves gracefully as load changes
  - Adding or deleting a process/KLT affects all others proportionally, independent of how many tickets each one possesses
- Task can hand over ticket to other tasks, e.g. a client to a server (called "*ticket donation*")
- *How to implement tickets and ticket picking?*



Example: Lottery Scheduling

- Assume short jobs get 10 tickets, long jobs get 1 ticket

# short jobs/ # long jobs	% of CPU each short jobs gets	% of CPU each long jobs gets
1/1	91%	9%
0/2	N/A	50%
2/0	50%	N/A
10/1	**9.9%**	**0.99%**
1/10	50%	5%



Fair Share Scheduling

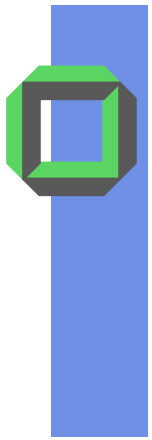
In a multi-user system, each user can run several tasks concurrently, each one consisting of some threads

Users may belong to user groups and each user group should have its fair share of the CPU

This is the basic philosophy of fair share scheduling

Example:

If there are 4 equally important departments (groups) and one department has more threads than the others, degradation of response time or turnaround time should be more pronounced for that department



The Fair Share Scheduler

Has been implemented on some Unix OSes. Processes (tasks) are divided into groups, group k gets a fraction W_k of the CPU-capacity

The priority Pj[i] of process j (belonging to group k) at time interval i is given by:

$$P_j[i] = B_j + (1/2)\ CPU_j\ [i-1] + GCPU_k[i-1]/(4W_k)$$

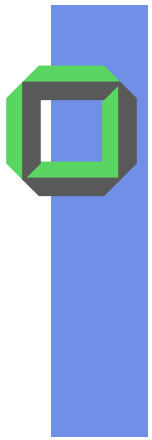
A high value means a low priority

Process with smallest $P_j[i]$ is executed next

B_j = base priority of process j

$CPU_j[i]$ = Exponentially weighted average of processor usage by process j in time interval i

$GCPU_k[i]$ = Exponentially weighted average processor usage by group k in time interval i



The Fair Share Scheduler

The exponentially weighted averages use a = 1/2:

$$CPU_j[i] = (1/2)\, U_j\,[i-1] + (1/2)\, CPU_j[i-1]$$

$$GCPU_k[i] = (1/2)\, GU_k\,[i-1] + (1/2)\, GCPU_k\,[i-1]$$

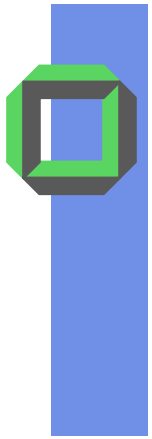
where

- $U_j[i]$ = processor usage by process j in interval i
- $GU_k[i]$ = processor usage by group k in interval i

Recall that

$$P_j[i] = B_j + (1/2)\, CPU_j\,[i-1] + GCPU_k[i-1]/(4W_k)$$

Priority decreases as the process and its group use the processor



Priority Scheduling

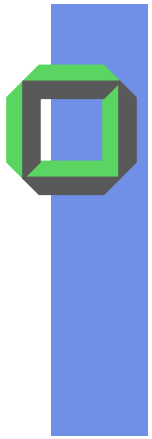
Selection function: ready thread with highest priority

Decision mode: preemptive (more complicated) or non preemptive

Drawbacks of non preemptive: Danger of

- *starvation and/or*
- *priority inversion*

Remark: Most PC-OSes offer priority based scheduling (with preemption and dynamic priorities.) + some sort of time slicing



Analysis of Scheduling Policies

Which scheduling policy is the best one?

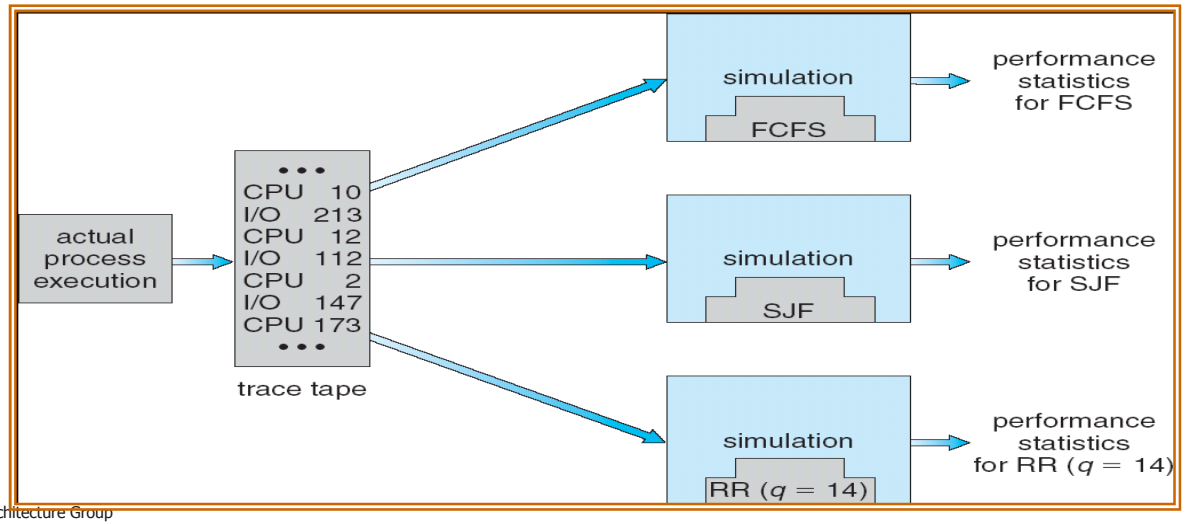
Answer may depend on:

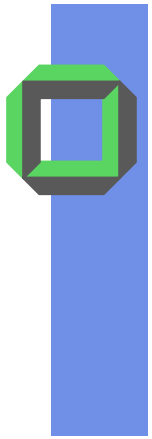
- system workload (extremely variable)
- hardware support for dispatcher
- relative weighting of performance criteria
- (response time, CPU utilization, throughput...)
- *evaluation method* used (each has its limitations)

⇒ Hence answer depends on too many factors to give a conclusive and satisfying answer

Evaluation of Scheduling Policies

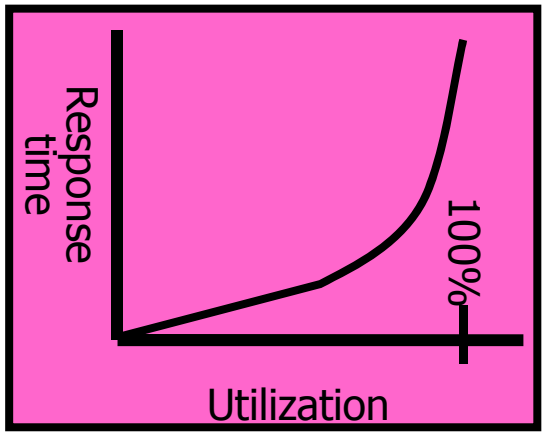
- Deterministic models
  - take a predetermined workload and compute the performance of each algorithm for that workload
- Queueing models
  - approach for handling stochastic workloads
- Implementation/Simulation:
  - Build system which allows actual algorithms to be run against real workloads/benchmarks



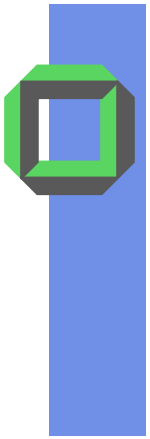


Concluding Remarks

- Some **scheduling gurus**: "You can do all with priorities"
- Others: **"Don't use priorities at all"**
- When do details of a scheduling policy really matter?
  - When there aren't enough resources to go around
- When should you simply buy a faster computer?
  - One approach: Buy it when it will pay for itself in better turnaround times
    - Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization⇒100%



- An interesting implication of this curve:
  - Most scheduling algorithms work fine in the "linear" portion of the load curve and fail otherwise
  - ⇒ buy a faster CPU when you hit "knee" of curve



Common Scheduling Policies

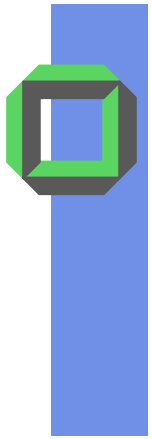
OSes supporting interactive applications often schedule with preemption

Commercial systems *often* use a combination of

- time slice mechanisms (i.e. preemption by time) and
- priorities (classifying different task classes)

Priorities are *often* a combination of

- static part (classifying the task type)
- dynamic part mirroring the behavior of the task and/or overall load of the system



Literature

- Silberschatz, A.: Operating System Concepts (5)
- Stallings, W.: Operating Systems (9,10)
- Tanenbaum, A.: Modern Operating Systems (2, 8)
- Bacon, J.: Operating Systems (6)
- Nehmer, J.: Grundlagen moderner BS (5)