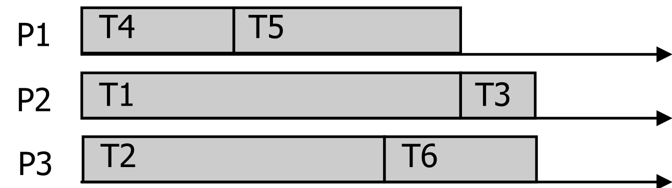
Furious activity is no substitute for understanding.
-- H. H. Williams

Chapter 4

Scheduling



4.1 Overview

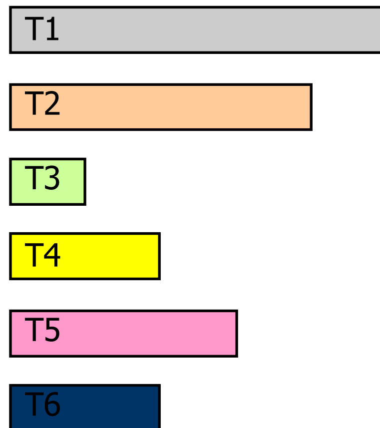
- Scheduling means the allocation of activities to functional units that can execute these activities in time and space.
- In operating systems scheduling usually means the assignment of threads to processors.
- The scheduling problem can be found at different granularity levels:
  - Processes as complete user programs have to be executed on a mono- or multiprocessor system.
  - Threads as pieces of a parallel program have to be executed on a parallel computer.
  - Mini-threads as single operations or short operation sequences have to executed on the parallel units of a processor (pipelining, superscalar architectures).

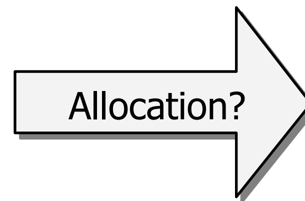
Examples

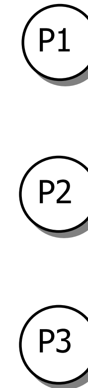
Freie Universität Berlin

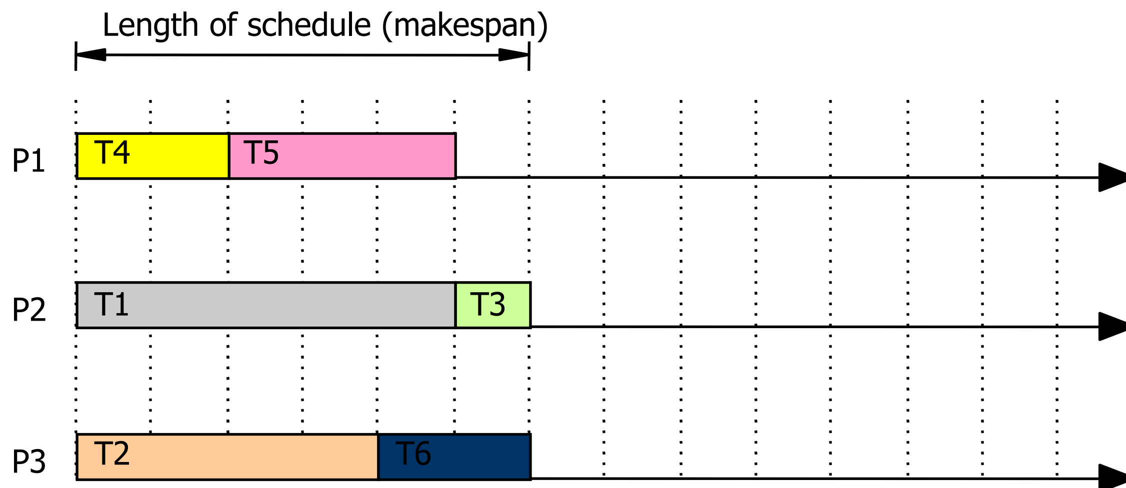
- Multi-user OS: Several threads are ready to run (ready list): Which one should be executed next?
- The compiler generates code for a superscalar processor with pipelining. It knows the instructions to be executed and the data dependencies between the instructions and has to find an allocation of the instructions to the pipelines such that the code is executed in minimal time.
- In a single user system an MPEG-video is played live from the Internet. The interplay of networking software, decoding, presentation at the display and output to the speaker has to lead to continuous synchronized play even if in the background a compiler is running.
- A production robot has to sample sensor data from different sensors at different rates and to react to them. E.g. to recognize a part on the conveyor belt it must perform time consuming calculations.
- In a multiprocessor system several parallel synchronous programs are running in a way that their threads must synchronize with each other using a barrier. Threads belonging to same program should be assigned to the processors at the same time (Gang Scheduling/co-scheduling).

Classic Scheduling Problem









Gantt-Diagram

Problem variants

- **Monoprocessor / Multiprocessor**

  When using multiprocessor machines: Processors homogeneous, i.e. all the same speed?
- **Thread set static or dynamic?**

  In the static case all threads are given and ready to run – no new arrivals.

  In the dynamic case new threads may arrive at any time during the execution.
- **Scheduling on-line** (at runtime) **or off-line** (prior to runtime)**?**

  In the Off-line-case all threads are known in advance (including future arrivals), i.e. we have complete information.

  On-line-algorithms only know the current threads and make their decisions based on incomplete information.
- **Execution times known in advance?**

  Known execution times of threads (or worst-case-estimates) are a prerequisite for real-time scheduling and helpful for algorithms.

Problem variants

- **Preemption possible?**
  Using preemption scheduling goals can be reached more easily.
- **Dependencies between the threads**
  Dependency relation (partial order)
  Synchronized allocation of parallel threads of a parallel program.
- **Communication times to be considered?**
- **Set-up times (e.g. switching) to be considered?**
- **Priorities to be considered?**
  Priorities are either given from outside (static) or defined during execution (dynamic).
- **Deadline to be considered?**
  In real-time systems some results (threads) must be available at specified times. Often those conditions have to be met periodically.
- **Which goal should be achieved**?
  Objective function to be maximized

Examples for Goals

Freie Universität Berlin

User-oriented goals

- Makespan (length of schedule) (min)
- Maximum response time (min)
- Mean (weighted) response time (min)
- Maximum lateness (min)
- ...

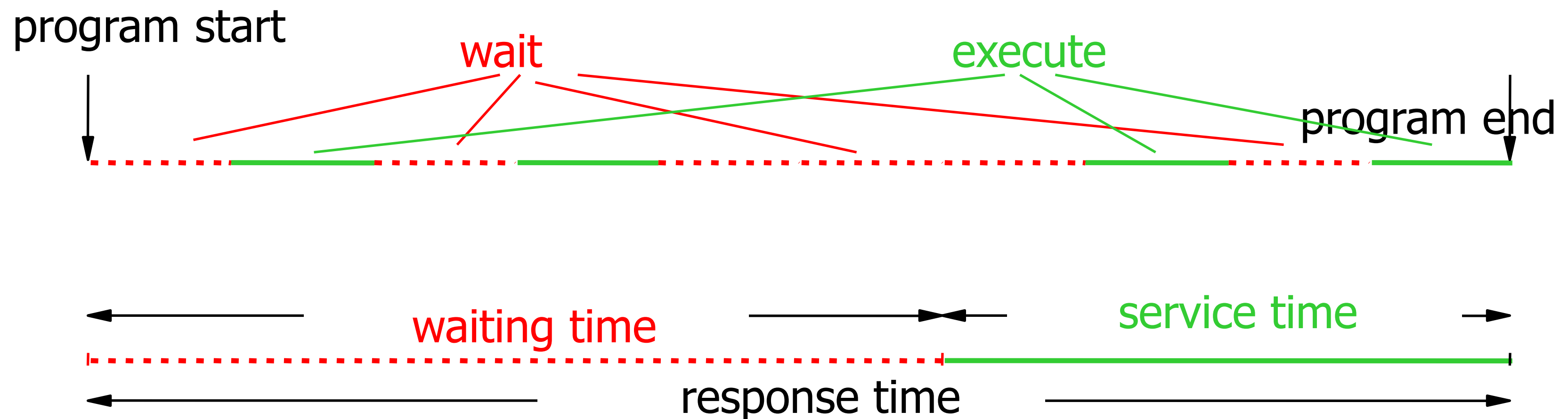
System-oriented goals

- Number of processors (min)
- Throughput (max)
- Processor utilization (max)
- ...

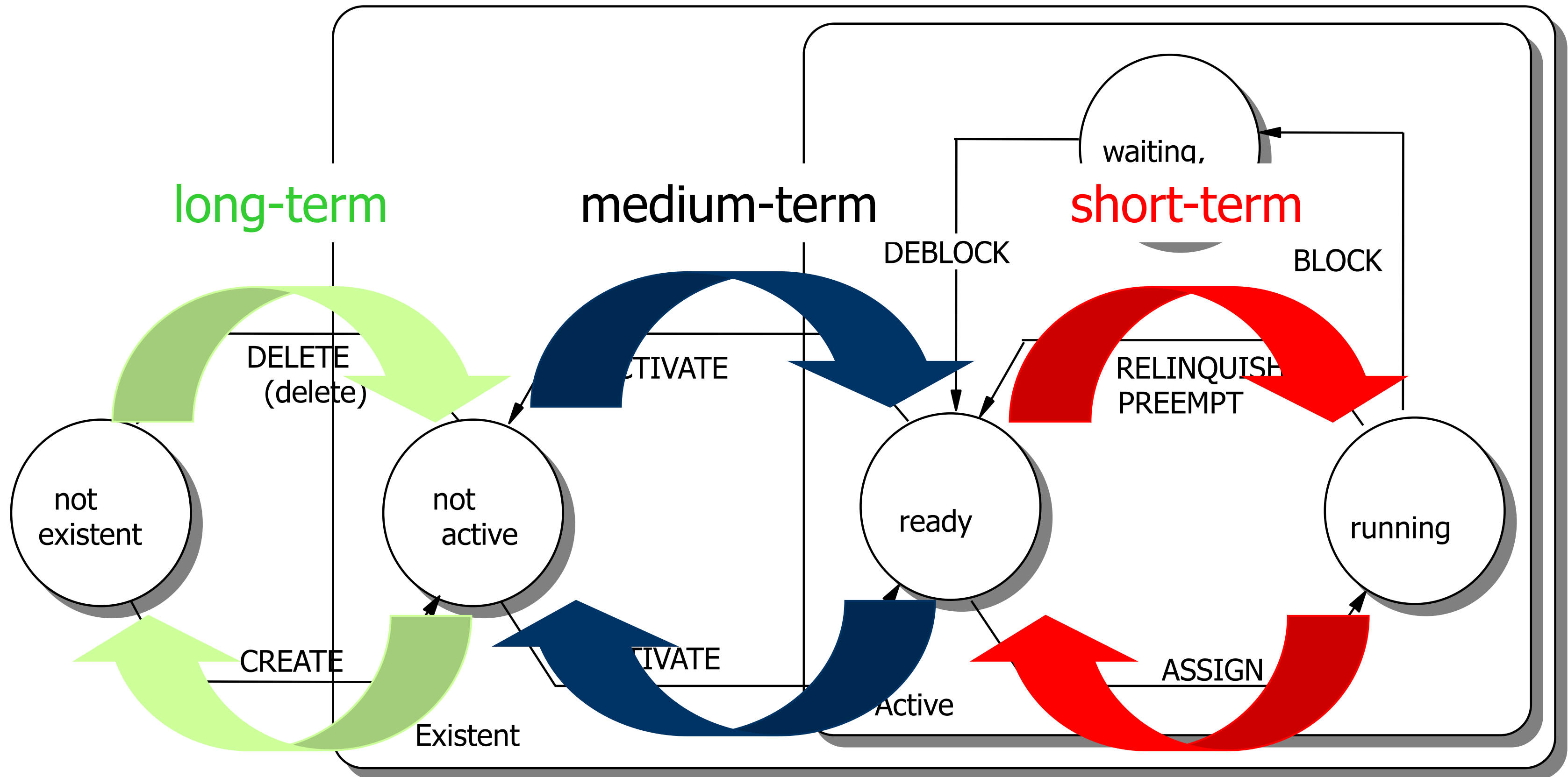
Freie Universität Berlin

Operation goals

- high efficiency — processor highly utilized
- low response time — with interactive programs
- high throughput — batch-processing
- fairness and capacity — fair distribution of processor and waiting times to the threads



Scheduling Levels



In the following, we focus on short-term scheduling.

Scheduling for Multiprogramming

Assumptions

- Homogeneous (symmetric) multiprocessor system
- Dynamic set of threads
- No dependencies between threads
- Dynamic on-line scheduling
- No deadlines

Strategic alternatives

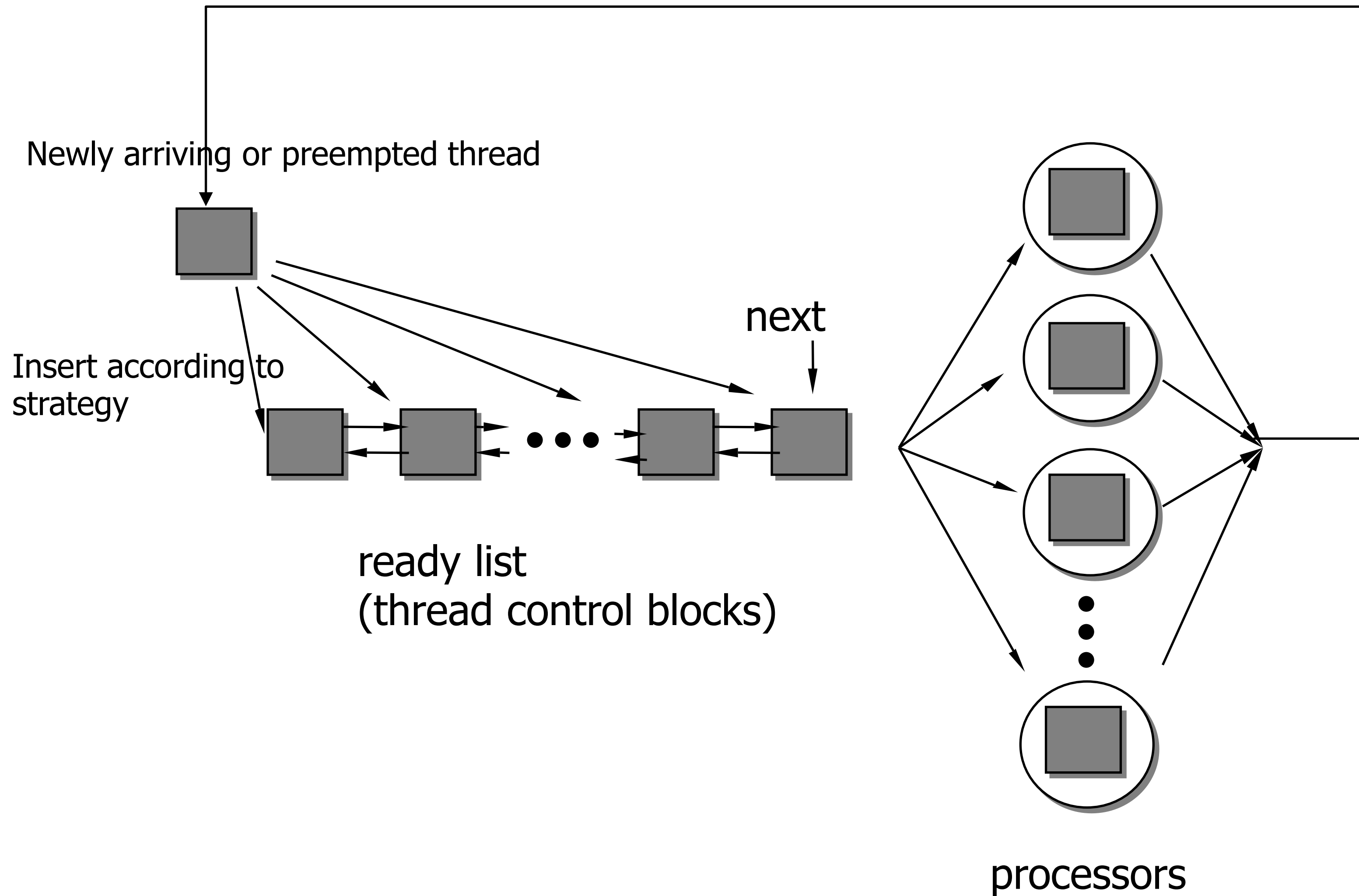
- with / without *preemption*

  thread stays on processor until it is finished (or gives it up voluntarily) or can be preempted before.
- with / without *priorities*

  threads are organized according to urgency.
- dependent / independent of *service time*

  The actual or estimated service time will be taken into account for assignment decisions.

Scenario for dispatching strategies



4.2.2 Standard strategies

FCFS	First Come First Served
LCFS	Last Come First Served
LCFS-PR	Last Come First Served-Preemptive Resume
RR	Round Robin
PRIO-NP	Priorities (non-preemptive)
PRIO-P	Priorities (preemptive)
SPN	Shortest Process Next
SRTN	Shortest Remaining Time Next
HRRN	Highest Response Ratio Next
FB	Multilevel Feedback

Running Example

Given the following five threads:

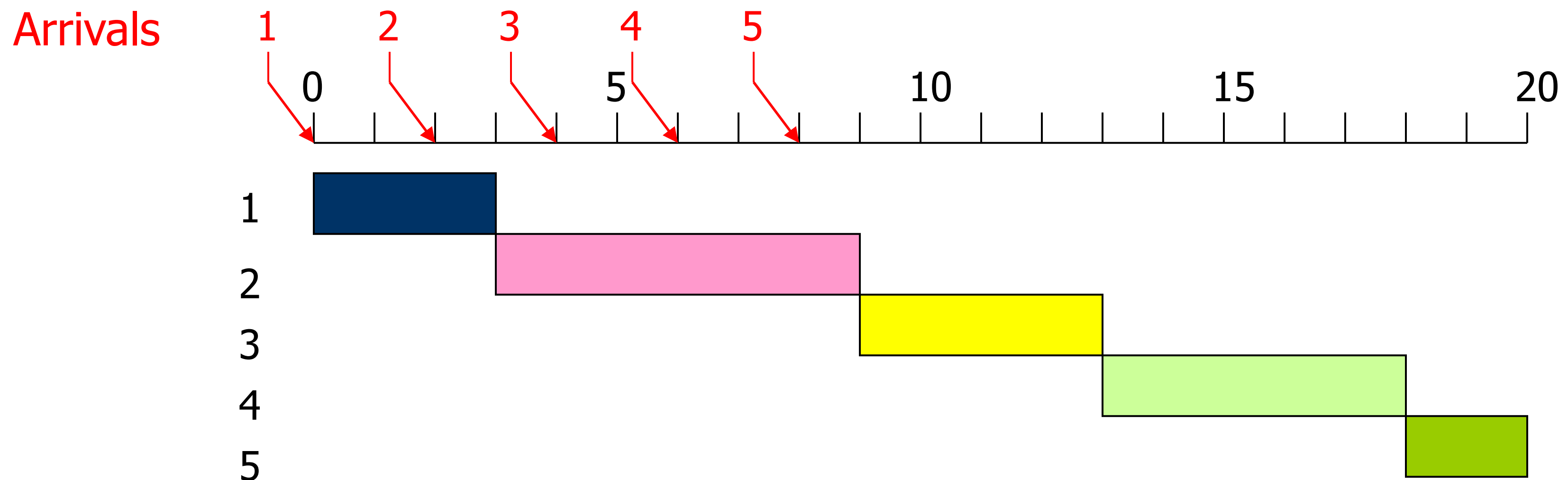
No.	Arrival	Service time	Priority
1	0	3	2
2	2	6	4
3	4	4	1
4	6	5	5
5	8	2	3

FCFS - First Come First Served

(aka FIFO (First In First Out))

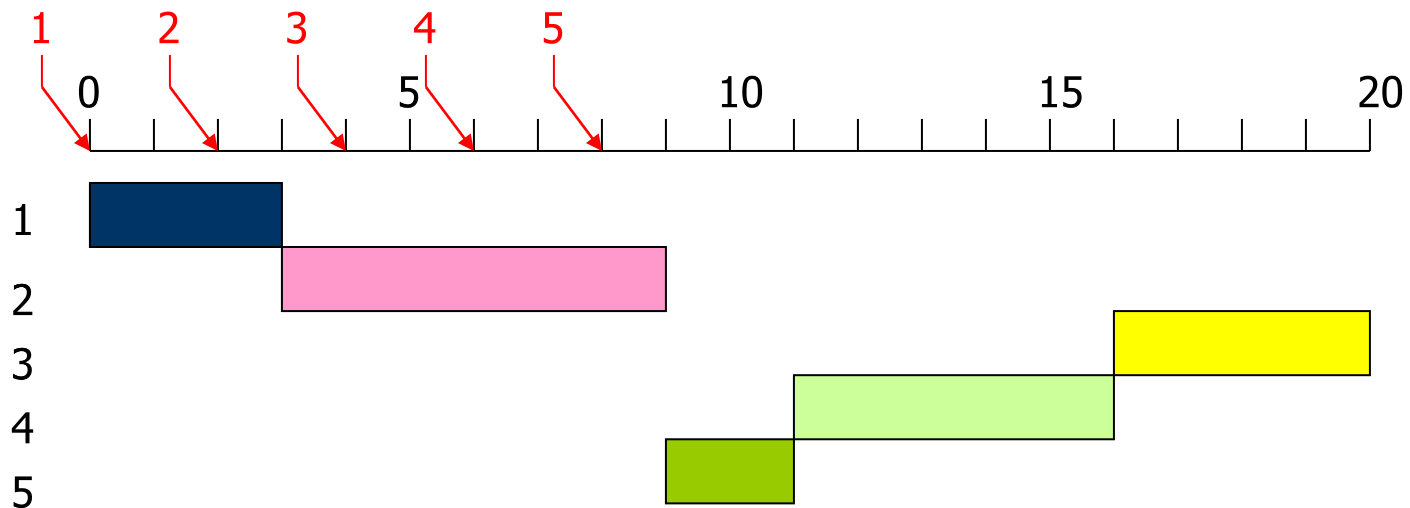
- How it works:
  - Execution of threads in the order of arrival at the ready list
  - Occupation of processor until end of voluntary yield

Remark: as in daily life (checkout counter at super market).



LCFS - Last Come First Served

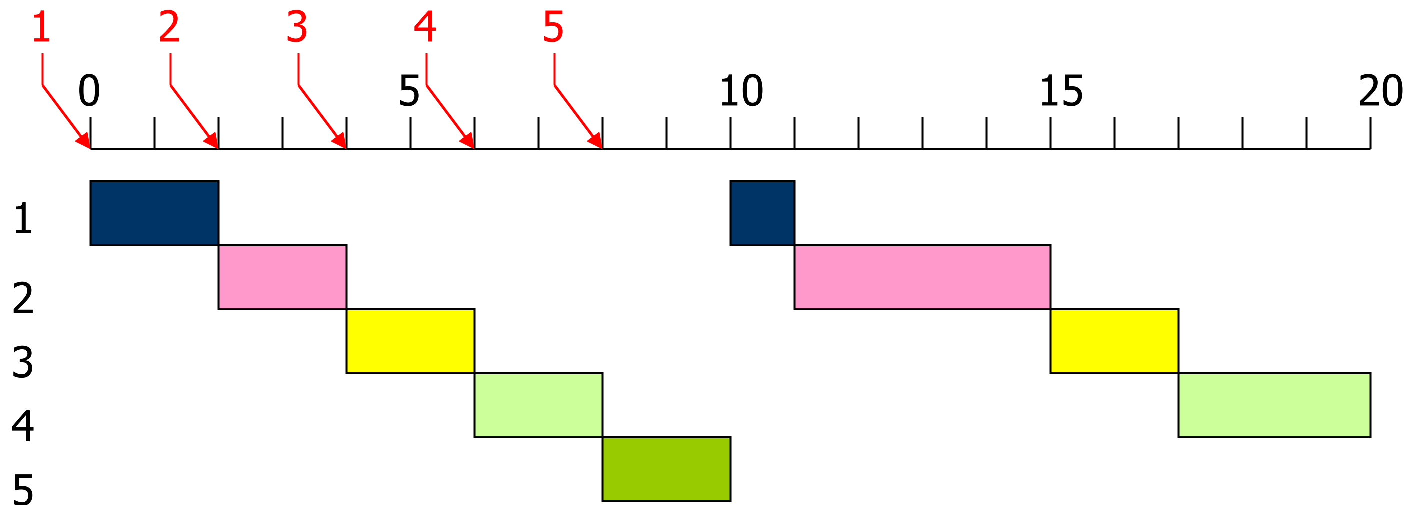
- How it works:
  - Execution of threads in reversed order of arrival at the ready list.
  - Occupation of processor until end or voluntary yield.
  - Remark: Rarely used in the pure form



LCFS-PR - Last Come First Served - Preemptive Resume

Freie Universität Berlin

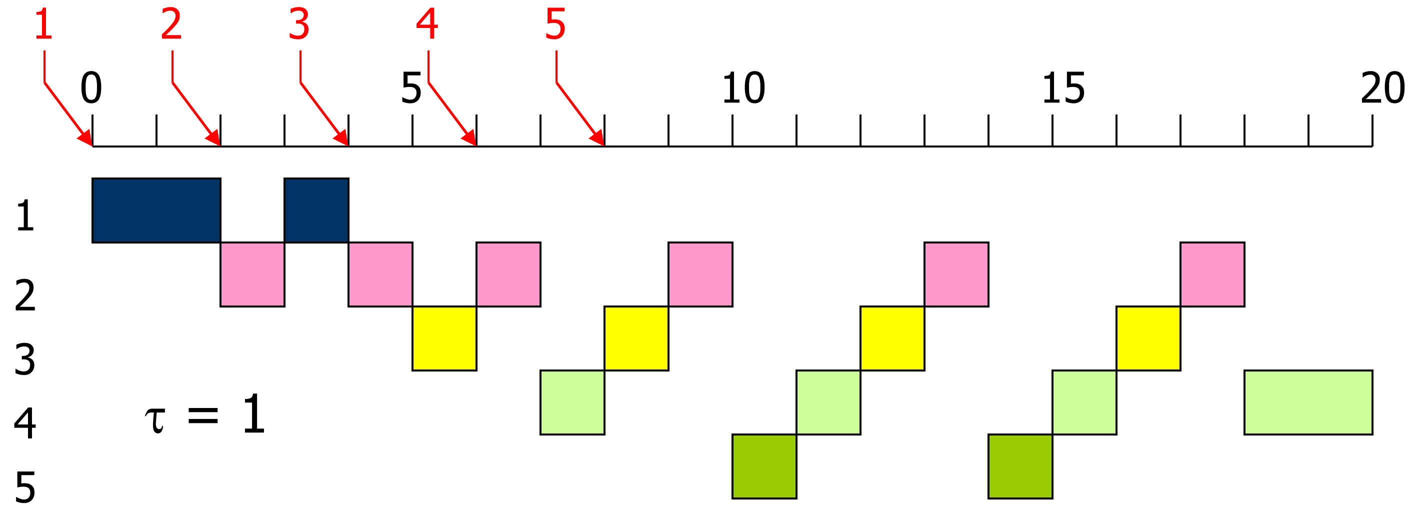
- How it works:
  - Newly arriving thread at ready list preempts the currently running thread.
  - Preempted thread is appended to ready list.
  - In case of no further arrivals, the ready list is processed without preemption.
- Remark:
  - Goal: Preference to short threads.
  - A short thread has a good chance to finish before another thread arrives.
  - A long thread is likely to be prempted several times.

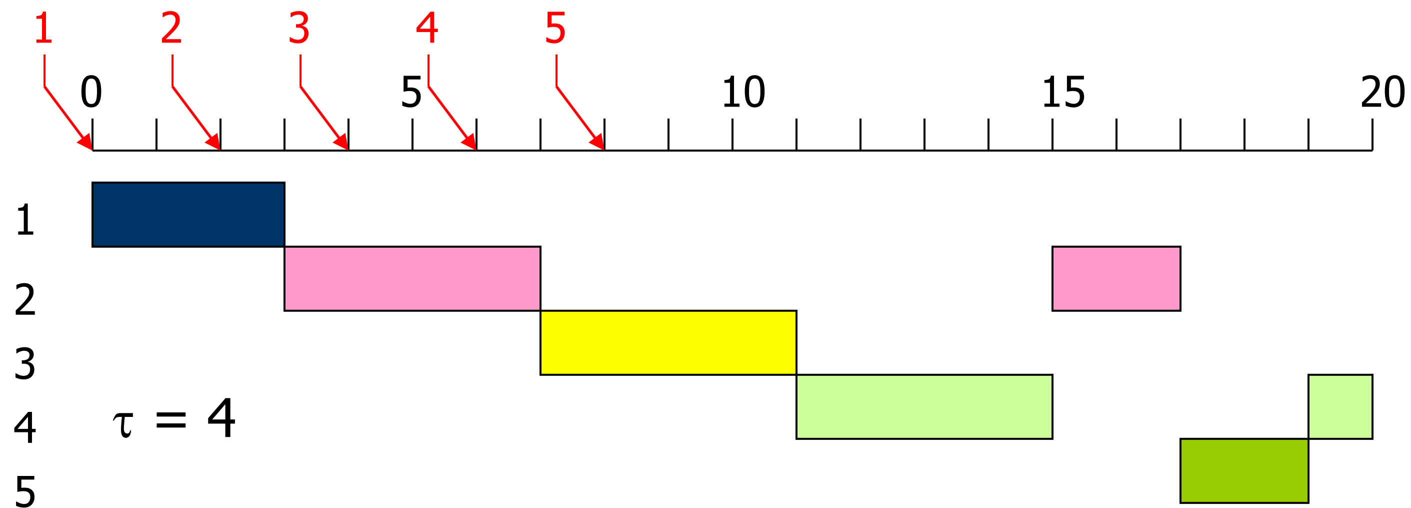


RR - Round Robin (time slicing)

- How it works:
    - Processing of threads in order of arrival
    - After some prespecified time (time slice, CPU-quantum) preemption takes place and we switch to the next thread.
- Remark:
    - Goal is the even distribution of processor capacity and waiting time to the competing threads.
    - Selection of time slice length τ is optimization problem:
        - For large τ RR approaches FCFS.
        - For small τ the overhead for frequent switching is a performance penalty.
    - Usual are time slices in the order of some tens of milliseconds

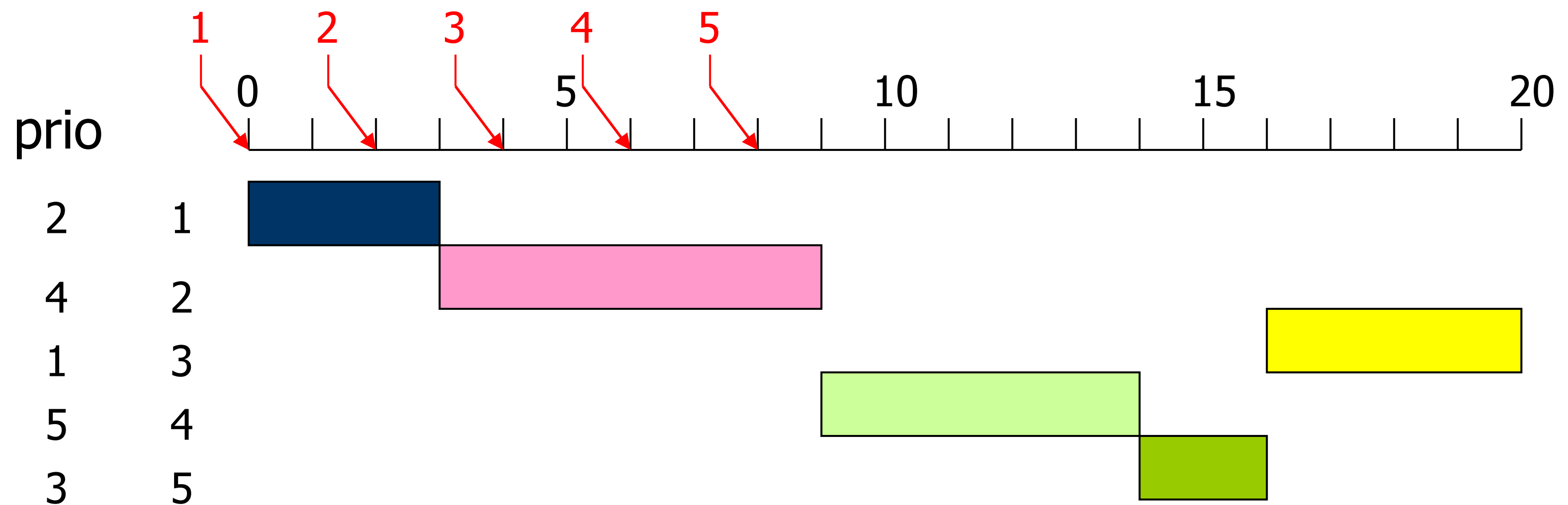
RR - Round Robin





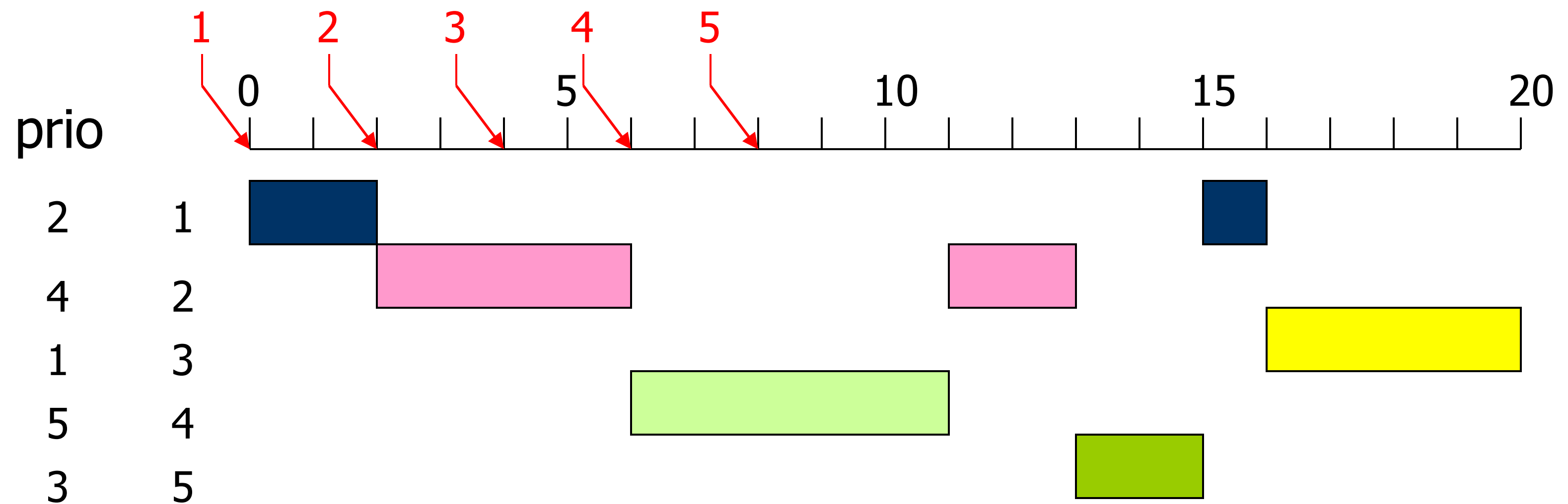
PRIO-NP - Priorities - nonpreemptive

- How it works:
  - Newly arriving threads are inserted in the ready list according their priority
  - Once assigned they stay running on the processor until they finish or voluntarily relinquish the processor.



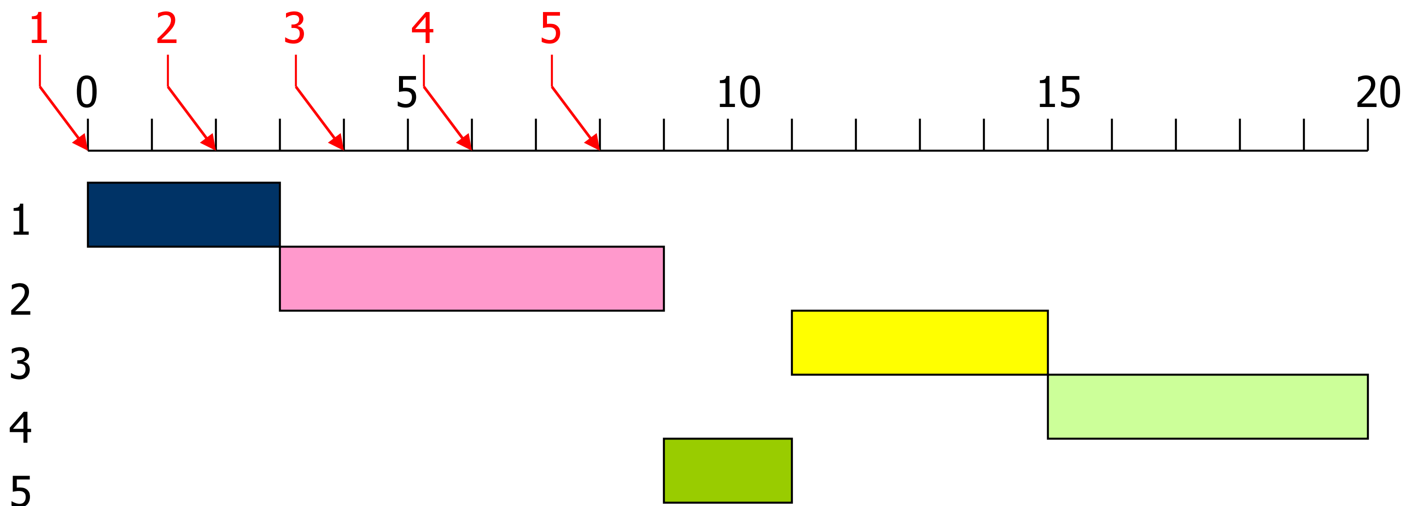
PRIO-P - Priorities - preemptive

- How it works:
  - Like PRIO-NP except that we check for preemption, i.e. the running thread is being preempted if it has a lower priority than the new thread.



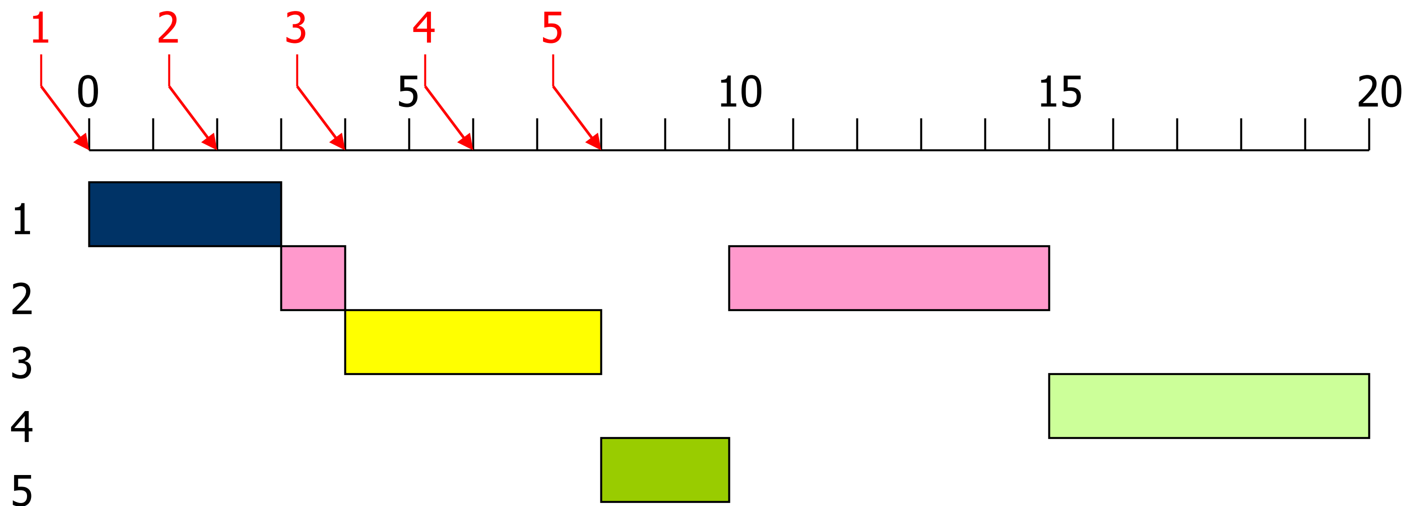
SPN - Shortest Process Next

- How it works:
  - Thread with shortest service time is executed next until it finishes.
  - Like PRIO-NP, if we consider service time as priority criterion.
- Remark:
  - Favors short threads and thus leads to shorter mean response times than FCFS.
  - Also knows as Shortest Job Next (SJN)



SRTN - Shortest Remaining Time Next

- How it works:
  - Thread with shortest remaining service time is executed next.
  - Currently running thread may be preempted.
- Remark:
  - Both strategies have the disadvantage that they need a-priori knowledge of service times that may be available only as user's estimates.
  - Long threads may starve when always shorter threads are available.



HRRN - Highest Response Ratio Next

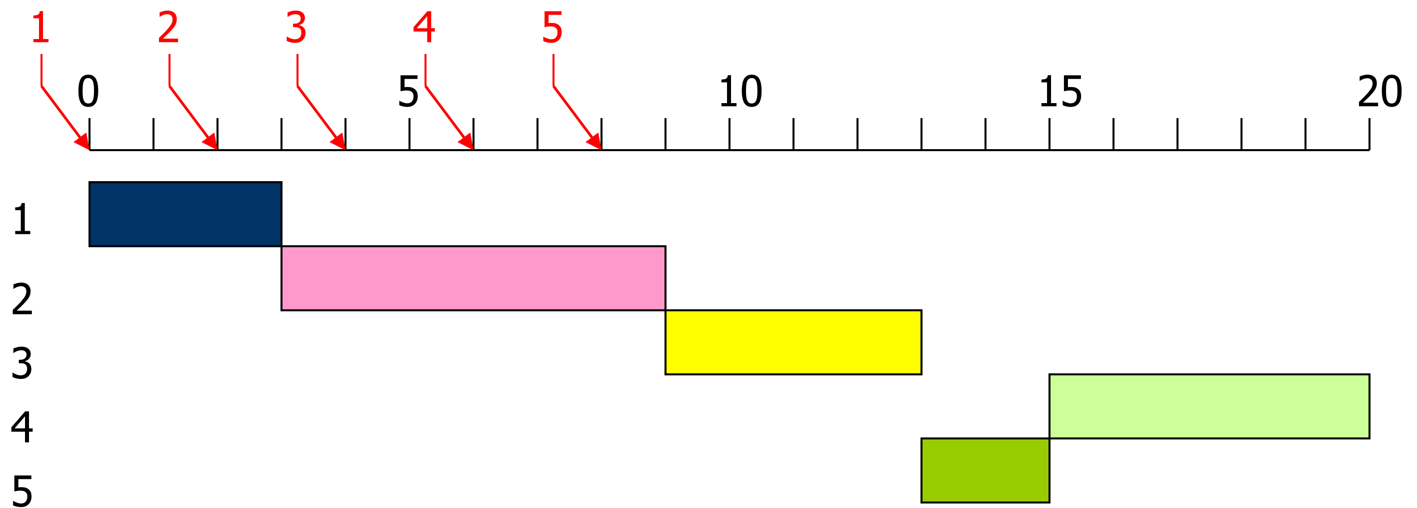
- How it works
  - The response ratio *rr* is defined as

$$rr := \frac{\text{waiting time} + \text{service time}}{\text{service time}}$$

  - *rr* is calculated dynamically and used as priority:
    The thread with the highest *rr* –value is selected.
  - The strategy is non-preemptive.

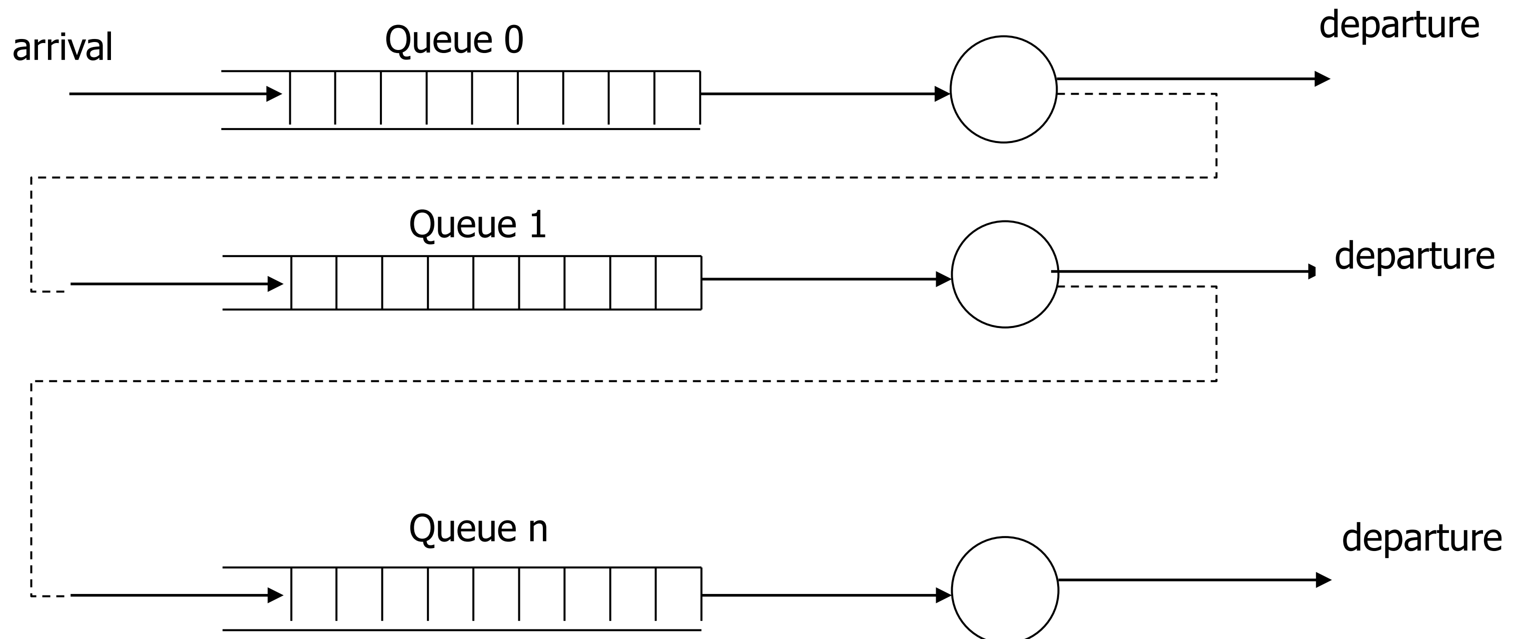
- Remark:
  - As with SPN short threads are favored. However, long threads do not need to wait forever but score some points by waiting.

HRRN - Highest Response Ratio Next



FB - (Multilevel) Feedback

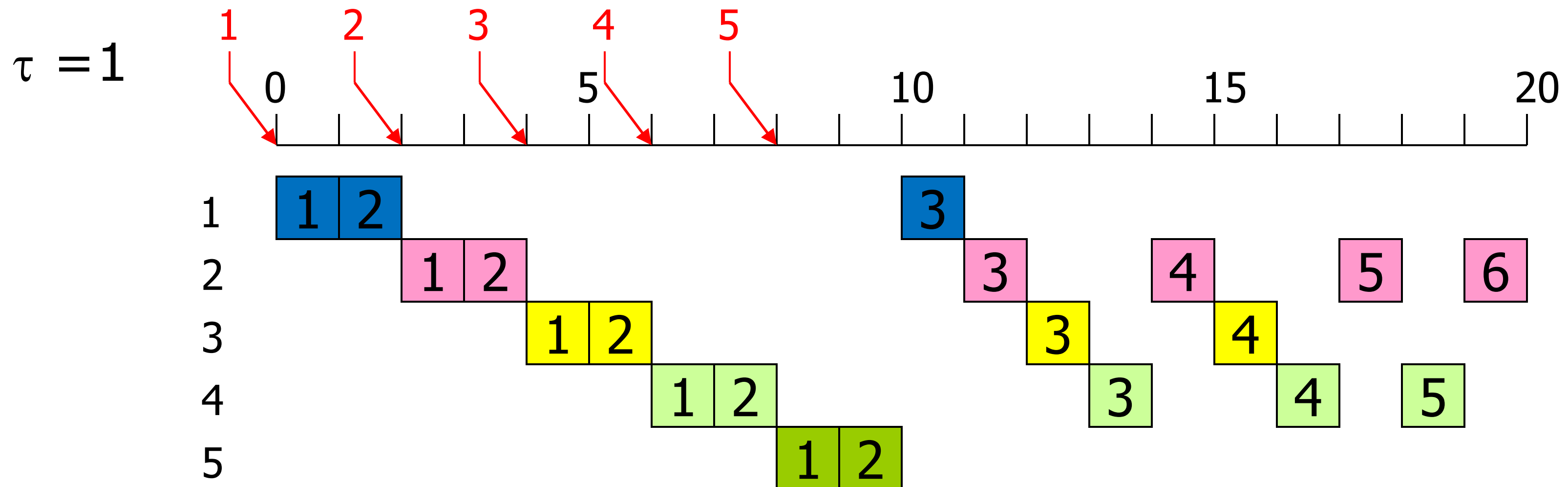
- If we do not know the service time a priori, but want to favor short threads we can reduce its "priority" stepwise at each CPU-usage.
- The individual waiting queues can be managed according to "round robin".
- Different values of time slices τ for the individual queues are possible.



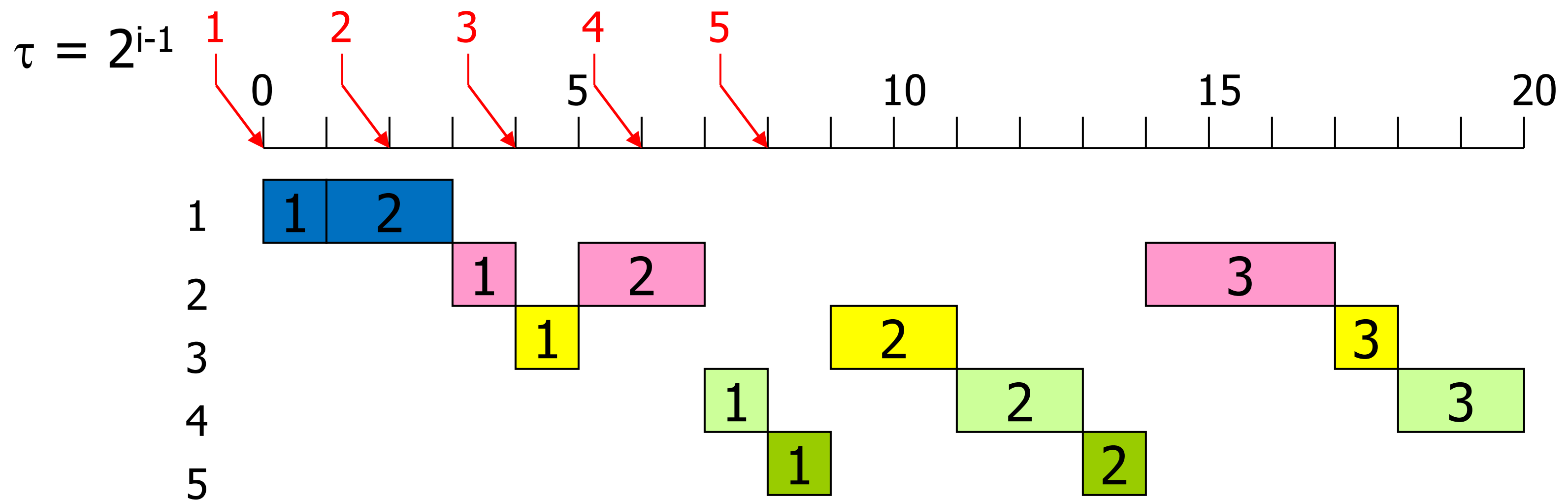
FB - (Multilevel) Feedback

Freie Universität Berlin

$\tau = 1$



$\tau = 2^{i-1}$



Standard strategies (summary)

Freie Universität Berlin

	without preemption		with preemption	
	without priorities	with priorities	without priorities	with priorities
Service time independent	**FCFS, LCFS**	**PRIO-NP**	**LCFS-PR, RR, FB**	**PRIO-P**
Service time dependent	**SPN, HRRN**		**SRTN**	

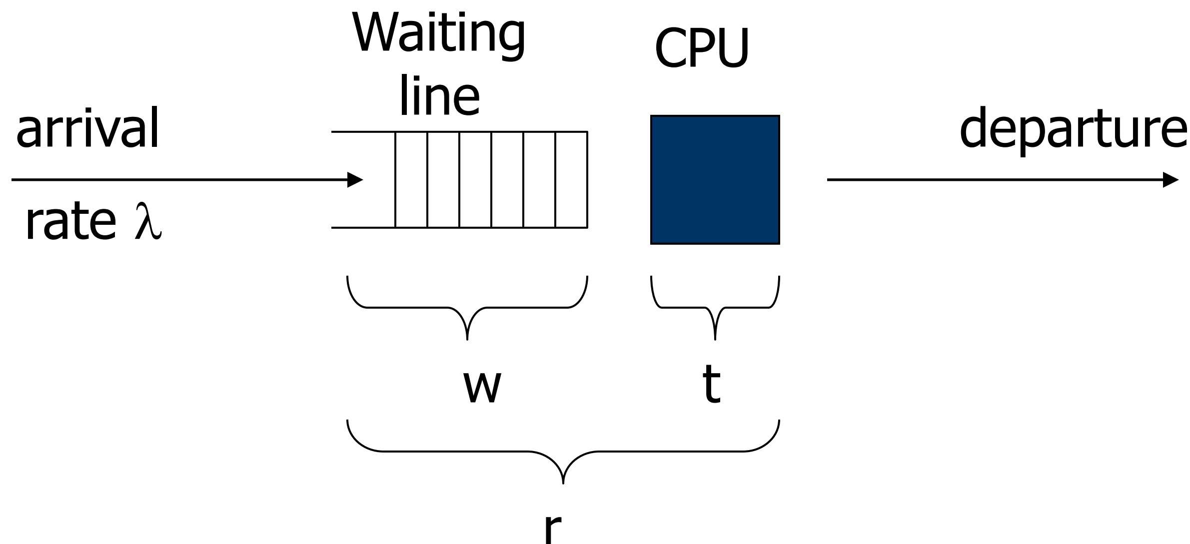
Scheduling with priorities

- **Static priorities** express the absolute importance of tasks.
  - A task is never executed as long as there is a task with a higher priority.
  - Subject to priority inversion.
  - Easy to implement, low overhead, usually O(1) complexity.

- **Dynamic priorities** are a bit less strict.
  - While the scheduler also always selects one of the tasks with the highest priority, it also adapts this priority over time.
  - Depending on how priorities are adjusted, certain effects can be achieved.
    - Priority inversion (see slide 4-32) can be avoided, if it is guaranteed that every task will execute eventually.
    - Short running tasks can get preference over long running tasks.
  - Can be realized on top of an implementation of static priorities.

Modelling

- λ arrival rate (# arrivals per time)
- t mean service time (pure execution time)
- w mean waiting time
- r mean response time
  (including waiting times): $r = w+t$
- r_n normalized response time: $r_n = r/t$
- ρ utilization ($0 \leq \rho < 1$)



Priorities

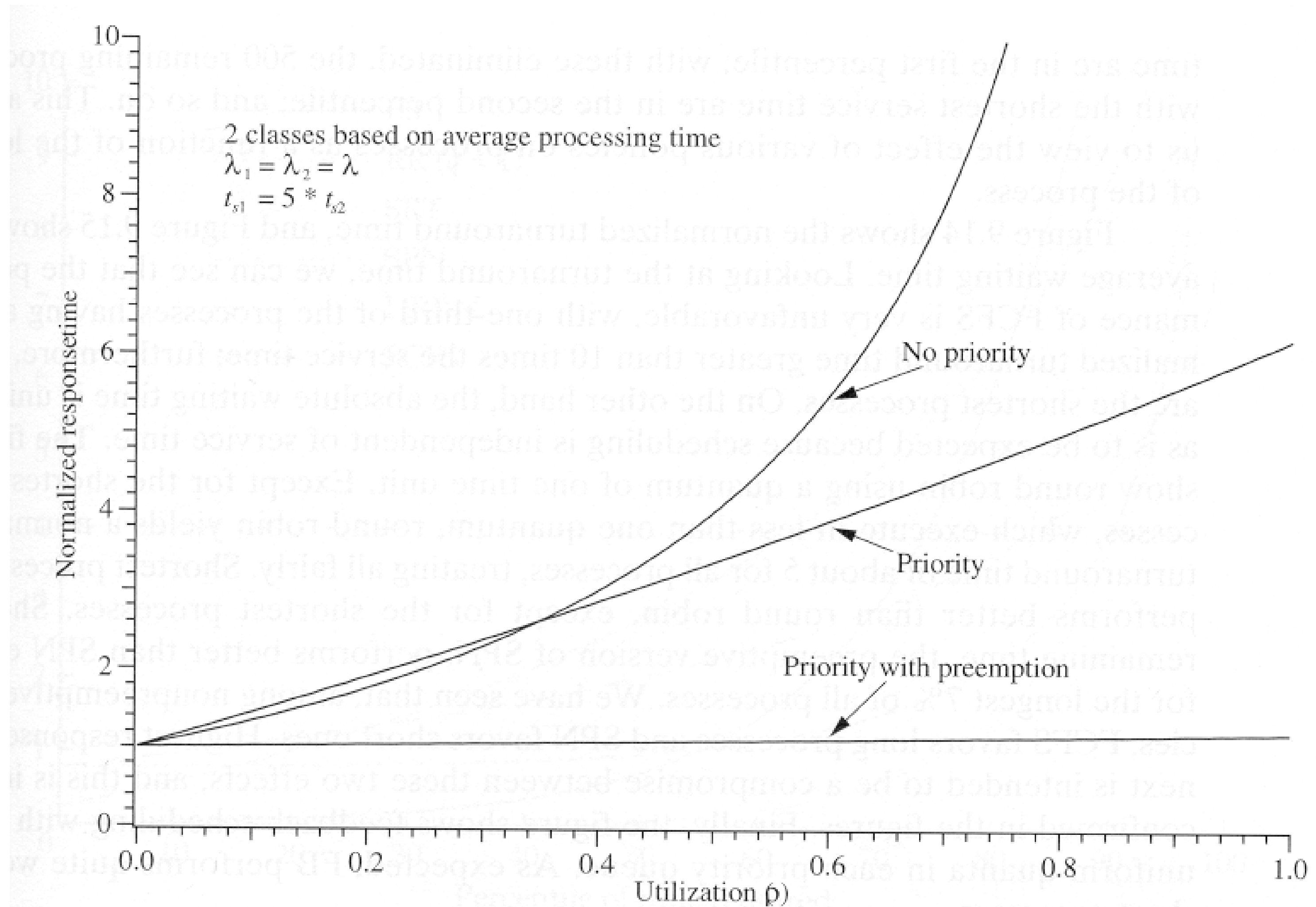


Figure 9.12 Normalized Response Times for Shorter Processes

Source:
Stallings, Chap 9

Priorities

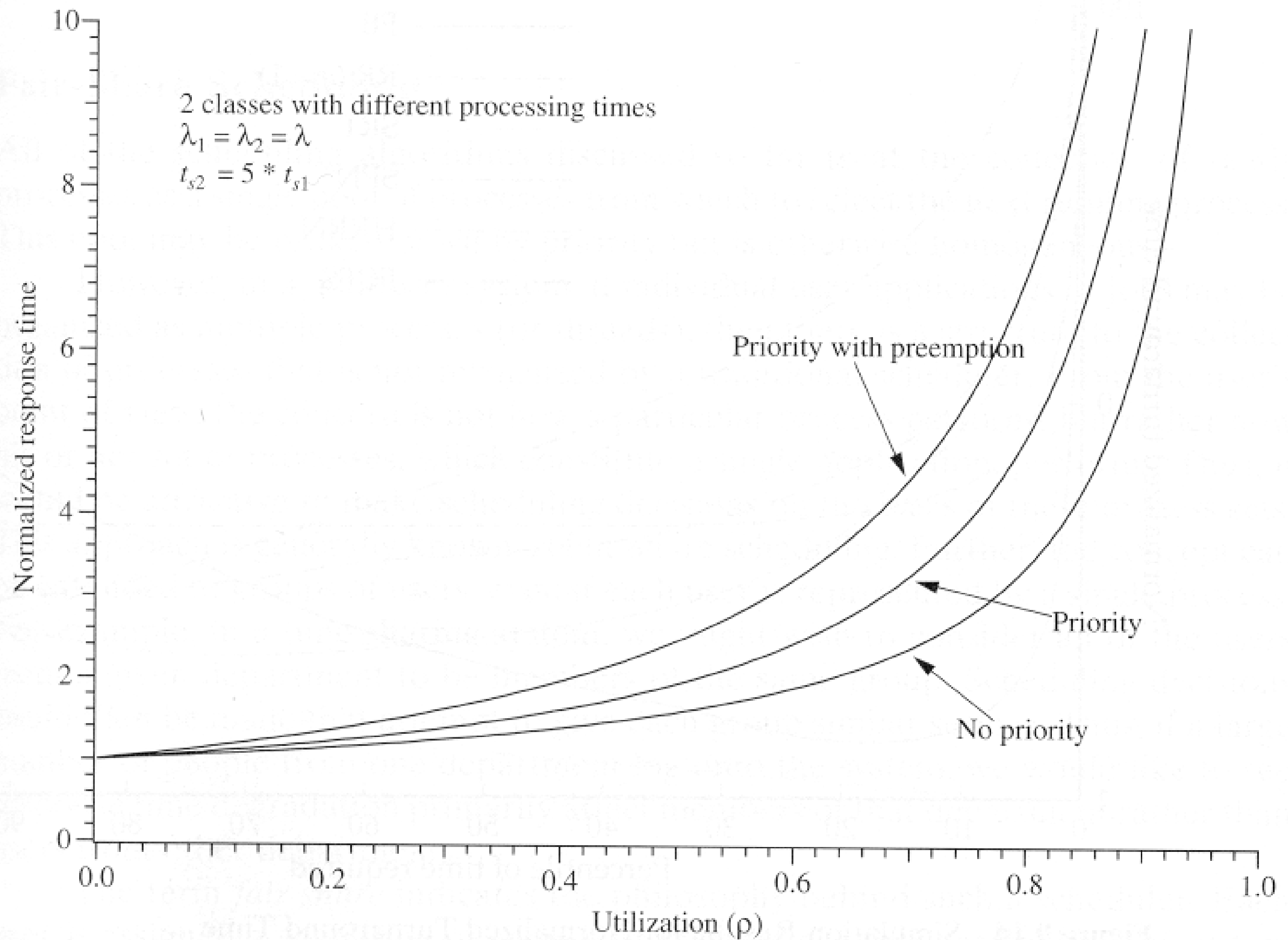
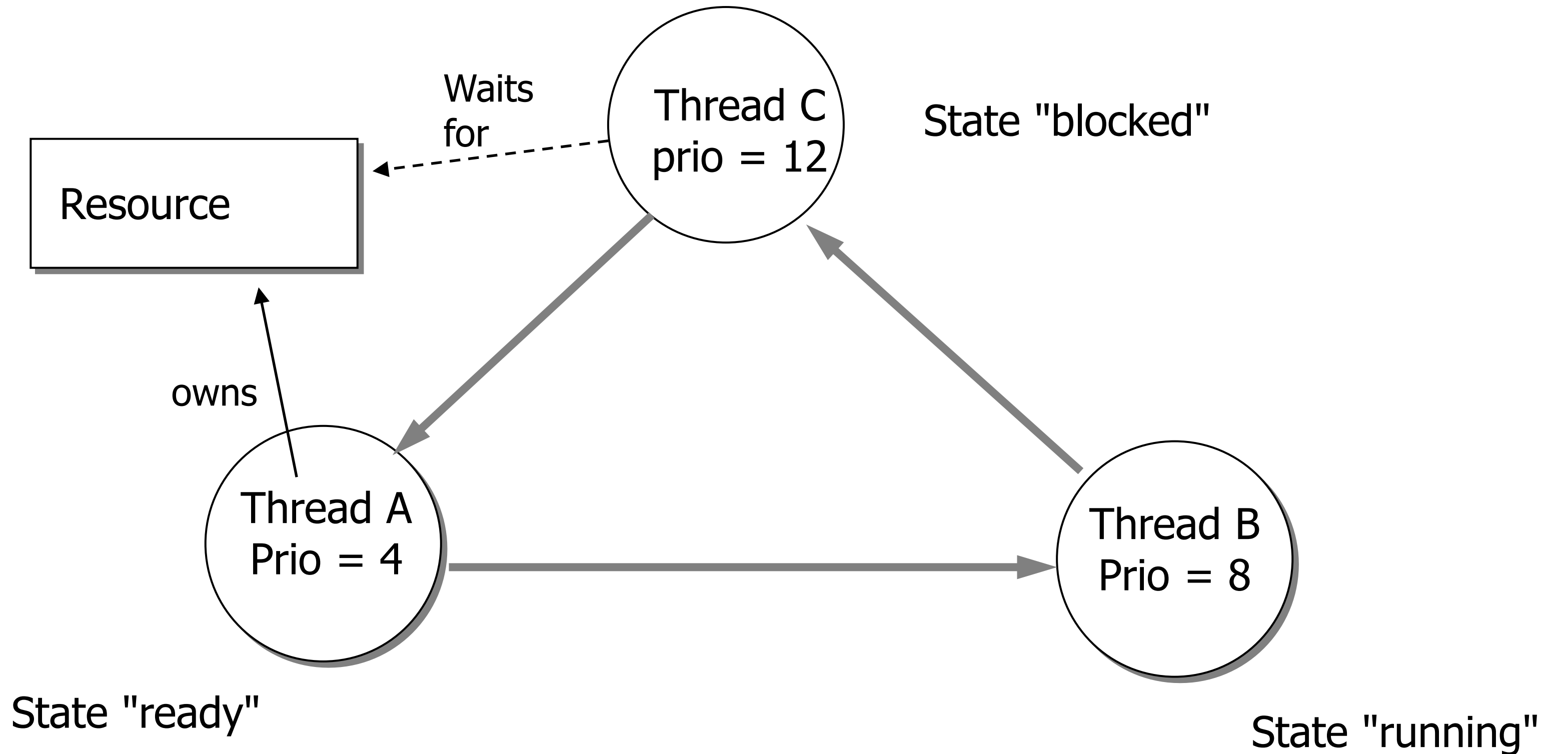


Figure 9.13 Normalized Response Times for Longer Processes

Source:
Stallings, Chap 9

Priority inversion

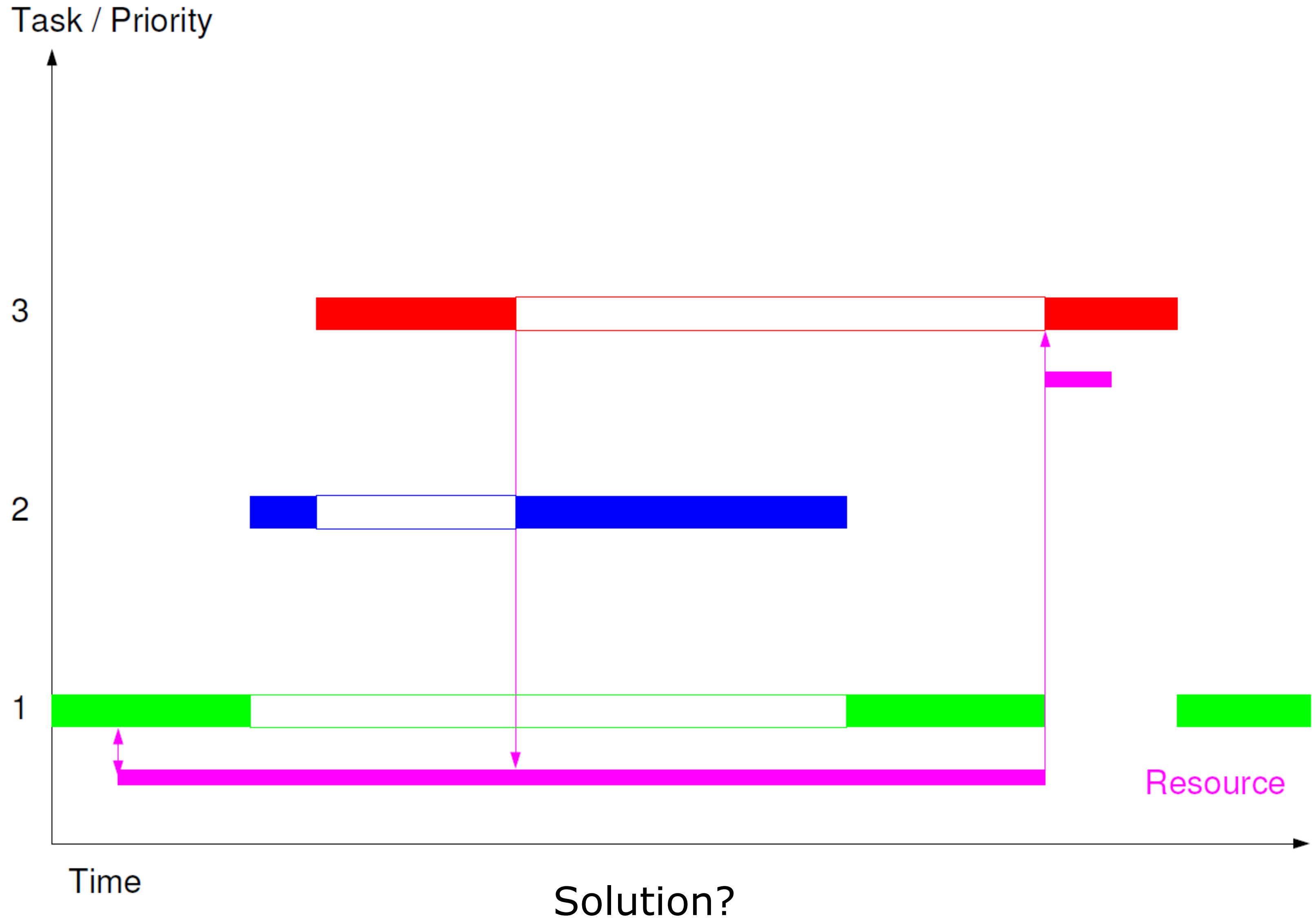


Thread C will "starve", although it has highest priority:

Thread B dominates the CPU, thread A cannot continue and therefore cannot release the resource, with the effect that thread C stays blocked for an unspecified time.

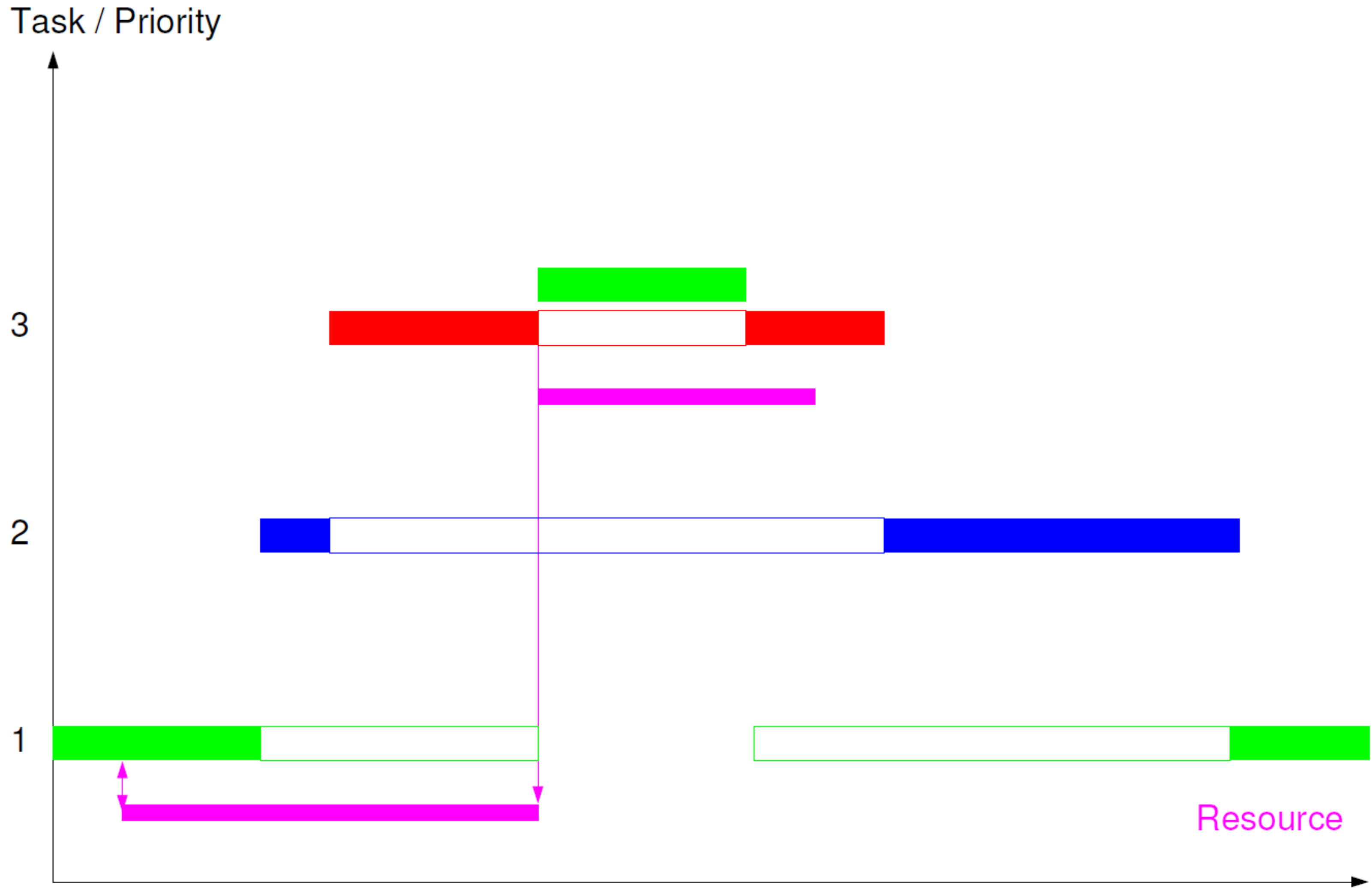
Solution?

Priority inversion



Solution?

Priority inheritance



Solution: Thread 1 “inherits” the priority of 3

Proportional Share Scheduling

- Each task is assigned a weight (or share) w_i.

- The allocated CPU time of a task is proportional to its share.
  - $t_i / \sum t_j = w_i / \sum w_j$

- That is, within a time frame T a task gets (with runqueue weight $W = \sum w_j$):
  - $t_i = T * w_i/W$

- This concept can also be applied to groups of tasks.
- Proportional share scheduling expresses the relative importance of tasks.
  - Tasks with a small share get CPU time, but not as much as tasks with larger shares.
  - Avoids priority inversion.
  - More complex than a strategy based on priorities.

4.2.3 Comparative evaluation

Behavior of scheduling strategies (Simulation of 50,000 threads)

The mean value of the shortest 500 threads is in the first percentile.

(NTT = response time / service time)

Source: Stallings, Chap 9

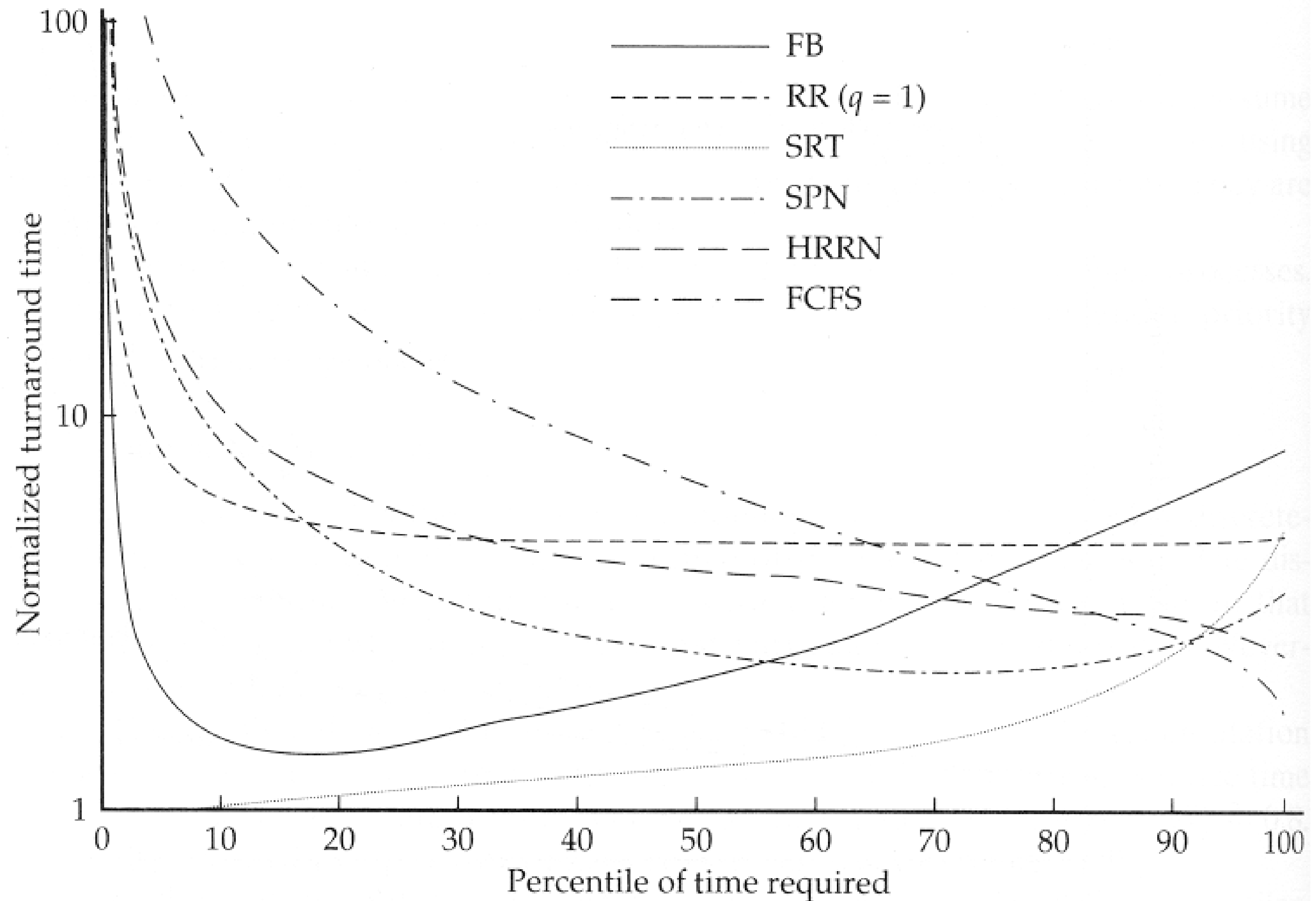
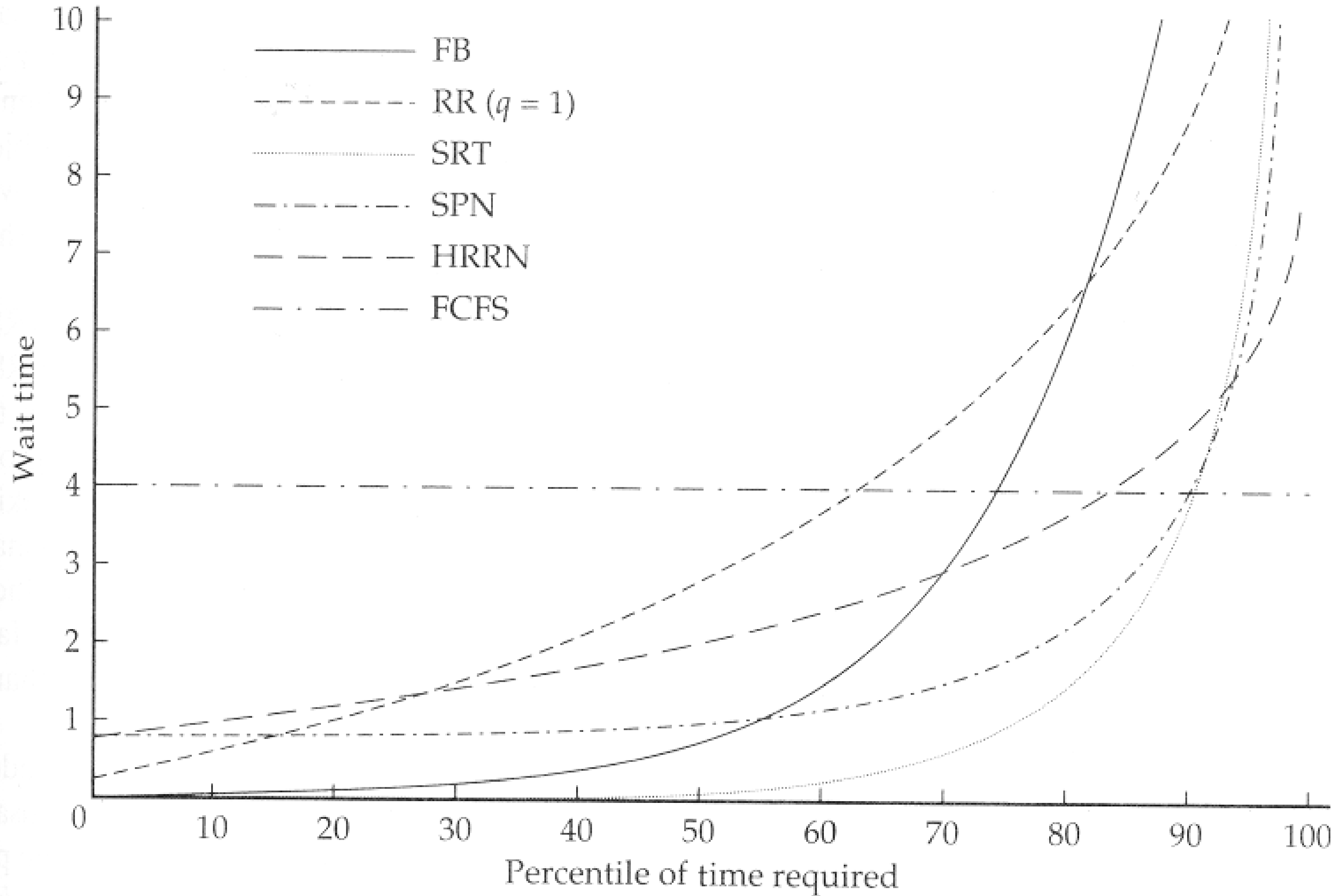


Figure 9.14 Simulation Results for Normalized Turnaround Time

Waiting time

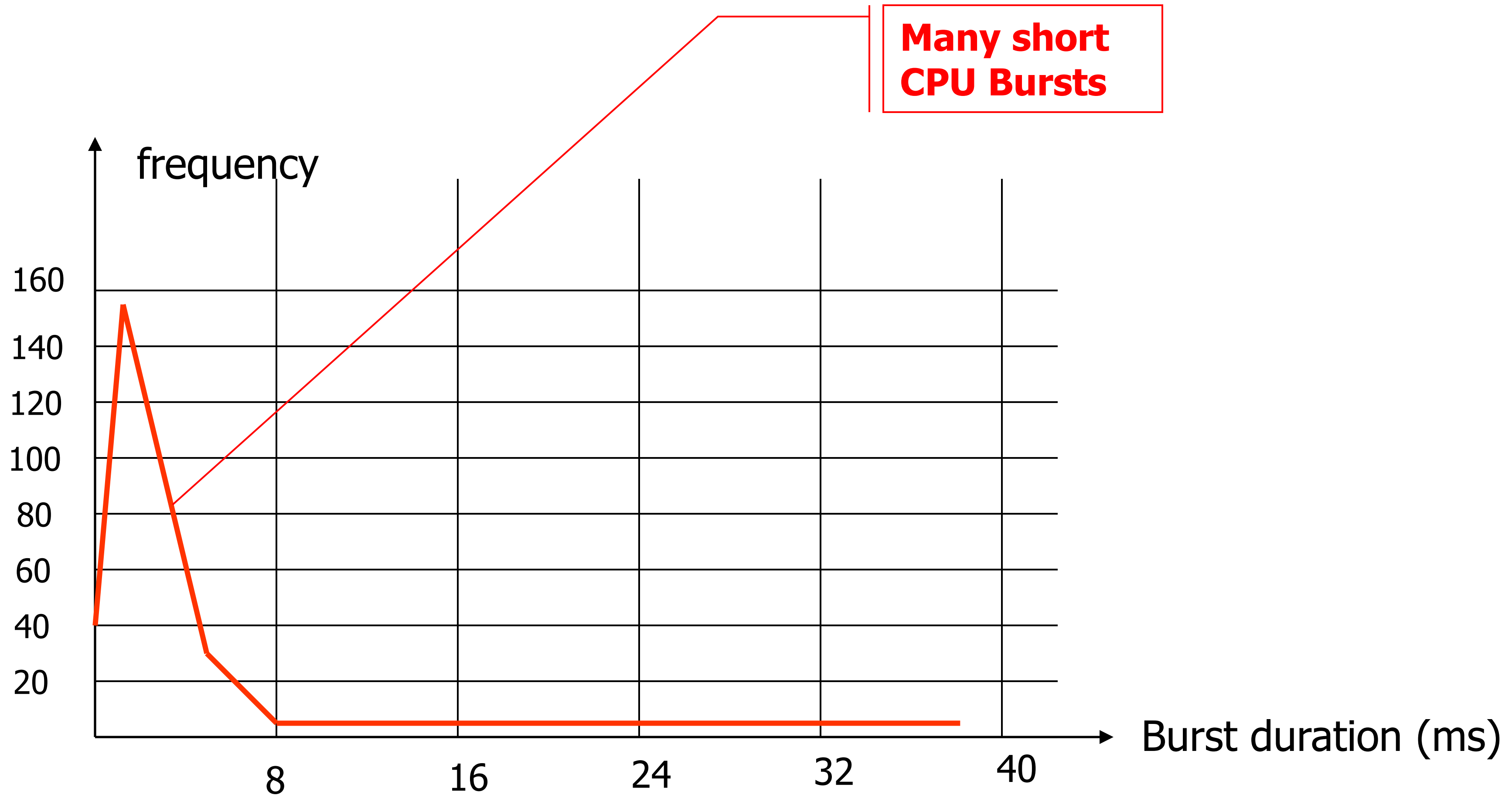


Source:
Stallings, Chap 9

4.2.4 Estimating the duration of computing phases

- For the scheduling strategies service time not necessarily means total service time, it can also relate to the compute time between two I/O-activities.
- This is sometimes called "CPU burst".
- The compute phases often follow other statistical distributions than the total service times.
- Threads often show some "stationary" behavior, i.e. the behavior of the most recent past is a good predictor for the short-term future.

Histogram of compute phases (CPU bursts)



Estimation of CPU Burst

- Let be T[i] the execution time of the ith compute phase of a thread.
- Let S[i] be the estimate for the ith compute phase of that thread i.e. for T[i] .
- In the most simple case we use:

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

- To avoid recalculation of the whole sum, we rewrite the formula as:

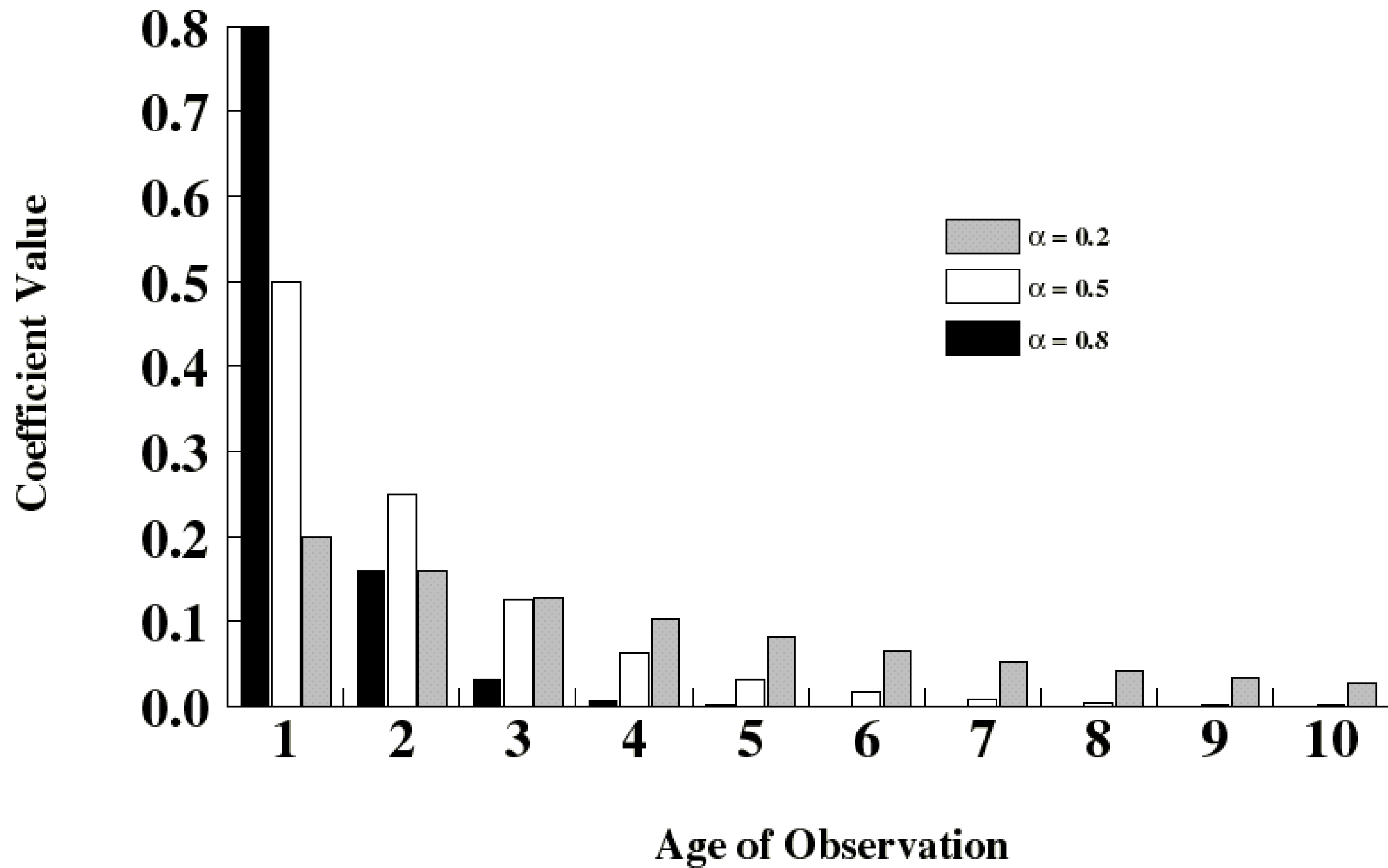
$$S[n+1] = (1/n)\ T[n] + ((n-1)/n)\ S[n]$$

- This convex combination gives equal weight to all summands.

Estimation of CPU Burst

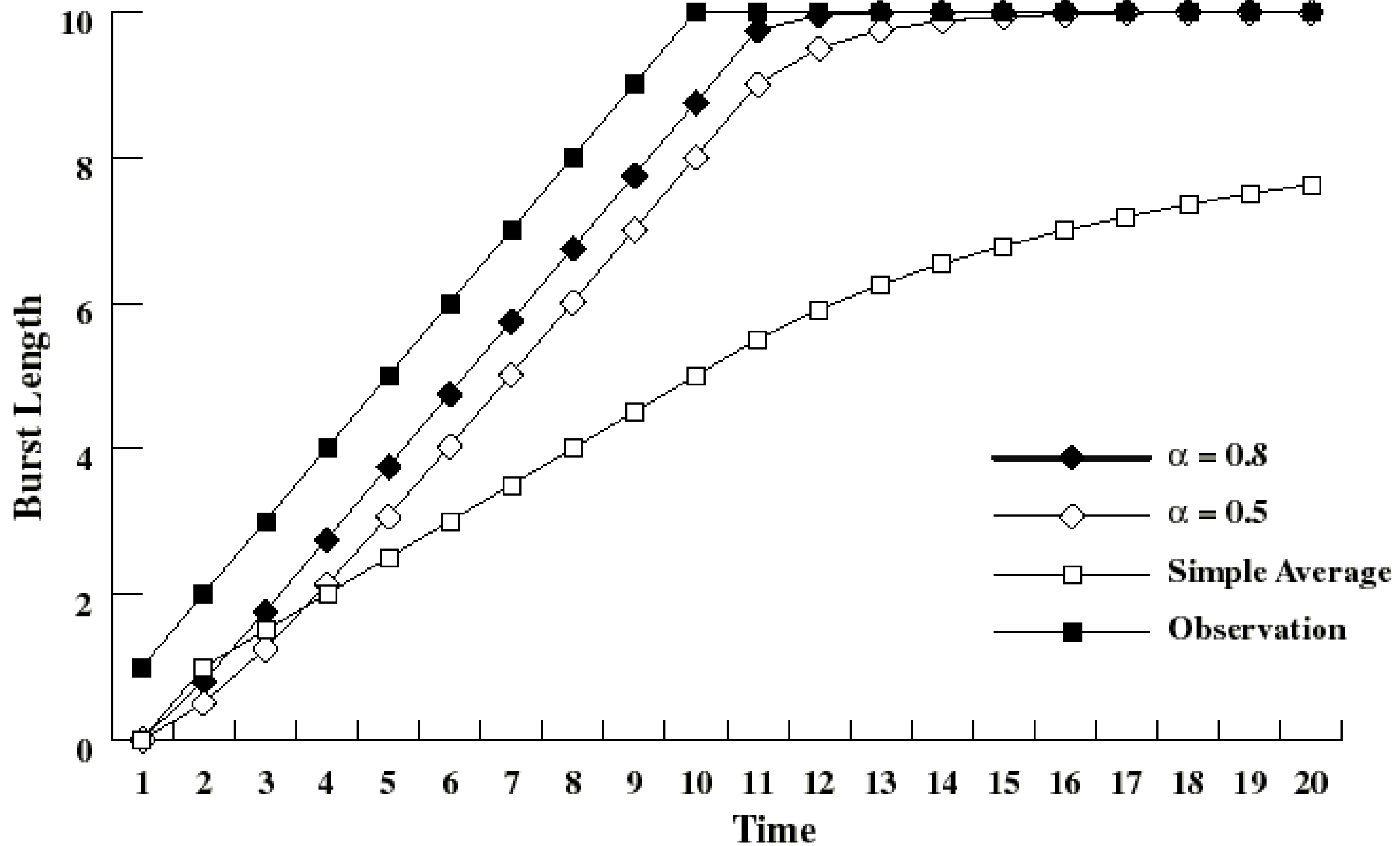
- Younger (=more recent) values usually characterize the future behavior better than older ones and should obtain higher weight.
- The usual technique is the **exponential averaging** (exponential smoothing).
  - $S[n+1] = \alpha\, T[n] + (1-\alpha)\, S[n];\ 0 < \alpha < 1$
  - If $\alpha > 1/n$, younger values get higher weights.
- Enrolling the recursion shows that the weights of past measurements fade out exponentially.
- $S[n+1] = \alpha T[n] + (1-\alpha)\alpha T[n-1] + \ldots (1-\alpha)^i \alpha T[n-i] + \ldots + (1-\alpha)^n S[1]$
- High α leads to better **responsiveness** (response to change in behavior), small α to better **stability** (filtering of outliers).
- The first estimate S[1] can be initialized with 0 to give new threads high priority.

Exponential decay of weights



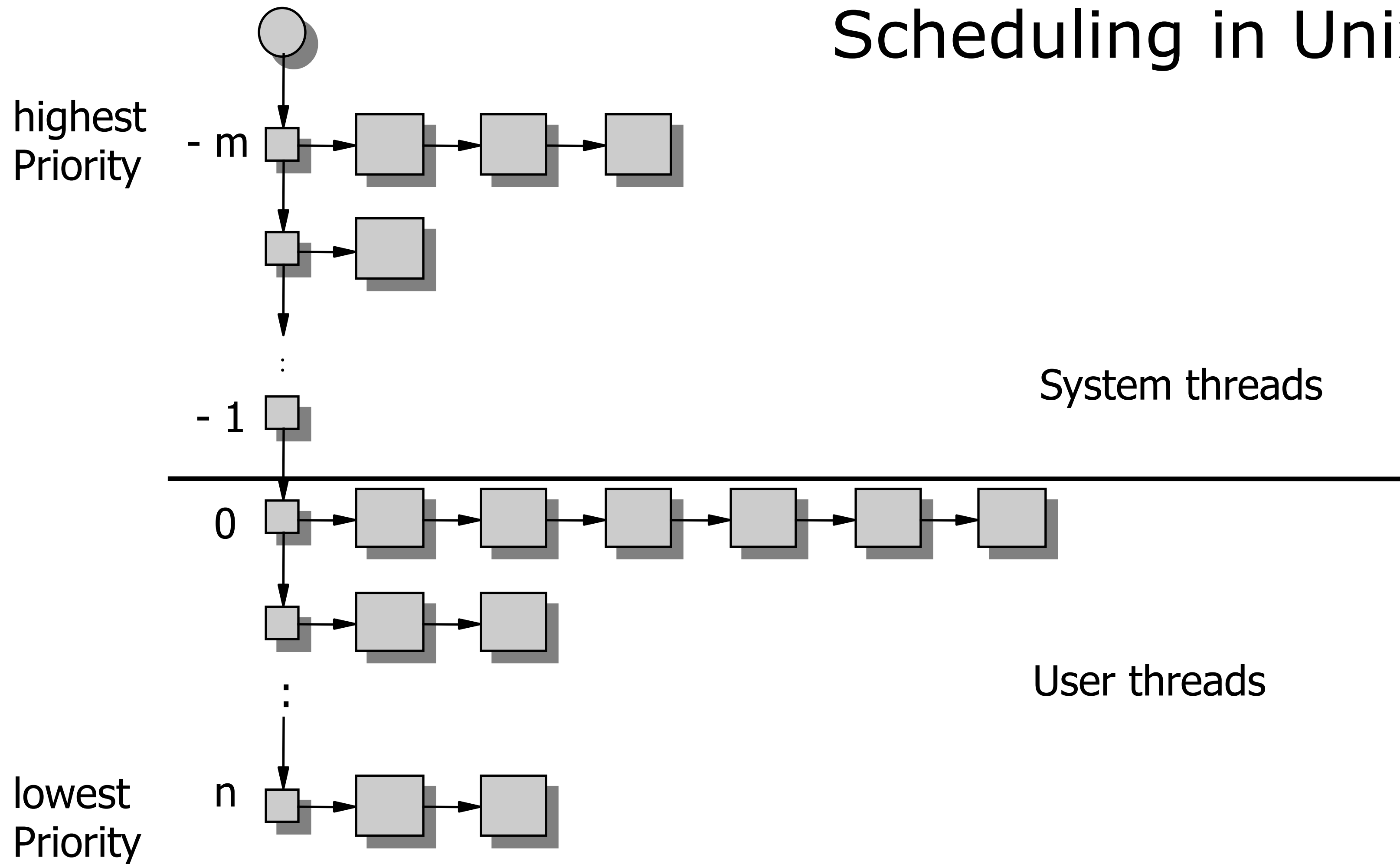
Source:
Stallings, Chap 9

Applying exponential smoothing



Source:
Stallings, Chap 9

Scheduling in Unix



Round-Robin within each priority class

Scheduling in Unix (System V)

- Time slice length in the order of 100 msec.
- User threads have a static base priority of 0 that can be increased (i. e. worsened) by "nice" (Unix command).
- Dynamic priority for thread *j* is adapted every second *i* according to the formula

$$dyn_prio_j[i] := base_prio_j + cpu_j[i]/2$$

- The processor usage cpu_j [i] is measured in clock ticks.
- Before calculating the priority we apply

$$cpu_j[i] := cpu_j[i-1]/2$$

  which means we incrementally "forget" the processor usage of the past (fading memory).

Example Unix

Low numbers mean high priority!

Time	Process A Priority	Process A CPU Count	Process B Priority	Process B CPU Count	Process C Priority	Process C CPU Count
0	60	0 1 2 • • 60	60	0	60	0
1	75	30	60	0 1 2 • • 60	60	0
2	67	15	75	30	60	0 1 2 • • 60
3	63	7 8 9 • • 67	67	15	75	30
4	76	33	63	7 8 9 • • 67	67	15
5	68	16	76	33	63	7

Shaded rectangle represents executing process.

Source:
Stallings, Chap 9

Scheduling in Unix (System V)

Heavy CPU-usage leads to low priority!

Why ?

- Compute intensive threads are put at disadvantage.
- I/O intensive threads are favored!
- They occupy the CPU only for a short time to submit the next I/O-request.
- By giving them high priority we keep the peripheral devices busy and achieve a high degree of parallelism between the CPU and I/O-devices.

Fair-Share-Scheduling

- In some Unix-systems the **Fair-Share-Scheduling** is used.
- It aims at guaranteeing a group of threads a fixed fraction of the processor capacity.
- Let *k* be the number of groups, W_k the capacity fraction that group k should obtain:

$$0 < W_k \leq 1, \quad \sum_k W_k = 1$$

- Let $cpu_j[i]$ be the CPU usage of thread *j* in group *k* and $gcpu_k[i]$ the CPU usage of all threads of the group *k* in interval *i*.
- Then we calculate the priorities as follows :

$cpu_j[i] := cpu_j[i-1]/2$

$gcpu_j[i] := gcpu_j[i-1]/2$

$dyn_prio_j[i] := base_prio_j + cpu_j[i]/2 + gcpu_j[i]/(4\ W_k)$

- The priority of thread *j* deteriorates not only due to heavy CPU usage of the thread itself but also when other threads of the same group are compute intensive.

Scheduling in Windows NT/2000/…

- Windows NT/2000/… also uses a preemptive strategy based on priorities and time slices.
- 32 priority classes are distinguished:
  - 16-31 for real time threads
  - 1-15 for normal threads
  - 0 for the idle thread
- The standard time quantum is 6 units for workstations and 36 for servers.
- At each interrupt by the timer (*clock tick*) the current time quantum is decremented by 3 units.
- The clock resolution is 10-15 msec (for Intel processors)

  i. e. a time quantum is 20-30 msec for a Workstation and 150-180 msec for a server.

Scheduling in Windows NT/2000/…

- Windows NT/2000/… also dynamically adapts priorities:
  - When an I/O-request is finished the issuing thread increases its priority by 1-8 priority levels (depending on the type of I/O) (*priority boost*).
  - After expiration of a time slice the priority of a thread is decremented by 1 until the original value is reached again.
  - Priority increase also takes place when a thread has spent a long time (3-4 seconds) in ready queue. (*CPU starvation*)
  - By that also the problem of priority inversion is alleviated.
- In addition, workstations provide the possibility of increasing the size of the time slice (*quantum stretching*):
  - A foreground thread (active window) can get a time slice twice or three times as large.

Scheduling in Linux

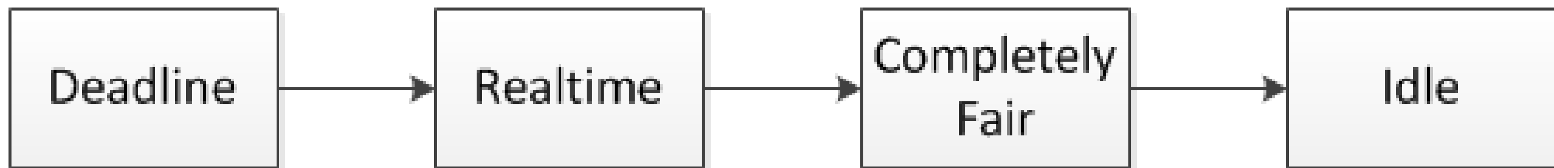
- Linux allows to stack scheduling policies in separated (sub-)modules.
- All policies have their own runqueue (or similar data structure).
- Processes are assigned to one scheduling policy only.
- Policies (sub-modules) are organized by a linked list. Position with in the list represents priority of the policy.
- Scheduler calls the entries of the policy list for a runnable process.
- Executing all runnable processes (tasks) of one scheduling policy before switching to the next policy.
  - Policy may return request of restart the search for runnable process at the begin of the list.

Scheduling in Linux

- Introduction of stacked policies with kernel version 2.6.23 with three classes:
- rt_sched_class
  - For “real-time” tasks.
  - Executed before anything else.
  - Fifo and Round Robin strategy.
  - Based on static priorities.
- fair_sched_class
  - For normal processes.
  - Completely Fair Scheduler (CFS), a proportional share scheduler.
- idle_sched_class
  - Just the idle thread.
  - Executed if there is nothing else.

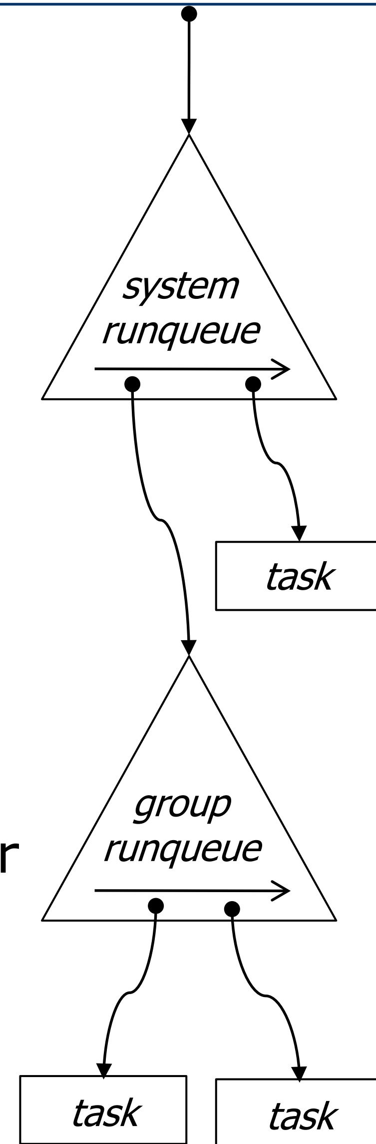
Scheduling in Linux

- Additional policy introduced with kernel version 3.14:
- sched_dl_class
  - For “real” real-time tasks.
  - Executed before anything else.
  - Executed Earliest Deadline First (EDF) strategy.
  - Uses three parameters, named "runtime", "period", and "deadline" for scheduling.



Completely Fair Scheduler

- Each task has a weight w_i and virtual runtime vruntime.
  - Virtual runtime is increased by the weighted actual runtime:
    - $vruntime_i \mathrel{+}= \Delta t_i * W/w_i$
- A red-black tree with vruntime as key is used as runqueue.
  - Complexity of O(log n) for (re-)inserting a task.
  - Newly created tasks are inserted to the right, woken tasks are inserted to the left.
  - Scheduler selects the task with the lowest vruntime.
- Allowing not only tasks within a runqueue but other runqueues as well allows to achieve (weighted) fairness between groups of tasks by recursively applying the scheduling logic.

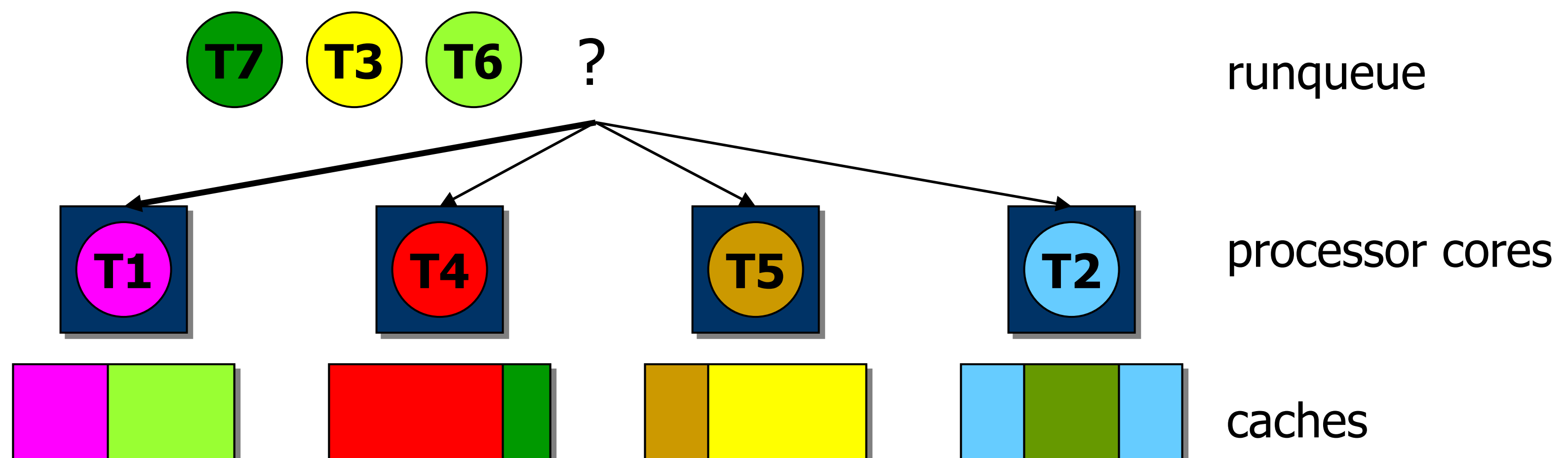


4.2.6 Multiprocessor aspects

- On a multiprocessor and/or multicore system, the scheduler must not only decide **when** to execute a task, but also **where**.
- Even if all processors or processors cores are identical, it is not irrelevant on which core a task is running or which other tasks are executing on other cores simultaneously due to shared and limited resources within the processors themselves.
  - Some caches are per core, some caches are shared by multiple cores.
  - Memory bandwidth is limited and shared by some cores.
  - Memory access can be non-uniform.
  - Logical cores share the execution units within a physical core.
  - …
- Different schedules vary greatly in their performance and their energy consumption.

Processor affinity

- If a thread was running on a processor, the corresponding caches were filled with data belonging to that thread. If this thread is scheduled again, there is a chance that significant parts of the cache still belong to that thread. (The cache is still warm/hot.)
- Therefore, we have the concept of processor affinity.
  - The scheduler memorizes the "favored" or recently used processor.
  - However, by using this, the priority principle is "watered down": It may happen that a thread of higher priority waits (for its favored processor) although another thread with lower priority is running.



Coscheduling/Gang scheduling

Freie Universität Berlin

- Threads (processes or tasks) of a parallel application work together to reach one goal.
- Processes should run in a coordinated manner:
  - Heavily communicating tasks:
    - Tasks do not need to block as they know that their communication partner is running and that they will receive an answer soon.
  - Tasks sharing data.
    - Shared caches are exclusively used by tasks of the coscheduled group. Thus, they do not have to compete with other tasks for capacity.
  - Tasks with contrary resource demands.
    - Coscheduling them causes less resource contention (e. g. execution units, memory bandwidth).

Coscheduling/Gang scheduling

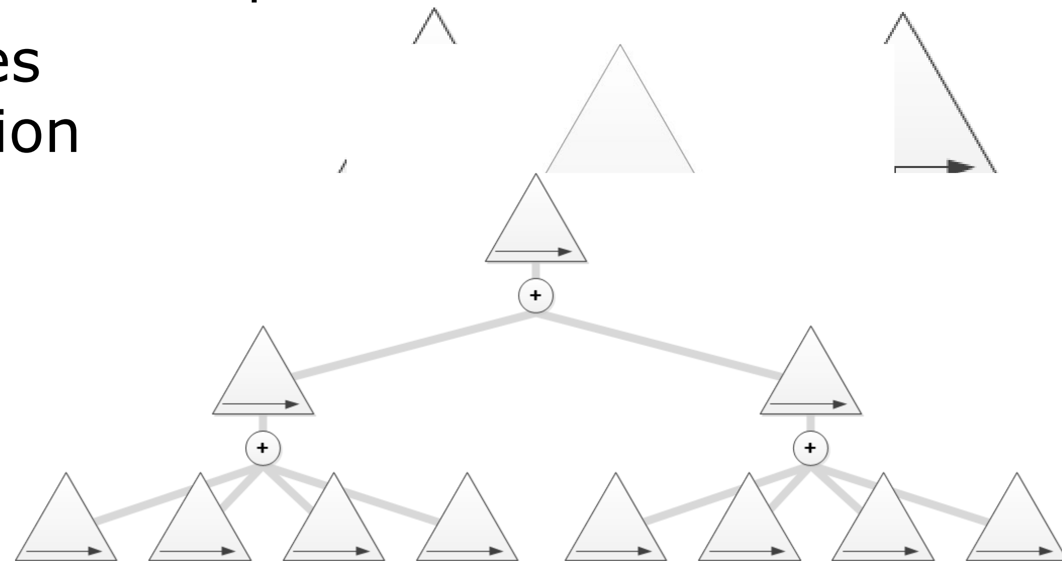
- **Gang scheduling** is the scheduling of a group of threads to run on a set of processors at the same time, on a one-to-one basis.
  - Providing time-slices and synchronized or collective preemption.
  - Prevents idle processors in the set from executing threads not belonging to the group.
    - Threads of the scheduling group are not forced to relinquish the CPU or will not be blocked.
- Create the illusion that the tasks of the gang scheduled/coscheduled group are running exclusively in the system.

Coscheduling/Gang scheduling

- A group of tasks is **coscheduled** on a set of processors, if as many of the group's tasks as possible are executed on the set of processors simultaneously.
  - I. e., no processor in the set does something else while there are runnable but currently not executed tasks in the group.
- Synchronization by central clock signal is needed.

Example of a Coscheduler – TACO

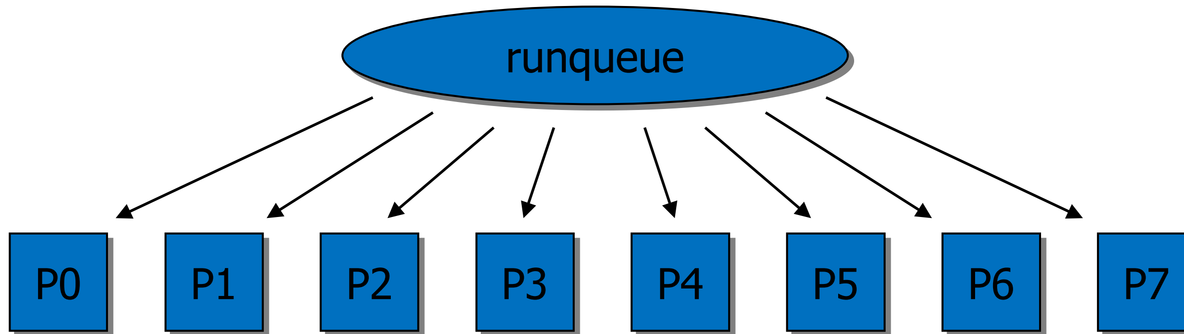
- Jan Schönherr (KBS, TUB):
  Topology-aware Coscheduling – TACO
- Building synchronization domains as set of processors with own runqueue.
  - Master CPU is responsible to enforce thread switch at all CPUs of the synchronization domain.
  - CPU picks thread out of the runqueue and notifies master.
- Maps sets of coschedules threads to synchronization domains.
- Prototyp for Linux CFS and FreeBSD.



Centralized scheduler design

Freie Universität Berlin

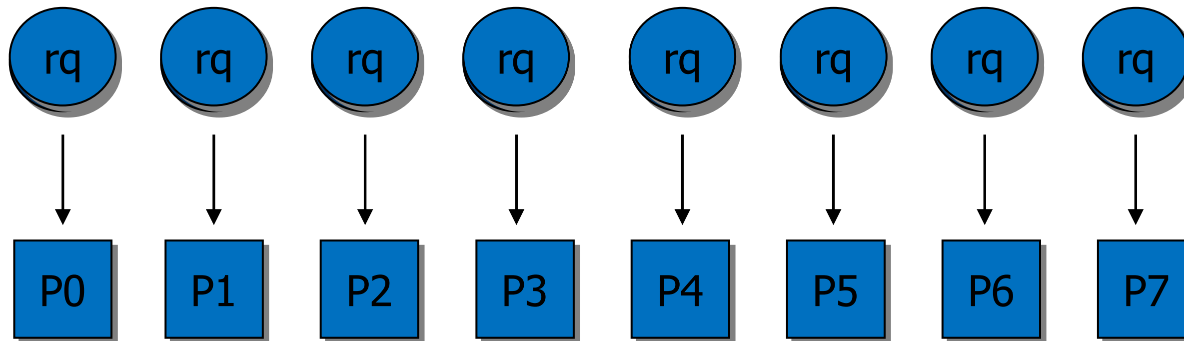
- One runqueue for all processors:



- Scales only up to 4 to 16 processors/processor cores.
- Global knowledge makes some types of scheduling easier (e. g. coscheduling, absolute priorities) and some harder (e. g. realizing processor affinity).
- Examples:
  - Linux BFS

(Fully) Decentralized scheduler design

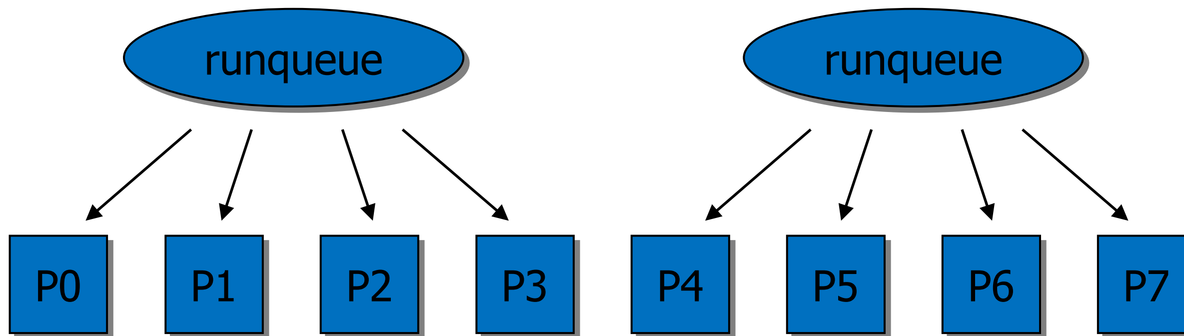
- One runqueue per processor core:



- Scales to very large systems.
- Needs some kind of load balancing to be versatile.
- Distributed knowledge makes global decisions (nearly) impossible. But processor affinity is very cheap.
- Examples:
  - Most current operating systems.

Semi-decentralized scheduler design

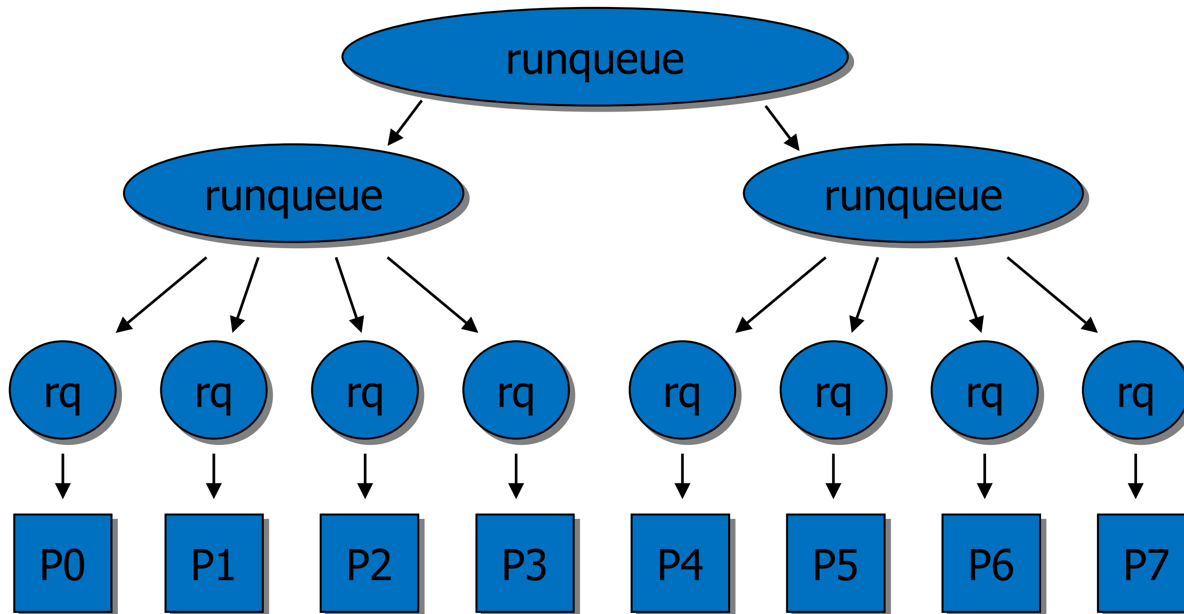
- Multiple cores share a runqueue:



- Tradeoff between both extremes.
- Examples:
  - VMWare ESX 3.x

Hierarchical scheduler design

- A hierarchy of runqueues, where runqueues further up in the hierarchy represent larger fractions of the system:



- For a more scalable coscheduling:
  - Multiple small coscheduled sets can be processed independently.
  - In the absence of coscheduled sets, this is similar to a decentralized scheduler.

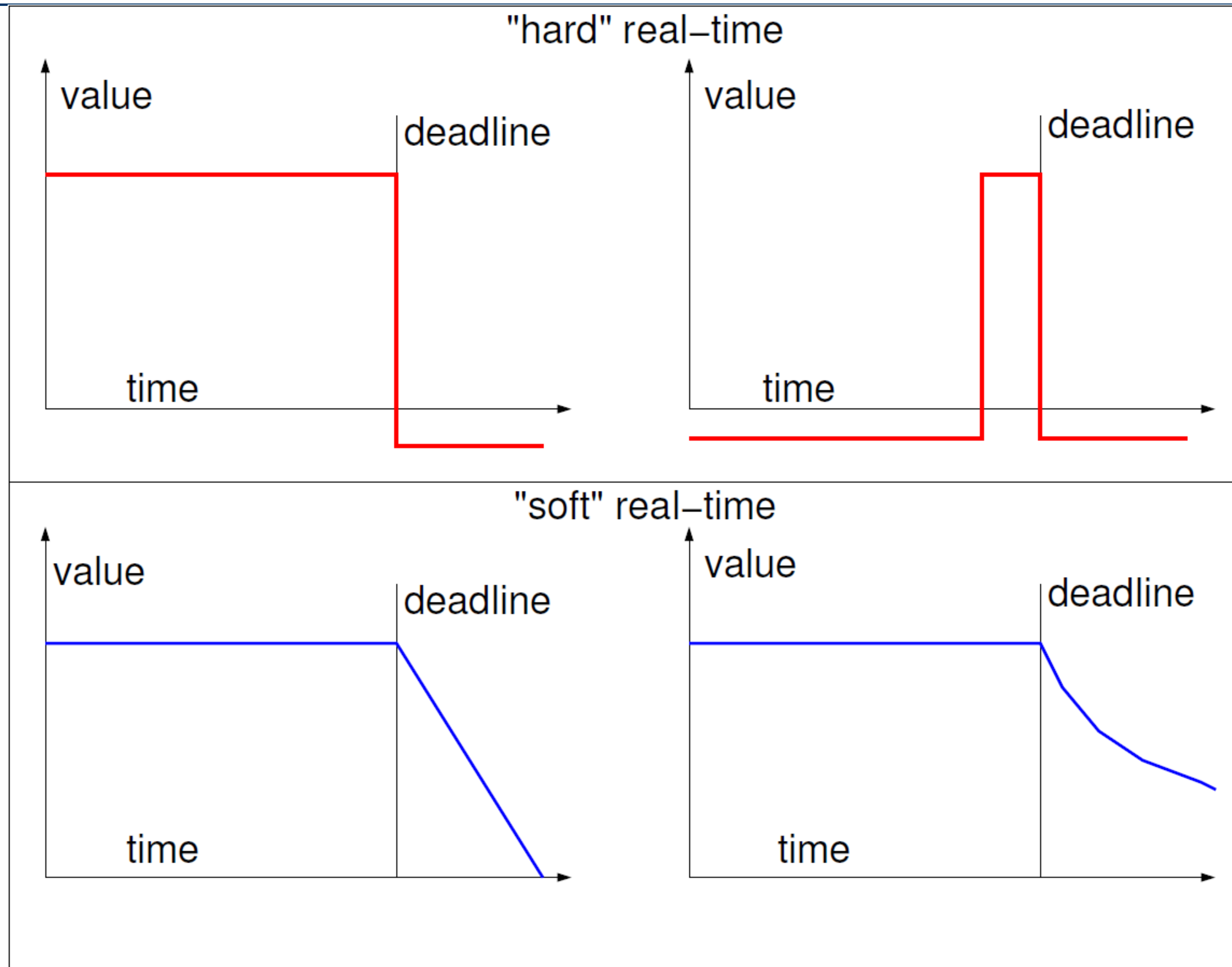
Case study: Linux CFS

- One runqueue per core.
- Load balancing is done periodically and on demand.
  - Honors system topology, tries to avoid costly migrations.
  - Two runqueues are balanced, if they have the same weight.
- Proportional share scheduling
  - Relies on a balanced system, i. e. proportional share scheduling is achieved only locally.
    - No processor is forced idle if a task gets more than its proportional share of CPU time.
  - Tasks of a task group might be scheduled on multiple processors.
    - Hence, a task group is also represented by multiple runqueues.
    - The share of a task group is split: each representative gets a fraction of the share proportional to the weight of the tasks within.

4.3 Real-time scheduling

- In real-time systems (e.g. controller for production lines, nuclear power plants, engine control,…) the goal of scheduling is different:
- Within short time constraints, measurements need to be evaluated *and based on that, action must be taken*.
- Threads are therefore associated with *deadlines*, at which they must be finished.
- Since meeting the deadlines sometimes is critical for the function of the complete system, they need to be considered in scheduling decisions.
- Concerning the implications violating deadlines we make the following distinction:
  - *hard real-time systems*: Violation means failure of the system and cannot be tolerated (Example: Airbag, antiblocking brake).
  - *soft real-time systems:* Violation means quality reduction but can be tolerated (Example: voice over IP, transmission and processing of video streams, synchronization of video and audio).
- In hard real-time systems often off-line-algorithms are employed to guarantee that deadlines can be met.

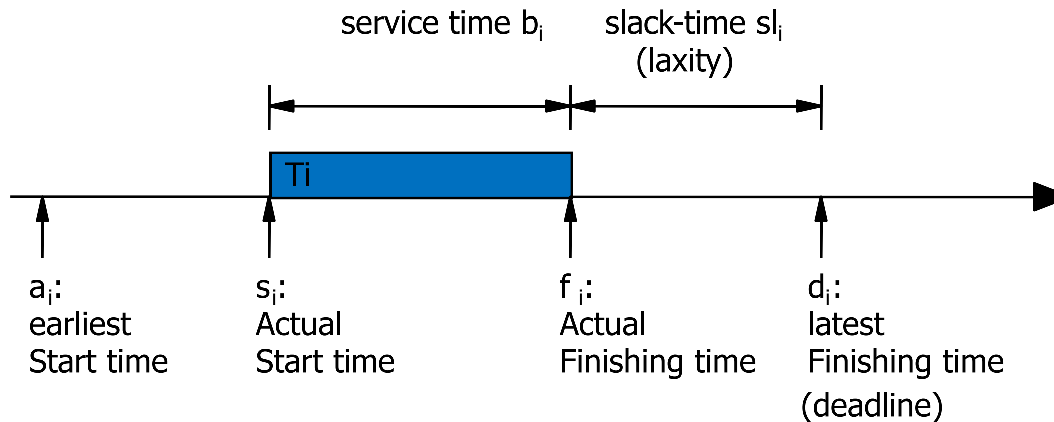
Hard Real-time vs. Soft Real-time



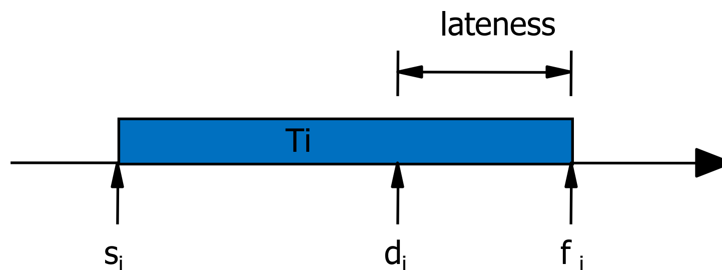
"Value" means value for the user or intended application

Basic Concepts

For real-time threads, it is given by the specification when they can start at the earliest and when they must be finished at the latest.



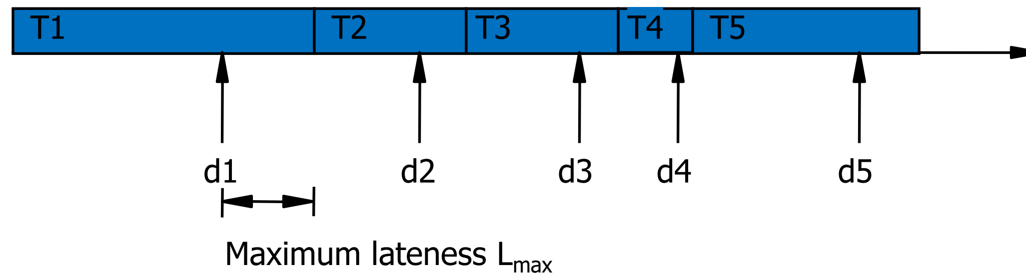
Exceeding the deadline ($f_i > d_i$) is called *lateness* $L := f_i - d_i$



Goals

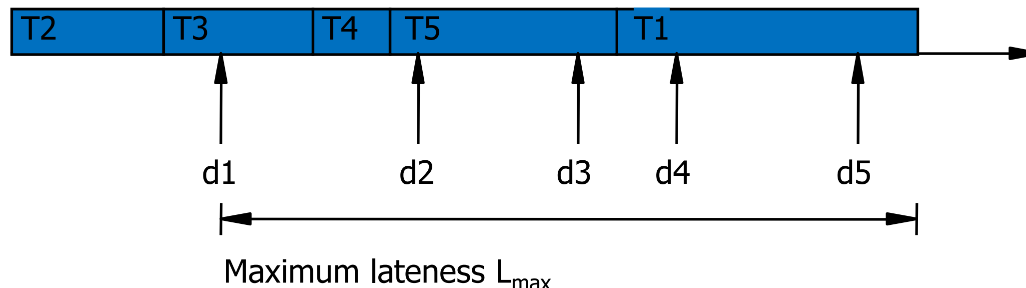
The objective functions often depend on whether the lateness can be tolerated or not.

In soft-RT-systems the maximum lateness L_{max} can be minimized.



Minimizing the maximum lateness in this case means that all other deadlines are violated.

In the following schedule we obtain a much larger lateness, but can meet all but one deadlines.



Minimization of maximum lateness

On single processor systems without preemption and without dependencies the scheduling problems boils down to finding an appropriate permutation of the threads.

In this case the following theorem (EDD = Earliest Due Date) holds:

Theorem (Jackson's Rule):

Each schedule, in which the threads are processed in the order of non-decreasing deadlines is optimal with regard to L_{max}

The threads need to be sorted according to their deadlines which can be done in O(n log n).

EDD assumes that all threads can start at any time.

However, in many applications we do have earliest start times a_i *(availability of measurement data).*

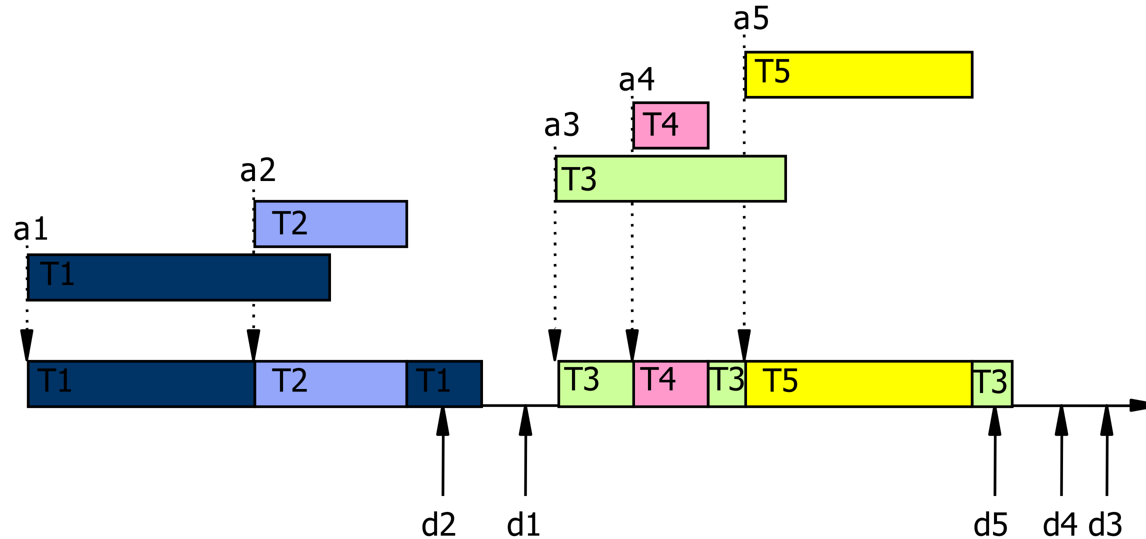
By introducing individual start times ($\exists\, i,j$: $a_i \neq a_j$) the problem becomes NP-hard, i.e. there is no optimal solution in polynomial time.

Minimizing maximum lateness (with start times and preemption)

If, however, preemption is possible the following theorem holds which again is based on Jackson's rule:

Theorem (EDF: Earliest Deadline First)

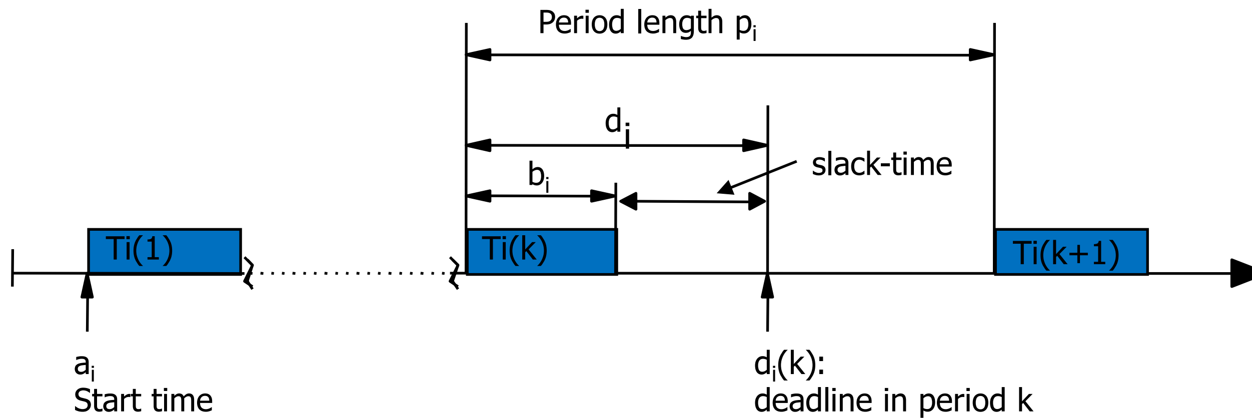
Each schedule in which at any time the thread with the earliest deadline is assigned is optimal with regard to the maximum lateness.



Periodical threads

In many real-time applications we have to deal with periodical tasks that have to be finished in due time. Each thread is characterized by its **period** or the **rate** (which is reciprocal to the period).

First we have to check whether the sequence of tasks can be executed anyway (schedulability test, feasibility test).



For each individual periodical thread the following must hold: $0 < b_i \le d_i$

For the set of all periodical threads the following is a necessary condition for the existence of a feasible schedule:

$$\sum_i \frac{b_i}{p_i} \le 1$$

Periodical threads

As scheduling strategy for periodic threads the so-called **rate monotonic scheduling (RMS)** is used in many cases:

Threads are assigned a **static** priority which is inversely proportional to their period, i.e. the thread with the smallest period gets the highest priority (the priority is according to the rates).

For independent threads and if the deadlines coincide with the periods the following theorem can be proved:

A set of *n* periodical threads can be scheduled by a rate-monotonic strategy if the following inequality holds (sufficient condition):

$$\sum_{i=1}^{n} \frac{b_i}{p_i} \leq n \left(2^{\frac{1}{n}} - 1 \right)$$

The left hand side of the inequality denotes the required processor capacity and the right hand side an upper bound that must be valid in order to find a feasible schedule.

For large *n* the upper bound means that the processor utilization must not be larger than log 2 ≈ 69,3 %.

Rate Monotonic Scheduling

Assumptions:

(1) Task T_i is periodical with period-length p_i

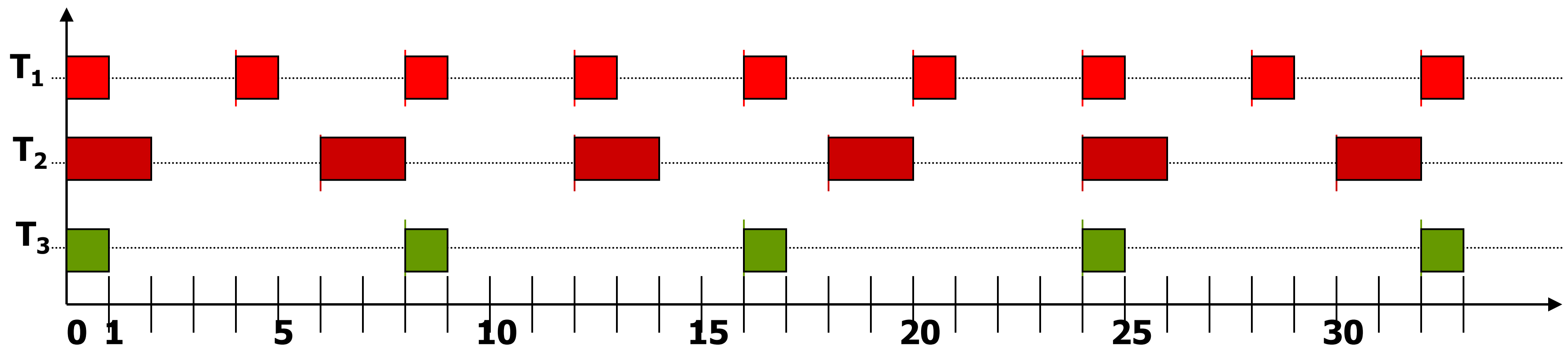
(2) Deadline $d_i = p_i$

(3) T_i is ready again immediately after p_i

(4) T_i has a constant execution time b_i ($<= p_i$)

(5) The smaller the period the higher the priority

<u>Example:</u> $T = \{T_1, T_2, T_3\}$, $p=\{4, 6, 8\}$, $b = \{1, 2, 1\}$



How to schedule on 1 CPU? Just use the above priority scheme!

Rate Monotonic Scheduling (RMS)

Assumptions:

(1) Task T_i is periodical with period-length p_i

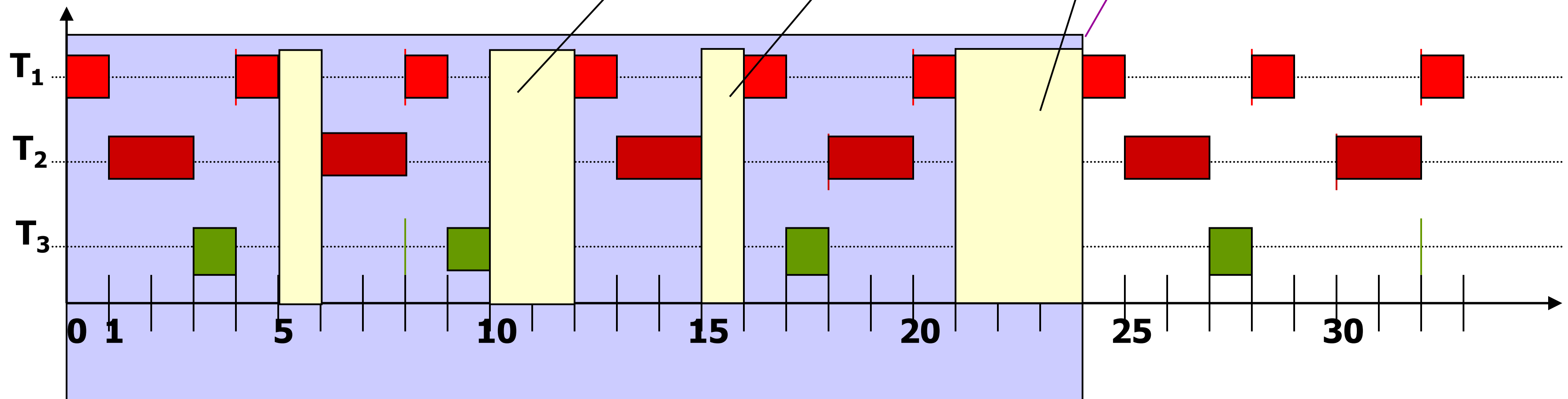
(2) Deadline $d_i = p_i$

(3) T_i is ready again immediately after p_i

(4) T_i has a constant execution time b_i ($\leq p_i$)

(5) The smaller the period the higher the priority

Example: $T = \{T_1, T_2, T_3\}$, $p = \{4, 6, 8\}$, $b = \{1, 2, 1\}$



Result of Rate Monotonic Scheduling Example

Freie Universität Berlin

The general necessary **feasibility criterion** is met:

$$\mathbf{(1/4 + 2/6 + 1/8) = 17/24 <= 1}$$

Also the RMS-criterion is satisfied:

$$\mathbf{(1/4 + 2/6 + 1/8) = 17/24 = 0{,}7083 < 3\,(\,2^{1/3}-1) \approx 78\ \%.}$$

Rate Monotonic Scheduling

Assumptions:

(1) Task T_i is periodical with period-length p_i

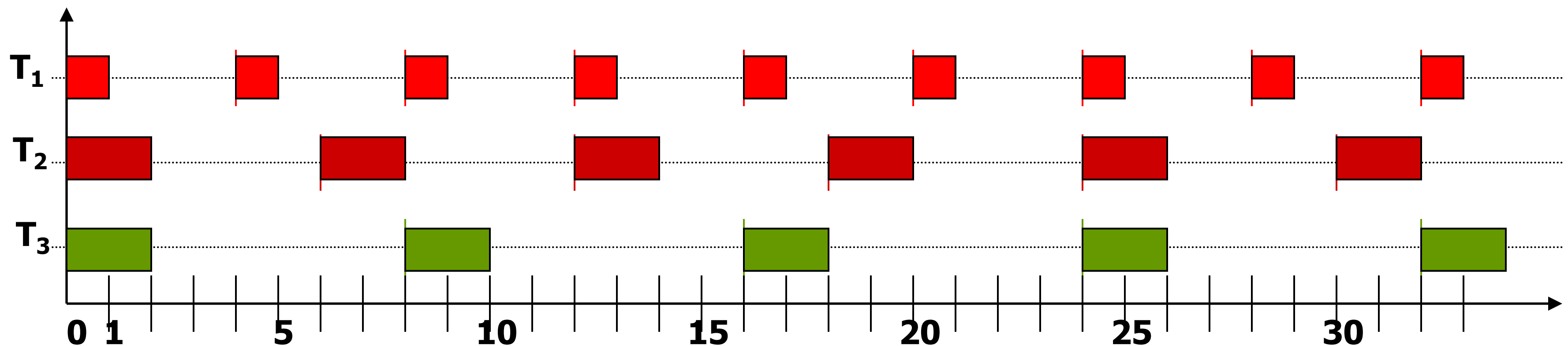
(2) Deadline $d_i = p_i$

(3) T_i is ready again immediately after p_i

(4) T_i has a constant execution time b_i ($<= p_i$)

(5) The smaller the period the higher the priority

Example: $T = \{T_1, T_2, T_3\}$, $p = \{4, 6, 8\}$, $b = \{1, 2, \mathbf{2}\}$



How to schedule on 1 CPU? Just use the above priority scheme!

Rate Monotonic Scheduling (RMS)

Freie Universität Berlin

Assumptions:

(1) Task T_i is periodical with with period-length p_i

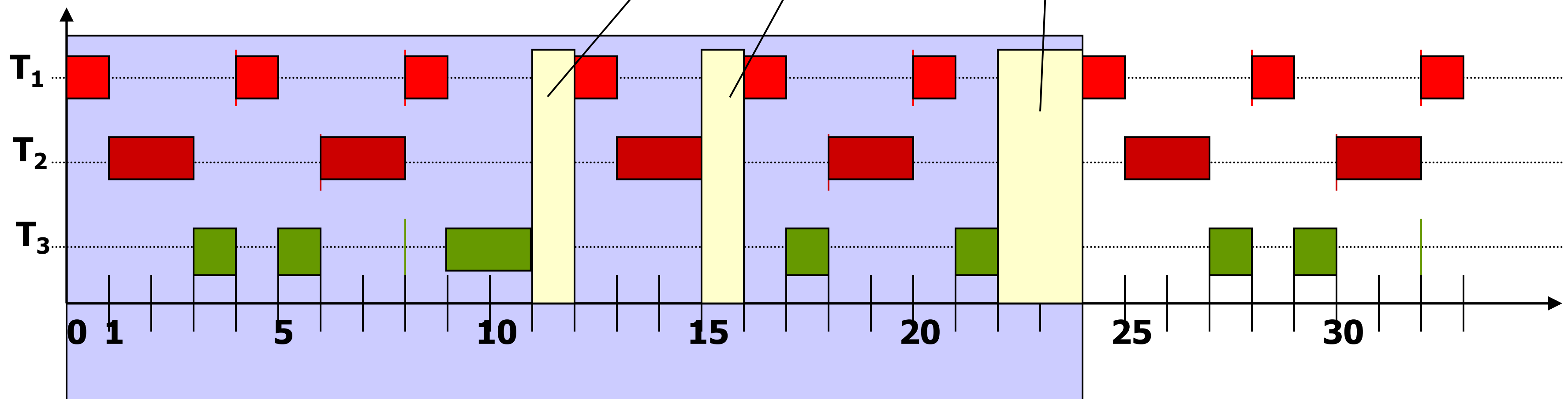
(2) Deadline $d_i = p_i$

(3) T_i is ready again immediately after p_i

(4) Ti has a constant execution time b_i ($<= p_i$)

(5) The smaller the period the higher the priority

Example: T ={T_1, T_2, T_3}, p={4, 6, 8}, b ={1, 2, **2**}



Result of Rate Monotonic Scheduling Example

Freie Universität Berlin

The general necessary **feasibility criterion** is met:

$$\mathbf{(1/4 + 2/6 + 2/8) = 20/24 = 0.833 <= 1}$$

However, the sufficient RMS-criterion violated:

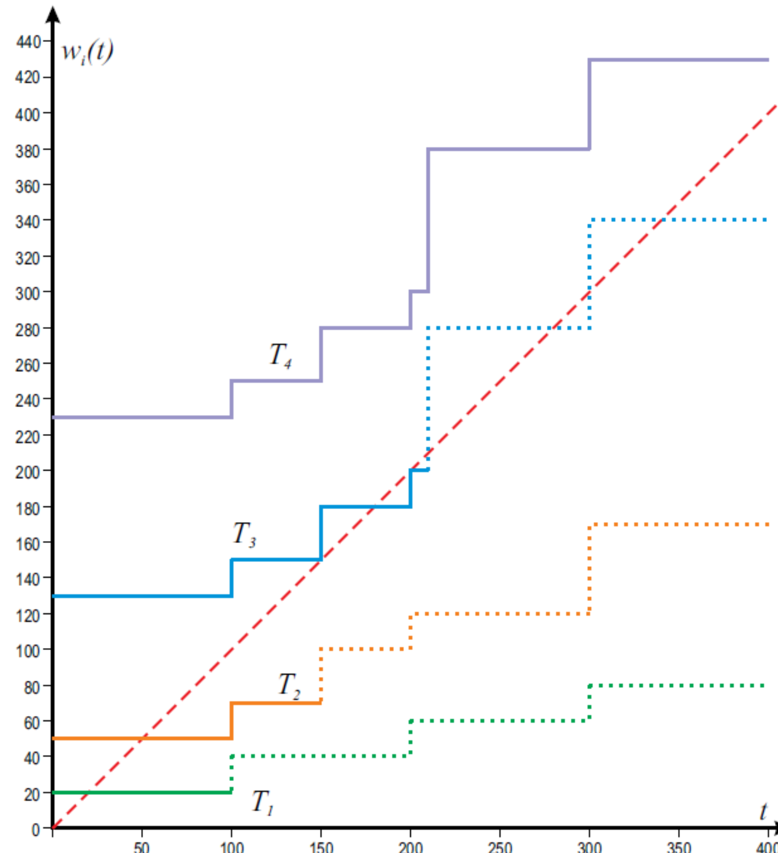
$$\mathbf{(1/4 + 2/6 + 2/8) = 20/24 = 0.8333 > 3\,(2^{1/3}-1) \approx 78\,\%.}$$

Solution: Time Demand Analysis (TDA)

Time Demand Analysis (TDA)

- Introduced by Lehoczky, Sha and Ding in 1989
- Idea: Calculate worst case time demand for all threads and compare to available time to deadline
  → Find point in time where enough time is available to finish

	e_i	P_i
T_1	20	100
T_2	30	150
T_3	80	210
T_4	100	400



Earliest Deadline First for Periodical threads

Freie Universität Berlin

Another scheduling strategy for periodic real-time threads is **Earliest Deadline First (EDF)**:

Threads are assigned a **dynamic** priority which is inversely proportional to the current distance to the deadline, i.e. the thread with the next deadline gets the highest priority. Preemption is usually allowed.

For independent threads and if the deadlines coincide with the periods the following theorem can be proved:

A set of *n* periodical threads can be scheduled by an earliest deadline first strategy if and only if the following inequality holds:

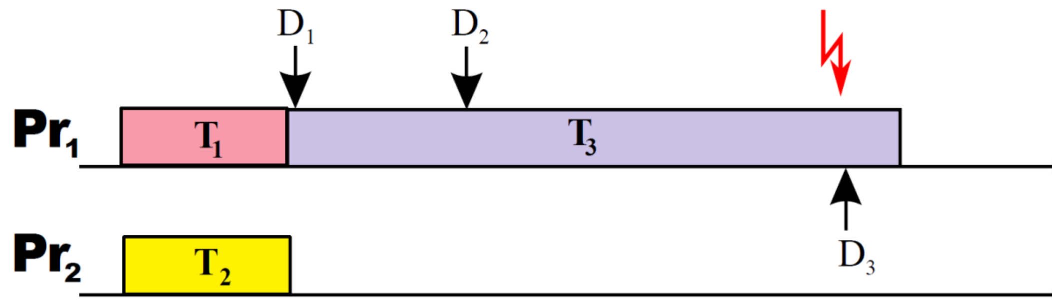
$$\sum_{i=1}^{n} \frac{b_i}{p_i} \leq 1$$

The left hand side of the inequality denotes the required processor capacity.

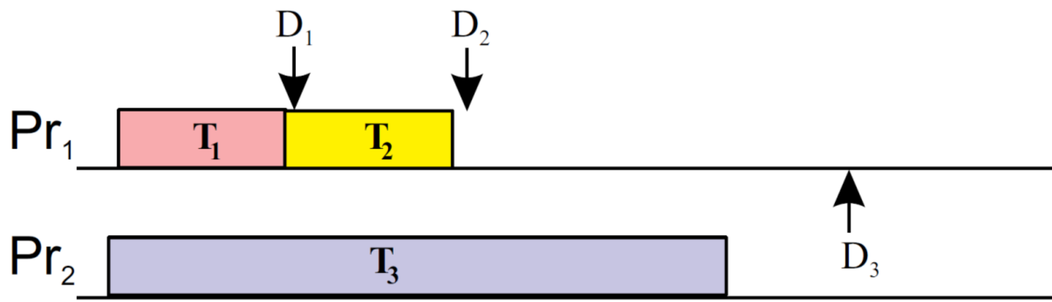
This means that if a schedule exists for a set of tasks, EDF is also able to schedule that set of tasks (optimality).

Multi-Processor Real-Time Scheduling

- Simple approaches such as RMS or EDF do not work for multiple processors.
- Example: Not executable using EDF on two processors.



- However, there is a schedule:



- Unfortunately, optimal scheduling is NP-complete in nearly all relevant cases.

Earliest Deadline Zero Laxity (EDZL)

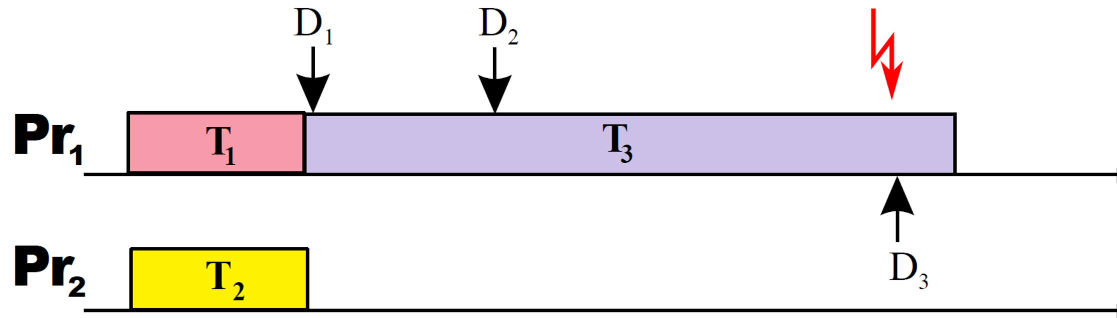
Freie Universität Berlin

- Idea
  - Global EDF scheduling, but
  - Tasks with zero laxitiy (i.e. that has to run now in order to meet the deadline) get highest priority

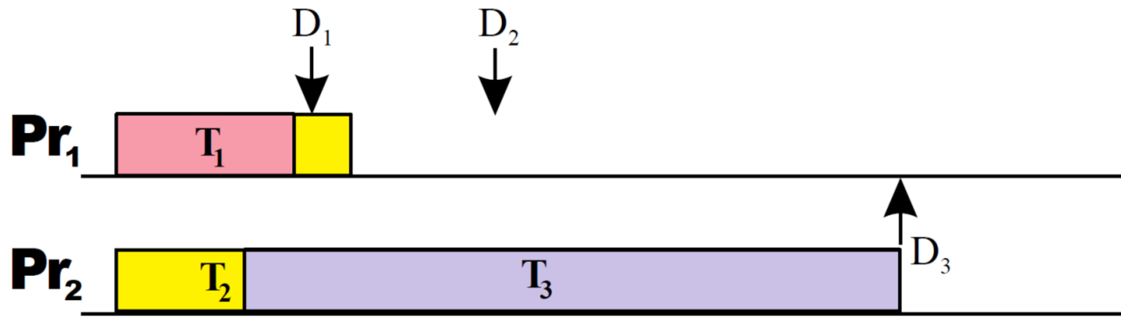
- Properties
  - Never worse that global EDF
  - Lot of ongoing research on criterias and bounds

Earliest Deadline Zero Laxity (EDZL)

- Not executable using EDF on two processors:



- Executable using EDZL on two processors:



4.4 Plan-based scheduling

- Scheduling in High Performance Computing (HPC)
  - Scheduling on different levels:
    - Parallel program (job) – whole machine
    - Processes or threads as part of the parallel program – compute node as part of the HPC system
  - Mapping of the processes to compute nodes
    (more about mapping
    and scheduling in CC)

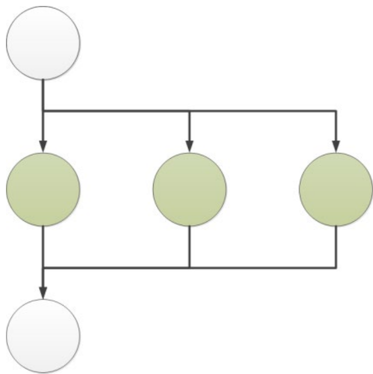
- Research is part of a project to build a OS for Grid computing – VRM
    - Jörg Schneider,
    - Lars-Olof Burchard,
    - Barry Linnert

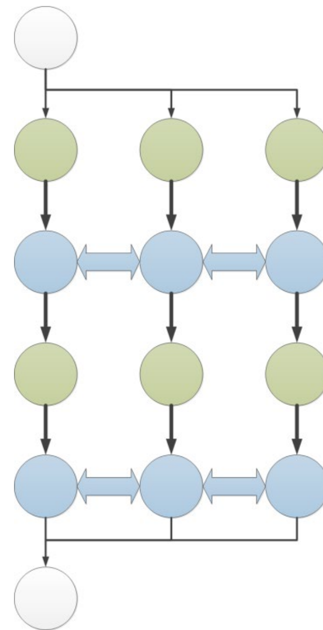
Oak Ridge National Lab

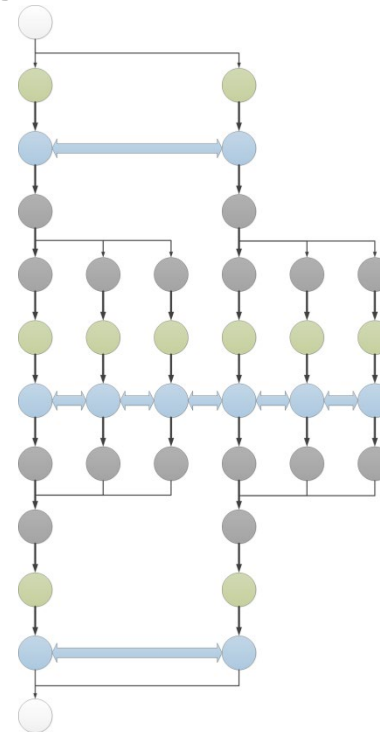
HPC applications

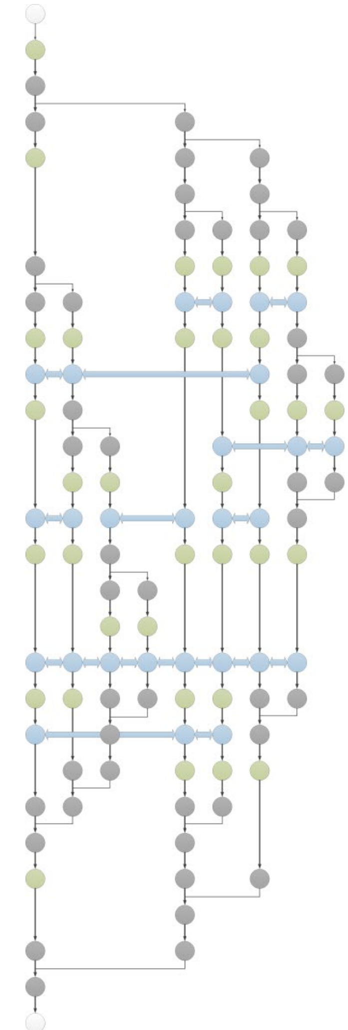
Freie Universität Berlin

- Parallel programs with different runtime behavior
  - Computational load
  - Communication
  - Dynamic creation and finishing of processes



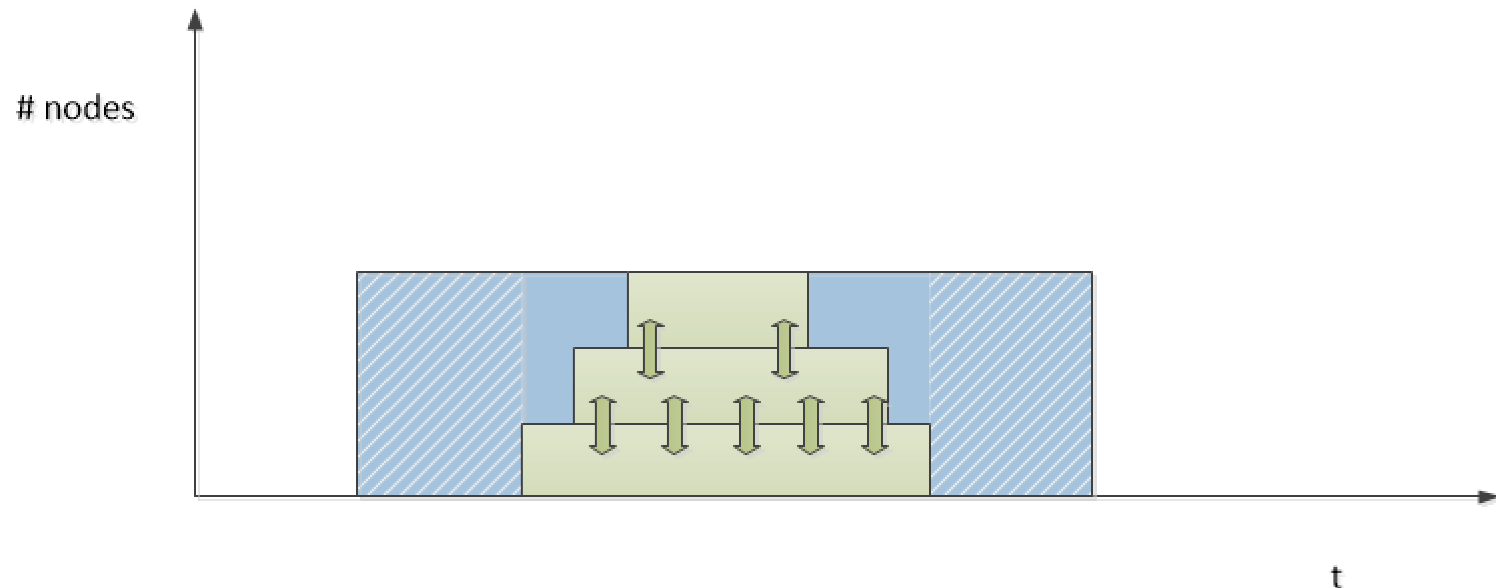






Conditions for running HPC jobs

- The result of the execution of the parallel program should be available at a specific time – deadline

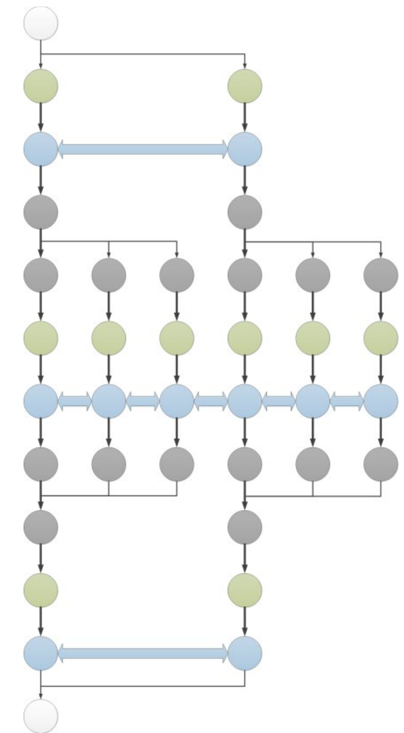


- Waste of compute resources should be reduced by using time sharing
  - Running various processes on the node

Runtime behavior models

Freie Universität Berlin

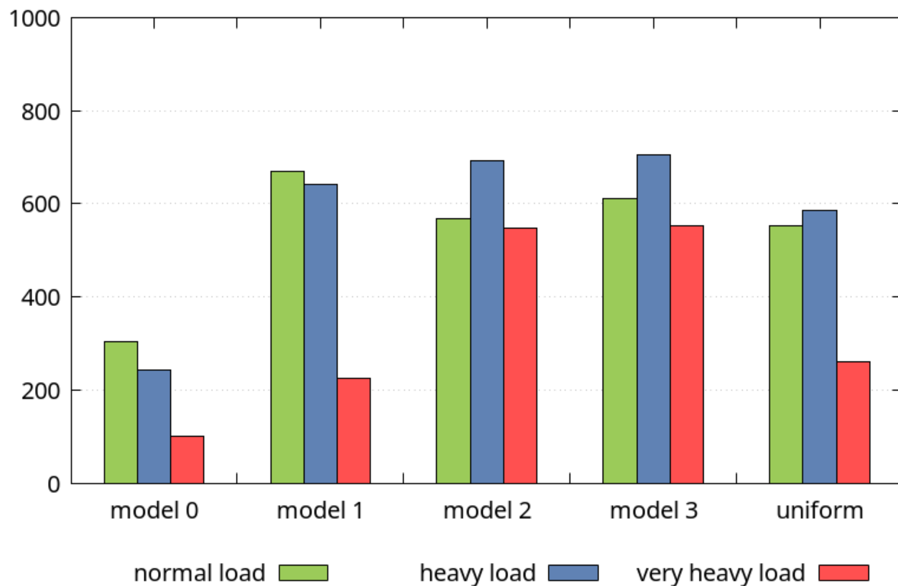
- Using runtime behavior models to schedule and map the job to the compute nodes (and network links)
  - Observing the runtime behavior and using this information for future runs

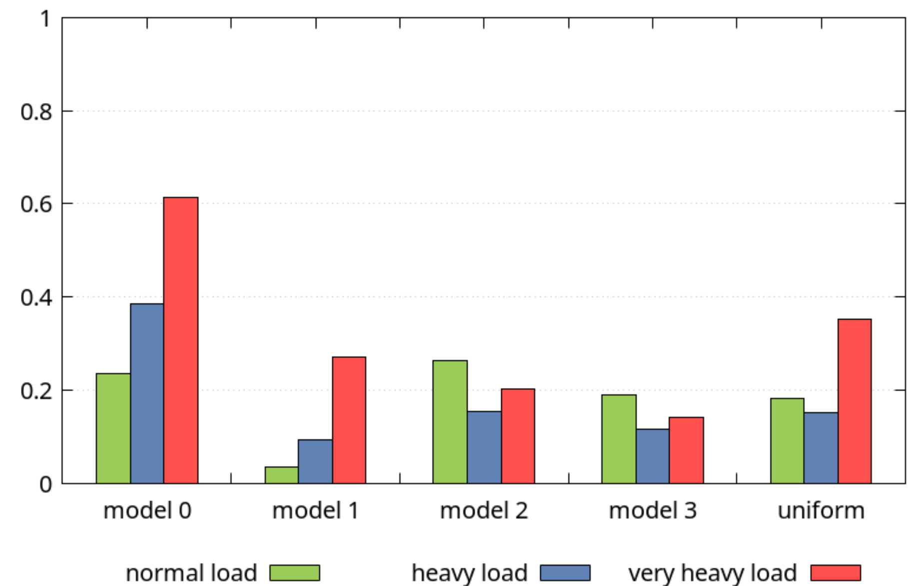


- The operating system running the node still schedules the processes of the parallel programs
- Most of the HPC systems are running Linux
  - CFS is the scheduler for the processes

CFS-based node scheduler

- Simulation results for various configuration
  - HPC systems, program types, estimations, communication patterns



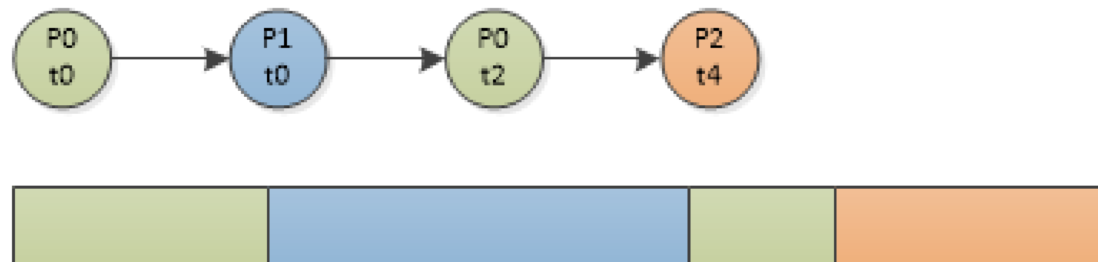


Results for 1000 jobs of different program model types on homogenous grid topology, 0% runtime overestimation with asynchronous communication, with RR and time slice of 200 ms

Plan-based node scheduler

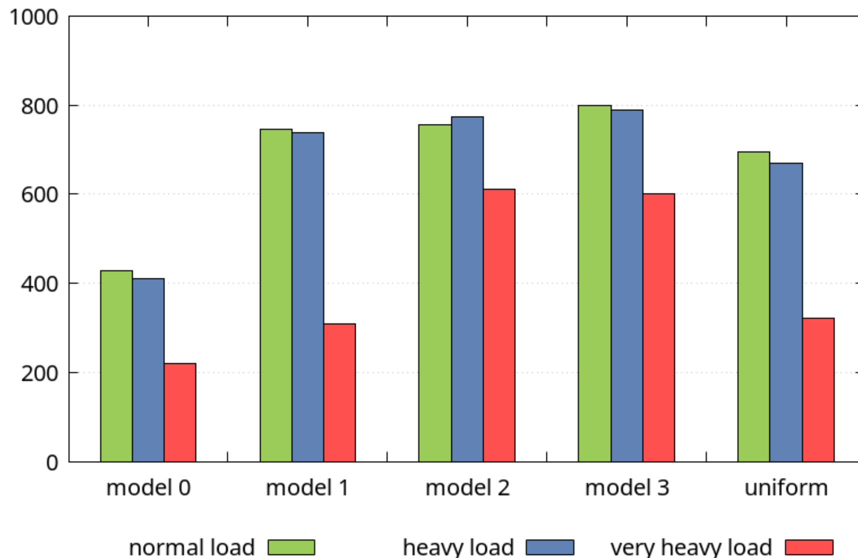
- Kelvin Glaß and Barry Linnert (FUB/KBS, TUB)
- Additional scheduling class to the Linux scheduler

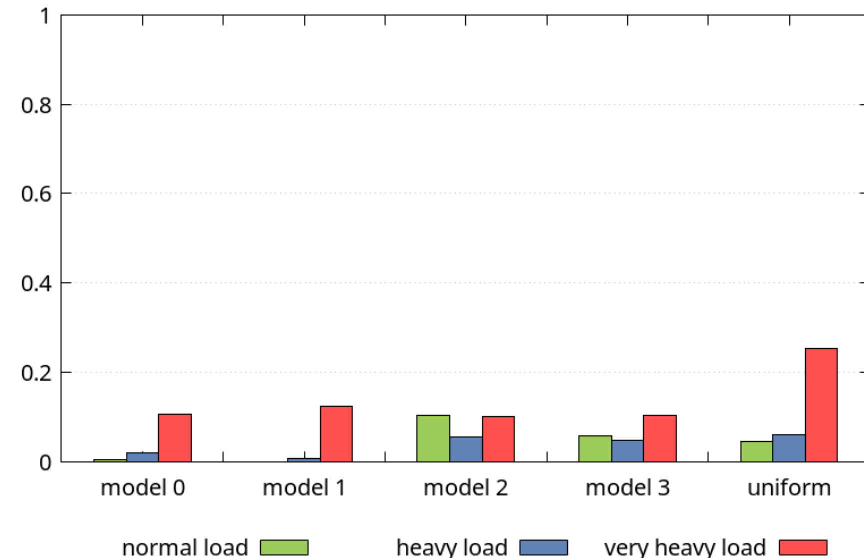
- Executes processes (parts of processes – tasks) following given plan
- Ensures implicit synchronization between different nodes of the cluster
  computer resource



Plan-based node scheduler

- Using the plan-based node scheduler the
  - rate of canceled jobs can be reduced to about 5%,
  - the number of successfully completed job can be increased up to 677 (out of 1000) jobs
  - for uniform distributed program behavior types on homogenous grid topology, under heavy load, 0% runtime overestimation with asynchronous communication.





Further References

- Stallings, W.: *Operating Systems* 5th ed., Prentice Hall, 2005
  Chapter 9 + 10
- Solomon, D. A.; Russinovich:
  Inside Microsoft Windows 2000,
  Microsoft Press, 1998
- D. Giani, S. Vaddagiri, P. Zijlstra:
  The Linux Scheduler, today and looking forward.
  October 2008.
- C.L. Lui und J.W. Layland:
  Scheduling Algorithms for Multiprogramming in a
  Hard-Real-Time Environment. 1973.
- Linux Kernel: sched/fair.c, sched/core.c, and CFS documentation.
  http://kernel.org
- D. G. Feitelson and L. Rudolph:
  Distributed hierarchical control for parallel processing,
  Computer, vol. 23, no. 5, pp. 65–77, May 1990.
- Jan H. Schönherr:
  Coscheduling in the Multicore Era, PhD thesis, TU Berlin, 2019
- Kelvin Glaß: Plan Based Thread Scheduling on HPC Nodes
  Master thesis, Freie Universität Berlin, 2018.