POLITECNICO
MILANO 1863

DIPARTIMENTO DI ELETTRONICA,
INFORMAZIONE E BIOINGEGNERIA

ADVANCED OPERATING SYSTEMS
Prof. William Fornaciari

Task Scheduling

William Fornaciari

william.fornaciari@polimi.it

Outline

- Introduction
- Classification of scheduling algorithms
- Scheduling algorithms
- Priority-based scheduling
- Multiprocessor scheduling

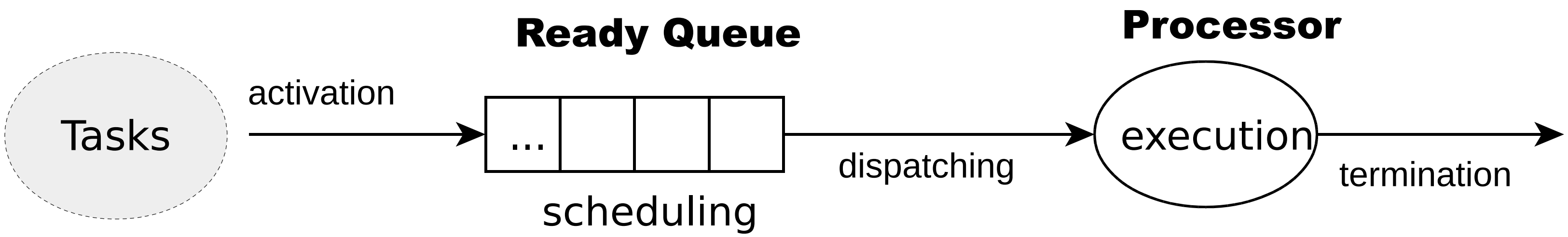
Introduction

Basic concepts and terminology

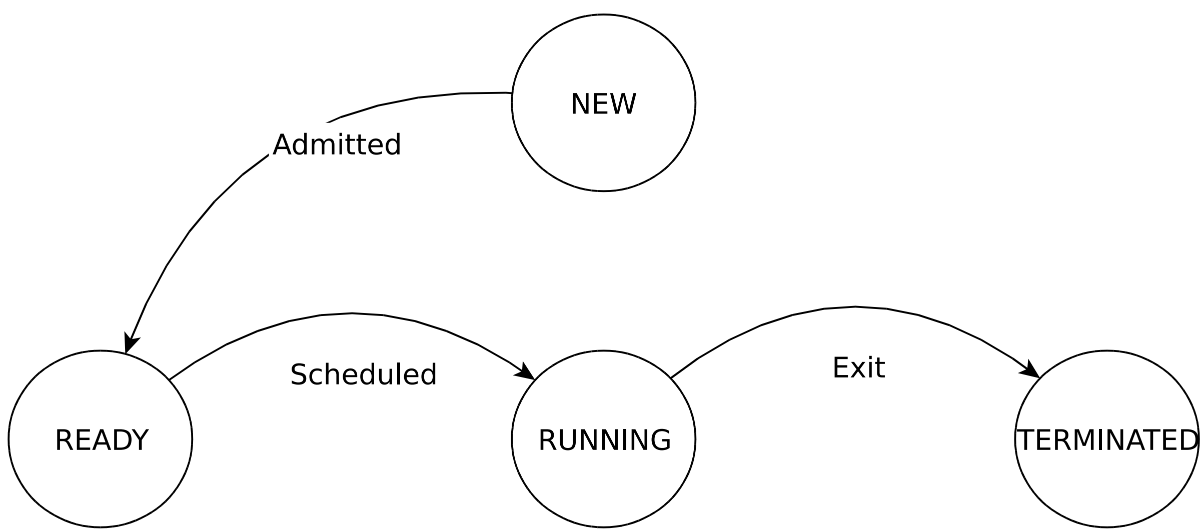
- A task is a computation performed by the processor in a sequential fashion. It is characterized by an execution status:
  - *Running* – the processor is executing the task
  - *Ready* – the task is waiting for the allocation of the processor
  - *Active* – used to refer to both running and ready tasks
- Ready tasks are kept in a ready queue by the operating system
- The scheduler is the OS component in charge of establishing the execution order of the tasks
  - Ordering algorithm for the ready queue (scheduling policy)
  - Scheduling policy invocation can be *periodic* or *event-based* (e.g., arrival or termination of tasks)
- Dispatching : allocating a (the) processor to a task

Introduction

Scheduling overview

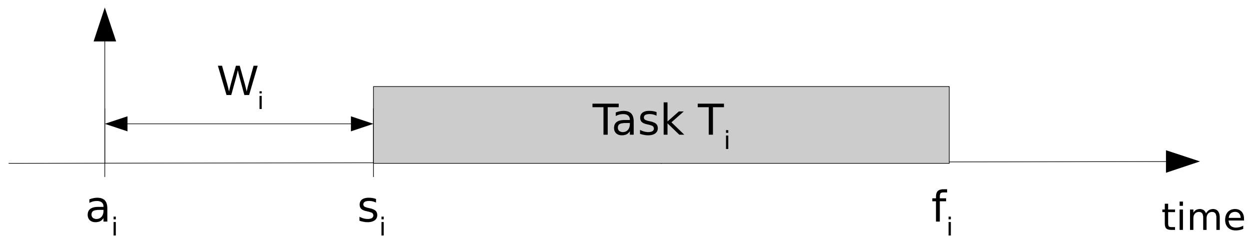


Task st



Introduction

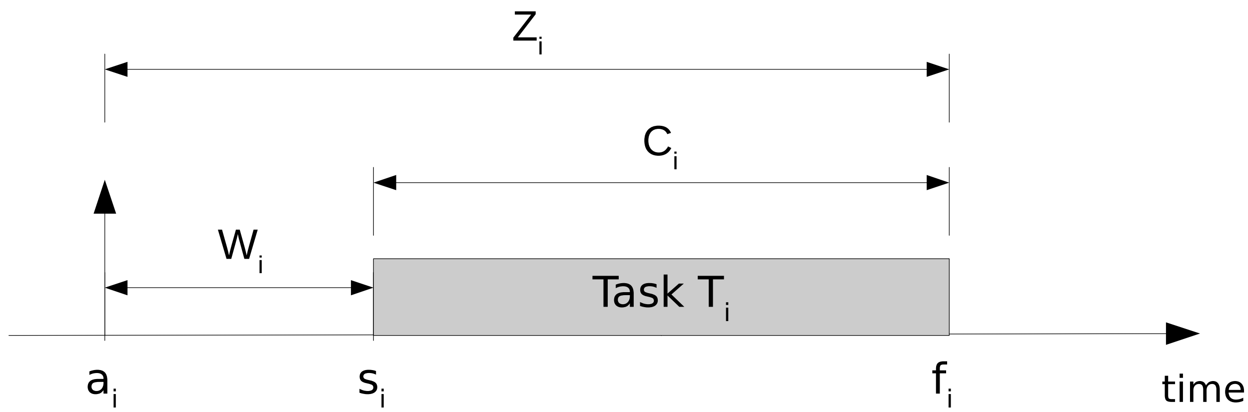
Task parameters



- ***a_i*** : Arrival time (or Request time)– time instant at which task is ready for execution (enter the ready queue)
- ***s_i***: Start time – the time instant at which execution starts
- W_i : Waiting time – time spent in the ready queue before being scheduled
- ***f_i***: Finishing time (or Completion time) – time instant at which the execution terminated

Introduction

Task parameters



- C_i: Computation time (or Burst time) – amount of time necessary to the processor to execute the task (without interruption)
- Z_i : Turnaround time – difference between finishing and arrival time ($f_i - a_i$)

Introduction

Preemption

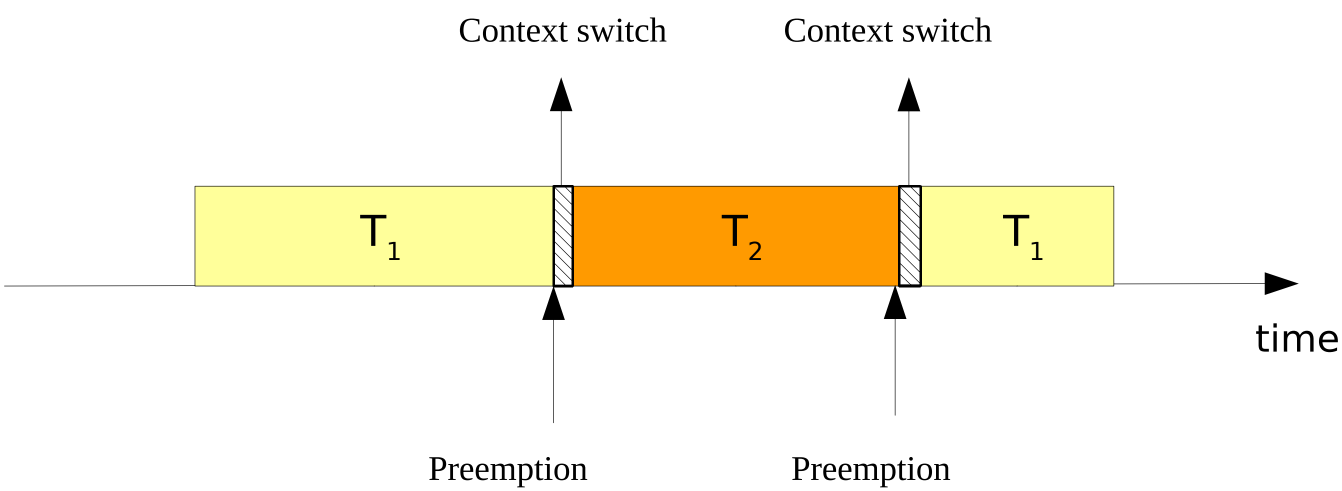
- According to the schema shown so far the tasks run in “run-to-completion” fashion, i.e., without being interrupted
- For reasons that we will explain later, operating systems typically need to have the possibility of interrupting a task execution

- Preemption : operation for suspending the execution of a task and allocate the processor to another task
- Context switch : required when a preemption is performed
  - Save the context (stack and registers) of the suspended task
  - Resume the context of the next ready task

Introduction

Preemption

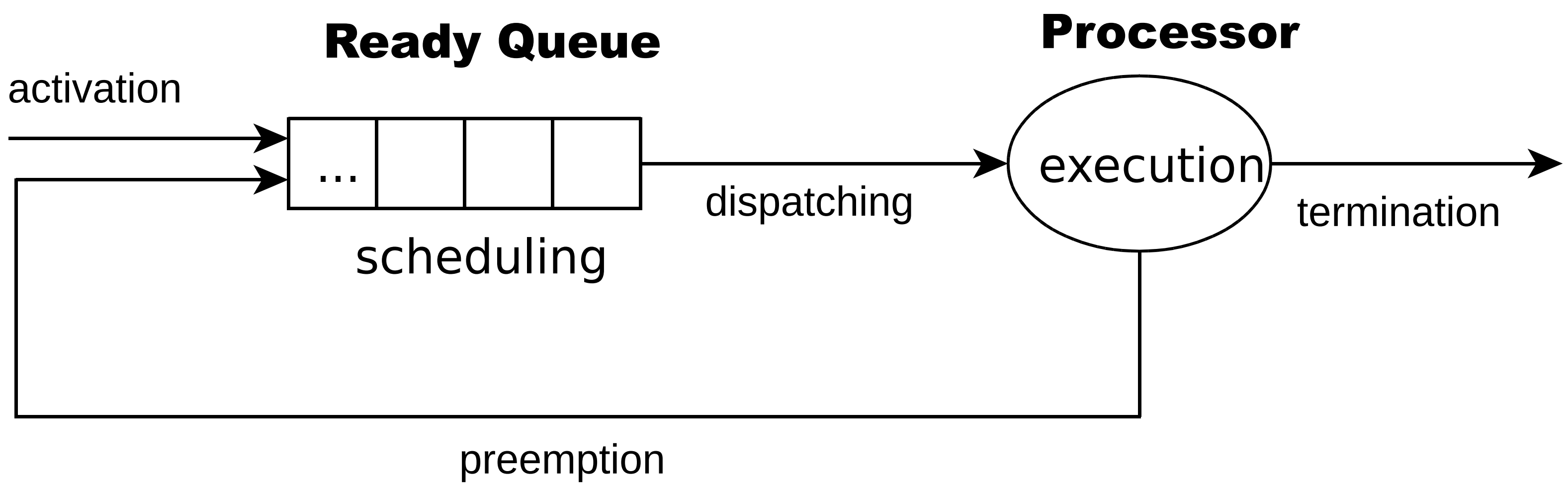
- Example
  - Task 1 runs for a while
  - Task 2 enters the system and the scheduler decides to suspend Task 1 to execute Task 2
  - The execution of Task 1 resumed later



Introduction

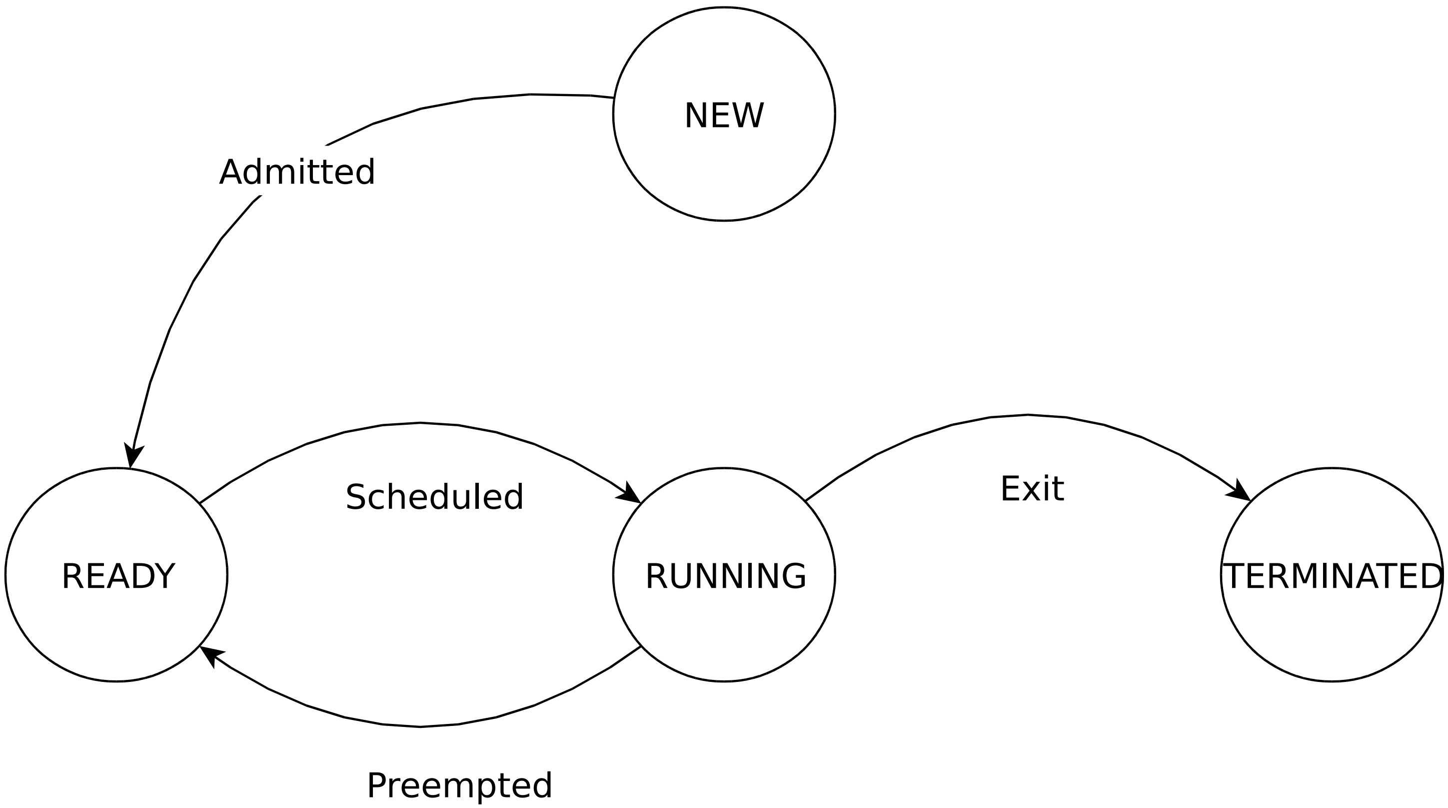
Scheduling overview with preemption

- Preempted tasks are moved back to the Ready queue



Introduction

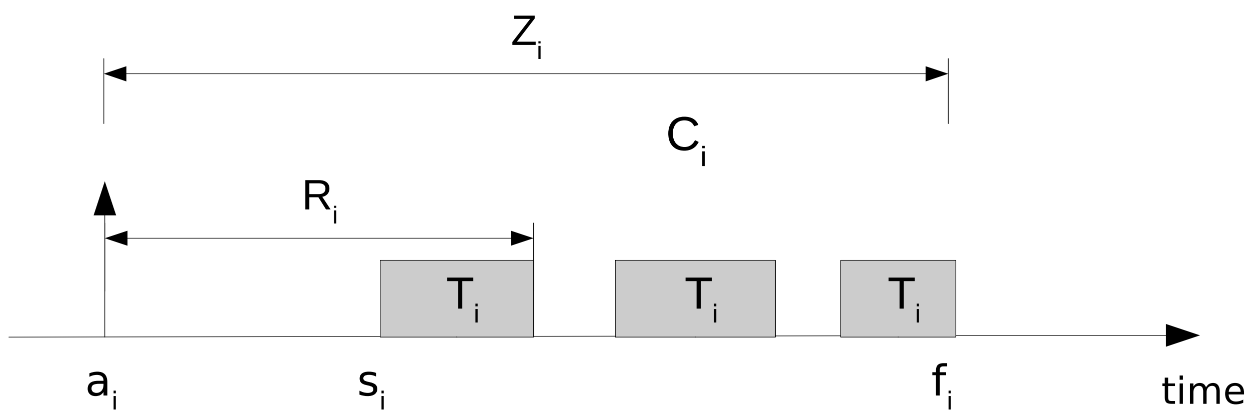
Task state diagram



Introduction

Task parameters

- If the task can be preempted, this means that we will have several runs of the same task

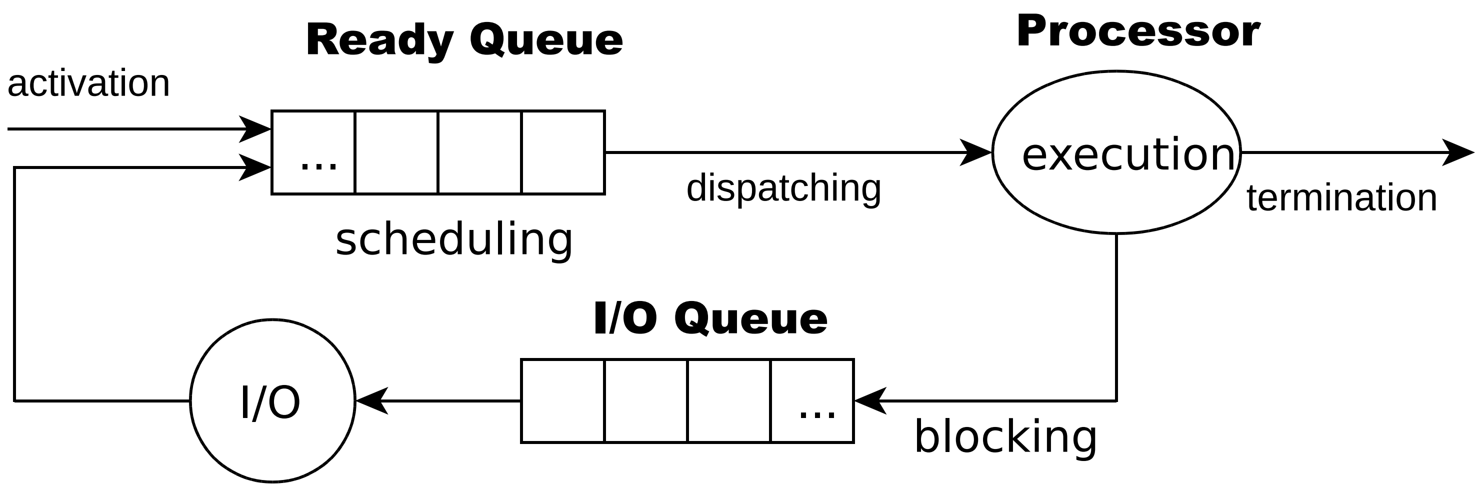


- R_i : Response time – time between arrival time and the completion of the first run of the task (first response produced)
  - In non-preemptive systems $Z_i = R_i$
- W_i : Waiting time – difference between turnaround time and computation (burst) time: (Z_i – C_i)

Introduction

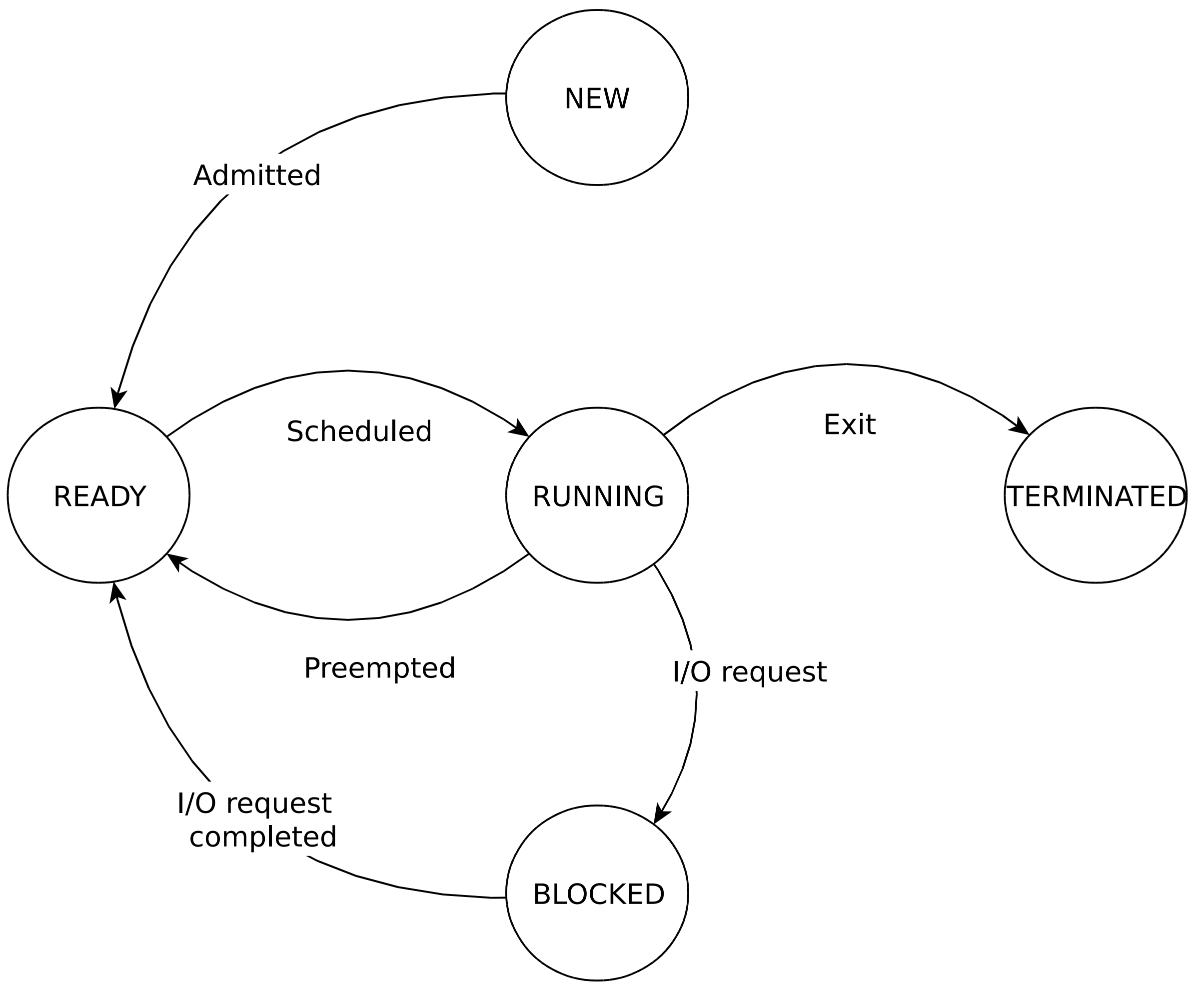
Scheduling and I/O

- Tasks may block their execution because waiting for data coming from a I/O requests (e.g., peripheral access, file read/write, etc...)
- Blocked tasks release the processor for other ready tasks



Introduction

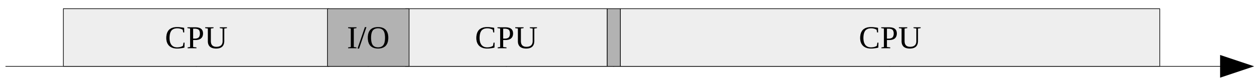
Task state diagram with I/O requests



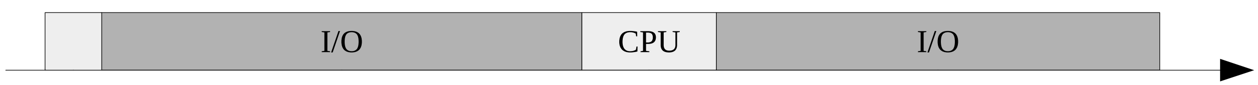
Introduction

Task boundness

- Depending on the type of operations dominating the lifetime of task, we may identify the “boundness” of a task
- CPU-bound
  - The task spent most of its time actually executing operations
  - Typical of “batch” / “background” tasks



- I/O bound
  - The task spent most of its time waiting for the completion of I/O operations
  - Typical of “interactive” / “foreground” tasks



Introduction

Scheduling metrics

- The scheduler aims at optimizing one (or more) objectives
  - We need metrics in order to evaluate the goodness of the policy
- Processor utilization : percentage of time the CPU is kept busy
- Throughput : number of tasks completing their execution per time unit
- Waiting time (avg) : average time the tasks spent in the ready queue
- Response time (avg) : average time the tasks spent in the ready queue before being served for the first time
- Fairness : do the tasks have a fair allocation of the processor?

Introduction

Problem statement

- Given a set of n tasks : $\boldsymbol{T} = \{ T_1, T_2, \ldots T_n \}$
- Given a set of processors : $\boldsymbol{P} = \{ P_1, P_2, \ldots, P_m \}$
  - $|\boldsymbol{P}| = 1 \rightarrow$ uniprocessor systems
  - $|\boldsymbol{P}| > 1 \rightarrow$ multiprocessor systems
- Given a set of resources : $\boldsymbol{R} = \{ R_1, R_2, \ldots, R_s \}$
- (Optional) Given a set of precedence relationships and constraints
- Define an ordered assignment of processors to tasks...
  - In order to optimize one or more objectives
  - ...and that constraints are not violated

- This problem has been shown to be NP-complete

Introduction

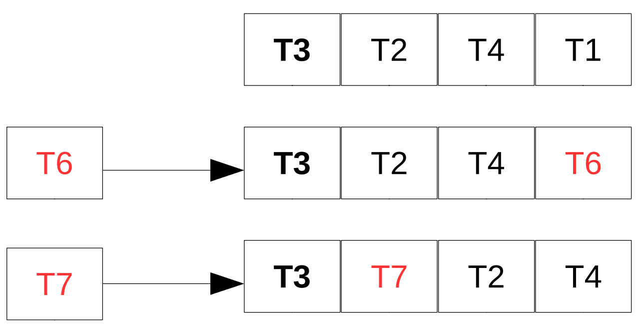
Common scheduling objectives

- Maximize *CPU utilization*, to keep it as busy as possible
- Maximize the *throughput*
- Minimize *turnaround time*
- Minimize *waiting time*
- Minimize *response time*
- Maximize the *fairness*
- ...

Introduction

Starvation

- Whatever is the objective of the scheduler, a good algorithm should guarantee that all the tasks are served
- *Starvation* is the undesirable perpetuated condition in which one (ore more) tasks cannot executed due to the lack of resources
  - e.g., a task is indefinitely postponed by the arrival of new ones



Classification of scheduling algorithms

Outline

- Preemptive vs Non-preemptive
- Static vs Dynamic
- Offline vs Online
- Optimal vs Heuristic

Classification of scheduling algorithms

Preemptive vs Non preemptive

- *Preemptive*
  - Running tasks can be interrupted at any time to allocate the processor to another active task
  - Good if we need a responsive system
  - Necessary for fairness objective

- *Non-preemptive*
  - Once started a task is executed until its completion (“run-to-completion”)
  - Scheduling decisions taken when tasks terminates
  - Good if we aim at minimizing tasks completion time
  - Negative impact on responsiveness
  - A single task my “monopolize” the processor

Classification of scheduling algorithms

Static vs Dynamic

- We can distinguish cases in which tasks parameters are known a priori, from cases in which we need to collect runtime information

- *Static*
  - Scheduling decisions are based on fixed parameters, whose values are known before task activation
  - Stronger assumptions required

- *Dynamic*
  - Scheduling decisions are based on parameters that typically changes at runtime, during the system activity
  - Need runtime feedback

Classification of scheduling algorithms

Offline vs Online

- *Offline*
  - The scheduler is executed offline on a set of known tasks (before their activation)
  - The outcome is stored into a data structure (e.g. a table) that is processed at runtime by a dispatcher
  - We need to know a priori the system workload (the set of tasks)
  - Necessary if we must provide some guarantee, by performing some preliminary check

- *Online*
  - The scheduler is executed at runtime
  - The system workload is variable, with tasks activated and terminated at random instant times

Classification of scheduling algorithms

Optimal vs Heuristic

- *Optimal*
  - The scheduler is based on an algorithm optimizing a given cost function, defined over the task set
  - The algorithm may be characterized by a not negligible complexity
- *Heuristic*
  - Algorithms based on a heuristic function
  - Tending to optimal scheduling, but without any guarantee about achieving it
  - Generally much faster that optimal algorithms

Scheduling algorithms

Outline

- First In First Out (FIFO)
- Shortest Job First (SJF)
- Shortest Remaining Time First (SRTF)
- Highest Response Ratio Next (HRRN)
- Round Robin (RR)

Scheduling algorithms

First In First Out (FIFO)

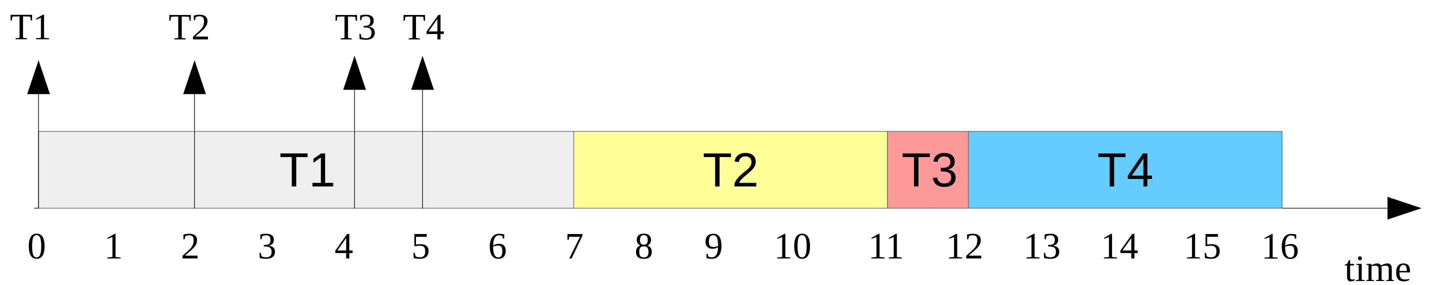
- Very simple algorithm
- Non-preemptive scheduling class
- Also known as “First Come First Served (FCFS)”
- Tasks are dispatched according to the arrival order
- When current running task terminates, the oldest one is selected

- Not good for responsiveness
  - Long tasks may monopolize the processor, penalizing short ones

First In First Out (FIFO)

- *Example*

Tasks	a	C	s	f	R
1	0	7	**0**	**7**	7
2	2	4	**7**	**11**	9
3	4	1	**11**	**12**	8
4	5	4	**12**	**16**	11



- Avg. waiting time = (0 + 5 + 7 + 7) / 4 = 4.75

  T_3 has to wait 7 time units before executing for 1
- Avg. response/turnaround time = (7 + 9 + 8 + 11) / 4 = 8.75

Scheduling algorithms

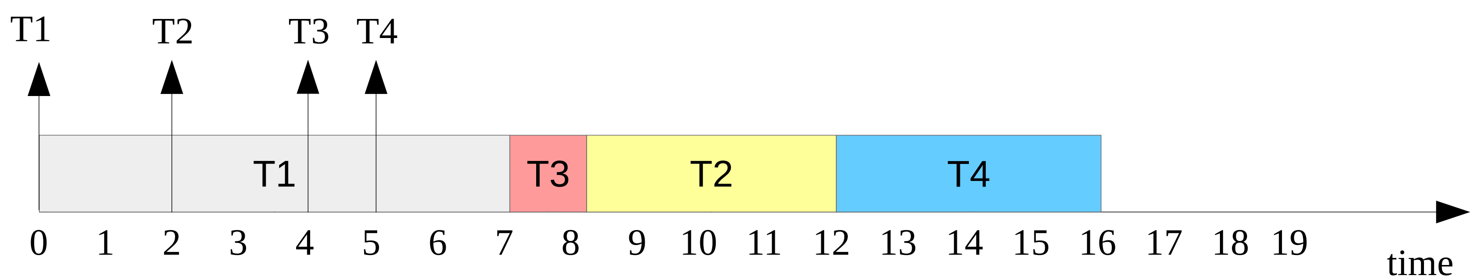
Shortest Job First (SJF)

- Also know as “Shortest Job Next (SJN)”
- Non-preemptive scheduling class
  - Tasks executed in run-to-completion mode
- Task are scheduled in ascending order of computation time (C_i)
- Optimal algorithm
  - Minimize the average waiting time of the task set
  - Useful for benchmarking
- Starvation may occur!
  - What if small tasks enter the system while a long one has not been scheduled yet?

Shortest Job First (SJF)

- *Example*

Tasks	a	C	s	f	R
1	0	7	**0**	**7**	**0**
2	2	4	**8**	**12**	**6**
3	4	1	**7**	**8**	**3**
4	5	4	**12**	**16**	**7**



- Avg. waiting time = (0 + 6 + 3 + 7) / 4 = 4
- Avg. response/turnaround time = (7 + 10 + 4 + 11) / 4 = 8

Scheduling algorithms

Shortest Remaining Time First (SRTF)

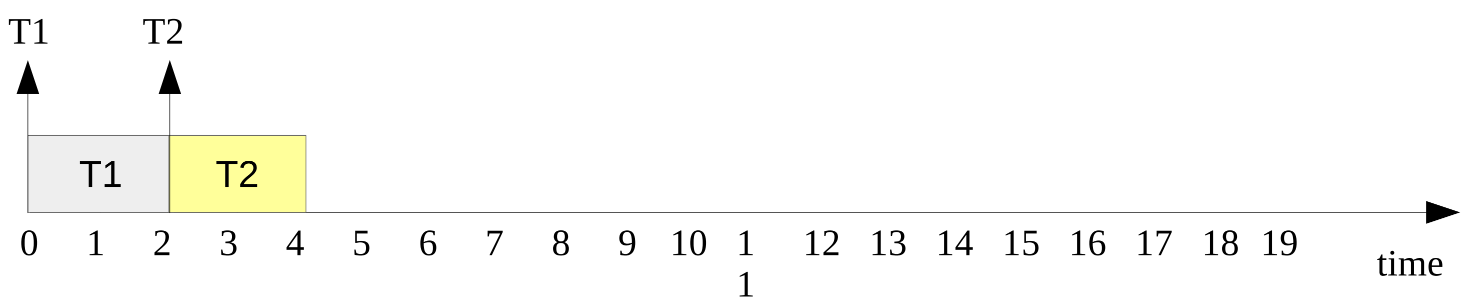
- Preemptive variant of SJF
- Preempted tasks had already spent part of the computation time, so...
- Compare the *remaining time* instead of the initial computation time C_i
- Improved responsiveness
- Anyway…. starvation may still occur!
  - We can still a end in a condition for which a task execution is indefinitely postponed because tasks with smaller C_i are continuously coming in the ready queue

Scheduling algorithms

Shortest Remaining Time First (SRTF)

- *Example*

Tasks	a	C	s	f	R	W
1	0	7	**0**			
2	2	4	**2**			
3	4	1				
4	5	4				



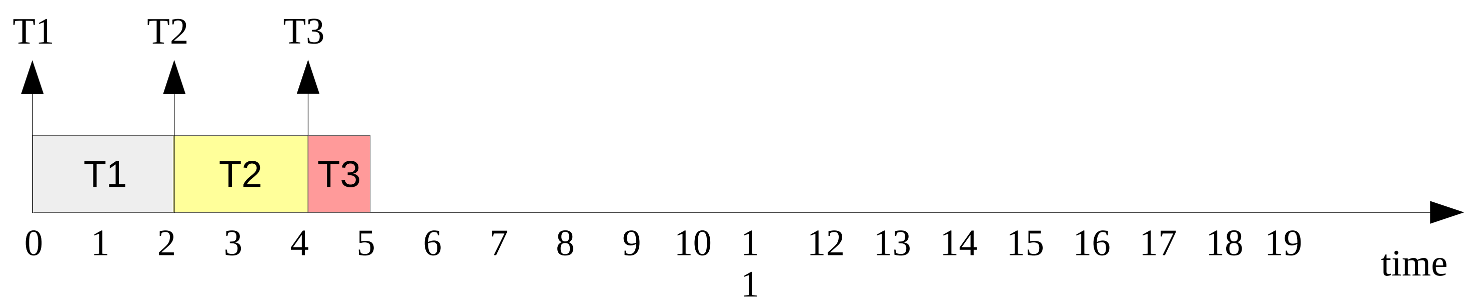
- At t_2 task T_2 is ready, $C_2 = 4$ while remaining time $RC_1 = 5$
  Task T_2 si scheduled

Scheduling algorithms

Shortest Remaining Time First (SRTF)

- *Example*

Tasks	a	C	s	f	R	W
1	0	7	**0**			
2	2	4	**2**			
3	4	1	**4**	**5**		
4	5	4				



- At t_4 task T_3 is ready, $C_3 = 1$ while remaining time $RC_1 = 5$ and $RC_2 = 2$

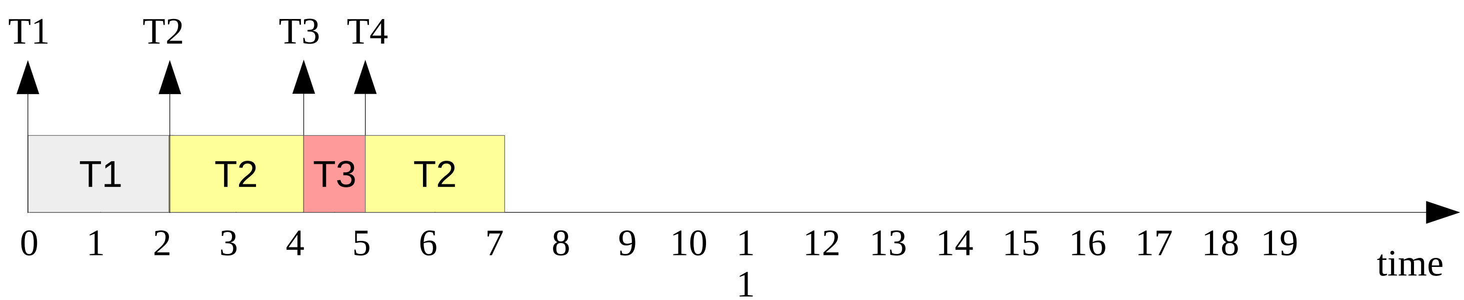
  Task T_3 is scheduled and terminates after 1 time unit

Scheduling algorithms

Shortest Remaining Time First (SRTF)

- *Example*

Tasks	a	C	s	f	R	W
1	0	7	**0**			
2	2	4	**2**	**7**		
3	4	1	**4**	**5**		
4	5	4				



→ At t_5 task T_4 is ready, C_4 = 4, RC_1 = 5, RC_2 = 2

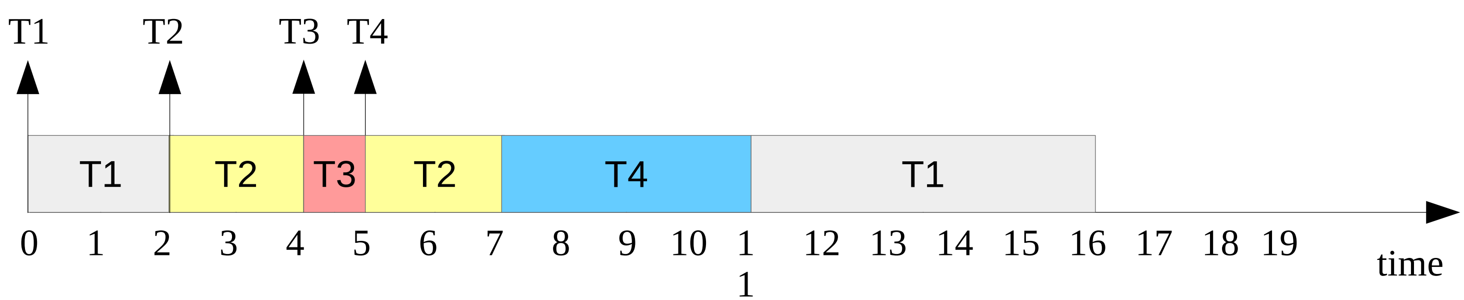
Task T_2 is scheduled and terminates after 2 time units

Scheduling algorithms

Shortest Remaining Time First (SRTF)

- *Example*

Tasks	a	C	s	f	R	W
1	0	7	**0**	**16**		
2	2	4	**2**	**7**		
3	4	1	**4**	**5**		
4	5	4	**7**	**11**		



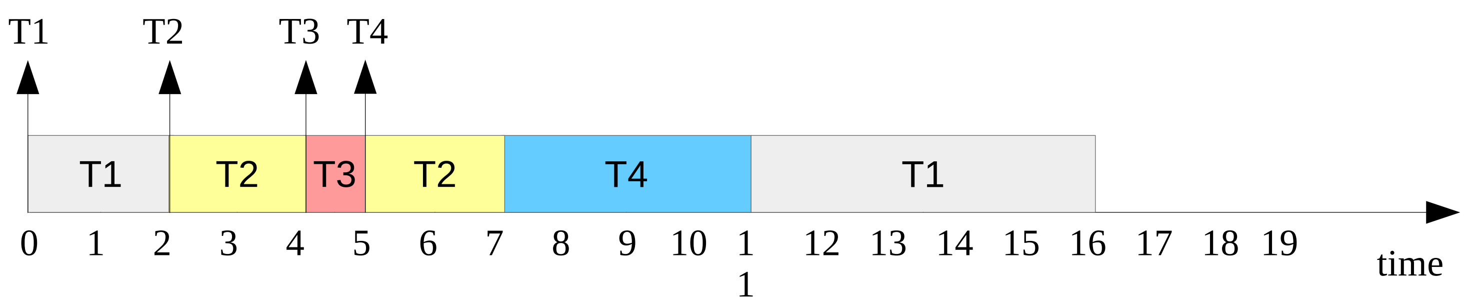
- At t_7 : $C_4 = 4$, $RC_1 = 5 \rightarrow$ Schedule T_4
- At t_{11} : T_4 terminates, T_1 is the last task left to schedule

Scheduling algorithms

Shortest Remaining Time First (SRTF)

- *Example*

Tasks	a	C	s	f	R	W	Z
1	0	7	**0**	**16**	**2**	**9**	**16**
2	2	4	**2**	**7**	**2**	**1**	**5**
3	4	1	**4**	**5**	**1**	**0**	**1**
4	5	4	**7**	**11**	**6**	**2**	**6**



- Avg. waiting time = (9 + 1 + 0 + 2) / 4 = 3.0
- Avg. response time = (2 + 2 + 1 + 6) / 4 = **2.75**
- Avg. turnaround time = (16 + 5 + 1 + 6) / 4 = 7

Scheduling algorithms

Highest Response Ratio Next (HRRN)

- Non-preemptive scheduling class
- Evolution of Shortest Job First
- Select the task with the highest Response Ratio:

$$RR_i = \frac{W_i + C_i}{C_i}$$

- Short tasks (small computation time C) OK but...
- Waiting time (W) must be taken into account
- This has the effect of preventing starvation

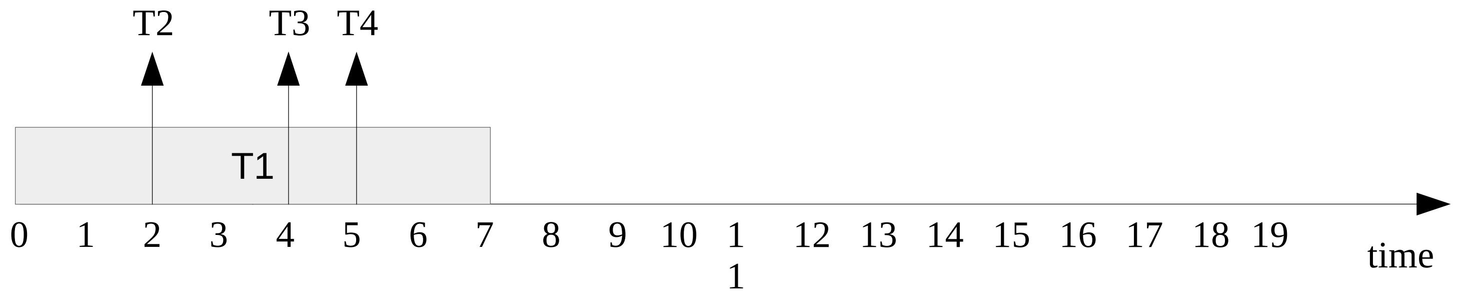
Scheduling algorithms

Highest Response Ratio Next (HRRN)

- *Example*

Tasks	a	C	s	W	RR
1	0	7	0	0	**1**
2	2	4			
3	4	1			
4	5	4			

- T_1 is immediately scheduled, meanwhile… other three tasks enter the ready queue



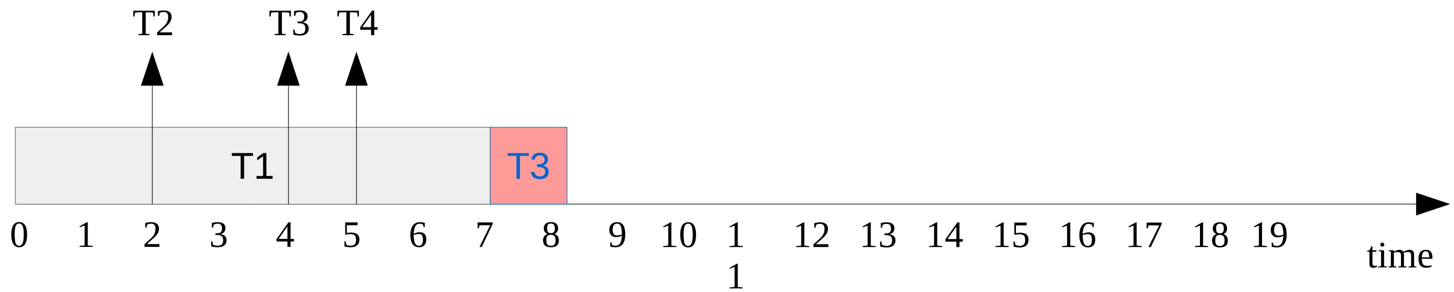
Scheduling algorithms

Highest Response Ratio Next (HRRN)

- *Example*

Tasks	a	C	s	W	RR
1	0	7	0	0	**1**
2	2	4	...	5	**2.25**
3	4	1	...	3	**4.00**
4	5	4	...	2	**1.50**

- Once T_1 terminates (t=7), we compute the response ratio and pick T_3, which has the highest ratio



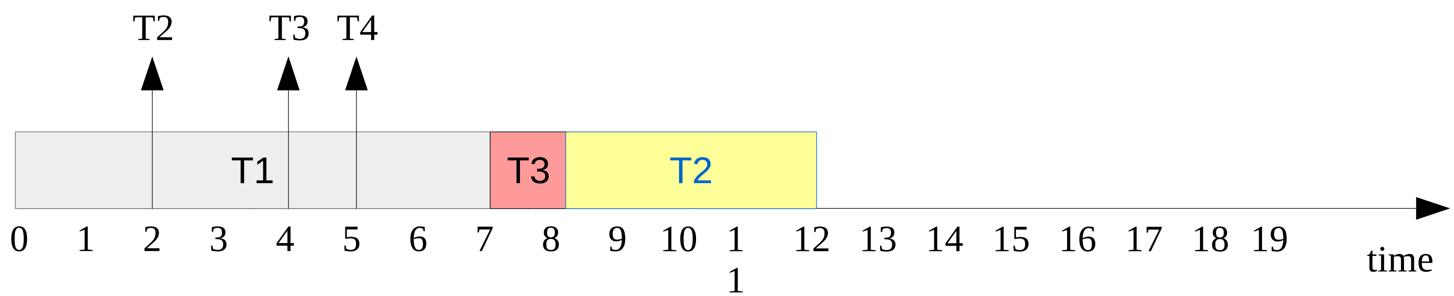
Scheduling algorithms

Highest Response Ratio Next (HRRN)

- *Example*

Tasks	a	C	s	W	RR
1	0	7	0	0	-
2	2	4	...	6	**2.50**
3	4	1	7	3	-
4	5	4	...	3	**1.75**

- Once T_3 terminates (t=8), we compute the HRR and pick T_2, which has the highest ratio

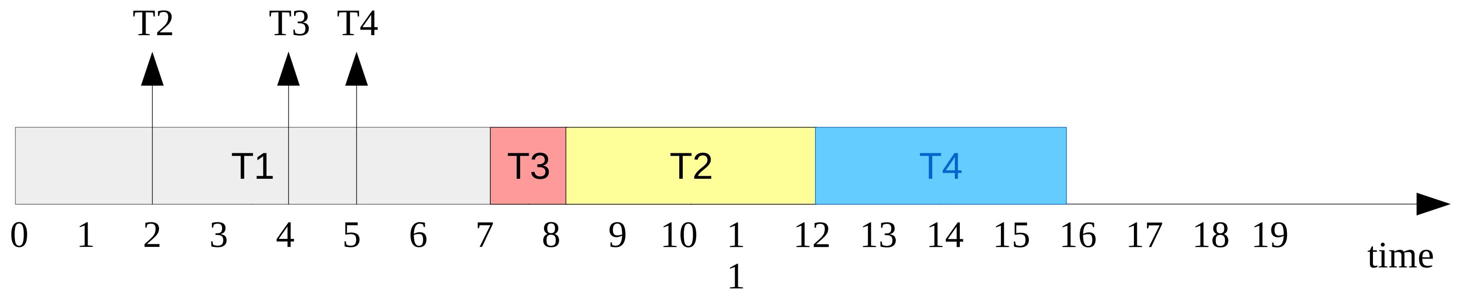


Highest Response Ratio Next (HRRN)

- *Example*

Tasks	a	C	s	W	RR
1	0	7	0	0	-
2	2	4	8	6	-
3	4	1	7	3	-
4	5	4	12	7	-

- T_4 is the last task in the queue and will start once T_2 finishes at t=12

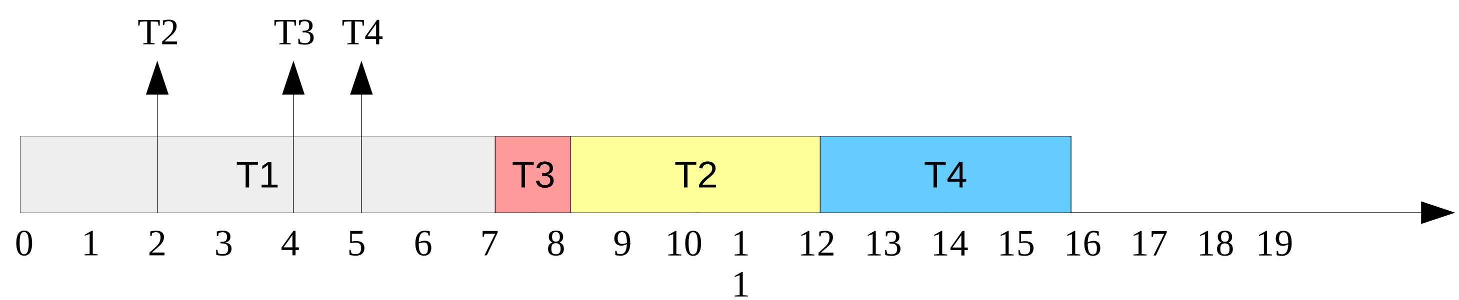


Scheduling algorithms

Highest Response Ratio Next (HRRN)

- *Example*

Tasks	a	C	s	W	f
1	0	7	0	0	**7**
2	2	4	8	6	**12**
3	4	1	7	3	**8**
4	5	4	12	7	**16**



- Avg. waiting time = (0 + 6 + 3 + 7) / 4 = 4
- Avg. response/turnaround time = (7 + 10 + 4 + 11) / 4 = 8

Scheduling algorithms

Round Robin (RR)

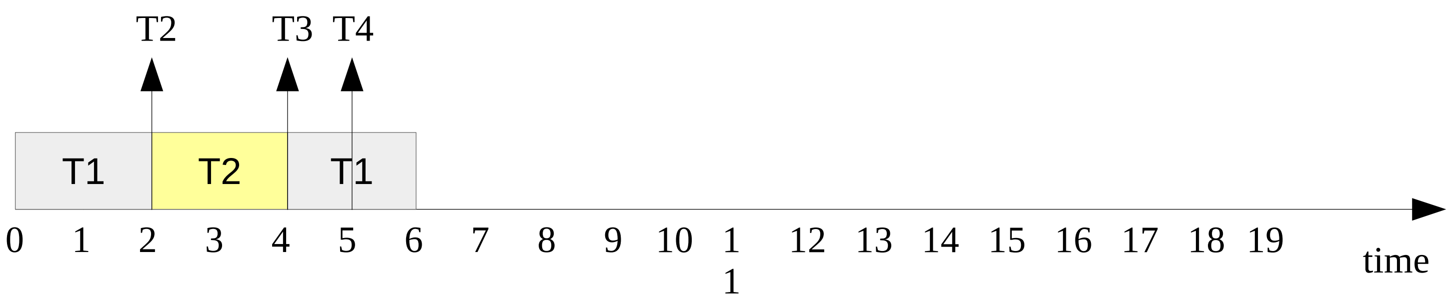
- Preemptive scheduling class
- *Time sharing* approach
  - Task are scheduled to process for a given time quantum (q) or (time slice)
  - When the time quantum expires the task is preempted and moved back to the ready queue
- For a known number of tasks n, the maximum waiting time is bound to $(n - 1) * q$
- Good for managing a *fair* allocation of the processor
- Good for managing the system *responsiveness*
- Turn-around time usually worst than SJF
- No starvation

Scheduling algorithms

Round Robin (RR)

- *Example 1*
  - Time quantum *q=2*

Tasks	a	C	s	f	R	W
1	0	7	**0**			
2	2	4	**2**			
3	4	1				
4	5	4				



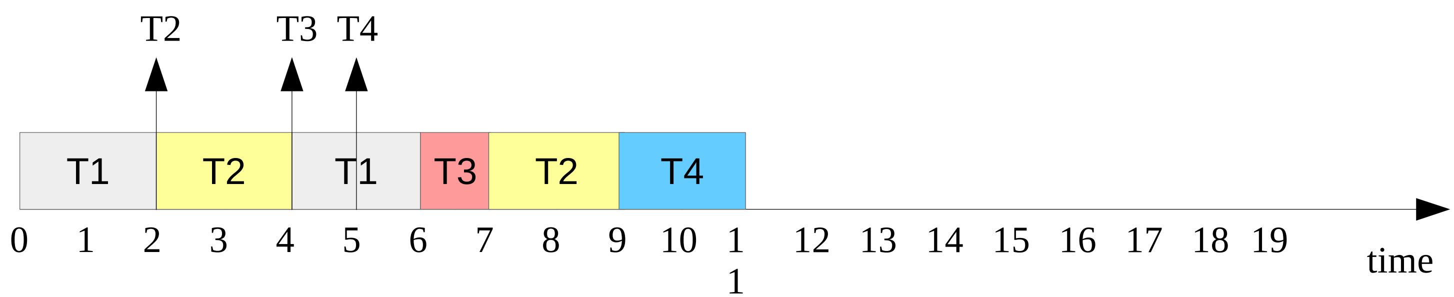
  - t_0 : Ready queue = { **T1** }
  - t_2 : Ready queue = { **T2**, T1 }
  - t_4 : Ready queue = { **T1**, T3, T2 }

Scheduling algorithms

Round Robin (RR)

- *Example 1*
  - Time quantum *q=2*

Tasks	a	C	s	f	R	W
1	0	7	**0**			
2	2	4	**2**	**9**		
3	4	1	**6**	**7**		
4	5	4	**9**			



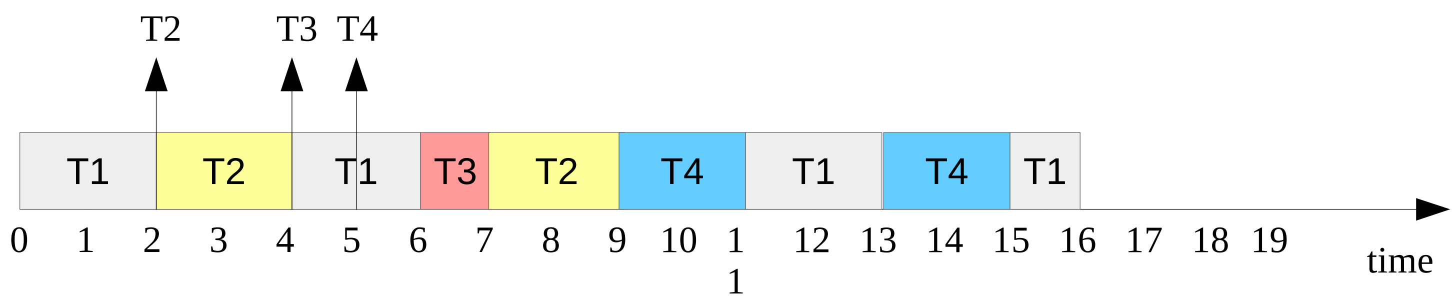
- t_6 : Ready queue = { **T3**, T2, T4, T1 }
- t_7 : Ready queue = { **T2**, T4, T1 }
- t_9 : Ready queue = { **T4**, T1 }

Scheduling algorithms

Round Robin (RR)

- *Example 1*
  - Time quantum $q=2$

Tasks	a	C	s	f	R	W
1	0	7	**0**	**16**		
2	2	4	**2**	**9**		
3	4	1	**6**	**7**		
4	5	4	**9**	**15**		



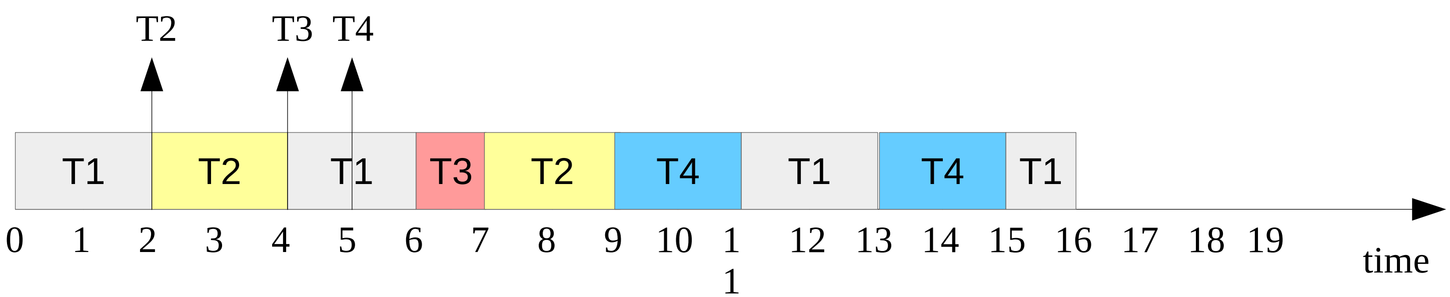
- t_{11} : Ready queue = { **T1**, T4 }
- t_{13} : Ready queue = { **T4**, T1 }
- t_{15} : Ready queue = { **T1** }

Scheduling algorithms

Round Robin (RR)

- *Example 1*
  - Time quantum *q=2*

Tasks	a	C	s	f	R	W	Z
1	0	7	**0**	**16**	**2**	**9**	**16**
2	2	4	**2**	**9**	**2**	**3**	**7**
3	4	1	**6**	**7**	**3**	**2**	**3**
4	5	4	**9**	**15**	**6**	**6**	**10**



- Avg. waiting time = (9 + 3 + 2 + 6) / 4 = **5**
- Avg response time = (2 + 2 + 3 + 6) / 4 = **4.33**
- Avg. turnaround time = (16 + 7 + 3 + 10) / 4 = **9**

Round Robin (RR)

- *Example 1*
  - Time quantum *q=2*

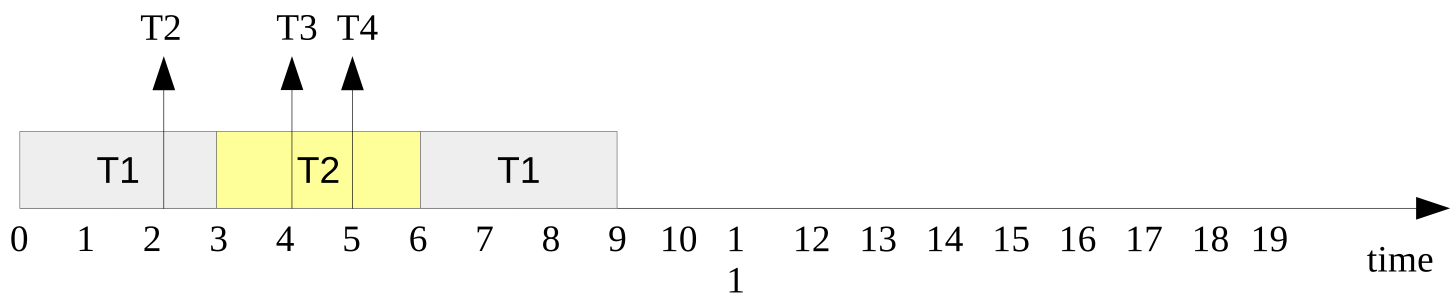
Tasks	a	C	s	f	R	W	Z
1	0	7	**0**	**16**	**2**	**9**	**16**
2	2	4	**2**	**9**	**2**	**3**	**7**
3	4	1	**6**	**7**	**3**	**2**	**3**
4	5	4	**9**	**15**	**6**	**6**	**10**

Let’s see what happens if we change the time quantum for example by setting **q=3**

Round Robin (RR)

- *Example 2*
  - Time quantum *q=3*

Tasks	a	C	s	f	R	W
1	0	7	**0**		**3**	
2	2	4	**3**		**4**	
3	4	1				
4	5	4				



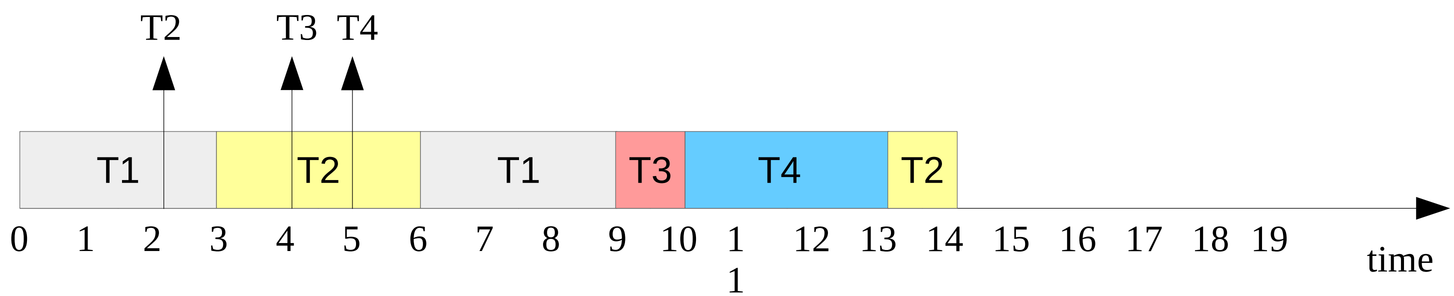
- t_0 : Ready queue = { **T1** }
- t_3 : Ready queue = { **T2**, T1 }
- t_6 : Ready queue = { **T1**, T3, T4, T2 }

Scheduling algorithms

Round Robin (RR)

- *Example 2*
  - Time quantum $q=3$

Tasks	a	C	s	f	R	W
1	0	7	**0**		**3**	
2	2	4	**3**		**4**	
3	4	1	**9**	**10**	**6**	
4	5	4	**10**		**8**	

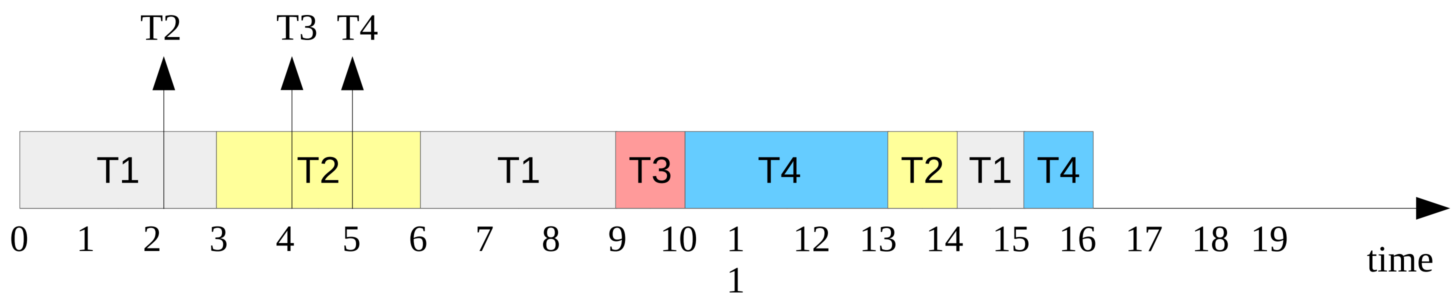


- t_9 : Ready queue = { **T3**, T4, T2, T1 }
- t_{10} : Ready queue = { **T4**, T2, T1 }
- t_{13} : Ready queue = { **T2**, T1, T4 }

Round Robin (RR)

- *Example 2*
  - Time quantum *q=3*

Tasks	a	C	s	f	R	W
1	0	7	**0**	**15**	**3**	
2	2	4	**3**	**14**	**4**	
3	4	1	**9**	**10**	**6**	
4	5	4	**10**	**15**	**8**	



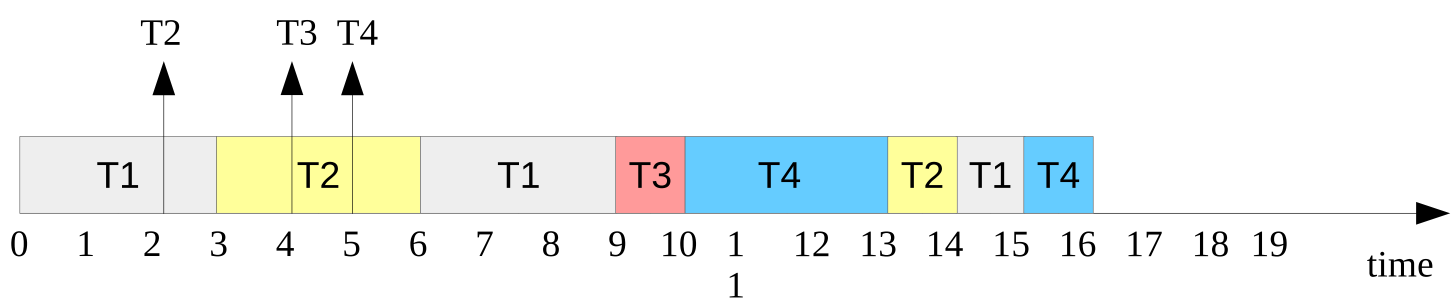
- t_{14} : Ready queue = { **T1**, T4 }
- t_{15} : Ready queue = { **T4** }
- t_{16} : Ready queue = { }

Scheduling algorithms

Round Robin (RR)

- *Example 2*
  - Time quantum $q=3$

Tasks	a	C	s	f	R	W	Z
1	0	7	**0**	**15**	**3**	**8**	**15**
2	2	4	**3**	**14**	**4**	**8**	**12**
3	4	1	**9**	**10**	**6**	**5**	**6**
4	5	4	**10**	**16**	**8**	**7**	**11**



- Avg. waiting time = (8 + 8 + 5 + 7) / 4 = **7**
- Avg response time = (3 + 4 + 6 + 8) / 4 = **5.25**
- Avg. turnaround time = (15 + 12 + 6 + 10) / 4 = **11**

For q=3 the responsiveness of the system results to be decreased

Scheduling algorithms

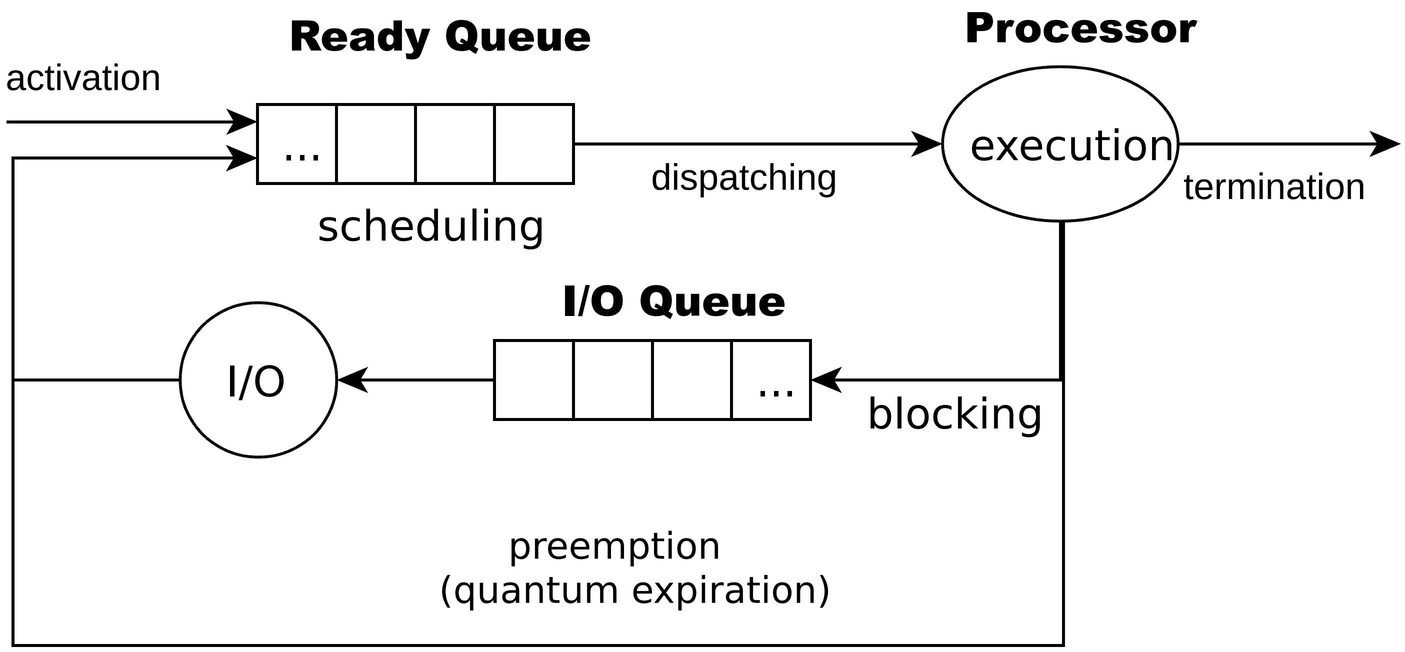
Time sharing

- The length of the time quantum is a critical choice
- “*Long*” quantum :
  - Scheduler performance → FIFO
  - Favor CPU-bound tasks
  - Less context switches → lower overhead
- “*Short*” quantum :
  - Reduced average waiting time
  - Good for responsiveness
  - Easier to achieve a fair assignment of the processor
  - Favor I/O-bound tasks
  - More context switches → higher overhead

Scheduling algorithms

Time sharing

- Scheduling with I/O occurrences and time sharing



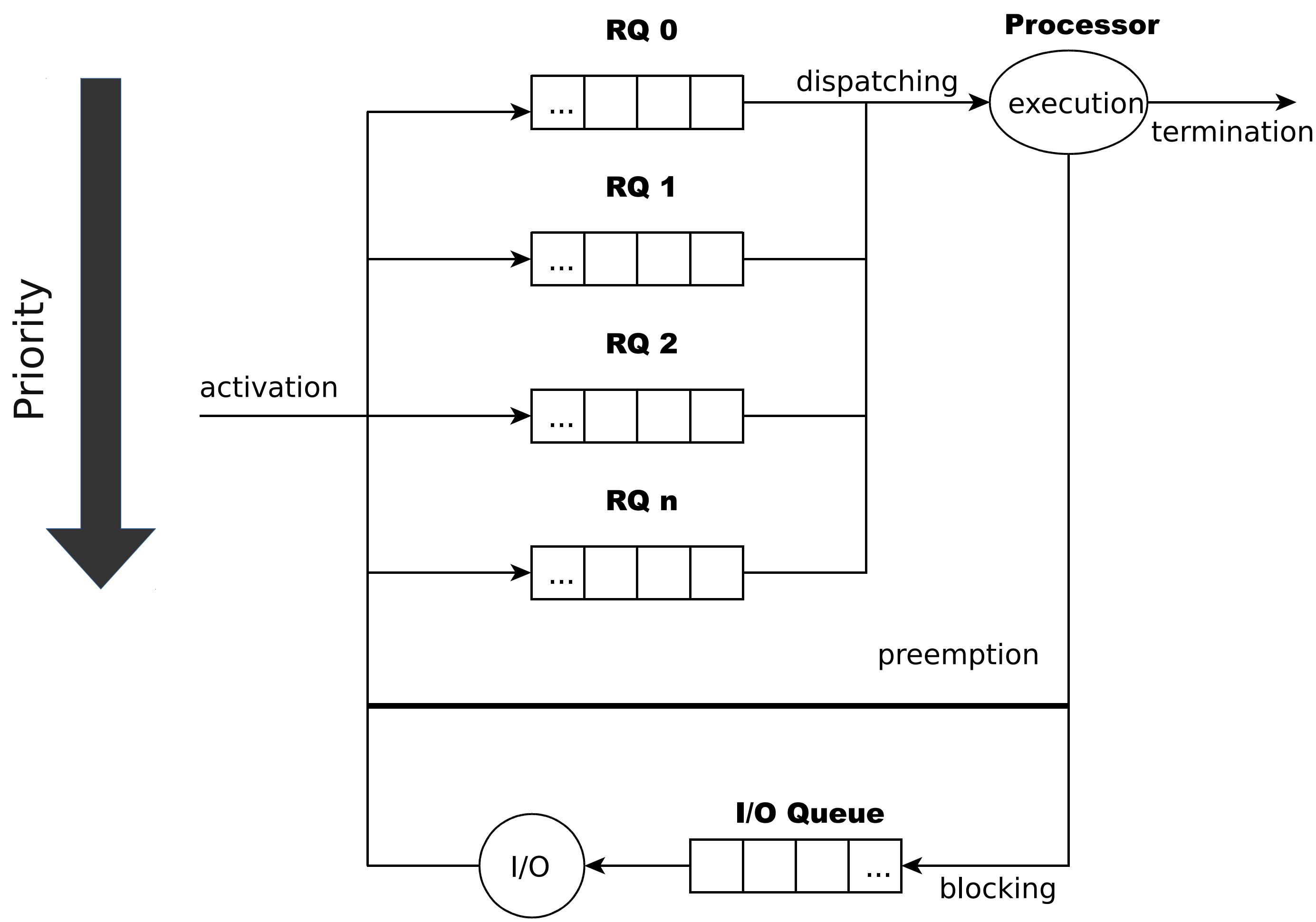
Priority based scheduling

Priority based scheduling

- *Priority* is a task parameter through which we can specify the importance of a task
  - **Fixed** or **dynamic** (runtime variable)
- Conventionally, the priority is expressed with an integer value
  - The lower the integer value the higher the priority
  - The higher the integer value the lower the priority
- Priority-based scheduler takes into account this parameter...
  - To drive the **dispatching time** (which task to schedule first)
  - To define the **length of the time quantum** assigned to each task
- Priority-based scheduler typically need to manage multiple ready queues
  - Each ready queue is associated to a different priority level

Priority based scheduling

Multi-level Queue Scheduling



Priority based scheduling

Multi-level Queue Scheduling

- For each queue we can specify a different scheduling algorithm (e.g., RR or FCFS)
- The first task to schedule is picked from the topmost non-empty queue (highest priority)
- Tasks cannot be moved from one ready queue to another
- Possible to assign a different time quantum per queue
  - *Foreground (interactive)* tasks and *background (batch)* tasks use CPU time differently
  - We may give high priority to interactive tasks
- Risk of **starvation**!
  - While highest priority queues are populated by new tasks, the scheduling of tasks in lower priority queues is delayed

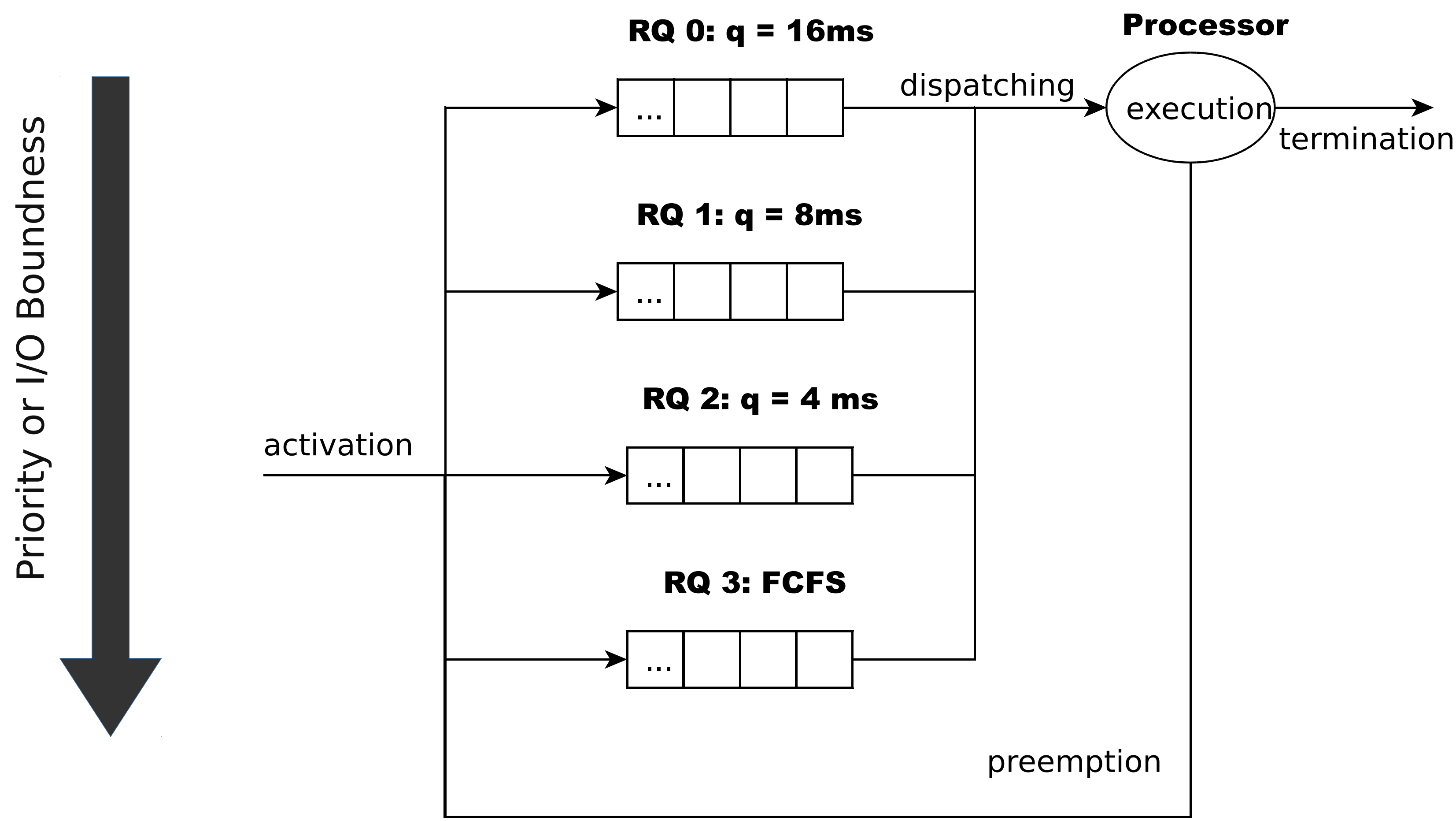
Priority based scheduling

Multi-level Feedback Queue Scheduling

- How to determine if a task is interactive or batch?
- How to practically infer at runtime if a task is CPU-bound or I/O-bound?
- Most of the scheduling algorithm previously introduced rely on information known “a priori”
  - How to know the *computation time* C_i in advance?
  - Often unrealistic assumptions
- Practical implementations need runtime feedback about how the scheduled tasks use the processor
- The Multi-level Feedback Queue Scheduling aims at overcome the limitations above

Priority based scheduling

Multi-level Feedback Queue Scheduling



Priority based scheduling

Multi-level Feedback Queue Scheduling

- A quantum length is associated to each queue
- A task enters a queue, depending on the priority
- Tasks are moved among queues
  - Priority is dynamic no longer fixed
- When scheduled, if the task does not terminate in the time quantum allocated, it is moved to another queue (with a different quantum length)
  - CPU-bound (batch) tasks are progressively moved in queues with longer time quantum
  - I/O-bound (interactive) tasks are progressively moved in queue with shorter time quantum

Priority based scheduling

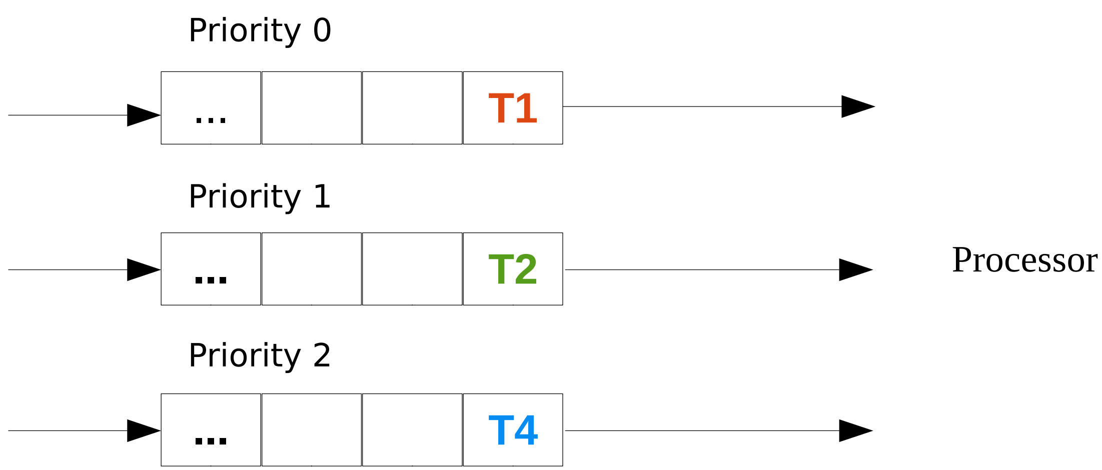
Multi-level Feedback Queue Scheduling

- What about starvation?
- We can exploit the possibility of moving the task among the queues (i.e., changing the priority at runtime)
- **Aging**
  - The priority of the task is increased as long as it spend time in ready queues (it gets “older”)
  - Prevent a task from being indefinitely postponed by new coming higher priority tasks

Priority based scheduling

Aging

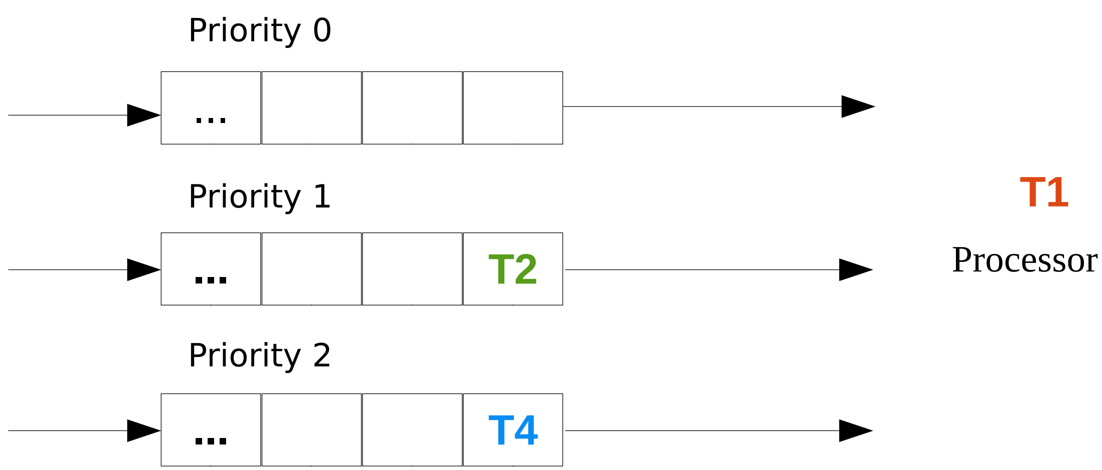
- Example
  - Three tasks at different priorities in the ready queues



Priority based scheduling

Aging

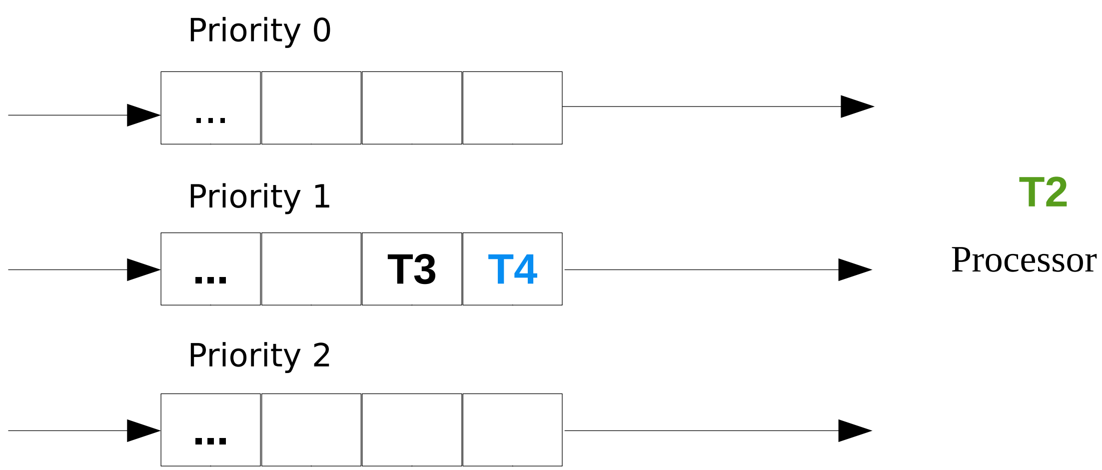
- Example
  - T1 is cheduled, so the priority 0 queue becomes empty



Priority based scheduling

Aging

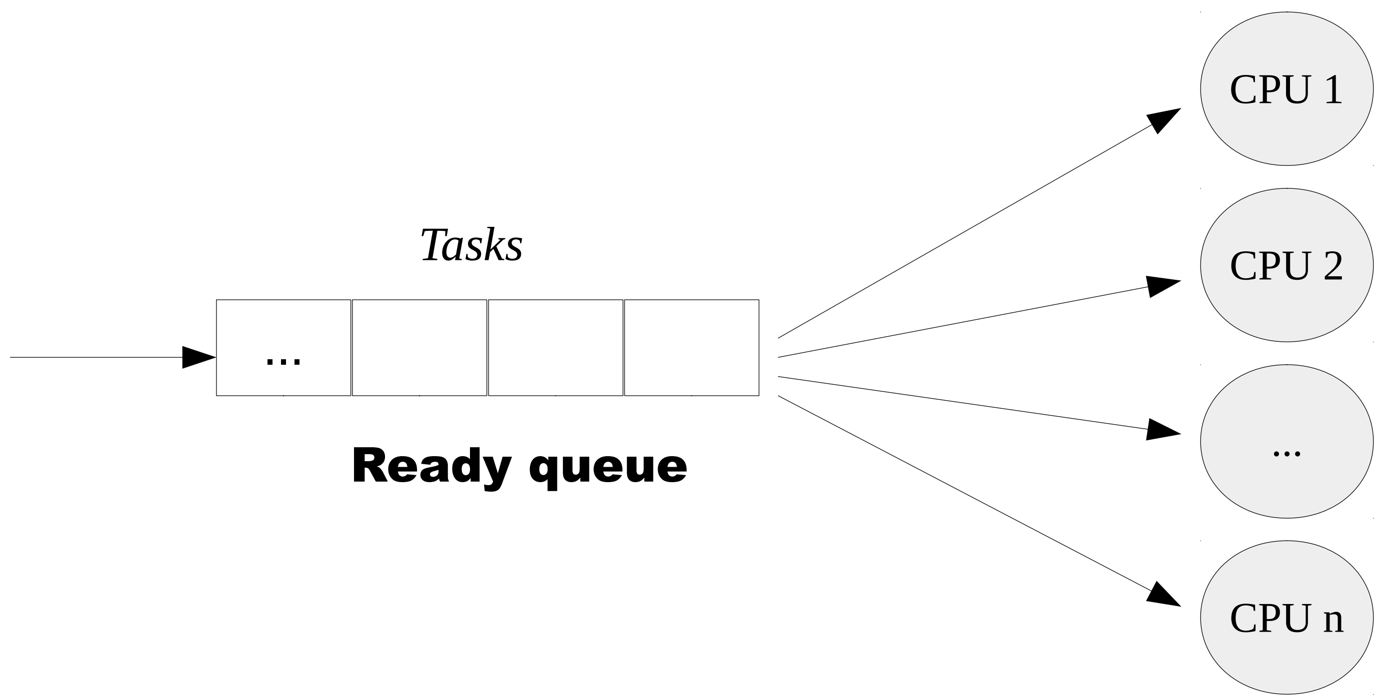
- Example
  - T2 is scheduled
  - T4 “ages” and its priority is increased to 1
  - T3 (priority 1) arrives, but this does not prevent T4 from running



Multi-processor scheduling

Problem overview

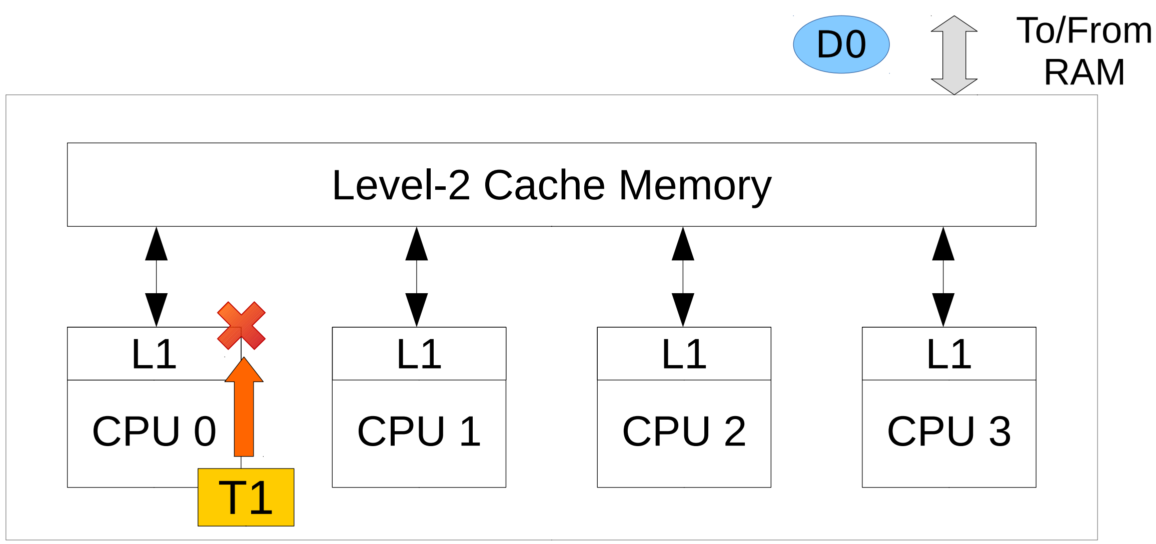
- The scheduler must choose not only the task to execute, but also to **which** processor dispatch it
- This introduces further aspect to consider....



Multi-processor scheduling

Processor affinity

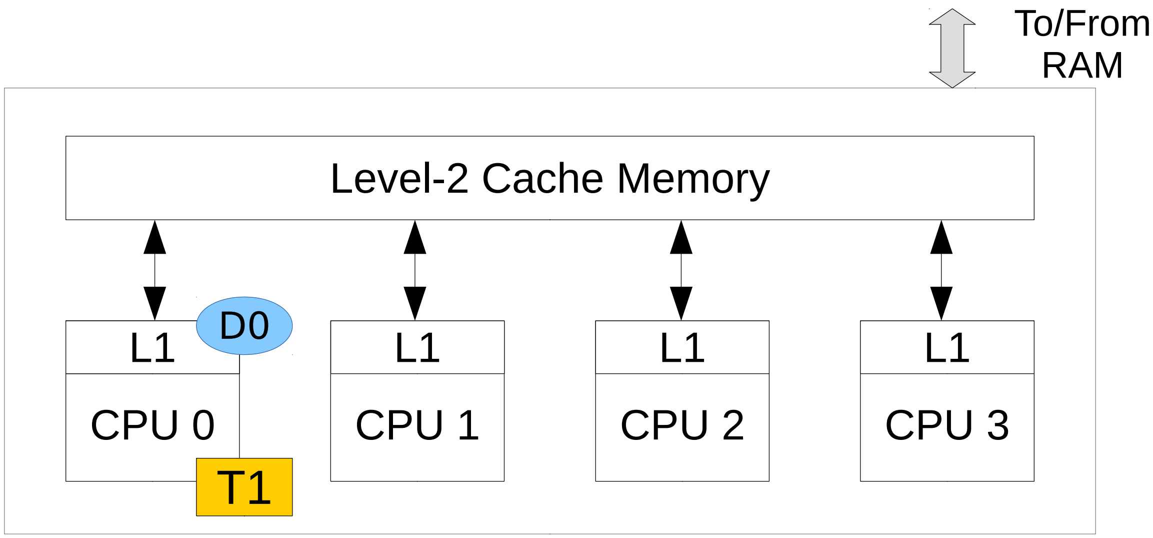
- Cache memory in modern multi-core processor dramatically affect performance
- Data should stay as closer to the CPU as possible
  - Less CPU cycles to complete load/store



Multi-processor scheduling

Processor affinity

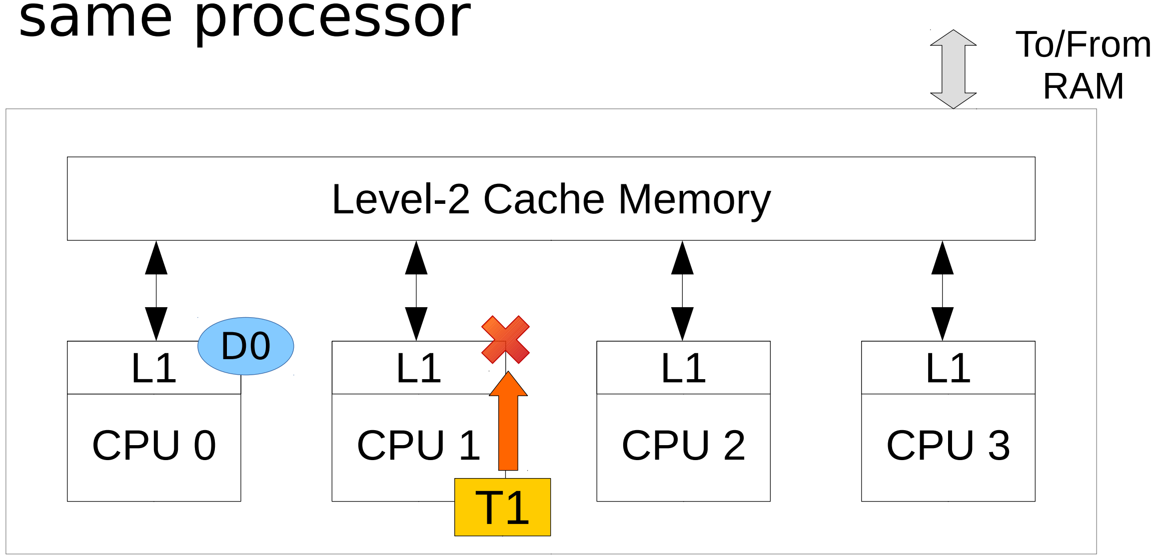
- First time a task looks for a data in L1-cache a “cache missed” is experienced
  - Lot of CPU cycles lost, waiting for the data being retrieved from higher memory levels
- Once data is retrieved we want it to remain in L1-cache



Multi-processor scheduling

Processor affinity

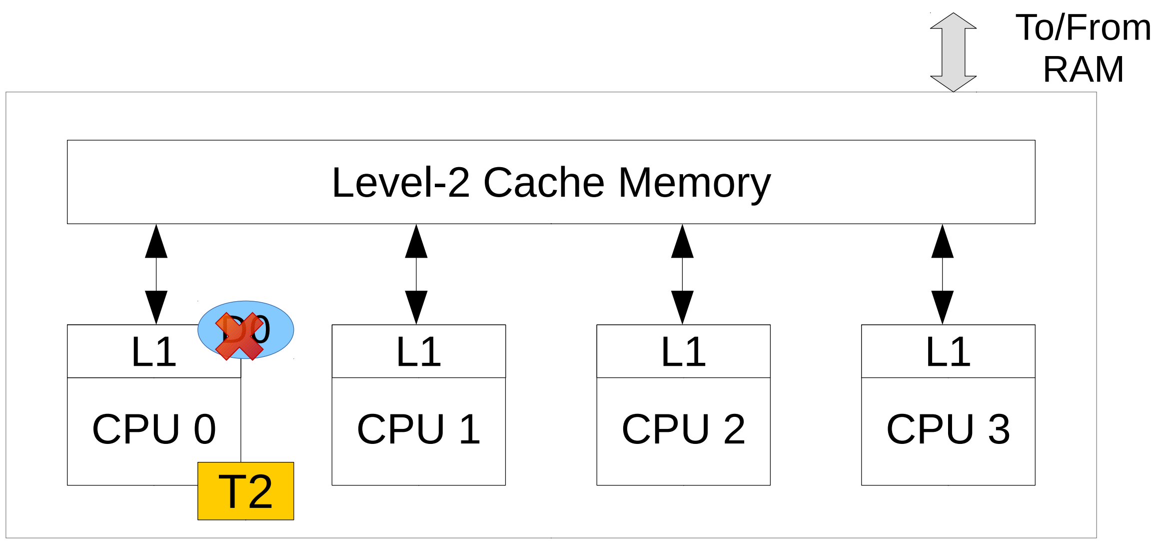
- If (in a preemptive scheduler) the next run of a task is scheduled on a different processor, new L1-cache misses are experienced
  - Loss of performance
- Processor affinity should drive the algorithm to reschedule the task on the same processor



Multi-processor scheduling

Processor affinity

- If a different task is scheduled on the processor running task T_1 (CPU 0), the cache is invalidated
  - Once task T_1 is rescheduled in CPU 0, cache misses are experienced again
- The ordering of the scheduling choices should take this into account



Multi-processor scheduling

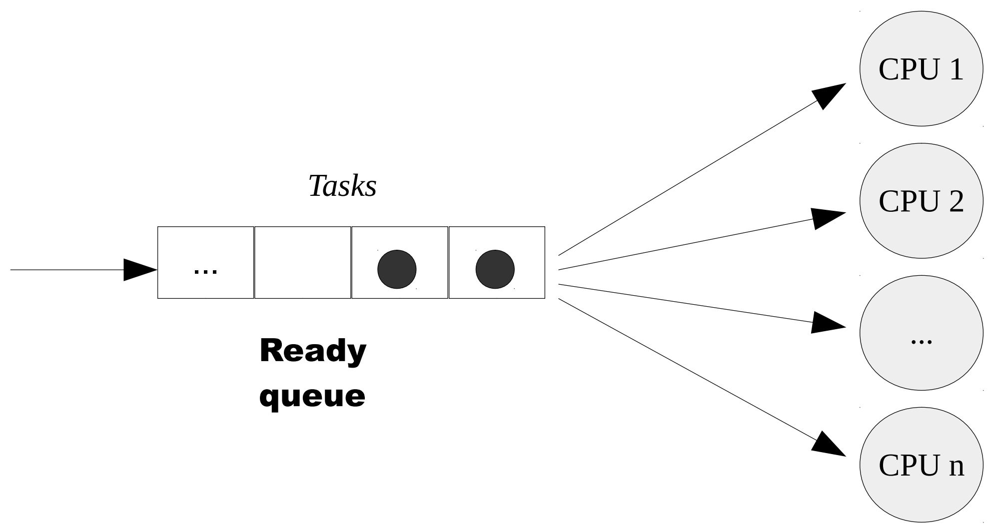
Scheduler design

- Single queue vs Multiple queues
- Single scheduler vs Multiple per-processor schedulers

Multi-processor scheduling

Single queue vs Multiple queues

- *Single queue*
  - All the ready tasks waits in the same global queue



Multi-processor scheduling

Single queue vs Multiple queues

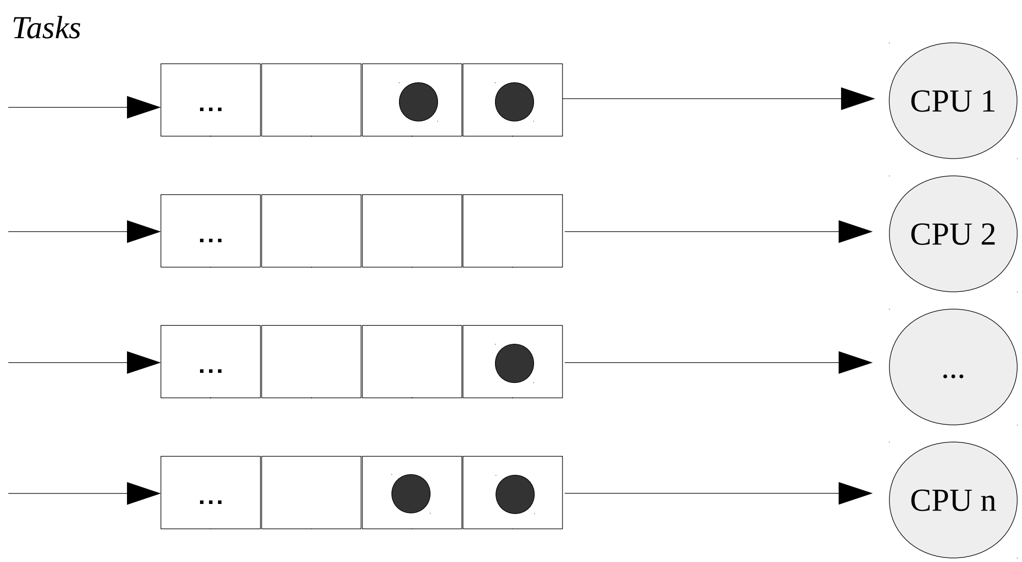
- *Single queue*
  - All the ready tasks waits in the same global queue
  - Simple design
  - Good for fairness
  - Good for managing CPU utilization
  - Scalability limitations

    Scheduling code can run in whatever processor (it may require a lock to safely access the ready queue)

Multi-processor scheduling

Single queue vs Multiple queues

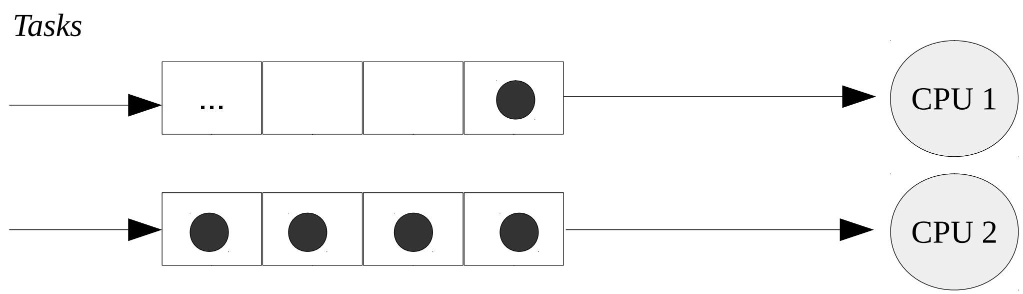
- *Multiple queues*
  - A ready queue for each processor



Multi-processor scheduling

Single queue vs Multiple queues

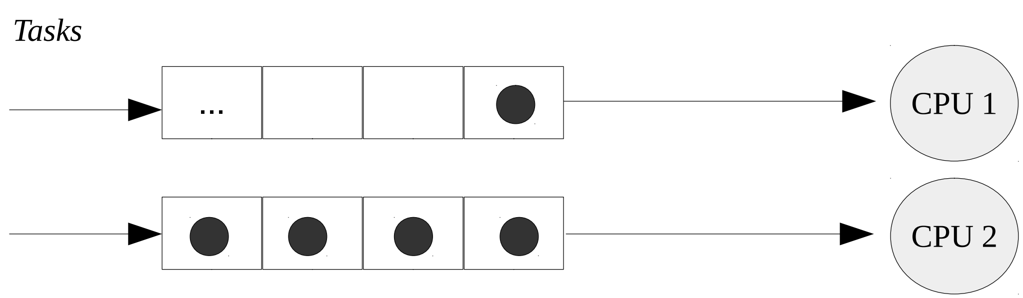
- *Multiple queues*
  - A ready queue for each processor
  - More scalable approach
  - Potentially more overhead (more data structures to handle)
  - Easier to exploit data locality (processor affinity)
  - Possibility of adopting a single global scheduler or per-CPU schedulers
  - Need of **load balancing**



Multi-processor scheduling

Load balancing

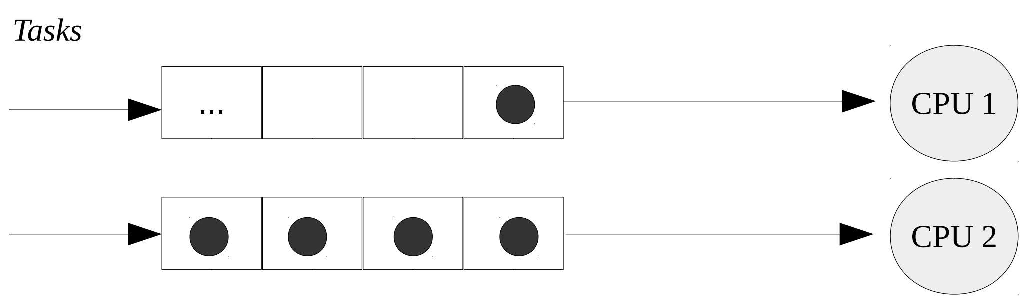
- Unbalanced ready queues have a negative impact from several point of views
- CPU utilization
  - Processors may be idle (due to empty queue), while other queues have waiting tasks
- Performance
  - Waiting times and response times can be reduced by moving the task in a queue with a lower number of tasks



Multi-processor scheduling

Load balancing

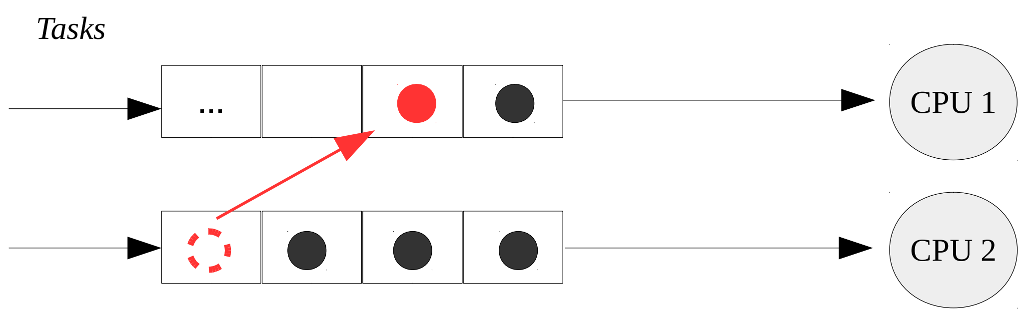
- Unbalanced ready queues have a negative impact on many aspects...
- Thermal management
  - By balancing the activity of the processors we can level the temperature distribution
  - Temperature has a direct impact on power consumption, reliability and system lifetime



Multi-processor scheduling

Task migration

- Load balancing is typically performed via task migration
  - Task are moved more to less loaded ready queue
  - Implications on processor/cache affinity

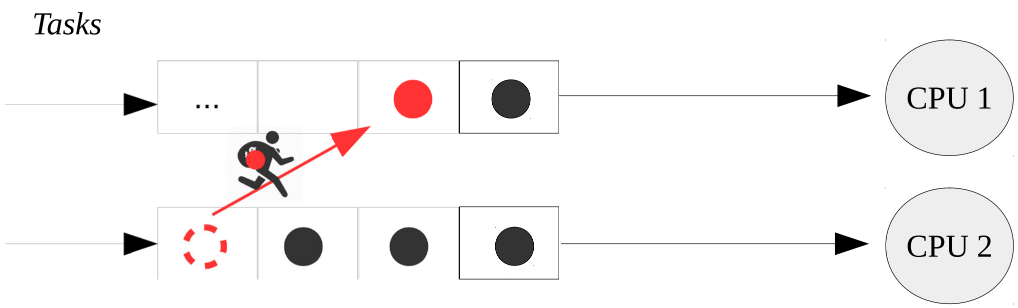


- *Push model* – the OS periodically checks queues length periodically moves tasks if balancing is required
- *Pull model* – each processor notifies an empty queue condition to the OS, or picks tasks from other queues

Multi-processor scheduling

Task migration

- *Work stealing* is an example of pull model based approach
  - Each per-CPU scheduler can “steal” a task from the other in case of empty queue

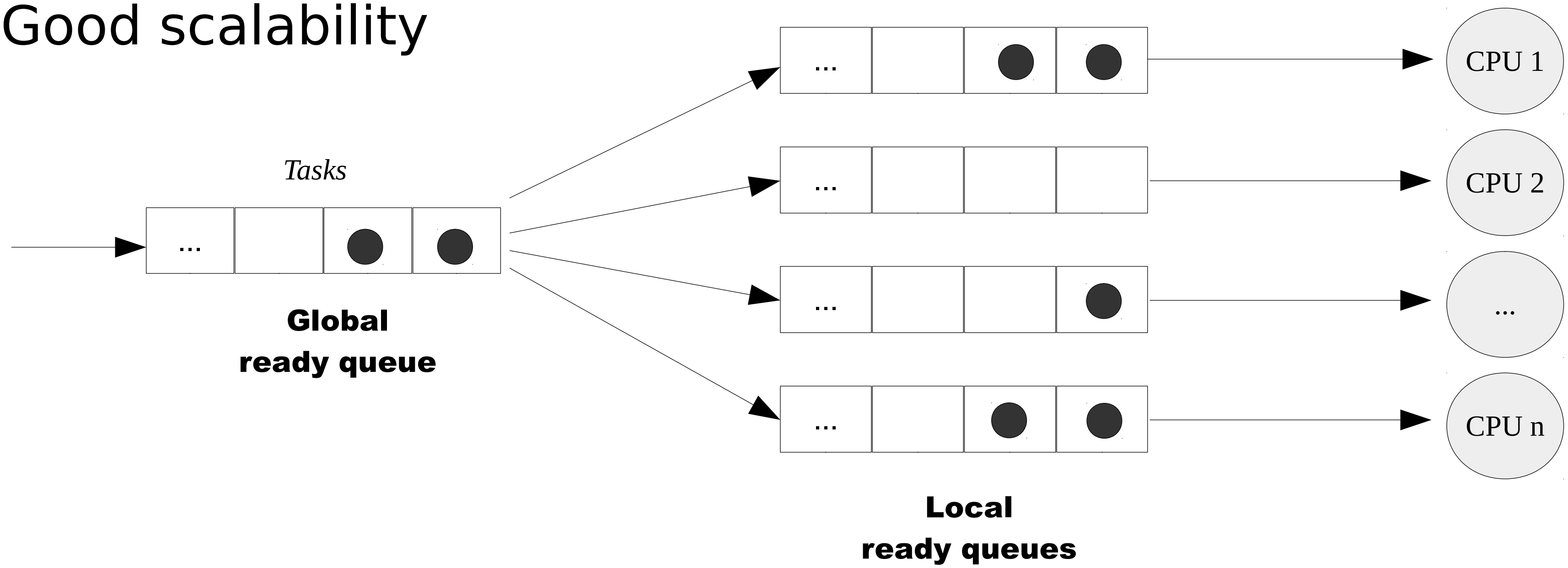


  - Scalable in theory but in practice we must consider the overhead...
    - We need to protect the access to ready queues (global) from concurrent accesses
    - How to identify the shortest queue? Do we need to keep a per-length order?

Multi-processor scheduling

Hierarchical Queues

- A global queue dispatching the tasks in the local ready queues
  - ➔ Hierarchy of schedulers
- Better control over CPU utilization / load balancing
- Good for managing processor affinity
- Good scalability



Questions?

Contacts

- William Fornaciari
  https://home.deib.polimi.it/fornacia
  william.fornaciari@polimi.it

- **HEAPLab**
  DEIB Politecnico di Milano
  Building 21, Floor 1
  Phone: 9613 / 9623

- **Slides credits**:
  - Giuseppe Massari <giuseppe.massari@polimi.it>