Operating Systems / Sistema de Operação

Processor scheduling

Artur Pereira / António Rui Borges

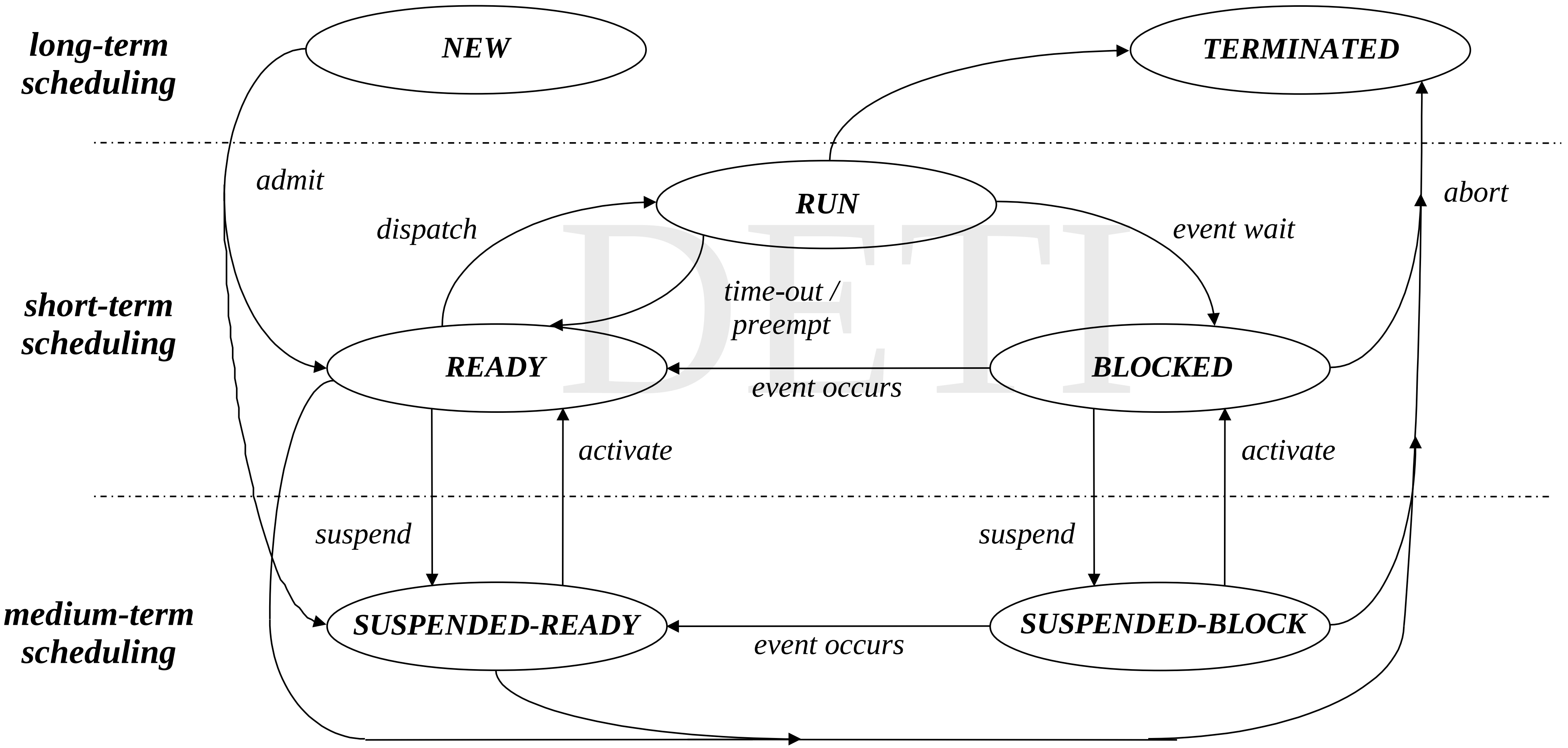
Contents

- *Basic concepts*
- *Types of processor scheduling*
- *Scheduling criteria*
- *Scheduling algorithms*
- *Scheduling in Linux*
- *Bibliography*

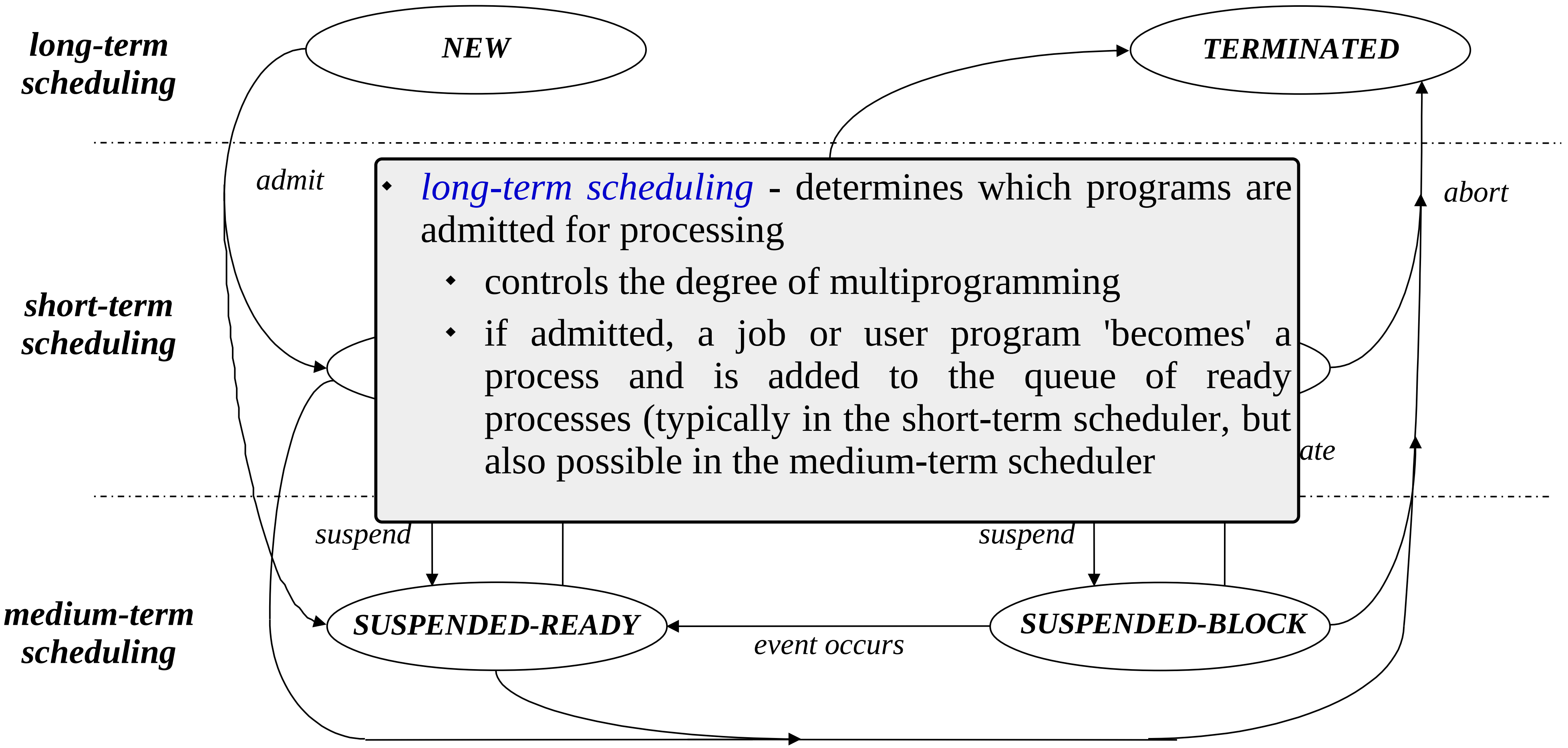
Basic concepts

- When seen by its own, the execution of a process is an alternate sequence of two type of periods:
  - executing CPU instructions (**CPU burst**)
  - waiting for the completion of an I/O request (**I/O burst**)
- A process can be classified as **I/O-bound** or **CPU-bound** (or **processor-bound**)
  - It is I/O-bound if it has many short CPU bursts
  - It is CPU-bound if it has (few) long CPU bursts
- The idea behind multiprogramming is to take advantage of the I/O burst periods to put other processes using the CPU
- The processor scheduler is the component of the operating system responsible for decising how the CPU is assigned for the execution of the CPU bursts of the several processes that coexist in the system

Types of processor scheduling

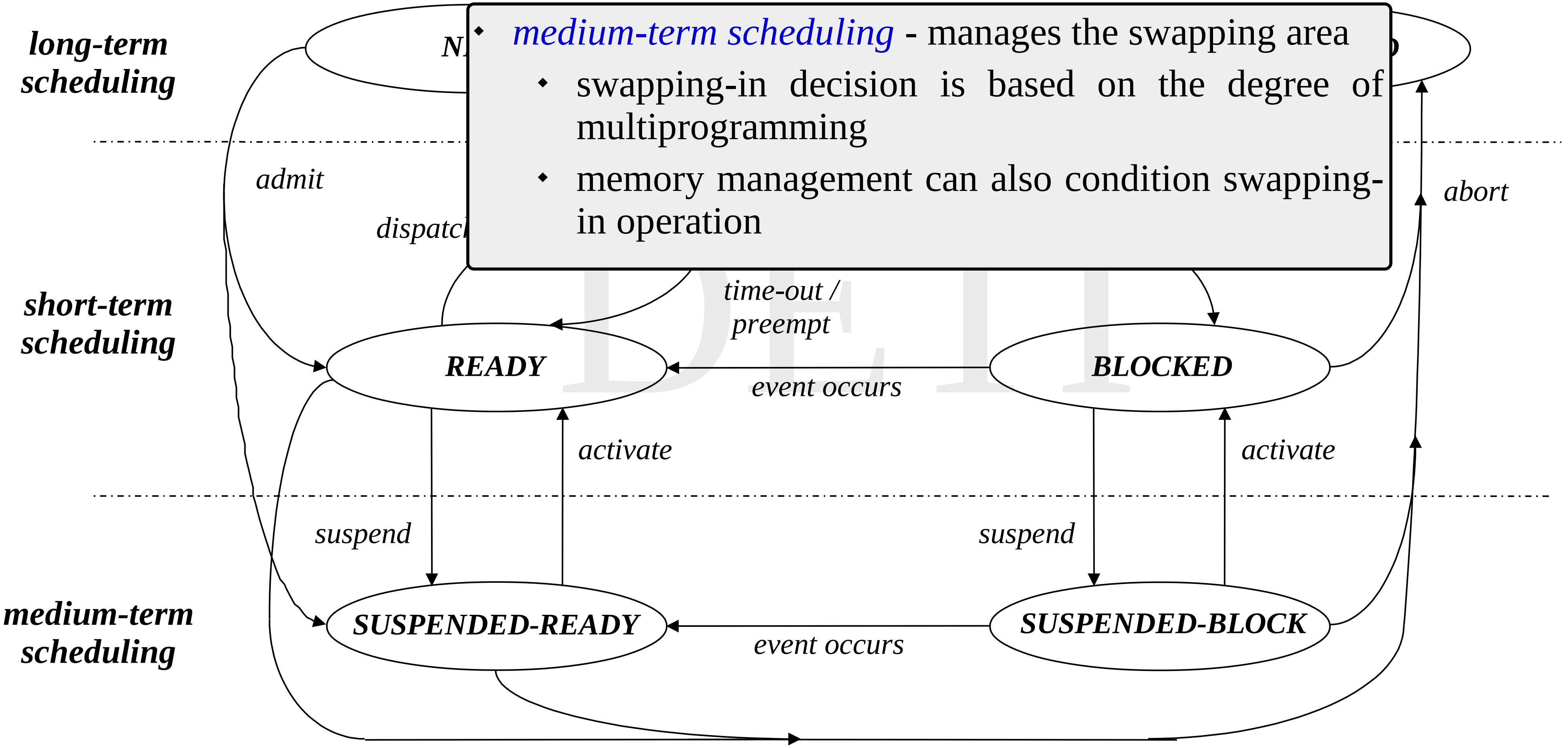


Types of processor scheduling



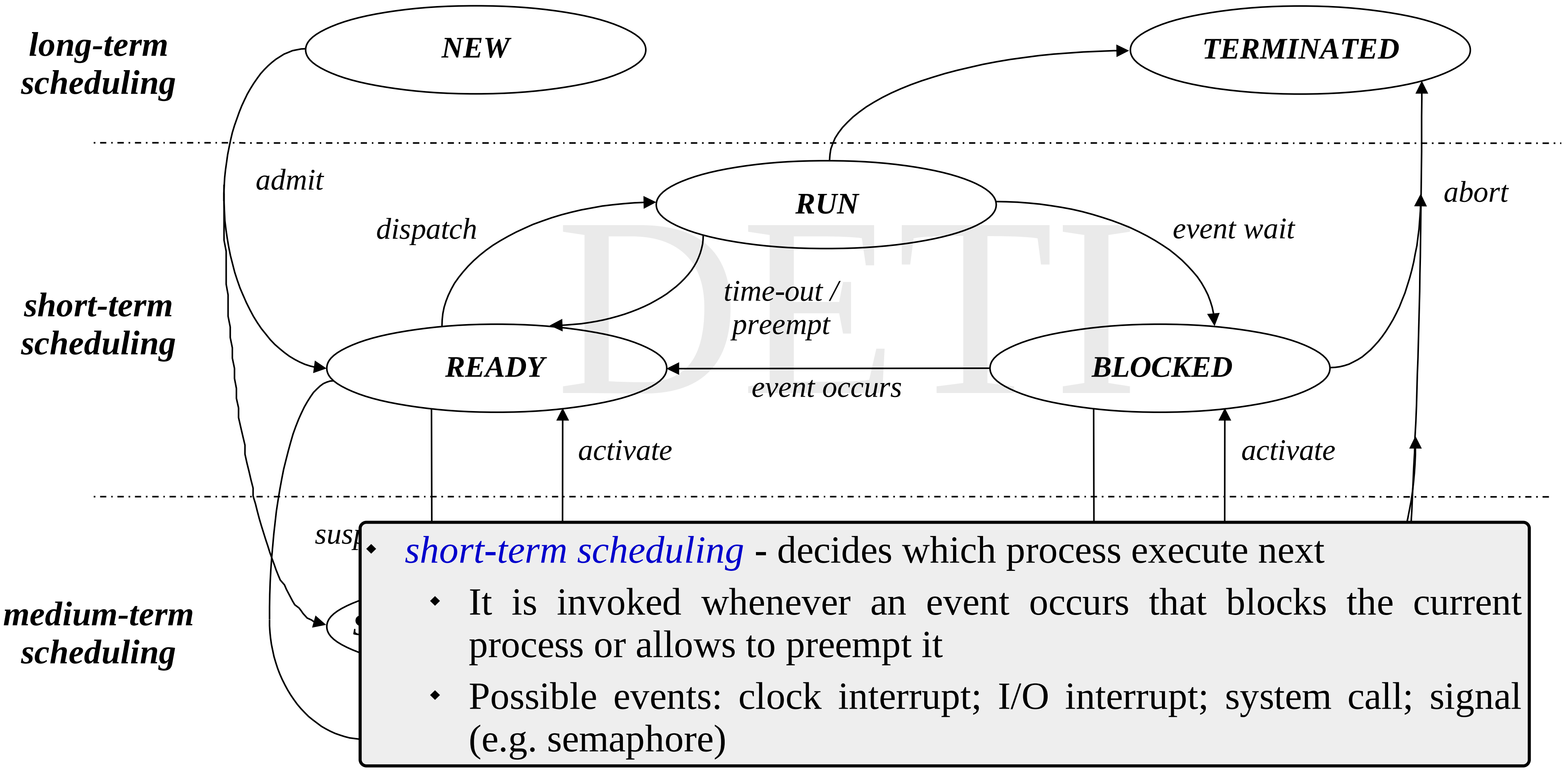
- *long-term scheduling* - determines which programs are admitted for processing
  - controls the degree of multiprogramming
  - if admitted, a job or user program 'becomes' a process and is added to the queue of ready processes (typically in the short-term scheduler, but also possible in the medium-term scheduler

Types of processor scheduling



- *medium-term scheduling* - manages the swapping area
  - swapping-in decision is based on the degree of multiprogramming
  - memory management can also condition swapping-in operation

Types of processor scheduling



- *short-term scheduling* - decides which process execute next
  - It is invoked whenever an event occurs that blocks the current process or allows to preempt it
  - Possible events: clock interrupt; I/O interrupt; system call; signal (e.g. semaphore)

Scheduling criteria (user oriented)

- *Turnaround time* – interval of time between the submission of a process and its completion (includes actual execution time plus time spent waiting for resources, including the processor)
  - appropriate measure for a batch job
  - should be minimized
- *Waiting time* – sum of periods spent by a process waiting in the ready state
  - should be minimized
- *Response time* – time from the submission of a request until the response begins to be produced
  - appropriate measure for an interactive process
  - should be minimized
  - but also the number of interactive processes with acceptable response time should be maximized
- *Deadlines* – time of completion of a process
  - percentage of deadlines met should be maximized, even subordinating other goals
- *Predictability* – how response is affected by the load on the system
  - A given job should run in about the same amount of time and at about the same cost regardless of the load on the system

Scheduling criteria (system oriented)

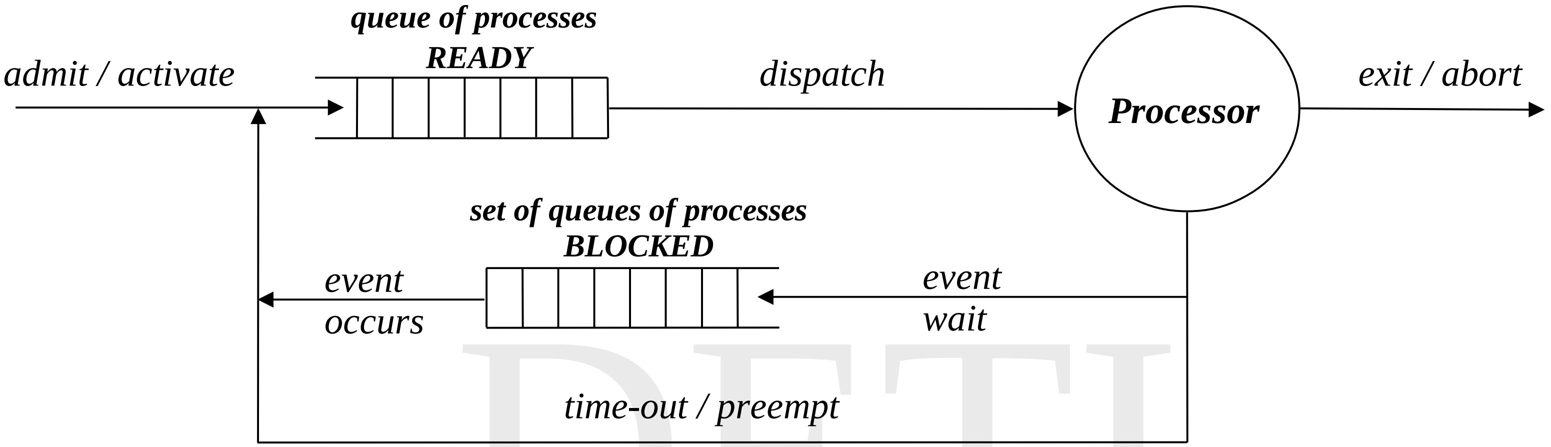
- *Fairness* – equality of treatment
  - In the absence of guidance, processes should be treated the same, and no process should suffer starvation
- *Throughput* – number of processes completed per unit of time
  - measures the amount of work being performed
  - should be maximized
  - depends on lengths of processes and scheduling policy
- *Processor utilization* – percentage of time that the processor is busy
  - should be maximized (specially in expensive shared systems)
- *Enforcing priorities*
  - higher-priority processes should be favoured

- Impossible to satisfy all criteria simultaneously
- Those to favour depends on application

Preemption & non-preemption

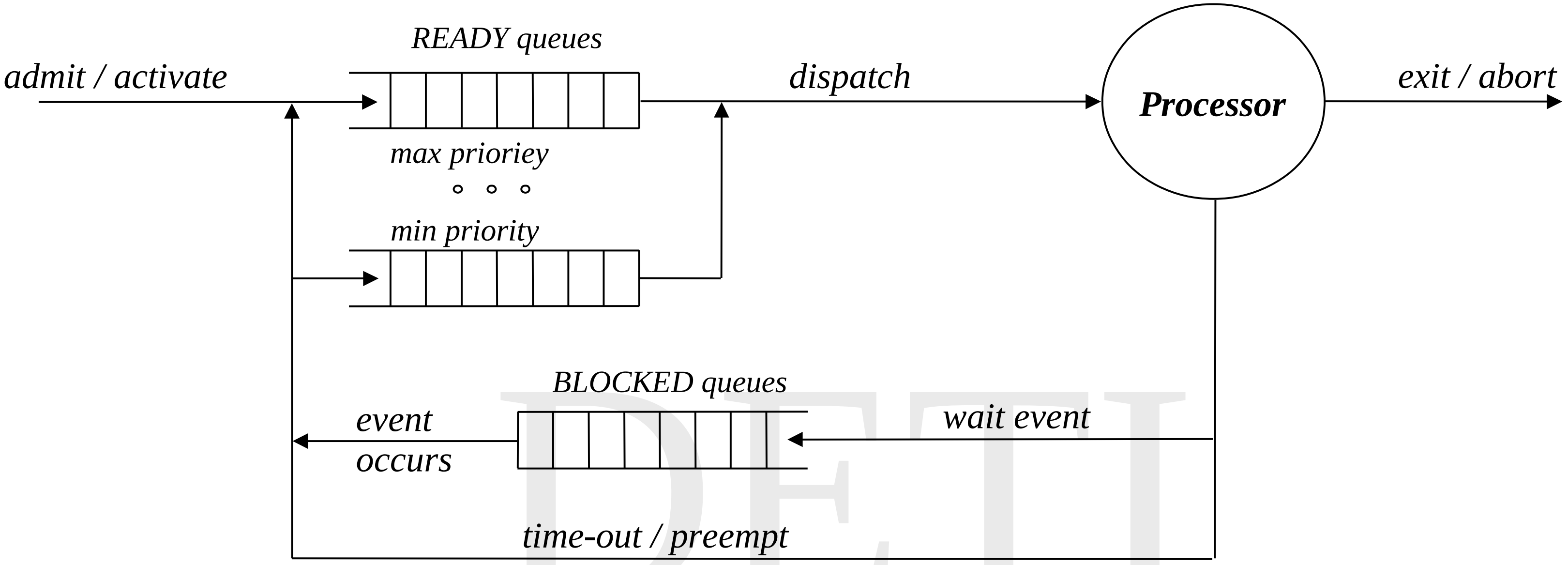
- *Non-preemptive scheduling* – a process keeps the processor until it blocks or ends
  - Transitions *time-out* and *preempt* do not exist
  - Typical in batch systems. *Why?*
- *Preemptive scheduling* – a process can lose the processor due to external reasons
  - by exhaustion of the assigned time quantum
  - because a more priority process is ready to run
  - Typical of interactive systems. *Why?*
- What type to use in a real-time system? *Why*?

Scheduling - favouring fearness



- All processes are equal, being served in order of arraival
  - In *non-preemptive scheduling* is normally referred to as *first-come, first served* (*FCFS*);
  - In *preemptive scheduling* is normally referred to as *round robin*.
- Easy to implement
- Favours CPU-bound processes over I/O-bound ones
- In interactive systems, the time quantum should be carefully chosen in order to get a good compromise between fairness and response time

Scheduling - priorities



- Often, being all the same is not the most appropriate:
  - minimizing response time requires I/O-bound processes to be privileged
  - in real-time systems, there are processes (for example, those associated with alarm events or operational actions) that have severe time constraints
- To address this, processes can be grouped in different levels of priority
  - higher-priority processes are selected first for execution
  - lower-priority processes can suffer starvation

Scheduling - static priorities

- Priorities can be:
  - *static* – if they are deterministically defined
  - *dynamic* – if they depend on the past history of execution of the processes
- Approach to static priorities:
  - Processes are grouped into fixed priority classes, according to their relative importance
  - Clear risk of starvation of lower-priority processes
  - The most unfair discipline
  - Typical in real-time systems (*why?*)
- Another approach to static priorities:
  - When a process is created, a given priority level is assigned to it
  - on *time-out* the priority is decremented
  - on *wait event* the priority is incremented
  - when a minimum value is reached, the priority is set to the initial value

Scheduling - dynamic priorities

- Approach to dynamic priorities:
  - priority classes are functionally defined
  - change of class is based on how the last execution window was used
  - For example:
    - *level 1 (highest priority):* ***terminals*** – a process enters this class on *event occurs* if it was waiting for data from the standard input device
    - *level 2:* ***generic I/O*** – a process enters this class on *event occurs* if it was waiting for data from another type of input device
    - *level 3:* ***small time quantum*** – a process enters this class on *time-out*
    - *level 4: (lowest priority):* ***large time quantum*** – a process enters this class after a successive number of *time-outs*
      - they are clearly CPU-bound processes and the ideia is give them large execution windows, less times

Scheduling - dynamic priorities

- Another approach to dynamic priorities:
  - In batch systems, the turnaround time should be minimized
  - if estimates of the execution times of a set of processes are known in advance, it is possible to establish an order for the execution of the processes that minimizes the average turnaround time of the group
  - Assume we have N jobs, whose estimates of the execution times are te_n, with $n = 1, 2, \ldots , N$. Then the *average turnaround time* is

    $$tm = te_1 + (N-1)/N \cdot te_2 + \ldots + 1/N \cdot te_N \;,$$

  - *tm* is minimum if jobs are sorted in ascending order of the estimated execution times
- This selection method is called *shortest job first* (*SJF*) or *shortest process next* (*SPN*)

Scheduling - dynamic priorities

- Yet another approach to dynamic priorities:
  - An approach similar to the previous one can be used in interactive systems
  - The idea is to estimate the occupancy fraction of the next execution window, based on the occupation of the past windows, and assign the processor to the process for which this estimate is the lowest
  - Let fe_1 be the estimate of the occupancy fraction of the first execution window assigned to a process and let f_1 be the fraction effectively verified. Then:
    - estimate 2 is given by

$$fe_2 = a \cdot fe_1 + (1-a) \cdot f_1 \;, \qquad \text{with } a \in [0, 1]$$

    - estimate N is given by

$$\begin{aligned} fe_N &= a \cdot fe_{N-1} + (1-a) \cdot f_{N-1} \;, \qquad \text{with } a \in [0, 1] \\ &= a^{N-1} \cdot fe_1 + a^{N-2} \cdot (1-a) \cdot f_1 + \ldots\, a \cdot (1-a) \cdot f_{N-2} + (1-a) \cdot f_{N-1} \;. \end{aligned}$$

    - coefficient a is used to control how much the past history of execution influences the present estimate

Scheduling - dynamic priorities

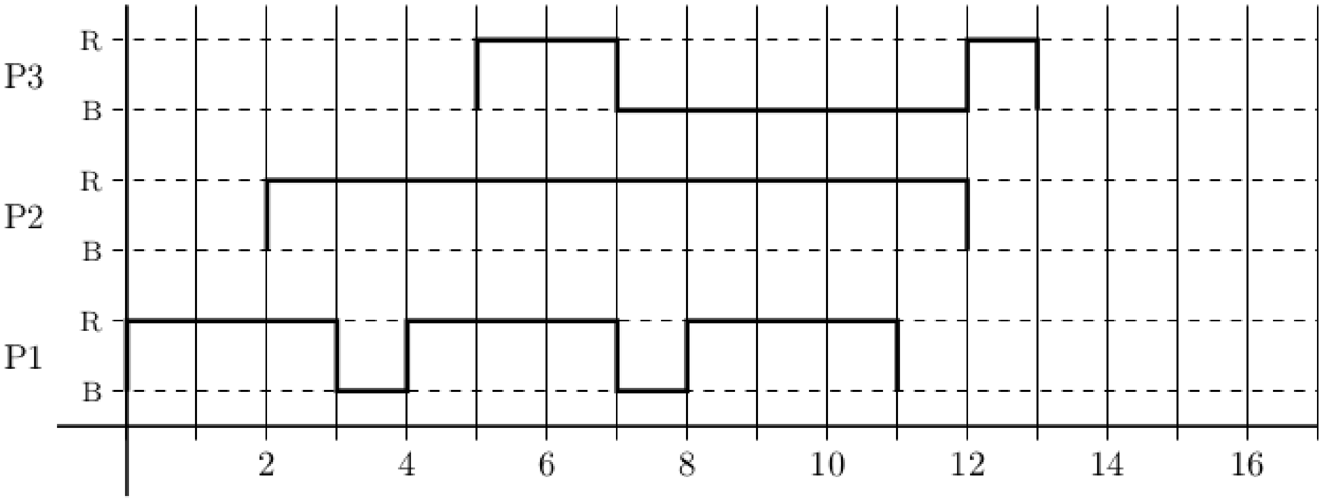
- In the previous approach, CPU-bound processes can suffer starvation
- To overcome that problem, the *aging* of a process in the READY queue can be part of the equation
- Let R be such time, typically normalized in terms of the duration of the execution interval. Then
  - priority p of a process can be given by

$$p = \frac{1 + b \cdot R}{fe_N}$$

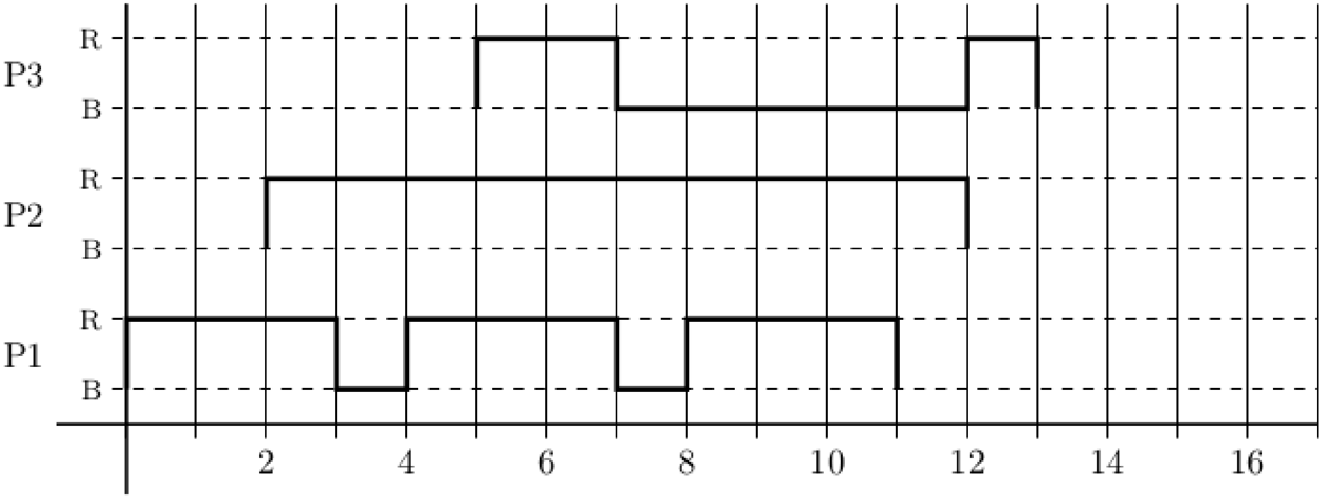
  where b is a coefficient that controls how much the aging weights the in priority

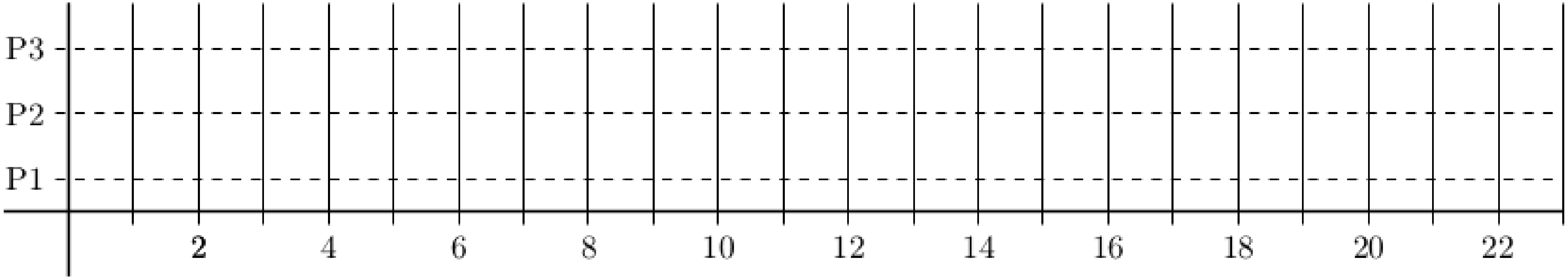
Scheduling policies - FCFS

- First-Come-First-Server (FCFS) scheduler
  - Also known as First-In-First-Out (FIFO)
  - The oldest process in the READY queue is the first to be selected
  - Non-preemptive (in strict sense)
    - Can be combined with a priority schema
  - Favours processor-bound processes over I/O-bound
  - Can result in bad use of both processor and I/O devices



Scheduling policies - FCFS



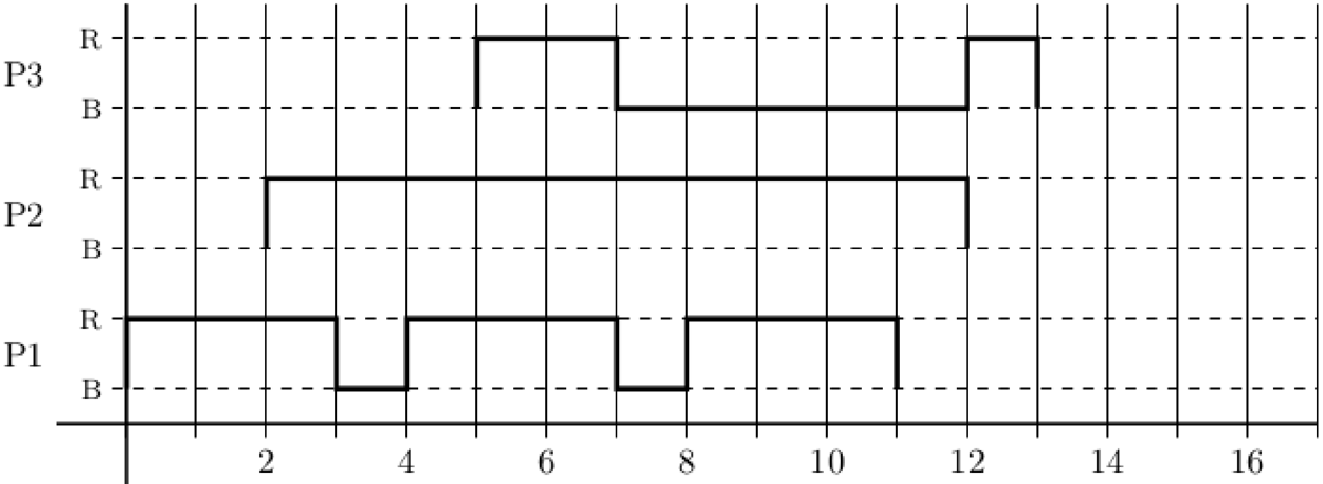


Scheduling policies - RR

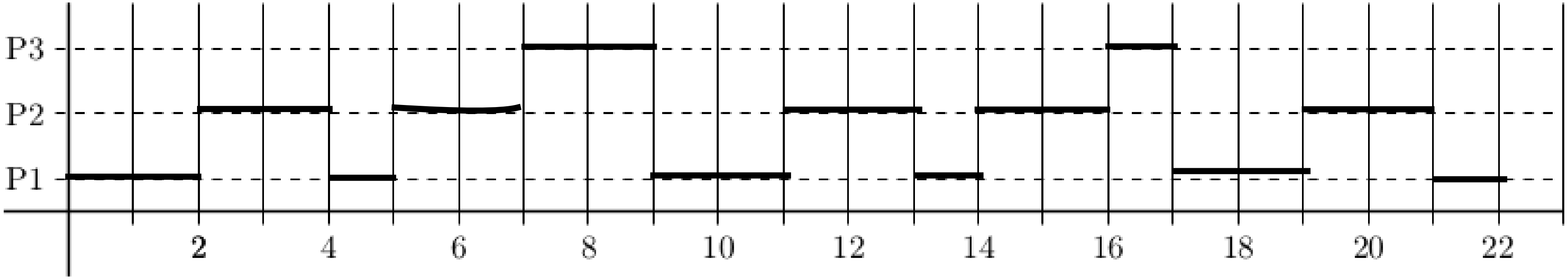
- Round robin (RR) scheduler
  - Preemptive (base on a clock) E ATRIBUIDO UM TIME QUANTUM MAXIMO PARA UTILIZAR O CPU.
    - Each process is given a maximum slice of time before being preempted (time quantum)
    - Also known as time slicing
  - The oldest process in the READY queue is the first one to be selected (no priorities)
  - The principal design issue is the time quantum
    - very short is good, because short processes will move through the system quickly
    - very short is bad, because every process switching involves a processing overhead
  - Effective in general purpose time-sharing systems and in transaction processing systems
  - Favours processor-bound processes over I/O-bound
  - Can result in bad use of I/O devices

TIME QUANTUM DE 2

Scheduling policies - RR



AO FIM DE DUAS UNIDADES DE TEMPO PASSA O SEU TIME QUANTUM ELE SAI E ENTRA AGAIN, O PROCESSO 2 TAMBEM ESTA PREPARADO PARA ENTRAR, ENTAO VAMOS VER QUEM É QUE PRIMEIRO ENTROU NA FILA. O P1 SAI DA FILA DE READY E VAI PARA TRAS DO P2



EM T5 O P1 BLOQUEOU E DEPOIS NO T6 ELE ENTRA NO ESTADO READY, EM T7 O P2 VAI PARA O FIM DA FILA

P3 BLOQUEOU EM T9 E O EVENTO QUE ESTA A ESPERA DEMORA 5 UNIDADES DE TEMPO, SO VOLTA A FICAR READY, NO ESTADO 14.

Scheduling policies - SPN

- Shortest Process Next (SPN) scheduler
  - Also known as shortest job first (SJF)
  - Non-preemptive

DEDICADO, A SISTEMAS, BASCH

  - The process with the shortest expected next CPU burst time is selected next
    - FCFS is used to tie up (in case several processes have the same burst time)
  - Risk of starvation for longer processes
  - Requires the knowledge in advance of the (expected) processing time
    - This value can be predicted, using the previous values
  - Used in long-term scheduling in batch systems
    - where users are motivated to estimate the process time limit accurately

Scheduling policies - Linux

- Linux considers 3 scheduling classes, each with multiple priority levels:
  - *SCHED_FIFO* – FIFO real-time threads, with priorities
    - a running thread in this class is preempted only if a higher priority process (of the same class) becomes ready
    - a running thread can voluntarily give up the processor, executing primitive `sched_yield`
    - within the same priority an FCFS discipline is used
  - *SCHED_RR* – Round-robin real-time threads, with priorities
    - additionally, a running process in this class is preempted if its time quantum ends
  - *SCHED_OTHER* – non-real-time threads
    - can only execute if no real-time thread is ready to execute
    - associated to user processes
    - the scheduling police changes along kernel versions
  - priorities range from 0 to 99 for real-time threads and 100 to 139 for the others
  - *nice* system call allows to change the priority of non-real time threads
    - which is 120 by default, with interval [-20,+19] managed by *nice*

Scheduling policies - Linux

Traditional algorithm

- In class *SCHED_OTHER*, priorities are based on credits
- Credits of the running process are decremented at every RTC interrupt
  - the process is preempted when zero credits are reached
- When all ready processes have zero credits, credits for all processes, including those that are blocked, are recalculated
  - recalculation is done based on formula

$$CPU_j(i) = \frac{CPU_j(i-1)}{2} + PBase_j + nice_j$$

  - Past history of execution and priorities are combined in the algorithm
- Response time of I/O-bound processes is minimized
- Starvation of processor-bound processes is avoided
- Not adequate for multiple processors and bad if the number of processes is high

Scheduling policies - Linux

New algorithm

- From version 2.6.23, Linux started using a scheduling algorithm for the SCHED_OTHER class known as completely fair scheduler (CFS)
- Schedule is based on a virtual rime time value (`vruntime`), which records how long a thread has run so far
    - the virtual run time value is related to the physical run time and the priority of the thread
    - higher priorities shrinks the physical run time
- The scheduler selects for execution the thread with the smallest `vruntime`
    - a higher-priority thread that becomes ready to run can preempt a lower-priority thread
    - thus, I/O-bound threads eventually can preempt CPU-bound ones
- The Linux CFS scheduler provides an efficient algorithm for selecting which thread to run next, based on a red-black tree;

Bibliography

Operating Systems Concepts; Silberschatz, Galvin & Gagne; John Wiley & Sons, 9th Ed

- Chapter 6: *Process scheduling* (sections 6.1 to 6.3 and 6.7.1)

Modern Operating Systems; Tanenbaum & Bos; Prentice-Hall International Editions, 4rd Ed

- Chapter 2: *Processes and Threads* (section 2.4)

Operating Systems; Stallings; Prentice-Hall International Editions, 7th Ed

- Chapter 9: *Uniprocessor Scheduling* (sections 9.1 to 9.3)
- Chapter 10: *Multiprocessor and Real-Time Scheduling* (section 10.3)