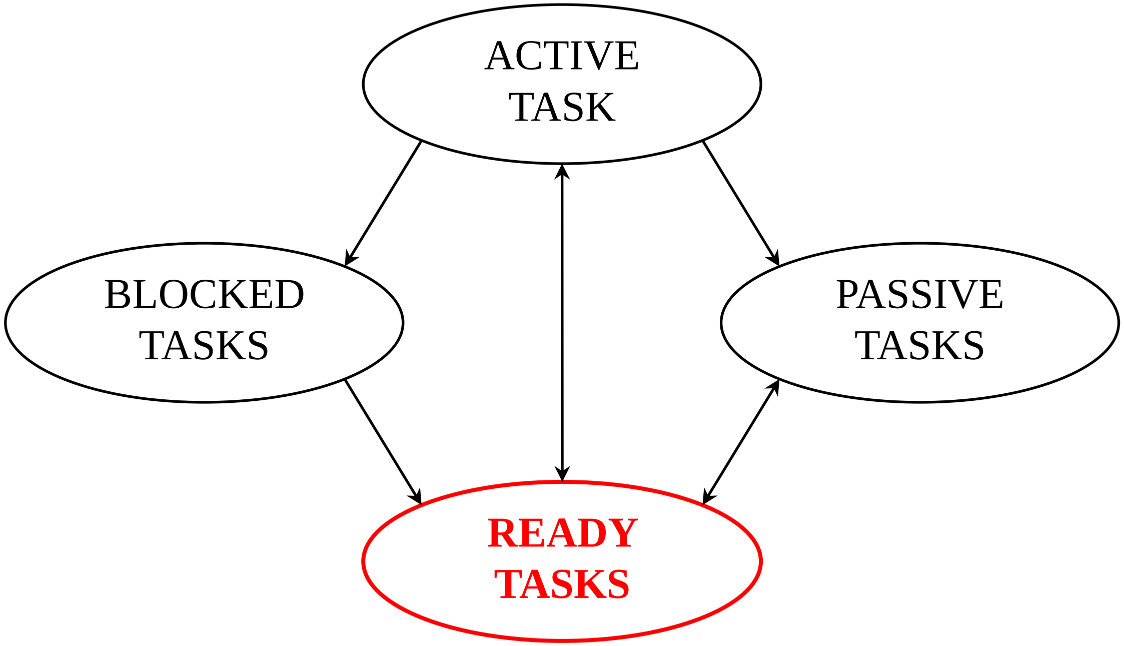
Operating system concepts

Task scheduling

Task scheduling (thread scheduling)

- Target of scheduling are ready tasks



- Active task – currently running on processor
- Ready tasks – ready to be put on processor
- Blocked tasks – wait for some resource (IO, semaphore)
- Passive tasks – finished or not started tasks

Task scheduling problem

- System has more ready tasks than processors
- Main problem:
  - how to divide processor(s) time to available ready tasks?
- How to schedule different tasks?
  - tasks have different properties and *expectations* from the scheduler
  - use per task "type" handling, or use same principles for all (general principles)?
  - how to measure scheduling quality? scheduling metric?

Scheduling environments

- Different environments have different tasks, different requirements, different primary objectives
  - Real-Time Systems (RTS), Embedded Systems
    - monitor, control “real process” – time management is crucial
  - Personal Computers, Workstations
    - user oriented (interactivity)
  - Servers
    - service oriented, process requests from different clients
  - Mobile devices (handhelds, phones, multimedia players)
    - single user and mostly only single task at a time

Scheduling quality

- In RTS, failure in scheduling can have **serious consequences**!

- In "normal" (other) systems (example quality measures)
  - multimedia player must timely provide audio/video subsystems with data or perception of quality may not be satisfactory

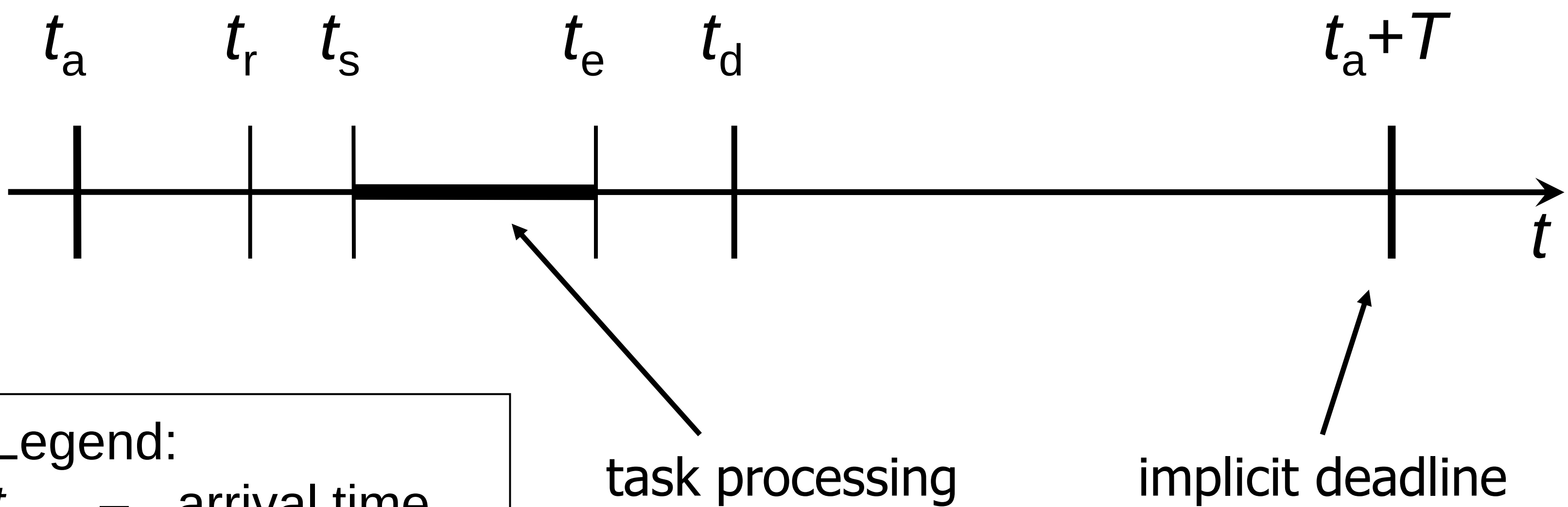
  - user interface should respond timely to user commands
    - e.g. mouse movements, text/commands entry, …

  - mathematic calculations, data compression, file transfers and similar activities that take time to complete ***could be delayed if required*** by other task types, with very small or no observable penalty for user/system

Task types

- Tasks may be divided into categories by many criteria's
- “How a task should be scheduled” categories:
  - “normal” tasks – the user programs
    - perform standard operations
    - don’t require special privileges
    - don’t have strict time constrains
    - may use system resources – through operating system
  - “system” tasks – perform system operations (services)
    - require special privileges
    - might have some (soft) time constrains
  - time constrained tasks (RT tasks, multimedia players)
    - deadline (must) be met – or system failure happens
    - if a task is periodic, it must get enough time to finish periodic processing before the start of next period (“implicit deadline”)

RT periodic task – characteristic times



Legend:

t_a	–	arrival time
t_r	–	ready time
t_s	–	start time
t_e	–	end time
t_d	–	deadline
T	–	period

Task scheduling decisions

- Static scheduling
  - **behavior is defined before system is started**
  - decisions may be “hard coded” in system
  - statically define task execution order and follow it
  - assign priorities to tasks at task creation and schedule tasks by priority (always the highest priority task first)

- Dynamic scheduling
  - **decisions are made at run time**
  - system state is evaluated (including tasks) and next task to be scheduled is chosen
  - examples:
    - consumed CPU time for all tasks is compared and task with lowest consumption is chosen (to obtain *fairness*)
    - deadlines are compared and task with nearest deadline is chosen (to meet all constraints)

Basic scheduling principles

- First Come First Served – FCFS
  - usually known as: First In First Out – FIFO
  - adequate for servers – requests are processed in receiving order

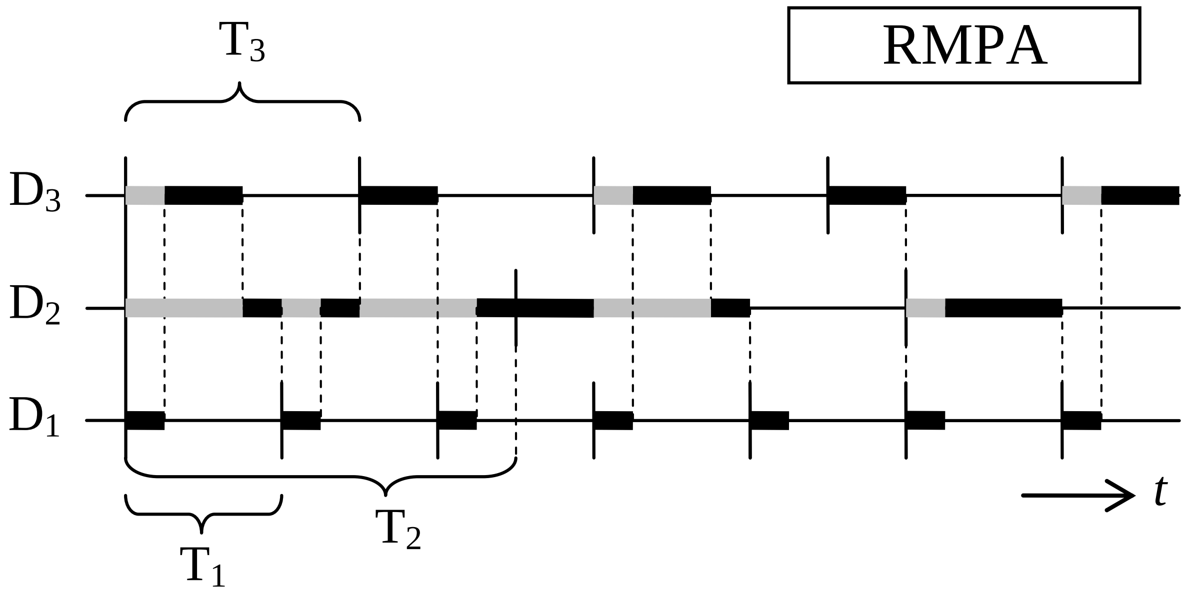
- Priority based scheduling
  - tasks with higher priority have precedence over lower priority tasks
  - a higher priority task will always *preempt* lower prior. task!
  - adequate for RTS, priority reflects task relevance

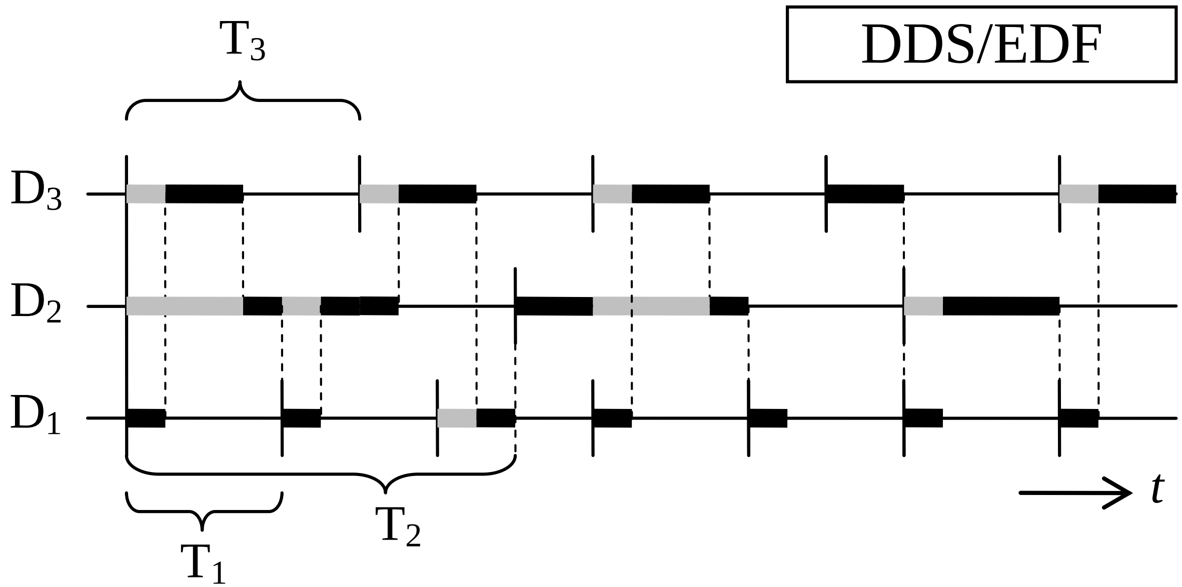
- Time share based scheduling – *Round Robin* and similar
  - tasks share (fairly) processors' time
  - adequate for multi-user workstations and servers
    - e.g. terminal based servers (basic shell or with GUI)

Principal RTS scheduling principles

- Rate Monotonic Scheduling – RMS (mostly used!)
  - also known as Rate Monotonic Priority Assignment – RMPA
  - assigns priorities to tasks according to their period lengths – frequent tasks (with shorter periods) get higher priority
  - RMS only predefines task priorities (base priority) – actual scheduling is performed later using priorities
- Deadline Driven Scheduling – DDS
  - task with nearest deadline is scheduled first (aka EDF, EDD)
- Sporadic scheduling
  - periodic task within single period can run “budget time” with defined *higher* priority, and if that is not enough, the rest of processing in this period is performed with *lower* priority

RMPA and DDS examples





Sporadic scheduling example

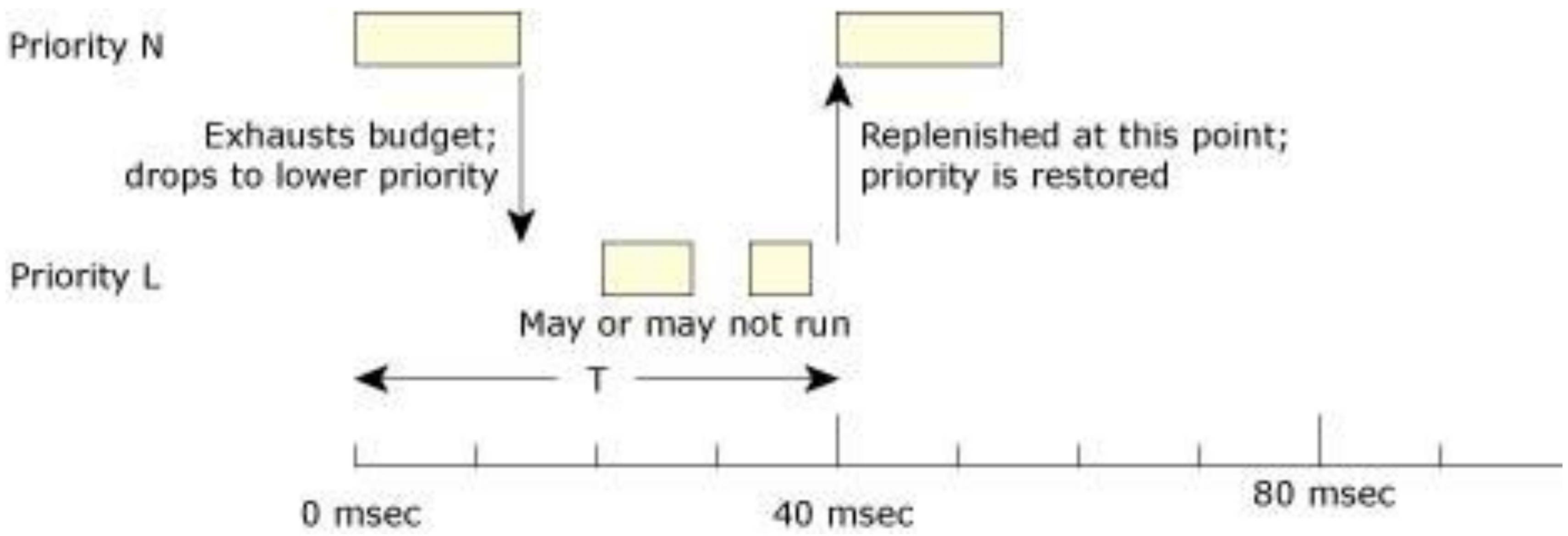


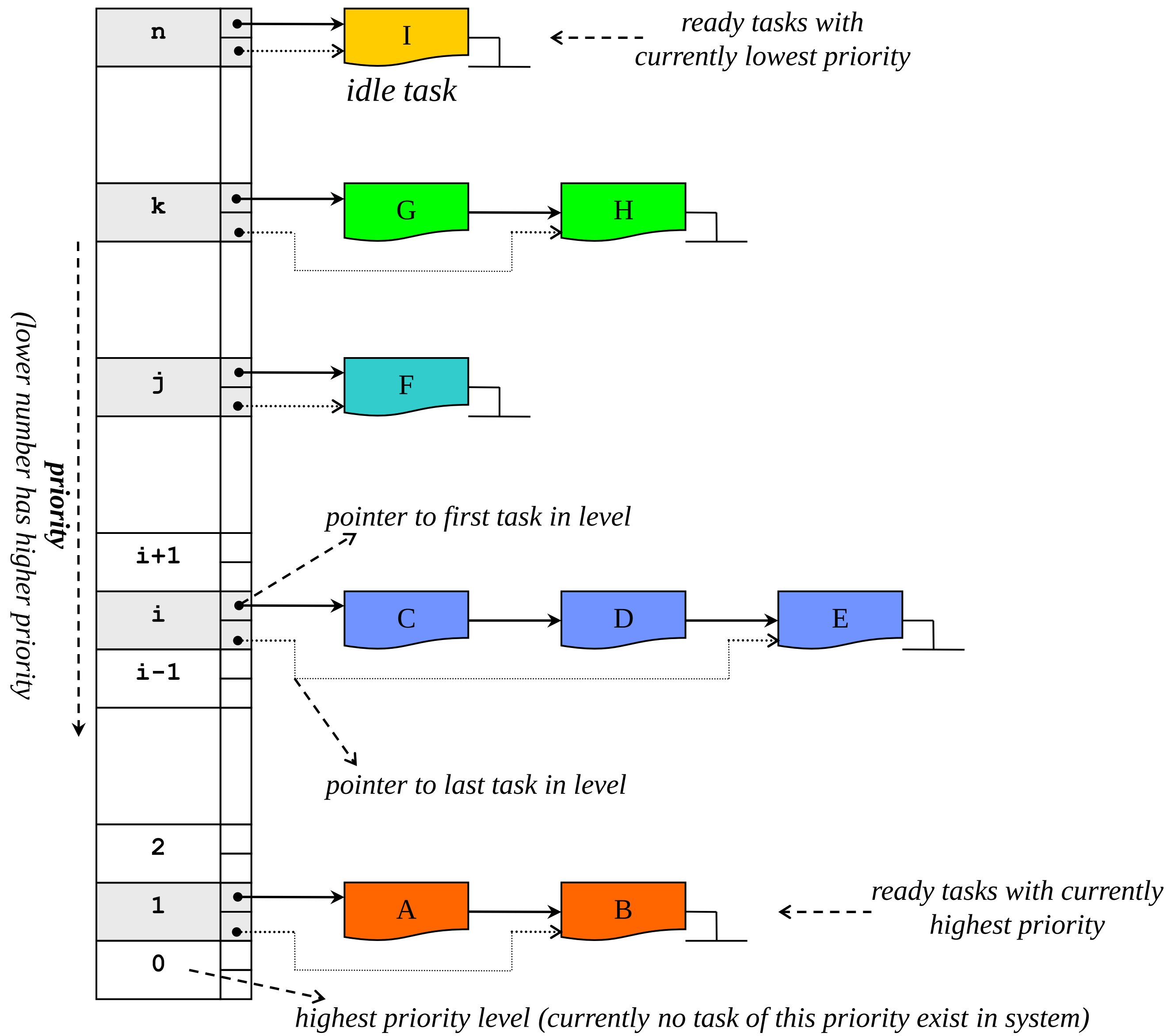
Image from: http://www.qnx.com/developers/docs/6.3.0SP3/neutrino/sys_arch/kernel.html

Mostly implemented scheduling principle

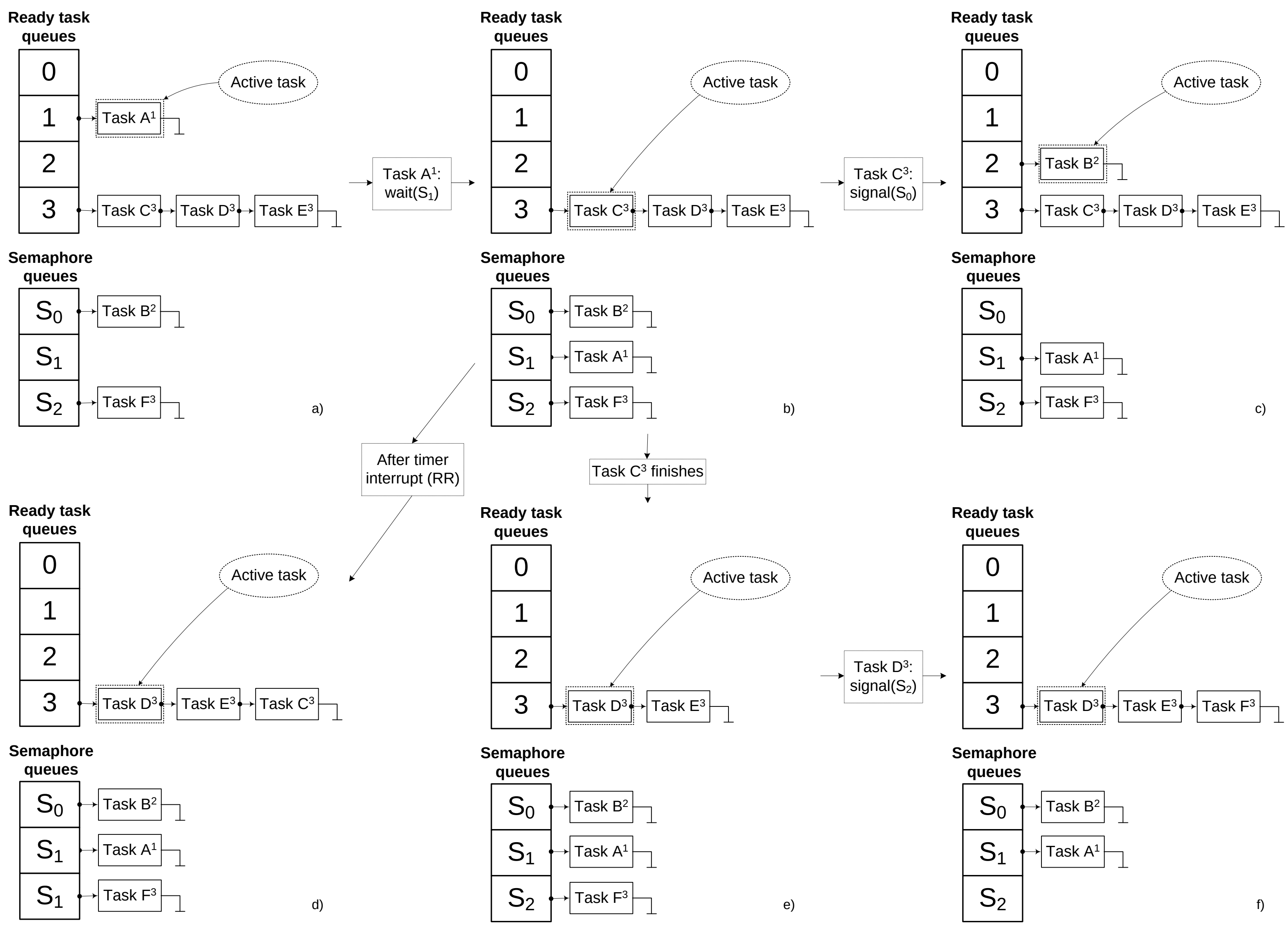
- Non-RTS have distinction between *normal tasks* and *RT tasks* – scheduling is different!

- RT task scheduling (implemented also on non-RTS)
  - **priority** is the main scheduling principle
  - higher priority task always preempts lower ones
  - tasks are organized into priority levels
    - tasks with same priority are in the same level (queue)
  - if more than one thread of the same priority exists then scheduler will schedule them by principle:
    - **FIFO** – first task will execute until its completed or until it becomes blocked (not ready)
    - **Round Robin** – each task will be given only a “time slice” of CPU before it is put at the end of queue, and first one from the queue is chosen next

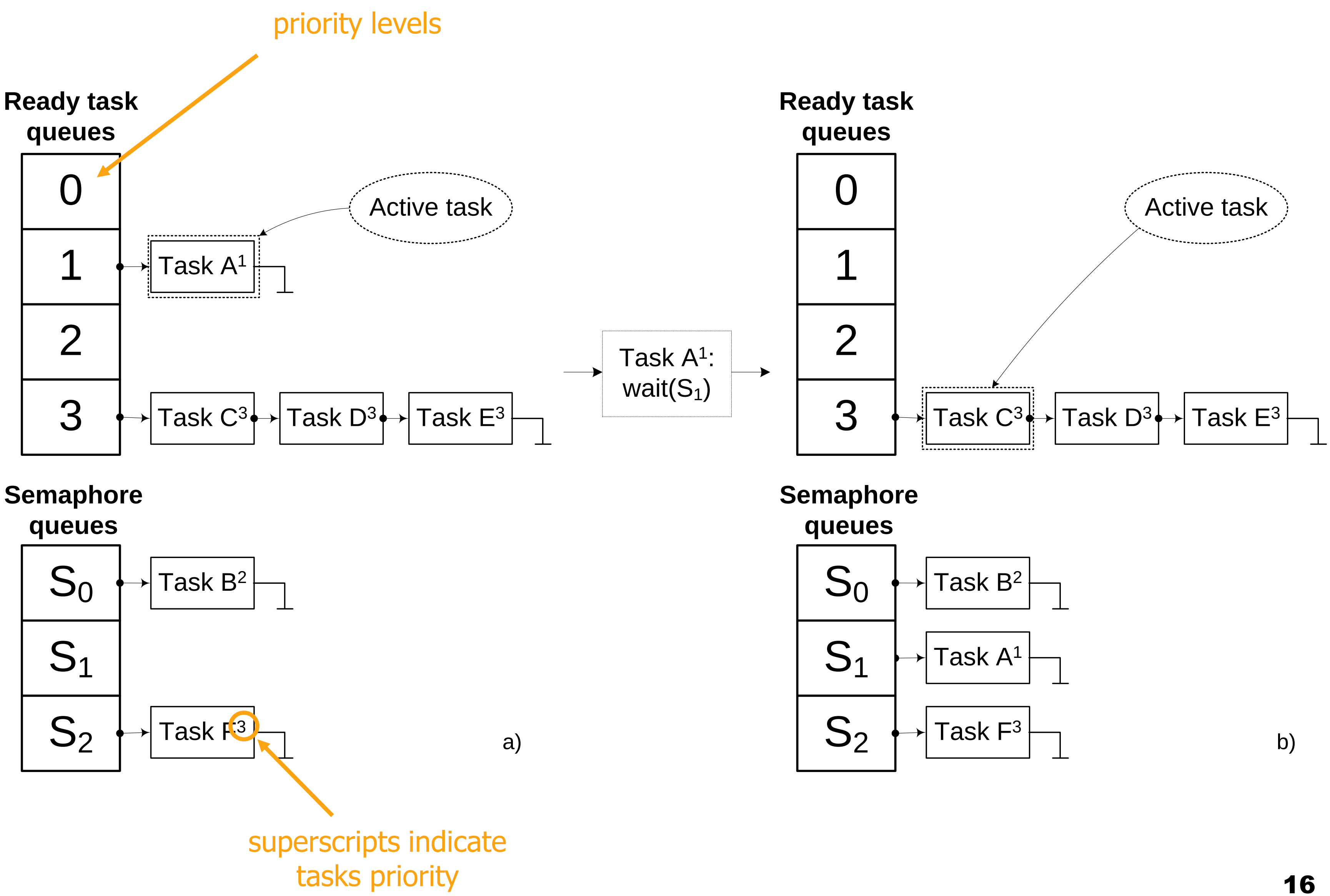
Priority levels – typical ready task organization



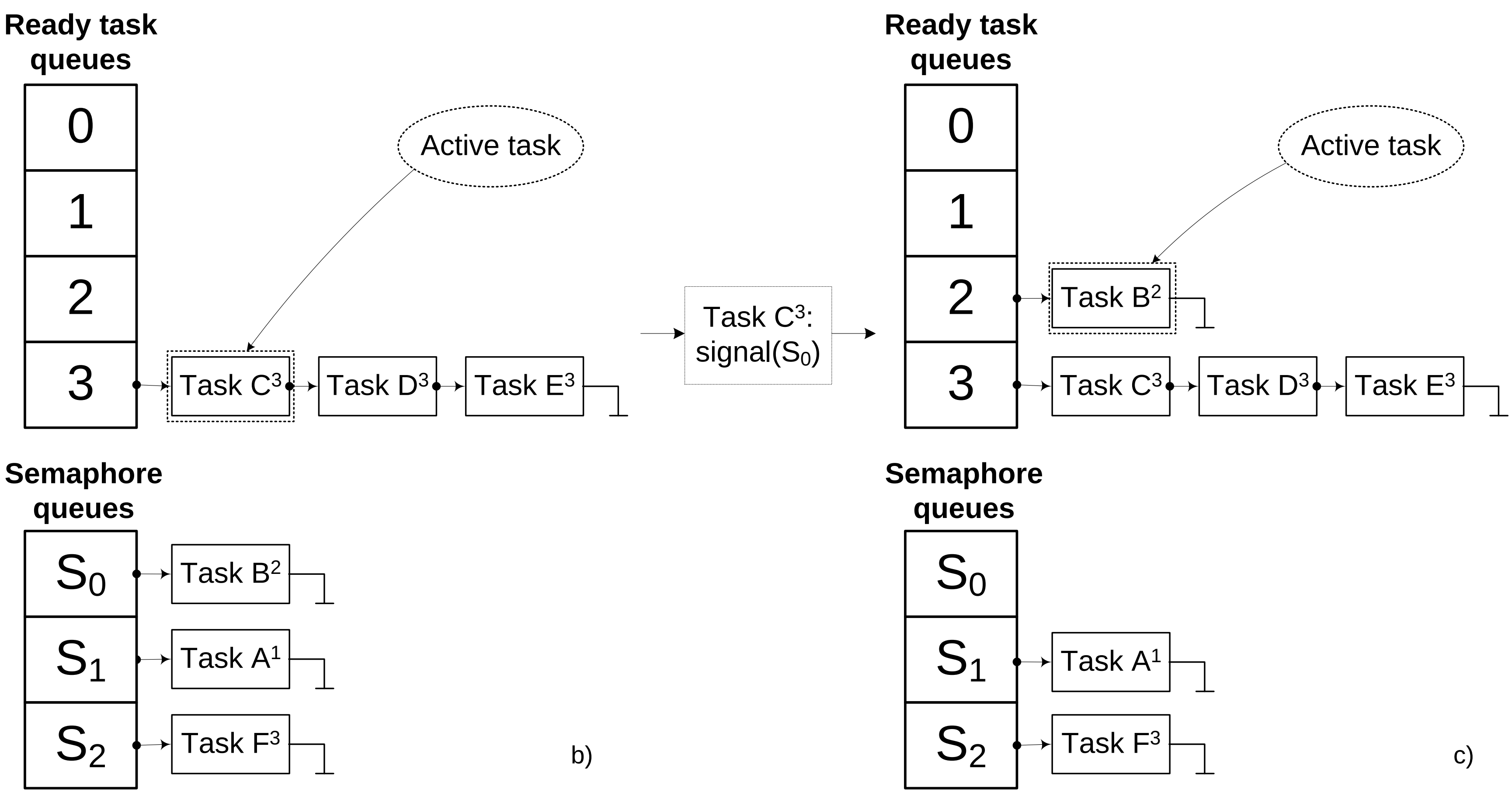
Example system state and scheduling



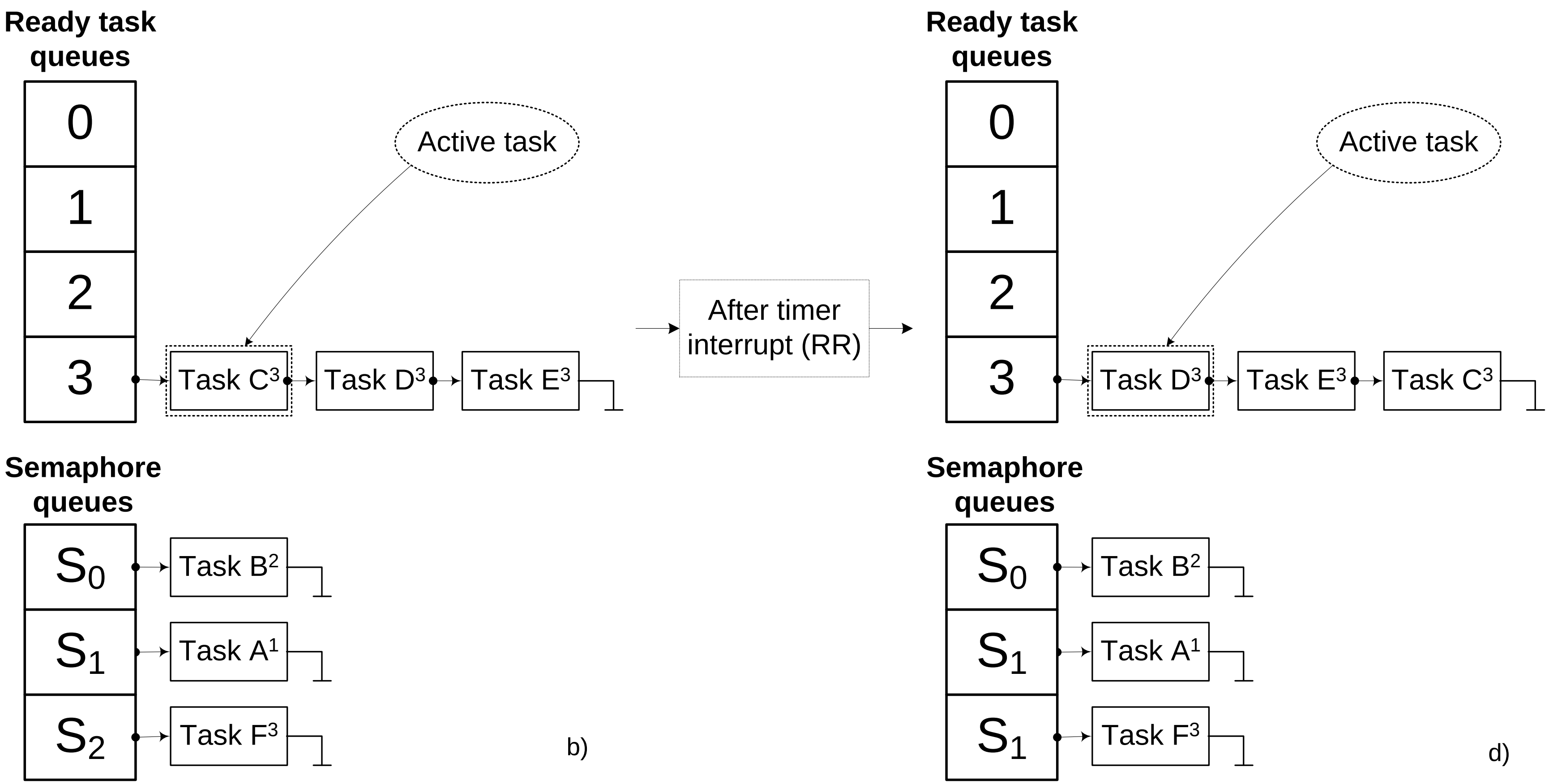
Example scheduling



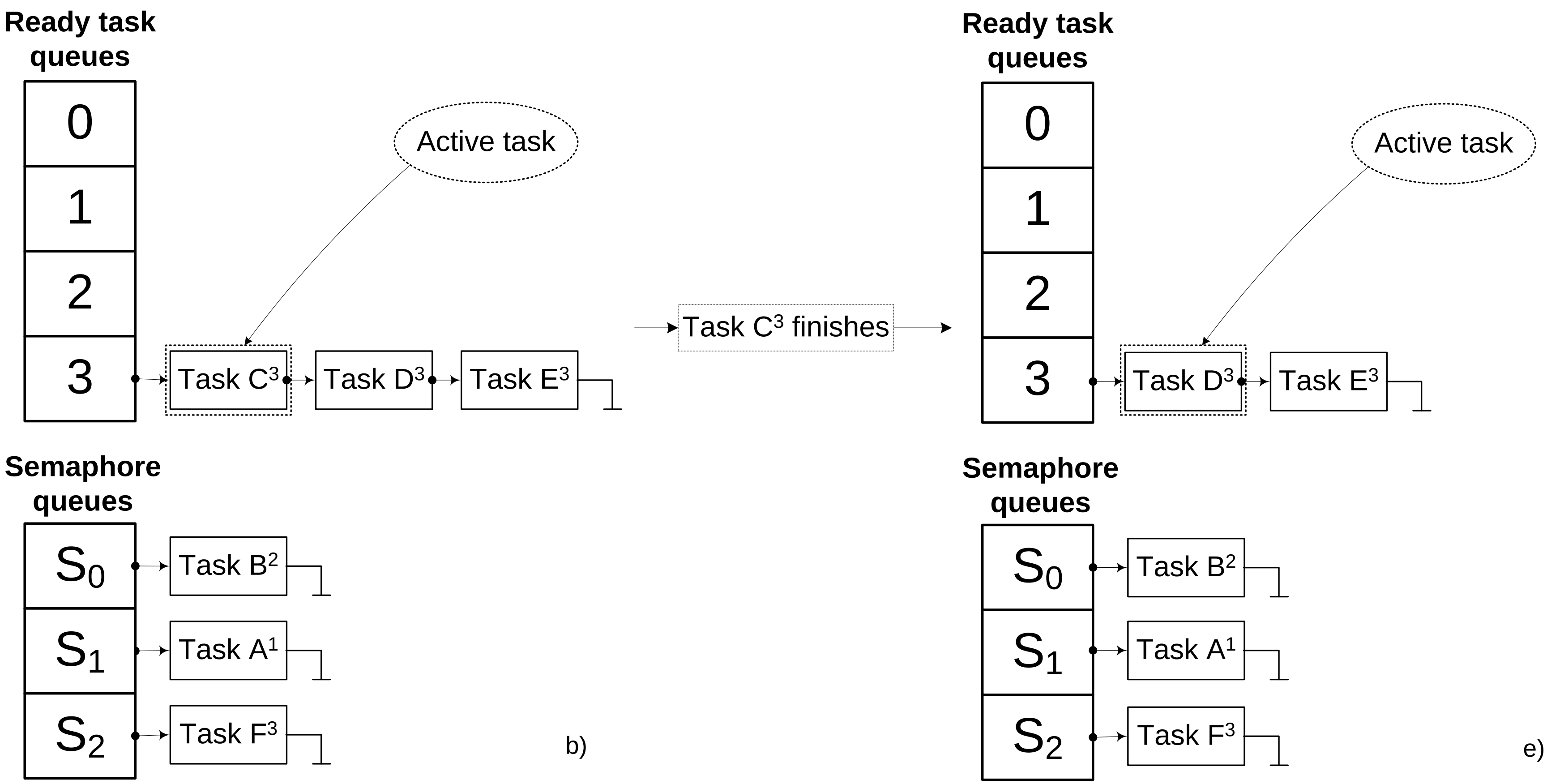
Example scheduling



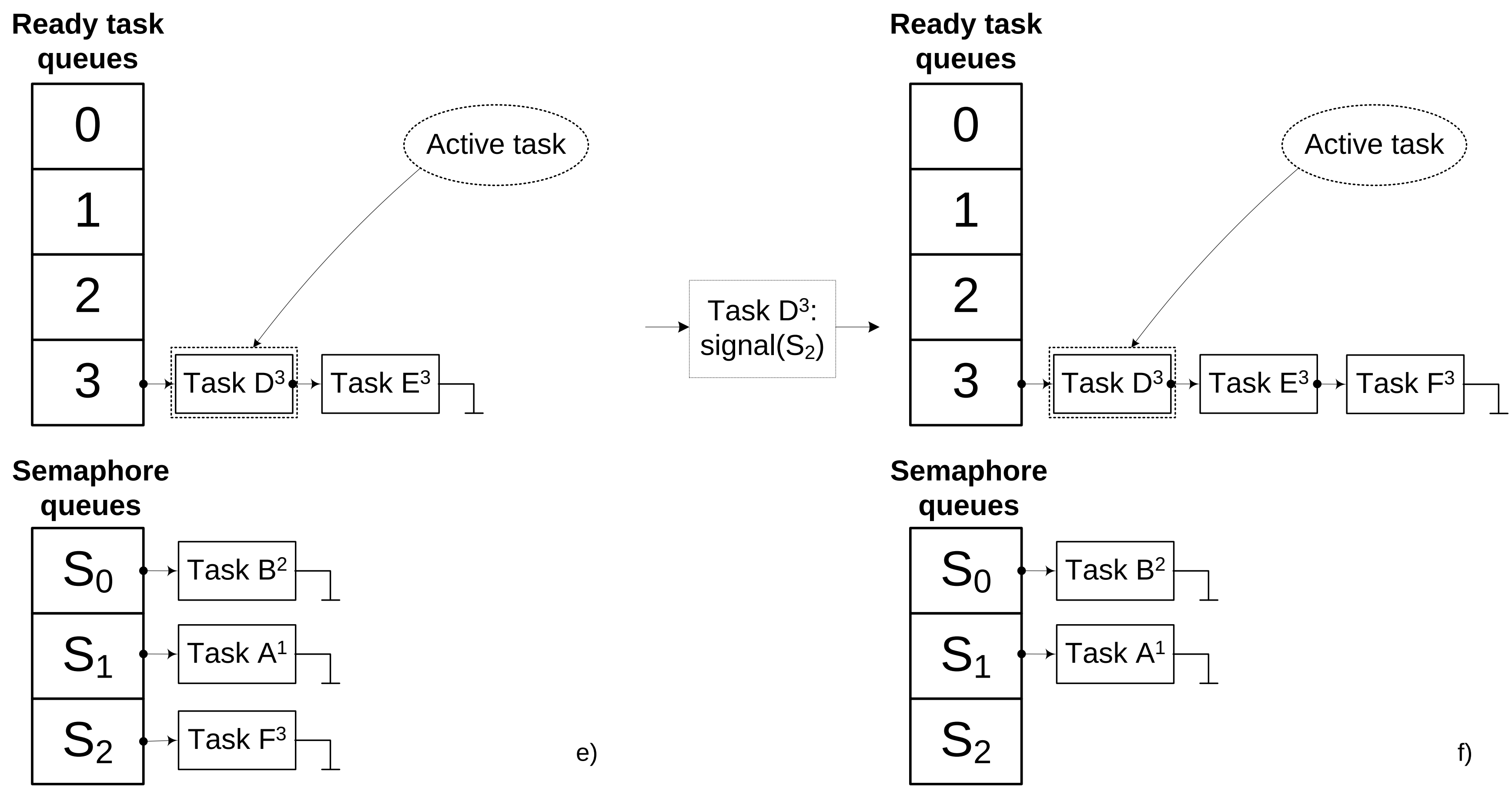
Example scheduling



Example scheduling



Example scheduling



Scheduling in non-real-time environments

- Scheduling "normal" tasks

- What is a "normal" task?
  - task with "soft time constraint", or with no time constrains
    - soft time constraint – allowed delay is e.g. > 100 ms
    - nothing critical will happen if some time constraint occasionally is not met
    - however, significant or frequent delays in responses will degrade user experience (satisfaction with computer system)

  - "worker" tasks – perform CPU intensive operations
    - time to completion is measured in (at least several) seconds
    - don't interact with user (at least not frequently)
    - don't require IO devices, don't control or monitor them

Non-RT scheduling principles examples

- Fairness – share CPU(s) time fair to all tasks in system
  - respect tasks priorities – higher priority = more CPU time
- Processor utilization
  - the more the better – more jobs are performed
- Tasks throughput
  - finish more tasks (“favor short tasks”)
- Minimize queue waiting times
  - shorten length of “time slice”, but …
- Response time
  - favor “interactive tasks” – they mostly don’t use CPU, but when they are activated (e.g. on keystroke) give them CPU as soon as possible
- “Hot cache” optimization on multiprocessors
  - maintain same tasks on same processors – they might find their data still in cache even after context change

Multilevel feedback queue

- Theoretical scheduling that is used in today's operating systems (basic ideas at least) is called *multilevel feedback queue*

Multilevel feedback queue (from Wikipedia)

- In computer science, a multilevel feedback queue is a scheduling algorithm.
- It is intended to meet the following design requirements for multimode systems:
  - Give preference to short jobs.
  - Give preference to I/O bound processes.
  - Quickly establish the nature of a process and schedule the process accordingly.

Multilevel feedback queue (from Wikipedia, cont)

- Multiple FIFO queues are used and the operation is as follows:
  - A **new process** is positioned at the end of the **top-level** FIFO queue.
  - At some stage the process reaches the head of the queue and is assigned the CPU.
  - If the process is completed it leaves the system.
  - If the process **voluntarily relinquishes control** it leaves the queuing network, and when the process becomes ready again it enters the system **on the same queue level**.
  - If the process **uses all the quantum time**, it is pre-empted and positioned at the end of the **next lower level** queue.
  - This will continue until the process completes or it reaches the **base level queue**.

Multilevel feedback queue (from Wikipedia, cont)

- At the base level queue the processes circulate in **round robin** fashion until they complete and leave the system.
- Optionally, if a process **blocks for I/O**, it is **'promoted**' one level, and placed at the end of the next-higher queue. This allows I/O bound processes to be favored by the scheduler and allows processes to 'escape' the base level queue.
- In the multilevel feedback queue, a process is given just one chance to complete at a given queue level before it is forced down to a lower level queue.

Scheduling of normal tasks in Linux

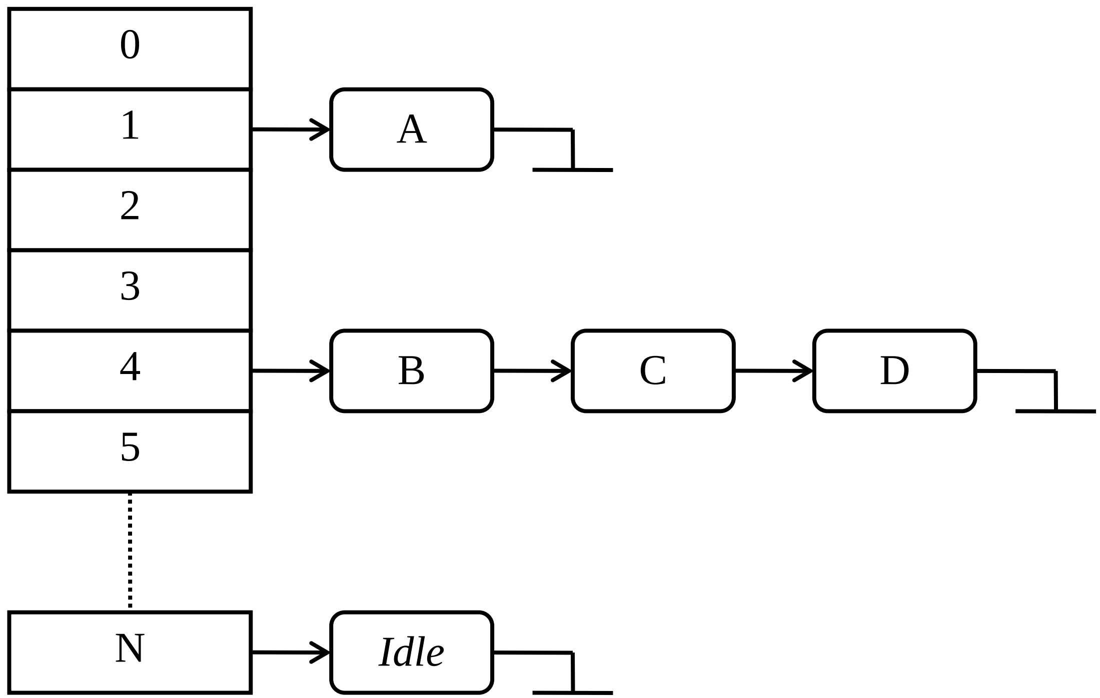
- Linux uses Completely Fair Scheduler - CFS
  - in use since kernel 2.6.23 (2007.)
- Basic CFS principles (theory)
  - if there are N processors and M tasks ($N < M$), each task should get N/M percent of computing time
  - for every task, "wait runtime" is tracked ("deserved time" minus "used time")
  - task with highest wait runtime is chosen by scheduler
- Example (all tasks have same priority – "nice level"):
  - Task τ_1 gets time slice *T*. After the slice is consumed, scheduler updates wait runtimes for all tasks:
    - $wr_1 = wr_1 + T/N - T$ – wait runtime is reduced!
    - $wr_2 = wr_2 + T/N$ – wait runtime is increased
    - $wr_3 = wr_3 + T/N$ – wait runtime is increased
    - …
- "Red-black trees" are used for task organization
  - wait runtime parameter defines task position

Win32 scheduling

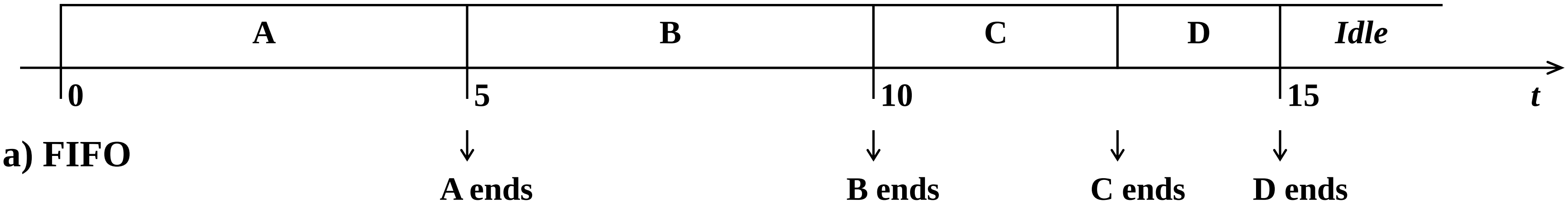
- Thread base priority is calculated from its process *priority class* and thread *priority level*
  - in range from 1 to 15 for normal tasks (16-31 are for RT)
- Priority can be boosted for:
  - “Foreground” (in focus) thread
  - IO bound threads (they rarely use CPU time)
- Priority can be lowered for CPU intensive threads
  - but only to its base priority
- Threads are scheduled to run based on their *scheduling priority* – highest priority thread gets most of the time
  - others get little, just to prevent starvation
- The system treats all threads with the same priority as equal
- The system assigns time slices in a round-robin fashion to all threads with the highest priority

Comparing FIFO, RR and normal task scheduling

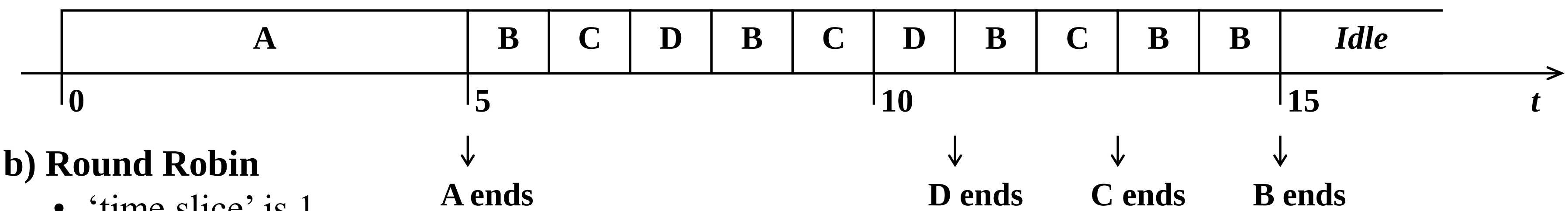
- Example (From book: *Operacijski sustavi*, L. Budin i ostali, 2010. (in Croatian)):
  - four tasks A, B, C and D with processing times $T_P(A) = 5$, $T_P(B) = 5$, $T_P(C) = 3$ and $T_P(D) = 2$ (time units)
  - priorities are shown on picture (lower number – higher priority)



Comparing FIFO, RR and normal task scheduling

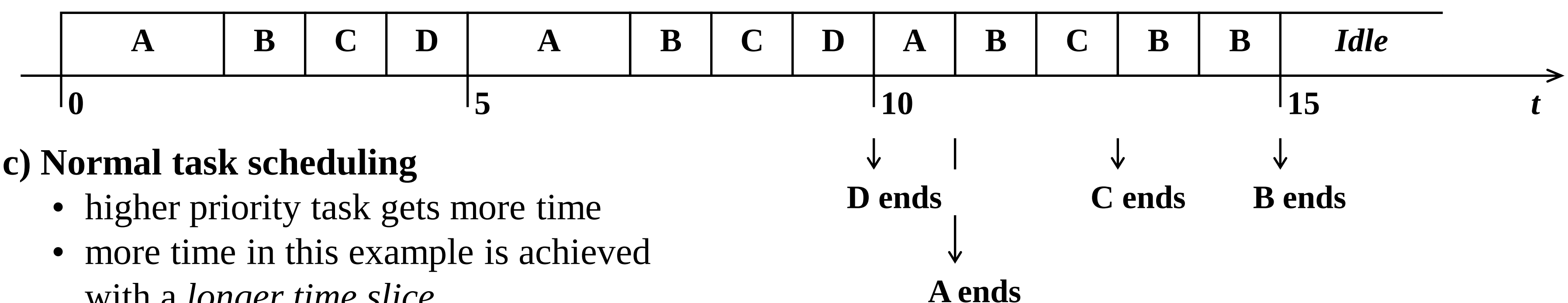


a) FIFO



b) Round Robin

- 'time slice' is 1
- B, C and D have same priority and use RR

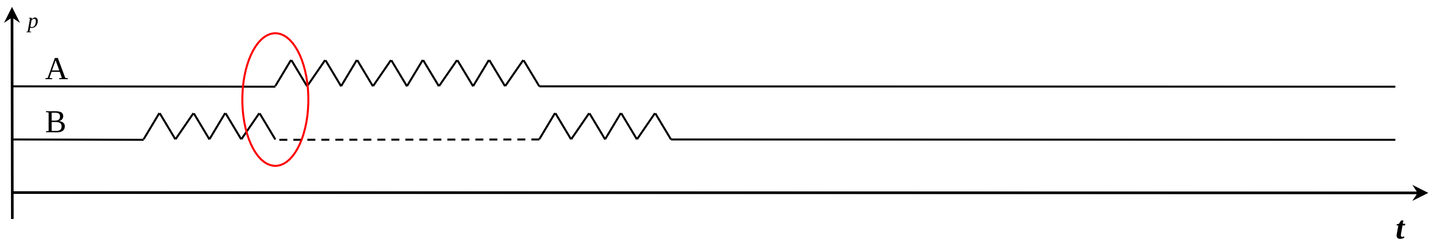


c) Normal task scheduling

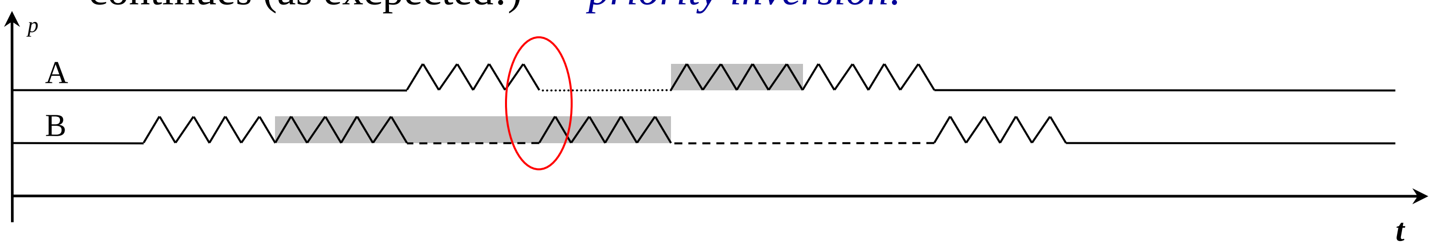
- higher priority task gets more time
- more time in this example is achieved with a *longer time slice*

Priority inversion problem (RT scheduling)

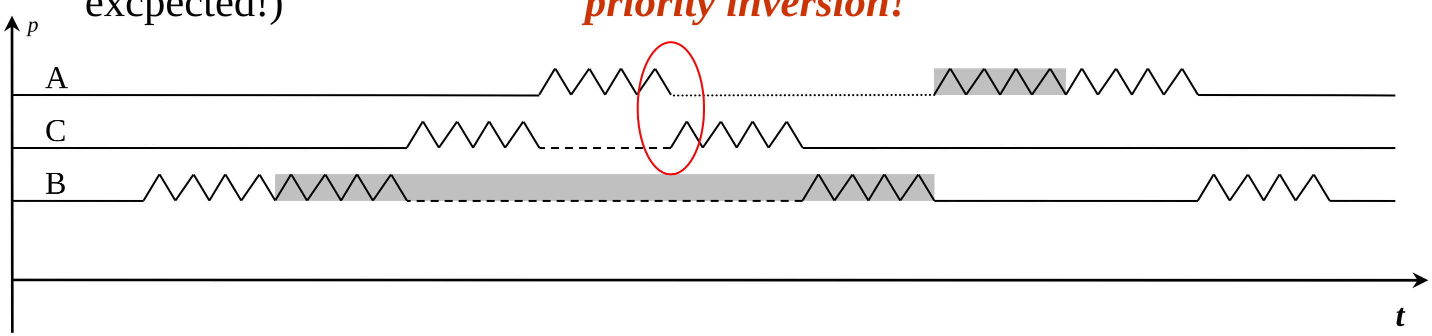
a) high priority task A preempts lower priority task B (as excpected)



b) high priority task A is blocked while lower priority task B, who has locked resource continues (as excpected!) *priority inversion!*



c) high priority task A is blocked while lower priority task C further delays task A (not as excpected!) ***priority inversion!***

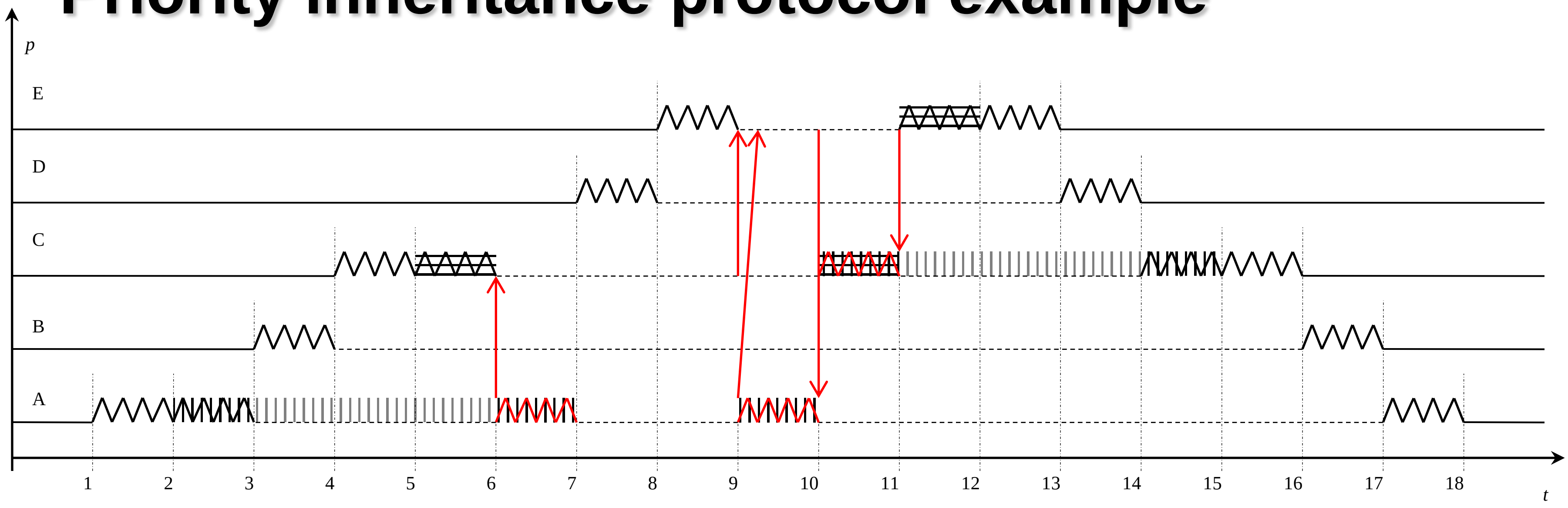


Priority inversion - solutions

- Priority inheritance protocol
  - when a higher priority task A is blocked on resource that is owned by a lower priority task B, then temporarily *boost priority* of B to the level of A, until B releases the resource, then return task B to original priority (that it had before boosting)
  - intermediate tasks will not further delay high priority task!
- Priority ceiling protocol
  - as task acquires resource X, raise task's priority to level defined in e.g. array **`priority_ceiling[X]`**
  - as task releases the resource, return previous priority to task

- Both protocols are embedded into synchronization functions (e.g. **`mutex_lock`**, **`mutex_unlock`**)
- Protocols are essential for RTS!

Priority inheritance protocol example



Event description:

1 – task A, as only ready task start executing (*active task*)

2 – task A acquire S_1 (vertical lines marks S_1 ownership)

3 – task B starts and preempt task A (higher priority at the moment)

4 – task C starts and preempt task B

5 – task C acquire S_2 (horizontal lines marks S_1 ownership)

6 – task C requires S_1 owned by task A and is blocked on it; task A inherit task C priority and continue its execution (with priority of task C)

7 – task D starts and preempt task A

8 – task E starts and preempt task D

9 – task E requires S_2 owned by task C and is blocked on it; task C inherit task E priority, but since task C is blocked by task A, task A inherits new priority of task C – priority of task E (implicitly task A inherits task E priority)

10 – task A releases S_1, and original priority is returned to task A (as it was before acquiring resource S_1); task C is released, S_1 is given to it; task C continues execution (highest priority ready task – inherited priority of blocked task E)

11 – task C releases S_2, and original priority is returned to task C; task E is released, S_2 is given to it; task E continues executing

12 – task E releases S_2 and continues executing

13 – task E finishes; task D continues executing

14 – task D finishes; task C continues executing

15 – task C releases S_1 continues executing

16 – task C finishes; task B continues executing

17 – task B finishes; task A continues executing

18 – task A finishes

Source for example: nas_prio.c

Scheduling - summary

- RT tasks:
  - **Priority based scheduling** is widely used as primary scheduling principle
  - When more tasks with same highest priority are ready: **FIFO** or **Round Robin** are used
  - FIFO and RR are implemented on most (all) systems
    - on all systems the implementation is identical (in respect to scheduling decisions)

- Normal tasks:
  - **Fair-share principle** is used (adjusted with priority)
  - Higher priority provides more CPU time
  - It doesn't guarantee immediate preemption of lower priority tasks
  - Differently implemented on different systems

Task scheduling

POSIX interface

Thread scheduling: policy and priority

- Scheduling is defined by:
  - **thread scheduling policy**:
    - `SCHED_FIFO`
    - `SCHED_RR`
    - `SCHED_SPORADIC`
    - `SCHED_OTHER`
  - **thread priority**
    - number in system specific range
      - `int sched_get_priority_min(int policy);`
      - `int sched_get_priority_max(int policy);`
- Policy and priority may be set on thread creation and/or changed later

Setting scheduling parameters for new thread

- Parameter of type **pthread_attr_t** that is passed at thread creation (**pthread_create**) contain scheduling attributes

- Changing/setting scheduling parameters function
  - **pthread_attr_setinheritsched**
    - inherit parameters from parent thread (PTHREAD_INHERIT_SCHED, PTHREAD_EXPLICIT_SCHED)?
  - **pthread_attr_setschedpolicy**
    - set policy (SCHED_FIFO, SCHED_RR, SCHED_SPORADIC or SCHED_OTHER)
  - **pthread_attr_setschedparam**
    - set thread priority

Changing scheduling parameters for thread

- Change policy and priority:
  - ```
    pthread_setschedparam(
            pthread_t thread,
            int policy,
            const struct sched_param *param)
    ```
- or just priority
  - ```
    pthread_setschedprio(
            pthread_t thread,
            int prio);
    ```
- Equivalent behavior on process level:
  - `sched_setscheduler(pid, policy, param)`
  - `sched_setparam(pid, param)`

Synchronization that influence scheduling

- For mutex (**`pthread_mutex_lock/unlock`**)
  - priority inheritance or priority ceiling can be set:

```
int pthread_mutexattr_setprotocol(
        pthread_mutexattr_t *attr,
        int protocol);
```

  - values for ***protocol***:

    `PTHREAD_PRIO_INHERIT` – priority inheritance
    `PTHREAD_PRIO_PROTECT` – priority ceiling
    `PTHREAD_PRIO_NONE`

  - Setting value for priority ceiling:

```
int pthread_mutex_setprioceiling(
        pthread_mutex_t *mutex,
        int prioceiling,
        int *old_ceiling);
```