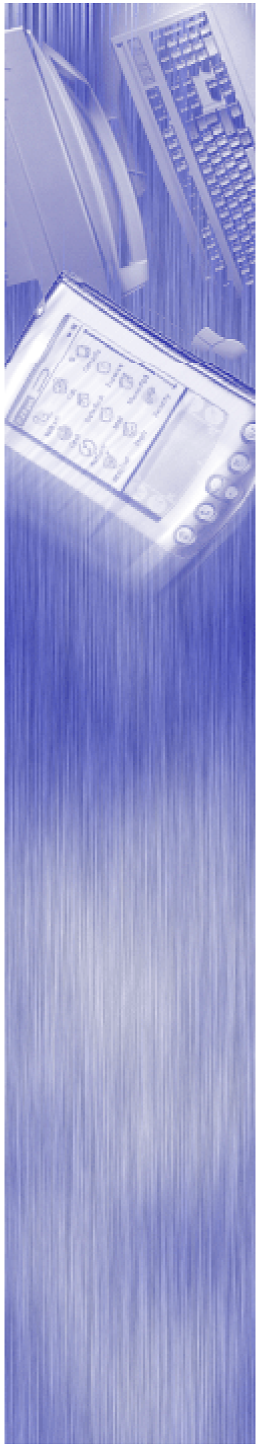


CS632/SEP564: Embedded Operating Systems (Fall 2008)

CPU Scheduling

KAIST

UNIX Scheduler (1)

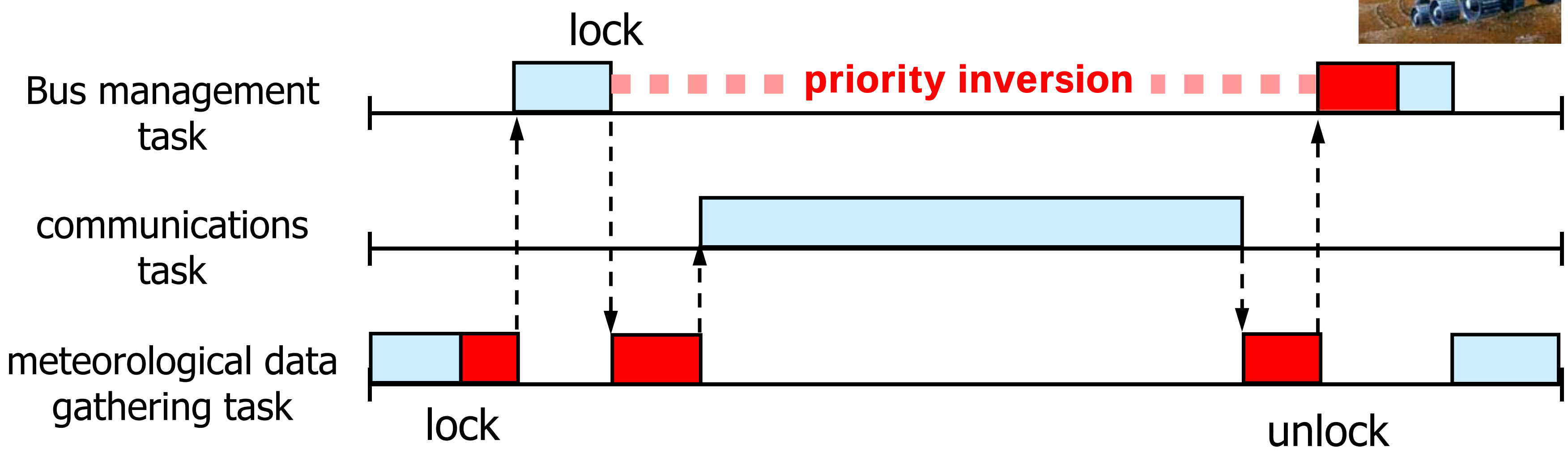
- **Characteristics**
  - Priority-based
    - The process with the highest priority always runs.
    - 3 – 4 classes spanning ~170 priority levels (Solaris 2)
  - Preemptive
  - Time-shared
    - Based on timeslice (or quantum)
  - MLFQ (Multi-Level Feedback Queue)
    - Priority scheduling across queues, RR within a queue.
    - Processes dynamically change priority.

UNIX Scheduler (2)

- **General principles**
  - Favor I/O-bound processes over CPU-bound processes
  - Aging to avoid starvation
  - Priority inversion?

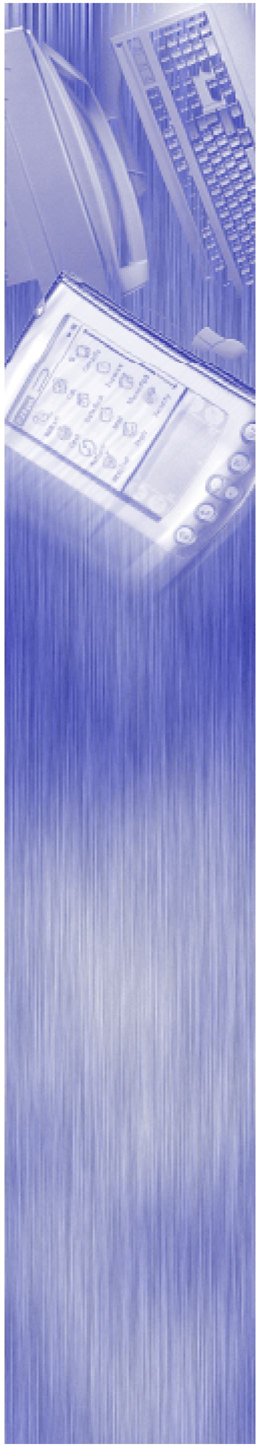
UNIX Scheduler (3)

- **Priority inversion problem**
  - A situation where a higher-priority job is unable to run because a lower-priority job is holding a resource it needs, such as a lock.
  - *What really happened on Mars?*



Scheduling Parameters

- **Priority**
  - Static priority vs. dynamic priority
  - How to distinguish I/O-bound processes from CPU-bound processes?
  - When and how much the priority of I/O-bound process is increased?
  - How to perform aging?
- **Timeslice (quantum)**
  - I/O-bound processes do not need longer timeslices.
  - CPU-bound processes crave long timeslices.
  - Short timeslice: switching overhead
  - Long timeslice: poor response time for interactive applications?
  - Higher priority means longer timeslice?



CS632/SEP564: Embedded Operating Systems (Fall 2008)

Linux Scheduling

Linux 2.4 Scheduler (1)

- **Priorities**
  - Static priority
    - The maximum size of the time slice a process should be allowed before being forced to allow other processes to complete for the CPU.
  - Dynamic priority
    - The amount of time remaining in this time slice; declines with time as long as the process has the CPU.
    - When its dynamic priority falls to 0, the process is marked for rescheduling.
  - Real-time priority
    - Only real-time processes have the real-time priority.
    - Higher real-time priority values always beat lower values.

Linux 2.4 Scheduler (2)

▪ Related fields in the task structure

long counter;	time remaining in the task's current quantum (represents dynamic priority)
long nice;	task's nice value, -20 to +19. (represents static priority)
unsigned long policy;	SCHED_OTHER, SCHED_FIFO, SCHED_RR
struct mm_struct *mm;	points to the memory descriptor
int processor;	processor ID on which the task will execute
unsigned long cpus_runnable;	~0 if the task is not running on any CPU (1<<cpu) if it's running on a CPU
unsigned long cpus_allowed;	CPUs allowed to run
struct list_head run_list;	head of the run queue
unsigned long rt_priority;	real-time priority

Linux 2.4 Scheduler (3)

- **Scheduling quanta**
  - Linux gets a timer interrupt or a *tick* once every 10ms on IA-32. (HZ=100)
    - Alpha port of the Linux kernel issues 1024 timer interrupts per second.
  - Linux wants the time slice to be around 50ms.
    - Decreased from 200ms (in v2.2)

```
/* v2.4 */
#if HZ < 200
#define TICK_SCALE(x)    ((x) >> 2)
#endif
#define NICE_TO_TICKS(nice)    (TICK_SCALE(20-(nice))+1)

/* v2.2 */
#define DEF_PRIORITY     (20*HZ/100)
```

Linux 2.4 Scheduler (4)

- **Epochs**
  - The Linux scheduling algorithm works by dividing the CPU time into epochs.
    - In a single epoch, every process has a specified time quantum whose duration is computed when the epoch begins.
    - The epoch ends when all runnable processes have exhausted their quantum.
    - The scheduler recomputes the time-quantum durations of all processes and a new epoch begins.
  - The base time quantum of a process is computed based on the nice value.

Linux 2.4 Scheduler (5)

- **Selecting the next process to run**

```
repeat_schedule:
        next = idle_task(this_cpu);
        c = -1000;
        list_for_each(tmp, &runqueue_head) {
                p = list_entry(tmp, struct task_struct, run_list);
                if (can_schedule(p, this_cpu)) {
                        int weight = goodness(p, this_cpu,
                                                          prev->active_mm);
                        if (weight > c)
                                c = weight, next = p;
                }
        }
```

Linux 2.4 Scheduler (6)

- **Recalculating counters**

```
if (unlikely(!c)) {                    /* New epoch begins ... */
        struct task_struct *p;

        spin_unlock_irq(&runqueue_lock);
        read_lock(&tasklist_lock);
        for_each_task(p)
                p->counter = (p->counter >> 1) +
                                        NICE_TO_TICKS(p->nice);
        read_unlock(&tasklist_lock);
        spin_lock_irq(&runqueue_lock);
        goto repeat_schedule;
}
```

Linux 2.4 Scheduler (7)

- **Calculating goodness()**

```
static inline int goodness (p, this_cpu, this_mm) {
        int weight = -1;
        if (p->policy == SCHED_OTHER) {
            weight = p->counter;
            if (!weight) goto out;
            if (p->mm == this_mm || !p->mm)
               weight += 1;
            weight += 20 - p->nice;
            goto out;
        }
        weight = 1000 + p->rt_priority;
out:  return weight;
}
```

weight = 0
  p has exhausted its quantum.
0 < weight < 1000
  p is a conventional process.
weight >= 1000
  p is a real-time process.

Linux 2.4 Scheduler (8)

- **Problem: NOT so scalable!**
  - A single run queue is protected by a run queue lock.
    - As the number of processors increases, the lock contention increases.
  - It is expensive to recalculate goodness() for every task on every invocation of the scheduler.
    - A profile of the kernel taken during the VolanoMark runs shows that 37-55% of total time spent in the kernel is spent in the scheduler.
    - The VolanoMark benchmark establishes a socket connection to a chat server for each simulated chat room user.
    - For a 5 to 25-room simulation, the kernel must potentially deal with 400 to 2000 threads.

Linux 2.4 Scheduler (9)

- **Other problems**
  - The predefined quantum is too large for high-end machines that have a very high system load.
  - I/O-bound process boosting strategy is not optimal.
    - When the number of runnable processes is very large, I/O-bound processes are seldom boosted.
    - I/O-bound processes with no user interaction?
    - Interactive processes that are also CPU-bound?
  - Support for real-time applications is weak.
    - Nonpreemptive kernel?
    - Priority inversion?
    - Hidden scheduling: Kernel performs work asynchronously on behalf of threads, without considering priority of requester.

Linux 2.6 Scheduler (1)

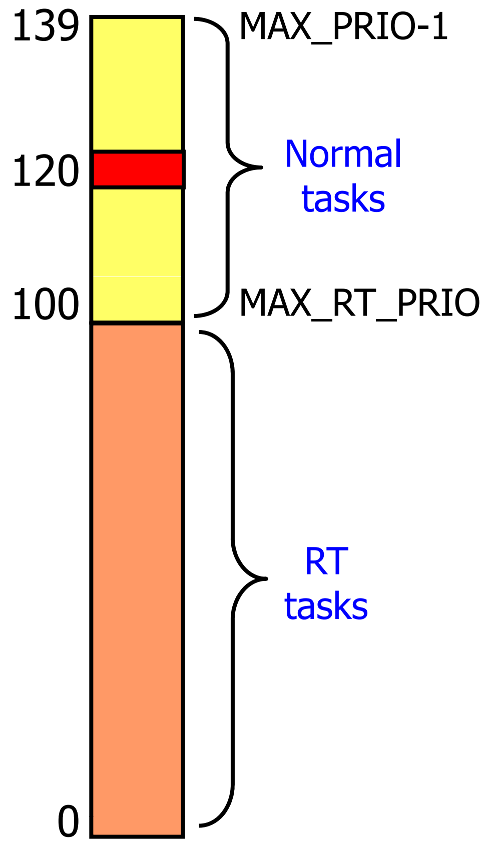
- **Goals**
  - Implement fully O(1) scheduling.
  - Implement perfect SMP scalability.
    - Each processor has its own locking and individual runqueue.
  - Implement improved SMP affinity.
    - Only migrate tasks from one CPU to another to resolve imbalances in runqueue size.
  - Provide good interactive performance.
  - Provide fairness.
  - Optimize for the common case of only 1-2 runnable processes.
    - Yet scale well to multiple processors each with many processes.

Linux 2.6 Scheduler (2)

▪ Priority

- Static priority (t->static_prio)
  - 100 (highest) ~ 139 (lowest)
  - Default static priority: 120
  - Nice value: -20 ~ +19

- Dynamic priority (t->prio)
  - effective_prio() determines the dynamic priority of the task
    = t->static_prio – bonus(t) + 5
  - bonus(t): 0 ~ 10 based on the interactivity of the task
  - Same as static priority for RT tasks



Linux 2.6 Scheduler (3)

- **Interactivity bonus**
  - Bonus(t): 0 ~ 10
    - Bonus < 5: penalty (dynamic priority decreased)
    - Bonus = b if $b \cdot 100\text{ms} \le$ average sleep time < $(b+1) \cdot 100\text{ms}$

*bonus(t) = (t->sleep_avg * 10) / MAX_SLEEP_AVG*

  - Average sleep time (t->sleep_avg)
    - The average number of nanoseconds that the process spent while sleeping.
    - Decreases while a process is running.
    - Never become larger than MAX_SLEEP_AVG (= 1 second).

Linux 2.6 Scheduler (4)

- **Base time quantum**
  - The static priority determines the base time quantum.

$$base\ time\ quantum\ (ms) = \begin{cases} (140 - static\_prio) \times 20 & (if\ static\_prio < 120) \\ (140 - static\_prio) \times 5 & (if\ static\_prio \geq 120) \end{cases}$$

  - – Highest priority: 800ms
  - – Default priority: 100ms
  - – Lowest priority: 5ms
  - – Higher priority processes usually get longer slices of CPU time.
- On fork(): the new child and the parent split the parent's remaining timeslice.

Linux 2.6 Scheduler (5)

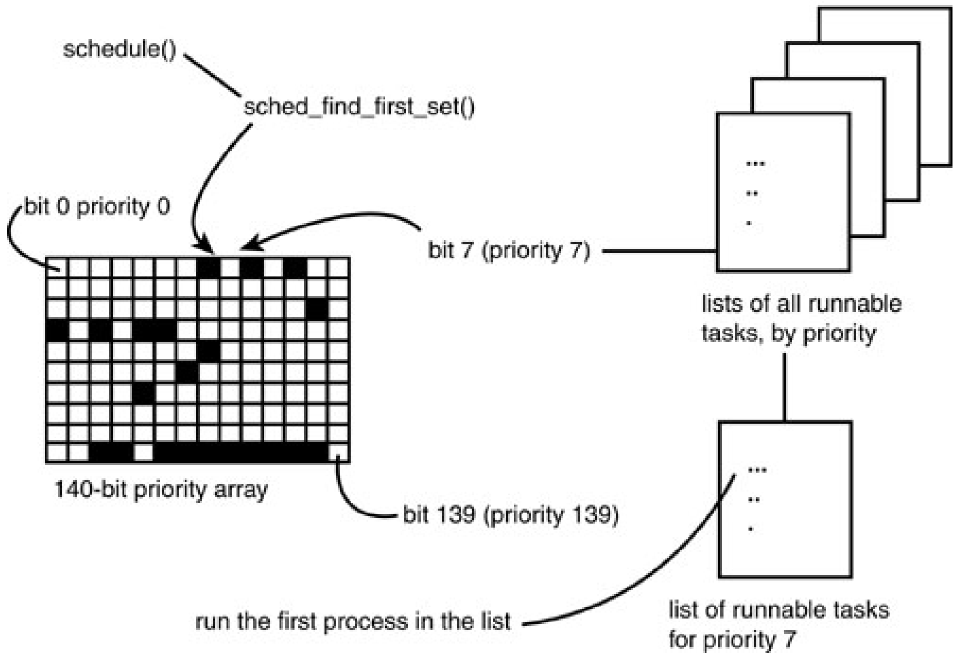
- **Interactivity of a task**
  - TASK_INTERACTIVE(t) is true if the task is classified as interactive.
    - True if (t->prio <= t->static_prio – DELTA(t))
      i.e., bonus(t) >= DELTA(t) + 5
      where DELTA(t) = (t->static_prio / 4 – 28) (value in [-3, +6])
  - It is easier for high priority task to become interactive.
    - A task with highest static priority (100) is considered interactive when its bonus value exceeds 2 (e.g., avg. sleep time 200ms).
    - A task with lowest static priority (139) is never considered interactive.

Linux 2.6 Scheduler (6)

- **Runqueue**
  - The list of runnable processes on a given processor.
    - There is only one runqueue per processor.
    - Implemented as a per-CPU structure
    - Each runnable process is on exactly one runqueue.
  - Runqueue should be locked before it can be accessed.
    - To avoid deadlock, code that wants to lock multiple runqueues needs always to obtain the locks in the same order: by ascending runqueue address
  - Each runqueue contains two priority arrays.
    - The active array
    - The expired array

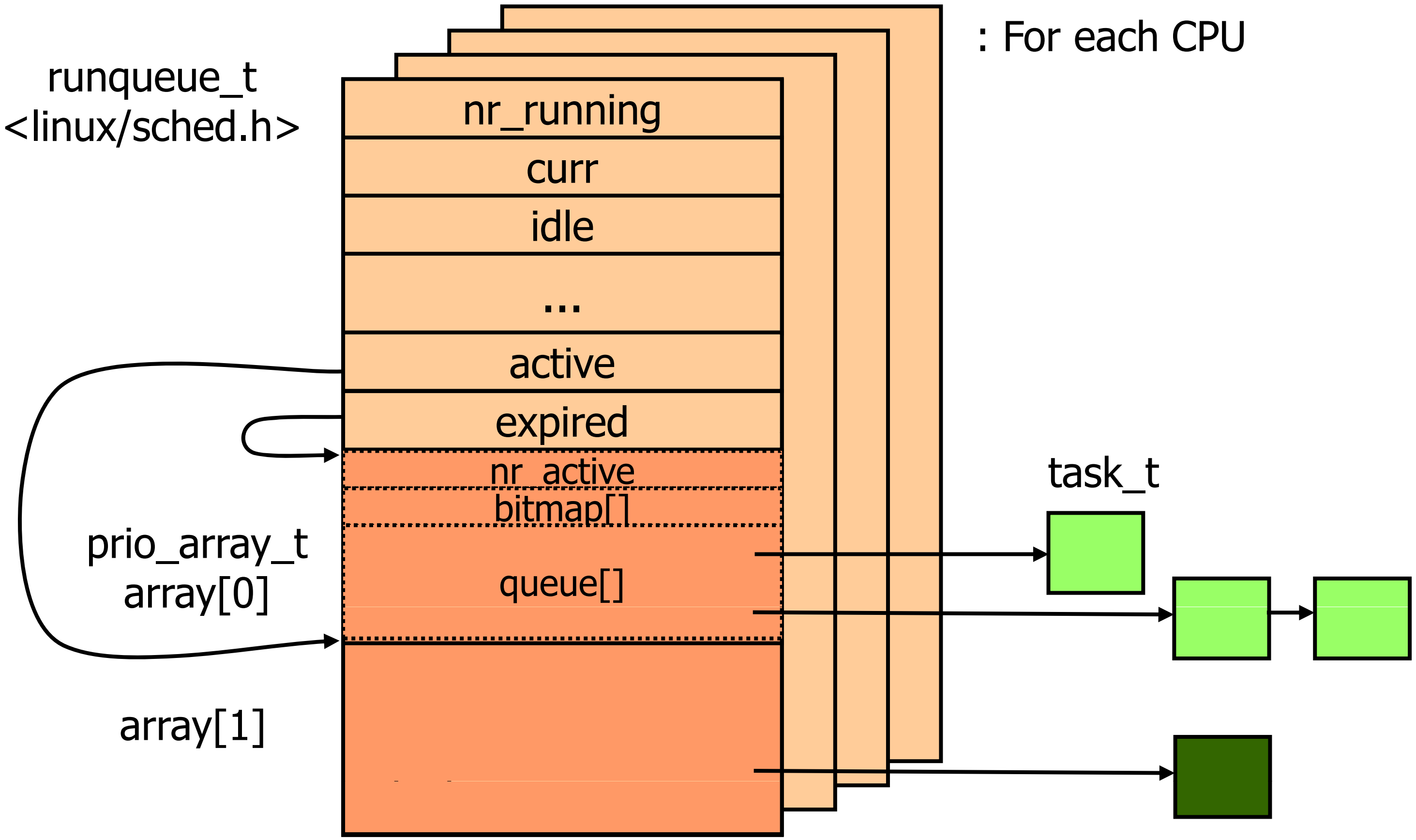
Linux 2.6 Scheduler (7)

- **Priority arrays**
  - Each priority array contains a queue of runnable processes per each priority level.
    - By default, there is 140 priority levels.
  - The priority array also has a bitmap.
    - Used to efficiently discover the highest priority runnable task.
    - At least one bit for every priority (five 32-bit words for 140 priority levels)



Linux 2.6 Scheduler (8)

- **Runqueues and priority arrays**



Linux 2.6 Scheduler (9)

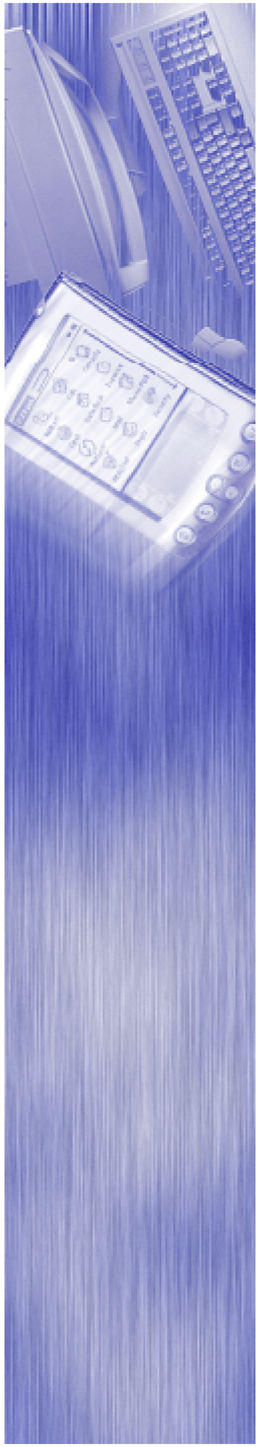
- **O(1) Scheduling**
  - When a task of a given priority becomes runnable, the corresponding bit in the bitmap is set to one.
  - Finding the highest priority task = finding the first set bit in the bitmap.
    - The time to find the first set bit is constant and unaffected by the number of running processes.
    - The *find first set* algorithm can be implemented efficiently.
  - Within a given priority, tasks are scheduled round robin.
  - When each task's timeslice reaches zero, its timeslice is calculated before it is moved to the expired array.
    - Recalculating all the timeslices is then as simple as just switching the active and expired arrays.

Linux 2.6 Scheduler (10)

- **Special handling for interactive processes**
  - An active interactive process that finishes its time quantum usually remains active.
  - Moved into the expired array later
    - if the first expired process had to wait for more than 1000 ticks * (the number of runnable processes + 1)
    - if an expired process has higher static priority (lower value) than the interactive process
  - The time quantum of interactive processes with high static priorities is split into several pieces of TIMESLICE_GRANULARITY size
    - Prevent them from monopolizing the CPU
    - Higher bonus(t) has shorter TIMESLICE_GRANULARITY.

Linux 2.6 Scheduler (11)

- **schedule()**
  - Directly invoked when the current process is blocked.
    1. Insert **current** in the proper wait queue.
    2. Set the state to TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE
    3. Invoke schedule()
    4. Checks whether the resource is available; If not go to step 2.
    5. Remove **current** from the wait queue.
  - Lazy invocation
    - TIF_NEED_RESCHED flag of current is checked before resuming the execution of a user mode process.
    - When **current** has used up its quantum of CPU time
    - When a process is woken up and its priority is higher than that of the current process
    - When a sched_setscheduler() system call is issued.



CS632/SEP564: Embedded Operating Systems (Fall 2008)

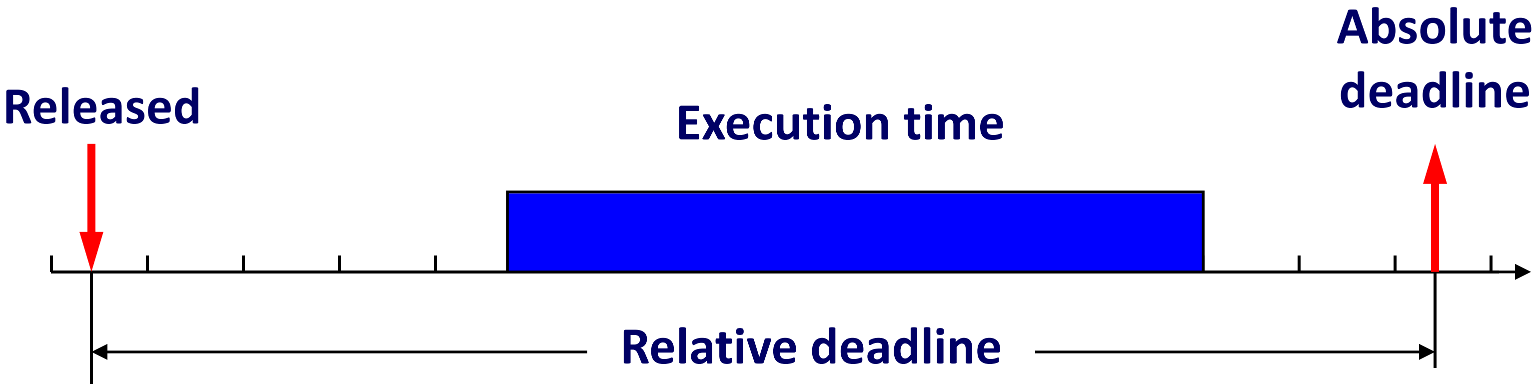
Real-Time Scheduling

Credit: Many slides are borrowed from Insup Lee (CIS700)

KAIST

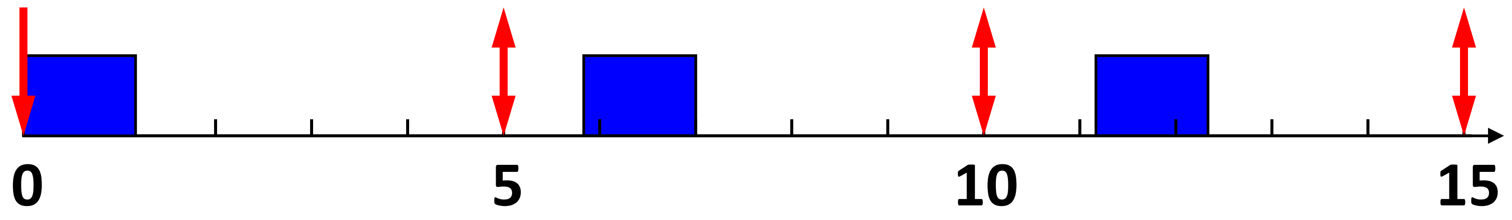
Real-Time Workload

- **Job (unit of work)**
  - A computation, a file read, a message transmission, etc.
- **Attributes**
  - Resources required to make progress
  - Timing parameters



Real-Time Task

- **Task**
  - A sequence of similar jobs
  - Periodic task (p, e)
    - Its jobs repeat regularly
    - Period p = inter-release time ($0 < p$)
    - Execution time e = maximum execution time ($0 < e < p$)
    - Utilization $U = e/p$



Deadlines: Hard vs. Soft

- **Hard deadline**
  - Disastrous or very serious consequences may occur if the deadline is missed
  - Validation is essential: can all the deadlines be met, even under worst-case scenario?
  - Deterministic guarantees
- **Soft deadline**
  - Ideally, the deadline should be met for maximum performance. The performance degrades in case of deadline misses.
  - Best effort approaches / statistical guarantees

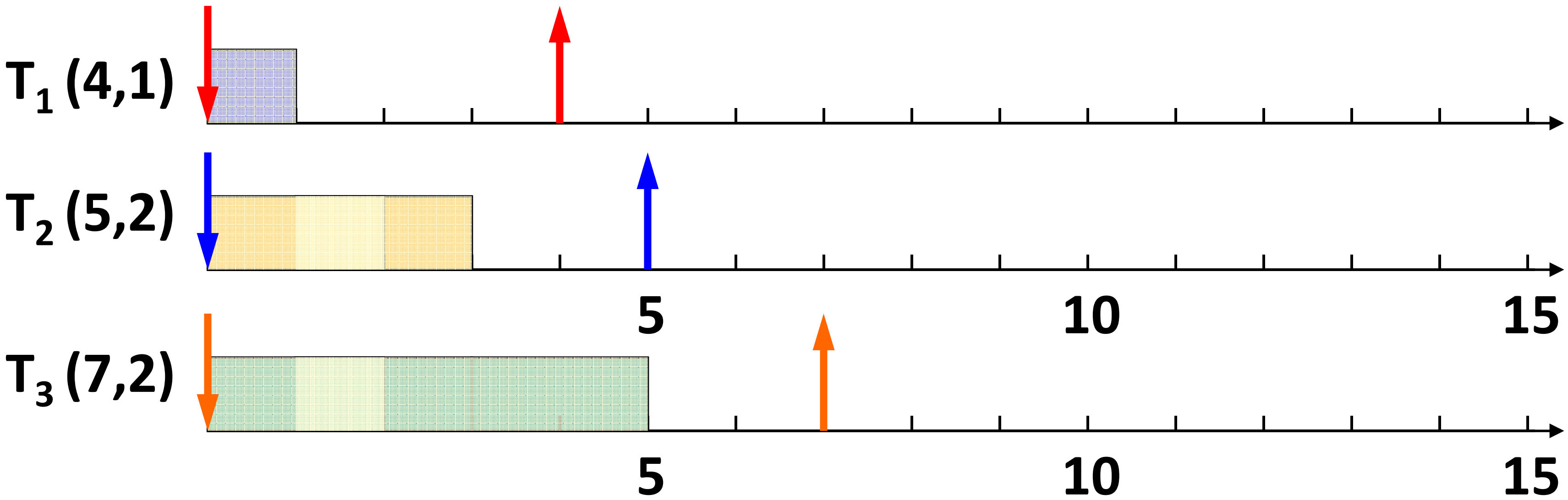
Real-time Scheduling

- **Schedulability**
  - Property indicating whether a real-time system (a set of real-time tasks) can meet their deadlines

- **Real-time scheduling**
  - Determines the order of real-time task executions
  - Static-priority scheduling: RM
  - Dynamic-priority scheduling: EDF

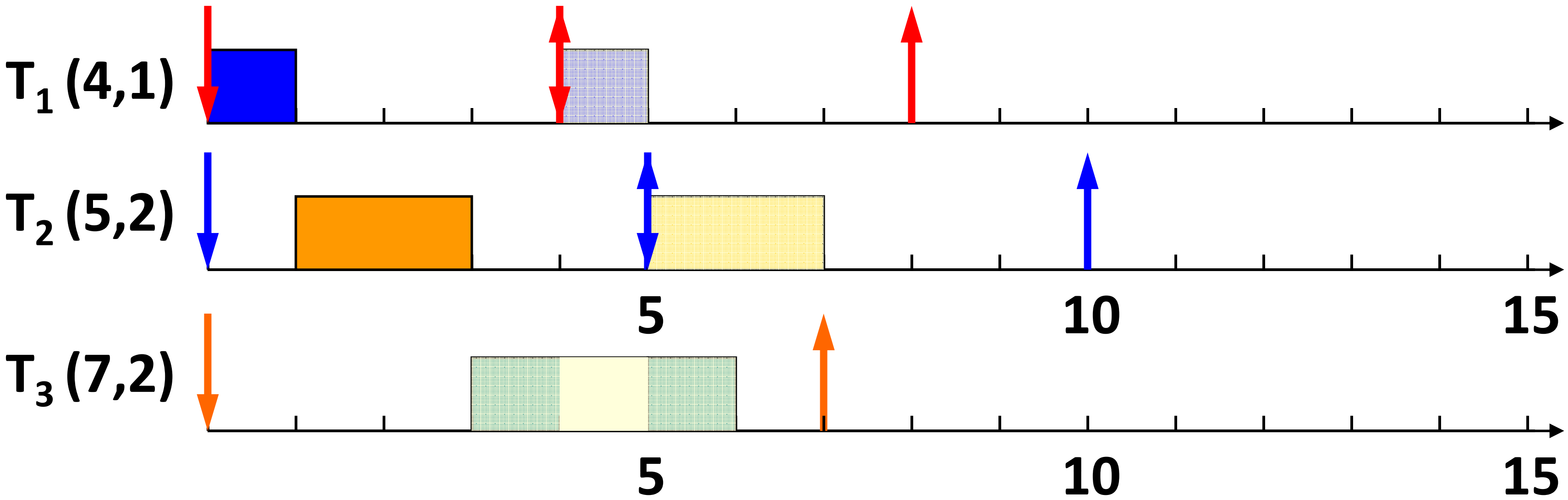
RM

- **Rate Monotonic**
  - Optimal static-priority scheduling
  - Assigns priority according to period
  - A task with a shorter period has a higher priority
  - Executes a job with the shortest period



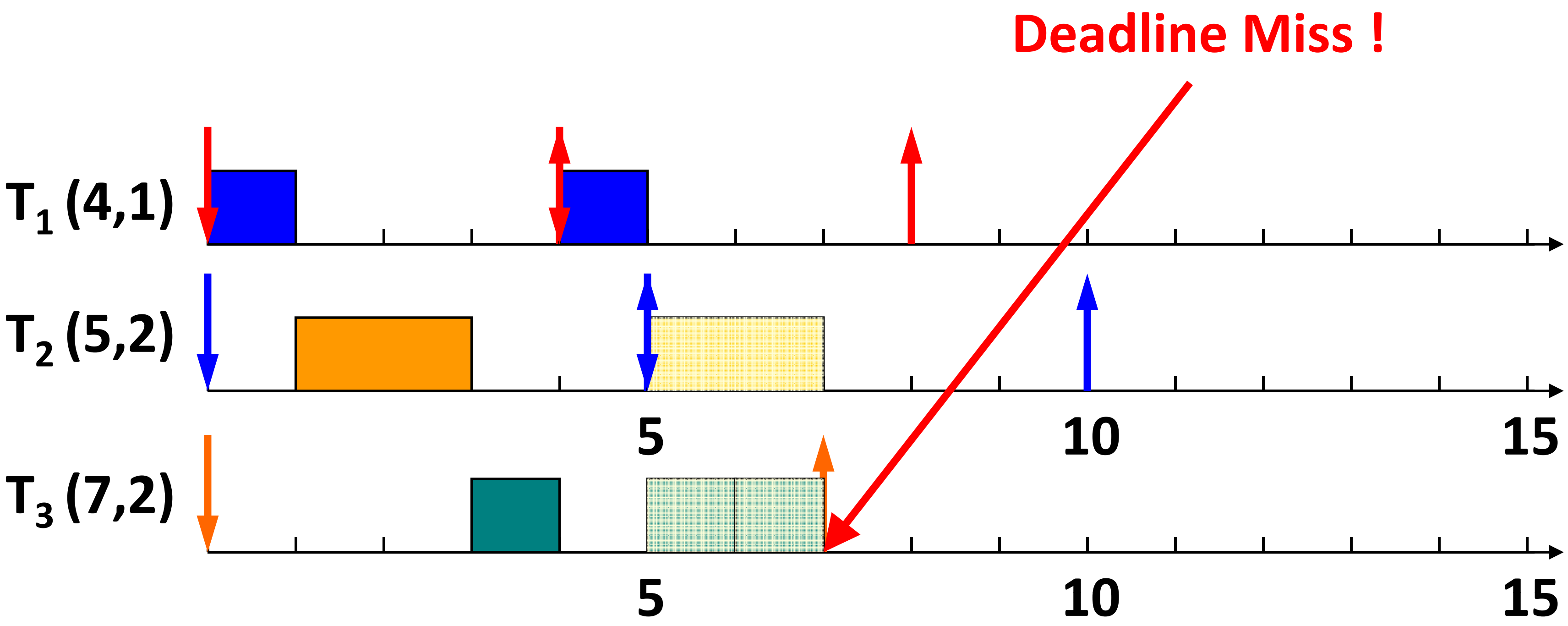
RM

- **Rate Monotonic**
  - Executes a job with the shortest period



RM

- **Rate Monotonic**
  - Executes a job with the shortest period



RM

- **Utilization bound**
  - Real-time system is schedulable under RM if

$$\sum U_i \le n(2^{1/n} - 1)$$

  - Example: $T_1(4,1)$, $T_2(5,1)$, $T_3(10,1)$

$$\sum U_i = 1/4 + 1/5 + 1/10 = 0.55$$

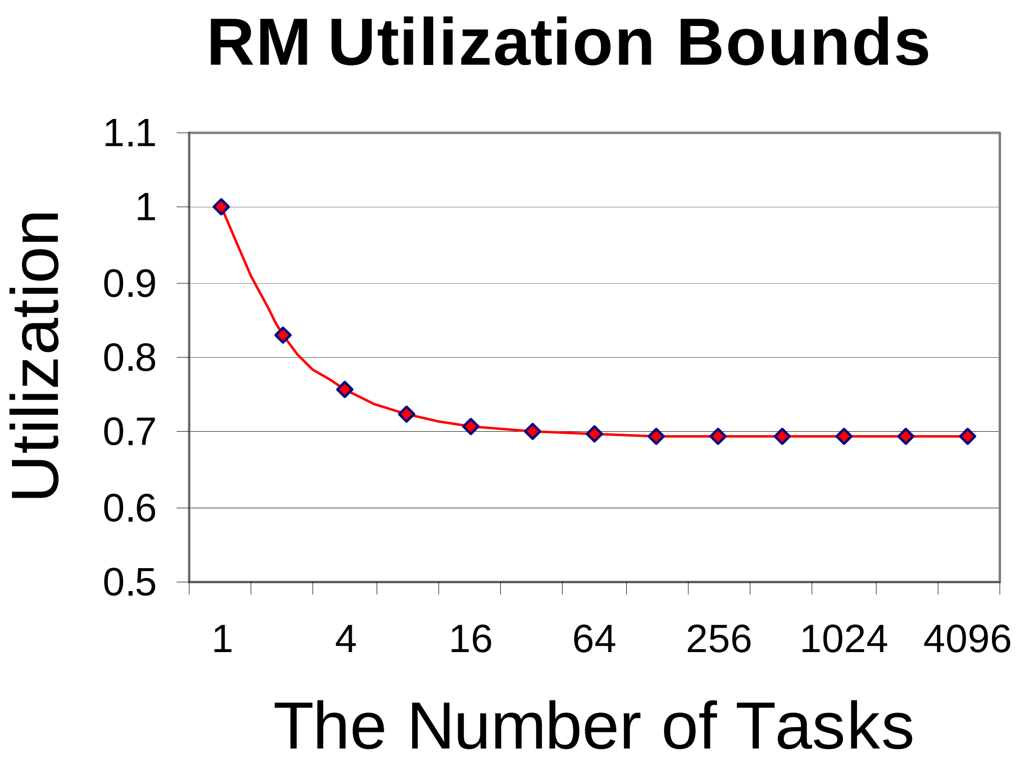
$$3(2^{1/3} - 1) \approx 0.78$$

Thus, $\{T_1, T_2, T_3\}$ is schedulable under RM.

RM

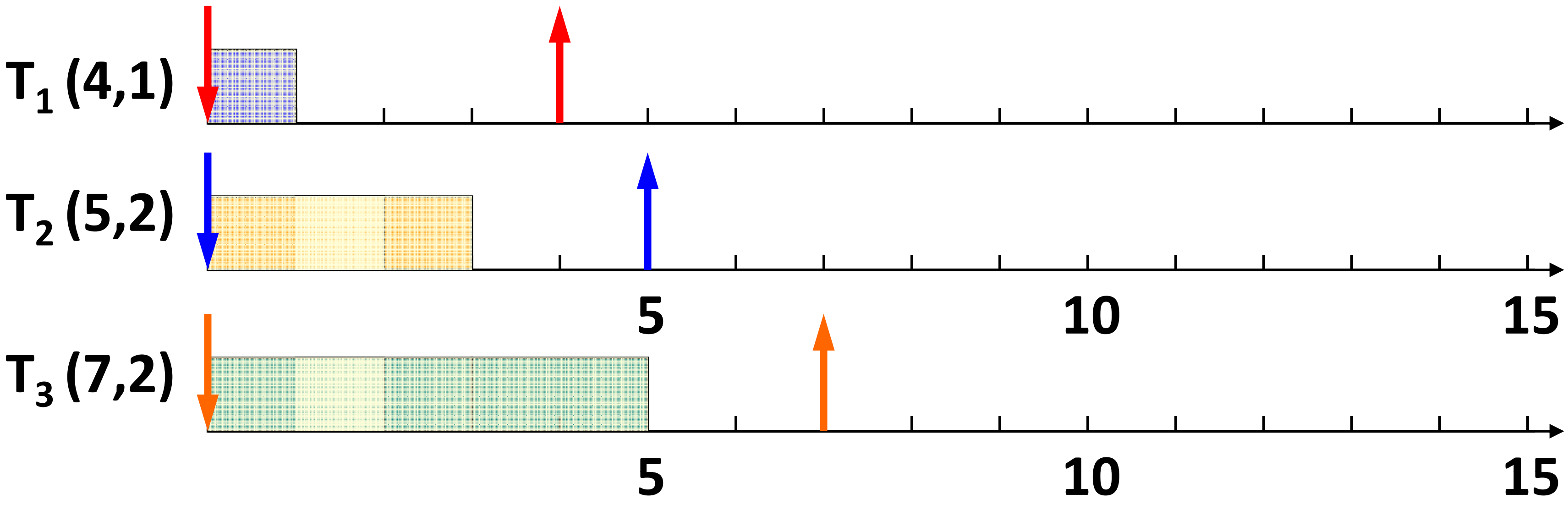
- **Utilization bound (cont'd)**

$$\sum U_i \le n(2^{1/n} - 1)$$



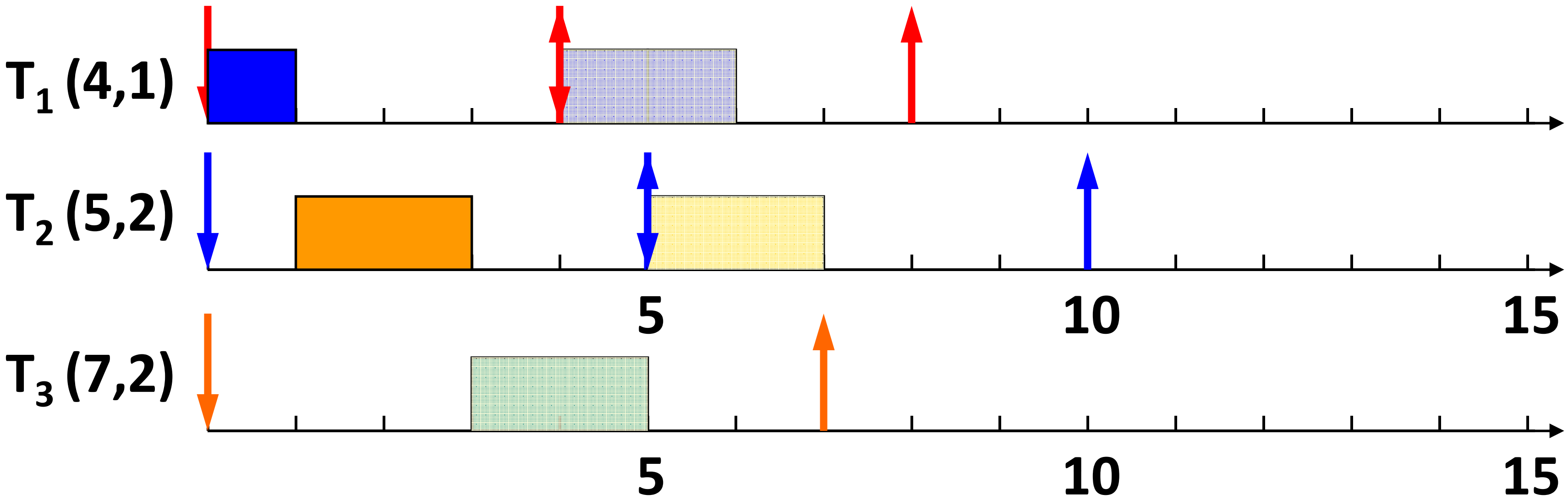
EDF

- **Earliest Deadline First**
  - Optimal dynamic priority scheduling
  - A task with a shorter deadline has a higher priority
  - Executes a job with the earliest deadline



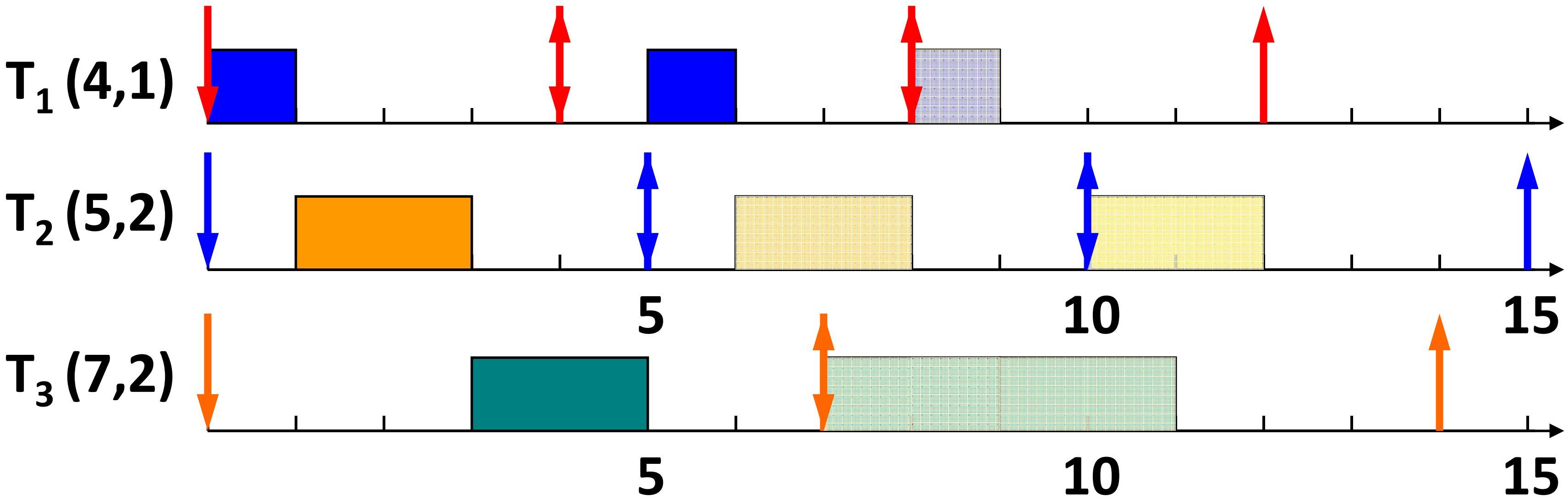
EDF

- **Earliest Deadline First**
  - Executes a job with the earliest deadline



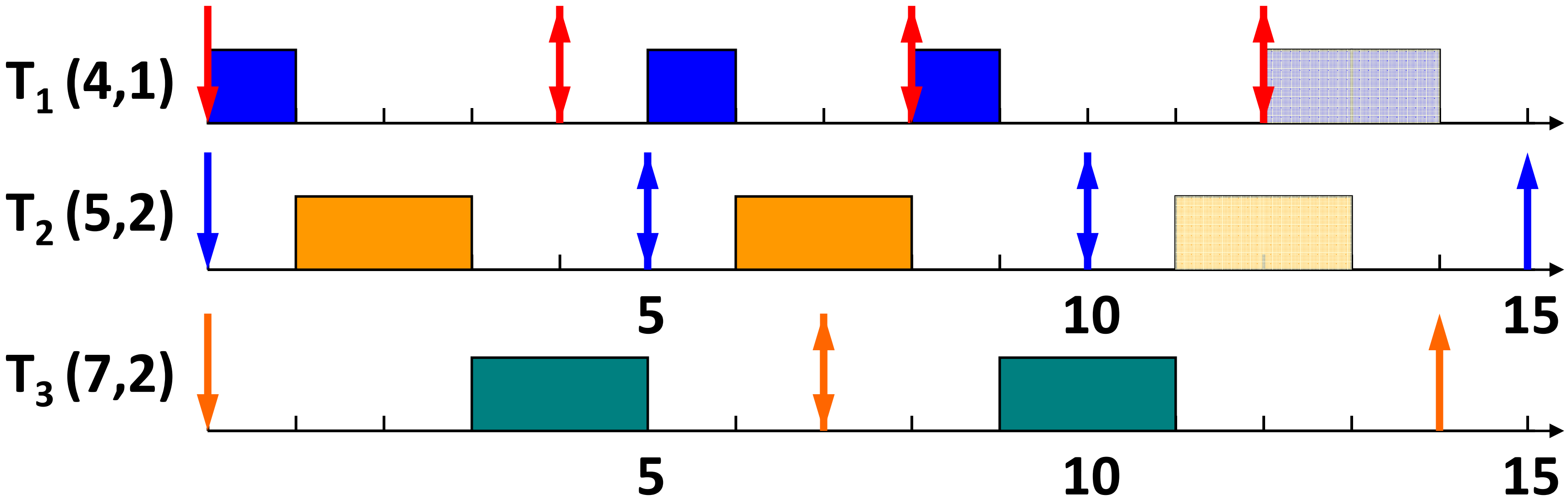
EDF

- **Earliest Deadline First**
  - Executes a job with the earliest deadline



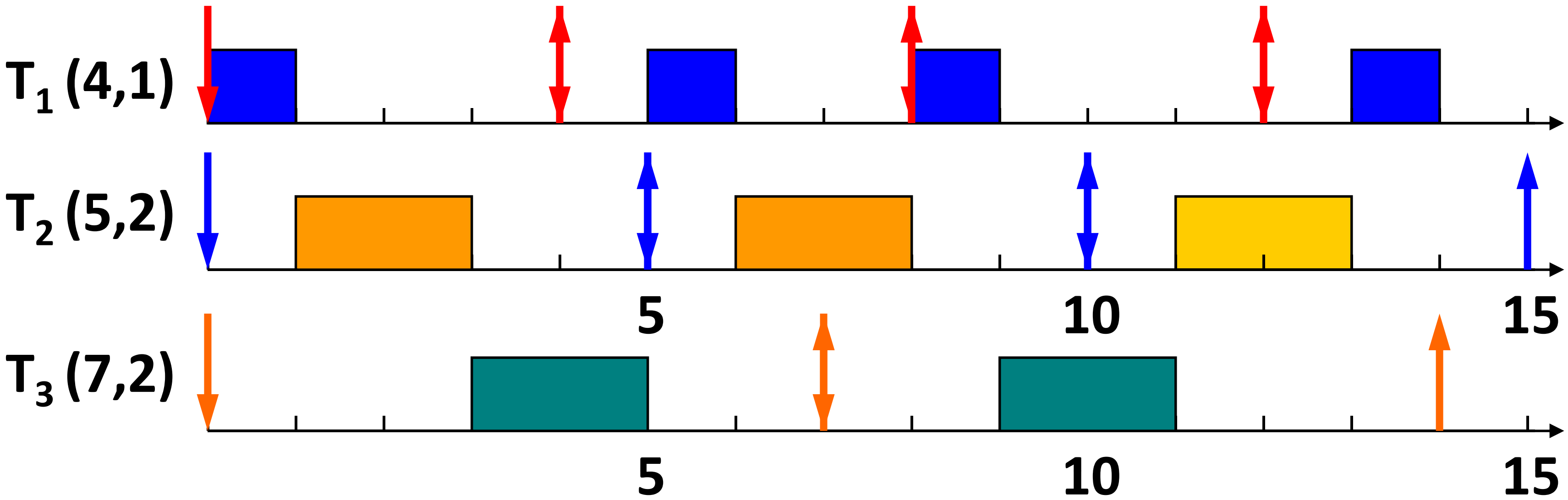
EDF

- **Earliest Deadline First**
  - Executes a job with the earliest deadline



EDF

- **Optimal scheduling algorithm**
  - If there is a schedule for a set of real-time tasks, EDF can schedule it.



EDF

- **Utilization bound**
  - Real-time system is schedulable under EDF if and only if

$$\sum U_i \leq 1$$

(cf) Liu & Layland, "Scheduling algorithms for multi-programming in a hard-real-time environment," *Journal of ACM*, 1973.

RM vs. EDF (1)

- **Rate Monotonic**
  - Simpler implementation, even in systems without explicit support for timing constraints (periods, deadlines)
  - Predictability for the highest priority tasks
- **EDF**
  - Full processor utilization
  - Misbehavior during overload conditions
  - Additional implementation complexity and runtime overhead due to dynamic priority management

RM vs. EDF (2)

- **Assumptions**
  - All tasks are periodic.
  - All tasks are released at the beginning of period and have a deadline equal to their period.
  - All tasks are independent.
  - All tasks have a fixed computation time, or at least a fixed upper bound on their computation times, which is less than or equal to their period.
  - No task may voluntarily suspend itself.
  - All tasks are fully preemptible.
  - All overheads are assumed to be 0.
  - There is just one processor.

In Reality (1)

- **Linux Scheduling policies**
  - SCHED_NORMAL (= SCHED_OTHER in v2.4.x)
  - SCHED_FIFO
    - A real-time process runs until it either blocks on I/O, explicitly yields the CPU, or is preempted by another real-time process with a higher rt_priority.
    - Acts as if it has no time slice.
  - SCHED_RR
    - It's the same as SCHED_FIFO, except that time slices do matter.
    - When a SCHED_RR process's time slice expires, it goes to the back of the list of SCHED_FIFO and SCHED_RR processes with the same rt_priority.

In Reality (2)

- **A RT task is replaced by another task when:**
  - The task is preempted by another task having higher real-time priority
  - The task performs a blocking operation, and it is put to sleep
  - The task is stopped or it is killed
  - The task voluntarily relinquishes the CPU by invoking the sched_yield()
  - The task is in SCHED_RR policy and it has exhausted its time quantum
    - The time quantum depends not on the RT-priority, but on the static priority of the task

Setting Priority (1)

- **nice()**
  - int nice(int inc);
  - -20 ≤ inc ≤ 19
  - For the current process (task)

- **setpriority()**
  - int setpriority(int which, id_t who, int value);
  - -20 ≤ value ≤ 19
  - For a process (PRIO_PROCESS), a process group (PRIO_PGRP), or a user (PRIO_USER)
  - Multi-threaded process?

Setting Priority (2)

- **sched_setparam()**
  - int sched_setparam(pid_t pid, struct sched_param *param);
  - Only for real-time processes (SCHED_FIFO & SCHED_RR)
  - sched_get_priority_min() ≤ param->sched_priority ≤ sched_get_priority_max() (1..99 in Linux)

- **pthread_attr_setschedparam()**
  - int pthread_attr_setschedparam(pthread_attr_t *attr, struct sched_param *param);
  - For real-time threads