Operating Systems

Jin-Soo Kim (*jinsookim@skku.edu*)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

Standalone Applications

- Often no OS involved
- One large loop
- Microcontroller-based embedded systems

"Real-Time" OS

- Scheduling of threads (or tasks)
  - Creation and termination of threads
  - Timing of thread activations
- Synchronization
  - Semaphores and locks
- Input and output
  - Interrupt handling with predictable latency
- A small kernel size

Full-Fledged OS

- Applications are huge and complex
- Multiple processes
- Virtual memory
  - Requires MMU-enabled CPU
- File systems with persistent storage
- Networking: TCP/IP stack
- Security: User vs. kernel space
- Windowing & GUI support

Classification

# threads per addr space: / # of addr spaces:	One	Many
One	MS/DOS Early Macintosh	Traditional UNIX
Many	Many embedded OSes (VxWorks, uClinux, uC/OS-II, …)	Mach, OS/2, Linux, Windows, Mac OS X, Solaris, HP-UX

Example: µC/OS-II

What is µC/OS-II?

- A priority-based preemptive RT kernel
- Memory footprint: ~20KB
- Written mainly in C (open but not free)
- Ported to over 100 processors
- Manage up to 64 tasks
- Support 8-bit to 64-bit processors
- GUI, file system, etc. modules available
- http://www.micrium.com

Task

- A task is a simple program that thinks it has the CPU all to itself
- Each task has
  - Its own stack space
  - A priority based on its importance
- A task contains your application code
- A task is an infinite loop

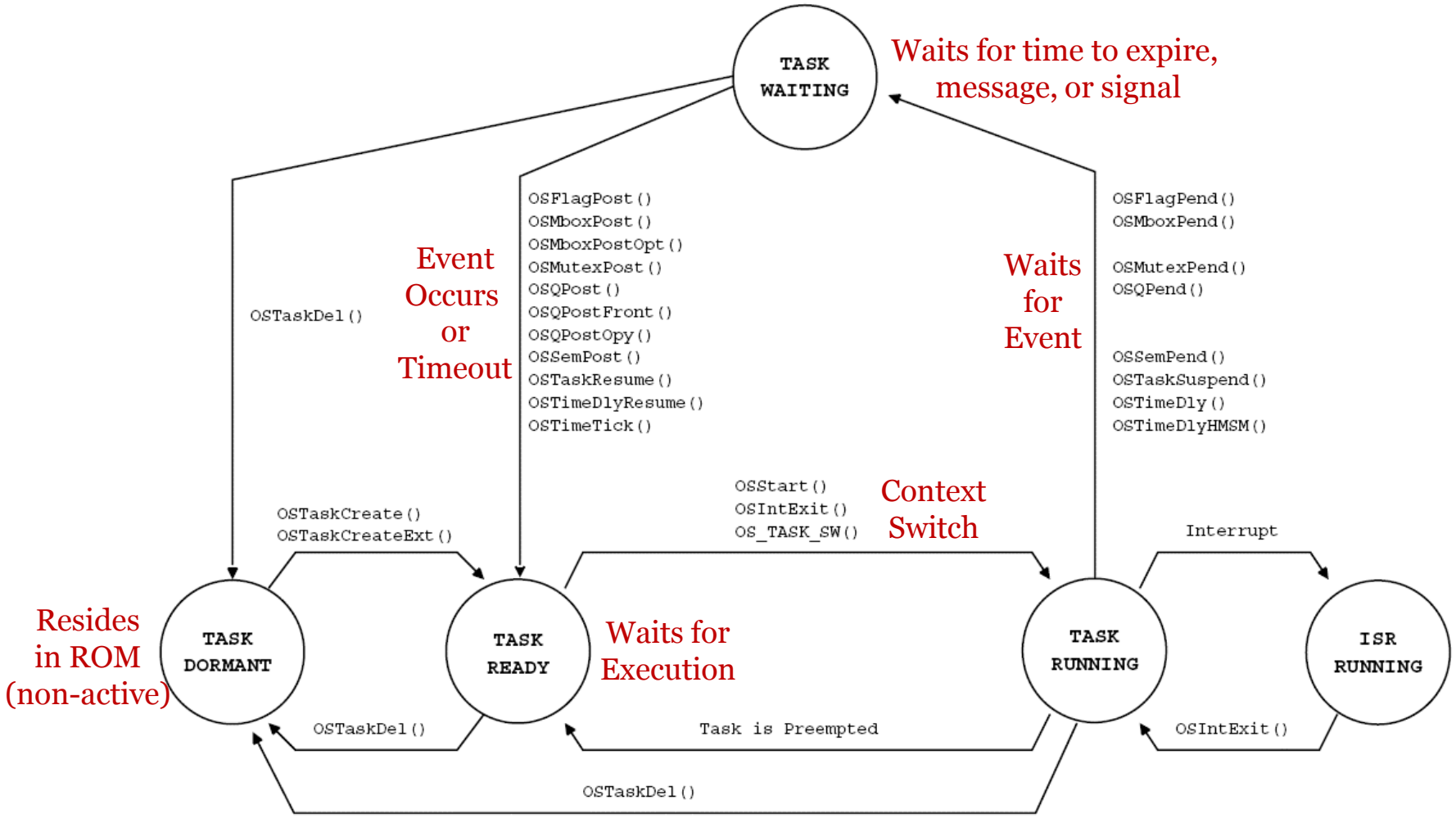
Task Example

```
void Task (void *p_arg)
{
    Do something with argument p_arg;
    Task initialization;
    for (;;) {
        Processing (your code)
        Wait for event;        /* Time to expire ..  */
                               /* Signal from ISR    */
                               /* Signal from task   */
        Processing (your code)
    }
}
```

Task Creation/Deletion

- A task is created by OSTaskCreate()
- Make the new task ready for multitasking
- The followings passed to the kernel
  - Starting address
  - Stack
  - Priority
  - Arguments passed to the task, etc.
- A task is deleted by OSTaskDel()

Task States


TASK WAITING
Waits for time to expire, message, or signal
OSTaskDel()
Event Occurs or Timeout
OSFlagPost()
OSMboxPost()
OSMboxPostOpt()
OSMutexPost()
OSQPost()
OSQPostFront()
OSQPostOpy()
OSSemPost()
OSTaskResume()
OSTimeDlyResume()
OSTimeTick()
Waits for Event
OSFlagPend()
OSMboxPend()
OSMutexPend()
OSQPend()
OSSemPend()
OSTaskSuspend()
OSTimeDly()
OSTimeDlyHMSM()
OSTaskCreate()
OSTaskCreateExt()
OSStart()
OSIntExit()
OS_TASK_SW()
Context Switch
Interrupt
Resides in ROM (non-active)
TASK DORMANT
TASK READY
Waits for Execution
TASK RUNNING
ISR RUNNING
OSTaskDel()
Task is Preempted
OSIntExit()
OSTaskDel()

Task Priority

- Max 64 priority levels
- 8 levels reserved for OS, 56 levels for user
- Low number means high priority
- Priority changed by OSTaskChangePrio()

- Suspending a task: OSTaskSuspend()
- Resuming a task: OSTaskResume()

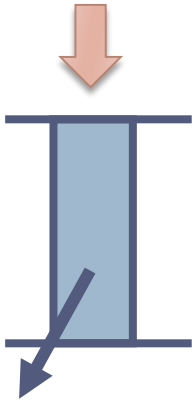
Task Synchronization

- Semaphore Mailbox Message Queue

OSSemPost()

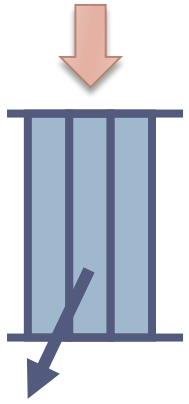
OSSemPend()

OSMboxPost()



OSMboxPend()

OSQPost()
OSQPostFront()



OSQPend()

Semaphores

- OSSemPend()
  - Counter--
  - If (counter < 0), the task is blocked and moved to the wait list
  - A time-out value can be specified
- OSSemPost()
  - Counter++
  - If (counter >= 0), the highest priority task in the wait list is removed from the wait list

Mailbox

- OSMboxPend()
  - The message in the mailbox is retrieved
  - If empty, the task is immediately blocked
  - A time-out value can be specified
- OSMboxPost()
  - A message is posted in the mailbox
  - If there is already a message, returns an error
  - The task with the highest priority is removed from the wait list and scheduled to run

Message Queues

- OSQPend()
  - A message is removed from queue
  - If empty, the task is moved to the wait list and becomes blocked
- OSQPost()/OSQPostFront()
  - A message is appended to the queue
  - OSQPost(): FIFO, OSQPostFront(): LIFO
  - The highest-priority pending task receives the message and scheduled to run, if any

Memory Management

- To reduce memory fragmentation
  - Use of memory blocks with fixed size
  - All blocks same size
  - The size of blocks is configurable
- Creation of a memory partition: OSMemCreate()
- Requesting a memory block: OSMemGet()
- Releasing a memory block; OSMemPut()

Real-Time Scheduling

Real-Time Systems

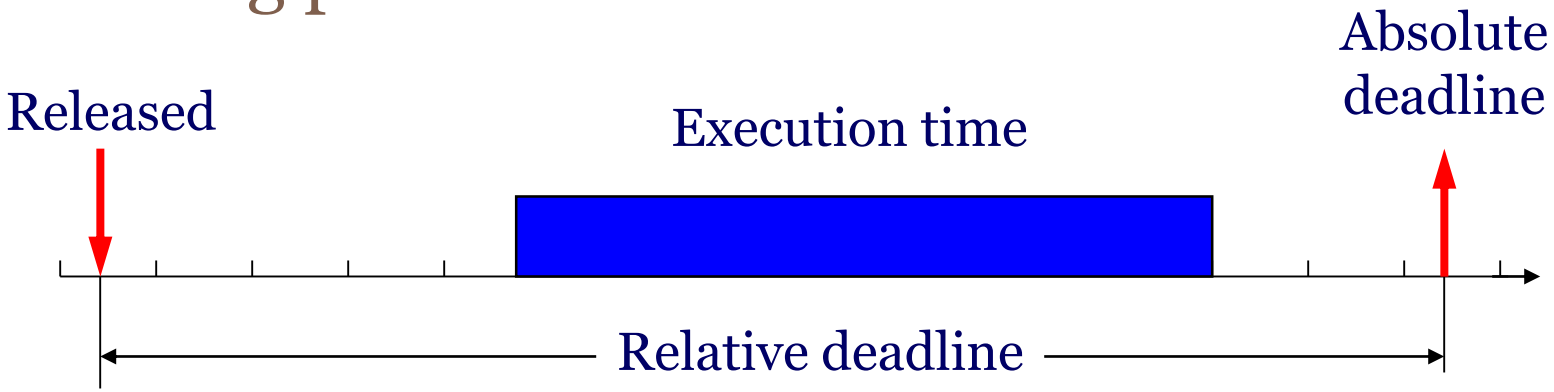
- Perform a computation to conform to external timing constraints
- Deadline frequency: periodic vs. aperiodic
- Deadline type:
  - Hard: failure to meet deadline causes system failure
  - Soft: failure to meet deadline causes degraded response (best effort, statistical guarantees)

Periodic vs. Aperiodic Tasks

- Periodic task: executes on (almost) every period
- Aperiodic task: executes on demand
- Analyzing aperiodic task sets is harder
  - Must consider worst-case combinations of task activations

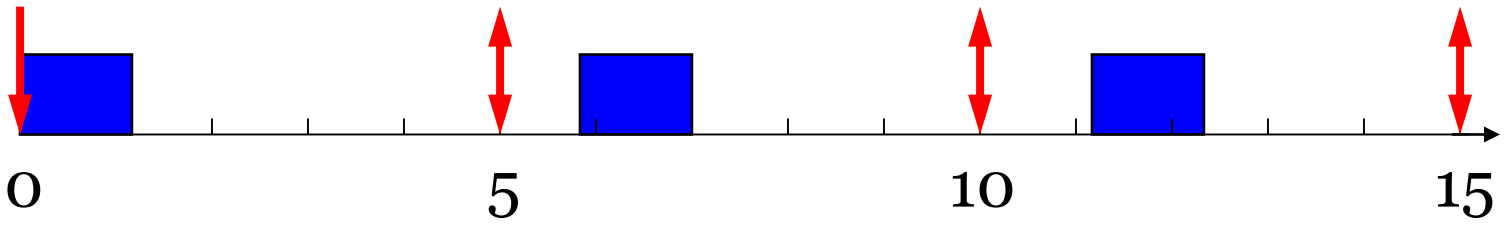
Real-Time Workload

- 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 $U = e/p$

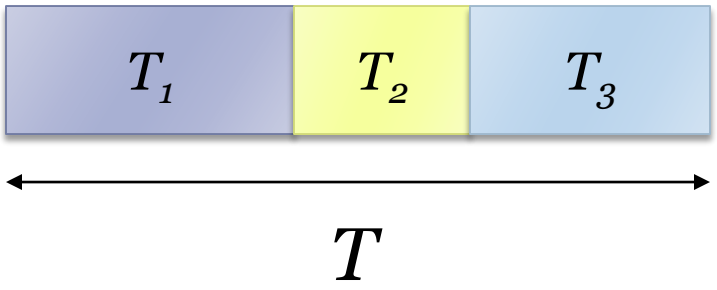


Real-Time Scheduling

- 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

Simple Feasibility Test

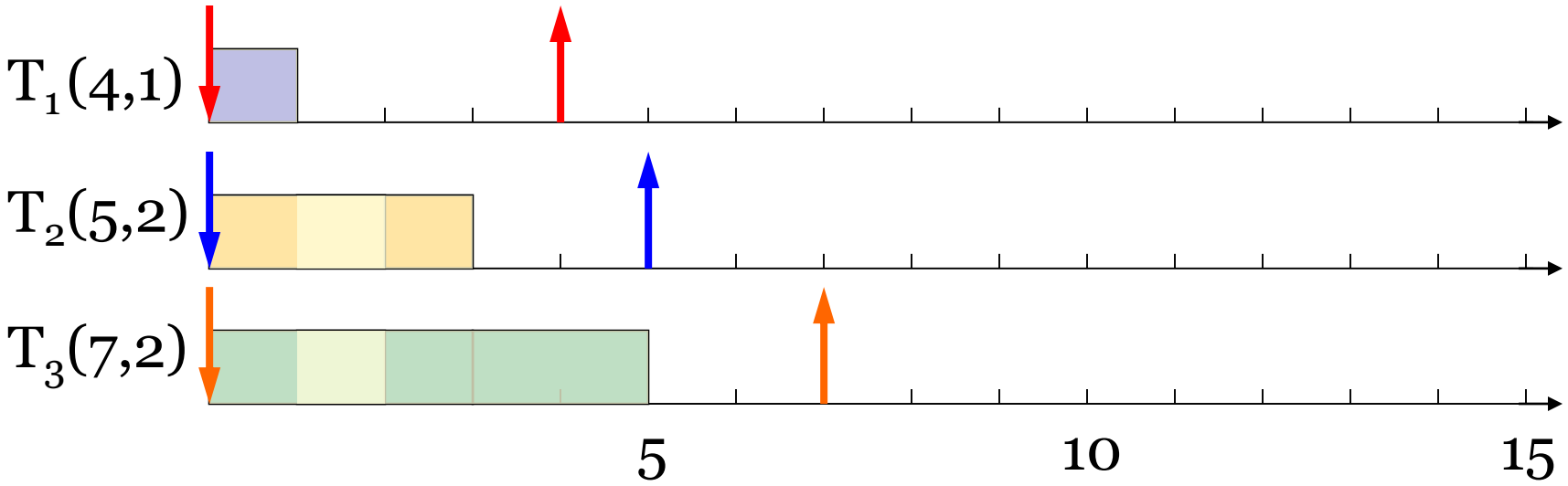
- Assume:
  - No resource conflicts
  - Constant process execution times



- Require:
  - $T \geq \sum_i T_i$
  - Can't use more than 100% of the CPU

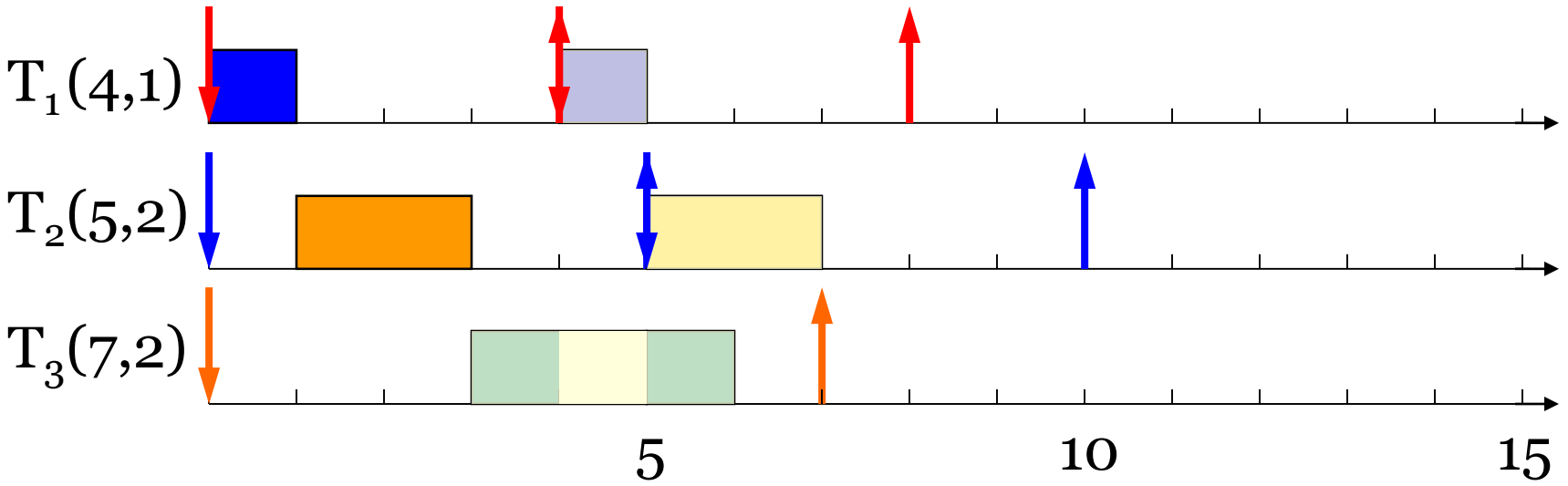
RM

- Rate Monotonic
  - Optimal static-priority scheduling
  - Assigns priority according to period
  - A task with a shorter period has a higher priority



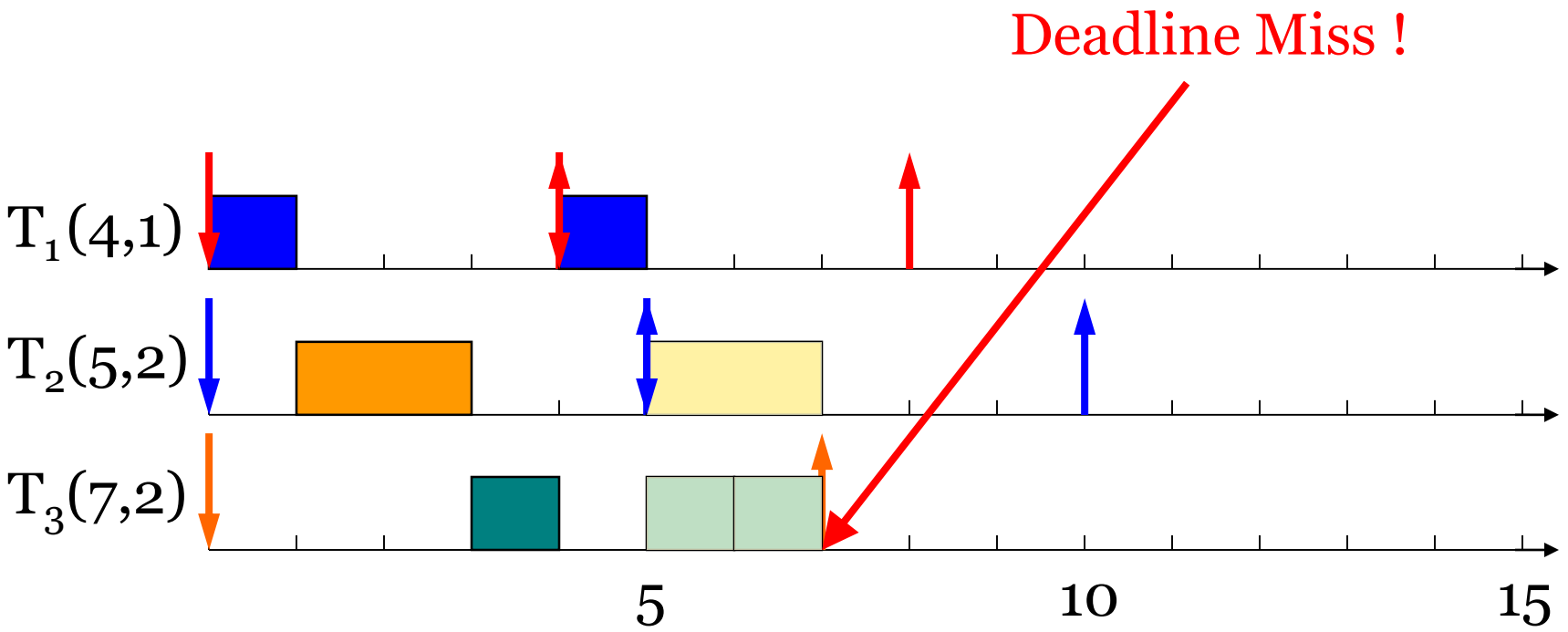
RM

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

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

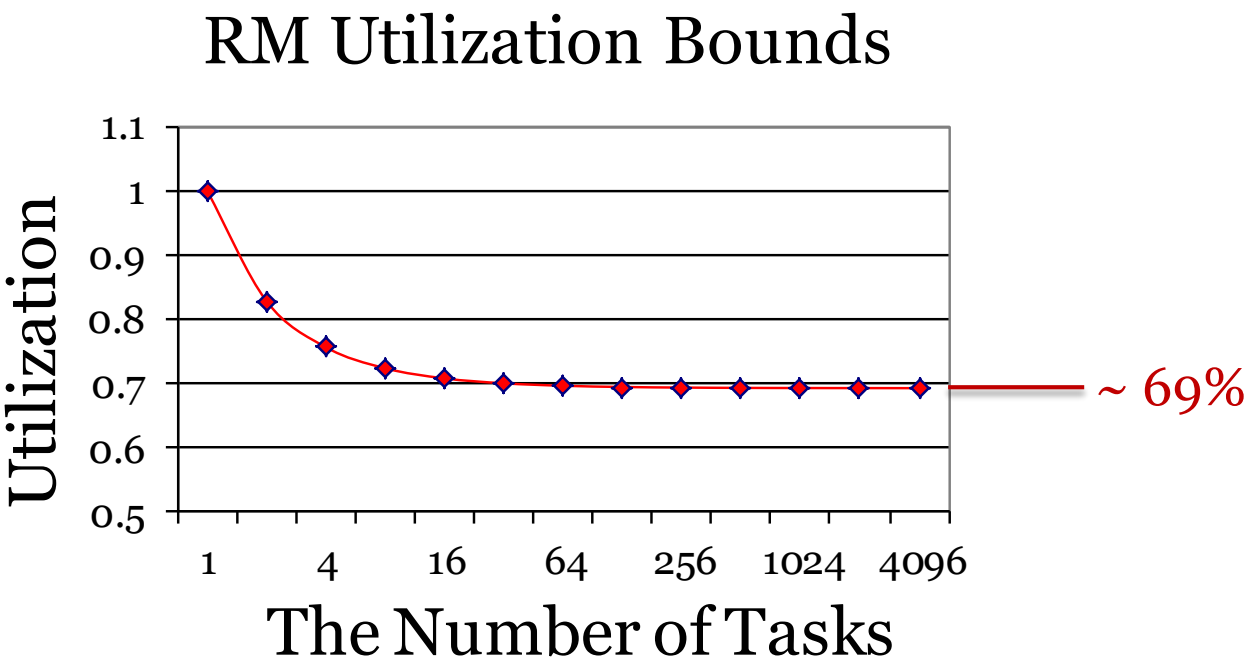
$$3(2^{1/3} - 1) \approx 0.78$$

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

RM

- Utilization bound (cont'd)

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

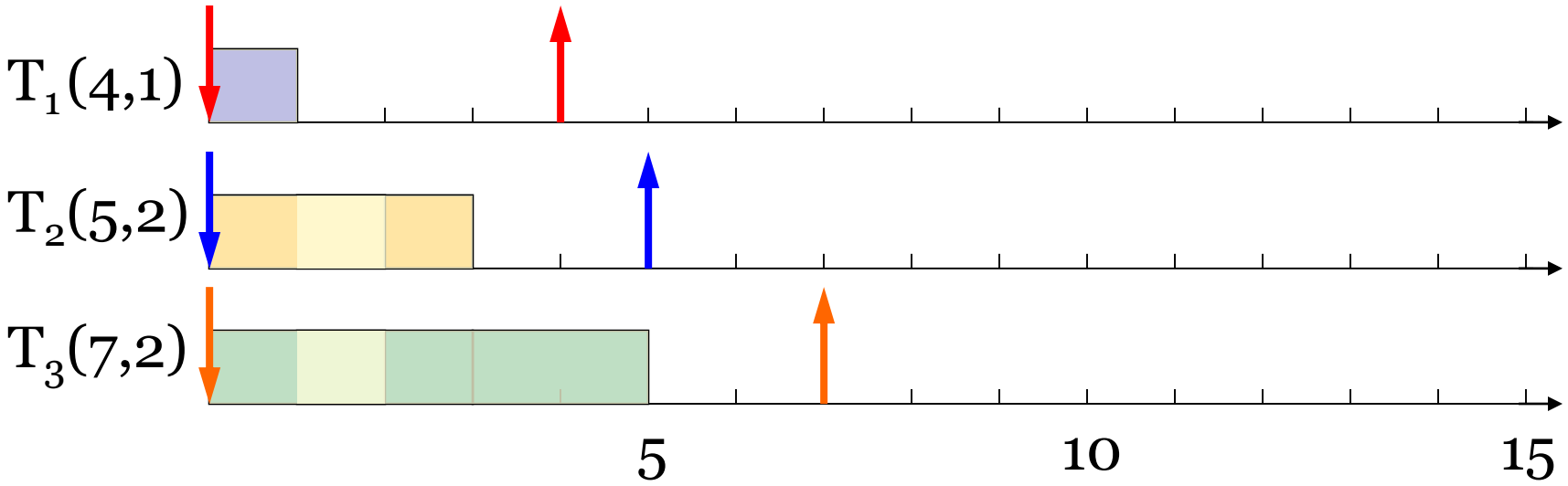


RM

- As the number of tasks approaches infinity, the maximum utilization approaches 69%
- RM cannot use 100% of CPU, even with zero context switch overhead
- Must keep idle cycles available to handle worst-case scenario
- However, RM guarantees all tasks will always meet their deadlines

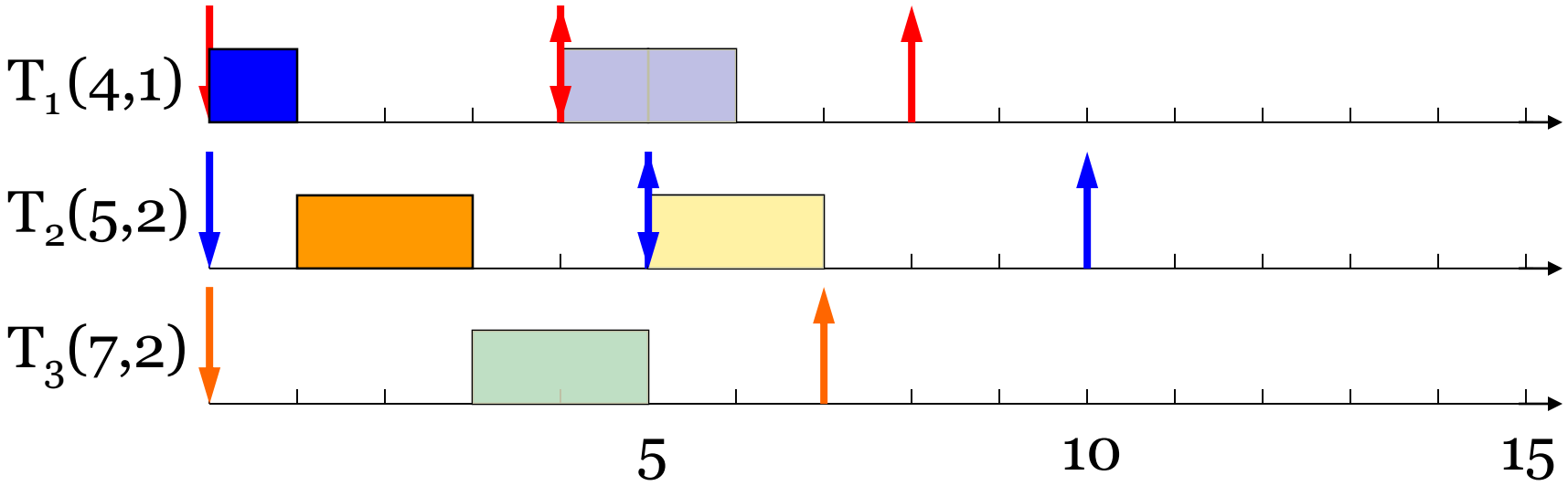
EDF

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



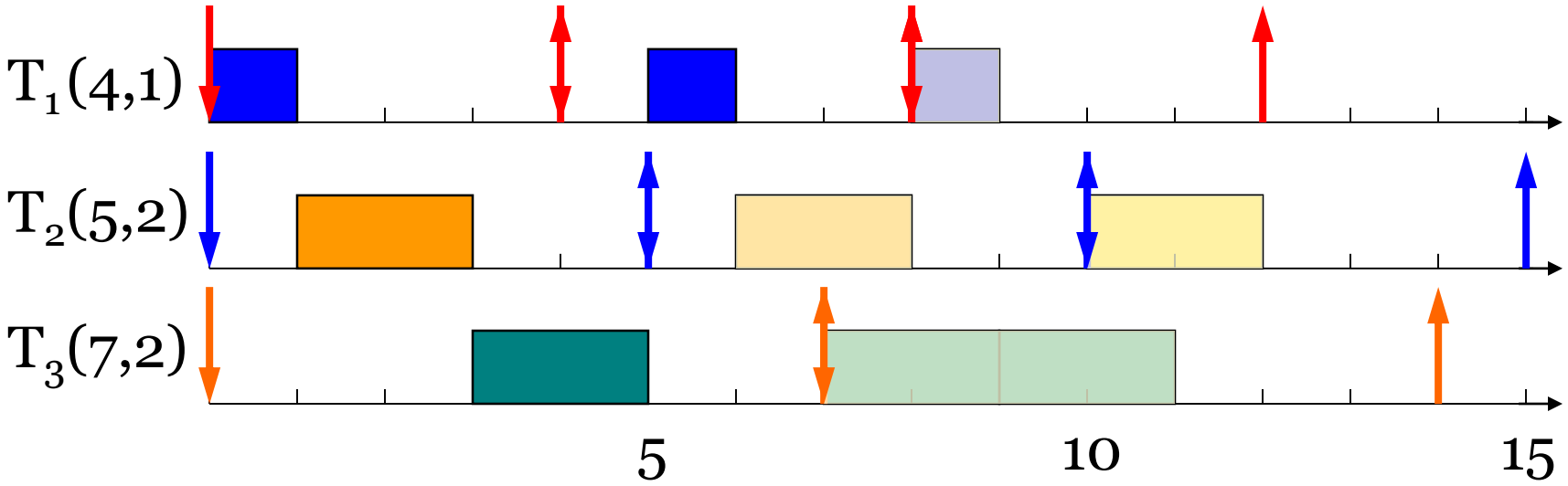
EDF

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



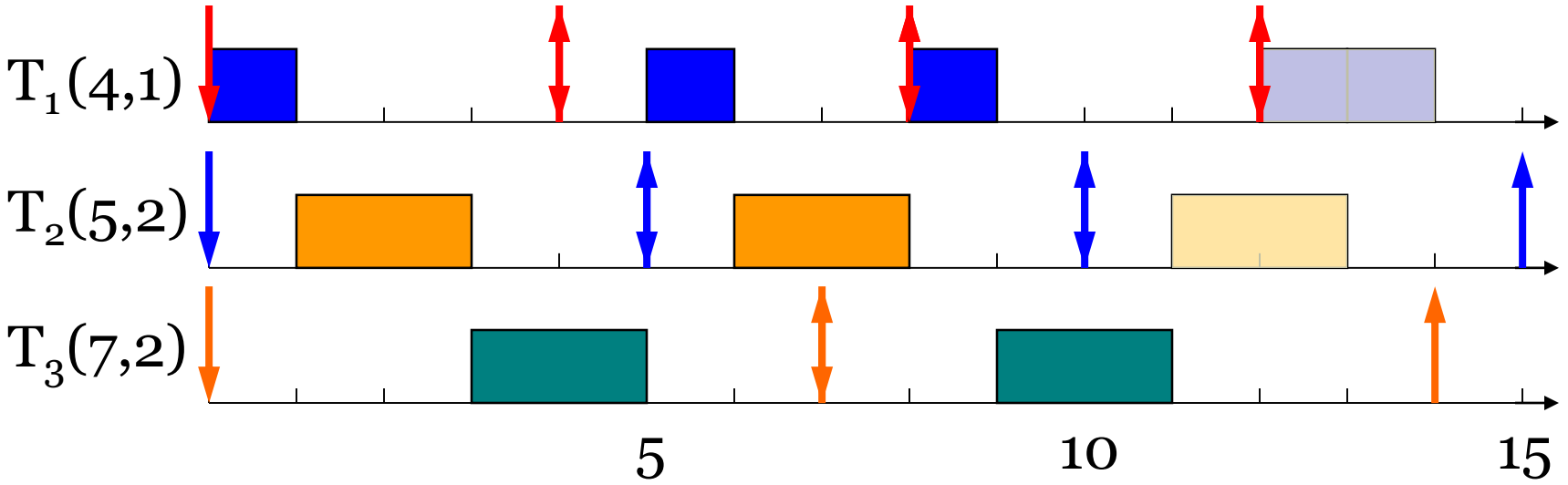
EDF

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



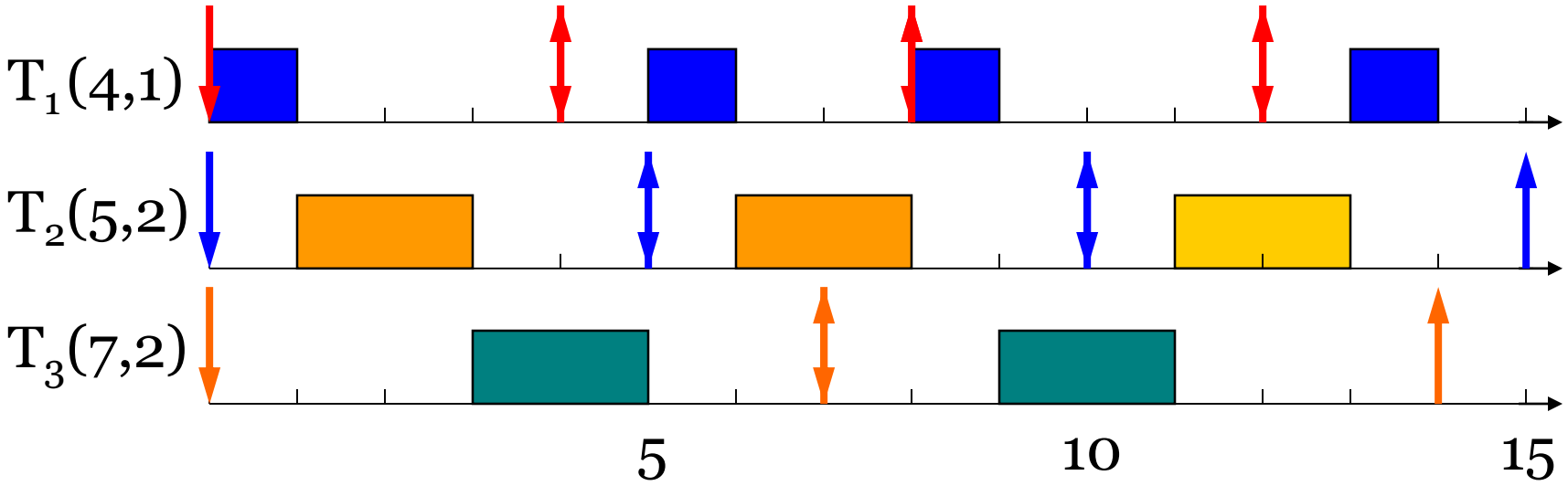
EDF

- 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
  - Predictability for the highest priority tasks
- EDF
  - Full processor utilization
  - Implementation complexity and runtime overhead due to dynamic priority management
  - Misbehavior during overload conditions

RM vs. EDF (2)

- Assumptions
  - All tasks are periodic and fully preemptible
  - 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
  - No task may voluntarily suspend itself
  - All overheads are assumed to be 0
  - There is just one processor