Real Time Operating Systems

Dongbing Gu

School of Computer Science and Electronic Engineering
University of Essex
UK

Spring 2018

Outline

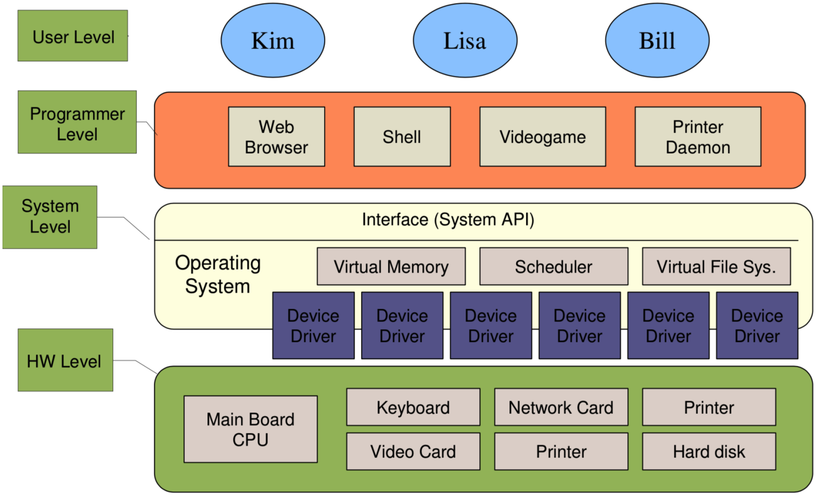
Section 1

Operating System, Process, and Thread

Operating System

- An operating system is a program that
  - Provides an “abstraction” of the physical machine
  - Provides a simple interface to the machine
  - Each part of the interface is a “service”
- An OS is also a resource manager
  - The OS provides access to the physical resources of a computing machine
  - The OS provides abstract resources (for example, a file, a virtual page in memory, etc.)

Levels of Abstraction



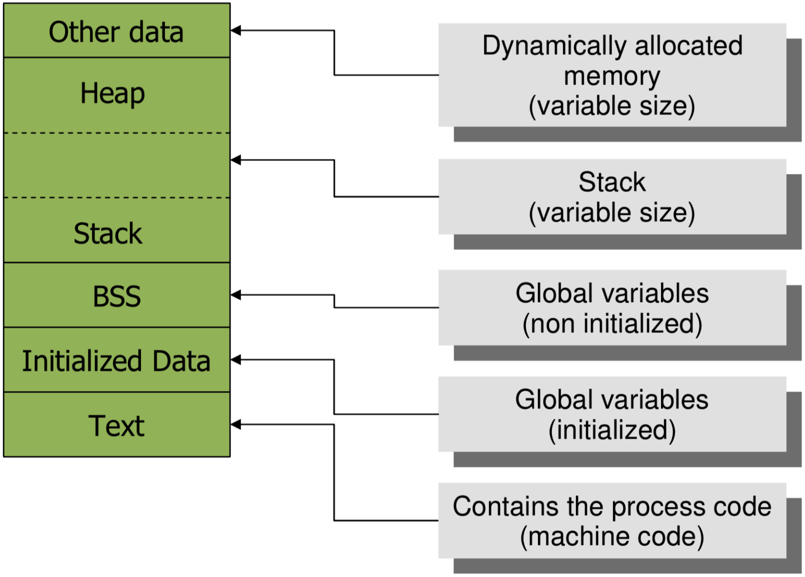
Abstraction Mechanism

- Application programming interface (API) provides a convenient and uniform way to access to services
- HW details are hidden to the high level programmer
- One application does not depend on the HW
- The programmer can concentrate on higher level tasks.

Process

- The fundamental concept in any operating system is the “process”
  - A process is an executing program
  - An OS can execute many processes at the same time (concurrency)
- Processes have separate memory spaces
  - Each process is assigned a private memory space
  - One process is not allowed to read or write in the memory space of another process
  - If a process tries to access a memory location not in its space, an exception is raised, and the process is terminated
  - Two processes cannot directly share variables

Memory of A Process

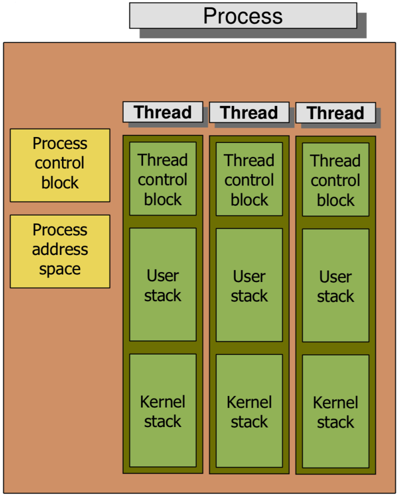

Other data
Heap
Stack
BSS
Initialized Data
Text
Dynamically allocated memory (variable size)
Stack (variable size)
Global variables (non initialized)
Global variables (initialized)
Contains the process code (machine code)

Threads

- One process can consists of one or more threads
- Threads are sometime called lightweight processes
- Therefore, one process can have many different (and concurrent) traces of execution.

Multi-threaded process model

- One address space
- One PCB (Process Control Blocks)
- Many stacks
- Many TCB (Thread Control Blocks)
- The threads are scheduled directly by the scheduler



Multi-threaded process model

- Generally, processes do not share memory
    - To communicate between process, it is necessary to user OS primitives
    - Process switch is more complex because we have to change address space
- Two threads in the same process share the same address space
    - They can access the same variables in memory
    - Communication between threads is simpler
    - Thread switch has less overhead

Section 2

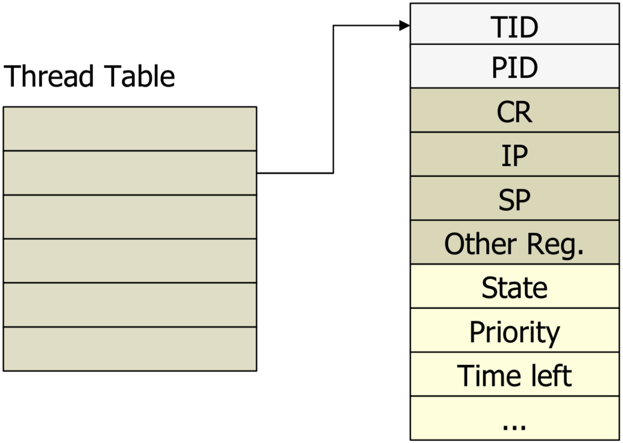
RTOS

RTOS

- Different OS implement threads in different ways: processes only, threads only, or both.
- In Real-Time Operating Systems, depending on the size and type of system we can have both threads and processes or only threads
- For efficiency reasons, most RTOS only support : one process and many threads inside the process
- Examples are RTAI, RTLinux, VxWorks, QNX, etc.

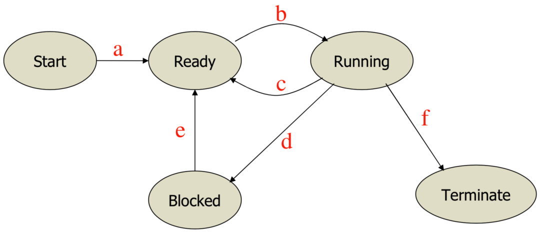
Thread Control Block

- Each thread is assigned a TCB (Thread Control Block)
- The PCB holds mainly information about memory
- The TCB holds information about the state of the thread



Thread States

- Each thread, during its lifetime can be in one of the following states:
  - Starting - the thread is being created
  - Ready - the thread is ready to be executed
  - Executing - the thread is executing
  - Blocked - the thread is waiting on a condition
  - Terminating - the thread is about to terminate



Thread Events

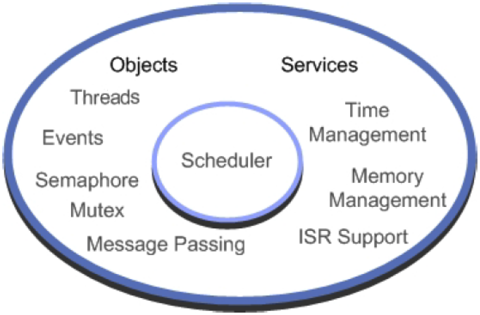
a. Creation - The thread is created
b. Dispatch - The thread is selected to execute
c. Preemption - The thread leaves the processor
d. Wait on condition - The thread is blocked on a condition
e. Condition true - The thread is unblocked
f. Exit - The thread terminates

Context Switch

- Context switch happens when
  - The thread has been preempted by another higher priority thread
  - The thread blocks on some conditions
  - In time-sharing systems, the thread has completed its “round” and it is the turn of some other threads
- We must be able to restore the thread later. Therefore we must save its state before switching to another thread

RTOS Kernel

- An RTOS consists of a scheduler that supports round-robin and pre-emptive multitasking of program threads, as well as time and memory management services.
- Inter-task communication is supported by additional RTOS primitives (objects), including event, semaphore, Mutex, mailbox, etc.



Creating Threads

- The first thread created is used to start additional threads required for the application.
- The first thread can launch the thread and assigns its thread ID number and priority.
- A thread can be created in response to a system event.
- When a thread is created, it is also assigned its own stack for storing data during the context switch.
- This stack is a fixed block of RAM, which holds all the thread variables.

RTOS Interrupt Handling

- While it is possible to run C code in an interrupt service routine (ISR), this is not desirable within an RTOS based application.
- The ISR could delay the timer tick and disrupts the RTOS kernel.
- It is still good practice to keep the time spent in interrupts to a minimum.
- Some microcontrollers support nested interrupts. The nested interrupts has unpredictable stack requirements.
- Interrupt handling can also be accomplished by prioritised threads, which are scheduled by the scheduler.

Thread Scheduling

- Preemptive vs. non-preemptive scheduling
- Periodic vs. aperiodic threads
- Fixed priority vs. dynamic priority
- Priority inversion

Preemptive Scheduling

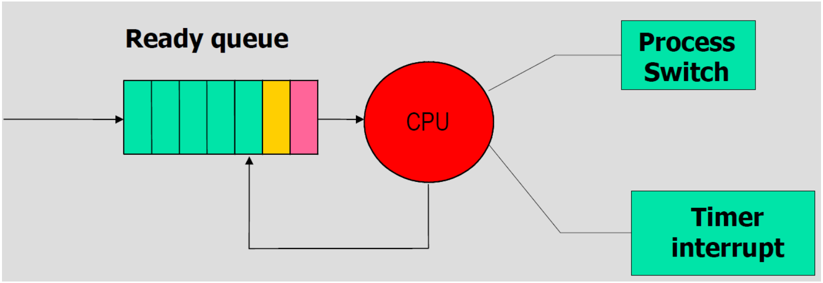
- Assumptions:
    - All threads have priorities, either statically assigned (constant for the duration of the thread) or dynamically assigned (can vary).
    - Kernel keeps track of which threads are enabled (able to execute).
- Preemptive scheduling:
    - At any instant, the enabled thread with the highest priority is executing.
    - Whenever any thread changes priority, the kernel can dispatch a new thread.

Non-Preemptive scheduling: Round Robin

- Time slices are assigned to each thread in equal portions and in circular order,
- Handling all threads without priority.
- Round-robin scheduling is simple, and easy to implement.

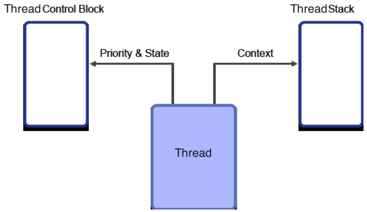
Time Sharing Systems

- Every process can execute for at most one round, for example, 10msec
- At the end of the round, the processor is given to another process



Time Sharing Systems

- In Cortex M3, the RTOS uses the System Tick timer to switch the threads.
- Each switch time , the RTOS saves the state of all the thread variables to a thread stack and stores the runtime information about a thread in a Thread Control Block.
- The "context switch time", the time to save the current thread state and load up and start the next thread, is a crucial value.
- In practice, OS context switch overhead is small (hundreds of clock cycles).

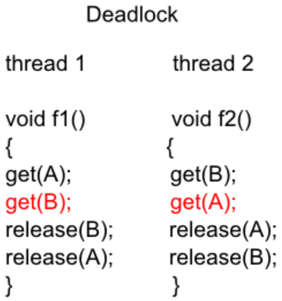


Preemptive Scheduling Problem

- The need to share resources between threads operating in a preemptive multitasking environment can create conflict.
- Two of the most common problems are deadlock and priority inversion.

Deadlock

- Deadlock occurs when some threads are blocked to acquire resources held by other blocked threads.
- To avoid deadlock:
  - Don't request another resource while holding one resource.
  - Don't wait for another thread if there's a chance it's waiting for you.
  - Try to avoid holding locks for longer time.



Deadlock

```
thread 1          thread 2

void f1()         void f2()
{                 {
get(A);           get(B);
get(B);           get(A);
release(B);       release(A);
release(A);       release(B);
}                 }
```

Priority Inversion

- A thread with lower priority gets a lock.
- When a higher-priority thread becomes ready, it preempts the lower priority thread.
- If the higher-priority thread needs that lock, it can't get it due to the lower priority thread has it.
- That means lower-priority thread blocks the higher-priority one. It prevents higher-priority thread from running.
- The solution is "Priority Inheritance": temporarily increase the thread's priority whenever it acquires a lock that is also needed by a higher-priority thread.

Section 3

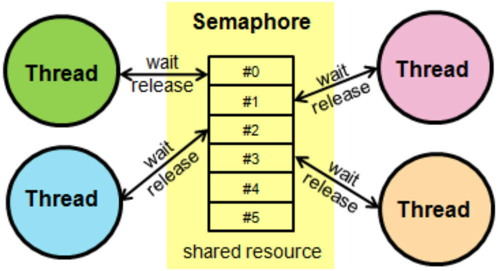
Interprocess Communication

Interprocess communication

- We need to be able to communicate between threads in order to make an application useful.
- A typical RTOS supports several different communication objects, which can be used to link the threads together to form a meaningful program.
- The mbed RTOS supports inter-task communication with signals, semaphores, mutexes, queues, memory pools, and mails.

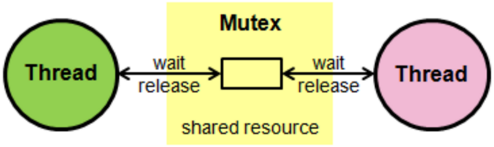
Semaphore

- Particularly useful to manage thread access to a pool of shared resources of a certain type, for example, the access to a group of identical peripherals can be managed.
- Threads can request access to the resource (decrementing the semaphore),
- and can signal that they have finished using the resource (incrementing the semaphore).



Mutex

- Mutex (mutual exclusion)
  - are typically used to serialise access to a section of code that cannot be executed concurrently by more than one thread.
  - A mutex object only allows one thread into a controlled section, forcing other threads which attempt to gain access to that section to wait until the first thread has exited from that section.

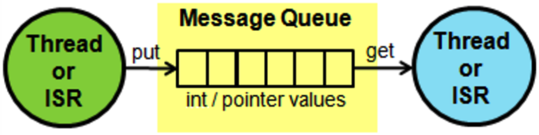


Signal

- The Signal class allows to control or wait signal flags. Each thread has assigned signal flags.
- Public member functions: set(), get(), clear(), wait()

Queue

- A Queue allows you to queue pointers to data from producer threads to consumer threads:

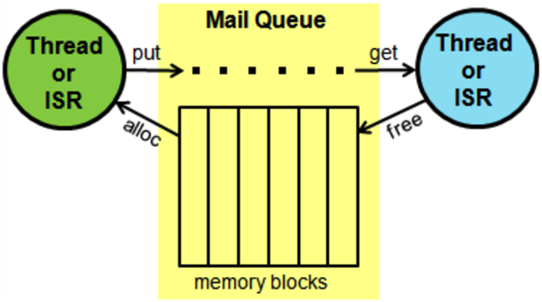


Memory Pool

- The MemoryPool class is used to define and manage fixed-size memory pools.
- Public member functions: MemoryPool(), alloc(), free()

Mail

- A Mail works like a queue with the added benefit of providing a memory pool for allocating messages (not only pointers):

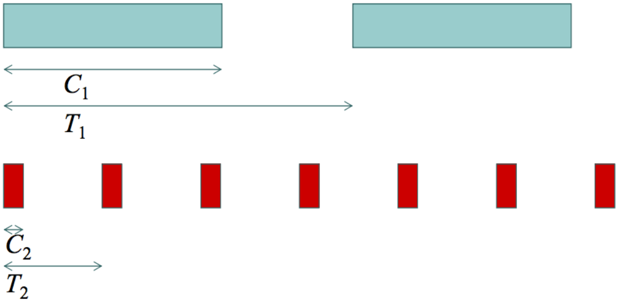


Section 4

RMS and EDF Scheduling

Rate Monotonic Scheduling

- n threads invoked periodically with:
  - periods $T_1, \ldots, T_n$ (impose real-time constraints, or deadlines)
  - worst-case execution times (WCET) $C_1, \ldots, C_n$
  - fixed priorities
  - preemptive scheduling
- Rate Monotonic Scheduling: priorities ordered by period (smallest period has the highest priority)



Feasibility for RMS

- Feasibility is defined for RMS to mean that every thread executes to completion once within its designated period.
- If any priority assignment produces a feasible schedule, then RMS also produces a feasible schedule.
- RMS is optimal in the sense of feasibility.

Scheduling metrics

- How do we evaluate a scheduling policy?
  - Ability to satisfy all deadlines.
  - CPU utilisation - percentage of time devoted to useful work.
  - Scheduling overhead - time required to make scheduling decision.

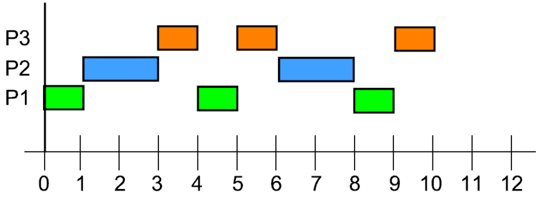
RMS Summary

- RMS: widely-used for real-time systems, analysable scheduling policy.
- All threads run periodically on single CPU.
- Assuming zero context switch time.
- No data dependencies between threads.
- Thread execution time is constant.
- Deadline is at end of period.
- Highest-priority ready thread is always selected for execution.
- The thread with the shortest period is assigned the highest priority.
- This fixed priority scheduling policy is optimal, which ensures that all threads meet their deadlines.

RMS example

- P1 has the highest priority, P2 has the middle priority, and P3 has the lowest priority.
- Construct the shortest repeating cycle equal in length to the **least common multiply** of the thread periods.

Thread	Execution time	Period
P1	1	4
P2	2	6
P3	3	12



RMS example

- Considering the following different set of execution times for these threads.

Thread	Execution time	Period
P1	2	4
P2	3	6
P3	3	12

- Is the RMS scheduling feasible?

Rate-monotonic analysis

- C_i is execution time of thread i; T_i is period of thread i.
- $\frac{C_i}{T_i}$ is the CPU utilisation of thread i.
- The schedulability test for RMS is:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1)$$

- For a set of n periodic threads, a feasible schedule that will always meet deadlines exists if the CPU utilisation is below a specific bound.
- Liu, C.L. and Layland, J.W. “Scheduling Algorithms for Multi-Programming in a Hard Real-Time”. Journal of the Association for Computing Machinery Vol. 20, 1 (January 1973), pp. 46-61.

Rate-monotonic analysis

- For a set of two threads P1 and P2 under RMS scheduling, the CPU utilisation

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} \leq 2(2^{\frac{1}{2}} - 1) = 0.83$$

- When the number of threads tends towards infinity, the CPU utilisation converges to the bound:

$$\lim_{n \to \infty} n(2^{\frac{1}{n}} - 1) = \ln 2 = 0.69$$

Rate-monotonic analysis

- The schedulability test is only sufficient, not necessary!
- Three possible outcomes:
  - $U \leq n(2^{\frac{1}{n}} - 1)$: schedulable
  - $n(2^{\frac{1}{n}} - 1) < U <= 1$: no conclusion
  - $1 < U$: overload, some threads will fail to meet their deadlines no matter what algorithms you use!
- The test may be too conservative.

RMS CPU utilisation

- RMS cannot use 100% of CPU, even with zero context switch overhead.
- Must keep idle cycles available to handle the worst-case scenario.
- However, RMS guarantees all threads will always meet their deadlines.
- No fixed-priority scheme does better.

RMS implementation

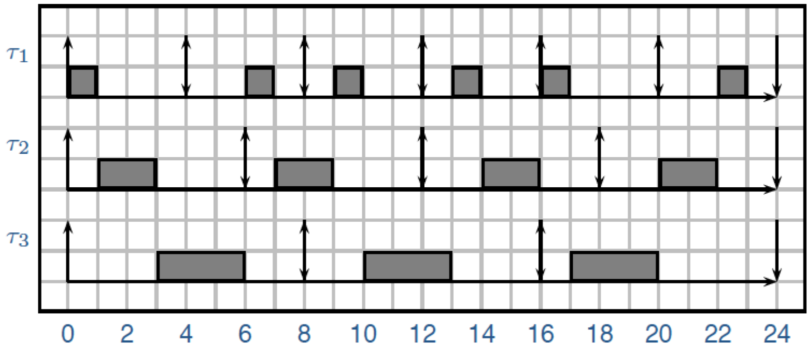
- The implementation of RMS (RMS scheduler) is very simple.
- It runs at a timer interrupt.
- The code scans through the list of threads in the priority order and select the highest priority ready thread to run.
- Because the priorities are static, the threads can be sorted by priority in advance.

Earliest-deadline-first scheduling

- EDF: dynamic priority scheduling scheme.
- Thread closest to its deadline has highest priority.
- Requires recalculating priorities at every timer interrupt.
- EDF can use 100% of CPU.
- Generally considered too expensive to use in RTOS.

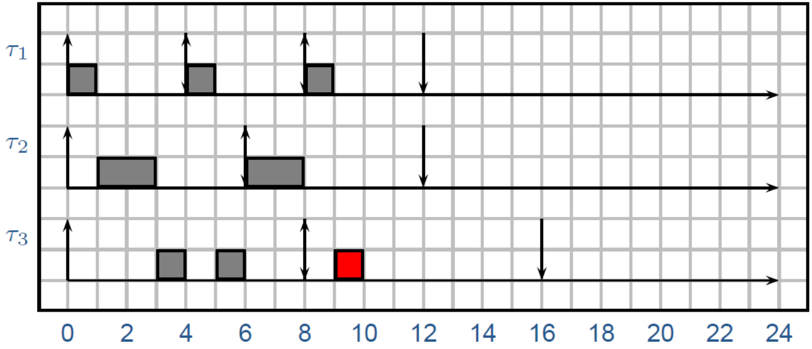
Earliest-deadline-first scheduling

- $\tau_1 = (1,4), \tau_2 = (2,6), \tau_3 = (3,8)$
- Utilisation $U = \frac{1}{4} + \frac{2}{6} + \frac{3}{8} = \frac{23}{24}$



RMS scheduling

- $\tau_1 = (1, 4), \tau_2 = (2, 6), \tau_3 = (3, 8)$

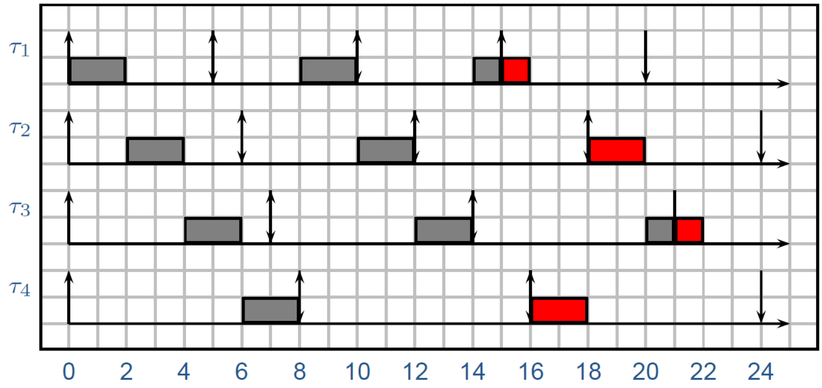


Earliest-deadline-first scheduling

- Fully utilise the processor, less idle times;
- If $U \leq 1$, then it is schedulable by EDF.
- In particular, EDF can schedule all threads that can be scheduled by RMS, but not vice versa
- In general, EDF has less context switches
- EDF is not provided by any commercial RTOS, because of some disadvantage:
    - Less predictable (The response time in EDF is variable). Look at the response time of thread 1 in the example.
    - Less controllable (the priority is not controlled easily). In RMS, the response time of a thread can be fixed by assigning a high priority.
    - More overhead for scheduling. RMS only needs an timer interrupt, but EDF requires more, such as keep the track of deadlines.

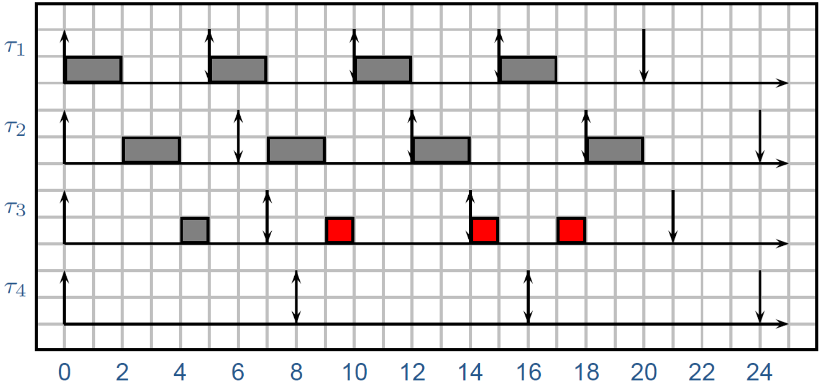
EDF Overhead Case $U > 1$

- All threads missed their deadline almost at the same time



RMS Overhead Case $U > 1$

- Only lower priority threads miss their deadlines!
- However, it may happen that some threads never executes in case of high overload, while EDF is more fair (all threads are treated in the same way)



Fixing scheduling problems

- What if your set of threads is unschedulable?
  - Change deadlines in requirements.
  - Reduce execution times of threads.
  - Get a faster CPU.