Processes

Roles of an operating system

An operating system, or run-time library, should offer

- Timing features
- A memory abstraction
- Multitasking (processes)
- Storage primitives (filesystems)
- Device drivers (peripherals)

Please evaluate whether and how to implement malloc()/free()

- Our code base, so far, misses a memory API

And we now talk about processes

Definitions

Process: a sequence of instructions

- It performs some useful activity
- It may need to communicate with other processes
- It may need to communicate with I/O devices
- It may have some time constraints

Task: almost a process

Job: each activation of a task

- In the real-time and embedded world we prefer "task"
- A task is a sequence of operations with time constraints
- Most tasks are periodic (once job every 50ms, for example)
- Non-periodic tasks may be activated by events

Scheduler: the part of the OS that allocates the CPU to tasks

- A real-time scheduler is quite different from a general-purpose on
- The scheduler of a PC, or a server, must serve all processes fairly
  - An editor must respond quickly, which a compiler may lag
- In control systems you want stuff to happen for sure
  - And within a maximum allowed delay

More definitions

Release time: when a job becomes "active" (allowed to run)
Deadline: when the job must complete
WCET: worst case execution time, for a task

Scheduling algorithm: the policy set forth by the scheduler
Schedulability: property (or lack thereof) of a task set

- **A set is schedulable (on a specific uC/...) if all constraints can be met**

Load (U): amount of CPU time used by the tasks

Lateness: how much late we are (can be negative)

Jitter: well... jitter

Assumptions for scheduling algorithms

Classic literature solves the problem for this situation:

- Tasks are periodic
- Deadline == next release time
- The scheduler is preemptive
- Scheduling decisions have not cost

Then, for simplicity of representation:

- All times are multiples of a timer tick

Everything is simple and linear

- There are mathematical demonstrations for all schedulers

Simplifications can be lifted later

- And math becomes more complex

There are a number of interesting schedulers

- All of them are mathematically demonstrated

The two most impotrtant schedulers

RM (Rate Monotonic)

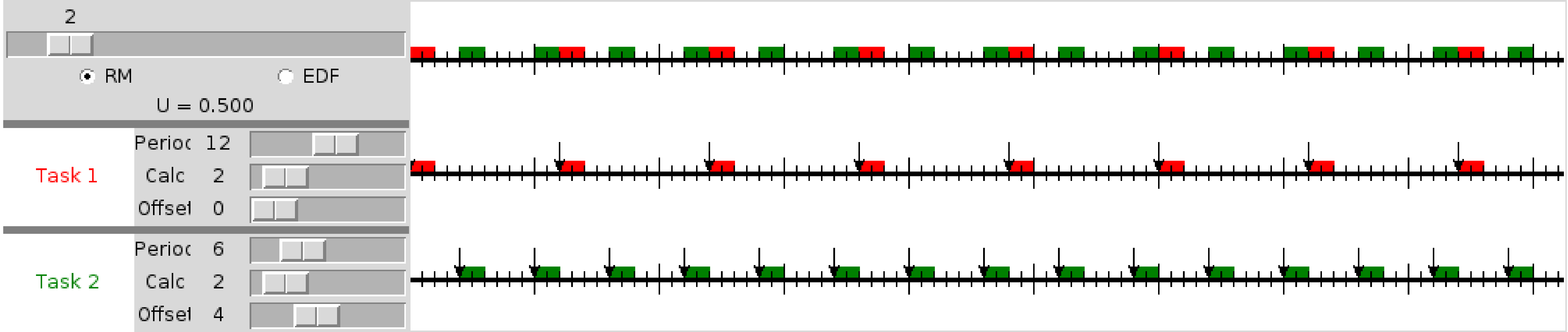
- The higher the rate, the higher the priority.
- When a task is released, it preempts any higher-period task

EDF (Earliest Deadline First)

- "Dynamic priority" (not fixed for each process)
- When a task is released, it preempts if its deadline is nearer

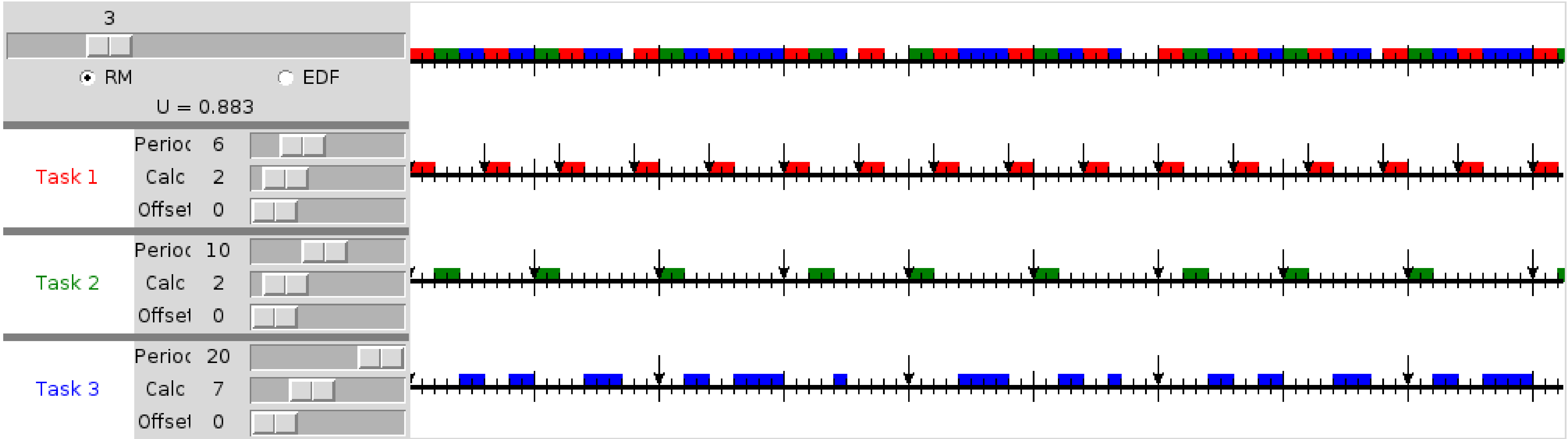
RM is simpler, EDF is better

- EDF guarantees schedulability up to U = 1

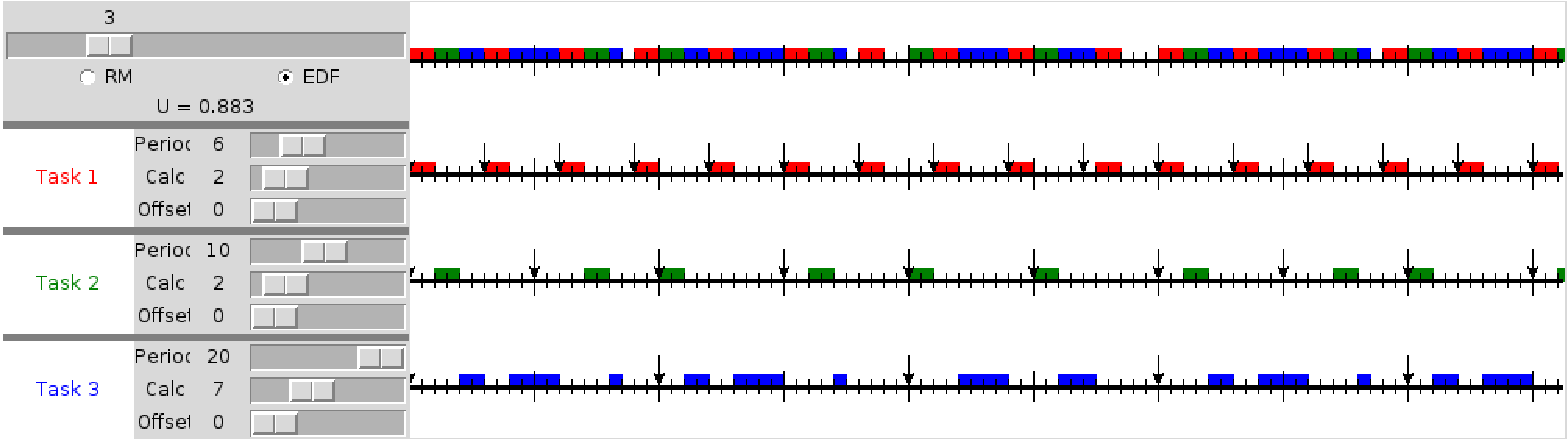


Examples

RM with a set of 3 tasks

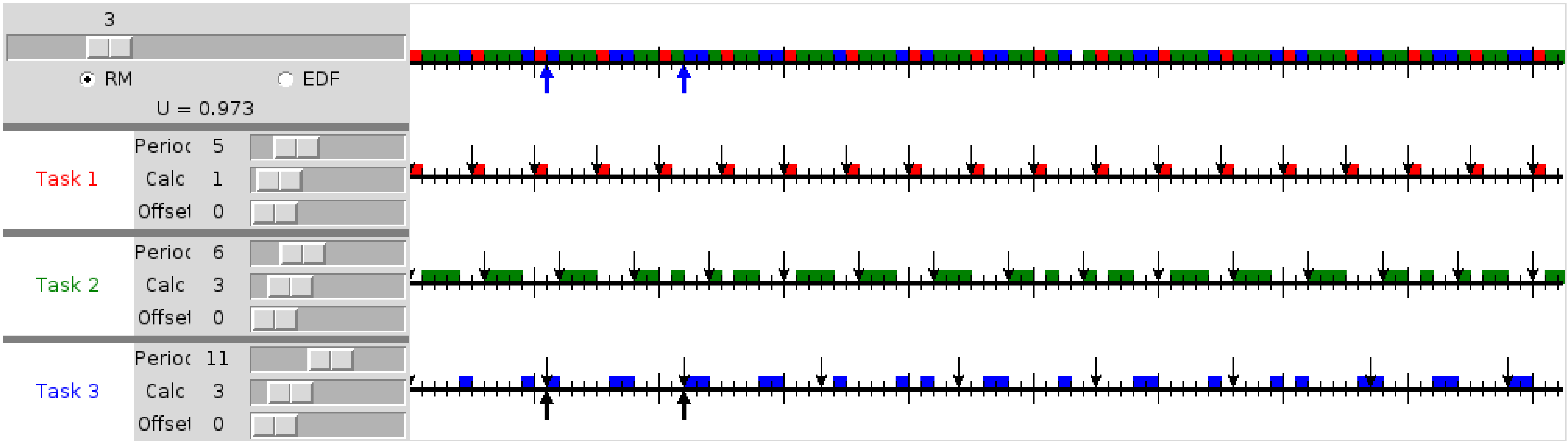


EDF with the same task set

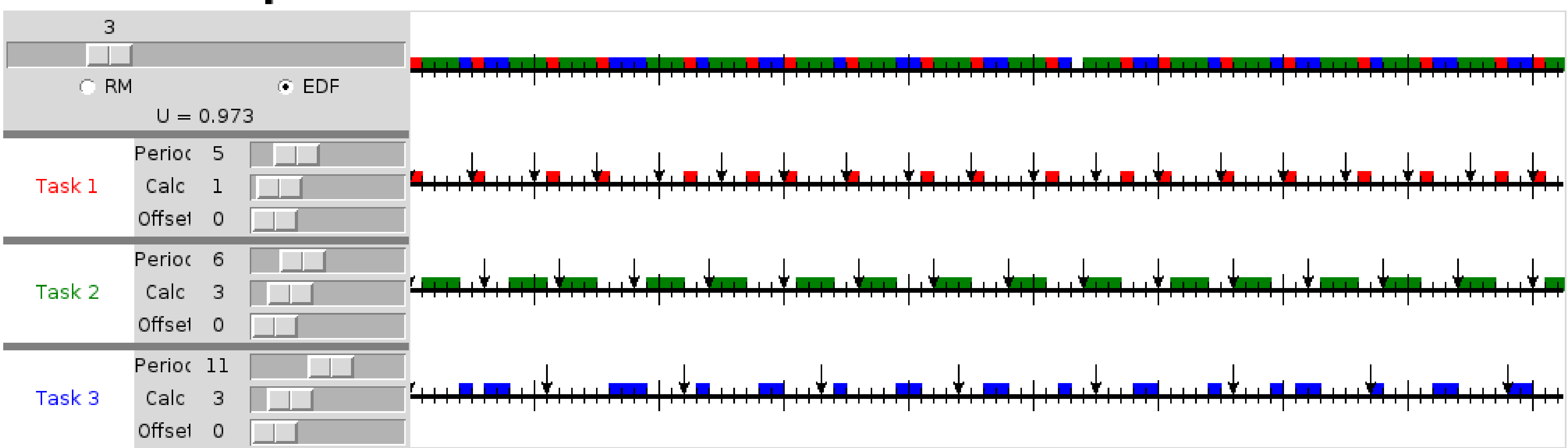


More examples

RM example: task 3 fails



EDF example: the same set is schedulable



Offline scheduling

Complex algorithms fit complex problems

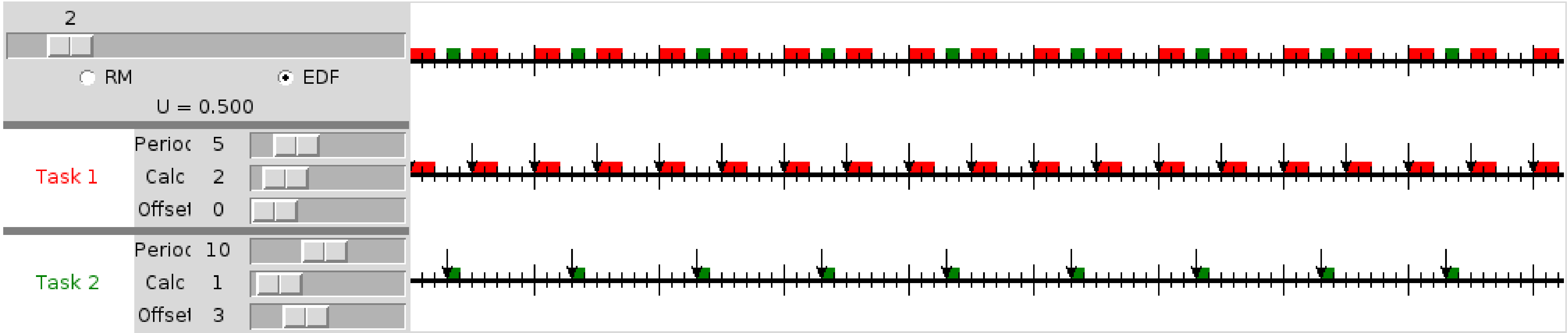
- Many tasks, long WCET, serious processing power, ...

But most real-world situations are much simpler

- Critical tasks feature a small WCET but are jitter-sensitive
- The number of critical tasks is limited
- Their period is usually "compatible" one another
  - e.g.: 5ms, 10ms, 50ms
- Jitter should be limited, ideally zero

This is best solved by manual placement of release times

- Which still requires a sort of scheduler, to manage context switches



Setting up a task

The "classic" approach to RT tasks is as follows:

```
void random_task(some_arg)
{
	init_this();
	init_that;

	while (1) {
		work_on_this();
		work_on_that;

		os_wait_next_period();
	}
}
```

Beautiful, isn't it?

No, not beautiful at all

This code prevents generic inspection of process status

- Process status is in local variables

It requires full context-switch support

- Multiple stacks
- Saving and restoring all registers
- ... even if we implement no preemption

Scheduler structures are opaque

Init time is unclean

- Some task may be initializing
  while other tasks are already running

```
void random_task(some_arg)
{
        init_this();
        init_that;

        while (1) {
            work_on_this();
            work_on_that;

            os_wait_next_period();
        }
}
```

Turning it inside out

By noting that init and job are separate, we can do:

```
struct task {
    char *name; void *arg;
    int (*init)(void *arg, struct task *t);
    int (*job)(void *arg, struct task *t);
    unsigned long nextrun, period;
};

struct task task_temperature = { ... };

DECLARE_TASK(task_temperature); /* creates entry in ELF section */
```

The code above can be made better (please suggest), but:

- It uses a single stack
- You can look at (or change) task status
- You can add extra info in the same structure.

Even if some experts dislike this, some love it like I do

And we can add preemption later (with a single stack)

The scheduler

With the task structure just described

- Where init and job and separate
- Where we rely on an ELF section

... the scheduler is trivial

```
while (1) {
    for (best = t = task_first; t < task_last; t++)
        if  (time_before(t->nextrun, best->nextrun)
            best = t;
    while (time_before(jiffies, best->nexrun)
        ;
    best->job(best->arg, best); /* maybe use retval */
    best->nextrun += best->period; /* BUG! */
}
```

And we can expand on this over time

- But please remember to keep it simple, or you loose

Long and background tasks

What we miss in the previous approach is

- Support for tasks that must use all "free" CPU time
- Support for the random "long" job duration

Typical "long" tasks:

- A command shell on the serial port
- Data communication, over serial or USB (or whatever)
- To support that, we really need preemption

We can special-case some of them

- We accept that the console is just a debugging tool
  - It can temporary halt scheduling
  - And we know for sure it won't run in production

But data communication takes time

- We must prepare our frames, possibly with printf too
- This happens rarely, but then it's a few hundred microseconds
- Or even several milliseconds on the UART port

Aperiodic servers

There are several textbook options for aperiodic tasks

The most easy to model: polling server

- It behaves like a periodic task, which serves aperiodic activities
- Definitely, it requires preemption and a real scheduler

The most easy to implement: background server

- A process that uses all otherwise-unused CPU time
- Again, it must be preempted by "real" jobs.
- We might accept to special-case it (see above)

Playing with aperiodic requests

In the www repository, directory tools/, you find "scheser"

It shows graphically three servers for async requests

- **Background server**
- **Polling server**
- **Deferrable server**

Below is an example with the "background server" (and RM)