Thermal-aware Resource Management for Modern Computing Platforms in the Next Generation of Aircraft

Deliverable type	Report
Deliverable Name	DEmOS Scheduler Design
Deliverable Number	X0
Work Package	WPxxx TODO
Responsible Partner	CVUT
Report Status	Draft
Dissemination Level	RE – Restricted
Version	0.1
Due Date of Deliverable	TODO

	Signature	Date
WP Leader		
Project Coordinator		
Topic Leader		

This project has received funding from the Clean Sky 2 Joint Undertaking under the European Union’s Horizon 2020 research and innovation programme under grant agreement No 832011.

Purpose & Scope

Revision History

Version	Date	Author/Reviewer	Comments
0.1 Draft	2019-11-24	CTU	Current revision

Contents

1 Introduction

DEmOS scheduler will be a Linux program, that will mimic Deos™ RTOS scheduler used in safety-critical avionics applications. In contrast to the real RTOS scheduler, the focus of DEmOS will not be on providing the hard real-time and safety guarantees, but on providing similar scheduling model on top of Linux for the purpose of developing benchmarks of avionics workloads. In the context of the THERMAC project, such benchmarks will be used for evaluation of proposed thermal management techniques.

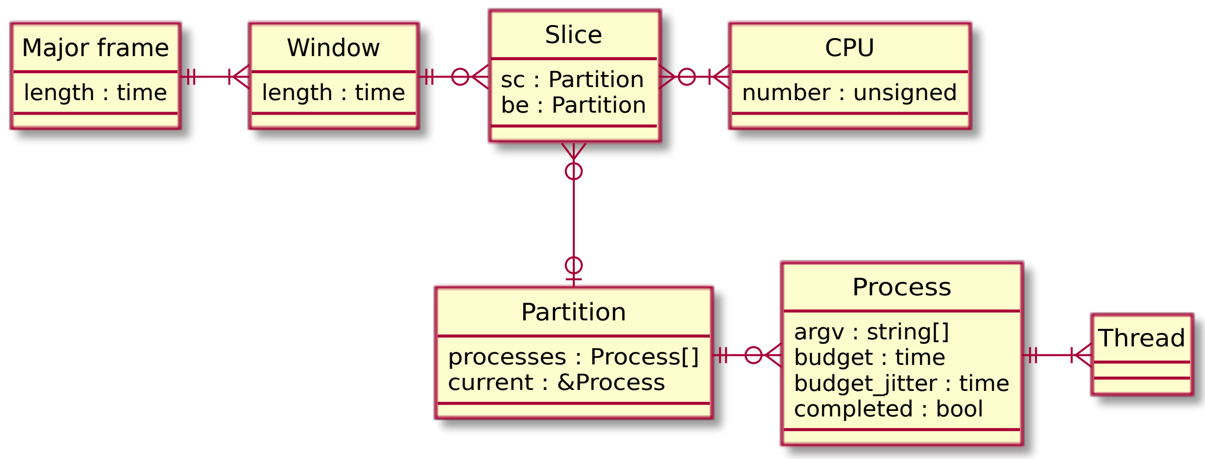

Figure 1: Entity Relationship diagram of DEmOS scheduler main entities

2 Scheduling model and terminology

DEmOS is a clock-driven multi-core scheduler, supporting safety critical and best effort tasks. The entities of the scheduler and their relations are depicted in Figure 1, example configuration is shown in Figure 2.

The configured schedule repeats periodically. A single period is called a *major frame*. Major frame is split into several *windows* of arbitrary length and each window can host several single-core and/or multi-core *slices*. Within a single window, any two slices cannot share the same CPU.

Slices are used to execute *partitions*. A partition is a set of Linux *processes* that are scheduled "together" – both in space (CPUs) and time (window). A partition can be either *safety critical* (SC) or *best effort* (BE), which determines when and how its processes are scheduled inside slices (see below). Each slice has assigned one SC partition and one BE partition. Each partition can assigned to at most one slice in a single window.

Partitions can be seen as a way to isolate one group of processes from another group. For example, one partition can contain all processes from one supplier, and another partition processes from another supplier. When those partitions are executed in different windows, no two processes from different suppliers will ever be executed together.

Each *process* has assigned an execution time *budget* b and optional *budget jitter* b_j (we use the jitter to model variation in execution times and non-determinism on the execution platform timing behavior). Internally, the process can be single-threaded or multi-threaded. DEmOS scheduler operates only at process level, individual threads are scheduled according to the Linux scheduling policy set by the process itself.

2.1 Process execution

Execution of processes inside slices will happen as follows. Each slice has assigned at most one safety-critical (SC) partition and at most one best effort (BE) partition[1].

2.1.1 Safety-critical partition

First all processes from the safety-critical partition will be executed. They will be released sequentially, one after another, starting from the first process. We denote the time the process is released as t_r. Each process will be executed either until it signalizes its completion to the scheduler (which is different from process termination) or until its *effective budget* b_e is exhausted, which happens at time $t_r + b_e$. Effective budget is calculated in each slice before releasing the process as $b_e = b - b_j/2 + b_j \cdot rnd$, where *rnd* is a random number in range $\langle 0, 1 \rangle$. In case of budget exhaustion, the process will be suspended and will be resumed in the next slice assigned to the same partition. It is a configuration error if execution budget of any SC process allows the process to execute until the end of the window. All SC processes should finish during the first $\approx$ 60% of the window.

2.1.2 Best-effort partition

After safety critical process in the slice are completed or suspended, processes from the best effort partition can be executed. Their execution can be skipped or slowed down arbitrarily due to power/thermal/energy saving mechanisms.

We want to support two types of best effort processes:

1. *One-shot* processes will be released at most once in each slice. They will specify this by signalling completion to the scheduler.
2. *Continuous* processes will consume all remaining CPU time of the partition slices. They will be given a share of that time proportional to their budget.

At the beginning of each slice, *completed* marks of all processes are removed. Then, the BE processes are executed sequentially as safety critical processes, with the difference that BE processes are not released starting from the first process in the partition, but from the so called $current_{be}$ process. Also, processes marked as *completed* are skipped. Initially, $current_{be}$ is set to the first process in the partition. Each BE process executes until satisfying any of the conditions below:

1. The processes signalizes its completion. In this case it is marked as *completed* and $current_{be}$ is changed to the next process.
2. The budget b_e is exhausted. In this case $current_{be}$ is changed to the next process.
3. End of the window (of length w) is reached. The process is suspended and its execution will continue in the next slice assigned to the partition and its budget b_e is decreased by $w - t_r$.

[1]If no BE partition is specified in the configuration, default BE partition will be used. It is, however, not clear how to handle the situation, when the default BE partition should be scheduled in multiple slices of the same window. The "problem" is that each slice can start its BE partition at different time instant.

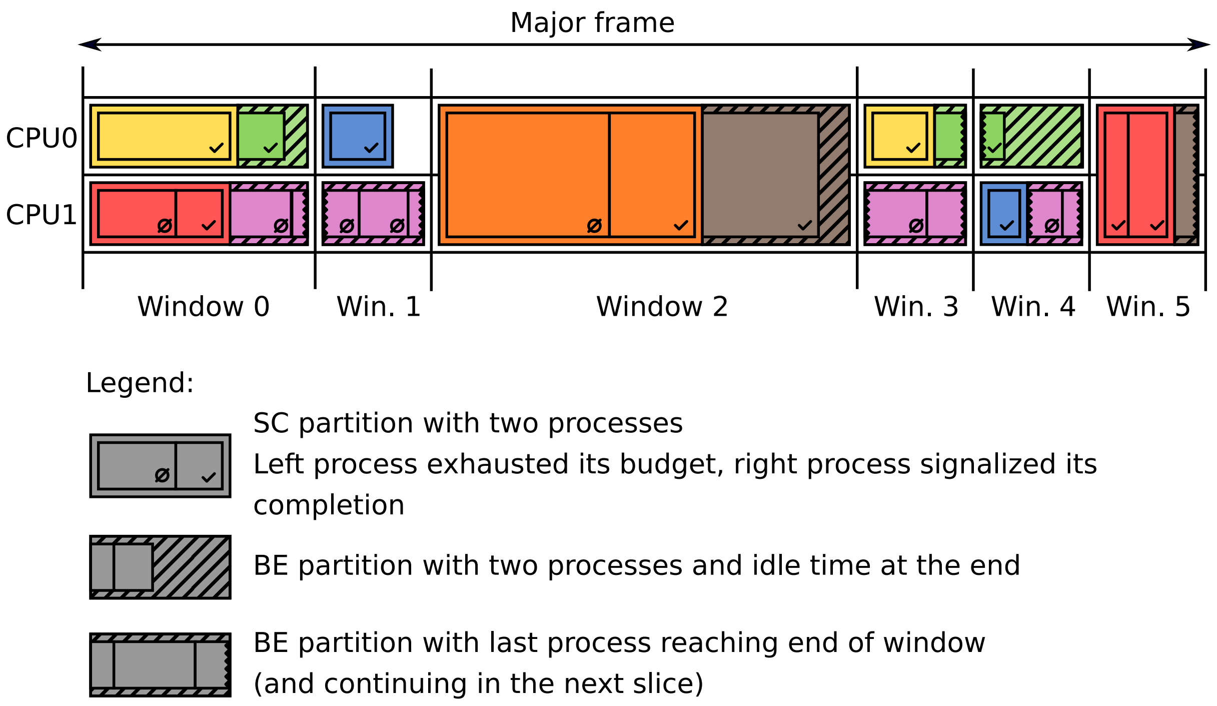
Major frame
CPU0
CPU1
Window 0
Win. 1
Window 2
Win. 3
Win. 4
Win. 5
Legend:
SC partition with two processes
Left process exhausted its budget, right process signalized its completion
BE partition with two processes and idle time at the end
BE partition with last process reaching end of window (and continuing in the next slice)

Figure 2: Example of scheduler and windows configuration (Gantt chart). Colors represent partitions (solid ones are *safety critical*, hatched are *best effort*). Rectangles outside of partitions are slices, rectangles inside partitions are running processes.

When $current_{be}$ reaches the end of the processes in the partition, it is reset to the first process and execution continues. When all BE processes are marked as *completed* the slice remains idle.

3 Scheduler configuration

Example configuration file in YAML format:

```yaml
# Major frame
period: 100 # all times in ms
windows:
  - length: 10
    slices:
      - cpu: 0
        sc_partition: Airbus
      - cpu: 1
        sc_partition: Boeing
  - length: 20
    slices:
      - cpu: [0, 1]
        sc_partition: BE_1
  - length: 30
    slices:
      - cpu: [0, 1, 2]
        sc_partition: BE2

partitions:
  - name: Airbus
    processes:
      - cmd: ./CPU/benchmark --fast
        initialization: yes
        budget: 1.5
      - cmd: ./sw-render -x 10 -y 15
        budget: 5

  - name: Boeing
    processes:
      - cmd: ./GPU/tool -a -b -c
        budget: 1.0±0.1
      - cmd: hackbench -p
        budget: 5

  - name: BE_1
    processes:
      - cmd: ./hello portugal
        budget: 3±1

  - name: BE_2
    proc_be:
      - cmd: ./hello czechia
        budget: 3
        jitter: 1
default_be: BE_1
```

4 Scheduler API

The following functions will be available to processes executed under the DEmOS scheduler. They will be implemented in a library.

As we want the DEmOS scheduler to execute arbitrary processes, calling neither of the functions below should be mandatory.

- `void demos_initialization_completed()`

  If the scheduler configuration file specifies that the process has an initialization part, the process will be started before the actual execution of the scheduler. The scheduler will not be started before the process calls to this function.

- `void demos_completed()`

  Signalizes to the scheduler that all work for the current window is completed.

- `void demos_register_overrun_handler(void (*handler)(enum reason))`

  (Do we need this?) Registers a function that will be called when a budget will be exhausted or frame end will be reached before signalizing completion. This function will be called at the beginning of the next window assigned to the process's partition.

4.1 Scheduler hooks

In order to implement various thermal policies, it is expected to hook them into the scheduler. As the scheduler will be developed by us, we can execute whatever we want at arbitrary times. Currently, we envision that the following hooks could be used for implementing thermal policies:

- one called when a process declares completion;
- one called when a process exhausts its budget;
- one called whenever a process starts executing;
- one called when all safety critical processes completed (or exhausted their budget) in an execution window;
- one called when all safety critical processes completed (or exhausted their budget) in a frame.

5 Proposed implementation

The scheduler will use Linux cgroups to implement process management and scheduling. The scheduler itself will run as a highest priority task so that it can preempt application tasks. For this purpose, SCHED_DEADLINE (or at least SCHED_FIFO) scheduling policy will be used, because it has higher priority than default Linux scheduling policy SCHED_OTHER. The cgroup *freezer* interface will be used to start and stop processes (or the whole partitions) in their assigned windows. The *cpusetc* cgroup controller will be used to confine the partition to slice's CPUs.

References