UPPSALA
UNIVERSITET

15 011
Examensarbete 45 hp
April 2015

Multicore mixed-criticality with a hierarchical real-time scheduler and resource servers

Emanuel Berg

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Multicore mixed-criticality with a hierarchical real-time scheduler and resource servers

Emanuel Berg

This project is a real-time mixed-criticality system on an desktop dual-core Linux PC. It implements a hardware/software architecture and a memory throttling method that makes it possible to construct modular real-time systems whose parts can be analyzed in isolation, which simplifies certification.

The architecture is of two asymmetrically dedicated cores. The critical core runs a hierarchical scheduler to execute critical software: hard-coded, or as Linux processes with real-time metadata. The best-effort core runs arbitrary software.

Isolation between the cores - where processes are executed - is achieved by configuring GRUB; thereafter, taskset upholds the separation when new processes are created.

Because the cores share DRAM and memory bus, activity on the best-effort core can delay execution of critical software. This project shows how such interference can be dispelled - throttled, and bounded - so that real-time guarantees are still computable.

hs is a C++ userspace hierarchical scheduler. Task schedulers are at the lowest level: they implement and isolate the criticality groups.

When necessary, hs also freezes (and thaws) the software of the best-effort core. This is by means of a Linux cgroup. Each task scheduler has a budget to bound the amount of best-effort core memory interference. If a budget is depleted, the best-effort core is frozen whenever a task of that task scheduler executes, as to protect the critical tasks from interference. Best-effort core memory accesses are booked with perf_event_open. All budgets are reset at the beginning/end of every global period.

The task scheduling algorithm is a modified EDF: it is preemptive, but only at scheduling interrupts; also, the critical groups have CPU budgets which limit their slices of processor time. The task system is the sporadic task model. Task/scheduler systems are specified in text files which are parsed at the initialization of hs.

There are two resource servers at work: to safeguard the shared memory, and the CPU on the critical core. Both quantities are included in the schedulability analysis to uphold uphold isolation between the cores (memory) as well as between tasks (crticial core CPU time).

There is a zsh and tmux framework to automatize the (re)execution of experiments.

Two of the experiments are of particular interest, because they test the entire project design and implementation on a high (or qualitative) as well as low (or quantitative) level.

The memory experiment shows that best-effort core activity with respect to the shared memory can be throttled on the fly depending on the needs of the critical task executing in parallel. Alas, best-effort core submission lacks precision. If only the worst-case scenario is considered, results are bad. A long-term project of the same nature will show how these ideas can be (re)implemented to produce a higher level of exactness in throttling.

In addition, there is an audio experiment which tries to save particular tasks from being delayed in the face of overblown memory interference. This is the qualitative test, to see if the result in the memory experience can be put to use to save an individual task from being delayed. However, because hs is itself suspectible to best-effort core interference, the throttling mechanism reverts to a dysfunctional state along with the rest of the system as all memory is congested. The conclusion is that general inexactness translates to this system not reaches the predictability levels where one can make statements as to individual tasks and their deadlines.

Handledare: Pontus Ekberg
Ämnesgranskare: Philipp Ruemmer
Examinator: Edith Ngai
15 011
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

Report update: April 11, 2015

The purpose of this project is to implement a *hierarchical scheduler* that manages a real-time, mixed-criticality task system.

A mixed-criticality task system is a system where tasks of different criticality levels are executed on the same physical platform. The tasks are divided into groups according to their assigned criticality levels.

Real-time System

"An operating system capable of responding to an external event in a predictable and appropriate way every time the external event occurs." [8, p. 374]

Hierarchical Scheduler

A hierarchical scheduler is a scheduler that consists of several schedulers that are organized hierarchically, in a tree structure. The leaf schedulers are not themselves hierarchical, but ordinary: they schedule processes (or tasks). All schedulers above them, however, are hierarchical schedulers: they do not schedule processes, but the schedulers at the level immediately below.

The hierarchical scheduler and the hard real-time tasks all execute on a single, separate core: the *critical* core.

In parallel, best-effort software runs on a dedicated core as well: the *best-effort* core.

The cores share main memory, the DRAM, as well as the memory bus.

The theoretic scope of the project is to give methods to compute the response times for the critical tasks, considering that the best-effort core software can interfere because the hardware that is shared between the two cores.

Response Time

The *response time* is the wall-clock time elapsed from when the user invoked a command up and until the completion of the program that is associated with that command. The user does not have to be a human: it can be another program, e.g., a real-time scheduler.

The criticality groups must be isolated from each other so that the groups can be separately executed and analyzed. What is more, the critical core must be isolated from the best-effort core less the best-effort core monopolizes the memory, shutting out the critical tasks from execution thus making them miss their deadlines.

Deadline

In real-time computing, a task (typically a small program instance) must not only be implemented to produce a correct result (like any other piece of software), it must also complete its computation before a certain time limit. Such a limit can be expressed in different ways. The sporadic task model used for this implementation expresses time constraints in terms of a *deadline* for each task. The deadline is relative to the task arrival time: it is thus a static parameter, attached to the task much like the source code that is to be executed. If the arrival of task $\mathtt{t_i}$ is at time $\mathtt{c}$, and $\mathtt{t_i}$ has deadline $\mathtt{D_i}$, that means in absolute time, $\mathtt{t_i}$ must be completed no later than $\mathtt{c} + \mathtt{D_i}$, otherwise $\mathtt{t_i}$ is considered a faulty piece of software, no less so than was there a bug in its code, making it crash or produce an incorrect result. (In some settings, a correct result – which is delivered late – can have even worse consequences than an incorrect result, that was delivered in time. [21, p. 9])

The solution is two resource servers. On the critical core a CPU-time resource server monitors and limits the CPU time of individual groups as to prevent starvation. Likewise, a resource server limits the allowed number of best-effort core memory accesses.

The purpose is to realize an architecture and implement a memory throttling method that will make it possible to construct modular real time computer systems whose parts can be analyzed in isolation.

1.1 goal

The goal of the project is to be able to:

1. execute tasks of different criticality levels on the critical core; while
2. executing best-effort software on the best-effort core; so that
3. software from either core can access the shared DRAM; but
4. best-effort core memory interference still does not break critical-task schedulability analysis.

1.2 criticality groups

A critical group is a subset of all the real-time tasks in the system. The critical groups are mutually exclusive: every task belongs to one (and only one) critical group.

The reason to divide the critical tasks into groups is to be able to express and employ a more flexible and fine-grained real-time system. In this project, there are other parameters apart from the criticality level attached to each group.

The task classifications are to be used by the real-time operating system to guide its actions. Real-time analysis is to be applied to the groups as isolated entities, and for an entire system one group at a time, as if the other groups did not exist.

Real-time Operating System

The *real-time operating system* (RTOS) is the software that makes sure that the real-time tasks are executed in such a way that they are all guaranteed to complete before their deadlines expire. In principle, there should be several ways to implement such a system. In practice, the most common solution is to have a real-time scheduler instead of the ordinary scheduler, and then to annotate each task with real-time metadata, such as worst-case execution time and deadline. The real-time scheduler will process the metadata, as well as the overall system state (including the time), so that tasks are enqueued and selected for execution in a manner that will ensure compliance with the real-time constraints.

The reason to create a task-based system is that real-time systems interact with the outside, physical world. Because there are so many aspects of the physical world that can be measured, and in so many ways – and all has to be done continuously, virtually in parallel – a powerful solution is to employ a set of tasks, wherein each task is responsible for collecting and processing data of one particular such aspect. For example, one task reports temperature, another volume, yet a third system proximity to other objects, and so on. This model mimics the general-purpose process-based computing model, which in itself has many advantages.

There are several advantages to having several criticality groups:

- The system is more **modular**: tasks can be brought in and out as groups, and each group can be analyzed for real-time schedulability independently of the other groups.
- The system is more **well-defined**, and **ordered**. For example, by assigning different CPU-budgets to different groups, the respective importance of the groups as subsystems are made explicit: in turn, this will affect the behavior of the system in execution.
- The system is more **flexible**: the different CPU-budgets assigned to different groups can be used to assign a budget that is adopted to the typical execution patterns of the tasks of that group.

Schedulability

For any task set (where the tasks adhere to a specific task model), and a particular scheduling algorithm, there is a test than can determine *schedulability*. That test answers the question: "Can the algorithm make a schedule, that involves all the tasks in the set, so that, if they are executed accordingly, all tasks will complete before their deadlines?" If so, the task set has passed the schedulability test for that algorithm: the task set is *schedulable*.

1.3 real-time isolation

The critical groups must be isolated from each other in real-time analysis. If any one group is deemed schedulable, this must hold for any run-time scenario which can involve the tasks of that group, as well as those of all others. That is, in order to provide guarantees for a predictable execution of the tasks of any one critical group, it is necessary to process the runtime behavior of the tasks of all other groups as well.

Predictable

Computer systems are deterministic by definition. If they are *predictable*, that determinism is lifted to a human level: the user is able to make a statement – that something particular will happen, and not later than at a certain time. The statement is then proved to hold by use of formal methods. If the statement holds in theory as well as practice, the system is predictable, at least with respect to that particular statement. A predictable system consists of a set of likewise predictable statements/behavior patterns.

In practice, the resource server solution is an attempt to formalize and simplify such considerations to the point where an individual group does not have to be analyzed in a way that actively has to consider the peculiarities of the other groups, their tasks, and statuses.

Whenever needed, the system must intervene to prevent unruly behavior, right before it happens. If the tasks of a certain group collectively have depleted their allocated resources, the group must be hindered from further activity, less it will jeopardize the predictability of tasks that belong to other groups.

Resources

The *resources* of a computer can be either in hardware or software. Hardware resources are the memory, the CPU, the disk, etc. Such resources can be either static – e.g., the disk size – or dynamic – e.g., the available memory at the time of a certain workload. Resources can also be in software. If a program has been assigned a certain number of memory accesses, this budget is a resource: if 0, it is a depleted resource until it gets resupplied.

1.4 different kinds of computer systems

The genealogy of computer systems – from a desktop OS to a fully-fledged real-time system in the physical world – can be sketched like this:

- *interactive* non-timesharing – This is a typical computer system for a user to use for various, mundane purposes. The software that runs on this system does not have any associated deadlines: everything is simply executed without any guarantees, but also without any cumbersome real-time layer or metadata attached to executing programs. Still, in some sense, even such systems are “real-time”, because human-computer interaction would be unbearable if for example a keystroke sometimes had to process for several seconds before the character appeared on the screen.
- *timesharing* – A timesharing system is an interactive system, but it is also a multiprogrammed and multitasked system that supports several active users to be logged in at the same time. The implementations of such systems often employs the process-based architecture, which is also what is found in many real-time systems. This shows the proximity between timesharing and real-time systems, the difference being the

absence of deadlines in a timesharing system. In general, a timesharing system is oriented toward active *humans*, who provide instructions what the computer should do, while a real-time system is oriented toward the outside *physical* world, and whatever goes on there is sampled and processed as input to trigger various responses. [23, p. 3]

- *soft* real-time – Here, software *is* associated with deadlines. If a deadline cannot be held, however, that does not imply system-wide disfunction. Rather it is up to the system designers (and sometimes the user) to figure out what to do in such an event.
- *firm* real-time – Here, deadlines are soft in the sense that breaches are not encouraged, but allowed; on the other hand, deadlines are hard in the sense that computation whose result arrives late has zero value, and is discarded. (In some settings, delayed-computation data can even be dangerous if used as if it had arrived in time.)
- *hard* real-time – Here, a deadline breach implies a system fault.
- *critical* real-time – Here, a deadline breach implies a hazard to people's health. Critical real-time obviously implies hard real-time. What is not as obvious is that hard real-time does not imply critical real-time. For example, consider a computer game: a fast-pace, interactive game is for all practical purposes hard real-time, even though the implementation probably lacks explicit deadlines – still, if the game does not react virtually instantly to inputs and in-game events, it is unplayable. However, an unplayable game is not a hazard to the health of any living thing, so for this reason it is not considered critical real-time.

1.4.1 hybrids and gray zones

The real-time system classifications are not cut in stone. Many system consist of subsystems of different natures. Actually, there can be combinations within a subsystem itself: consider a real-time system where software is associated with *both* a soft and a hard deadline. [6, pp. 2-3] If the program does not complete before the soft deadline, the system reallocates resources so the program will be more likely to complete, quicker. Only when the task still has not completed at the time of the hard deadline, system fault is flagged.

1.4.2 where does this project fit in?

The hierarchical scheduler schedules pieces of software which all have parameter values – the real-time metadata. Thus, any system that runs on top of `hs` will be at least a soft real-time system. But it could also be a firm or hard real-time system, or something in between, with none or minimal changes to the code required. Ultimately, it is the user who decides what software will run and how that software will behave, and when. Those decisions will shape the entire system, including its place in the real-time ladder.

As for now, what is hard-coded is the sporadic task model: software tasks are annotated with the parameters WCET, deadline, and period.

WCET

WCET (Worst Case Execution Time) is the wall-clock time, or response time, of a program: the amount of time that passes from invocation to completion, in the *worst* possible case (i.e., it is impossible that it could ever take any longer to successfully terminate the task). The WCET parameter is helpful in models, to do computation, and it can also steer software behavior. Nonetheless, it is most often not possible to derive the WCET, in spite of much Computer Science research devoted to methods to that end. [19, p. 110]

There are several reasons WCET may be incomputable: task-execution conditions may vary – externally (the physical context), but also internally (e.g., system load); also, tasks may depend on other tasks to complete before them, or to release resources they have monopolized; and, the system may be a multicore, or even distributed with asynchronous message passing in between. [4, p. 3]

In this project, both in software and in analysis, it is assumed the WCET is correct, wherever it appears. Actually, WCETs are simply assigned by the user with no questions asked if it is realistic or not. Wherever available, indata verification is done in only terms of *other* parameter indata: the associated software is not examined.

The criticality side to it is even more unrestrictive. While the "criticality" groups indeed can be arranged to express criticality (as in physical hazard) they can just as well be thought of as labels of task groups that belong to a certain local scheduler, with no criticality implication whatsoever – in execution, the parameter values of the scheduler are what counts, not any qualitative implication that the human mind is fond of doing.

Scheduler

Computers are capable of executing several programs concurrently – or seemingly so. Actually, the CPU switches from program to program, executing each just so much as to create the illusion that everything happens in parallel. [13, p. 117] This makes for responsiveness to the user, but also better use of hardware resources: for example, while one program waits for I/O, it yields the CPU to another, computation-intense program (as the program-put-on-hold cannot make use of the CPU anyway, lacking I/O data). A special program decides what program executes, and when: the *scheduler*. This program is part of the kernel as it is needs memory and otherwise hardware access. There are several scheduling algorithms that the scheduler can employ to make its choise what program to run. Programs are associated a run-time data structure – the process control block, or PCB – that will provide the scheduler with uniform metadata on the state of the program in execution. Contrary, on multicores, true concurrency do exist as programs are executed in parallel, on different CPUs (or cores).

The hierarchical scheduler does not do anything by itself, but in combination with different software sets, it can schedule as many different kinds of systems.

1.5 why a real-time multicore?

Multicore architectures is one way the CPU industry brings faster computation to computers.

Also, with multicores, there is more computation out of less power. This is important not the least to embedded systems, which often have to operate on a limited power supply. Embedded systems – because they apply to problems outside the world of computers – are a huge implementation field for real-time systems.

Real-time systems on multicores are yet rare, in part because of the increased unpredictability due to the multiple sources of computation. This poses difficulties both theoretically – guaranteeing real time analytically – as well as to enforce it in practice, because distributed computation implies communication and synchronization delays that must be bounded.

1.6 meta

The writer of this report as well as the programmer of all new related software can be reached at `embe8573@student.uu.se`.
This is a BibTeX entry for this report:

```
@techreport{multicore-mixed-criticality-with-a,
 author      = {Emanuel Berg},
 title       = {Multicore mixed-criticality with a hierarchical real-time scheduler and resource servers},
 institution = {Department of Information Technology, Uppsala University, Sweden},
 type        = {\unskip\space},
 year        = 2015
}
```

The report as well as the source code can be found at the author's home page: `http://user.it.uu.se/∼embe8573`

2 Related work

The classical paper on hierarchical scheduling is [20]. However, the scheduler implementation of this project is not perfectly layered. Optimally, any hierarchical outline relies on a one-way communication data flow, where superior entities never ask (but only tell) the lesser components what to do (and, whenever necessary, provide them with what they need to do it). For a hierarchical scheduler, that translates to the root scheduler scheduling the schedulers below solely based on their properties, without looking any further below, and only after one particular scheduler has been favored would that scheduler decide what of its task to execute, without the interference of the above scheduler. Instead, because of practical considerations when implementing the EDF algorithm, the root scheduler is not ignorant to what happens at task level: contrary to just barking orders, it must poll the below layers for information that will determine its decision.

EDF

EDF (Earliest Deadline First) is a real-time scheduling algorithm. It requires each task be annotated a deadline (typically an integer). This is a static, a-priori property; it should be interpreted as the time interval between the release of the task and when it must be completed, less it will be considered delayed (i.e., a failure for hard real-time systems). When scheduling, the EDF-scheduler will select for execution the task with the most immediate absolute deadline – that is, the earliest deadline, first. Because the absolute deadline is a function not only of the deadline task parameter, but also of the run-time instant the task was released, EDF is a dynamic scheduling algorithm.

Previous work to a large extent consider only two criticality levels: i.e., one crticial level and one level which is best effort. [2] [24, pp. 299-308] In this project, it is possible to implement and analyze an arbitrary number of critical groups, and have best-effort software run in parallel.

In this project, unlike [2], there is not a best-effort sphere of possible memory access once the allocated budget has been depleted. And, unlike [24], this implementation is userspace only.

In [5], the unusual memory architecture of a network-on-a-chip system (a NoC) is used to provide isolation between criticality levels. Here, specialized hardware is used, including a predictable multiprocessor, whereas our project uses commodity components which are by original design unpredictable.

In [10], a monitor is (in part) instructing a scheduler what process to execute from the suggestions of four queues. The queues are a division based on qualitative properties of the processes, so that such properties also influence what process to run. The algorithm is “slack”: context switches are not always carried out at times of priority inversion. Instead, to decrease overhead and improve best-effort quality-of-service, preemption occur only at the very moment it has to, less a critical process will not be able to make its deadline. Our project is similar to their but their solution is more sophisticated (and complicated). In our project, the critical groups (which can come in any number) are holder of scheduling and real-time metadata that is arbitrary and cannot be computationally derived from properties of the processes which make up the groups. Also, our monitor serves only to freeze the best-effort core – it doesn’t influence any scheduler on either core. And, freezing *always* happens when the maximum allowed DRAM accesses have been accounted, so our algorithm is not slack, either.

One of the most sensible suggestions to the problem how to uphold predictability while sharing resources is presented in [11]. Here, the shared memory is divided into banks. However, the banks are not merely distributed over the critical tasks – while that would indeed provide isolation between criticality levels, it would do so at the price of no longer sharing the memory and thus have the whole system constantly operate at the bottom end of what it is capable of. Instead, the tasks are dynamically mapped to the banks by means of the scheduling algorithm. Consequently, the entire scope of the capabilities of the system can be employed: at the very least, there is the subdivision and distribution of memory; at most, there is unrestricted access. The challenge is to find or (re)design an algorithm that maps the tasks to the banks in such a way that the system can run on the most cylinders the most of the time. In effect, the once contradictory poles – predictability vs. resource sharing – are transformed into an optimization problem.

Also, in [9] the solution is a subdivision of the DRAM between the critical tasks. Each critical task is assigned a DRAM bank. There is one critical task for each bank. This means the critical tasks do not share memory anymore: instead, they use their designated “virtual memory” bank which is a fraction in size to the original DRAM. Nonetheless, what is innovative about this project is that for each bank, alongside the singular critical task, there can be an arbitrary number of best-effort tasks. Access to the bank memory is granted according to a priority algorithm. Priorities are fixed with the critical task in each bank having the highest priority. This uncomplicated scheme has the benefit of providing complete isolation for the critical tasks, both from each other and from the best-effort tasks. The drawback is that resource utilization is sensitive to both the setup and the dynamic situation of every execution.

Another impressive suggestion is [12]. Here, there are no dedicated cores – all critical tasks, of any and all criticality levels, coexist across the cores. Nor are there any budgets to be distributed by resource servers. This makes for minimal a-priori arrangements. Also, there is no reliance on specialized hardware.

The solution is based on subdividing all cores as well as the shared memory along the criticality levels. A dynamic scheduling algorithm synchronizes the cores so that only tasks from the same criticality level access memory at the same time.

Interestingly, the authors comment on the type of solution that is presented in our paper:

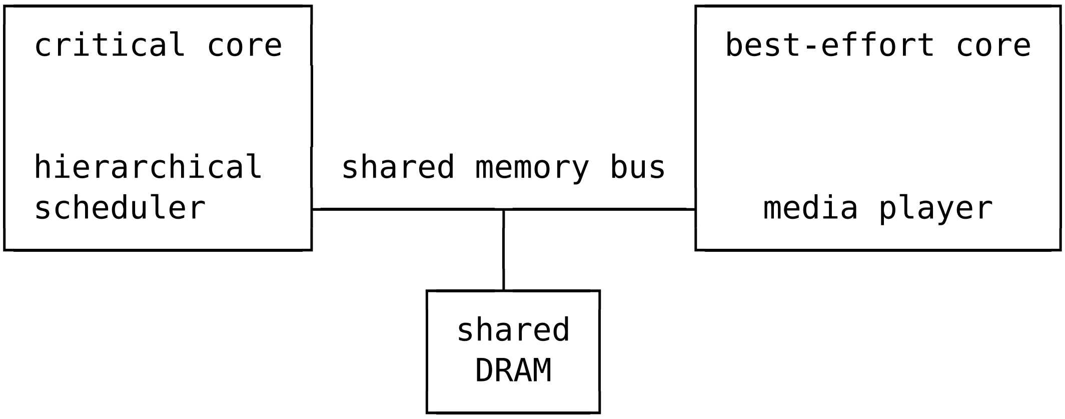
- Our solution is flexible in terms of managing processes – e.g., to include arrivals. This is true, especially compared to monotonic scheduling. However, their solution is even more flexible as all cores share all work. The lack of a-priori metadata markup and budget assignments is a two-sided sword. On the one hand, there is no room for humans to screw up the algorithm. On the other hand, no expert human can tweak it to perfection. In general, we say their solution is better, assuming a time-tested dynamic algorithm to do the allocations.

- Our solution requires “a high design and implementation overhead”. Here, it is unclear if this aims at the scheduling and throttling software, or the a-priori setup of task systems and resource budgets. If it aims at the software, our project shows such software is not insurmountable to create. However, if it aims at the setup, we agree, and especially if compared to their project, which does not require any of that.

- Our solution cannot be done with COTS components. COTS is “commercial off the shelf” computer hardware. What truth there is to that of course depends on the definition of “COTS”. In fact, a performance monitor counter (PMC) is all specialized hardware that is required, and such are found in many desktop PCs.

3 System overview: two cores, shared DRAM

The system consists of two cores: the *critical* core, and the *best-effort* core: Those cores share DRAM and memory bus.



Multicore Architecture

3.1 the critical core

The critical core runs critical tasks. The mixed-criticality scheduler, as well as all the critical groups, reside on the critical core.

3.2 isolation between the cores

Whatever happens on the best-effort core cannot be allowed to influence the execution of critical tasks to the extent that even a single critical task cannot complete before its deadline. To that end, the interference from the best-effort core is bounded by a parameter for each critical group. The value specifies how many memory accesses the best-effort core is allowed to do for a specific time span, while a task of that group is executing.

3.3 the best-effort core

The best-effort core runs a media player.

To run a media player as the best-effort task is interesting because a media player task does not lend itself to static real-time analysis. A media player is user-oriented software. Some downgrade in quality can be tolerated. If the media content is streamed from a network, data may be sketchy, and arrival spotty; and, the service is most often designed to work on different end-hardware platforms. [1, pp. 4-13] Also, any substantial quality improvement can be detected by inspection.

As the real-time system can block the best-effort software from accessing memory, the performance of the best-effort software can be much worse compared to if it did not have to subject itself to the demands of the real-time system. The extent of this performance downgrade is evaluated by executing and measuring the media player, when it runs in parallel with the critical task system.

4 The Linux implementation

4.1 the real-time system

The hierarchical scheduler, or `hs`, that runs the critical tasks is written in C++: C++ is fast, which matters because it is desirable to reduce the scheduling overhead. The object oriented support in by C++ is also helpful to express the different components of the hierarchical scheduler, which is modular by concept. More generally, C++ is an extension of C, which is the system programming language of choice since the early-mid 70s. [22, pp. v-vi, 6, 41-42]

Programmers of real-time systems have discussed what makes for a good implementation language for such systems. [7, pp. 20-22] provides a list which in terms of technology amounts to:

- The language should come with debugging tools, including a compiler which provide compile-time warnings and error messages, in order to detect problems at the earliest stage possible.
- The language offers features for modular design and development, e.g. user-defined datatypes (or classes).
- The language is portable, fast, and vast with respect to what problems it can express and solve.

C++ does that, of course.

There is also a discussion whether such a programming language should in itself include real-time functionality, and provide it transparently on any platforms for which it is ported, *or* if real-time functionality can be delegated the underlying OS, and just be accessed through an API. [7, pp. 22, 131] (Bear in mind that those approaches are not mutually exclusive.)

In this project no attempt is made to avoid using OS functionality, and the C++ used is all vanilla.

Both critical and best-effort cores run on Debian GNU/Linux. Linux is a good all-around platform for programming; it comes with both general and specific tools that were indispensible while developing the hierarchical scheduler, and while setting up the overall dedicated-core architecture as well as the framework to carry out the experiments.

The Linux kernel and distribution used for this project are:

```
$ uname -a
Linux debian 3.16-2-amd64 #1 SMP Debian 3.16.3-2 (2014-09-20) x86_64 GNU/Linux
```

4.2 the critical tasks

The software of the critical tasks is written in C++. The task software is hard-coded functions that are compiled together with the hierarchical scheduler – but, the task systems are independently defined, in text files, that are read at run-time.

The tasks can run synthetic workloads or benchmarks to collectively produce a behavior similar to a real system.

`hs` can also run ordinary Linux processes. If so, they are forked and controlled by signals. The interface to define and later execute such systems is the same as for the hard-coded task software, only, as for now, task software and Linux processes cannot be mixed in a single system.

4.3 `perf_event_open(2)`

At every global tick, the hierarchical scheduler invokes the Linux `perf_event_open` tools to poll the number of last level cache (LLC) misses that originates from the best-effort core. An LLC miss implies a DRAM fetch and use of the memory bus to communicate both the request and the data. (Details on the caches are in [section 5.3, page 28].)

By booking the number of such requests, it is possible to calculate the number of LLC misses since the beginning of the global period.

The outcome is at every tick compared to the maximum-allowed accesses for the executing critical task. If the performed best-effort accesses exceed the number allowed, the best-effort core is frozen: all software there will be suspended, and only resume whenever a task executes that belongs to a task scheduler with a non-depleted memory budget.

Because data is *polled*, and polled periodically, the best-effort core will be able to exceed the number allowed often, as it can only be frozen on a scheduling interrupt. This problem can be minimized by increasing the frequency of the global scheduling interrupts. However, there are other problems associated with a high frequency: [section 16.2, page 77]

The problem of periodic, non-immediate enforcement of memory budgets could possibly be avoided if over-use instead would trigger a kernel interrupt, that would immediately freeze the best-effort core. In such an implementation, the memory budgets could be communicated to the kernel by means of a syscall.

4.4 `cgroups`

To freeze and thaw the best-effort core, `cgroups` is used. This method stems from the high-performance computing world (HPC) where it is often desired to collectively suspend or activate process groups.

5 Real-time Linux and Unix

Conventional wisdom has it that UNIX and Unix-like systems like Linux cannot host real-time systems without considerable changes to their kernels.

It is argued, because of the layered architecture – a kernel space, a userspace, and asynchronous syscalls in between – that any userspace-only application is inherently unpredictable. [15, pp. 17-18]

Furthermore, many Unix-like systems do not have a scheduler suitable for shuffling real-time processes. (Linux has such features, but they need to be enabled explicitly.)

While those arguments are true to form, whether they are true in essence depends on the preferred definition of a real-time system. The definition used for this project – "An operating system capable of responding to an external event in a predictable and appropriate way every time the external event occurs." [8, p. 374] – is very much attainable on Unix-like systems, with or without kernel surgery.

5.1 Linux real-time schedulers

As for kernel version 3.14, Linux provides four schedulers that may be more fit for real-time purposes than the default one, which for its part maximizes average throughput to benefit a varied dose of interactive computer use (e.g., editing).

The real-time schedulers are:

- `SCHED_FIFO` is a fixed-priority scheduler, only real-time processes do not get preempted by common processes. If several real-time processes have equal priority, the one who has the CPU will not let it go until completed at what time another contender monopolizes the CPU.
- `SCHED_RR` is like `SCHED_FIFO`, only when several real-time processes have equal priority, they do Round Robin, still excluding every other process from the CPU.
- `SCHED_OTHER` is the round-robin, time-sharing algorithm where processes execute for certain timeslices.

- SCHED_DEADLINE is the EDF algorithm. This can likely be used to enhance this project, which employs a somewhat modified brand of EDF, and – when dealing with real Linux processes – enforces it with userspace signals.

For this project, the ordinary Linux scheduler is used. To use a real-time scheduling policy complicates matters, especially when hs forks processes and then attempts to control them through signals. But in all fairness, Linux has many features that ultimately can transform it to just about any system – including a hard real-time system. For example, mlock(2) can be used to lock the virtual address space of a process into the RAM. As nothing of that has been touched upon, scheduling is left alone as well.

5.2 the Linux and C++ clocks

To uphold an even rate of periodic scheduling, the C++ library function std::this_thread::sleep_until is used, which however may "block for longer [than what has been specified] due to scheduling or resource contention delays". [1]

Actually, there can be widely diverging results due to many factors. To illustrate the extent of imperfect periodicity -l (or equivalently --log) can be used to have the hierarchical scheduler output the current time in nanoseconds, at every tick. (Again, the same library functions are used.)

It is likely that using a real-time scheduling policy for hs would result in a more even tick rate, especially if hs is intended to run among many other processes that are not real-time.

Here is a sample output for a period of one millisecond:

```
3594454136974
3594456742892
3594456842067
3594459262486
3594459326740
3594459349369
3594460102258
3594461102664
3594462098601
```

[1] http://en.cppreference.com/w/cpp/thread/sleep_until

```
3594463098448
3594464103883
3594465098703
3594466102182
3594467103985
3594468097687
...
```

Then, the following method is used to calculate the offsets from the intended tick times:

```
offset_0 = time_1 - time_0 - DESIRED_TICK
offset_1 = time_2 - time_1 - DESIRED_TICK
...
```

This produces, for the example trace:

```
 1524064
-958095
-488762
 323073
-289016
 1605918
-900825
 1420419
-935746
-977371
-247111
 406
-4063
-153
5435
...
```

Last, for all offsets acquired, some statistical data is computed:

```
readings:             9999
mean:                 7.000000      # drift size
```

```
variance:              1828079556
standard deviation:    42756.047011  # drift stability
min:                  -977371        # observed worst cases
max:                   1605918
```

Note: It has been observed that the early tick times of a run are much more inexact than those of the rest of the trace. This is the case in the above example as well. It is telling that the first mere 15 readings contain both the minimum value (`-977371`), and the maximum value (`1605918`), of all 9999 readings, whose mean is only seven! (For a longer trace, the mean would be even smaller.) As for now, this behavior remains a mystery.

To exemplify, here are the offsets found at the end-most part of another trace – a trace that likewise has a specified rate of a single millisecond. As is typical, these offsets are much smaller than their early brethren.

```
 10743
-431
-4902
 3480
-4622
-153
 6832
 2642
-7696
 3480
 127
-152
-4622
-153
```

5.3 hardware

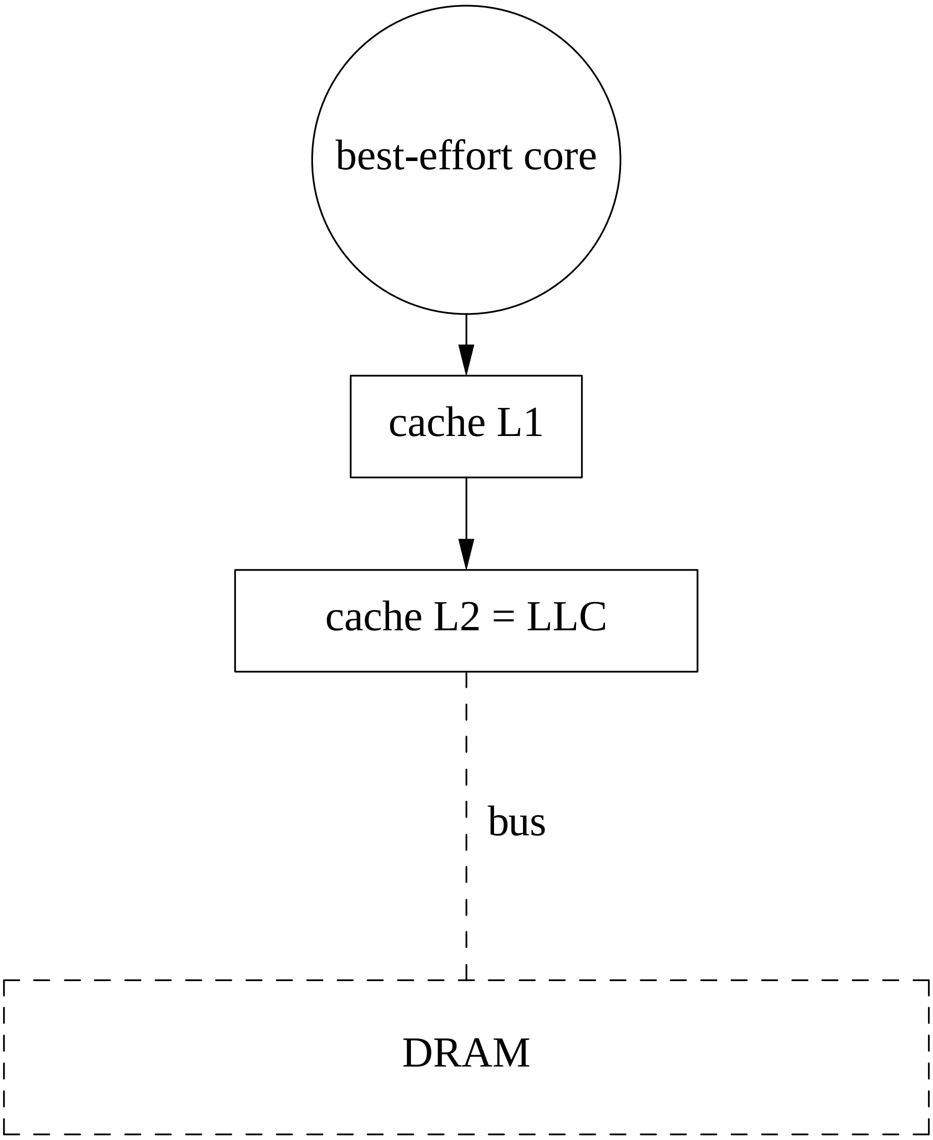
The hardware platform must be considered because the real-time task superstructure and all the individual tasks share hardware resources. Specifically, the critical tasks – including the hierarchical scheduler and the task software, and all forked processes – share the critical core for CPU-execution; and, they share the DRAM and the memory bus with the best-effort core. That has to be accounted for both in analysis, *and* in practice, where monitoring and throttling is employed to hinder over-use of shared resources.

The CPU is a dual core `x86_64`:

```
$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                2
On-line CPU(s) list:   0,1
Thread(s) per core:    1
Core(s) per socket:    2
Socket(s):             1
NUMA node(s):          1
Vendor ID:             AuthenticAMD
CPU family:            15
Model:                 35
Stepping:              2
CPU MHz:               1000.000
BogoMIPS:              1989.83
L1d cache:             64K
L1i cache:             64K
L2 cache:              512K
NUMA node0 CPU(s):     0,1
```

DRAM and memory bus usage is monitored with a PMC, which is accessed by means of the `perf_event_open` Linux tools.

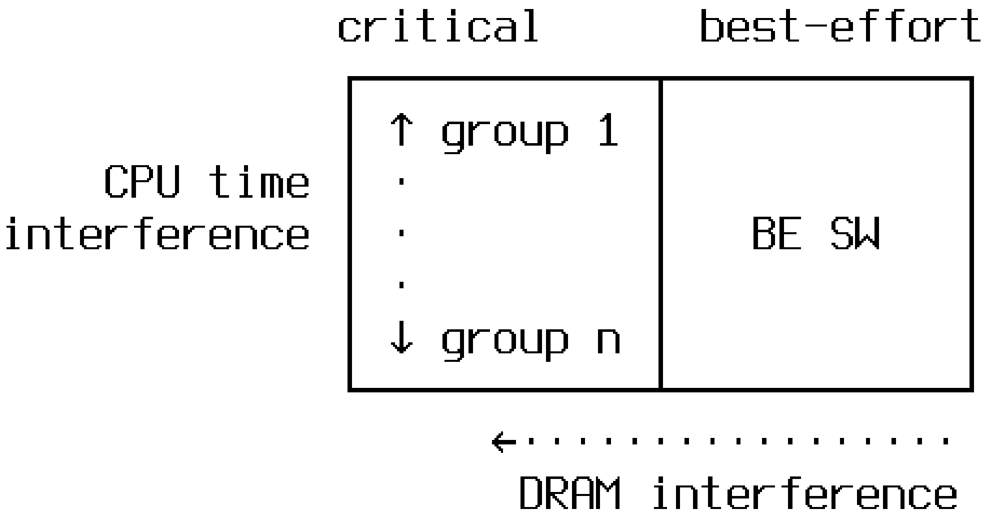
The `L1` and `L2` caches are not shared between the cores but appear pairwise, with one pair for each core. Save for the CPU registers, `L1` is the smallest memory storage unit, with `64K` for data, and the same amount for instructions. Below `L1` is the `512K` `L2`, which is the last level cache, or LLC, right above the shared DRAM. Thus, a failed memory lookup at the second level is indicative of an imminent DRAM access. In this project, LLC misses are counted as a way to count fetches from the shared DRAM.



Memory Architecture. First a miss in the last-level-cache (LLC), then a DRAM fetch.

5.4 isolation

All critical groups must be isolated from all other critical groups, but they must also be isolated from the best-effort software. Here, the intersection is not the CPU, but the shared DRAM and the shared memory bus.



Isolate us! Resource sharing interference.

Isolation must be implemented in practice, as well as digested into something that is formally computable, so it can be used to determine task response times, and, later on, overall system schedulability. In this, software as well as hardware are considered.

It is important to note that the word “isolation” is not used in the general sense, but in terms of real time schedulability analysis and task response-time computation. If `A` is isolated from `B` in terms of `i`, that does not mean `A` is physically separated from `B` so that `B` cannot ever influence `A`. On the contrary, it means that `A` *is* influenced by `B` precisely through `i`, only the designers and implementors of the system has accounted for this, for example by providing throttling mechanisms that in practice will provide a bound for the worst-case interference `B` can have on `A` (through `i`). This bound is then used in the response-time calculations, which is the theory mirror image of the real-time implementation.

5.4.1 starvation

Because the best-effort core and the critical core share the DRAM, and because the mechanism that throttles the best-effort core resides on the critical core, it is possible that the best-effort core can overflow the memory, shutting out the critical core to the extent that the critical core cannot throttle the best-effort core, whose memory budget thus cannot be enforced. In terms of real-time, that would immediately break the system.

5.4.2 isolating the cores on Linux

The cores on the project computer are originally general-purpose: the Linux scheduler can execute any process on either. In order to turn these cores into one critical core and one best-effort core, a kernel parameter is set that will stop the Linux scheduler from using the best-effort core; then, a userspace tool is used to start best-effort processes so they will only execute on their designated core.

This turns the system into one where there are dedicated cores for specific software: distribution of processes over the cores can be a function of a-priori policy decisions.

To isolate one of the cores from the Linux scheduler until explicitely told otherwise, in:

```
/etc/default/grub
```

put (or change) `GRUB_CMDLINE_LINUX_DEFAULT` into:

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet isolcpus=1"
```

Then, run

```
sudo update-grub
```

and reboot.

After that, here is how to execute a program, say, `forever`, exclusively on CPU 1:

```
taskset -c 1 forever
# to confirm:
ps -o psr,comm -p `pgrep forever` # psr = processor = core
```

This method is used to invoke all best-effort software.

The method to isolate the best-effort core by setting a kernel parameter in a GRUB initiation file does isolate the core from most processes that are not explicitly told otherwise. Still, a few system processes remains on the best-effort core, unaffected by the GRUB configuration. This is why, on the critical core, the hierarchical scheduler and the critical-task software co-exists with some modest Linux system software:

```
watchdog/1
migration/1
ksoftirqd/1
kworker/1:0
kworker/1:0H
kworker/1:1
kworker/1:2
kworker/1:1H
```

Fortunately, these processes are low-key: they demand CPU or memory resources in quantities that are zero or negligible in the face of userspace processes that are intentionally and explicitly invoked to be executed exclusively on the best-effort core.

To illustrate, one execution of a media player on the best-effort core for 10 seconds booked `70464921` memory accesses. During this execution, no activity what so ever could be detected from the above processes, which were monitored with `top(1)`.

A consecutive run of `hs` for another 10 seconds but with an idle best-effort core (i.e., without the media player) booked the number of memory accesses as: `0`.

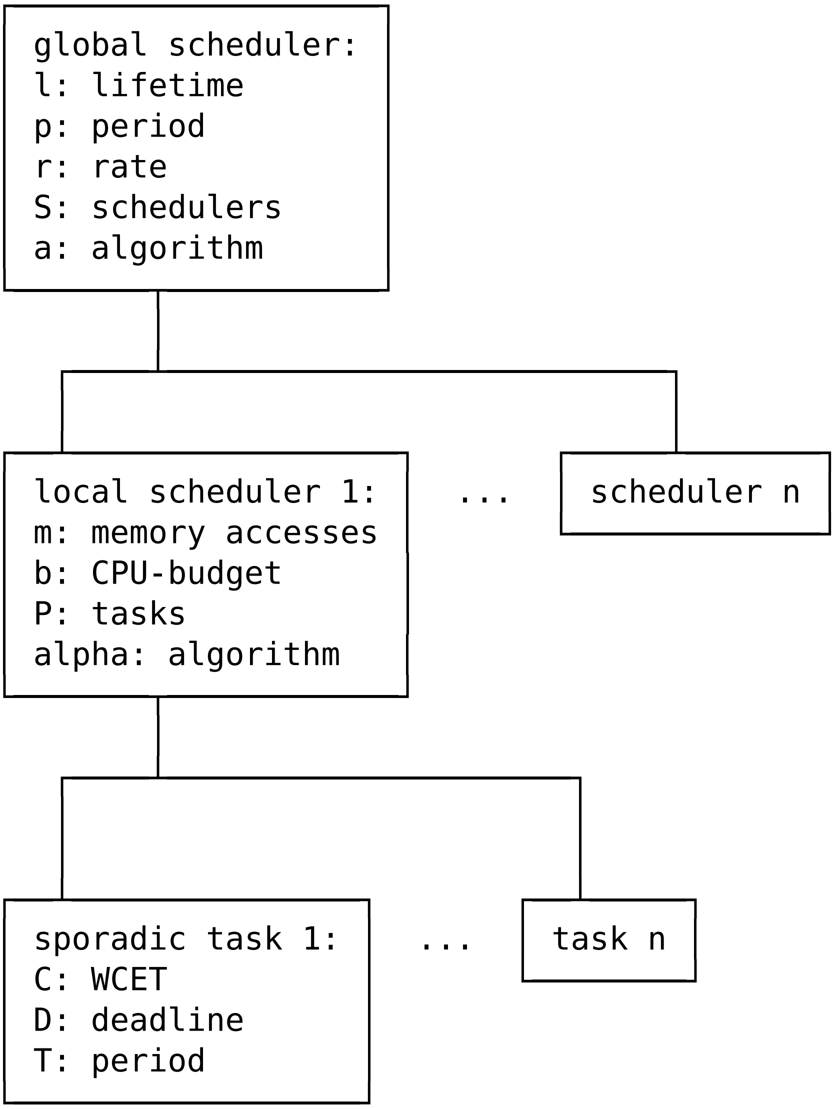
6 Scheduling

There is a single, top-scheduler, that schedules task-schedulers at the level below. The task-schedulers schedule tasks. At the down-most level reside the individual tasks. The tasks are grouped and assigned one unique task-scheduler: such a group of tasks, along with its scheduler, is the implementation of a criticality group.

Scheduling is done *hierarchically*. The top-scheduler is the highest entity. The task-schedulers are at the level below, themselves at the same level: they all belong to the top-scheduler. Scheduling is carried out at two levels: at the level of the top-scheduler, and at the level of the task-schedulers.

Scheduling is also, conceptually, done in parallel, or vertically: every task-scheduler schedules its task set. The tasks are themselves at the same, down-most level, below the task-schedulers to which they belong. But, unlike the task-schedulers, the tasks do not all belong to the same above scheduler: they are grouped, together with their designated task-scheduler, into criticality groups. There can be an arbitrary number of task-schedulers and each can have a task set which contains an arbitrary number of tasks.

The task system can be visualized as a tree: the root (or top node) is the top-scheduler; the `n` internal nodes immediately below are the task-schedulers; and, at the third and lowest level, the `k` leafs (or terminal nodes) are the individual tasks.



Hierarchical Scheduler

6.1 algorithm

The scheduling algorithm is global, budgeted, and preemptive EDF.

- It is **global** as it considers the critical tasks of *all* task-schedulers, and compares them on a scheduler as well as task-by-task basis.
- It is **budgeted** as all task-schedulers are assigned a CPU-budget. For each global period, the tasks of a particular task-scheduler can execute

for a total time not exceeding that budget.

- It is **preemptive** as a task, once assigned the CPU, can later be replaced, and put back into the ready queue, before it has completed. Here, the algorithm does not differ from the commonplace preemptive EDF of real-time scheduling, except for one implementation-derived detail: preemption can only be done at the time of a global scheduling interrupt. Those occur regularly and frequently according to a system parameter, which is set beforehand and thereafter does not change. Arrival of new tasks also takes place at those interrupts, so a task with the highest priority that just arrived will immediately preempt the CPU, as long as the task-scheduler it belongs to has not depleted its budget. So the algorithm is “polled-preemptive EDF”.

6.2 step-by-step description

Scheduling is defined as follows:

1. Execution regularly comes to a halt in order to do scheduling.
2. Whenever so, the top-scheduler tells all task-schedulers to do scheduling of their respective task sets. In this implementation, all task-schedulers do the same, commonplace EDF. The task states are updated. The task-schedulers all propose a task to the top-scheduler.
3. Of the proposed tasks, the task with the most immediate deadline is favored, as long as its task-scheduler has a non-depleted CPU-budget.

6.3 task priority

EDF is a priority scheduling algorithm where the priority is a dynamic property.

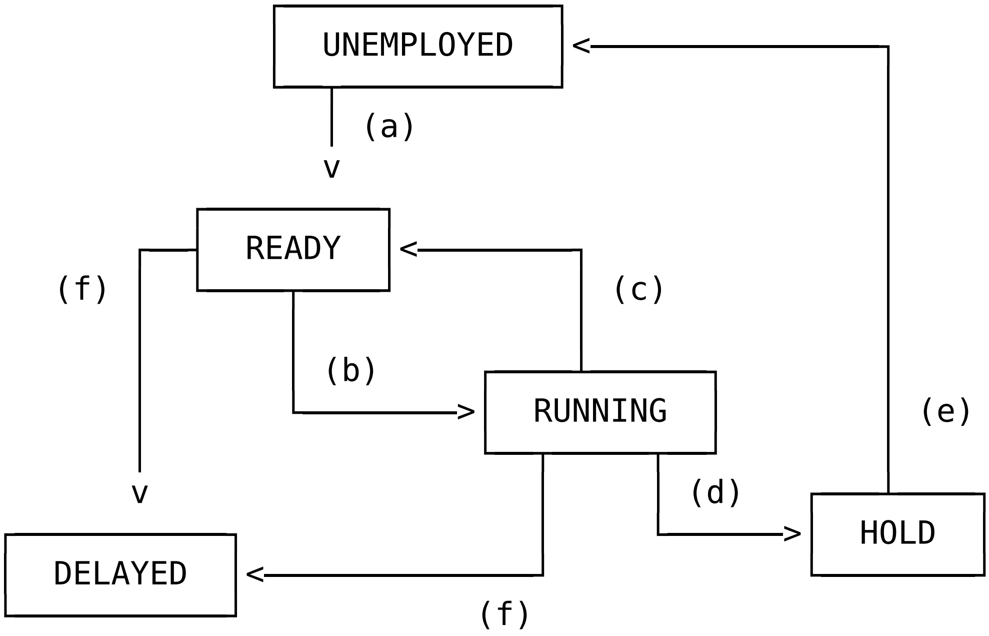
For this project, *locally*, at the level of an individual task-scheduler, the highest-priority task is the task with the most immediate absolute deadline, just like commonplace EDF.

However, *globally*, the highest-priority task must also belong to a task-scheduler that has not depleted its CPU-budget. If the task-scheduler has a depleted budget, its proposal to the top-scheduler is academic as no matter what deadline, no task of that task-scheduler will ever be favored. If a task belongs to a depleted task-scheduler, that is equivalent to the task having the lowest possibly priority. (If all schedulers are depleted the CPU will idle.)

Note that a depleted scheduler has a zero remaining CPU-budget. If the budget is non-zero, it is not depleted and its task will be considered, even if that task requires more than the remaining CPU-budget to terminate. (This is a possible window for optimization that has not been explored.)

7 The task finite state machine

Each individual task belongs to one (and only one) state, always.



Sporadic Task Implementation

7.1 transitions

Transitions occur regularly, and at the same time for all tasks, namely when the global scheduler interrupts execution to re-asses the state of the system.

The transitions are:

(a) The task arrives: it is employed. This happens after a semi-random delay: in practice, the task should arrive sooner rather than later. (The random delay can be disabled with the `-i` or `--immediate` option.)

(b) The task has the most immediate deadline in the system of all tasks that belong to non-depleted schedulers and is thus passed to the CPU for execution.

(c) The task is preempted by another task from a non-depleted task scheduler, a task with a more immediate deadline.

(d) The task is completed and is released by the CPU.

(e) The minimum inter-arrival time has passed, so a new task instance can be released.

(f) Timeout – the absolute deadline of the task is in the past, but the task still has not completed.

7.2 states

`UNEMPLOYED` Either of:

- The task has never executed, and has not yet been placed in the ready queue (where it would have state `READY`).

- The task has had at least one instance of itself executing (the `RUNNING` state); it completed its computation, and has been put on hold (the `HOLD` state). However, the time of necessary holding (due to the sporadic task model) has transpired, so another, new instance of the task can indeed be placed in the ready queue – only this has not happened, yet.

Which one of these situations is at hand is not of practical or analytical importance, but they illustrate the cyclic nature of the life of a task in this system.

The reason for this state is to get the sporadic, semi-random behavior of task arrivals.

Note that an unemployed task has not arrived so it does not have a pending deadline, and it cannot be selected for execution.

`READY` The task is queuing. It has been deployed – it has arrived – and may in part have been executed, only, if so, it has been preempted – either way, it is not completed so it waits, queuing. All time spent in this state is added to the response time of the task.

`RUNNING` The task is executing: the scheduler that holds the task has not depleted its CPU-budget, and the task has the most immediate deadline for all tasks that run on non-depleted schedulers. When executing, the task is closing in on the absolute deadline as much as when it idles in the ready queue. However, contrary to when idling, the task is also closing in (at the same speed) on its own termination, at what point it has received enough CPU time to complete its computation and terminate successfully.

`HOLD` The previous task instance has completed, and before another instance of the same task can arrive, it must hold until its period is over, in accordance with the minimal-interarrival, one-instance-at-a-time policy of the sporadic task model.

`DELAYED` The task has missed its deadline: the absolute deadline is in the past, and the task has not been assigned the CPU (state `RUNNING`) for a time that equals or exceeds the WCET of the task. In this system, a delayed task will not ever be completed, nor will any further instances of that task be released.

8 System description

Terminology: This section is a *description* of the hierarchical scheduler as a piece of software. The parameters correspond directly to C++ data types: in some cases classes, in some cases primitives. Most parameters are part of the interface and can thus be specified by the user without any changes to the software required. Only the parameter *names* are simplified, most often to single letters; and OO relationships are expressed as parenthesized lists: this is to make the description more accessible, and, later, to facilitate the use of the parameters in formulas. Nonetheless this remains a description of actual software and is not a model to be verified by formal methods – which, alas, at best would verify only the model itself.

8.1 the top-scheduler

`g` is the singular, or root, top-scheduler:

$$\mathtt{g = (l, p, r, S, a)}$$

eq. (1), top_scheduler

parameter	name	(unit)	description
`l`	lifetime	ms	The system lifetime. This is the amount of time that the top-scheduler will execute, from the time of invocation. As the top-scheduler is the topmost body of the system, `l` equals the system lifetime.
`p`	period	ms	The global period. At the beginning of each period the task-schedulers have their budgets (re)supplied to their individual, pre-assigned levels.
`r`	rate	ms	The scheduling rate by which `g` will schedule `S`. At every `r`, execution comes to a halt: scheduling is done, and execution resumes.
`S`	schedulers		The set of task schedulers that the top-scheduler schedules.

a	algorithm	The scheduling algorithm that g applies to S, in order to select what task to execute (for all $\mathtt{s} \in \mathtt{S}$). The algorithm updates and maintains the statuses of all other tasks and schedulers as well.

8.2 how the time parameters relate

$$\mathtt{0} < \mathtt{r} < \mathtt{p} < \mathtt{l} \qquad \text{eq. (2), relate}$$

If not $\mathtt{0} < \mathtt{r}$ there will be constant preemption and not any task execution at all.

If not $\mathtt{r} < \mathtt{p}$, then at each tick, the budgets of all $\mathtt{s} \in \mathtt{S}$ will have been resupplied, and what happened before the tick will not influence the selection of the next $\mathtt{s}'$, as then (and always), every $\mathtt{s}' \in \mathtt{S}$ (including s) is equally ready. If so, there will not be any isolation between the critical groups as that is expressed and implemented as the CPU-budgets of the schedulers. If the budgets cannot be depleted, they cannot play a role in this system.

If not $\mathtt{p} < \mathtt{l}$, there will not be any resupplies of budgets.

8.3 task scheduler

S is the set of task-schedulers.

The task-schedulers are the implementation of the critical groups.

$\mathtt{S} \neq \emptyset$ as, unless so, there is not any $\mathtt{s} \in \mathtt{S}$ for g to schedule (or any tasks to execute).

The set of task schedulers is:

$$\mathtt{S} = \{\mathtt{s_i}\} = \{(\mathtt{m_i}, \mathtt{b_i}, \mathtt{P_i}, \alpha_\mathtt{i})\} \qquad \text{eq. (3), task\_scheduler}$$

Notation: The subscripts are only given to indicate that the properties belong to a certain task scheduler. A single task-scheduler s would be fully described as $s = (m, b, P, \alpha)$.

8.3.1 m_i, the maximum best-effort core memory accesses

For each global period (of time p), the software that runs on the best-effort core must not exceed a certain number of memory accesses – m_i – during the execution of the critical tasks of s_i.

That way interference from the best-effort core vis-a-vis a specific critical group can be quantified, and thus bounded. The bound is used in response-time analysis of the critical tasks on a task-scheduler basis.

8.3.2 b_i, the CPU-budget

b_i is the budget assigned to s_i. The time the computer spends executing a $t \in P_i$ will reduce the working budget of s_i by the same amount of time. When the working budget is depleted for s_i, g will not consider s_i; however, g will restore the working budget of s_i to b_i at the transitions between global periods.

$b_i > 0$ as otherwise g will never consider s_i.

The CPU-budget feature makes it possible to isolate the critical groups from each other. For each global period, the tasks of s_i – P_i – cannot delay any task that belongs to another group by any more than b_i, the budget of s_i.

8.3.3 α_i, the (task) scheduling algorithm

α_i is the scheduling algorithm s_i uses to propose a $t \in P_i$ for execution.

In principle, α_i can be an altogether different algorithm than the global scheduling algorithm a. α_i can also be different from one task-scheduler to another. In this project, all task-schedulers do commonplace EDF, while the top-scheduler do a slightly modified EDF, the most notable addition being the ruling-out of schedulers that have depleted budgets.

8.4 $\mathtt{P_i}$, the sporadic task set

$$\mathtt{P_i = \{t_j\} = \{(C_j, D_j, T_j)\}} \qquad \text{eq. (4), sporadic\_task\_set}$$

$\mathtt{P_i} \neq \emptyset$ as otherwise there are not any tasks to schedule.

8.5 the sporadic task model

The sporadic task model is a small-but-important extension to Liu and Layland's *periodic* model in which each task has only two parameters: WCET, and period. [14, pp. 46-61] The sporadic model is different from the periodic model because it decouples the deadline from the period [16] whereas the period remains as an upper bound for the deadline and also determines the frequency of task instance releases.

The sporadic task model of this system does not differ from the sporadic task model commonly used in real-time scheduling:

$$\mathtt{t_j = (C_j, D_j, T_j)} \qquad \text{eq. (5), sporadic\_task}$$

8.5.1 task parameters

- $\mathtt{C_j}$ is the WCET.
- $\mathtt{D_j}$ is the deadline, relative to the arrival time.
- $\mathtt{T_j}$ is the minimal inter-release time: i.e., there can be at most one instance of $\mathtt{t_j}$ for each $\mathtt{T_j}$.
- $\mathtt{0 < C_j} \leq \mathtt{D_j} \leq \mathtt{T_j}$

8.5.2 task properties

- the tasks do not share any task-specific resources
- there are not any task-precedence constraints
- there is not any communication between tasks

9 Schedulability method: response time

One method to ensure schedulability is to assume the worst-case scenario: i.e., that the best-effort core software produces a maximum number of memory accesses, at all times, and at the *worst* times with respect to the critical tasks completing before their deadlines – while, simultaneously, critical tasks are released in the most unfavorable way – the way that causes maximum queuing.

9.1 $\mathtt{r_j}$, the worst-case response-time for $\mathtt{t_j}$

$\mathtt{r_j}$ is the worst-case response-time for the task $\mathtt{t_j}$: it is the maximum-length time span from the arrival of $\mathtt{t_j}$, to its completion.

The following terms add to the delay of $\mathtt{t_j}$. The sum of those, plus $\mathtt{C_j}$, the WCET of $\mathtt{t_j}$, is equal to the worst-case response time:

$$\mathtt{r_j = Q_j + o_j + i_j + C_j} \qquad \texttt{eq. (6), response\_time}$$

9.1.1 $\mathtt{Q_j}$, queuing

$\mathtt{Q_j}$ is the worst-case time spent queuing for $\mathtt{t_j}$. When a task is queuing, it is in the `READY` state. The definition of queuing is whenever $\mathtt{t_j}$ cannot execute, while some other task not only can, but is. (As a consequence, it is not considered queuing when *all* tasks are inactive, as is the case during a global scheduling interrupt.) There are two (or three) situations when $\mathtt{t_j}$ can be queued, which makes up for a total queuing delay formula of:

$$\mathtt{Q_j = Q_j^g + Q_j^s} \qquad \texttt{eq. (7), queuing}$$

9.1.2 z, the response time in global periods

The worst-case (largest) number of global periods that can transpire during the existence of a task is, without rounding:

$$z = \frac{r_j}{p} \qquad \text{eq. (8), periods}$$

9.1.3 Q^g_j, group deadline queuing

Here, t_j is queuing because it has a less immediate deadline than another task in the same group – a task that belongs to the same scheduler as t_j.

Note that just about any task can have t_j preempted and/or queued at some point: a higher priority is equivalent to a more immediate *absolute* deadline. This is a dynamic property that does not imply $D_i \leq D_j$ for a higher priority task t_i: if t_i was released before t_j, the absolute deadline of t_i may still be more immediate than that of t_j. The set $hp(t_j)$ is all tasks that at some point can have a higher priority than t_j.

The worst-case system queuing is:

$$Q^g_j = \sum_{\{t_j, t_k\} \subseteq P_i, t_k \in hp(t_j)} \left\lceil \frac{r_j}{T_k} \right\rceil C_k \qquad \text{eq. (9), group\_queuing}$$

Note: $hp(t_j)$ is effectually *all* tasks except for t_j itself! Because of that, this schedulability method is reduced to that of a fixed-priority system, only worse, as here, all tasks would have equal priority. While there exist methods to compute the response time for individual tasks under EDF, those are complicated and typically not worth the effort as long as the purpose is to compute general schedulability. Still, the formulas here are included to illuminate the mechanics of this implementation.

9.1.4 Q^s_j, cross-group (system) deadline queuing

The task t_j of the scheduler s_i can be delayed by a higher-priority task that belongs to another scheduler. In the worst case, *all* other schedulers will spend *all* of their budgets before $\mathtt{t_j}$ can execute:

$$\mathtt{Q^s_j = (\lceil z \rceil + 1)(p - b_i)} \qquad \text{eq. (10), system\_queuing}$$

9.1.5 depletion queuing

During depletion queuing, $\mathtt{t_j}$ cannot execute because its scheduler $\mathtt{s_i}$ has depleted its CPU-budget. Fortunately, the worst-case time $\mathtt{t_j}$ can be delayed for this reason is already included in $\mathtt{Q^s_j}$.

9.1.6 $\mathtt{o_j}$, the scheduling overhead

The worst-case time required to do a single execution of global scheduling is denoted `o` (just `o`, without an index). It is the the scheduling overhead.

As the top-scheduler controls all of the task-schedulers, `o` includes task-scheduling overhead as well.

$\mathtt{o_j}$ is the worst-case scheduling overhead during a single execution of any one instance of $\mathtt{t_j}$.

Because scheduling occurs once every global tick (`r`), the worst-case delay is:

$$\mathtt{o_j = \left\lfloor \frac{r_j}{r} \right\rfloor o} \qquad \text{eq. (11), schedulability\_overhead}$$

9.2 $\mathtt{i_j}$, interference from the best-effort core

Interference from the best-effort core is due to best-effort core software contention for the shared DRAM and the shared memory bus.

For each period `p`, with respect to the critical tasks of scheduler $\mathtt{s_i}$, the worst-case number of memory accesses that can arise from the best-effort core is bounded by $\mathtt{m_i}$.

The worst-case interference for any task is a situation in which the best-effort core makes *all* its allowed memory accesses during the employment of that task, for *all* global periods during which that task instance exists.

The assumption is that during a best-effort memory fetch, the critical core task cannot do anything, but idles. (This assumption is probably very pessimistic.)

If the worst-case response time for a DRAM access (for either core) is 58.5 nanoseconds [3], then the worst-case memory interference from the best-effort core is:

$$\mathtt{i_j = \lceil z \rceil m_i 58.5\ ns} \qquad \text{eq. (12), interference}$$

10 Schedulability method: resource server

Another method to guarantee schedulability is to consider an individual task-scheduler a *resource server*. The resource provided is CPU-time; it is consumed by the tasks that belong to that particular scheduler.

This method involves less advanced computation than the response-time method, because here, there is not any reliance on the dynamic set of tasks with higher priorities than the task whose response time is calculated. (Another implementation of this idea is described in [18, pp. 301-324]. There, only one processor is considered, and the notation is different.)

This method relies on two functions: a *supply* bound function, `sbf`, and a *demand* bound function, `dbf`.

Both functions have two parameters. $\Delta\theta$ is the same for both: it is the length of the time interval for which supply or demand is computed. This interval can occur anywhere in time, and during that interval the system executing can be in any sound state.

The other parameter, for `sbf` and `dbf`, are the supplier and demander themselves, respectively. If the functions are used to compare supply and demand, they must be used pair-wise: if $\mathtt{s}_\mathtt{i}$ is the argument to `sbf`, $\mathtt{P}_\mathtt{i}$ is the argument to `dbf`. Indeed, the tasks of $\mathtt{P}_\mathtt{i}$ are the only tasks that can demand CPU-time supplied by $\mathtt{s}_\mathtt{i}$.

$\mathtt{sbf}(\Delta\theta, \mathtt{s}_\mathtt{i})$ is the *minimum* possible CPU-time supplied by $\mathtt{s}_\mathtt{i}$ during any time period of length $\Delta\theta$.

Correspondingly, $\mathtt{sbf}(\Delta\theta, \mathtt{s}_\mathtt{i})$ is the *maximum* possible CPU-time demanded.

The worst-case scenario is: the maximum possibly demanded vs. the minimum possibly supplied.

For any task-scheduler, schedulability is guaranteed if the maximum demand for any particular interval is below or equal to the minimum supply for all interval of the same length.

The system is schedulable if all task-schedulers are.

10.1 interval length

If the interval starts at θ_1 and ends at θ_2 the length of the interval is:

$$\mathtt{I} = \Delta\theta = \theta_2 - \theta_1$$

eq. (13), interval

10.2 sbf

During each complete global period p, the task-scheduler $\mathtt{s_i}$ supplies exactly $\mathtt{b_i}$ of CPU-time for its tasks.

The number of complete periods is:

$$\mathtt{w} = \left\lfloor \frac{\mathtt{I}}{\mathtt{p}} \right\rfloor - 1$$

eq. (14), supply_complete_periods

The CPU-time supplied by $\mathtt{s_i}$ for those periods is:

$$\mathtt{B_i} = \mathtt{wb_i}$$

eq. (15), supply_time_complete_periods

Moreover, there can be zero, one or two incomplete periods that are part of the interval. If so, they exist before and/or after the consecutive w complete periods.

The incomplete periods have the total length:

$$\mathtt{R} = \mathtt{I} - \mathtt{wp}$$

eq. (16), supply_length_incomplete_periods

The worst case, when the task-scheduler supplies the least, is when there are two incomplete periods of the same length. Then, it is the most likely that the task-scheduler supplies all of its CPU-budgets, both times, during the stretches of the incomplete periods that are outside the interval.

Both time spans that belong to incomplete periods outside of the interval has length:

$$\mathtt{U} = \mathtt{p} - \frac{\mathtt{R}}{2} \qquad \text{eq. (17), supply\_length\_incomplete\_outside}$$

If $\mathtt{U}$ is longer than $\mathtt{b_i}$, the minimum supply case for $\mathtt{s_i}$ is when all of $\mathtt{b_i}$ is supplied during $\mathtt{U}$. However, if $\mathtt{b_i}$ exceeds $\mathtt{U}$, the part of $\mathtt{b_i}$ that exceeds $\mathtt{U}$ is supplied:

$$\beta_{\mathtt{i}} = 2\mathtt{max}(0, \mathtt{b_i} - \mathtt{U}) \qquad \text{eq. (18), supply\_incomplete\_periods}$$

However, because of the best-effort core interference, the actual CPU-time supplied is less.

As seen, there are $\mathtt{w}$ complete periods in each interval. In addition, there can be two more periods that are incomplete: one at each side of the consecutive, complete periods. In all, there can be a maximum of $(\mathtt{w} + 2)$ periods. The task-scheduler $\mathtt{s_i}$ allows $\mathtt{m_i}$ best-effort core LLC misses for each period, and for the entire interval: $(\mathtt{w}+2)\mathtt{m_i}$. If the data and assumptions in [section 9.2, page 48] are reused, the total supply reduction is:

$$\rho = (\mathtt{w} + 2)\mathtt{m_i}58.5\ \mathtt{ns} \qquad \text{eq. (19), supply\_reduction}$$

All in all, the minimum supply from $\mathtt{s_i}$ to all tasks $\mathtt{t} \in \mathtt{P_i}$ is:

$$\mathtt{sbf}(\mathtt{s_i}, \mathtt{I}) = \mathtt{B_i} + \beta_{\mathtt{i}} - \rho \qquad \text{eq. (20), supply}$$

10.3 dbf

The number of complete periods of $\mathtt{t_j}$ that exists within the interval is:

$$\mathtt{f} = \left\lfloor \frac{\mathtt{I}}{\mathtt{T_j}} \right\rfloor \qquad \text{eq. (21), demand\_complete\_periods}$$

Once the complete periods have been accounted for, it is possible that an incomplete period remains, during which $\mathtt{t_j}$ can demand CPU-time. However, it cannot request more than either the computation time of the task, or the entire incomplete period:

$$\mathtt{h} = \mathtt{min(I - fT_j, C_j)} \qquad \text{eq. (22), demand\_incomplete\_period}$$

Note that this is the opposite to the supply function. There, what is desired is the minimum supply possible, so the spill-over global period is divided into two. Here, the aim is maximum demand so it is assumed the spill-over task period is not spread over two periods.

The maximum demand made by the task is:

$$\mathtt{dbf(t_j, I)} = \mathtt{fC_j} + \mathtt{h} \qquad \text{eq. (23), demand\_task}$$

The maximum demand made by $\mathtt{P_i}$ is the sum of the demands made by all $\mathtt{t} \in \mathtt{P_i}$:

$$\mathtt{dbf(P_i, I)} = \sum_{\mathtt{t_j} \in \mathtt{P_i}} \mathtt{dbf(t_j, I)} \qquad \text{eq. (24), demand\_group}$$

11 Experiments

This project carries out several experiments. But there can be many other experiments as well, that are not necessarily less interesting. Because of a building-block approach, it is quick to set up and execute an experiment. Even most data is collected automatically. Rather, the challenge is to grasp the meaning of certain combinations, as well as to correctly analyze the results.

11.1 building blocks

All experiments involve the same technology and the same tools, but they are employed in different combinations, and with different parameter values.

The building blocks are:

- `hs` is the hierarchical scheduler. It is specifically developed for this project. `hs` can be invoked in different ways. Apart from executing task systems, `hs` can also output the global tick trace, which can be included in an experiment, if desired.

- `taskset(1)` is used to isolate a process to a specific hardware CPU, or core. How to isolate a core from most other processes is described in [section 5.4.2, page 31].

- `stress(1)` is used to generate memory traffic. It can be used on any and all cores. For most experiments, `stress` is used to simulate interfering software: on the best-effort core, `stress` is used to simulate arbitrary non-critical software.

  On the critical core, `stress` is used simulate processor and memory usage of the software that would be executed by the tasks that are scheduled by `hs`. Unless `hs` is told to fork processes, the tasks that `hs` schedules do not utilize computer resources to the extent that real software would.

  Note: While `stress` can be made to behave virtually like any piece of software or software set – and this makes it useful in simulations – it is just as easy to run ordinary Linux software, on either core, to attain natural contention for computer resources. Simulation may be useful to create certain scenarios, that would otherwise require extensive setups of ordinary software, but most scenarios are just as easily attained without simulation. What to do depends on the nature of the experient: `stress` can be used to show that something actually can happen, under rare-but-possible conditions. On the other hand, ordinary software can be used to produce a realistic setting, say when the aim of the experiment is to illustrate every-day system use.

Warning: `stress` does not behave like a normal piece of software. Instead, `stress` can be made to congest the entire memory, at what point the computer is nonresponsive. After all memory has been congested, it is not enough to kill the `stress` processes to remedy the situation. While the memory is thus released, any running application still needs to reassert its footprint before it can resume proper work and respond to user IO. This usually takes a couple of seconds for the application immediately grasped for – but for dormant processes, the effect can remain hidden, and appear unexpectedly. This makes `stress` even less reliable to use in experiments as the memory state of one experiment instance can linger on to affect other, where perhaps `stress` is not even used.

- `perf_event_open` is used to measure the memory usage. It does so by booking LLC misses, as they imply DRAM fetches and memory-bus traffic. `perf_event_open` can be used on a single process, on a set of processes, or on any and all cores, collectively. If need be, there can be parallel instances with different focus. Here, `perf_event_open` is used by `hs` so that at the end of its execution it outputs the total number of memory accesses that stem from the best-effort core.

- `perf_event_open` is also the way that `hs` monitors the best-effort core, to decide weather the software that runs there is supposed to be freezed, thawed, or left alone. If `hs` decides to change the status of the best-effort core, this is effectuated by means of a Linux `cgroup`: such a group must be initialized before `hs` is run.

- To measure the execution time of a process, `time(1)` is used. This tool can be used to measure performance downgrade by clocking the completion time of identical commands, only in different setups.

There is some material appended to this document – [section D, page 135] – to facilitate interaction with not only `hs` but the associated tools and entities (including the best-effort core) as well as the setup and execution of experiments.

11.2 design

An experiment can be anything that employs `hs` with a task system and any combination of the mentioned tools. This checklist how to arrange and execute an experiment should cover most cases:

1. the setup: what happens on the best-effort core, and what happens on the critical core (except for `hs`), while the experiment runs
2. the `hs` task system
3. a time-interval how long the experiment will run – recall that a `hs` system has a lifetime parameter, which can fill this purpose
4. what processes are investigated with `time` and `perf_event_open`
5. the experiment should also have a verbalized rationale – “we do this because...” (implementation details of an experiment can contradict the purpose of the experiment: if deemed acceptable still, the problem should be mentioned as well as what is exciting about the experiment)
6. results (after the experiment is done)
7. a possible post-processing of the results by some number-crunching tool
8. conclusion

11.3 zsh wrappers

To manage the experiments, and make sure they stay the same for different input values (e.g., task systems), zsh functions are used to wrap the specific commands that make up the experiments.

This method is one of clearly defined functions which are reproducible, highly automatized, and easily re-executed after say a change to the involved software.

There is one function for each experiment. Such a function can, for example, execute several input task systems, while booking their results, and later comparing them.

The experiment functions: [section D.c, page 140]

12 Contention experiment

zsh function: `run-contention-experiment` [section D.c, page 140]

This experiment attempts to show that processing, even though separated on different cores, still affects, and is affected by, simultaneous processing.

12.1 setup

The experiment consists of compiling some 50 Elisp files on the critical core, in order to measure how long it takes to complete compilation for different setups. The changes involve the critical core as well as the best-effort core.

In parallel with compilation, `hs` runs, likewise exclusively on the critical core. The purpose of the compilation is to simulate the memory accesses and the CPU strain that the task software of `hs` would produce, if indeed they carried real software and not mock-offs.

On the best-effort core, for certain setups of this experiment, `mplayer2(1)` runs: whenever so, it displays a three gigabyte mp4 file. To display such a file on the experiment computer is deemed expensive, but not excessively or unrealistically so.

In addition, `stress(1)` sometimes run on the critical core to simulate the resource demand and consumption of would-be critical software.

The results are, in nanoseconds:

```
* (compilation time without hs)
29664764093
* with hs:
24764116386
* with hs and the best-effort media player:
24709304113
* with hs and stress:
50052877386
* with hs, stress and the BE media player:
57734584696
```

12.2 conclusion

Intuitively, the more contention, the more time is needed to complete the compilation. Contrary to this, in this experiment, neither `hs` nor the media player influence the speed of the parallel compilation process.

13 Best-effort core memory experiment

zsh function: `run-memory-experiment` [section D.c, page 140]

The contention experiment shows that the critical core is affected by best-effort core activity: [section 12, page 56] Now, the intention is to show that influence goes both ways: in particular, how the memory budgets of task systems affect the best-effort core whenever `hs` is instructed to, whenever appropriate, either freeze or thaw it.

The influence is quantified as the number of memory accesses that originates from the best-effort core during executions of task systems with different memory budgets. The best-effort core executes the media player.

Here, `hs` is invoked with `--memory-budget-add`, which add its argument to all memory budgets. Thus, the same system is is executed repeatedly, only with different memory-budgets. At termination, the number of best-effort core memory accesses are output.

13.1 the importance of this experiment: delays and overheads

This project describes a problem, and looking at the big picture, the implementation attempts to solve it by controlling the number of best-effort core memory accesses. The degree of success is possible to quantify by executing this very experiment. Likewise, later on, the experiment in [section 16, page 74] will quantify the "small picture" success, which directly relates to the real-time deadlines of the involved tasks.

In this experiment, the interpretation of the result is: if `x` best-effort core memory accesses are allowed for an execution of `hs`, and `y` are those actually performed, then the difference `y - x` should be as close to zero as possible (and not less than zero, for a busy system).

A perfect execution, where `x = y`, is unrealistic for several reasons, some of which appear at every global tick:

- the `hs` C++ scheduling and freeze/thaw overhead
- the `hs` inexact tick, discussed in [section 5.2, page 25]
- the PMC data fetch overhead

- the polled EDF, discussed in [section 4.3, page 22], and otherwise the userspace implementation, notably the signal control of forked processes

In addition, some unpredictable factors appear at most once for every task scheduler and global period:

- cgroup freezing and thawing overhead (an experiment indicates that this overhead is all but nonexistent)

While all those factors do reduce the quality of the outcome, their respective weights are at this point uncertain, as is weather they actually do influence enough to qualify as scapegoats, should the experiment outcome be unsatisfactory.

13.2 systems

The experiment includes two systems. One has a single task scheduler; the other has two. The reason to do the experiment twice is to see if the fallout is responsive to setup details, in what case the causality should be further explored. Both systems have full budgets, with implicit deadlines ($\mathtt{D_i = T_i}$) for all tasks, and less than full utilization:

$$\mathtt{U_T = \sum_{t_i \in T} \frac{C_i}{T_i} \leq 1}$$

The experiment results are showed below in two figures; also, in [section A, page 82] are the complete tables of data from which the figures are generated.

13.2.1 one-scheduler system: base-faculty-1

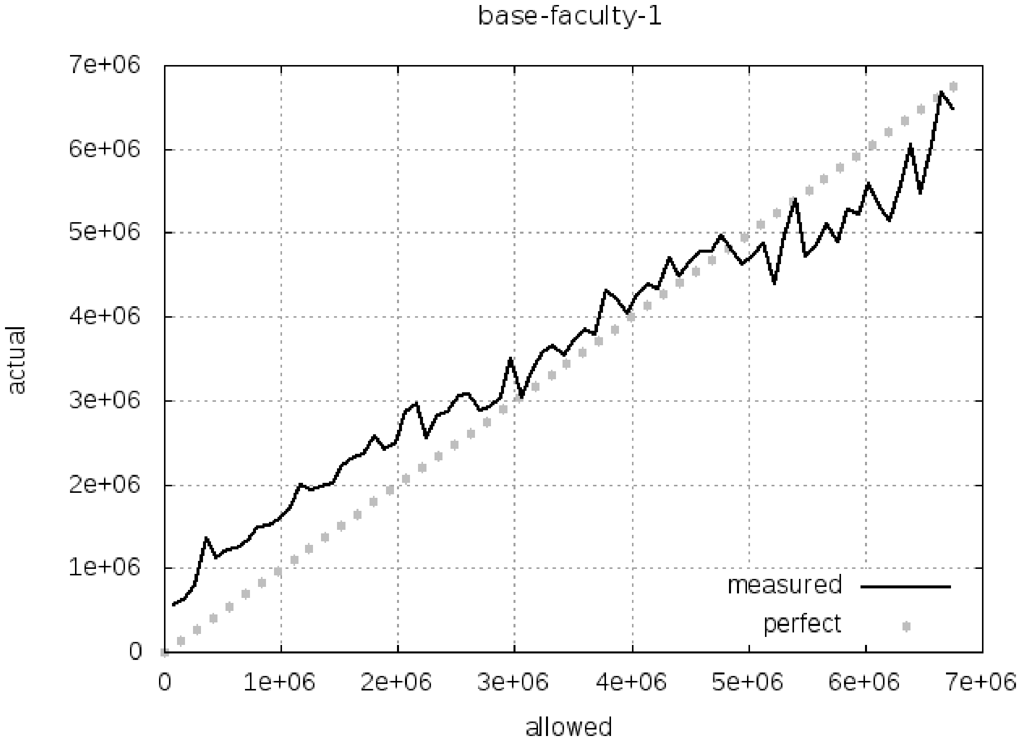
System:

```
Global scheduling rate: 1
Global period : 120
Global lifetime: 1000
Global scheduling algorithm: EDF

Critical level: 1
Budget: 120
```

```
Max BE accesses: 1
Task scheduling algorithm: EDF
t1 = (40, 120, 120) faculty(7)
t2 = (40, 120, 120) faculty(7)
```

Fallout:



13.2.2 two-scheduler system: `base-faculty-2`

System:

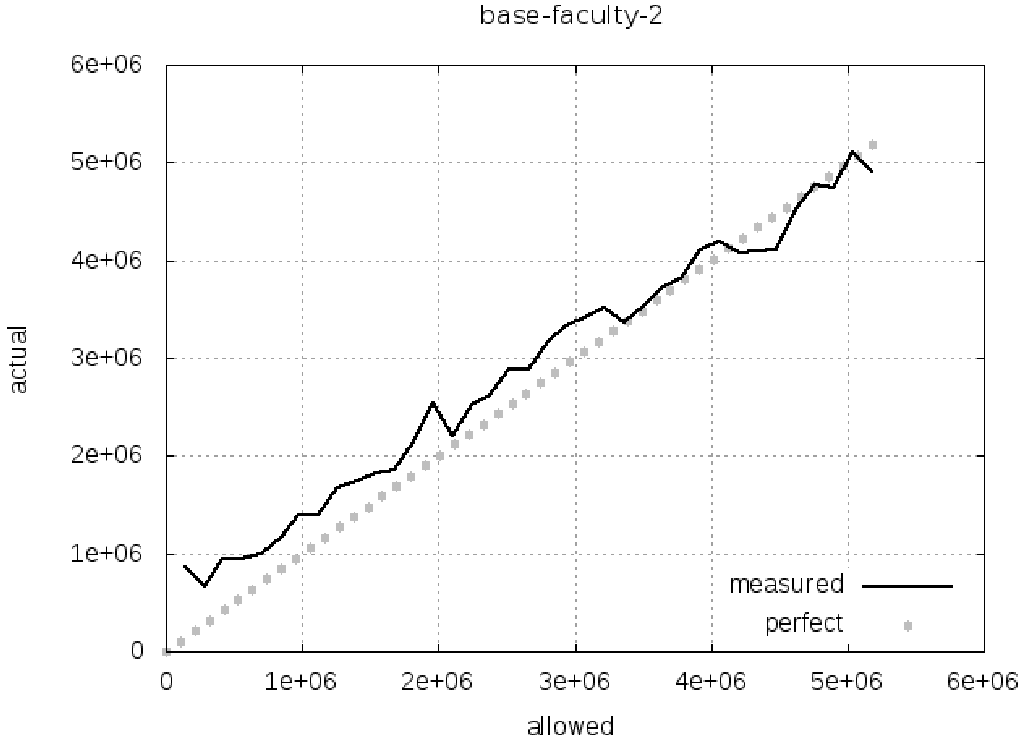
```
Global scheduling rate: 1
Global period : 150
Global lifetime: 1000
Global scheduling algorithm: EDF

Critical level: 1
Budget: 60
Max BE accesses: 10
Task scheduling algorithm: EDF
```

```
t1 = (60, 150, 150) faculty(7)

Critical level: 2
Budget: 60
Max BE accesses: 1
Task scheduling algorithm: EDF
t2 = (60, 150, 150) faculty(7)
```

Fallout:



13.3 conclusion

The correspondence between memory budgets and best-effort core memory accesses should be: the bigger the budgets, the more accesses – up and until where budgets are not depleted because the best-effort core software does not generate that much memory traffic. (What is beyond that point is not included in the above plots.) The linear tendency in both plots shows that in general, memory throttling works as expected. However, the high error percentages are discouraging. In a real-time system, the specific, worst case has to be considered, not the general tendency.

14 hs task systems experiment

zsh function: run-task-system-experiment [section D.c, page 140]

All instances of this experiment are of the same form. Only one input parameter differs: the hs task system. Everything else is the same to facilitate comparisons of the fallout data: if the setup does not change, different results are a consequence of different task systems.

14.1 setup

On the critical core, hs runs, as well as the Elisp compilation, which is also timed. On the best-effort core, the media player runs

What is different from the first experiment is the behavior of hs: it will now freeze, and thaw, the best-effort core. What happens is a function of how many memory fetches the best-effort core produces, as well as the dynamic state of the hs task system.

The only way to affect the results is thus to change the parameter values of the task system – and those of the scheduler itself (those parameters are in the same task system file).

The method is to execute several task systems, with widely different parameter values at key points, and then examine if and how that affects the outcome. To this end, each system is given a personality name which describes what property is emphasized: e.g., one system is called short_rate, another small_memory_budgets, and so on. There can be many such personalities and combinations thereof: only a small subset is tested here.

Because there are so many parameters, it is a challenge to assign neutral values that will not affect the outcome and thus neither morph or hide the part of the outcome that is a consequence of the spelled-out personality. To circumvent this problem, the systems have identical values except for their defining traits.

14.2 what is a task system?

A task system contains parameter values for each task, according to the sporadic task model: for each task there is a WCET, a deadline, and a minimum-interarrival time, or period, as described in [section 8.5, page 43]. But there is more than tasks to a task system. The system definition also specifies the parameters values for each task *scheduler*, or critical group: e.g., the memory budget that `hs` uses to freeze and thaw the best-effort core. Also, a system definition contains parameter values of the top-most scheduler: e.g., the scheduling rate and the total system lifetime. So technically, a more exact phrasing would be a multicore task/scheduler system. Here, "task system" is used for simplicity.

An example task system can be found in [section D.a, page 135].

14.3 simulation issues

This experiment is the first step toward an experiment that is more realistic, and less simulated. Although there is still heavy reliance on simulation, simulation is implemented as well-defined puzzle pieces with clear purposes. The thought is that they can be easily replaced, one by one, further down the line.

The issues are:

- `hs` does real scheduling, only what it schedules is not real software in the sense that it actually does anything useful (it *can* be hard-coded to do useful things, but as for now it is not).
- Although the compilation is an example of real, useful software in execution, it actually simulates all the instances of would-be critical-task software.
- The best-effort interference is measured in terms of how much it can delay the critical-core Elisp compilation, even though `hs` is in control of the best-effort software. That is, interference is not measured in terms of the real-time parameters – the deadlines – that belongs to the `hs` tasks.

14.4 the result data

After the completion of an instance of this experiment, there are four newly created files which contain all parts of the result. Those files are in a directory that bears the same name as the executed task system.

The result is organized like this:

file name	description
hs_output.log	The hs run: all the task state changes, as well as any task software output. It is the execution trace of the task system as scheduled, and executed, by hs. Also, this file contains the number of best-effort core memory accesses during the system run.
lisp.log	The time required to compile the Elisp source files.
tick_times.log	The time readings in nanoseconds for every global tick. Details and examples are in [section 5.2, page 25].
stats.log	This file contains the offset from the specified global tick rate, for every tick. It is computed of the tick_times.log file; likewise, it is discussed elsewhere.

There are also the files stats.txt and BE.txt that are convenience parsings of data already available in the above files.

14.5 systems and results

The first system is base. It is a basic system that the other systems model, save for some characteristic trait changed. The purpose is to examine how any step away from base makes – or does not make – its way into the result data, and then analyze how this comes to happen.

14.5.1 base

- The system:

```
Global scheduling rate: 1
Global period : 10
Global lifetime: 20000
Global scheduling algorithm: EDF
```

```
Critical level: 1
Budget: 5
Max BE accesses: 1000
Task scheduling algorithm: EDF
t1 = (5, 10, 10) helloworld()

Critical level: 2
Budget: 5
Max BE accesses: 7050
Task scheduling algorithm: EDF
t2 = (5, 10, 10) helloworld()
```

- Lisp compilation time, in nanoseconds:

```
25395474282
```

- Tick stats:

```
readings: 19999
mean: 2.000000
variance: 23467123592.000000
standard deviation: 153189.828618
min: -987988
max: 9296281
```

- Best-effort core memory accesses:

```
41569979
```

14.5.2 long-ticks

The long-ticks system has a much slower global scheduling rate than the base system. This affects presumably all aspects of any system execution because it is only at the scheduling interrupts that one task can be replaced by another as the one executed by the CPU; moreover, it is only at the scheduling interrupts that hs freezes or thaws the best-effort core.

- The system:

```
Global scheduling rate: 5
Global period : 10
Global lifetime: 20000
Global scheduling algorithm: EDF
```

```
Critical level: 1
Budget: 5
Max BE accesses: 1000
Task scheduling algorithm: EDF
t1 = (5, 10, 10) helloworld()

Critical level: 2
Budget: 5
Max BE accesses: 7050
Task scheduling algorithm: EDF
t2 = (5, 10, 10) helloworld()
```

- Lisp compilation time, in nanoseconds:

  ```
   24976418787
  ```

- Tick stats:

  ```
  readings: 3999
  mean: 14.000000
  variance: 599515787400.000000
  standard deviation: 774284.048267
  min: -4968152
  max: 8919367
  ```

- Best-effort core memory accesses:

  ```
  48739762
  ```

14.5.3 `long-period`

The `long-period` system has a much longer global period than the `base` system.

- The system:

  ```
  Global scheduling rate: 1
  Global period : 30
  Global lifetime: 20000
  Global scheduling algorithm: EDF

  Critical level: 1
  ```

```
Budget: 5
Max BE accesses: 1000
Task scheduling algorithm: EDF
t1 = (5, 10, 10) helloworld()

Critical level: 2
Budget: 5
Max BE accesses: 7050
Task scheduling algorithm: EDF
t2 = (5, 10, 10) helloworld()
```

- Lisp compilation time, in nanoseconds:

```
 20692567908
```

- Tick stats:

```
readings: 19999
mean: 2.000000
variance: 40455151873.000000
standard deviation: 201134.661043
min: -988825
max: 5962338
```

- Best-effort core LLC misses:

```
56681084
```

14.6 conclusion

The different outcomes of this experiment show that interference from the best-effort core can be throttled as a function of the memory access profile of the best-effort core software in combination with what happens on the critical core, down to the individual task that is currently executed.

15 Linux processes experiment

zsh function: `run-linux-process-experiment` [section D.c, page 140]

This experiment is similar to the previous one – [section 14, page 63] – with the huge exception that `hs` does not run mock software anymore, instead, it spawns and schedules ordinary Linux processes.

The purpose of the experiment is to show that real software is as much susceptible to interference as the software measured in the previous simulation.

At best, the previously used hard-coded task software provides a simulation of an actual system. Here, that is replaced by Linux processes. The processes are scheduled exactly like the task software, only – were it fits – the processes are suspended and resumed from `hs`, by means of `kill(2)` signals. Those are sent on the basis of process group IDs (PGIDs), so that processes spawned down the ladder will be controlled by `hs` as well.

15.1 advantages

- Because real software is being executed it puts a definite end to everything simulated or artificial about this project.
- Because the processes are now separated from `hs`, they can be monitored individually, isolated from each other as well as from `hs`. Recall that in the previous experiment, there was a compilation process that run in parallel with `hs`. The slowdown in compilation due to interference from the best-effort core was a measure of how, collectively, critical-core software was affected. Here, with real processes, the processes are not representing anything but themselves, so any slowdown detected (for whatever reason) is actual slowdown and no mere indication of such a tendency.
- Because the processes are ordinary Linux processes, when they are resumed (after being suspended), they take on where they were preempted. That means total execution wall-clock time will not only be prolonged by the the best-effort core interference, it will to a much higher degree depend on the associated real-time parameters of the software, because those determine how `hs` will schedule the processes.

15.2 disadvantages

- If a system employs lengthy background processes, but still is desired to be highly responsive with frequent context switches, the task parameters of the sporadic task model must be reinterpreted. The `C` parameter for WCET, as well as the other parameters, no longer express times for an entire process, but rather the burst patterns in which it will execute until completion. It can be thought of as a version of the multiframe model [17, pp. 635-645]; however, it does not come with fixed frames in terms of exactly what part of the software is being executed. (Actually, a hard-framed model would be easy to implement with the hard-coded task software: a static C++ variable – as in persistent, not class – to hold the current frame, and then a branch is all it takes.)
- Although slowdown is measured with actual software – not some representation thereof – slowdown is not measured in terms of the real-time parameters associated with the software. This further contrasts with the sporadic task model, in which everything is fine as long as all tasks complete before their deadlines. Even if deadlines are reinterpreted as bounds of bursts of processes in execution, this experiment still does not examine how the context in which processes execute either fails to interfere enough, or actually breaks task deadlines.

15.3 systems and results

15.3.1 `base-p`

The `base-p` system consists of real Linux processes. One is an infinite loop outputting a string; the other compiles the by now familiar Elisp files. However, compilation in this experiment is not equivalent to what happened in the previous experiment, with the `base` system – then, there was a compilation process that ran interwovenly with `hs` on the critical core; here, compilation is critical software that `hs` schedules. Thus, the compilation process is subject to its sporadic task representation and the values it holds. The by far biggest consequence of this is that the result data cannot be interpreted in terms of interference alone, although interference has not decreased. Rather, a much bigger factor is competition on `hs` from other critical tasks, and the task parameter values of the process itself.

The result for `base-p`:

- The system:

  ```
  Global scheduling rate: 1
  Global period : 30
  Global lifetime: 30000
  Global scheduling algorithm: EDF

  Critical level: 1
  Budget: 4
  Max BE accesses: 10000
  Task scheduling algorithm: EDF
  t1 = (1, 4, 4) ./compile_lisp()

  Critical level: 2
  Budget: 14
  Max BE accesses: 7050
  Task scheduling algorithm: EDF
  t2 = (2, 4, 4) ./forever()
  ```

- Lisp compilation time, in nanoseconds:

  ```
  80020028235
  ```

- Tick stats:

  ```
  readings: 29999
  mean: 2.000000
  variance: 8287884602205.000000
  standard deviation: 2878868.632329
  min: -987429
  max: 43918558
  ```

- Best-effort core LLC misses:

  ```
  60420
  ```

15.3.2 `base-p-no-memory-budget`

These are the results for an identical system, besides having a virtually nonexistent memory budget.

- The system:

  ```
  Global scheduling rate: 1
  ```

```
Global period : 30
Global lifetime: 30000
Global scheduling algorithm: EDF

Critical level: 1
Budget: 4
Max BE accesses: 1
Task scheduling algorithm: EDF
t1 = (1, 4, 4) ./compile_lisp()

Critical level: 2
Budget: 14
Max BE accesses: 1
Task scheduling algorithm: EDF
t2 = (2, 4, 4) ./forever()
```

- Lisp compilation time, in nanoseconds:

```
43186420057
```

- Tick stats:

```
readings: 29969
mean: 2.000000
variance: 9936160858837.000000
standard deviation: 3152167.644469
min: -987429
max: 44845212
```

- Best-effort core LLC misses:

```
30284774
```

15.4 conclusion

While the results of this experiment do enlighten the inefficiency of the method, equally revealing was the process of setting up the experiment.

The sporadic task model is not suited for continuous processes. In order to have an interactive system, the computation times (the WCET parameter values) must be small. On the other hand, the deadlines cannot be too tight less you risk having delayed processes (or delayed parts of processes). Yet, on the third hand, if deadlines are wide, so are periods, which means the processes will not get back to action right away but must idle until the period ends. This is sometimes acceptable, even welcome, as long as other processes can execute to fill the gap; if not, it is a waste of resources.

Also, remember that the task model parameters are not alone in defining a system: there are also the global scheduler parameters, and those of every task schedulers. All in all, it is a jungle just to execute a program and have it complete within a reasonable amount of time.

For this reason, there are only two systems included in this experiment, to show the mechanics.

Of course, there is no ruling out that a system implemented this way could be functional, but that would have to involve experimentation and evaluation as to what to assign each end every parameter. And that is not a good idea: the parameters should reflect reality, or at the least intuition, and pretty much instantly so – they are not to be tweaked just so a simple program can be executed.

16 Real real-time: audio experiment

zsh function: `run-audio-experiment` [section D.c, page 140]

This experiments puts it all together. Just like the previous experiment, it uses real software: `hs` schedules Linux processes.

But unlike the previous experiment, all tasks are expected to terminate within their deadline frames, as expressed by their sporadic task representations.

This time, nothing is measured. Instead, everything is done in terms of the real-time task system, and the associated software.

Furthermore, the process that is examined executes software that measures an external, physical property: the sound level. This is measured with a microphone plugged into the computer. (To make the system even more interesting, it can be augmented with all sorts of gadgets: an USB thermometer, for starters.)

The audio process executes a script that logs the sound level by appending it to a file:

```
#! /bin/zsh

# use once:
#
#     a-level t
#
# continous: invoke with signal USR1;
#            terminate with C-c C-c
#
#     a-level
#     kill -s USR1 PID

# should be faster than 300 ms
do-a-level () {
    (($#)) || (($+AUDIO_OUTPUT)) || set /dev/stdout
    rec --null stat trim 0 0.1 2>&1 | \
        grep 'Maximum amplitude'    | \
        cut -d' ' -f 7 >> $1
}
```

```
trap 'do-a-level' USR1

(($+1)) && do-a-level || while ((1)) {}
```

As seen, that script is not optimized but uses a combination of commonly-used utilities.

The script is a server: it idles until it receives an `USR1` signal, which triggers the function that reads and stores the sound level. This scheme makes it unnecessary to create a new process for every instance released.

The rationale of this experiment is that if realistic real-time parameter values are assigned the sporadic task representation of the process that executes the audio script, then this system can be asserted not in terms of increased memory/CPU contention, but instead: failure is whenever the absolute deadline of the audio task is in the past, while the task still has not booked the audio level.

If such a failure is deducted to be the result of best-effort core memory contention, it will be attempted to nullify that influence in an execution run where `hs` throttles the best-effort core. If this changes the outcome, then it has been showed that this project assesses a real problem that it is able to solve as well.

16.1 system

The strategy is to create a system that for a virtually unloaded computer will execute successfully: it will not suffer a single deadline miss. However, the margin of error should be the slimmest possible.

That way, if the same system thereafter executes in parallel with best-effort core activity, the scale will tip and interference will be detectable in the form of missed deadlines.

The third step is to execute the system anew, again along with interference, only this time with throttling enabled, and examine if that counteracts the interference, and has the system execute successfully once again, just as if the interference had not been there.

The system is:

```
Global scheduling rate: 200
Global period : 800
Global lifetime: 10000
Global scheduling algorithm: EDF

Critical level: 1
Budget: 400
Max BE accesses: 1
Task scheduling algorithm: EDF
t1 = (400, 800, 800) ./a-level()

Critical level: 2
Budget: 400
Max BE accesses: 1
Task scheduling algorithm: EDF
t2 = (400, 800, 800) ./forever()
```

16.2 execution

In order to understand in what way a system is executed and how it is influenced by activity on the best-effort core, a framework of `tmux(1)` panes is used. The screenshot below shows this – only the colors have been inverted, to facilitate reading on paper.

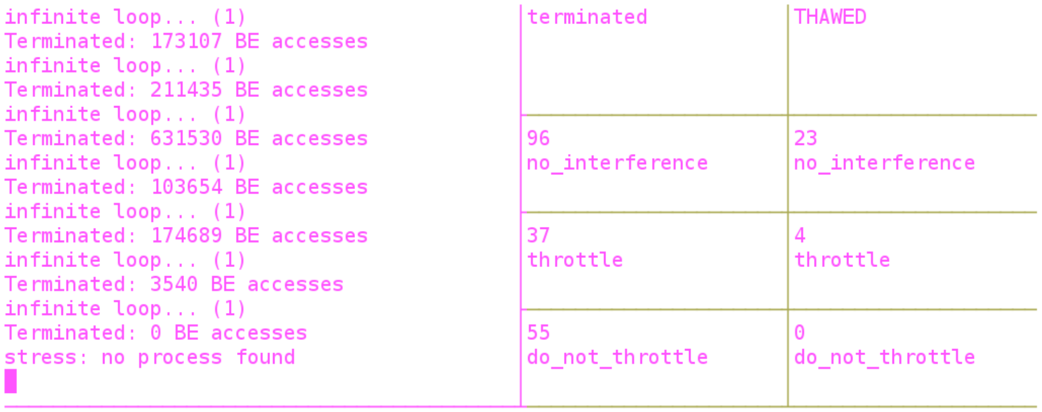
In the comic-book reading-order – left-to-right and top-to-bottom – the panes are:

- The IO of `hs`, and that which `hs` schedules (be it the hard-coded task-software or, as here, Linux processes).
- The best-effort core policy: `no BE` means there is not any activity on the best-effort core; `tamed BE` means there is activity on the best-effort core but `hs` is instructed to throttle it; and, `wild BE` means that there is activity on the best-effort core, but it is never throttled.

  The parenthesis next to the best-effort core policy shows the system execution number. In order not to interpretate uncommon events as recurring, it is beneficial to execute the same system several times, under the same circumstances.
- The next pane shows the state of the best-effort core: `FROZEN`, or `THAWED`. Here, a couple of PIDs are sometimes shown: those are the PIDs of the processes that are frozen, whenever the best-effort core is. There is only room to show three PIDs: if there are more processes on the best-effort core, they are treated the same as those of the PIDs

mentioned.

- The number of signals sent to the audio task during the `no BE` run(s). If there are several iterations this number is the sum of all signals sent during past executions plus the current, so far.
- The number of outputted audio levels during the `no BE` phase, so far.
- Ditto `tamed BE` (two consecutive panes).
- Ditto `wild BE`.
- The processes on the best-effort core: the data shown is core, PID, and command. (`1` is the best-effort core.)



```
infinite loop... (1)
Terminated: 173107 BE accesses
infinite loop... (1)
Terminated: 211435 BE accesses
infinite loop... (1)
Terminated: 631530 BE accesses
infinite loop... (1)
Terminated: 103654 BE accesses
infinite loop... (1)
Terminated: 174689 BE accesses
infinite loop... (1)
Terminated: 3540 BE accesses
infinite loop... (1)
Terminated: 0 BE accesses
stress: no process found
█
```

```
terminated                THAWED

96                        23
no_interference           no_interference

37                        4
throttle                  throttle

55                        0
do_not_throttle           do_not_throttle
```

Note: In the course of carrying out this experiment, it was discovered that the global preemption rate cannot be set too low when scheduling real processes. If so, presumably, the overhead handling processes – interrupting and otherwise controlling them through signals – will hinder actual software execution. All but instantly, that will result in missed deadlines. This situation can be remedied without changing any other parameter value save for the rate, which must be increased.

16.3 ideal fallout

A perfect fallout for this experiment would be:

- For a system that is schedulable, and executed without any best-effort core memory contention, the aim should be set that if there are `s` signals sent, then there should be `o = s` audio level outputs.
- Keeping the same system, but adding contention at the time of execution (contention which nonetheless can be throttled), then if there are $\mathtt{s}'$ signals, there should still be $\mathtt{o}' = \mathtt{s}'$ outputs. The assumption is that the throttling mechanism is configured so it can, if necessary, put the best-effort core entirely out of business.
- Keeping the same system and the same (severe) contention, only this time with throttling disenabled, then the number of outputted audio levels $\mathtt{o}''$ should be considerable smaller than `o` and $\mathtt{o}'$.

16.4 conclusion

This experiment shows that `hs` can control the best-effort so that some critical tasks (though very few) will complete before their deadlines in spite of best-effort interference, which, without the `hs` throttle, would not have been the case.

What is disappointing is the lack of precision. It appears as though `hs` can barely remedy a dysfunctional overall state, while not sophisticated by far to tweak or improve a functional system with any granularity to it.

It is telling that if `stress` is substituted for a piece of normal software – e.g., the media player, which has a small footprint compared to the entire memory space available – there are not not any deadlines misses due to memory contention, or, if occurring, they are never remedied by throttling.

Another thing disappointing is the amount of deadline misses even without any interference. A likely reason is that the audio process has an execution time that varies too much so that often the WCET do not reflect reality.

However, it is clear that this experiment fails not because of shortcomings in `hs` and/or the wider architecture, but already due to the inability to present the critical core with best-effort core interference that in a realistic and sensible way affects the performance of the critical-core software.

Nevertheless, it is even more clear that `hs` is incapable of stopping `stress` from breaking the entire system: because `hs` is itself suspectible to best-effort core interference, the throttling mechanism reverts to a dysfunctional state along with the rest of the system as all memory becomes unavailable.

17 Conclusion

The implementation process showed that a hierarchical scheduler is suited to host a mixed criticality system. Here, critical groups correspond one-to-one to the task schedulers: because of its very composition, the hierarchical scheduler provides isolation between critical groups.

Because a hierarchical scheduler is modular by definition, and here serves as a system host, it was desired to use a fast, general-purpose, and object-oriented langauge. C++ worked well: the implementation process never ran into any obstacles that were perceived as insurmountable.

Also, the many tools used in this project, all of them available on a common Linux system, worked as intended, and for the most part without complications. Among the tools and technologies used were the GRUB configuration to isolate the cores, then `taskset` to steer processes to the best-effort core; moreover, there was a cgroup to control the best-effort core software; `perf_event_open` to poll the number of best-effort core LLC misses; and, `tmux` and zsh to make an interface to automatize experiments.

Last, the idea to specify task systems in text files that were parsed by `hs` at runtime proved powerful, not the least in experiments that required fast employment of different systems. It could all be accounted for, and automatized.

Regretably, somewhere along the way, the sum of the project became less than the sum of its parts. While `hs` can reduce the number of best-effort core memory accesses, it fails to do so with sufficient precision, as is evident by the memory experiment.

At this point, there is not any theory as to what part of the system is the weakest link of the chain. It is not even clear if the lack of precision is the consequence of one or a few such parts, or the overall architecture, and, if so, at what level(s).

Moreover, there is a worse, even fatal system exposure, which nonetheless is more satisfying because the reason is known: because `hs` is itself suspectible to best-effort core interference, in face of massive (but possible) best-effort core memory traffic, the throttling mechanism – a part of `hs` – reverts to a dysfunctional state along with `hs` and the rest of the system, as all memory becomes unavailable.

Here, I should be noted that the relative success of the memory experiment is achieved in the face of *moderate* best-effort contention – most likely, it would not work at all if exposed to the same interference as the audio experiment.

Note: Some of the experiments that were carried out in the course of this project are not mentioned here. The reason is that those experiments, while not exactly failing, did not offer any key insights that are not conveyed by subsequent experiments.

18 References

[1] Abeni and Buttazzo. “Integrating Multimedia Applications in Hard Real-Time Systems”. *Real-Time Systems Symposium. 19th IEEE Proceedings.* 1998.

[2] Heechul Yun et al. *MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms.* University of Illinois at Urbana-Champaigne, 2013.

[3] Hyoseung Kim et al. *Bounding memory interference delay in COTS-based multi-core systems.* Carnegie Mellon University, 2014.

[4] Mathai Joseph et. al. *Real-time Systems: Specification, Verification, and Analysis.* Prentice Hall, 1996. ISBN: 0-13-455297-0.

[5] Neil Audsley. *Memory architectures for NoC-based real-time mixed criticality systems.* Department of Computer Science, University of York, UK, 2013.

[6] Burns and Wellings. *Real-Time Systems and Programming Languages.* Second edition. Addison-Wesley, 1997. ISBN: 0-201-40365-X.

[7] Burns and Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX.* Fourth edition. Addison-Wesley, 2009. ISBN: 978-0-321-41745-9.

[8] Peter Dyson. *The Unix Desk Reference: The hu.man Pages.* SYBEX, 1996. ISBN: 0-7821-1658-2.

[9] Leonardo Ecco et al. *Privatization and Fixed Priority Scheduling.* Institute of Computer and Network Engineering, TU Braunschweig, Germany, 2014.

[10] Leonardo Ecco et al. *Workload-aware Shaping of Shared Resource Accesses in Mixed-criticality Systems.* Institute of Computer and Network Engineering. Technische Universität Braunschweig, Germany, 2014.

[11] Georgia Giannopoulou et al. *Mapping Mixed-Criticality Applications on Multi-Core Architectures.* Computer Engineering and Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland, 2013.

[12] Georgia Giannopoulou et al. *Scheduling of Mixed-Criticality Applications on Resource-Sharing Multicore Systems.* Computer Engineering and Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland, 2013.

[13] Michael Kofler. *Linux: Installation, Configuration, and Use.* Second Edition. Addison-Wesley, 1999. ISBN: 0201596288.

[14] James Layland and Chang Liu. “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”. *JACM* 20.1 (1973).

[15] Ronald Leach. *Advanced Topics in UNIX: Processes, Files, and Systems*. John Wiley and Sons, 1994. ISBN: 0-471-03663-3.

[16] Aloysius Mok. *Fundamental design problems of distributed systems for the hard-real-time environment*. Massachusetts Institute of Technology, 1983.

[17] Aloysius Mok and Deji Chen. “A Multiframe Model for Real-Time Tasks”. *IEEE Transactions on Software Engineering* 23 (1996).

[18] Rodney Howell Sanjoy Baruah Louis Rosier. “Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor”. *Real-Time Systems* 2 (4 1990).

[19] Alan Shaw. *Real-Time Systems and Software*. John Wiley and Sons, 2001. ISBN: 0-471-35490-2.

[20] Insik Shin and Insup Lee. “Periodic Resource Model for Compositional Real-Time Guarantees”. *2003 Proceedings of the 24th IEEE International RTSS* (2003).

[21] Martin Stigge. *Real-Time Workload Models: Expressiveness vs. Analysis Efficiency*. Uppsala University, 2014. ISBN: 978-91-554-8888-8.

[22] Bjarne Stroustrup. *The C++ Programming Language*. Second Edition. Addison-Wesley, 1992. ISBN: 0-201-53992-6.

[23] Douglas Troy. *UNIX Systems*. Addison-Wesley, 1990. ISBN: 0-201-19827-4.

[24] Heechul Yun et al. “Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality”. *Real-Time Systems (ECRTS) 2012 24th Euromicro Conference*. 2012.

A Appendix A: Memory experiment fallout

Note: If and whenever the tables above are spotwise incomplete, that is because a failed hard real-time task system has made `hs` exit preemptly, without a result to output. Correspondingly, the digits actually outputted are all of systems whose tasks completed before or at their deadlines, without fail.

A.a base-faculty-1

supposed	actual	error ratio
90009	569553	0.841966
180009	637292	0.717541
270009	785984	0.65647
360009	1346240	0.732582
450009	1113195	0.59575
540009	1214229	0.555266
630009	1240293	0.492048
720009	1338046	0.461895
810009	1497965	0.45926
900009	1508547	0.403393
990009	1593194	0.378601
1080009	1713363	0.369655
1170009	1996471	0.413961
1260009	1931254	0.347569
1350009	1973170	0.315817
1440009	2003326	0.281191
1530009	2215148	0.309297
1620009	2310808	0.298943
1710009	2359580	0.275291
1800009	2581487	0.302724
1890009	2421919	0.219623
1980009	2489843	0.204766
2070009	2853090	0.274468
2160009	2962518	0.270887
2250009	2545778	0.11618
2340009	2829721	0.17306
2430009	2856887	0.149421
2520009	3063667	0.177453
2610009	3070623	0.150007
2700009	2875349	0.0609804
2790009	2915481	0.0430365
2880009	3030739	0.0497337
2970009	3506273	0.152944
3060009	3038450	0
3150009	3350510	0.0598419
3240009	3583130	0.0957601
3330009	3654194	0.0887159
3420009	3529574	0.031042
3510009	3720604	0.0566024
3600009	3843631	0.0633833
3690009	3796497	0.028049
3780009	4305817	0.122116
3870009	4210867	0.0809472
3960009	4041210	0.0200932
4050009	4255133	0.0482062
4140009	4389235	0.0567812
4230009	4338487	0.0250036
4320009	4690722	0.0790311
4410009	4491996	0.0182518
4500009	4658630	0.0340489
4590009	4766878	0.0371037
4680009	4767864	0.0184265
4770009	4961792	0.038652
4860009	4780019	0
4950009	4625017	0
5040009	4743901	0
5130009	4863488	0
5220009	4389665	0
5310009	4963470	0
5400009	5402580	0.000475884
5490009	4721948	0
5580009	4861183	0
5670009	5114396	0
5760009	4900641	0
5850009	5282853	0
5940009	5217269	0
6030009	5578874	0
6120009	5293155	0
6210009	5141524	0
6300009	5575973	0
6390009	6046491	0
6480009	5481410	0
6570009	6046994	0
6660009	6675867	0.00237542
6750009	6480429	0
...	...	0

A.b base-faculty-2

supposed	actual	error ratio
140077	864362	0.837942
280077	667593	0.580467
420077	948082	0.556919
560077	936698	0.402073
700077	993038	0.295015
840077	1160923	0.276371
980077	1398604	0.299246
1120077	1396711	0.198061
1260077	1673783	0.247168
1400077	1735809	0.193415
1540077	1820067	0.153835
1680077	1848655	0.0911895
1820077	2140132	0.149549
1960077	2537678	0.22761
2100077	2208120	0.0489299
2240077	2515258	0.109405
2380077	2614506	0.0896647
2520077	2881609	0.125462
2660077	2885903	0.0782514
2800077	3165003	0.1153
2940077	3332968	0.11788
3080077	3420032	0.0994011
3220077	3505998	0.081552
3360077	3362380	0.000684932
3500077	3537422	0.0105571
3640077	3716196	0.020483
3780077	3810985	0.00811024
3920077	4104536	0.0449403
4060077	4195423	0.0322604
4200077	4076776	0
4340077	4097467	0
4480077	4112332	0
4620077	4532331	0
4760077	4780345	0.00423986
4900077	4739089	0
5040077	5108784	0.0134488
5180077	4898015	0
...	...	0

B Appendix B: Formulas

B.a system

$\mathtt{g} = (\mathtt{l}, \mathtt{p}, \mathtt{r}, \mathtt{S}, \mathtt{a})$ where $0 < \mathtt{r} < \mathtt{p} < \mathtt{l}$

$\mathtt{S} = \{\mathtt{s_i}\} = \{(\mathtt{m_i}, \mathtt{b_i}, \mathtt{P_i}, \alpha_\mathtt{i})\}$

B.b sporadic task model

$\mathtt{P_i} = \{\mathtt{t_j}\} = \{(\mathtt{C_j}, \mathtt{D_j}, \mathtt{T_j})\}$

$\mathtt{t_j} = (\mathtt{C_j}, \mathtt{D_j}, \mathtt{T_j})$

B.c response time

$\mathtt{r_j} = \mathtt{Q_j} + \mathtt{o_j} + \mathtt{i_j} + \mathtt{C_j}$

$\mathtt{z} = \frac{\mathtt{r_j}}{\mathtt{p}}$

$\mathtt{o_j} = \left\lfloor \frac{\mathtt{r_j}}{\mathtt{r}} \right\rfloor \mathtt{o}$

$\mathtt{i_j} = \lceil \mathtt{z} \rceil \mathtt{m_i} \mathtt{58.5\ ns}$

Queuing:

$\mathtt{Q_j} = \mathtt{Q_j^g} + \mathtt{Q_j^s}$

$\mathtt{Q_j^g} = \sum_{\{\mathtt{t_j}, \mathtt{t_k}\} \subseteq \mathtt{P_i}, \mathtt{t_k} \in \mathtt{hp}(\mathtt{t_j})} \left\lceil \frac{\mathtt{r_j}}{\mathtt{T_k}} \right\rceil \mathtt{C_k}$

$\mathtt{Q_j^s} = (\lceil \mathtt{z} \rceil + 1)(\mathtt{p} - \mathtt{b_i})$

B.d resource server

$\mathtt{I} = \Delta\theta = \theta_2 - \theta_1$

B.d.a supply

$$w = \left\lfloor \frac{I}{p} \right\rfloor - 1$$

$$B_i = w b_i$$

$$R = I - wp$$

$$U = p - \frac{R}{2}$$

$$\beta_i = 2\max(0, b_i - U)$$

$$sbf(s_i, I) = B_i + \beta_i - \rho$$

B.d.b demand

$$h = \min(I - fT_j, C_j)$$

$$dbf(t_j, I) = fC_j + h$$

$$dbf(P_i, I) = \sum_{t_j \in P_i} dbf(t_j, I)$$

$$f = \left\lfloor \frac{I}{T_j} \right\rfloor$$

C Appendix C: hs code

This appendix contains the complete C++ source code for hs as well as a groff man page which describes the use of all command-line options.

C.a man page for hs

```
HS(1)                      Emanuel Berg Computing Manual



NAME
 hs - hierarchical scheduler


SYNOPSIS
 hs
    [ -d | --debug           ]   [ -f | --freeze-core                ]
    [ -h | --hard            ]
    [ -l | --log             ]   [ -m | --memory-budget-add-on  TERM ]
    [ -p | --poll-llc        ]
    [ -P | --fork-processes  ]
    [ -q | --quiet           ]   [ -Q | --really-quiet               ]
    [ -r | --run             ]   [ -s | --system  TASK-SYSTEM-FILE   ]
    [ -v | --verbose         ]   [ -w | --wait-for-forked-processes  ]


DESCRIPTION
 hs  is a hierarchical scheduler. It executes either hard-coded software, or
 forked processes, according to a  polled-preemptive  global  EDF  algorithm
 acting on the real-time parameters of the sporadic task model.


OPTIONS
 -d, --debug
        Output various hard-coded debug information.


 -f, --freeze-core
        Do  freeze  the best-effort core when it exceeds its DRAM budget. To
        do this, perf_event_open(2) is used along with a Linux cgroup.   The
```

```
       use  of  -f  implies --poll-llc because that is how DRAM fetches are
       booked.  Note: For this to work, either run hs with 'sudo'; or,  set
       the owner of hs to root, and then set the SUID bit; or, do something
       else that amounts to the same.


-h, --hard
       Exit the scheduler with error code  -1  immediately  if  a  task  is
       delayed.


-l, --log
       Log the time in nanoseconds at every tick to the file tick_times.log
       in the same directory as hs.


-m, --memory-budget-add-onTERM
       Add TERM to all memory budgets.


-p, --poll-llc
       Every tick, poll the DRAM last-level-cache (LLC)  to  find  out  how
       many non-cached DRAM accesses the best-effort core has made.


-P, --fork-processes
       Don't  use hard-coded mock software; fork processes.  This means the
       system file must consist of  commands  (including  their  arguments)
       that are executable on the underlying system. E.g., to do the -equiv
       alent of echo hello fool put /bin/echo(hello  fool)  in  the  system
       file. (At this point hs cannot mix mock software and real processes;
       and, absolute paths to executables are required.) The easiest way to
       use  this  is  to  put  all  commands in a script, and then use that
       script in the system file. (When freezing and thawing,  the  process
       group  id  (PGID)  is used, as to affect offsprings of the script as
       well.)


-q, --quiet
       Don't output task state transitions. Hard-coded task software should
       typically  be quiet as well although that has to be coded explicitly
       in hs.


 -Q, --really-quiet
       As --quiet only shut up hard-coded task software as well.


 -r, --run
       Run the system when it is loaded without  confirmation.   (Sometimes
       though it is useful to have the system only loaded, not executed, to
       be triggered exactly when needed.)


 -s, --systemTASK-SYSTEM-FILE
       Load the system from the specified file. Creating  systems  -interac
       tively  is  just fun and games: it is much better to exclusively use
       files. Use this with --run to execute a system from a file.


 -v, --verbose
       Every tick, output the state of the entire system.


 -w, --wait-for-forked-processes
       At the end of the execution of hs, wait(2) for all forked  processes
       to  terminate.  Use  with  care: with non-terminating processes this
       makes hs non-terminating as well. This option overrides the  "Global
       lifetime"  parameter as long as there are children left.  -w implies
       --fork-processes because otherwise there are none to wait for.


TASK SYSTEM
 A task system is defined in a task-system text file.  There are a couple of
 examples  in  ./hs-linux/sys - otherwise, run hs interactively to see how a
 system is expressed, then put the exact same in a text file.  If  need  be,
 later  modify  the  selfsame text file to fine-tune the system, rather that
 creating one anew interactively.


DOCUMENTATION AND CREDITS
 There is an ambitious PDF  document  that  describes  this  project:  ./hs-
```

linux/docs/report.pdf

QUESTIONS AND FEEDBACK

Written by Emanuel Berg <embe8573@student.uu.se> **for** Uppsala University, 2014.

SEE ALSO

fork(2), signal(2), **wait**(2), perf_event_open(2)

C.b ask

```
#ifndef ASK_HH
#define ASK_HH

int ask_how_many(const char* what);
bool ask_yes_or_no(const char* q);

#endif
```

```
#include "ask.hh"
#include <iostream>

int ask_how_many(const char* what) {
  int number = 0;
  while (number <= 0) {
    std::cout << "Create this many " << what << ": ";
    std::cin >> number;
  }
  return number;
}

bool ask_yes_or_no(const char* q) {
  const char no_input  = 'n';
  const char yes_input = 'y';
  char input = '?';
  while ((input != no_input) and
         (input != yes_input)) {
    std::cout << std::endl << q << " [enter y or n] ";
    std::cin >> input;
  }
  return (input == yes_input);
}
```

C.c be

```
#ifndef BE_HH
#define BE_HH

#include <stdio.h>
```

```
void init_BE_core ();
void close_BE_core();
void unfreeze_BE_core();
void freeze_BE_core(int fd);

#endif

#include "be.hh"
#include "options.hh"
#include "llc.hh"
#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include <unistd.h>


bool is_frozen;
FILE* f = NULL;

void init_BE_core() {
  const char* state_file = "/sys/fs/cgroup/freezer/0/freezer.state";
  f = fopen(state_file, "w");
  if (f) {
    system("echo THAWED > /sys/fs/cgroup/freezer/0/freezer.state");
    is_frozen = false;
  }
  else {
    std::cerr << "Couldn't open file: " << state_file        << std::endl
              << "(Did you init the cgroup freezer system?)" << std::endl;
    exit(EXIT_FAILURE);
  }
}

void unfreeze_BE_core() {
  if (is_frozen) {
    if (debug) { std::cout << "BE core THAWED." << std::endl; }
    system("echo THAWED > /sys/fs/cgroup/freezer/0/freezer.state");
    is_frozen = false;
  }
}
```

```
void freeze_BE_core(int fd) {
  if (!is_frozen) {
    if (debug) { std::cout << "BE core FROZEN!" << std::endl; }

    if (test_cgroup_overhead) {
      system("echo FROZEN > /sys/fs/cgroup/freezer/0/freezer.state");
      long long count_at_freeze  = query_BE_monitor(fd);
      std::cout << "Froze at " << count_at_freeze << " misses." << std::endl;
      std::cout << "Sleeping for 10 seconds..."                 << std::endl;
      sleep(10);
      long long count_at_awake   = query_BE_monitor(fd);
      long long slipped_accesses = count_at_awake - count_at_freeze;
      std::cout << "Slipped accesses: " << slipped_accesses << std::endl;
    }
    else { system("echo FROZEN > /sys/fs/cgroup/freezer/0/freezer.state"); }

    is_frozen = true;
  }
}

void close_BE_core() { fclose(f); }
```

C.d file_io

```
#ifndef FILE_IO_HH
#define FILE_IO_HH

#include <fstream>

void file_goto_next_char (std::ifstream& f, char c);
void file_goto_next_colon(std::ifstream& f);
void file_goto_next_newline(std::ifstream& f);

#endif

#include "file_io.hh"
#include <fstream>

void file_goto_next_char(std::ifstream& f, char c) {
  f.ignore(73, c);
}
```

```
void file_goto_next_colon(std::ifstream& f) {
  file_goto_next_char(f, ':');
}

void file_goto_next_newline(std::ifstream& f) {
  file_goto_next_char(f, '\n');
}
```

C.e global_scheduler

```
#ifndef GLOBAL_SCHEDULER_HH
#define GLOBAL_SCHEDULER_HH

#include "task_scheduler.hh"
#include "main.hh"
#include "time_io.hh"
#include "log.hh"

#include <string>
#include <ctime>

extern bool crash;

class Global_Scheduler {
private:
  Log logger;
  FILE* get_log_file();
  void log_time();

  ms_t scheduling_rate;
  ms_t lifetime;
  ms_t period, period_left;

  void resupply();
  int number_of_schedulers;
  Task_Scheduler** schedulers;

  std::string scheduling_algorithm;
  static const std::string scheduling_algorithms[];
  void set_tasks_path();
  void read_parameters_from_keyboard();
  void read_tasks_from_keyboard();
  std::string tasks_path;
  std::string filename;

  long long sum_memory_budget();
  int number_of_periods();

public:
  Global_Scheduler();
 ~Global_Scheduler();
  static void check_schedulers(Global_Scheduler* gs, Task_Scheduler* ts[]);
  void load(std::string file);

  void schedule();
  void schedule_EDF();

  void tick(ms_t tick_time, int llc_fd);
  void check_period(ms_t tick_time);

  void run();

  void store();

  friend std::ostream& operator<<(std::ostream& os, Global_Scheduler& gs);
  void get_system_file(std::ifstream& f, std::string file);
  void set_global_scheduler_parameters(std::ifstream& f);
  void set_global_scheduling_algorithm(std::ifstream& f);
  void set_task_schedulers(std::ifstream& f);
  void init_from_file(std::string file);
  void init_from_keyboard();
};

#endif

#include "global_scheduler.hh"
#include "task_scheduler.hh"
#include "options.hh"
#include "time_io.hh"
#include "file_io.hh"
```

```
#include "ask.hh"
#include "llc.hh"
#include "be.hh"
#include "log.hh"

#include <cassert>
#include <cstddef>
#include <ctime>
#include <cmath>

#include <chrono>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <string>
#include <thread>
#include <unistd.h>

#include <sys/types.h>
#include <sys/wait.h>

bool crash = false;

const std::string Global_Scheduler::scheduling_algorithms[] = { "EDF" };

void Global_Scheduler::load(std::string file) {
  std::ifstream f;
  get_system_file(f, file);
  set_global_scheduler_parameters(f);
  set_global_scheduling_algorithm(f);
  set_task_schedulers(f);
}

void Global_Scheduler::get_system_file(std::ifstream& f, std::string file) {
  if (file != "") {
    f.open(tasks_path + file, std::ios_base::in);
    if (!f.good()) {
      std::cerr << "Can't find file provided with the -s (--system) option: "
                << tasks_path << file << std::endl;
      exit(EXIT_FAILURE);
    }
  }
  else {
    bool first_try = true;
    do {
      if (first_try) { first_try = false; }
      else           { std::cout << "No such file. Try again!" << std::endl;
      std::cout << std::endl << "In " << tasks_path << ", load what file: ";
      std::cin >> filename;
      f.open(tasks_path + filename, std::ios_base::in);
    } while (!f.good());
  }
}

void Global_Scheduler::set_global_scheduler_parameters(std::ifstream& f) {
  file_goto_next_colon(f);
  f >> scheduling_rate;
  assert(scheduling_rate > ms_t(0));

  file_goto_next_colon(f); f >> period;
  assert(period > ms_t(0));

  period_left = period;

  file_goto_next_colon(f); f >> lifetime;
  assert(period <= lifetime);
}

void Global_Scheduler::set_global_scheduling_algorithm(std::ifstream& f) {
  file_goto_next_colon(f);
  f >> scheduling_algorithm;
  bool found_algorithm = false;
  for (int i = 0,
           num_scheduling_algorithms = sizeof(scheduling_algorithms)/
                                       sizeof(scheduling_algorithms[0]);
       i < num_scheduling_algorithms;
       i++) {
    if (scheduling_algorithm == scheduling_algorithms[i]) {
      found_algorithm = true;
      break;
    }
  }
  if (!found_algorithm) {
    std::cerr << " Couldn't find global scheduler algorithm: "
              << scheduling_algorithm << std::endl;
    exit(EXIT_FAILURE);
  }
}

void Global_Scheduler::set_task_schedulers(std::ifstream& f) {
  number_of_schedulers = 0;
  std::streampos pos = f.tellg();
  file_goto_next_newline(f);
  while (f.good() and (f.peek() != EOF)) {
    char current = f.get();
    if ((current == '\n') or (current == ' ')) { number_of_schedulers++; }
    file_goto_next_newline(f);
  }
  f.seekg(pos);
  assert(f.good());
  assert(number_of_schedulers > 0);

  size_t schedulers_size = number_of_schedulers*sizeof(Task_Scheduler *);
  schedulers = (Task_Scheduler **)malloc(schedulers_size);
  assert(schedulers);

  for (int i = 0; i < number_of_schedulers; i++) {
    schedulers[i] = new Task_Scheduler(f);
  }
  check_schedulers(this, schedulers);
}

void Global_Scheduler::store() {
  std::cout << "[ to store this SYSTEM, put this in the file "
            << tasks_path << "SYSTEM ]" << std::endl;
  std::cout <<
    "Global scheduling rate: "       << scheduling_rate       << std::endl <<
    "Global period: "                << period                << std::endl <<
    "Global lifetime: "              << lifetime              << std::endl <<
    "Global scheduling algorithm: "  << scheduling_algorithm  << std::endl;
  for (int i = 0; i < number_of_schedulers; i++) {
    std::cout << std::endl;
    schedulers[i]->store();
  }
  std::cout << "[ done ] " << std::endl;
}

Global_Scheduler::~Global_Scheduler() {
  for (int i = 0; i < number_of_schedulers; i++) {
    delete schedulers[i];
  }
  if (schedulers) { free(schedulers); }
}

void Global_Scheduler::init_from_file(std::string file) {
  set_tasks_path();
  load(file);
}

void Global_Scheduler::init_from_keyboard() {
  set_tasks_path();
  read_parameters_from_keyboard();
  read_tasks_from_keyboard();
}

void Global_Scheduler::set_tasks_path() {
  const int max_path_len = 85;
  char tasks_abs_path[max_path_len];
  readlink("/proc/self/exe", tasks_abs_path, max_path_len);
  std::string tap_str = tasks_abs_path;
  tasks_path = tap_str.substr(0, tap_str.find_last_of('/'));
  tasks_path += "/../sys/";
}

void Global_Scheduler::read_parameters_from_keyboard() {
  std::cout << std::endl;
  do {
    do {
      std::cout << "Global scheduling rate: ";
      std::cin >> scheduling_rate;
    } while (!(scheduling_rate > ms_t(0)));
    do {
      std::cout << "Global scheduler period: ";
      std::cin >> period;
    } while (!(period > ms_t(0)));
    do {
      std::cout << "System lifetime: ";
      std::cin >> lifetime;
    } while (!(lifetime > ms_t(0)));
  } while (!(period <= lifetime));
  period_left = period;

  // just set this, there is only one so far anyway
  scheduling_algorithm = scheduling_algorithms[0];
  std::cout << "Global scheduling algorithm: "
            << scheduling_algorithm << std::endl;
}

void Global_Scheduler::read_tasks_from_keyboard() {
  number_of_schedulers = ask_how_many("task schedulers");

  size_t schedulers_size = number_of_schedulers*sizeof(Task_Scheduler *);
  schedulers = (Task_Scheduler **)malloc(schedulers_size);

  for (int i = 0; i < number_of_schedulers; i++)
    { schedulers[i] = new Task_Scheduler(); }

  check_schedulers(this, schedulers);
}

Global_Scheduler::Global_Scheduler() {
  period_left = ms_t(0);
  number_of_schedulers = 0;
  if (log_tick) { logger.open_log_file(); }
}

void Global_Scheduler::schedule() {
  if (scheduling_algorithm == "EDF") { schedule_EDF(); }
  else {
    std::cerr << "Unknown algorithm: " << scheduling_algorithm << std::endl;
    exit(EXIT_FAILURE);
  }
}

void Global_Scheduler::schedule_EDF() {
  int selected_scheduler;

  tp_t earliest_deadline;
  tp_t candidate_deadline;

  bool is_data = false;
  for (int i = 0; i < number_of_schedulers; i++) {
    schedulers[i]->schedule();
    if (schedulers[i]->is_ready()) {
      candidate_deadline = schedulers[i]->earliest_deadline();
      if (!is_data or (candidate_deadline < earliest_deadline)) {
        is_data = true;
        earliest_deadline = candidate_deadline;
        selected_scheduler = i;
      }
    }
  }
  if (is_data) { schedulers[selected_scheduler]->run(); }
  // else if (do_freeze_BE_core) {
  //   if (debug) { std::cout << "No scheduler is ready: unfreeze BE!" << std
  //   unfreeze_BE_core();
  // }
}

void Global_Scheduler::resupply() {
  period_left = period;
  for (int i = 0; i < number_of_schedulers; i++) {
    schedulers[i]->resupply();
  }
}

void Global_Scheduler::check_period(ms_t tick_time) {
  period_left -= tick_time;
  if (period_left < ms_t(0)) { resupply(); }
}

void Global_Scheduler::tick(ms_t tick_time, int llc_fd) {
  long long current_BE_accesses = (poll_llc ? query_BE_monitor(llc_fd) : 0);
  if (log_tick) { logger.print_time_to_file(); }
  schedule();
  for (int i = 0; i < number_of_schedulers; i++) {
    schedulers[i]->tick(tick_time, current_BE_accesses, llc_fd);
  }
  check_period(tick_time);
}

void Global_Scheduler::run() {
  if (do_freeze_BE_core) { init_BE_core(); }

  int llc_fd = 0;
  if (poll_llc) { llc_fd = setup_BE_monitor(); }

  tp_t system_start(std::chrono::system_clock::now());
  tp_t system_end(system_start + lifetime);
  tp_t re_sched_time;
  int i = 0;
  do {
    re_sched_time = system_start + ++i*scheduling_rate;
    tick(scheduling_rate, llc_fd);
    std::this_thread::sleep_until(re_sched_time);
  } while (!crash and std::chrono::system_clock::now() < system_end);

  pid_t status;
  if (wait_for_forked_processes) {
    std::cout << "Waiting for children to terminate..." << std::endl;
    do {
      wait(&status);
      if (status == -1 && errno != ECHILD) {
        std::cerr << "Error while waiting for children to terminate." << std::endl;
        exit(EXIT_FAILURE);
      }
      re_sched_time = system_start + ++i*scheduling_rate;
      tick(scheduling_rate, llc_fd);
      std::this_thread::sleep_until(re_sched_time);
    } while (status > 0 );
  }

  if (poll_llc) {
    long long BE_accesses = query_BE_monitor(llc_fd);
    if (latex) {
      long long execution_memory_budget = sum_memory_budget()*number_of_periods();
      long long errors = BE_accesses - execution_memory_budget;
      float error_ratio = (errors > 0 ? (float)errors/(float)BE_accesses : 0
      std::cout << "  "
                << execution_memory_budget << " & "
                << BE_accesses             << " & "
                << error_ratio             << " \\\\ " << std::endl;
    }
    else {
      std::cout << "Terminated: " << BE_accesses << " BE accesses" << std::en
    }
    close_BE_monitor(llc_fd);
  }

  if (do_freeze_BE_core) { close_BE_core(); }
}

long long Global_Scheduler::sum_memory_budget() {
  long long sum_budget = 0;
  for (int s = 0; s < number_of_schedulers; s++) {
    sum_budget += schedulers[s]->get_max_BE_accesses();
  }
  return sum_budget;
}

int Global_Scheduler::number_of_periods() {
  int int_lifetime = std::chrono::duration_cast<std::chrono::milliseconds>(li
  int int_period   = std::chrono::duration_cast<std::chrono::milliseconds>(pe
  int periods      = int_lifetime/int_period;
  int remainder    = int_lifetime%int_period;
  if (remainder) { periods++; }
  return periods;
}

std::ostream& operator<<(std::ostream& os, Global_Scheduler& gs) {
  os << "*** global (scheduler) scheduler ***"     << std::endl
     << "Rate: "       << gs.scheduling_rate       << std::endl
     << "Period: "     << gs.period
     << " (left: "     << gs.period_left << ")"    << std::endl
     << "Lifetime: "   << gs.lifetime              << std::endl
     << "Schedulers: " << gs.number_of_schedulers  << std::endl << std::endl;
  for (int i = 0; i < gs.number_of_schedulers; i++) {
    os << gs.schedulers[i];
  }
  return os;
}

void Global_Scheduler::check_schedulers(Global_Scheduler* gs,
                                        Task_Scheduler*   ts[]) {
  ms_t total_budget(0);
  for (int i = 0; i < gs->number_of_schedulers; i++) {
    ts[i]->check_tasks();
    total_budget += ts[i]->get_budget();
  }
  assert(total_budget <= gs->period);
}
```

C.f llc

```
#ifndef LLC_HH
#define LLC_HH

#include <unistd.h>
#include <linux/perf_event.h>

long perf_event_open(struct perf_event_attr* hw_event,
                            pid_t pid,
                            int cpu,
                            int group_fd,
                            unsigned long flags);
void config_event(struct perf_event_attr* pe);
int init_event(struct perf_event_attr* pe);
void activate_event(int fd);
void close_event(int fd);
long long query_BE_monitor(int fd);
bool do_block_BE(int fd, long long max_misses);
int setup_BE_monitor();
void close_BE_monitor(int fd);

const int           ALL_PIDS  = -1;
const int           ALL_CPUS  = -1;
const int           LEADER    = -1;
const unsigned long NO_FLAGS  =  0;
```

```
#endif

#include "llc.hh"

#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <linux/perf_event.h>
#include <asm/unistd.h>
#include <time.h>
#include <unistd.h>

long perf_event_open(struct perf_event_attr* hw_event,
                     pid_t pid,
                     int cpu,
                     int group_fd,
                     unsigned long flags) {
  return (syscall(__NR_perf_event_open, hw_event, pid, cpu, group_fd, flags))
}

void config_event(struct perf_event_attr* pe) {
  pe->type = PERF_TYPE_HW_CACHE;
  pe->size = sizeof(struct perf_event_attr);
  pe->config = PERF_COUNT_HW_CACHE_LL                    |
               PERF_COUNT_HW_CACHE_OP_READ      <<  8  |
               PERF_COUNT_HW_CACHE_RESULT_MISS  << 16;
  pe->disabled        = 1;
  pe->exclude_kernel  = 1;
  pe->exclude_hv      = 1;
}

int init_event(struct perf_event_attr* pe) {
  int core = atoi(std::getenv("BE_CORE"));
  int fd = perf_event_open(pe, ALL_PIDS, core, LEADER, NO_FLAGS);
  if (fd == -1) {
    std::cerr << "Error with the LLC counter: " << pe->config;
    exit(EXIT_FAILURE);
  }
  return fd;
}
```

```
void activate_event(int fd) {
  ioctl(fd, PERF_EVENT_IOC_RESET, 0);
  ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);
}

void close_event(int fd) {
  ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);
  close(fd);
}

int setup_BE_monitor() {
  struct perf_event_attr pe_BE_monitor;
  memset(&pe_BE_monitor, 0, sizeof(struct perf_event_attr));
  config_event(&pe_BE_monitor);
  int fd = init_event(&pe_BE_monitor);
  activate_event(fd);
  return fd;
}

long long query_BE_monitor(int fd) {
  long long count = 0;
  read(fd, &count, sizeof(count));
  return count;
}

bool do_block_BE(int fd, long long max_misses) {
  return (query_BE_monitor(fd) > max_misses);
}

void close_BE_monitor(int fd) { close_event(fd); }
```

C.g log

```
#ifndef LOG_HH
#define LOG_HH

#include <iostream>

class Log {
private:
  const char* log_file;
  FILE* log;
  bool opened;
  void close_log_file();
  long long unsigned int now_in_ns();
public:
  Log();
 ~Log();
  void open_log_file();
  void print_time_to_file();
};

#endif

#include "log.hh"
#include "time_io.hh"
#include <chrono>
#include <iostream>

Log::Log() {
  log_file = "./tick_times.log";
  opened = false;
}

Log::~Log() {
  if (opened) { close_log_file(); }
}

void Log::open_log_file() {
  log = fopen(log_file, "w");
  if (!log) {
    std::cerr << "Couldn't create or open file: " << log_file << std::endl;
    exit(EXIT_FAILURE);
  }
  else { opened = true; }
}

void Log::close_log_file() { fclose(log); }

long long unsigned int Log::now_in_ns() {
  std::chrono::time_point<std::chrono::system_clock> now = std::chrono::syste
  time_t tnow = std::chrono::system_clock::to_time_t(now);
  tm *date = std::localtime(&tnow);
  date->tm_hour = 0;
  date->tm_min = 0;
  date->tm_sec = 0;
  tp_t midnight = std::chrono::system_clock::from_time_t(std::mktime(date));

  return std::chrono::duration_cast<std::chrono::nanoseconds>(now - midnight).count
}

void Log::print_time_to_file() { fprintf(log, "%llu\n", now_in_ns()); }
```

C.h main

```
#ifndef MAIN_HH
#define MAIN_HH

int main(int argc, char* argv[]);

#endif

#include "main.hh"
#include "global_scheduler.hh"
#include "ask.hh"
#include "options.hh"

#include <iostream>

int main(int argc, char* argv[]) {
  set_options(argc, argv);
  bool read_from_file = ((task_system != "") or
                         ask_yes_or_no("Load an existing task system?"));
  Global_Scheduler gs = Global_Scheduler();
  if (read_from_file) { gs.init_from_file(task_system); }
  else {
    gs.init_from_keyboard();
    gs.store();
  }
  if (!quiet) { std::cout << gs; }
  if (run or ask_yes_or_no("System loaded. Run it?")) { gs.run(); }

  if   (crash) { return -1; }
  else         { return  0; }
}
```

C.i options

```
#ifndef OPTIONS_HH
#define OPTIONS_HH

#include <string>

void print_help_and_exit(std::string);
void set_options(int argc, char* argv[]);

extern bool debug;
extern bool print_task_tick;
extern bool quiet;
extern bool really_quiet;
extern bool log_tick;
extern bool latex;

extern std::string task_system;
extern bool run;
extern bool hard;

extern bool poll_llc;
extern bool do_freeze_BE_core;
extern int memory_budget_add_on;
extern bool test_cgroup_overhead;

extern bool fork_processes;
extern bool wait_for_forked_processes;
extern bool random_delay;

#endif

#include "options.hh"
#include <getopt.h>
#include <iostream>
```

```
#include <string>

// verbose
bool debug               = false;
bool print_task_tick     = false;
bool quiet               = false;
bool really_quiet        = false;
bool log_tick            = false;
bool latex               = false;

// critical system
bool hard                       = false;
bool fork_processes             = false;
bool wait_for_forked_processes  = false;
bool run                        = false;
bool random_delay               = true;
std::string task_system         = "";

// be-core
bool poll_llc              = false;
bool do_freeze_BE_core     = false;
bool test_cgroup_overhead = false;
int memory_budget_add_on  = 0;

void print_help_and_exit(char* program) {
  std::cerr << "Incorrect usage. This program has a man page." << std::endl;
  exit(EXIT_FAILURE);
}

static struct option const long_options[] = {
  {"debug",                no_argument,        NULL, 'd'},
  {"fork-processes",       no_argument,        NULL, 'P'},
  {"freeze-core",          no_argument,        NULL, 'f'},
  {"hard",                 no_argument,        NULL, 'h'},
  {"no-random-delay",      no_argument,        NULL, 'i'},
  {"log-tick",             no_argument,        NULL, 'l'},
  {"memory-budget-add-on", required_argument,  NULL, 'm'},
  {"poll-llc",             no_argument,        NULL, 'p'},
  {"quiet",                no_argument,        NULL, 'q'},
  {"really-quiet",         no_argument,        NULL, 'Q'},
  {"run",                  no_argument,        NULL, 'r'},
  {"system",              required_argument,  NULL, 's'},
  {"test-cgroup-overhead",  no_argument,        NULL, 't'},
  {"verbose",              no_argument,        NULL, 'v'},
  {"wait",                 no_argument,        NULL, 'w'},
  {"latex",                no_argument,        NULL, 'x'},
  {0, 0, 0, 0}
};

void set_options(int argc, char* argv[]) {
  char* program = argv[0];
  while (true) {
    int oi = -1;
    int c = getopt_long(argc, argv, "dhivQqrs:tlm:Ppfwx", long_options, &oi);
    if (c == -1) { break; }
    switch (c) {
    case 'i': random_delay      = false; break;
    case 'd': debug             = true;  break;
    case 'f': do_freeze_BE_core = true;
              poll_llc          = true;  break;
    case 'h': hard              = true;  break;
    case 'l': log_tick          = true;  break;
    case 'v': print_task_tick   = true;  break;
    case 'Q': really_quiet      = true;
    case 'q': quiet             = true;  break;
    case 'r': run               = true;  break;
    case 'p': poll_llc          = true;  break;
    case 'P': fork_processes    = true;  break;
    case 's': if    (optarg) { task_system = optarg;         }
              else           { print_help_and_exit(program); }
              break;
    case 't': do_freeze_BE_core    = true;
              poll_llc             = true;
              test_cgroup_overhead = true;
              break;
    case 'm': if    (optarg) { memory_budget_add_on = atoi(optarg); }
              else           { print_help_and_exit(program); }
              break;
    case 'w': wait_for_forked_processes = true;
              fork_processes = true;
              break;
    case 'x': latex = true; break;
    default: print_help_and_exit(program);
    }
  }
}
```

C.j program

```
#ifndef PROGRAM_HH
#define PROGRAM_HH

#include <string>
#include "time_io.hh"

void chase_pointers(ms_t allowed_time);
void run_program (std::string program, std::string** program_args);
int number_of_args(std::string program);
void helloworld();
void faculty(int highest);

#endif

#include "program.hh"
#include "time_io.hh"
#include "options.hh"

#include <stdlib.h>
#include <limits.h>
#include <unistd.h>
#include <sys/times.h>
#include <math.h>
#include <iostream>
#include <sstream>
#include <chrono>

int number_of_args(std::string program) {
  if      (program == "helloworld")      { return 0; }
  else if (program == "faculty")         { return 1; }
  else {
    std::cerr << "Unknown program: " << program << std::endl;
    exit(EXIT_FAILURE);
  }
}

void run_program (std::string program, std::string** program_args) {
  if       (program == "helloworld") { helloworld(); }
  else if  (program == "faculty")    {
    int highest = atoi(program_args[0]->c_str());
    faculty(highest);
  }
  else {
    std::cerr << "Can't find program: " << program << std::endl;
    exit(EXIT_FAILURE);
  }
}

void helloworld() {
  if (!really_quiet) { std::cout << "Hello, world!" << std::endl; }
}

void faculty(int highest) {
  static int current = 0;
  static int sum = 1;
  bool news = true;
  switch (current) {
  case 0:   { current = highest; sum = 1;  break; }
  case 1:   { news = false;                break; }
  default:  { sum *= current--;            break; }
  }
  if (news and !really_quiet) {
    std::cout << "Faculty is now: " << sum << std::endl;
  }
}
```

C.k sporadic_task

```
#ifndef SPORADIC_TASK_HH
#define SPORADIC_TASK_HH

#include <iostream>
#include <fstream>
#include <string>
#include <ctime>
```

```
#include "main.hh"
#include "time_io.hh"

class Sporadic_Task {
private:
  int id;
  ms_t c, d, t;
  std::string program;
  std::string** program_args;
  int argc;
public:
  Sporadic_Task();
  Sporadic_Task(std::ifstream& f);
  int  get_id()  { return id; }
  ms_t get_c()   { return c;  }
  ms_t get_d()   { return d;  }
  ms_t get_t()   { return t;  }
  void store();
  void run(ms_t duration_secs);
  static void check_task(Sporadic_Task* t);
  friend std::ostream& operator<<(std::ostream& os, const Sporadic_Task* t);
  void launch_extern_program();
};

#endif

#include "sporadic_task.hh"
#include "time_io.hh"
#include "file_io.hh"
#include "options.hh"
#include "program.hh"
#include <unistd.h>

#include <iostream>
#include <fstream>
#include <string>
#include <cassert>
#include <ctime>

#include <thread>
```

```
void Sporadic_Task::run(ms_t duration) {
  if (!quiet) {
    std::cout << "task " << id << " executes: " << program << std::endl;
  }
  // for extern processes, this is implicit
  if (!fork_processes) { run_program(program, program_args); }
}

void Sporadic_Task::launch_extern_program() {
  assert(fork_processes);
  char* const program_name = const_cast<char*>(program.c_str());
  int arg_array_size = argc + 1;
  char* program_argv[arg_array_size];
  program_argv[0] = program_name;
  for (int i = 0; i < argc; i++) {
    program_argv[i + 1] = const_cast<char*>(program_args[i]->c_str());
  }
  program_argv[arg_array_size] = NULL;
  if (execv(program_name, program_argv) == -1) {
    std::cerr << "Error executing program: " << program_name << std::endl;
    exit(127);
  }
}

void Sporadic_Task::check_task(Sporadic_Task* t) {
  assert(      0  <   t->id);
  assert(ms_t(0)  <   t->c);
  assert(   t->c  <=  t->d);
  assert(   t->d  <=  t->t);
}

Sporadic_Task::Sporadic_Task() {
  std::cout << "Create task. State ID, C, D, and T: ";
  std::cin >> id >> c >> d >> t;

  argc = -1;
  while (argc == -1) {
    std::cout << "State program of t" << id << ": ";
    std::cin >> program;
    argc = number_of_args(program);
  }
  if (argc > 0 ) {
    size_t strings_size = argc*sizeof(std::string *);
    program_args = (std::string **)malloc(strings_size);
    assert(program_args);

    std::cout << "Input the "     << argc
              << " arguments of " << program << ": ";
    for (int i = 0; i < argc; i++) {
      std::string *s = new std::string();
      std::cin >> *s;
      program_args[i] = s;
    }
  }
  else { program_args = NULL; }
  check_task(this);
}

Sporadic_Task::Sporadic_Task(std::ifstream& f) {
  file_goto_next_char(f, 't'); f >> id;
  file_goto_next_char(f, '('); f >> c;
  file_goto_next_char(f, ','); f >> d;
  file_goto_next_char(f, ','); f >> t;
  file_goto_next_char(f, ' '); std::getline(f, program, '(');
  int arg_list_start_pos = f.tellg();
  argc = 0;
  char current;
  if (fork_processes) {
    bool done = false;
    while (!done) {
      current = f.get();
      switch (current) {
      case ')': done = true; break;
      case ' ': continue;
      default:
        argc++;
        bool argument_done = false;
        while (!argument_done) {
          current = f.get();
          switch (current) {
          case ',': argument_done = true;         break;
          case ')': argument_done = done = true;  break;
          }
        }
      }
    }
  }
  else { argc = number_of_args(program); }
  f.seekg(arg_list_start_pos);
  if (argc > 0 ) {
    size_t strings_size = argc*sizeof(std::string *);
    program_args = (std::string **)malloc(strings_size);
    assert(program_args);

    char delim = ',';
    for (int i = 0; i < argc; i++) {
      if (i != 0) { f.get(); }
      std::string *s = new std::string();
      if (i == argc - 1) { delim = ')'; }
      std::getline(f, *s, delim);
      program_args[i] = s;
    }
  }
  else { program_args = NULL; }

  check_task(this);
  if (debug) {
    std::cout << "The program is " << program << "." << std::endl;
    for (int i = 0; i < argc; i++) {
      std::cout << "Argument " << i << ": " << *(program_args[i]) << std::end
    }
  }
}

void Sporadic_Task::store() { std::cout << this << std::endl; }

std::ostream& operator<<(std::ostream& os, const Sporadic_Task* t) {
  os << "t" << t->id << " = ("
     << t->c << ", " << t->d << ", " << t->t << ") "
     << t->program;
  int argc = t->argc;
  os << "(";
  for (int i = 0; i < argc; i++) {
    os << *(t->program_args)[i];
    if (i != argc - 1) { os <<  ", "; }
  }
  os << ")";
  return os;
}
```

C.1 task_scheduler

```
#ifndef TASK_SCHEDULER_HH
#define TASK_SCHEDULER_HH

#include "tcb.hh"
#include "main.hh"
#include "time_io.hh"

#include <fstream>
#include <string>

class Task_Scheduler {
private:
  static const std::string scheduling_algorithms[];
  std::string scheduling_algorithm;

  long long max_BE_accesses;
  long long memory_budget_left;

  int critical_level;
  ms_t cpu_budget, cpu_budget_left;

  int number_of_tasks;
  tcb** tasks;
  tcb* hp_task;
  bool selected;
public:
  Task_Scheduler(std::ifstream& f);
  Task_Scheduler();
 ~Task_Scheduler();
  void check_tasks();
  void store();
  void run();
  void check_memory_budget(long long current_BE_accesses, int fd);
  void tick(ms_t tick_time, long long current_BE_accesses, int fd);
  void schedule();
  void schedule_EDF();

  long long get_max_BE_accesses();

  bool is_ready();
  tp_t earliest_deadline();
  void resupply();
  ms_t get_budget() { return cpu_budget; }

  friend std::ostream& operator<<(std::ostream& os, Task_Scheduler* ts);
  void set_tasks(std::ifstream& f);
  void set_scheduler_parameters(std::ifstream& f);
  void set_scheduling_algorithm(std::ifstream& f);
};

#endif

#include "task_scheduler.hh"
#include "file_io.hh"
#include "ask.hh"
#include "tcb.hh"
#include "time_io.hh"
#include "be.hh"
#include "llc.hh"
#include "options.hh"

#include <cstddef>
#include <cassert>
#include <ctime>

#include <iostream>
#include <fstream>
#include <string>

const std::string Task_Scheduler::scheduling_algorithms[] = { "EDF" };

Task_Scheduler::Task_Scheduler(std::ifstream& f) {
  set_scheduler_parameters(f);
  set_scheduling_algorithm(f);
  set_tasks(f);
}

Task_Scheduler::~Task_Scheduler() {
  for (int i = 0; i < number_of_tasks; i++) { delete tasks[i]; }
  free(tasks);
}

void Task_Scheduler::set_scheduler_parameters(std::ifstream& f) {
  selected = false;

  // critical level
  file_goto_next_colon(f); f >> critical_level;
  assert(critical_level > 0);

  // CPU budget
  file_goto_next_colon(f); f >> cpu_budget;
  assert(cpu_budget > ms_t(0));
  cpu_budget_left = cpu_budget;

  // memory budget (memory_budget_add_on is zero unless -m MEMBUD)
  file_goto_next_colon(f);
  long long stated_max_BE_accesses;
  f >> stated_max_BE_accesses;
  max_BE_accesses = stated_max_BE_accesses + memory_budget_add_on;
  assert(max_BE_accesses >= 0);
}

void Task_Scheduler::set_scheduling_algorithm(std::ifstream& f) {
  bool found_algorithm = false;
  file_goto_next_colon(f);
  f >> scheduling_algorithm;
  for (int i = 0,
           num_scheduling_algorithms = sizeof(scheduling_algorithms)/
                                       sizeof(scheduling_algorithms[0]);
       i < num_scheduling_algorithms;
       i++) {
    if (scheduling_algorithm == scheduling_algorithms[i]) {
      found_algorithm = true;
      break;
    }
  }
  if (!found_algorithm) {
    std::cerr << " Couldn't find global scheduler algorithm: "
              << scheduling_algorithm << std::endl;
    exit(EXIT_FAILURE);
  }
}

void Task_Scheduler::set_tasks(std::ifstream& f) {
  number_of_tasks = 0;
  std::streampos pos = f.tellg();
  file_goto_next_newline(f);
  while (f.good() and (f.peek() != EOF)) {
    char c = f.get();
    if (c == 't') { number_of_tasks++; }
    else { break; }
    file_goto_next_newline(f);
  }
  assert(number_of_tasks > 0);
  f.seekg(pos);
  assert(f.good());

  size_t tcbs_size = number_of_tasks*sizeof(tcb *);
  tasks = (tcb **)malloc(tcbs_size);

  assert(tasks);

  for (int i = 0; i < number_of_tasks; i++) { tasks[i] = new tcb(f); }
}

void Task_Scheduler::check_tasks() {
  ms_t test_budget = cpu_budget;
  int max_instances;
  ms_t max_c;
  for (int i = 0; i < number_of_tasks; i++) {
    max_instances = cpu_budget/tasks[i]->get_t();
    max_c = tasks[i]->get_c()*max_instances;
    test_budget -= max_c;
  }
  if (test_budget < ms_t(0)) {
    std::string Bi = "B" + std::to_string(critical_level);
    std::cout << "Warning: sum((" << Bi
              << "/Ti)*Ci) > "    << Bi << std::endl;
  }
}

void Task_Scheduler::store() {
  std::cout << "Critical level: "            << critical_level        << std::endl
            << "Budget: "                    << cpu_budget            << std::endl
            << "Task scheduling algorithm: " << scheduling_algorithm  << std::endl
            << "Max BE accesses: "           << max_BE_accesses       << std::endl;
  for (int i = 0; i < number_of_tasks; i++) { tasks[i]->store(); }
}

Task_Scheduler::Task_Scheduler() {
  selected = false;

  std::cout << "Critical level: ";
  std::cin >> critical_level;
  assert(critical_level > 0);

  std::cout << "Budget: ";
  std::cin >> cpu_budget;
  assert(cpu_budget > ms_t(0));
  cpu_budget_left = cpu_budget;

  std::cout << "Max BE accesses: ";
  std::cin >> max_BE_accesses;
  assert(max_BE_accesses >= 0);

  scheduling_algorithm = scheduling_algorithms[0];
  std::cout << "Task scheduling algorithm: "
            << scheduling_algorithm << std::endl;

  number_of_tasks = ask_how_many("tasks");
  size_t tcbs_size = number_of_tasks*sizeof(tcb *);
  tasks = (tcb **)malloc(tcbs_size);
  assert(tasks);
  for (int i = 0; i < number_of_tasks; i++) { tasks[i] = new tcb(); }
}
```

```
void Task_Scheduler::resupply() {
  cpu_budget_left = cpu_budget;
  memory_budget_left = max_BE_accesses;
}

void Task_Scheduler::schedule() {
  if (scheduling_algorithm == "EDF") { schedule_EDF(); }
  else { std::cerr << "Can't schedule. Unknown algorithm: "
                   << scheduling_algorithm << std::endl;  }
}

tp_t Task_Scheduler::earliest_deadline() {
  assert(hp_task);
  return hp_task->get_rt_c();
}

void Task_Scheduler::schedule_EDF() {
  int ED_task_index;
  std::chrono::system_clock::time_point earliest_deadline;
  std::chrono::system_clock::time_point candidate_deadline;

  bool is_data = false;
  for (int i = 0; i < number_of_tasks; i++) {
    tasks[i]->stop();
    if (tasks[i]->is_employed()) {
      candidate_deadline = tasks[i]->get_rt_c();
      if ((!is_data) or (candidate_deadline < earliest_deadline)) {
        is_data = true;
        earliest_deadline = candidate_deadline;
        ED_task_index = i;
      }
    }
  }
  hp_task = (is_data ? tasks[ED_task_index] : NULL);
}

void Task_Scheduler::run() {
  assert(hp_task);
  selected = true;
}
```

```
bool Task_Scheduler::is_ready() {
  return ((cpu_budget_left > ms_t(0)) and
          hp_task);
}

void Task_Scheduler::check_memory_budget(long long current_BE_accesses,
                                         int fd) {
  static long long last_number_of_BE_accesses = 0;
  if (poll_llc) {
    memory_budget_left -= current_BE_accesses - last_number_of_BE_accesses;
    last_number_of_BE_accesses = current_BE_accesses;
    if (debug and (memory_budget_left < 0) ) {
          std::cout << "Memory budget remaining: "
                    << memory_budget_left
                    << std::endl;
    }
    if (do_freeze_BE_core) {
      if (memory_budget_left < 0) {   freeze_BE_core(fd); }
      else                        { unfreeze_BE_core();   }
    }
  }
}

void Task_Scheduler::tick(ms_t      tick_time,
                          long long current_BE_accesses,
                          int       fd) {
  if (selected) {
    hp_task->run();
    cpu_budget_left -= tick_time;
    check_memory_budget(current_BE_accesses, fd);
  }
  for (int i = 0; i < number_of_tasks; i++) { tasks[i]->tick(tick_time); }
  selected = false;
}

std::ostream& operator<<(std::ostream& os, Task_Scheduler* ts) {
  os << "== scheduler "     << ts->critical_level << " ==" << std::endl
     << "Critical level: "  << ts->critical_level          << std::endl
     << "Max BE accesses: " << ts->max_BE_accesses         << std::endl
     << "CPU-budget: "      << ts->cpu_budget
     << " (left: "          << ts->cpu_budget_left << ")"  << std::endl
     << "Algorithm: "       << ts->scheduling_algorithm    << std::endl
     << "Tasks: "           << ts->number_of_tasks         << std::endl;
  for (int i = 0; i < ts->number_of_tasks; i++) {
    std::cout << ts->tasks[i];
  }
  os << std::endl;
  return os;
}

long long Task_Scheduler::get_max_BE_accesses() { return max_BE_accesses; }
```

C.m tcb

```
#ifndef TCB_HH
#define TCB_HH

#include "sporadic_task.hh"
#include "time_io.hh"

#include <fstream>
#include <ctime>
#include <sys/types.h>

class tcb {
private:
  enum class State { NOT_INIT, UNEMPLOYED, READY,
                     RUNNING,  HOLD,       DELAYED };
  Sporadic_Task *t;
  State s;
  ms_t CPU_time;
  pid_t pid;
  pid_t kill_group_pid;
  std::chrono::system_clock::time_point rt_a, rt_c, rt_d, rt_t;
  void wierd_task_state();
  void stop_process();
  void cont_process();
  void kill_process();
  void usr1_process();
  void extern_process_change_state(State state);
public:
  tcb();
  tcb(std::ifstream& f);
 ~tcb();

  void change_state(State state);

  bool is_employed();

  void tick(ms_t tick_time);
  void tick_running(ms_t tick_time);

  void update_state();
  void check_release();
  void check_period_over();
  void check_delayed();
  void check_done();

  void release();

  std::chrono::system_clock::time_point get_rt_c();

  int get_id() { return t->get_id(); }
  ms_t get_c() { return t->get_c();  }
  ms_t get_d() { return t->get_d();  }
  ms_t get_t() { return t->get_t();  }

  void run();
  void stop();

  void store();

  void print_state(std::ostream& os = std::cout);
  void print_relevant_runtime_data(std::ostream& os = std::cout);
  void print_runtime_data(std::ostream& os = std::cout);
  void print_task_parameters(std::ostream& os = std::cout);
  void print_end_of_period(std::ostream& os = std::cout);
  void print_completed(std::ostream& os = std::cout);
  void print_actual_deadline(std::ostream& os = std::cout);
  void print(std::ostream& os = std::cout);
  friend std::ostream& operator<<(std::ostream& os, tcb* tcb);
};
```

```
#endif

#include "tcb.hh"
#include "global_scheduler.hh"
#include "sporadic_task.hh"
#include "options.hh"
#include "time_io.hh"

#include <iostream>
#include <iomanip>

#include <cstdlib>
#include <cassert>
#include <fstream>
#include <chrono>
#include <thread>

#include <signal.h>
#include <unistd.h>

void tcb::store() { t->store(); }

tcb::~tcb() {
  bool error = false;
  if (fork_processes) {
    if (kill(kill_group_pid, SIGKILL) == -1) {
      std::cerr << "Couldn't send kill signal." << std::endl;
      error = true;
    }
  }
  delete t;
  if (error) { exit(EXIT_FAILURE); }
}

tcb::tcb() {
  t = new Sporadic_Task();
  s = State::UNEMPLOYED; // todo: duplicate
}

tcb::tcb(std::ifstream& f) {
  t = new Sporadic_Task(f);
  s = State::UNEMPLOYED; // todo: duplicate
  if (fork_processes) {
    pid = fork();
    if (pid == -1) {
      std::cerr << "Fork failed." << std::endl;
      exit(EXIT_FAILURE);
    }
    else if (pid ==  0) { t->launch_extern_program(); }
    else if (pid >   0) {
      if (setpgid(pid, pid) == -1) {
        std::cerr << "Couldn't change PGID." << std::endl;
        exit(EXIT_FAILURE);
      }
      kill_group_pid = -1*pid;
      sleep(1); // can't have this
      if (kill(kill_group_pid, SIGSTOP) == -1) {
        std::cerr << "STOP to " << kill_group_pid << " failed." << std::endl;
        exit(EXIT_FAILURE);
      }
    }
  }
  if (debug) { std::cout << "Contruction of TCB done." << std::endl; }
}

bool tcb::is_employed() { return ((s == State::READY) or
                                  (s == State::RUNNING)); }

void tcb::wierd_task_state() {
  std::cerr << "Wierd task state. Go find the bug." << std::endl;
  exit(EXIT_FAILURE);
}

void tcb::cont_process() {
  if (kill(kill_group_pid, SIGCONT) == -1) {
    std::cerr << "Couldn't CONT: " << kill_group_pid << std::endl;
    exit(EXIT_FAILURE);
  }
}

void tcb::stop_process() {
  if (kill(kill_group_pid, SIGSTOP) == -1) {
    std::cerr << "Couldn't STOP: " << kill_group_pid << std::endl;
    exit(EXIT_FAILURE);
  }
}

void tcb::usr1_process() {
  if (kill(kill_group_pid, SIGUSR1) == -1) {
    std::cerr << "Couldn't USR1: " << kill_group_pid << std::endl;
    exit(EXIT_FAILURE);
  }
  std::cerr << "USR1 sent to: " << kill_group_pid << std::endl;
}

void tcb::kill_process() {
  if (kill(kill_group_pid, SIGKILL) == -1) {
    std::cerr << "Couldn't KILL: " << kill_group_pid << std::endl;
    exit(EXIT_FAILURE);
  }
}

void tcb::extern_process_change_state(tcb::State state) {
  switch (state) {
  case State::NOT_INIT:    wierd_task_state();  break;
  case State::READY:
  case State::HOLD:
  case State::UNEMPLOYED:  stop_process();      break;
  case State::RUNNING:     cont_process();      break;
  case State::DELAYED:     kill_process();      break;
  }
}

void tcb::change_state(State state) {
  if (!quiet) {
    std::cout << "task " << get_id() << ":  " << std::setw(10);
    print_state();
  }

  switch (state) {
  case State::NOT_INIT:     wierd_task_state();                 break;
  case State::READY:        assert((s == State::UNEMPLOYED) or
                                   (s == State::RUNNING));      break;
  case State::RUNNING:      assert(s == State::READY);          break;
  case State::HOLD:         assert(s == State::RUNNING);        break;
  case State::UNEMPLOYED:   assert(s == State::HOLD);           break;
  case State::DELAYED:      assert(is_employed());              break;
  }
  s = state;
  if (fork_processes) { extern_process_change_state(s); }
  if (!quiet) {
    std::cout << "  ->  ";
    print_state();
    std::cout << std::endl;
  }
}

/* release is UNEMPLOYED -> READY */
void tcb::release() {
  change_state(State::READY);
  CPU_time = ms_t(0);
  rt_a = std::chrono::system_clock::now();
  rt_c = rt_a + get_c();
  rt_d = rt_a + get_d();
  rt_t = rt_a + get_t();
  if (fork_processes) { usr1_process(); }
}

/* run is READY -> RUNNING */
void tcb::run() { change_state(State::RUNNING); }

/* stop is RUNNING -> READY.
   The reason there is no assertion is it might be
   useful to just tell a bunch of tasks to stop and
   don't bother what state they have, unless for the
   one RUNNING in what case it is READY (i.e.,
   stopped). */
void tcb::stop() {
  if (s == State::RUNNING) { change_state(State::READY); }
}

void tcb::tick(ms_t tick_time) {
  switch (s) {
```

```
  case State::NOT_INIT:
  case State::DELAYED:
  case State::READY:
  case State::HOLD:
  case State::UNEMPLOYED:  break;
  case State::RUNNING:     tick_running(tick_time);  break;
  }
  update_state();
  if (print_task_tick) { std::cout << this; }
}

void tcb::tick_running(ms_t tick_time) {
  assert(s == State::RUNNING);
  t->run(tick_time);
  CPU_time += tick_time;
  check_done();
}

void tcb::update_state() {
  check_release();
  check_period_over();
  check_delayed();
}

/* check_release is UNEMPLOYED -> READY */
void tcb::check_release() {
  if (s == State::UNEMPLOYED) {
    if (random_delay) {
      int r = rand() % 10;
      if (r == 0) { release(); }
    }
    else { release(); }
  }
}

/* check_done is RUNNING -> HOLD */
void tcb::check_done() {
  if (CPU_time >= get_c()) { change_state(State::HOLD); }
}

/* check_period_over is HOLD -> UNEMPLOYED */
```

```
void tcb::check_period_over() {
  if (s == State::HOLD) {
    if (std::chrono::system_clock::now() >= rt_t) {
      change_state(State::UNEMPLOYED);
    }
  }
}

/* only a READY or RUNNING task can be DELAYED.
   HOLD is as "unemployed" as UNEMPLOYED */
void tcb::check_delayed() {
  if (is_employed()) {
    if (std::chrono::system_clock::now() >= rt_d) {
      change_state(State::DELAYED);
      if (not really_quiet) { std::cout << "Oh, no! Delayed task!" << std::endl; }
      if (fork_processes) { kill_process(); }
      if (hard) { crash = true; }
    }
  }
}

void tcb::print_state(std::ostream& os) {
  switch(s) {
  case State::UNEMPLOYED:  os << "unemployed";  break;
  case State::NOT_INIT:    os << "not_init";    break;
  case State::HOLD:        os << "hold";        break;
  case State::READY:       os << "ready";       break;
  case State::RUNNING:     os << "running";     break;
  case State::DELAYED:     os << "delayed";     break;
  }
}

void tcb::print_runtime_data(std::ostream& os) {
  print_completed(os);
  print_actual_deadline(os);
  print_end_of_period(os);
}

void tcb::print_end_of_period(std::ostream& os) {
  os << "Earliest possible next instance: " << rt_t << std::endl;
}
```

```
void tcb::print_completed(std::ostream& os) {
  os << "Running time (CPU-time) so far: " << CPU_time << std::endl;
}

void tcb::print_actual_deadline(std::ostream& os) {
  os << "Actual deadline: " << rt_d << std::endl;
}

void tcb::print_relevant_runtime_data(std::ostream& os) {
  switch (s) {
  case State::NOT_INIT:
  case State::UNEMPLOYED:  break;
  case State::HOLD:        print_end_of_period(os);    break;
  case State::DELAYED:
  case State::READY:
  case State::RUNNING:     print_completed(os);
                           print_actual_deadline(os);  break;
  }
}

void tcb::print_task_parameters(std::ostream& os) {
  if (s != State::NOT_INIT) { os << t << std::endl; }
  else { std::cerr << "Not initialized: " << this << std::endl; }
}

void tcb::print(std::ostream& os) {
  if (!quiet) {
    os << "--- task " << get_id() << " ---" << std::endl;
    print_task_parameters(os);
    print_state(os);
    print_relevant_runtime_data(os);
    std::cout << std::endl;
  }
}

std::ostream& operator<<(std::ostream& os, tcb* tcb) {
  tcb->print(os);
  return os;
}
```

```
tp_t tcb::get_rt_c() { return rt_c; }
```

C.n time_io

```
#ifndef TIME_HH
#define TIME_HH

#include <iostream>
#include <chrono>

typedef std::chrono::system_clock::time_point  tp_t;
typedef std::chrono::milliseconds              ms_t;

std::ostream& operator<<(std::ostream& os, tp_t tp);
std::ostream& operator<<(std::ostream& os, ms_t msd);
std::istream& operator>>(std::istream& is, ms_t& msd);

#endif

#include "time_io.hh"
#include <iostream>
#include <chrono>

std::ostream& operator<<(std::ostream& os, tp_t tp) {
  os << std::chrono::system_clock::to_time_t(tp);
  return os;
}

std::ostream& operator<<(std::ostream& os, ms_t msd) {
  os << std::chrono::duration_cast<ms_t>(msd).count();
  return os;
}

std::istream& operator>>(std::istream& is, ms_t& msd) {
  int ms;
  is >> ms;
  msd = std::chrono::duration<int, std::milli>(ms);
  return is;
}
```

D Appendix D: experiment code and example data

D.a example hs task system

```
Global scheduling rate: 1
Global period : 10
Global lifetime: 20000
Global scheduling algorithm: EDF

Critical level: 1
Budget: 5
Max BE accesses: 1000
Task scheduling algorithm: EDF
t1 = (5, 10, 10) helloworld()

Critical level: 2
Budget: 5
Max BE accesses: 7050
Task scheduling algorithm: EDF
t2 = (5, 10, 10) helloworld()
```

D.b useful zsh commands

```
#! /bin/zsh

# here are some useful zsh commands
# that will facilitate running experiments
# using the system described in this report

# misc
now-ns    () { echo $((`date +%s`*10**9 + `date +%N`)) }
get-group () { awk '{print $5}' < /proc/$1/stat        }

# this is cgroup 0
# a cgroup is a set of processes
# that can be stopped and resumed collectively
BE_PATH=/sys/fs/cgroup/freezer/0   # the cgroup path
STATE_FILE=$BE_PATH/freezer.state  # and state: either FROZEN or THAWED
TASK_FILE=$BE_PATH/tasks           # PIDs of processes included

# if cgroup 0 has not been created, use this to create it:
# the permissions are set so that you don't need root
# either to add/remove a process to the group,
# nor to have it stopped or resumed
be-init () {
    if [[ ! -d $BE_PATH ]]; then
       su -c "mkdir $BE_PATH; chmod a=wr $STATE_FILE $TASK_FILE"
    fi
    be-start
}

# manually set the state of the cgroup:
# normally, this will not be done
# but for debugging
be-stop  () { echo FROZEN > $STATE_FILE }
be-start () { echo THAWED > $STATE_FILE }

# this runs a program on BE, and
# adds the PID to the cgroup
be-task () {
    taskset -c $BE_CORE $@ &
    local pid=$!
    echo $pid >> $TASK_FILE
}

# play a multimedia file with mplayer2(1) on BE
be-mp () {
    be-task mplayer --ao=null --loop=0 -noconsolecontrols -fs \
            $1 &> /dev/null
}

# memory stress(1) with N spinners of malloc(3)/free(3)
# on either core
cc-stress      () { stress --vm $1 -t 60 }
cont-be-stress () { while (true) { be-stress 10 10 } }
be-stress () {
    local amp=$1
    local time=$2
    local  io=$((4*$amp))
    local  vm=$((2*$amp))
    local vmb=$((128*2**((log($amp)/log(2)))))M
    be-task stress --io $io         \
                   --vm $vm         \
                   --vm-bytes $vmb  \
                   --timeout $time  > /dev/null
}

be-movie () { be-mp ~/box/GGG_Rubio.mp4 }

be-kill () { killall -9 $@ 2> /dev/null }

#! /bin/zsh

## ~/.zsh/be-task

ps-core () {
    local core=$1
    ps -eo psr,pid,stat,comm | grep "^ $core "
}
ps-be () { ps-core       $BE_CORE }
ps-cc () { ps-core $CRITICAL_CORE }

monitor-processes () {
    local core=$1
    watch -t -n 0.1 "ps -eo psr,pid,stat,comm | grep '^  $core '"
}
monitor-be        () { monitor-processes       $BE_CORE }
monitor-cc        () { monitor-processes $CRITICAL_CORE }

monitor-be-stress () {
    watch -t -n 0.1 "ps -eo psr,pid,comm | grep -e '^  $BE_CORE ' | grep stress"
}

monitor-be-status () {
    be-init
    watch -t -n 0.1 cat $STATE_FILE $TASK_FILE
}

monitor-tty () { watch -t -n 0.1 "ps -eo tty,pid,comm,state | grep 'pts/$1 '" }
alias monitor=monitor-tty

#!/bin/zsh

MEM_EXP_RES=~/public_html/hs-linux/results/memory-experiment-tables.log

LISP_PATH=~/.emacs.d/emacs-init-cp

PROJECT_PATH=~/public_html/hs-linux
SRC_PATH=$PROJECT_PATH/src
RESULTS_PATH=$PROJECT_PATH/results

CONTENTION_RESULT=$RESULTS_PATH/cr.log

EXPERIMENT_PATH=$SRC_PATH/experiment
EXPERIMENT_SIGNAL_PATH=$EXPERIMENT_PATH/signal
EXPERIMENT_STATUS_FILE=$EXPERIMENT_PATH/status.txt

AUDIO_OUTPUT_PATH=$EXPERIMENT_PATH/audio-output

BE_PATH=/sys/fs/cgroup/freezer/0
STATE_FILE=$BE_PATH/freezer.state
TASK_FILE=$BE_PATH/tasks

#! /bin/zsh
```

```
source ${0:h}/be-paths # ./be-paths

count-lines () {
    if [[ $1 == "--clear" ]]; then
        local dir=$2
        for f in $dir/*(N); do
            sudo chown $USER $f
            echo -n > $f
        done
    else
        local dir=$1
        local file=$2
        local file_path=$dir/$file
        watch -t -n 0.1 "wc -l $file_path | cut -d\" \" -f 1; echo -n $file"
    fi
}

clear-signals () { count-lines --clear $EXPERIMENT_SIGNAL_PATH  }
clear-outputs () { count-lines --clear      $AUDIO_OUTPUT_PATH  }

count-signals () { count-lines $EXPERIMENT_SIGNAL_PATH $1 }
count-outputs () { count-lines      $AUDIO_OUTPUT_PATH $1 }

count-signals-no-be         () { count-signals no_interference  }
count-signals-freeze        () { count-signals throttle         }
count-signals-do-not-freeze () { count-signals do_not_throttle  }

count-outputs-no-be         () { count-outputs no_interference  }
count-outputs-freeze        () { count-outputs throttle         }
count-outputs-do-not-freeze () { count-outputs do_not_throttle  }
```

D.c zsh wrapper to run experiments

```
#! /bin/zsh

## other files:
##   ~/public_html/hs-linux/src/times.el            [1]
##   ~/public_html/hs-linux/docs/report/report.tex  [2]

##   ~/public_html/hs-linux/src/compile_lisp
##   ~/public_html/hs-linux/src/forever
##   ~/scripts/a-level

source ${0:h}/be-paths # ./be-paths

monitor-experiment () { watch -t -n 0.1 "cat $EXPERIMENT_STATUS_FILE" }

## this crunches the tick readings into stats using [1]
## see: "the Linux and C++ clocks" in [2]
do-tick-stats () {
    emacs -Q \
          --insert $1 \
          --script $SRC_PATH/times.el \
          --eval "(tick-stats (* 1000000 $2))"
}

## for the -l switch to work
## hs must have root as owner
## and the SUID bit set (gulp)
## i.e.:  sudo chown root hs
##        sudo chmod +s hs
## it also works to use sudo here
## but if so do sudo warm-up
## so not to type the password
## thus affecting the experiment outcome
HS_OUTPUT=hs_output.log
run-hs   () { $SRC_PATH/hs -s $1 -r -l -f    > $RESULTS_PATH/$1/$HS_OUTPUT }
run-hs-p () { $SRC_PATH/hs -s $1 -r -l -f -P > $RESULTS_PATH/$1/$HS_OUTPUT }

# run hs with SCHED_FIFO -
# see "Linux real time schedulers" in [2]
# for this to work without sudo:
```

```
#   sudo chmod +s /usr/bin/chrt
# (this hasn't been used in the experiments so far)
run-hs-rt () { chrt --fifo 99 ./hs $@ }

run-all-experiments () {
    run-contention-experiment     # ~/.zsh/be-contention-experiment
    run-memory-experiment         # ~/.zsh/be-memory-experiment
    run-task-system-experiment    # ~/.zsh/be-system-experiment
    run-linux-process-experiment  # ~/.zsh/be-process-experiment
    run-audio-experiment 10       # ~/.zsh/be-audio-experiment
}

#! /bin/zsh

source ${0:h}/be-paths

clear-exp-data () {
    echo "no execution" > $EXPERIMENT_STATUS_FILE
    clear-signals
    clear-outputs
}

setup-panes () {
    clear-exp-data

    tmux split-window -h
    tmux split-window -v
    tmux split-window -v
    tmux split-window -h
    tmux select-pane  -U
    tmux split-window -h
    tmux select-pane  -U
    tmux split-window -v
    tmux split-window -h
    tmux select-pane  -U
    tmux split-window -h

    tmux send-keys -t 1 "monitor-experiment" Enter
    tmux send-keys -t 2 "monitor-be-status" Enter

    tmux send-keys -t 3 "count-signals-no-be" Enter
    tmux send-keys -t 5 "count-signals-freeze" Enter
    tmux send-keys -t 7 "count-signals-do-not-freeze" Enter

    tmux send-keys -t 4 "count-outputs-no-be" Enter
    tmux send-keys -t 6 "count-outputs-freeze" Enter
    tmux send-keys -t 8 "count-outputs-do-not-freeze" Enter

    tmux send-keys -t 9 "monitor-be-stress" Enter

    tmux select-pane -t 0
    cd $SRC_PATH
}

#! /bin/zsh

source ${0:h}/be-paths # ./be-paths

now-ns () { echo $((`date +%s`*10**9 + `date +%N`)) }

clean-elisp () { make -C $LISP_PATH clean > /dev/null }

time-elisp () {
    (($#)) || set ./process_lisp_time

    clean-elisp

    local start=`now-ns`
    /usr/bin/make -C $LISP_PATH > /dev/null
    local done=`now-ns`

    local nanos=$(($done - $start))
    echo "\t" $nanos "\n" >> $1
}

#!/bin/zsh

source ${0:h}/be-paths # ./be-paths

ce-hs () { hs -s base -r -Q }

ce-time-elisp () { time-elisp $CONTENTION_RESULT }
```

```
ce-echo () { echo $@ "\n" >> $CONTENTION_RESULT }

run-contention-experiment () {
    rm -f $CONTENTION_RESULT
    be-init

    echo "Be patient..."
    ce-echo "* (compilation time without hs)"
    ce-time-elisp

    ce-echo "* with hs:"
    ce-hs &
    ce-time-elisp
    be-kill hs

    ce-echo "* with hs and the best-effort media player:"
    ce-hs &
    be-movie &
    ce-time-elisp
    be-kill mplayer hs

    ce-echo "* with hs and stress:"
    local system=$PROJECT_PATH/sys/base
    local system_time_ms=`grep "Global lifetime:" $system | cut -d" " -f3`
    local system_time_s=$(($system_time_ms/1000))
    ce-hs &
    be-stress 10 $system_time_s &
    ce-time-elisp
    be-kill stress hs

    ce-echo "* with hs, stress and the BE media player:"
    ce-hs &
    be-movie &
    be-stress 10 $system_time_s &
    ce-time-elisp
    be-kill stress mplayer hs
}

#! /bin/zsh
# systems:  ~/public_html/hs-linux/sys/base-faculty-1
#           ~/public_html/hs-linux/sys/base-faculty-2
# results:  ~/public_html/hs-linux/results/memory-experiment-tables.log

source ${0:h}/be-paths # ./be-paths

echo-and-file () { echo $1 | tee -a $MEM_EXP_RES }

exit-memory-experiment () { be-start; be-kill mplayer }

run-memory-experiment () {
    rm -f $MEM_EXP_RES
    be-init
    be-start
    be-movie &
    echo-and-file "\\\tiny\n"
    for system in base-faculty-{1,2}; do
        echo-and-file "\\\subsubsection{\\\texttt{$system}}\n"
        echo-and-file "\\\begin{tabular}{l l l}"
        echo-and-file "supposed & actual & error ratio \\\\\\ \\\hline"
        for budget in {1000000..1..-10000}; do
            hs -s $system -i -h -r -Q -f -x -m $budget \
                2> /dev/null | tee -a $MEM_EXP_RES
            if [[ $? == 255 ]]; then exit-memory-experiment; return; fi
        done
        echo-and-file "\\\end{tabular}\n"
    done
    echo-and-file "\\\normalsize\n"
    exit-memory-experiment
}

#! /bin/zsh

source ${0:h}/be-paths # ./be-paths

run-experiment () {
    # result dir
    local name=$1
    local system=$PROJECT_PATH/sys/$name
    local this_result_path=$RESULTS_PATH/$name
    rm -rf $this_result_path
    mkdir -p $this_result_path
    # Lisp file
    local lisp_file=$this_result_path/lisp.log

    # to-be stats files
    local tick_times_file=tick_times.log
    local stats_file=stats.log
    rm -f $stats_file $tick_times_file

    #### BE
    be-init
    be-movie &

    #### critical core
    time-elisp $lisp_file &
    if [[ $# == 2 ]]; then run-hs-p  $name
    else                   run-hs    $name
    fi

    # store results
    local global_tick=`grep "Global scheduling rate:" $system | cut -d" " -f4
    do-tick-stats $tick_times_file $global_tick
    mv $tick_times_file $stats_file $this_result_path
    tail -n 1 $this_result_path/$HS_OUTPUT | \
        cut -d ' ' -f 2 > $this_result_path/BE.txt
    head -n 6 $this_result_path/stats.log > $this_result_path/stats.txt

    # clean
    be-kill mplayer
    be-start
}

run-task-system-experiment () {
    # system:   ~/public_html/hs-linux/sys/base
    # results:  ~/public_html/hs-linux/results/base
    run-experiment base

    # ~/public_html/hs-linux/sys/long-ticks
    # ~/public_html/hs-linux/results/long-ticks
    run-experiment long-ticks
    # ~/public_html/hs-linux/sys/short-period
    # ~/public_html/hs-linux/results/short-period
    run-experiment long-period
}

#! /bin/zsh

run-linux-process-experiment () {
    # system:   ~/public_html/hs-linux/sys/base-p
    # results:  ~/public_html/hs-linux/results/base-p
    run-experiment base-p t

    # system:   ~/public_html/hs-linux/sys/base-p-no-memory-budget
    # results:  ~/public_html/hs-linux/results/base-p-no-memory-budget
    run-experiment base-p-no-memory-budget t
}

#! /bin/zsh

source ${0:h}/be-paths # ./be-paths

run-audio-experiment () {
    # $1 is iterations per system (default: 1)
    (($#)) || set 1
    local its=$1

    cd $SRC_PATH

    clear-exp-data
    be-init

    local ni=$EXPERIMENT_SIGNAL_PATH/no_interference
    local dnt=$EXPERIMENT_SIGNAL_PATH/do_not_throttle
    local dt=$EXPERIMENT_SIGNAL_PATH/throttle
    echo -n > $ni > $dnt > $dt

    # ~/public_html/hs-linux/sys/base-audio-2
    local system=base-audio-2
    local system_file=$PROJECT_PATH/sys/$system

    export AUDIO_OUTPUT=$AUDIO_OUTPUT_PATH/no_interference
    for i in {1..$its}; do
        echo "no BE ($i of $its)" > $EXPERIMENT_STATUS_FILE
        $SRC_PATH/hs -s $system -i -h -P -r -Q -p  2>> $ni
    done

    local sys_time_ms=`grep "Global lifetime:" $system_file | cut -d" " -f3`
    local sys_time=$(($sys_time_ms*$its/1000))
    local stress_time=$(($sys_time*3))
    local stress_its=10
    local stress_factor=1

    repeat $stress_its be-stress $stress_factor $stress_time &
    export AUDIO_OUTPUT=$AUDIO_OUTPUT_PATH/throttle
    for i in {1..$its}; do
        echo "tamed BE ($i of $its)" > $EXPERIMENT_STATUS_FILE
        $SRC_PATH/hs -s $system -i -h -P -r -Q -f  2>> $dt
    done
    killall -9 stress

    repeat $stress_its be-stress $stress_factor $stress_time &
    be-start
    export AUDIO_OUTPUT=$AUDIO_OUTPUT_PATH/do_not_throttle
    for i in {1..$its}; do
        echo "wild BE ($i of $its)" > $EXPERIMENT_STATUS_FILE
        $SRC_PATH/hs -s $system -i -h -P -r -Q -p  2>> $dnt
    done
    killall -9 stress

    echo "terminated" > $EXPERIMENT_STATUS_FILE

    local result_file_name=audio_results
    local result_file=${result_file_name}.png
    dumpx $result_file_name
    local inverted_file=${result_file_name}_i.png
    convert -negate $result_file $inverted_file
    mv -f $inverted_file ~/public_html/hs-linux/docs/report/pics/
    rm $result_file
}
```

D.d tick trace cruncher in Elisp

```
;;; -*- lexical-binding: t -*-

;; for a buffer/file of tick-time integers
;; one at each line
;; this Elisp number cruncher
;; is used to process the tick times
;; in order to find out how much
;; the ticks deviated
;; from the desired, fixed-interrupt rate

;; for the particular trace
;; stats are presented
;; as well as every drift from the specified ideal
;; as an indicator of an inexact clock
;; or if this needs to be further analyzed

(require 'cl-macs)

(defun get-variance (mean)
  (save-excursion
    (goto-char 1)
    (let ((sum 0) (offsets 0))
      (cl-loop
       (let ((offset (thing-at-point 'number)))
         (if offset
             (progn
               (cl-incf offsets)
               (setq sum (+ sum (expt (- offset mean) 2)))
               (forward-line 1) )
           (cl-return) )))
      (/ sum offsets) )))

(defun tick-stats (desired-tick)
  (interactive "n desired tick: ")
  (save-excursion
    (goto-char 1)
    (let ((sum 0) (offsets 0) (max nil) (min nil) )
      (cl-loop
       (let((t0 (thing-at-point 'number)))
         (forward-line 1)
         (let((t1 (thing-at-point 'number)))
           (save-current-buffer
             (set-buffer (get-buffer-create "offsets.log"))
             (if t1
                 (let((offset (- t1 t0 desired-tick)))
                   (cl-incf offsets)
                   (setq sum (+ sum offset))
                   (if (or (not max) (> offset max)) (setq max offset))
                   (if (or (not min) (> min offset)) (setq min offset))
                   (insert (format "%d\n" offset)) )
               (progn
                 (goto-char 1)
                 (insert
                  (let*((mean      (/ sum offsets))
                        (variance  (get-variance mean)) )
                    (format
                     "readings: %d\nmean: %f\nvariance: %f\nstandard deviatio
                     offsets mean variance (sqrt variance) min max)))
                 (write-file "stats.log")
                 (cl-return) )))))))))
```

Index