Execution Time Monitoring in Linux*

Mikael Åsberg, Thomas Nolte
MRTC/Mälardalen University
P.O. Box 883, SE-721 23,
Västerås, Sweden
mikael.asberg@mdh.se

Clara M. Otero Pérez
NXP Semiconductors/Research
High Tech Campus 32 (102)
5656 AE Eindhoven, Holland
clara.otero.perez@nxp.com

Shinpei Kato
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku,
Tokyo 113-8656, Japan
shinpei@rt.k2.keio.ac.jp

Abstract

This paper presents an implementation of an Execution Time Monitor (ETM) which can be applied in a resource management framework, such as the one proposed in the Open Media Platform (OMP) [4]. OMP is a European project which aims at creating an open, flexible and resource efficient software architecture for mobile devices such as cell phones and handsets. One of its goals is to open up the possibility for software portability and fast integration of applications, in order to decrease development costs. The task of the ETM is to measure task execution time and provide this information to the scheduler which then can schedule tasks in a more efficient and dynamic way. This implementation is our first step towards a full resource management framework that later will include a hierarchical scheduler, for soft real-time systems.

1 Introduction

Our previous work includes, among others, a synchronization protocol for hierarchical scheduling [9] and the implementation of a two-level Hierarchical Scheduling Framework (HSF) in VxWorks [8]. This paper presents a continuation of our research and aims towards HSF in soft real-time scheduling, such as media applications. HSF provides temporal isolation among concurrent executing applications. This isolation enables guaranteed resource availability for multiple media applications in the context of open platforms, such as the one targeted by the Open Media Platform (OMP) [4]. In this project, media components encapsulating functionality (such as video/audio decoders) have the capability to trade Quality Of Service (QoS) for resource usage, e.g., memory or CPU resources. The aim of the OMP project is to standardize the interfaces, used to adapt QoS, to achieve portability, reusability and easy integration. In particular, to trade resources, the Resource Management Framework (RMF) offers an interface to allocate a guaranteed amount of resource (e.g., CPU) to components, based on the component resource estimate and the current system state. For example, if only one component is active, then it can run at the maximum QoS using 75% of a given resource. However, if a second identical component is started, then both components will have to adapt to a lower QoS where each of them uses 50% of the same resource.

In media processing systems, the load is very dynamic and the worst case is often unknown (unpredictable) or too high to do worst case resource allocation (since it will imply wasting resources). The initial resource estimate provided by the component is used to create the initial allocation of resources. However, this initial allocation has to be monitored, at run time, to make sure that the allocation matches the actual resource utilization. For example, the estimated processor cycles of decoding one frame of an H264 stream, measured in a given platform, can vary more than 5 times depending on the video content (and type of frame, number of reference frames .etc). The RMF keeps track of the actual usage (with the use of an ETM) and signals an alarm when a component consistently under or over utilizes its initial allocation. This alarm mechanism enables the upper software layers to learn and adapt the resource allocation to the real need. The aim of the ETM is to measure CPU resource utilization and give this information to the RMF, in order for it to conduct resource allocation adjustments during runtime. The implementation of the ETM is done in Linux because it has a high degree of functionality (an abundance of libraries and device drivers) in embedded systems and mobile phones [2].

Resource management The **Resource manager** (Figure 1) is in charge of allocating/deallocating resources to application components. It also provides admission control (checking feasibility before allocating requested resources) and enforcement to ensure that the allocated resources are also guaranteed. The resource manager has an overview of the system resources, whereas for every shared resource, the corresponding **Resource controller** implements the resource allocation and monitoring mechanisms for that resource. For each resource, the resource controller keeps track of the allocation of resources (**Resource budget**), its users (**Resource users** which is basically components) and the runtime usage (**Resource**

*The work in this paper is supported by the Swedish Foundation for Strategic Research (SSF), via the research programme PROGRESS.

usage) of a given resource user on a given resource budget.

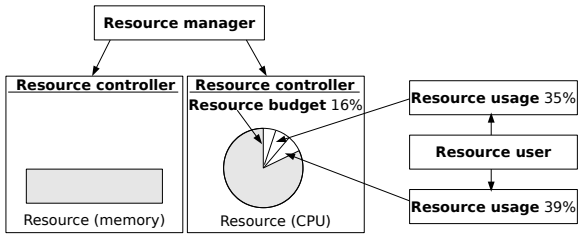

Figure 1. Resource management

CPU resource allocation The CPU resource controller is realized with a HSF, illustrated in Figure 2. Our next step is to implement this scheduler with minimum modification on the operating system, similar to the implementation in [8]. The $Global\ scheduler$ will schedule the $Budgets$ according to their allocation ($Time\ interface$) and an arbitrary scheduling scheme. $Tasks$ are scheduled within the budget according to the scheduling strategy of the $Local\ scheduler$. The global scheduler schedules the budgets periodically, the budget runs according to a predefined amount of time and the order of execution of the budgets are decided by the budget priorities. The admission control checks whether the current configuration of budgets are schedulable. This check is done if new budgets are activated or if running budgets need to change parameters in their time interface. Tasks within a budget are assumed to be schedulable, with respect to the budget time interface. Parameters in the budget time interface such as period, execution time and priority may be dynamic, e.g., they may change during runtime. Unused budget may be allocated to tasks belonging to other budgets (weak enforcement).

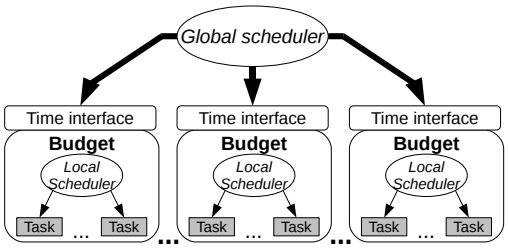

Figure 2. Hierarchical scheduling framework

Scheduling classes in Linux Starting from Linux kernel version 2.6.23 (released in October of 2007) [6], there is a distinction between $real-time$ and $regular$ tasks. This is done by the introduction of two scheduling classes: real-time and fair scheduling ($sched\_rt.c$ and $sched\_fair.c$). Basically, a Linux process is either $regular$ or $real-time$ depending on which scheduling policy that it is configured with. The processes which are $real-time$ have higher priority (0-99 where 0 is the highest) than $regular$ processes. Whenever there is a scheduling event (time-slice expiration, task suspend/sleep, etc.), the CPU (for which the scheduling event belongs to) runqueue (ready-queue) is fetched. The Linux core scheduler then iterates trough a chain of scheduling classes (Figure 3), where each of these classes tries to fetch the next running task. Eventually, idle class will pick a next task to run if no higher class has succeeded with this. All scheduling classes share the same interface (set of functions), this makes it is easy to add new scheduling classes without changing the core scheduler.

```
class = sched_class_highest   /* The latter is a global kernel variable,
                                 1st: real-time scheduling class. */
WHILE (TRUE)
    p = class.pick_next_task(run_queue)
    IF(p)
      RETURN p
    ENDIF
    class = class.next   /* 2nd: fair scheduling class. */
ENDWHILE
```

Figure 3. Sample pseudo-code of function **pick_next_task**, which is part of the Linux scheduler ($sched.c$)

The outline of the paper is as follows: Section 2 presents related work. In Section 3 we present the implementation of our ETM in Linux and discuss how it can be applied. Finally, Section 4 concludes.

2 Related work

The AQuoSA framework [11] is a RMF based on CBS scheduling with advanced adaptive resource reservation. The framework is built on a patch which exports appropriate scheduling hooks.

In [12], the authors present a RMF consisting of server based scheduling of tasks and a predictable disk scheduling mechanism. CPU usage monitoring is used within the framework to calibrate application resource usage.

RTAI [7] is a collection of loadable modules that provides a rich real-time Application Programming Interface (API) to the user, trough the usage of a hardware abstraction layer named ADEOS [1]. The RTAI API includes add/delete hooks for every task start, switch and delete.

RT-Linux [5] is a patch to the Linux kernel and introduces a layer between the OS and the hardware. Linux is scheduled as low priority task while real-time tasks have higher priority (similar to [7]).

Related to CPU allocation, $cgroups$ [3] is a Linux patch which partitions processes into groups and give them a specified share of the processor. Each partition is scheduled according to its period and will run a predefined amount of time. All groups must have the same period.

3 Execution time monitoring

The ETM provides the resource manager with information, in order for it to analyze the suitability and to fit a resource budget for a given resource user(s). Most monitors are time-stamp based with a coarse time-base, e.g., time consumed by a task during the past 5 seconds.

In this case we get an average value, without knowledge of whether the time was consumed from its own budget or from a different one. To accurately fit a periodic budget we need to monitor resource usage per scheduling event, e.g., how much of a given budget allocation a task consumed during the past budget period. This, so called $event\ based\ monitoring$, also enables the possibility that a task can keep track of its own progress. A decoder that has decoded half of a frame can make a request to the monitor of how much budget that is left for the period. In this way, it can derive whether it is able to decode the remainder of the frame within this budget period.

Implementation The Linux operating system offers, to the best of our knowledge, only the $proc\ file-system$ and $taskstats/cgroupstats$ [6] as a way to monitor process/thread information. However, these approaches are not suitable for our application for several reasons. It requires additional interrupt-routines/tasks (extra overhead). Alternatively, it must be fully integrated with the HSF. However, this is not a modular solution. Also, the presented solutions [7, 13] in Section 2 require to much kernel modifications. We have chosen to implement our ETM as a task switch hook interface (event based monitoring) that can be used in kernel space. The hook is implemented in a kernel module (similar to our work in [10]) which makes it easy to modify (without the need to modify and re-compile the kernel). It is just a matter of loading the module into the kernel. Our interface is simply a function called **task_switch_hook** which is called at every scheduler tick. The supplied information is the previous and next running tasks thread identification (TID) and a flag, informing whether the task switch is between a $regular$ and $real-time$ task (or vice versa) or not.

The task switch hook implementation requires a minimum patch, simply by removing the `const` statement from each of the two data-structures **fair_sched_class** and **rt_sched_class** (Figure 4). Security and kernel stability is not considered in this implementation.

```
static ~~const~~ struct sched_class rt_sched_class = {
.
.
.
    .pick_next_task          = pick_next_task_rt,
```

Figure 4. Sample of scheduling class **rt_sched_class** ($sched\_rt.c$)

The reason for removing the read-only access for these two data-structures is because we can then reassign the **.pick_next_task** (Figure 4), of a scheduling class, to point to our own defined function (Figure 5). The **.pick_next_task** (in **fair_sched_class** and **rt_sched_class**) are called by the scheduler (Figure 3) whenever there should be a task context switch (at least at those occasions). This implies that our function will also be called. The next step is to define **hook_rt** (Figure 6) and **hook_fair**.

```
org_pick_next_task_rt = sched_class_highest.pick_next_task
sched_class_highest.pick_next_task = hook_rt
```

Figure 5. Redirection of **.pick_next_task** in **rt_sched_class**

```
FUNCTION hook_rt(run_queue)
    prev_task = current   /* The latter is a global kernel variable. */
    next_task = org_pick_next_task_rt(run_queue)
    task_switch_hook(prev_task, next_task, HOOK_RT)
    RETURN next_task
ENDFUNCTION
```

Figure 6. Function **hook_rt**

Applicability of the ETM This chapter gives a proposal of how the ETM can be used and also compares the two following approaches to monitor execution time in Linux: 1) having tasks or interrupts reading total execution time of tasks from the kernel ($proc\ file-system$ or $taskstats/cgroupstats$) or, 2) by using task switch hooks and timestamp task start and end. Note that approach 2) could alternatively be integrated in the HSF, which would not require it to have tasks or interrupts. However, as mentioned previously, it is more modular to keep the monitor and scheduler separate.

The main task of the ETM is to measure the execution time of a group of tasks within time intervals. The intervals could be the Least Common Multiplier (LCM) of all task periods within a budget, defined as LCM_{budget}. This is illustrated in Figure 7, where the LCM_{budget} of the tasks within the budget is 120. If tasks are fixed within a budget, then LCM_{budget} can be computed off-line.

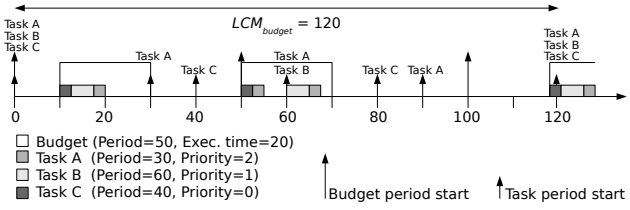

Figure 7. Tasks executing within a budget

In this way, the CPU utilization of the tasks within a budget could be derived and compared against the resource budget periodically (at each LCM_{budget}). With approach 1), it would require either to have one or several tasks or interrupt handlers running periodically (at end and start of all budgets LCM_{budget}) and reading the task execution time counters. In the example in Figure 7, the task/handler would run at time 0 and 120 and log the tasks execution time counters at those two events. With approach 2), if we assume that **pick_next_task** (Figure 3) is executed whenever the Linux scheduler is invoked, even if there is no task context switch, then our monitor will

run at the precise moments when LCM_{budget} starts and ends. This assumption is valid since the scheduler checks whether **pick_next_task** has chosen another task (other than the current running task) to run (according to the Linux scheduler function **schedule()** in $sched.c$). In the example in Figure 7, monitoring the tasks would start (and end) when the hook is executed at period start of either task A, B or C at time 0 and 120 (our hook will notice this by checking the status of the run-queue). The advantage with approach 2) is that it does not add to extra context switch overhead (besides the execution of the monitor) since the task context switch overhead would exist even if we did not use the hook. Also, the monitor is executed more precisely (than approach 2 which would have to rely on a periodic timer) at the moments when the measuring periods start and end.

If the global scheduler uses weak enforcement then any unused budget will be given to background tasks (belonging to other budgets). This approach increases CPU utilization. The problem here is that approach 1) would consider task execution time within other budgets (Not Allocated Consumed (NAC)) as execution time within its own budget (Allocated Consumed (AC)), since the total task execution time counter cannot know the difference in which context the task executes. Approach 2) would consider in which context the task executes in. At each hook execution instance, it could check the Task Control Block (TCB) or the global scheduler to see if a task is within NAC or AC.

The motivation for monitoring AC and NAC time is that it reveals not only if the budget supply is sufficient enough but also, if budget execution time is properly distributed. A task may use NAC time if it has insufficient budget supply and/or if the budget execution time distribution is insufficient (task is mostly active when there is no budget execution time available). This is illustrated in Figure 8 where example **a)** shows that the amount of budget supply is sufficient with respect to the task demand, but the budget execution time distribution is inadequate. In **b)**, clearly the budget supply is insufficient, that is why the task uses NAC time.

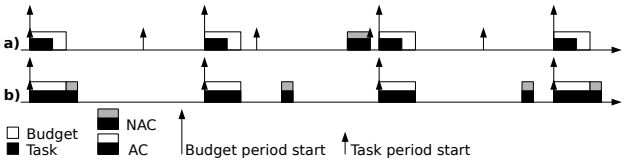

Figure 8. NAC and AC execution time

4 Summary

Task execution monitoring in Linux is challenging due to its lack of real-time support. Most real-time operating systems support the task switch hook mechanism, which is a popular method for monitoring task execution time. With respect to a Resource Management Framework (RMF), such as the one proposed in Open Media Platform (OMP) [4], a monitor should give as exact results as possible and it should also support task execution monitoring within different contexts. For example, a task may be executing within its own budget or in other budgets (depending on the enforcement of the global scheduler). Also, the monitor should be as independent as possible with respect to the Hierarchical Scheduling Framework (HSF) in order the have a modular design. We have motivated why our solution (implementation of the task switch hook mechanism) is best fitted for monitoring in a RMF in Linux. Implementing task switch hooks seems to impose less overhead in comparison with reading task execution information from the kernel. This makes it interesting (and will be part of our future work) to perform an evaluation of our solution together with, for example, $proc\ file-system$ and $taskstats/cgroupstats$ [6] solutions.

Future work will include the implementation of a HSF in Linux. The task switch hook implementation presented in this paper can be useful in that it can suppress the Linux scheduler functionality (because we are in control of the actual task switch). The goal however, is to implement a hierarchical scheduler in Linux as a middleware (similar to [8]) without any patches of the kernel. This requirement is inherent from embedded systems, which have higher reliability and stability requirements and they prefer to use proven versions of the Linux kernel. None of the techniques/solutions presented in Section 2 [5, 7, 11, 12] fulfills this requirement except for using $cgroups$ [3]. The final step is to perform an evaluation of our HSF and preferably compare it against other similar schedulers [11, 12] as well as native Linux mechanisms such as $cgroups$ [3].

References

[1] Adeos. `http://home.gna.org/adeos`.

[2] Limo Foundation. `http://www.limofoundation.org`.

[3] LXR. `http://lxr.linux.no`.

[4] Open Media Platform project. `http://www.openmediaplatform.eu`.

[5] RTLinuxFree. `http://www.rtlinuxfree.com/`.

[6] The Linux Kernel Archives. `http://www.kernel.org`.

[7] Rtai-the realtime application interface for linux from diapm. `https://www.rtai.org`.

[8] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril. Towards hierarchical scheduling on top of vxworks. In $OSPERT'08$, July 2008.

[9] M. Behnam, I. Shin, T. Nolte, and M. Nolin. Sirap: A synchronization protocol for hierarchical resource sharing in real-time open systems. In $EMSOFT'07$, 2007.

[10] S. Kato and N. Yamasaki. Modular real-time linux. In $RTLWS'08$, October 2008.

[11] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. Aquosa—adaptive quality of service architecture. Softw. Pract. Exper., 39(1):1–31, 2009.

[12] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: a resource-centric approach to real-time and multimedia systems. pages 476–490, 2001.

[13] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In $USENIX'02$, August 2002.