30
Real Time in a Real Operating System

Sape J. Mullender, Pierre G. Jansen

Introduction

The quality of an operating system is more a subject of religious debate than of technical merit. The Windows community is like the Catholic Church; it has the largest following, and its members are mostly laymen who do not participate much in religious debates. The community is organized on strong hierarchical lines.

The Unix community is like the mainstream Protestant Church; it has not as large a following as the Windows community, and its members define the system and run the community. Like the Protestant Church, there are many flavors of observance: Linux, FreeBSD, NetBSD, Mach; the list is as long as the list of protestant variants. Most are highly evangelical—a good Protestant trait—with Linux perhaps being the most fanatical.

The Macintosh community hangs somewhere in the lurch between Windows and Unix, the Catholics and the Protestants, a bit like the Anglican Church; they're Protestants acting like Catholics.

Plan 9 from Bell Labs is like the Quakers: distinguished by its stress on the 'Inner Light,' noted for simplicity of life, in particular for plainness of speech. Like the Quakers, Plan 9 does not proselytize.

Plan 9 is relatively little known and has but a small user community (a few thousand installations). Nevertheless, it is a complete operating system, and it is the only operating system booted by many of its users. Plan 9 is also used in several embedded environments. For instance, it is the system inside the *Viaduct*, a computer system the size of a packet of cigarettes that provides an encrypted bridge between Lucent employees' home computers and the corporate intranet. It is also beginning to find use in experimental wireless base stations.

New technologies (the printing press, organ transplants, birth control) and changing world views (the solar system, evolution) have always been upsetting to churches, causing violent debates and schisms. This is just as true in the operating system community, where new things like object-oriented programming, copyleft licensing, Ethernet vs. token ring and real-time support can cause similar violent debates and schisms.

It is the doctrine of real-time support in a general-purpose operating system that will, in this paper, be stamped with ecclesiastical authority.

We have integrated a real-time CPU scheduler in our operating system Plan 9 [7]. Although our scheduler is a new scheduler in terms of sharing the operating system resources, it has its fundaments in the EDF scheduler as first introduced by Liu and Layland [6]. Instead of only considering the CPU resource, our scheduler also considers other shared OS resources: applications indicate which resources they require (including processor use), and our scheduler determines if the set of applications can run concurrently and remain *schedulable*.

Although other operating systems may also have real-time support, we believe there are only few general-purpose operating systems with a comparable native support for real-time applications.

In many embedded systems, some applications have stringent real-time requirements, while others can be best effort. Traditionally, general-purpose operating systems have never been good at guaranteeing deadlines. Various attempts have been made to introduce real-time schedulers to general-purpose operating systems. A few systems deal with real-time applications by shutting out other applications (the general modus operandi for the Windows family of operating systems).

In the subsequent sections, we shall describe our system and the theory behind it, omitting, for lack of space, most proofs and a discussion of related work. As such, this paper has the status of an extended abstract more than a full-fledged paper. For a more formal introduction, see Jansen & Laan [4], and Jansen's forthcoming thesis.

Practicalities

Adding real-time functionality to Plan 9 as a layer below regular user programs was deemed to be undesirable. At best it would make the API for writing real-time applications a subset of the standard API; at worst, it would be completely different. We wanted to give real-time applications access to all operating system services and access to an interface to control an application's real-time behavior as well. The price one has to pay in this approach is that real-time applications may risk missing their deadline by using non-real-time services.

Although we consider this to be clumsy programming, we have no desire to forbid it. We envision that, with time, real-time versions of various operating system services will become available, e.g., a real-time file server along the lines of Nemesis' Clockwise mixed-media server [3]. Plan 9 makes extensive use of *file servers,* which, through their name space mounted in a per-process *mount table*, provides access to much more than secondary storage. The window system's interface is a file system; a *play list* file system may be associated with an audio device; mail messages present themselves as subdirectories in a mail file system, and so on. Talking to file systems is important to most applications, so it cannot be forbidden. In fact, our real-time scheduler presents itself as a file system too.

Another issue was how to deal with processes whose deadlines depend on one another. The most common example of this is a set of processes in a pipeline, for instance, a process decrypting a video stream feeding another that renders it. Scheduling theory has problems with such dependencies. We chose to allow several processes to share a single allocation of resources: one period, one deadline, and one slice of the CPU equal to the sum of the run times required by each of the member processes.

Resources are identified to the scheduler by name. A resource is shared when tasks share the name of the resource. When a resource is acquired or released, tasks inform the scheduler. This is the only involvement the scheduler has with shared resources. Resources can, therefore, be anything. One important assumption is that tasks give up any resources they hold when they give up the processor. Tasks can cause themselves to be scheduled non-preemptively with respect to each other by sharing a resource full time. When they share no resources, a task with an earlier deadline can always preempt a task with a later one.

Theory

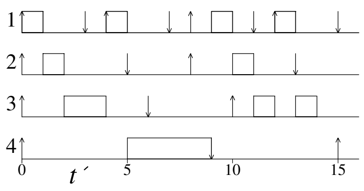
A *task set* Ω consists of a set of preemptable tasks τ_i ($i = 1 \ldots n$). Each task τ_i is specified by a *period* T_i, a *deadline* D_i, a *cost* C_i, and a *resources specification* ρ_i. It is *released* every T_i seconds and must be able to consume at most C_i seconds of CPU time before reaching its deadline D_i seconds after release ($C_i \leq D_i \leq T_i$). We use capital letters for intervals (e.g., T, D, C) and lower case for points in time: in particular, r for the next release time and d for the next deadline.

The *utilization* U of Ω is defined as $U = \sum_{i=0}^{n} C_i / T_i$

For Ω to be schedulable, $U \leq 1$ must hold. We define two functions, *processor demand* $H(t)$, introduced by Baruah et al. [2], and *workload* $W(t)$, introduced by Audsley et al. [1], $H(t)$ represents the total amount of CPU time that must be available between 0 and t for Ω to be schedulable. $W(t)$ represents the cumulative amount of CPU time that is consumable by all task releases between time 0 and t.

Figure 1 illustrates the functions for an example task set. All tasks in Ω are released simultaneously at $t = 0$. This is known as a *critical instant*, the time at which the release of tasks will produce the largest response time. If Ω is schedulable from a critical instant, it is schedulable from any other starting point. A critical instant occurs in resource-free preemptive EDF scheduling when all tasks are released simultaneously. This is a well-know result, but we have also proven it for our EDF scheduler.

task	D	T	C
τ_1	3	4	1
τ_2	5	8	1
τ_3	6	10	2
τ_4	9	15	4



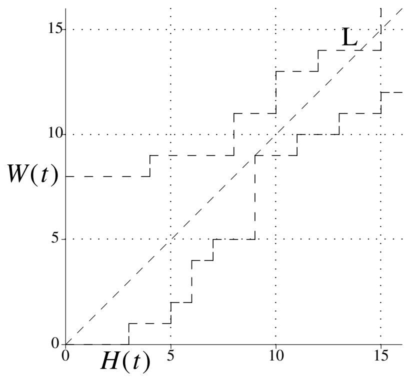


Figure 1: Example task set and its EDF schedule on the left, and the processor demand and workload functions on the right.

The right half of the figure shows the functions H and W as a function of time. It also illustrates the schedulability analysis. Note that the vertical distance between W and the diagonal in the graph represents the amount of work still to do in released tasks. At point L, there is no more work to do, and the system becomes idle. H represents the amount of work that must be finished. If H crosses the diagonal, then more work would have to be finished than there is time available. The schedulability analysis tracks W and H until either W *touches* the diagonal or H *crosses* it. If H crosses the diagonal, the task set is not schedulable. If W touches it, the task set is schedulable. The example task set is thus schedulable. Task sets can be constructed in which neither W nor H reaches the diagonal. The schedulability analysis, therefore, traces these functions for only a predetermined maximum number of steps and rejects a task set if this maximum is reached.

The scheduler manages the set of admitted tasks using two queues and a stack: The *Wait Queue* holds tasks awaiting their release. When a task gives up the processor or reaches its deadline, it is put on this queue, in release-time order, from which it will be transferred to the next queue when it is released. The *Released Queue* holds processes that have been released but have not yet run. This queue is maintained in deadline order, earliest deadline first. The *Run Stack* holds the tasks that have already run; the currently running task is at the top of the stack and pre-empted the task immediately below.

The scheduler maintains two timers. The *Release Timer* goes off when the task at the head of the Wait Queue needs to be released. Released tasks are then transferred to the Released Queue. The *Deadline Timer* goes off when the currently running task reaches its deadline. When this timer goes off, the currently running task is removed from the (top of the) stack and put back in the Wait Queue.

When a task gets to the front of the Released Queue or when a task is popped from the Run Stack, the deadlines of the task at the head of the Released Queue τ and the task at the top of the stack τ' are compared. If $d_\tau < d_{\tau'}$, it is removed from its queue and pushed onto the Run Stack. Then the Run Timer is set and the task gets the processor. If both Run Stack and Released Queue are empty, best effort processes are scheduled.

A *resource specification* ρ is a series of zero or more quadruples *name*, *R*, *C*, $\{\rho'\}$, where *name* names the resource, *R* indicates whether the resource is a shared-read or (in its absence) an exclusive-access resource, *C* is the *cost* of the resource (the time the resource is held), and $\{\rho'\}$ is a sub-specification which specifies nested resources, or may be absent. An example of a task set with a resource specification is:

```
D=4s T=5s  C=1s resources='a R 900ms { b }'
D=5s T=8s  C=1s resources='a R 800ms {b 200ms { c 100ms }}'
D=6s T=10s C=2s resources='b R 200ms c R 1.7s { b R 1.3s }'
D=9s T=9s  C=3s resources='a R 1.8s { c R }'
```

When costs are omitted, they are inherited from their parent resource specification or, in the case of a top-level specification, from the task's cost *C*. Note, by the way, that the strings in this example can be written precisely as they are to the scheduler file system to specify a task's real-time parameters.

Task 1 has a period of 5 seconds, a deadline of 4 seconds (if it is released at *t*, its deadline is at $t + 4$ and its next release is at $t + 5$); it needs at most 1 second of CPU time between release and deadline. Resource *a* is shared by tasks 1, 2, and 4. In all cases it is a shared-read resource, so it imposes no restrictions on the schedulability of these tasks. Resource *b* is shared by tasks 1, 2, and 3. Task 1 needs exclusive access to it, and for the full 900 ms, it also holds resource *a*. Task 3 needs shared-read access to resource *b* for 200 ms and again for 100 ms while holding resource *c*.

The principle behind scheduling a task set with shared resources is that we keep tasks on the Released Queue until there are no tasks left in the Run Stack holding resources that the task on the Released Queue may claim. Thus, it is not possible for a task to (try to) claim a resource already held by another task. Such a task would simply not have been scheduled. Tasks never need to be preempted waiting for a resource.

Here's how we enforce this: every resource *R* is assigned an *inherited deadline* $\Delta_R = \min_{\tau\in\Omega} D_\tau \mid R \in \tau$, the minimum of the deadlines of all tasks using *R*. Every task τ also receives an inherited deadline $\Delta_\tau = \min_R \Delta_R \mid R \in \tau$, the minimum of the inherited deadlines of all resources used by the task. A task's Δ thus changes as the task acquires and releases resources; Δ is only relevant for running tasks.

Each released task is now characterized by the triple $\{d, D, \Delta\}$, where *d* is the current *absolute* deadline (*D* is the deadline *interval*; *d* is the absolute deadline).

Earlier, we presented the scheduling rule that the task τ at the head of the Released Queue would move to the top of the Run Stack if its d_τ was less than $d_{\tau'}$ of the task τ' on top of the Run Stack—a released task with an earliest deadline will pre-empt the currently running task. Now we modify that rule:

$$\tau \text{ preempts } \tau' \text{ iff } d_\tau < d_{\tau'} \wedge D_\tau < \Delta_{\tau'}$$

Figure 2 shows an example Run Stack (rectangles) and Released Queue (ellipses). At this time, the task at the head of the Released Queue may not preempt the one on top of the Run Stack ($9 < 7 \wedge 3 < 4$ is false). For every task τ, $\Delta_\tau \leq D_\tau$ and, because of the scheduling rule, for a task τ higher on the Run Stack than another task τ', $D_\tau < \Delta_{\tau'}$. There is, therefore, a partial ordering from D to Δ to D, etc. up and down the Run Stack. This is indicated by the arrows.

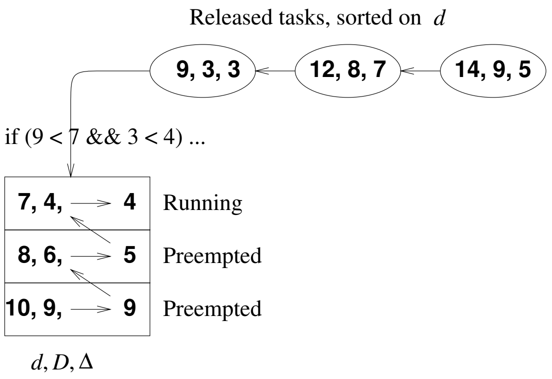


Figure 2: Example Run Stack (rectangles) and Released Queue (ellipses); the arrows indicate the partial order between the parameters.

This ordering, plus the definition of Δ, establishes the property that the currently running task—which is at the top of the Run Stack—will not attempt to acquire any resources held by preempted tasks, which are further down in the Run Stack, because, if they held such resources, their Δ would be less than or equal to the D of the running task, and this the scheduler does not allow.

A second property is that there is no transitive blocking, because a process that is blocked due to shared resource usage only has to wait for the blocker to release the resource. This property was already known from the Priority Ceiling protocol [8], a protocol that was the first to introduce static priority inheritance, similar to our static deadline inheritance.

The schedulability analysis is only moderately more complex with resource sharing. The processor demand and workload functions do not change, because the work that needs to be done and when it needs to be done is the same. But we do have to take into account now that one task may *block* another's access to the CPU.

This causes ‘spikes’ on the processor demand function. The height of the spikes encodes the time a task may have to wait for a task with a later deadline that holds a resource the task needs. A task set is inadmissible if one of the spikes crosses the diagonal. If there are no shared resources, there is no blocking (and there are no spikes), and the schedulability test reduces to the normal preemptive-EDF schedulability test. If there is one resource, shared full-time by all tasks, the schedulability test reduces to Jeffay’s [5] non-preemptive schedulability test. Our schedulability test spans the range between the extremes of completely preemptive and completely non-preemptive scheduling.

Implementation

We implemented the scheduler in Plan 9. This was a fairly straightforward process, although we had to change the behavior of spin locks in the kernel slightly. A process is now allowed to finish its critical section before being subject to scheduling. None of the spin locks hold the CPU longer than 50 μs or so.

As explained earlier, two timers control the real-time portion of the scheduler: the Release Timer goes off when the task at the head of the Wait Queue must be released. If that task gets to the front of the Release Queue, a scheduling decision is made, otherwise, the current task continues running. When the Deadline Timer goes off, the running task has used up its quantum, and the processor is taken away from it until the next release. We also raise an exception in the process.

The interesting part about the implementation is the use of a file system to control the system. In the default mount point of `/dev/realtime` we find three files, `clone`, `resources`, `time`, and a directory: `task`. Existing tasks are represented by files (whose names are numbers) in the `task` directory. A new task is created by opening the file `clone`, which then behaves like the corresponding (new) file in the `task` directory. The main loop for a typical real-time process would look something like the following:

```
char *clonedev = "/dev/realtime/clone";
void processvideo(void){
  int fd;
  fd = open(clonedev, ORDWR);
  if (fprint(fd,
       "T=33ms D=20ms C=8ms procs=self admit") < 0)
           sysfatal("%s: admission: %r", clonedev);
  while (processframe())
           fprint(fd, "yield");
  fprint(fd, "remove");
  close(fd); }
```

This sequence creates a new task by opening `/dev/realtime/clone`, sets period, deadline and cost, and puts the running process into the process group of the task. It then asks the scheduler to admit the new task by running the schedulability test. If the write succeeds, the task was admitted.

The main loop processes a video frame and then gives up the processor `(yield)` while waiting for the next frame. When the application has finished, it removes the task from the system and exits.

Conclusion

The real time scheduler is installed in the currently distributed version of Plan 9 (obtainable through `plan9.bell-labs.com`). It has already been used in several applications, one of them an experimental wireless base station. But there have not been any applications that have challenged the scheduler much.

We have had some lively debates over whether it is worthwhile to have a real-time scheduler that can manage shared resources. Most of the real-time applications we considered do not have any resources that are shared. But one real-time application we built has nothing but shared resources: the Clockwise mixed-media file system has many real-time processes, with varying periods and costs, sharing disks. As it turned out, scheduling the disks was much more important than scheduling the CPU, so the Plan9 scheduler would not have been adequate for this application.

The battle about whether or not to include support for resource sharing in our real-time scheduler was won by the resource-sharing camp when the algorithms presented here emerged: the schedulability test is not overly complicated and the run-time complexity is practically O(1): only the queue insertions are not constant-time operations, but the queues are invariably very short. In addition, the scheduler prevents resource contention from causing gratuitous context switches, and it is completely deadlock free. Finally, the same scheduler can trivially be used for preemptive or non-preemptive real-time EDF scheduling.

References

1. Audsley, N.C., Burns, A., Richardson, M.F., and Wellings, A.J., ‘Hard real-time scheduling: the deadline monotonic approach,’ in *Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, Georgia, 1991. Available at: http://citeseer.nj.nec.com/article/audsley91hard.html
2. Baruah, S.K., Mok, A.K., and Rosier, L., ‘Preemptively scheduling hard-real-time sporadic tasks on one processor,’ in *Proc. of the Real-Time Systems Symposium*, 1990, pp. 182–190.
3. Bosch, P., Mullender, S.J., and Jansen, P.G., ‘Clockwise: a mixed-media file system,’ in *Proc. of the IEEE Intl. Conf. on Multimedia Computing and Systems (ICMCS)*, II, Firenze, Italy, 1999, pp. 277–281. Available at: http://www.cwi.nl/~peterb/papers/icmcs99.ps.gz

4. JANSEN, P. G., AND LAAN, R., ‘The stack resource protocol based on real-time transactions,’ in *IEEE Proc. Software*, vol. 146, no. 2, 1999, pp. 112–119.
5. JEFFAY, K., STANAT, D.F., AND MARTEL, C.U., ‘On non-preemptive scheduling on periodic and sporadic tasks,’ in *Proc. of the Real-Time Systems Symposium*, 1991, pp. 129–139.
6. LIU, C. L., AND LAYLAND, J. W., ‘Scheduling algorithms for multiprogramming in a hard real-time environment,’ *Journal of the ACM*, vol. 20, no. 1, 1973, pp. 46–61.
7. PIKE, R., PRESOTTO, D., DORWARD, S., FLANDRENA, B., THOMPSON, K., TRICKEY, H., AND WINTERBOTTOM, P., ‘Plan 9 from Bell Labs,’ *Computing Systems,* vol. 8, no. 3, 1995, pp. 221–225. Available at:
http://plan9.bell-labs.com/sys/doc/9.html
8. SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P., ‘Priority inheritance protocols: an approach to real-time synchronization,’ *IEEE Trans. on Computers*, vol. 39, no. 9, 1990, pp. 1175–1185.