Real Time in Plan 9

Sape Mullender
Jim McKie

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

We describe our experience with the implementation and use of a hard-real-time scheduler for use in Plan 9 as an embedded operating system.

1. Introduction

Plan 9 from Bell Labs is a general purpose operating system that is finding increasing use as an embedded systems platform. Recently, we have started using Plan 9 as a basis for a wireless base station, which requires a hard real-time system.

1.1. Real Time

In this section, we give a very brief introduction to real-time concepts, enough to make this paper readable by itself.

A real-time process is characterized by three basic parameters: its **period**, *t*, its **deadline**, *d*, and its **cost**, *c*; they are illustrated in Figure 1.

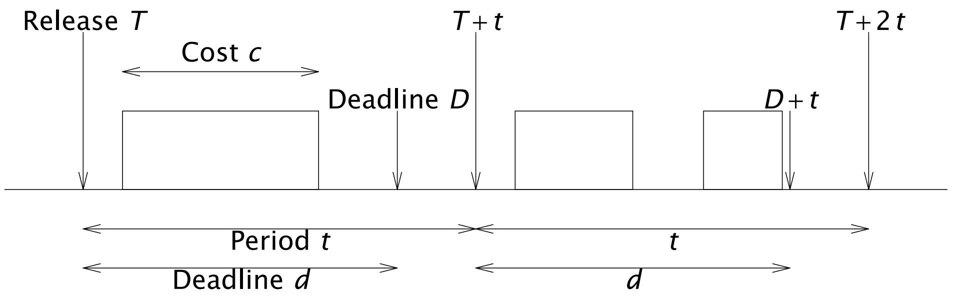

Figure 1 Parameters characterizing a real-time process; note that intervals are indicated by lower-case letters, while times are indicated using capitals.

The **release**, *T*, of a process is the time at which it is allowed to start running. The **deadline**, *D*, is the time at which the work has to be finished. The **cost**, *c*, is the amount of time the job takes.

Real-time processes are usually classified as periodic or sporadic. A **periodic** process must do work at regular intervals of *t* seconds, the **period**, within a **deadline**, *d* (note the use of deadline both as a time and an interval). A **sporadic** process must do its job within a deadline, *d*, after some external event, its **trigger**.

The cost, *c*, must be less than the deadline, *d*, for a process to be schedulable.

If there is more than one real-time process, there are additional conditions that must be met to allow all processes to meet their deadlines. A minimum condition is that the costs consumed per unit of time must add up to less than 100%: $\sum c/t \leq 1$.

An **admission test** makes sure that the deadlines of all real- time processes can be met. In most systems, additional real- time processes are not allowed to be started if the admission test fails. An admission test for sporadic processes can only be made if the minimum interval of the trigger events is known. We call this the **period** of the sporadic process and, for the admission test, treat it as a periodic process.

Assuming processes can be preempted — that is, the execution of a process can be temporarily suspended to allow another process with more urgent needs to be run — the Earliest-Deadline First algorithm is optimal. In fact, if $t = d$ for all processes, $\sum c/t \leq 1$ is a sufficient test.

Admission testing and scheduling get more complicated if processes compete for resources. In software systems, these resources can be critical sections, or peripherals.

To illustrate these complications, consider a low-priority (i.e., deadline in the distant future) process that obtains a lock. Then a medium-priority (i.e., deadline in the near future) process is released and it preempts the low-priority one. Next, a high-priority (i.e., deadline imminent) process is released. It preempts the medium-priority process and attempts to get the lock that the low-priority process holds. This blocks the high-priority process until the lock is released. But before the low- priority process can release the lock it has to run and before it runs, the medium-priority process must give up the processor.

This is a classic example of **priority inversion**: the low- priority process should receive a higher priority than the medium-priority process until it gives up the lock, so that the high-priority process gets to run again as soon as possible.

Priority inheritance does just that: a process holding a lock receives the priority of the highest-priority process waiting for it.

To do a schedulability analysis for a collection of real-time (and best-effort) processes that share certain resources, one needs to know period, deadline and cost for each of the real- time processes. In addition, for each process and combination of resources, period and cost must be known also. Best-effort processes sharing resources with real-time processes are thus drawn into the schedulability analysis too.

With this minimal background in real-time, the rest of the paper should be readable.

1.2. Real time in a general purpose operating system

Now let's get practical and see what obstacles there are to putting good theory into practice. We are by no means the first to do this. There are quite a few real-time systems out there. Most of them are specialized for embedded platforms (VxWorks, RTOS, Carrier-grade Linux, etc.). Some systems provide real-time functionality in a general-purpose operating system (RTLinux, AIX, LynxOS, etc.). What tends to distinguish the latter from the former is that the latter has different processes run in different address spaces and that the operating system provides protection of one process against the accidental or malicious misbehavior of another.

Another complicating factor is that the collection of system calls and operating-system services available to applications is usually not geared towards real-time applications. This means that, if a real-time application invokes certain system calls or operating-system services, it may well miss its deadline as a result.

Device drivers and device interrupts are another source of potential deadline misses. Usually, operating systems do not limit the frequency of device interrupts, nor the time spent in interrupt routines.

A final implementation issue is the overhead of the various mechanisms. Obviously, a switch form one process to another takes time, so, when theory says that a set of real-time processes is schedulable provided $\sum c/t \leq 1$, practice says that, at the very least this should be $\sum (c+\delta)/t < 1 - \varepsilon$, for some positive ε and δ that are only approximately known — ε being a term to take general system overhead into account, and δ a term to express (a lower bound on) the cost of context- switching overhead.

But this is simple relative to the problem of what to do about system calls, cost overruns and shared resources. We'll tackle them one at a time.

System calls can cause arbitrary delays for the caller; reading from a terminal is a good case in point. One approach to dealing with system calls is to divide them into real-time okay and real-time not-okay ones and to allow only the okay ones in real-time applications. Whether this is useful is a matter of opinion.

Cost overruns are difficult to deal with. If a process uses more time than reserved, it may miss its deadline, but it also threatens other processes to miss theirs. The latter can be prevented by judicious scheduling. The former presents a problem in API design: Should the application be stopped when this happens "deadline miss, core dumped", should the application receive a signal "SIGMISS", or should the application be resumed and perhaps be notified at the next release?

All of these solutions are hard to program to for an application programmer. Obviously, a cost overrun is a timing bug and it shouldn't happen, but, in many cases, it can be recovered from, although the recovery is almost always application dependent. The signal method is probably the most effective and the least disruptive. It gives the application programmer choices for how to recover.

Shared resources in real-time applications can be shared data structures, shared files or devices, or even resources such as the space between two robots on an assembly line; if one robot is working in that space, the other shouldn't enter it. In all cases, the resource can be represented by a lock that is acquired and released as the shared resource is acquired and released.

Locks have very efficient implementations in shared memory: a test-and-set (TAS) or compare-and-swap (CAS) instruction — at least one of which is available on pretty much every modern uniprocessor or multiprocessor — does the basic trick nicely. In cases where the lock cannot be acquired, a process will have to wait until it is released. Busy waiting is usually too expensive, so a sleep/wakeup mechanism is usually employed to allow the process releasing the lock to bring the process waiting for it back to consciousness. However, locking conflicts are usually not that frequent and implementations can and have been conceived to avoid taking system calls unless there is context-switching work to do.

In real-time systems, all this changes, because the priorities of real-time and best-effort processes change as they acquire and release resources which leads to potential context switching. Thus, even the simple act of locking a shared data structure requires a system call when the data structure is used by at least one real-time process.

2. Application Programmer's Interface

Plan 9 is probably most famous for using file systems for its client-server interfaces. Real-time is no exception. Plan 9 processes are already managed largely through a file system interface, normally located in the directory /proc, described in the Plan 9 manual[2], in *proc*(3). A process with process ID pid can be stopped, killed, or debugged through the various files in the directory `/proc/`*pid*. For instance, writing the string "`kill`" to `/proc/`*pid*`/ctl`, will kill that process.

We extended this interface by adding commands to regulate the real-time behavior of processes. Scheduling information is given by writing strings of the form

```
period tμs
deadline dμs
cost cμs
sporadic
```

to the process' control file. Units can be ns, μs, ms, or s. The sporadic command indicates that release of the process is governed by external events and that, therefore, the process does not necessarily run a regular intervals. The *period* of a sporadic process is the minimum amount of time between consecutive releases; i.e., if the external release occurs earlier, the actual release is delayed. When all parameters have been specified, the command admit is written to the control file. If this write succeeds, the process will henceforth be scheduled as a real-time process. If the admission test fails, either because of conflicting real-time settings, or because there are insufficient resources, the write fails (returning the reason for failure in the error string).

The good thing about this interface is that it allows real- time scheduling processes from inside those processes, from other processes, or from the command-line interface.

Once a periodic process is scheduled in real time, `sleep(0)` will suspend the process until the next release. A sporadic process will be suspended on entering any blocking system call and be released when both the system call returns and a period has elapsed since the last release. If the `yieldonblock` command has been given, periodic processes will also give up the CPU until the next release whenever a blocking system call is invoked.

If a periodic process blocks on a system call, it stops consuming CPU resources until the system call ends. However, any temporary blocking may cause a deadline miss. If a process is blocked at the deadline, it will be resumed no earlier than at the next release.

3. Real Time Implementation in Plan 9

We implemented a real-time scheduler in Plan 9, so that we could use Plan 9 as a vehicle for experimentation with embedded systems, in particular, base stations for wireless networks.

The initial implementation [3] provided scheduling for real- time processes with shared resources. The current implementation no longer has provisions for shared resources (although they could easily be brought back). This paper explains and justifies our design and discusses our experience with the implementation.

3.1. Kernel Locks

The Plan 9 kernel uses a set of queues of runnable processes, one for each priority level (there are 20). It was a simple matter to add two more queues, one for active real-time processes and one for real-time processes that have been released but have not yet been scheduled. The scheduler, looking for a process to run, would scan the queues in order and pick the first process encountered on any queue. With the two new queue entries, the scheduler first looks for real-time processes.

The way kernel locks were handled had to be changed. There are, in fact, three kinds of kernel locks, they are called `ilock`, `lock`, and `qlock`. `Locks` are spin locks: to acquire the lock, a *test-and-set* on the lock field is repeated until it succeeds. An `ilock` behaves like `lock`, except for the processor priority which is raised to disallow interrupts until the ilock is released. Interrupt routines may not acquire regular locks, because the process they interrupted may have been holding it, thus leading to deadlock. But interrupt routines can safely acquire `ilocks`, because by the way they work, an interrupt routine can never interrupt the process holding one. In fact, on a uniprocessor, it is impossible to have to wait for an `ilock`. `Ilocks` are of little concern to the scheduler, because processing holding them cannot be interrupted and, therefore, they can not be preempted.

`Qlocks` are locks with a queue of waiting processes attached to them. If a `qlock` can not be obtained, a process attaches itself to this queue and calls `sleep()`. When a process gives up a `qlock`, it calls `wakeup()` for each of the waiting processes in the queue. Because if this behavior, any system call that invokes `qlock` will either cause a deadline miss (programming error) or is intentionally used to wait for an event. Unlike `lock` and `ilock`, they receive no special attention for real-time scheduling.

This just leaves regular locks.

The pre-real-time Plan 9 kernel would, when a process blocked on a lock, lower its priority to the lowest level and add it to a list of waiting processes attached to the lock. Upon release of the lock, all waiting processes would be restored to their old priority (and be allowed to compete for the lock).

This, obviously, did not work for real-time processes. Today, the Plan 9 kernel will not schedule a process holding a lock until that process gives it up. Each process maintains a count of the number of locks it holds. When a process holding locks must be pre-empted, a flag is set in its process structure. When the process releases its last lock, it examines this flag and, if it is set, will invoke the scheduler again.

It is a kernel programming error to allow processes to return to user space with a lock held (remember, we are talking about kernel locks); likewise, it is an error to allow a process to sleep (wait for an event) with a lock held. Various assertions in the kernel check for violations of this invariant. As a result, real-time processes are scheduled with no more extra delay than the time it takes for the previously running process to give up its locks.

To measure this extra latency, we instrumented our kernel `locks` and `ilocks` to see what latency they cause. We found that no locks of either type are held for more than about 100 μs on the 300 MHz PowerPC that we use for our wireless base- station experiments (with the instrumentation itself being responsible for about 500 ns).

To allow more precise real-time scheduling, we started to program the clock to give interrupts exactly when scheduling events (release, deadline, or end of CPU slice) should occur. To maintain track of wall-clock time, we use a second clock (present on most architectures) to give interrupts at regular intervals that we use for counting the (milli)seconds.

The programmable timer uses a linked list of programmed timer events, sorted by time. The first timer to go off is at the head of the list. When a timer interrupt occurs, the list is processed for timers that have expired. Timers can be one shot, or periodic. Periodic timers are put back on the linked list after processing.

4. Programming Real-Time Applications

The thing that is hard about writing real-time applications is the interaction with the environment. Most useful real-time applications are multithreaded and share memory and resources with non-real-time processes or real-time processes with different periods and deadlines.

Among the most difficult interactions with the environment are the real-time process' interaction with the operating system and services contained in libraries. These are usually specified only in terms of what they eventually do, not in terms of when they do it.

A good — and frustrating — case in point is dynamic memory allocation. One thinks of `malloc()` and `free()` as routines that instantly allocate and free memory. In practice, they require locks to protect against race conditions in multithreaded applications. This makes the use of these calls in multithreaded real-time applications dangerous: the real-time process may waste its entire CPU slice by spinning on a lock held by a non-real-time process that is not allowed to run.

This sort of thing could be solved by using a more intelligent locking method, but that would require significant overhead in letting the scheduler know what locks are held/required and involving all non-real-time processes that use locks in the schedulability analysis.

We tried going this route initially, but gave it up. The overhead is too large and the specification of the real-time behavior of all critical sections was an impossible task.

Our approach has become to radically go the other route: all interactions with the environment avoid resources that require exclusive access.

We found that we could get along perfectly with just three mechanisms for managing shared resources. They are described in the next three subsections. Queues A great deal of communication between a real-time process and another necessarily has to go through some sort of buffer to allow a synchronous and non-synchronous environment to exchange information. A queue is the typical embodiment of a buffer in a computer application.

Queues can simply be made non-blocking, provided there is exactly one producer process and one consumer process. The atomicity of placing an item on the queue or removing one from the queue is realized by the atomicity of reading and writing (aligned) words in memory. Most modern computers have 32-bit integers that have this property, even on shared- memory multiprocessors.

The routines to enqueue and dequeue are shown in Figure 2.

```
typedef struct Q Q;
struct Q{
    int   produced, consumed, mask;
    void  *data[0];
};

int
qproduce(Q *q, void *x)
{
    if(q->produced - q->consumed > q->mask)
        return 0;

    q->data[q->produced & q->logmask] = x;
    q->produced++;
    return 1;
}

void *
qconsume(Q *q)
{
    void *x;

    if (q->produced == q->consumed)
        return nil;

    x = q->data[q->consumed & q->logmask];
    q->consumed++;
    return x;
}
```

Figure 2 Implementation of a non-blocking queue

The producer reads the variable `q->consumed` which is written by the consumer. If the consumer modifies this variable just after the producer has read it, but before the producer has had the opportunity to act on it, no harm is done: By updating `q->consumed`, the consumer only created more room in the queue for the producer to use. A symmetric argument holds for the consumer and `q->produced`.

We allow `q->produced` and `q->consumed` to count the total number of items ever produced or consumed without resetting it when the end of the queue is reached; thus, updating these variables is always atomic. When one of the variables wraps around and becomes negative, the difference between the two is still the number of items currently in the queue, thanks to 2's-complement arithmetic.

These routines are not merely atomic and thus deadlock and race-condition free, they are also much more efficient than any method using locks.

The only downside is that there can be only one consumer and one producer per queue. But this tends to be the case anyway. But not always ...

For debugging the real-time environment we use, we have optional print statements. These, of course, affect the real-time behavior, so our prints are queued for a non-real-time process (the reporter) to print them at its leisure. We have several real- time (and non-real-time) processes and they all do their reporting to the reporter process. But these processes must all use separate queues, so the reporter must scan, and read from, multiple queues. This is not complicated but it has to be thought about.

4.1. Free lists

As argued, real-time applications must avoid using locks and must therefore be careful about using dynamic memory allocation. In our real-time application, we tend to preallocate the buffers we are likely to need and we allocate them from a linked list. After use, they are placed back on this linked list. Many buffers are transferred between processes, so the process allocating a buffer may not be the same one that frees it later on. The salient point about a linked list holding free buffers is that buffers on the free list have no identity of importance: one can always allocate the buffer at the head of the list and one can always deallocate a buffer by placing it at the head of the list (if buffers of different sizes are used, one uses a different linked list for each size).

Atomically (de)allocating from a linked list can be done using a Compare-and-Swap (CAS) instruction, which is present on all current processors I know (and certainly on all those that run Plan 9). The CAS instruction takes three arguments, a pointer, an old value and a new value. If the pointer points to a value equal to the old value, it replaces it by the new. The test and the replacement are done atomically.

```
Block *
allocb(Block *list)
{
    Block *b;

    do
        if((b = list) == nil)
            error("out of blocks!");
    while(!cas(&list, b, b->next));
    b->next  = nil;
    return b;
}

void
freeb(Block *list, Block *b)
{
    do
        b->next = list;
    while(!cas(&list, b->next, b));
}
```

Figure 3 Implementation of atomic linked-list operations

This suggests a simple implementation for atomic allocate and free which is shown in Figure 3. There is a threat of starvation here, but we believe this is only theoretical in nature: there is no mechanism in the operation that could cause processes to get in lock step, even on a multiprocessor where two processes could, truly concurrently, be fighting over the allocation of a block.

4.2. Semaphores

When non-real-time processes are waiting for data/output/attention from a real-time process, they can either poll for an event, or busy-wait for it, or go to sleep until they are awakened by the real-time process. The latter of these three is, of course, the only elegant way and Plan 9 traditionally deals with this through the use of Channels in the thread library, or by using the rendezvous system call (see *thread*(2) and *rendezvous*(2) in [2]). Neither of these works, because both may block the real-time process.

We added system calls for manipulating counting semaphores, semacquire and semrelease that guarantee that the caller of semrelease never blocks. A caller of semacquire blocks when the value of the semaphore has reached zero. The caller continues when a call to semrelease has made the value positive again.

5. Experience

We used this real-time infrastructure at Bell Labs to build a research wireless base station that we have been using for experiments with protocols and measurements. Some of the radio hardware we've been using demanded a hard-real-time system: missing a 10ms deadline would force us to reboot the radio (not great hardware design, but that happens in research).

Most of the application programming techniques for real- time were developed perfecting this application. Currently, it sports four real-time processes, one that drives the hardware and the protocol stack (the engine, periodic at 2ms, with a 1ms CPU allocation), two that, respectively, manage the Ethernet input and output queues for the stack (the backhaul managers, sporadic at 2ms with a 100μs CPU allocation), and one that does the printing of debugging output (the reporter, periodic at 10ms with a 400μs CPU allocation). In addition, we have best-effort processes managing the management interface (the file server, a Plan 9 file system, of course) and a separate application (URM) using this interface to manage and configure the wireless connections.

There are three free lists, one for Ethernet-packet-sized blocks, one for radio-transmission-sized blocks, and one for command blocks exchanged with the radio hardware. There are atomic queues between the backhaul managers and the engine, queues between each process and the reporter, and queues between the file server and the engine.

The file server is multithreaded and has a thread for each outstanding operation. Some operations block, waiting for a response from the engine, or from the radio hardware. These file server threads use semaphores as the sleep/wakeup mechanism (with the engine as the waker upper).

The whole application is some 100K lines of code and runs on Plan 9, which was ported to a PowerPC processor on the base station. It took us a while before we developed the real- time programming techniques we now use. As we improved our techniques, we started missing fewer and fewer deadlines. Early deadline misses were almost always caused by blocking on locks, usually in user space, sporadically in the kernel. We ended up instrumenting the user-space locks with assert statements testing whether the calling process was real time. Most of the triggers were malloc() calls in unexpected places.

Using a reporter process to do the printing of any debugging works very well. Each report has a time stamp and the reporter scans its input queues to print the reports in time- stamp order.

We briefly experimented with a non-blocking multi- consumer/multi-producer queue but gave up. The implementation was too complicated to maintain. Understanding that code remained hard. One problem with such a queue is that producers must claim a place in the queue before filling it. This gives rise to a situation in which Producer 1 claims the first slot, but, before filling it, gets preempted in favor of Producer 2, which claims and fills the second slot. Then, a consumer may be scheduled which finds nothing to consume until the first slot is filled, so it must wait even though an item is present in the queue.

Currently, the application does very well without multi- producer atomic queues and that tells us we don't really need it.

So, it is perfectly possible to build real-time application that do not do any exclusive resource sharing and, therefore, do not require any form of mechanism for priority-inversion. Nevertheless, we recognize that there is a class of applications where shared resources (the space between two robots on an assembly line, a section of single track on a railway line, an actuator or a sensor) are real and part of the real-time environment. However, in these environments, the cost of claiming and releasing the resource usually remains very small compared to the cost of actually using the resource, even if claiming and releasing requires system calls, priority adjustments and rescheduling.

6. References

[1]R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, P. Winterbottom, "Plan 9 from Bell Labs", *Computing Systems*, **8**(3), Summer 1995, pp. 221—254

[2]*Plan 9 Manual*, current edition published on-line only at http://plan9.bell-labs.com/sys/man

[3]S. J. Mullender and P. G. Jansen, "Real Time in a Real Operating System", in **Computer Systems: Theory, Technology, and Applications -- A Tribute to Roger Needham**, A. Herbert and K. Spärc Jones (eds.), Springer-Verlag, New York, 2004, pp. 213-222.