From Fast to Predictably Fast

Dominic Duval
Red Hat Inc.
dduval@redhat.com

Abstract

Many software applications used in finance, telecommunications, the military, and other industries have extremely demanding timing requirements. Forcing an application to wait for a few extra milliseconds can cause vast sums of money to be lost on the stock markets, important phone calls to be dropped, or an industrial welding laser to miss its target. Highly specialized realtime operating systems have historically been the only way to guarantee that timing constraints would always be respected.

Several enhancements to the Linux kernel have recently made it possible to achieve predictable, guaranteed response times. The Linux kernel is now, more than ever before, well equipped to compete with other realtime operating systems. However, applications may still need to be modified and adjusted in order to run predictably and fully benefit from these realtime extensions. This paper describes our findings, experiences and best practices in reducing latency in user-space applications. This discussion focuses on how applications can optimize the realtime extensions available in the Linux kernel, but is also relevant to any software developer who may be concerned with application response times.

1 Before we start

It is worth emphasizing that no special libraries or application programming interfaces are required under the realtime kernel in order to use the real time capabilities discussed in this document. The `CONFIG_PREEMPT_RT` realtime patch, as opposed to other realtime initiatives in the Linux community, follows standard Posix API's. The realtime patch only affects code in the kernel: user space applications should not notice any difference other than better determinism in how they perform. Consequently, there is no need for applications to invoke special libraries in order to benefit from the realtime kernel.

Some programming techniques, however, are known to create large sources of latencies in applications. This could cause a realtime application to lack the level of determinism that is required from it and to behave unpredictably to some degree. These techniques and habits are precisely the ones we will address in this document and for which alternative methods will be discussed.

2 Testing the system for realtime capabilities

Any environment from which realtime capabilities are expected should first make sure that the underlying system meets realtime requirements. This involves making sure that:

- The application may lock memory
- The scheduler API is available and the application is allowed to set the scheduler to a non-default scheduling strategy such as `SCHED_FIFO`
- The kernel was compiled with `CONFIG_PREEMPT_RT=y`
- Both user space and kernel space support robust mutexes
- High Resolution Timers are available
- Clock resolution is high enough to meet the application's requirements

All these tests could be executed by the application itself, in user space. Alternatively, a tool named `rtcheck` is also available for that purpose. `rtcheck` consists of a simple application that automatically conducts the tests listed above and returns 0 if they all succeeded. A non-zero value will be returned if any of the following tests fails:

- Memory Lock. Verify ability to lock 32K of memory in user space and ensure not limits are set by default for memory locking.
- Scheduler. Exercise the scheduler API to determine if it supports real-time; namely setting the scheduler to `SCHED_FIFO`. If it takes this setting, we know we have this capability. We can imply from this test that `SCHED_RR` can also be set.
- `CONFIG_PREEMPT_RT`. Look for the presence of `/proc/loadavgrt`. If found, we can deduce that the system is running with `CONFIG_PREEMPT_RT=y`.
- Robust Mutexes: Do lookups on a few symbols indicative of user space support of robust mutexes and then do test calls of these symbols to confirm kernel support of robust mutexes.
- High Resolution Timers. Nanosleep is timed using `clock_nanosleep()` and `clock_gettime()` calls. The value is compared against a threshold large enough to be infeasible on a system using hrtimers and small enough to be too fine-grained for a system not using hrtimers. The threshold currently being used is 20us.
- Clock Resolution. Using `clock_getres()`, make sure the clock resolution is under 200us.

3 Setting Scheduling Policies Right

One of the most basic attribute of a realtime operating system is the ability to run processes and threads with different realtime policies and priorities. This mechanism is strictly enforced by the scheduler of a realtime kernel. Correctly setting priorities for all threads of an application involves first of all evaluating what parts of the application need to behave in a realtime way and which don't have to. Tasks that are best left to regular (i.e., `SCHED_OTHER`) or low priority threads include:

- Interaction with devices for which the speed is considered unpredictable (i.e., storage devices)
- Dynamic memory allocation
- Filesystem operations
- Logging

It does, however, make sense to set a realtime priority for any other task that requires predictability. In this case, the first step involves setting the scheduling policy using the `sched_setscheduler()` system call. We typically need to assign realtime threads the `SCHED_FIFO` or `SCHED_RR` policy here. Threads running with realtime scheduling policies (also called scheduling classes) are fairly different from others running under the regular `SCHED_OTHER` policy:

- They always preempt regular threads (assuming they're ready to run and not blocked).
- They do not expire. There's no such thing as a time slice for real time processes (unless two or more processes rely on `SCHED_RR`)
- They will completely starve other threads of lower priority, realtime or not, if they don't go to sleep.

3.1 Setting priorities right

The last parameter in the `sched_setscheduler()` system call consists in a priority ranging from 0 to 99, 99 being the highest possible priority. This value should be the result of careful analysis of your realtime application and how threads interact. Typical priorities look as follows:

Real Time Priorities	
99	Watchdog and migration threads
90-98	High priority realtime application threads
81-89	IRQ threads
80	NFS
70-79	Soft IRQs
2-69	Regular applications
1	Low priority kernel threads

It is worth emphasizing from the example above that interrupts have been assigned priorities (usually around 85) to handle work resulting from the use of a network interface, for instance. One can always reorder those priorities. To make sure network traffic will get prioritized at the driver level, for example, it would be possible to adjust the realtime priority of the corresponding IRQ thread in such a way that it would preempt other high priority realtime threads. This scheme provide complete control over what should be executed first.

Setting the scheduling policy and priority involves invoking the `sched_setscheduler()` call like this:

```
struct sched_param sp;
int policy = SCHED_FIFO
sp.sched_priority = 70;
if (sched_setscheduler(0,policy,&sp)) {
  perror("Could not set policy and priority");
  exit(1);
}
```

Alternatively, a tool such as `rtctl` can be used to set priorities manually at runtime or automatically when the system boots up.

4 Avoiding Page Faults

Performance associated to memory access can be affected by two factors: pages of memory being swapped out to disk and memory allocation. By default, Linux does not provide any guarantees about whether memory allocated by an application will remain in physical memory or get swapped out. Memory that ends up on the swap device may eventually need to be copied back to physical memory in order to be accessed, thus causing a major page fault. This is an extremely large source of latency in applications since that delay depends essentially on the speed of the storage device, which is usually a few orders of magnitude slower than physical memory accesses (milliseconds vs nanoseconds).

Likewise, Linux does not guarantee that memory actually gets allocated at all in physical RAM. In fact, memory allocated on the stack or via functions such as `malloc()` is usually not immediately allocated in RAM: pages of physical memory can be allocated on the fly whenever the application writes data to that memory. This mechanism relies on minor page faults. While minor faults can be dealt with relatively quickly, they can also introduce delays in the application.

In order to address these sources of latency we recommend making use of the `mlockall()` system call, which ensures that memory allocated on behalf of a process never causes a page fault. `mlockall()` supports two flags:

- `MCL_CURRENT`: Lock all pages which are currently mapped into the address space of the process.
- `MCL_FUTURE`: Lock all pages which will become mapped into the address space of the process in the future. These could be for instance new pages required by a growing heap and stack as well as new memory mapped files or shared memory regions.

For most applications both flags are needed: this will make sure no present or future allocated memory areas cause any major page faults, i.e.:

```
if (mlockall(MCL_CURRENT|MCL_FUTURE) == -1) {
  perror(Could not lock memory.'');
  exit(1);
}
```

Make sure you have an idea what the memory consumption of your application should look like before you use `mlockall()`. All pages of memory belonging to the process making a call to `mlockall()` will be locked in physical memory. That includes code, data and library contents. Locked memory can be unlocked if needed using the `munlockall()` system call.

For large applications or systems with limited resources, this system call can easily cause memory starvation issues on the entire system. As an alternative to `mlockall()`, applications that require more flexibility may rely on the `mlock()` system call. In this case, specific memory regions may be locked and unlocked later on with `munlock()`.

Furthermore, if the application does not get invoked by the superuser you will need to set the appropriate process capability, i.e., `CAP_IPC_LOCK`. If set, no limits will be placed on the amount of memory the process can lock. Similarly, for unprivileged processes the `RLIMIT_MEMLOCK` limit can be set instead in order to define the maximum amount of memory that can be locked.

You may verify default settings by running the `rtcheck` utility discussed previously in this document or by invoking

```
$ ulimit -l
```

Modifying the `RLIMIT_MEMLOCK` value typically involves assigning the user running the application to the `realtime` group. Limits listed under `/etc/security/limits.d/realtime.conf` will then automatically be applied next time this user logs in. Such a configuration file may look as follows:

```
@realtime   soft  cpu       unlimited
@realtime   -     rtprio    100
@realtime   -     nice      40
@realtime   -     memlock   unlimited
```

Similarly, limits may be set programmatically by the superuser with the `setrlimit()` system call. In the following example we're setting limits for the current process to unlimited:

```
#include <sys/resource.h>
struct rlimit rlim = {RLIM_INFINITY,
                      RLIM_INFINITY};
setrlimit(RLIMIT_MEMLOCK, rlim);
```

4.1 Pre-faulting the stack

The stack of a process is another potential source of page faults to be avoided: whenever a local variable gets defined or a function gets executed, the stack grows by a few extra bytes. At some point (i.e., when we run out of physical memory in a stack page) this will trigger a minor page fault and allocate one more page of RAM. This can obviously cause some latency that's not desirable in the context of a realtime application. As a solution, we recommend pre-faulting the stack when the application or the thread get started. This can be done as follows:

```
#define MAX_STACK_SIZE (128*1024)
void prefaulter(void) {
        unsigned char dummy[MAX_STACK_SIZE];
        memset(&dummy, 0, MAX_STACK_SIZE);
}
```

Note that the main challenge here is to determine ahead of time what the maximum stack usage in the application will be. A practical way to determine the stack size is to pre-fault the entire stack to a known value at startup time and determine, when the application gets terminated, which pages of the stack got modified. This iterative technique is not without its limits but the result can at least be used as a starting value in the example listed above.

4.2 Working around malloc's unpredictability

glibc's `malloc` function comes with a default set of tunings that works well in the common (non realtime) case, but lacks the predictability expected from realtime applications. Fortunately, a function known as `mallopt()` makes it possible to eliminate many issues associated with `malloc()`.

Calls to `malloc` and `free` are not always necessarily translated to a corresponding call to `sbrk()`. In fact, `malloc()` usually makes use of a set of preallocated subheaps in order to reduce memory fragmentation and speed up memory allocation. This also means that any application calling `free()` will use `sbrk()` to give memory back to the system. We obviously don't want that to happen in the context of a realtime application. The `mallopt()` function, in conjunction with the `M_TRIM_THRESHOLD` flag lets us modify this behavior and completely disable the memory reclaim:

```
if (!mallopt(M_TRIM_THRESHOLD, -1) {
  return 1;
}
```

`malloc` uses `mmap()` by default in order to allocate any block of memory larger than 128k. This can create undesirable situations where a call to `free` will result in the `munmap` system call being invoked, thus giving back locked pages to the kernel. Since we want to keep those locked pages in our address space, we need to disable the use of `mmap()` in `malloc()` context:

```
if (!mallopt(M_MMAP_MAX, 0)) {
  return 1;
}
```

4.3 Real time dynamic memory allocator

As we have just seen, allocating memory dynamically is generally considered incompatible with the requirements of a realtime system. Memory locking is widely viewed as the only alternative to this problem.

There are cases, however, where memory locking just is not practical. There are applications for which we just cannot predict the memory footprint. We must thus come up with a solution that makes it possible to rely on a dynamic allocator that offer guaranteed response times. The Two-Level Segregate Fit (TLSF) allocator is a constant cost memory allocator that can be used in these cases. While it cannot avoid major page faults (disabling swap may thus be required), it does provide guarantees about the time required to allocate any amount of memory: TLSF operates with O(1), i.e. constant, cost.

4.4 Dynamic library preloading

Linux uses a technique known as "lazy linking" in order to load libraries into memory at execution time. This

```
static bool dumpPageFaults(void) {
  bool l_PageFaultsDetected = false;
  static bool ls_Init = false;
  static struct rusage Previous;
  struct rusage Current;

  getrusage(RUSAGE_SELF, &Current);
  int a_NewMinorPageFaults = Current.ru_minflt - Previous.ru_minflt;
  int a_NewMajorPageFaults = Current.ru_majflt - Previous.ru_majflt;
  Previous.ru_minflt = Current.ru_minflt;
  Previous.ru_majflt = Current.ru_majflt;

  if (ls_Init) {
    if ((a_NewMinorPageFaults > 0) || (a_NewMajorPageFaults > 0)) {
      printf("New minor/major page faults: %d/\d{\n",
              a_NewMinorPageFaults,
              a_NewMajorPageFaults);
      l_PageFaultsDetected = true;
    }
  }
  ls_Init = true;
  return l_PageFaultsDetected;
}
```

Figure 1: An example of how to monitor page faults.

method essentially means that a library to which an executable is linked will not be loaded in RAM unless a call is made to a component of that library. Page faults can therefore be avoided in the common case where no calls are made to some parts of the library and, most importantly, physical memory usage can be minimized.

Applications that link and use libraries will likely cause page faults. As we have seen in the last section, page faults are detrimental to the application's responsiveness and need to be avoided in realtime environments. This means we need a way to load libraries in advance, i.e., whenever the application gets started.

Lazy linking can be controlled by the environment variable `LD_BIND_NOW`. The loader reads this variable in order to determine whether it should load libraries in memory right now (`LD_BIND_NOW`) or when it will actually be invoked (also known as lazy linking). Setting this environment variable can be done on the command line with:

```
$ export LD_BIND_NOW=1
```

Alternatively, lazy linking can be configured on the executable file itself with the `-z` linker option at compile time:

```
$ gcc app.c -o app -Wl,-z,lazy
```

As a result, all dynamic libraries will be automatically loaded when the application gets invoked.

4.5 Monitoring page faults

As we have seen in the previous sections, page faults are generally one of the largest sources of latency in realtime applications. Minimizing or eliminating them will help you achieve better latency results. In order to validate the strategies documented in this whitepaper we recommend using the `getrusage()` function defined in `resource.h`:

```
int getrusage(int who, struct rusage
*usage);
```

The first parameter is generally `RUSAGE_SELF`: this will let you access statistics for the current process, which includes all its threads but excludes any child process that may have been created before. The second argument is a reference to a data structure that contains a number of statistics related to the current process:

```
struct rusage {
 [...]
 long   ru_minflt;
 long   ru_majflt;
 [...]
};
```

What we really want to focus on here are the `ru_minflt` and `ru_majflt` fields, which report the number of minor and major faults since the process started, respectively. An increasing number of minor and major faults in a realtime application should be a concern only after the application has been fully initialized: at that point very few operations should generate faults.

As an example, an application can keep track and report page fault statistics by implementing something similar to what's listed in Figure 1 above.

5 Efficient Locking

The realtime patch is known to expose lock-related bugs more easily due to the fine grained nature of the realtime kernel. These bugs are not necessarily new in user space applications: they were typically just left unnoticed due to the less dynamic behavior of the standard kernel (this is particularly true for non-SMP systems). Solving locking issues should first involve analyzing which locks are used in the application and ensuring that lock contention won't everbecome a problem as the application scales.

Other steps can also be taken in order to improve lock efficiency in a user space application running on the realtime kernel:

- Never use SysV semaphores in order to protect shared data resources in your application. These mechanisms are not designed and implemented to support priority inheritance, which is a requirement for any realtime system. Pthread mutexes should be used instead.
- Pthread mutexes should be given the `PTHREAD_PRIO_INHERIT` attribute in order to turn priority inheritance on.
- Rely on mutex priority ceilings if priority inheritance can't be used.
- If a counting (i.e., non-binary) semaphore is required in the application, implement it using condition variables.

6 Priority Inversion

Extra latency can occur when threads with different priorities share a common resource that's protected by a lock. Priority inversion happens when a high priority thread tries to obtain exclusive access to a resource that's already locked by a lower priority thread. This phenomenon is expected in threaded applications and cannot be avoided in most cases.

A worst scenario (defined as unbounded priority inversion) would consist of three threads A (high priority), B (medium priotiy) and C (low priority), all relying on a common resource. Imagine a situation where thread C starts first and locks the resource. Thread A (which is of higher priority) then wakes up, preempts thread C and tries to access the common resource. Since the resource is already locked, thread A will go to sleep. If thread B wakes up at this point it will delay execution of thread C since it runs with a higher priority. End result is that thread C is delaying thread A (which does not make sense priority-wise) and thread B is not giving thread C a chance to complete its work. This unbounded priority inversion situation will persist as long as thread B keeps executing, which has some obvious latency effects on thread A.

Two mechanisms are available to help with this situation:

- Priority inheritance
- Priority ceilings

6.1 Priority Inheritance

With priority inheritance, a low priority thread that holds a mutex can automatically have its priority increased in order to complete its work faster and let a higher priority thread gain access to the lock more quickly. End result is better response time for high priority threads and better predictability.

It is worth noting here that recompiling the application may be required in order to benefit from priority inheritance. Since this feature depends on a synchronization mechanism called *pi futexes*, applications that were compiled under an older glibc release will not get access to the corresponding feature.

6.2 Priority Ceilings

The validity of priority inheritance as a way to mitigate problems associated with priority inversion has historically been a debate in the realtime systems community. Another way to address this unbounded priority inversion problem is to design the application in such a way that low priority threads get a temporary boost when they acquire a resource. This makes it impossible for other threads to preempt the one that's currently holding the resource. Moreover, this can minimize the time spent while the resource is locked since the thread that holds it can now run at a higher priority. This requires use of some functions available in the pthread interface:

```
#include <pthread.h>

pthread_mutexattr_setprotocol(&attr,
 PTHREAD_PRIO_PROTECT)
pthread_mutexattr_setprioceiling(&attr,
 PTHREAD_PRIO_PROTECT)
```

7 Scheduling

The `sched_yield()` system call has historically been very popular with developers preoccupied with application latency. This function used to make it possible for a process to give up its share of CPU time and let the scheduler determine what should be executed next. The idea was that without calls to `sched_yield()`, the process would consume its CPU time slice and eventually get replaced with another process. This would obviously takes a relatively long period of time to complete and the process would be ready to run again after `sched_yield()` would return.

There are challenges associated with `sched_yield()` usage: this system call gives very little visibility to the scheduler about what the application is expecting to do next. Moreover, POSIX is extremely vague about what should happen with the calling process.

Use of the `sched_yield()` system call is now discouraged under the real time kernel. The new Completely Fair Share (CFS) scheduler was designed in such a way that `sched_yield()` should not be needed in any case. In fact, `sched_yield()` now accomplishes essentially nothing if it's called from a realtime process. Moreover, based on our experience we can conclude that in most cases we can actually re-architect applications that depend on `sched_yield()` in such a way that `pthread_mutex_*` calls or condition variables can replace it. These methods work much more nicely with the realtime kernel, and provide more visibility to the scheduler. It can thus make better decisions.

Under normal circumstances, on a system running the realtime kernel, using `sched_yield()` should not result in better results. For instance, in the case of two CPU-bound programs we extracted the ps output below. We can see that the two processes were allocated the same CPU time even though one of them, `loop_yield`, was making `sched_yield()` calls in every loop:

```
  PID USER PR  NI  S %CPU TIME+   COMMAND
15021 dd   20   0  R 50.1 1:42.33 with_yield
15022 dd   20   0  R 50.1 1:42.96 without_yield
```

There are very rare cases where application developers might actually need true `sched_yield()` capabilities. It is possible, with the realtime kernel, to turn `sched_yield()` into a more aggressive mode that will move the process that makes the call at the very end of the rbtree. This behavior is close to what is available under the standard kernel. With this sysctl turned on (`kernel.sched_compat_yield=1`), a process such as `loop_yield` would end up never getting invoked on the CPU if it was competing against another process that kept the CPU busy with no `sched_yield()` calls:

```
  PID USER PR  NI  S %CPU  COMMAND
15044 dd   20   0  R 99.5  0:25.94 without_yield
15046 dd   20   0  R  0.3  0:00.06 with_yield
```

8 Signals

Relying on signals to execute specific code in a realtime application is generally considered a bad idea. The code paths involved in the Linux kernel vary depending on the interface used: it is thus very hard to make this operation deterministic.

A better alternative to signals is to use POSIX Threads to distribute the workload and use condition variables, mutexes or barriers to let them communicate together. This method also provides much greater visibility to the scheduler, which can then make the right decisions to prioritize tasks.

If eliminating signals is not possible at all, we suggest creating one or multiple threads which will be blocking on the `sigwait()` system call. Assuming all other threads in the application have blocked signals, the signal handler thread will be the only one processing the corresponding code. This can lead to better signal response time and will make the entire application more deterministic by eliminating interruptions due to signals.

9 ioctls

Some applications make heavy use of the `ioctl()` system call. This is generally used to acquire control data or statistics from hardware devices. One aspect about `ioctl()` is of critical importance for realtime applications: the code executed in the kernel is invoked by default while the Big Kernel Locki (BKL) is held. The BKL is required in a number of different contexts in the kernel. Executing the `ioctl()` system call can end up delaying other threads or processes, and ultimately produce a large source of latency in the application.

A special unlocked version of the `ioctl()` system call may be provided by some drivers. Whenever that is true, any `ioctl()` call will automatically rely on the unlocked version. For that reason, unlocked `ioctl`s are generally considered safe for use in realtime environments. Do keep in mind, however, that the `ioctl` behavior (and the amount of time it takes to return) is ultimately dependent on the driver implementation, where some amount of locking can also happen and where latency may be introduced.

Developers of applications depending on the `ioctl()` system call can address latency issues by first assessing the delay caused by `ioctl()` in the application itself. If that delay is not acceptable, we recommend:

1. Verifying if a version of the driver contains a BKL-free `ioctl` implementation. If so, use it!
2. Wrap any `ioctl()` invocation around a low-priority thread of execution that will not affect the latency-sensitive parts of the application.

10 References

Arnaldo Carvalho de Melo. *Earthquaky kernel interfaces.* `http://vger.kernel.org/~acme/unbehaved.txt`. 2008.

Bill O. Gallmeister. *POSIX.4 Programmers Guide - Programming for the Real World.* O'Reilly and Associates, 1995.

The GNU C Library - Memory Allocation
`http://www.gnu.org/s/libc/manual/html_node/Memory-Allocation.html`

Philippe Gerum, Karim Yaghmour, Jon Masters, Gilad Ben-Yossef. *Building Embedded Linux Systems.* O'Reilly, 2008.

Real-Time Linux Wiki
`http://rt.wiki.kernel.org`

Red Hat Enterprise Linux Real Time Wiki
`http://rt.et.redhat.com`

rtcheck
`ftp://ftp.redhat.com/pub/redhat/linux/enterprise/5Server/en/RHEMRG/SRPMS/rtcheck-0.7.4-2.el5rt.src.rpm`

TLSF: Memory Allocator for Real-Time
`http://rtportal.upv.es/rtmalloc/`

Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*

Programme Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
James Bottomley, *Novell*
Bdale Garbee, *HP*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Alasdair Kergon, *Red Hat*
Matthew Wilson, *rPath*

Proceedings Committee

Robyn Bergeron
Chris Dukes, *workfrog.com*
Jonas Fonseca
John 'Warthog9' Hawley

With thanks to
John W. Lockhart, *Red Hat*