Temporal inventory and real-time synchronization in RTLinuxPro(R)

Victor Yodaiken

Finite State Machine Labs (FSMLabs)

September 23, 2004

Abstract

This paper is a short tour of the synchronization methods for real-time with special attention to the RTCore (RTLinux) real-time operating system. A glossary is provided for some tutorial material.

1 Real-time software is like an automobile factory

Synchronization operations impose temporal order on a software system by forcing some computations to wait until other computations complete. Waiting and negotiating over which computation should take place next can easily absorb significant fractions of processing power and can cause significant delays. To compensate, engineers can specify more powerful processors and more resources, maybe even multiple processors — but such things are not free. Even worse, synchronization can easily produce timing failures or deadlocks that may elude testing and this becomes more of a hazard as more complex and sophisticated synchronization operations are employed. It's not uncommon for control system software to spiral in complexity as more hardware is added to provide the compute power needed for synchronization, necessitating more synchronization which requires more resources and also more sophisticated special case software to compensate for hard to find sporadic timing problems and so on.

This note is a tutorial on synchronization of real-time software, focused on applications running under the RTCore operating system (RTLinux is RTCore plus Linux). Throughout the design and ongoing development of RTCore, we have tried to solve problems by simplification instead of by adding features. One of the most brilliant practitioners of this approach to engineering is Taishi Ohno, who helped develop Toyota's manufacturing process. Ohno noticed that large parts inventories in production plants were a significant cost and that they masked inefficiencies. When inventory was reduced and the factory was designed to run, *just-in-time* problems in quality and reliability were exposed and fixed. The result was greatly increased productivity. Using up parts inventory and using up slack time "inventory" during synchronization are analogous. If a task is scheduled to start at time t and only starts at time $t+w$ because of delays waiting and negotiating for resources, we have to relax deadlines and build slack time into the system to compensate for w. The slack time is temporal inventory — time we stockpile in order

to compensate for resource conflicts, the overhead of arbitration, and other scheduling inefficiencies. We cannot expect to reduce inventory to zero or to do without any synchronization at all, but we can produce cleaner and more reliable software by working to minimize synchronization time.

2 Basics

Some of the technical terms here are defined in a glossary section indicated by a reference to such as [G1] for glossary reference 1. I will use the term *task* to indicate a section of executable code plus state. Tasks include processes, POSIX style threads[G12] and interrupt handlers[G7].

2.1 Goals

Designers of any concurrent[G3] software must ensure that synchronization[G2] operations are (1) usually fast (2) take up a small percentage of total computation time, (3) do not deadlock[G1] and (4) do not defeat modularity. The last item is perhaps the only surprise on this list, but synchronization points can be among the most dangerous global data structures — causing scheduling dependencies between components that are logically unrelated. Designers of real-time[G4] software must impose two additional requirements for synchronization operations; (5) worst-case delays are bounded and small, and (6) interactions among logically distinct components do not cause unacceptable timing changes. If the average delay for starting a critical task is within tolerance but the worst-case is out of tolerance, the system may still fail spectacularly. The often misunderstood Mars Pathfinder[G8] near-disaster is an example of how a hidden shared synchronization point changed system timing enough to cause a catastrophic failure. I will go into some depth on pitfalls in section 4.2, but satisfying all six constraints is usually not too hard as long as sensible design procedures are followed.

2.2 Priorities

Real-time programming is generally based on a priority ranking of tasks and simple rule.

Rule 1 *Nothing should delay the highest priority runnable tasks.*

Priority makes synchronization more difficult because priority and synchronization can come into conflict and because too many priority levels can sabotage performance.

2.2.1 Too many levels

When a system spends its time rescheduling differently prioritized computations and doesn't get enough work done, this may be a sign that there are too many computations with too many levels of priority. If you have n computations at different priority levels, the worst case is n preemptions with no real work done. One question to always ask is *what is the difference between computations at different priority levels?* Suppose you have 100 priority levels. Can you clearly specify the rationale for the priority level differences between computations at priority 95,96,... 100?

Scheduling large numbers of prioritized computations is a problem that is incorrectly believed to be solved by rate monotonic scheduling[**?**]. RMA provides a formula for deriving priorities from rates, based on the sensible rule of thumb that the highest frequency tasks are often the most important. There is a large literature on rate-monotonic scheduling, and it may be useful for your application, but RMA

assumes a great deal about the application that is not generally true and one is better off simplifying when determining priority becomes too complex to do by hand.

Rule 2 *Do not add priority levels unless you can precisely specify why tasks at one level should pre-empt tasks at the next lower level. Do not add tasks to avoid thinking about control flow.*

2.2.2 Too much pre-emption

The complementary problem is when a critical task does not get to finish its computations because it is interrupted by higher priority tasks. Here we have to face an unfortunate reality: we are designing real-time systems without the appropriate mathematical and engineering tools. The current state of the art provides no way to precisely analyze the worst case timing of anything but the most elementary computations. Even in the simpler cases, the uncertainties of cache misses, DMA, memory, and processor pipelines cause great difficulties. Note that RMA depends on a quite precise calculation of worst case timing for each task — something that is generally not available. Testing and approximate analysis are the only tools we have. Which brings us to the next rule.

Rule 3 ***Design so that you can reliably test worst case timing****. Keep real-time tasks simple and their primary control loops simple because if there are too many code paths, tests and analysis will not find all code paths or identify all delays.*
And run tests for long periods under varying worst-case loads. Do not assume you know what is the worst case load.

2.2.3 Priority Inversion

Priority inversion is the term used to describe violations of rule 1 due to synchronization. If task A has exclusive use of resource R and higher priority task B becomes runnable, B must wait for A - inverting priorities. Priority inversion is possible when there is a resource that requires synchronized access and is shared between tasks with unequal priority. If we think about it, such situations are intractably bad. If Task A is more important than task B, and A can preempt B, then what possible reason can we have for allowing B to get exclusive use of a resource that A requires? Priority inheritance and priority ceiling algorithms are widely, and incorrectly, believed to fix priority inversion problems. I'll look at these in more detail in section 4.2.1, but the best solution is to avoid the entire problem.

3 Just-in-time scheduling

The best synchronization is no synchronization. Synchronization permits us to use sloppy schedules and to delay execution of a scheduled task until the resources are ready.

Rule 4 ***It is best to schedule just in time****. Whenever possible, real-time systems should make sure that no two real-time tasks are scheduled to run during a common interval and that scheduled real-time tasks are never blocked waiting for resources.*

3.1 Scheduling just-in-time

For many years, engineers designed real-time software as explicit loops or slot schedulers and these are still perfectly good designs for many software systems. A classical data acquisition application for example, may simply poll a set of sensors on a periodic schedule. No synchronization required. A control system may stagger a data acquisition task and a control output task. If the periods and start times are correctly chosen, the system runs just-in-time. No synchronization is needed. As an example, suppose we poll the sensor every k nanoseconds and generate an output every 3k nanoseconds. In RTLinux we could design this quite simply. We would create two real-time threads. The input task waits for its time to run, gets data, queues it, and loops. Note the three lines marked DEBUG TEST. These would be used during test phase to make sure that the two tasks never overlap in execution time.

Example 1 *Alternating threads: input*

```
task_input:
    clock_gettime(CLOCK_REALTIME,&t0);  // read the time
    while(!stop)
    {
        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &t0, NULL);
        /*DEBUG TEST*/if(sem_trylock(&overlap_sem))fail();}
        collect_data_from_a2d_device(data);
        queue_data(data);
        timespec_add_ns(&t0,INPUT_DELAY_NS);
    }
```

The output task does much the same thing, but it waits for the offset interval before starting to ensure that the two tasks are out of phase.

Example 2 *Alternating threads: output*

```
task_ouput:
    clock_gettime(CLOCK_REALTIME,&t1);  // read the time
    timespec_add_ns(&t1,OFFSET_NS); // compute start time
    while(!stop){
        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &t1, NULL);
        /*DEBUG TEST*/if(sem_trylock(&overlap_sem))fail();}
        dequeue_data(data);
        compute_output();
        output_control();
        timespec_add_ns(&t1,OUTPUT_DELAY_NS);
        /*DEBUG TEST*/sem_post(&overlap_sem);
    }
    ....
```

What happens if there is an overlap? This can only happen if the execution times of the threads are too long for the period. Testing and analysis should eliminate the possibility of such an error during

development. Making the two threads equal priority ensures that in the case of a failure, there will be no preemption – as long as the clock_nanosleep operation is the only blocking operation of either thread. For some applications, the fail operation may allow for runtime recovery and in this case the overlap test may be left in production code.

This same system can be exceptionally efficient on an SMP multi-processor. In RTCore, when you create threads, you can set the CPU attributes of the two threads so they run on different processors. The offset is still needed to make sure that there is fresh data to output when the output thread runs.

This section has illustrated an old and time-tested design technique that requires absolutely no synchronization. No time is wasted negotiating over access to shared objects, there are no false starts, and no required preemption. For many projects a design like this can reduce processing requirements — potentially allowing for a lower power lower speed processor.

3.2 Asymmetric just-in-time and non-blocking I/O

One design issue that can cause programmers to abandon just-in-time is the use of low priority non-real-time tasks for housekeeping and other functions that are either not critical or that can use buffering to compensate for timing fluctuations. For example, consider a system where a real-time task collects video frames from a camera and a second task displays the video and stores it in a file. Given a big enough buffer and a "makes progress during any 1 second interval" assurance from the non-real-time scheduler, the second task does not need to be made real-time. Real-time applications of this kind consist of a real-time component and a (typically much larger) non-real-time component. RTCore/RTLinux is is specifically designed to simplify such systems by running an non real-time operating system from a real-time kernel.

Rule 5 Asymmetric just-in-time: Pre-empting and blocking non-real-time tasks is not a violation of the just-in-time scheduling rule.

Note that rule 5 is not the same as saying that pre-empting lower priority threads is ok. All real-time threads have time constraints. Pre-empting non-real-time threads is entirely different from pre-empting real-time threads.

The standard method to connect the real-time and non-real-time components in RTCore is via an asymmetric device interface in which the real-time side always sees a non-blocking interface. For example, consider the following code from an application that logs data from an analog-to-digital device.

Example 3 *Using asymmetric FIFO*

```
clock_gettime(CLOCK_REALTIME,&t);
while(!stop)
{
    collect_data_from_a2d_device(&data);
    write(fd,&D,sizeof(D)); /* async write to rt-fifo */
    timespec_add_ns(&t,DELAY_NS);
    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &t, NULL);
}
```

If the `write` cannot advance because buffers are full, data is simply dropped[1].

On the other side of the write may be something as simple as a UNIX shell script:

```
cat < /dev/rtfifo0 > logfile
```

This would be implemented by a UNIX process that had no real-time requirement, could be preempted at will and made to block when no data is present in the fifo. The `read` operations that `cat` does to collect data are synchronous and block when there is no data. The `write` operations that `cat` does to `logfile` are also synchronous and block. The real-time task preempts any UNIX process every `DELAY_NS` nanoseconds. But none of this violates the just-in-time rule and none of it interferes with precise timing of real-time tasks because *the* `cat` *process is not a real-time task.*

3.3 Event driven scheduling

Blocking operations, operations that may cause a task to be suspended until data is available or a lock is released must be treated with great care in real-time programming. If our task is time constrained, what is it doing invoking an operation that may suspend it for some indefinite period? Often times code analysis shows that blocking operations are used because programmers are not thinking carefully about system design and are reusing non-real-time programming techniques in the wrong place. But event driven programming is one of the most useful methods in real-time software and it depends on blocking threads until some event triggers the activation of the thread. Blocking I/O operations and semaphores[G6]are the traditional tools for this type of programming.

Consider the following code

Example 4 *Blocking producer.*

```
Thread A:
do {
     read data by a blocking operation
     process the data
     if space on output queue
          enqueue data
     else
          discard data
} while(not done);
```

Thread *A* uses the blocking operation to drive its schedule. After completing the loop, the task blocks waiting to run again at when new data shows up. The same design can be implemented using semaphores. Suppose that an interrupt handler is called when data is produced by a device.

Example 5 *Blocking consumer*

```
Thread A:
do {
```

[1] Error handling is omitted in this example, just to make the example simpler. In a real system, we dropping data might be correct or we might want to take some corrective action.

```
     semaphore decrement. /* the interrupt routine does post*/
     do something
     yield
} while(not done);
```

In this case the schedule for A is determined by an interrupt handler that presumably does some simple processing and leaves A to complete the job. Just as in the first case, the blocking operation drives the schedule.

Here's an example where a high frequency thread checks the temperature, does a simple operation, and wakes up a thread that will do some background processing if the temperature is too high

Example 6 *Semaphore powered event driven.*

```
check_temp_thread(){
  if(temp > 100) sem_put(&sem_help_me);
  timespec_add_ns(&t0,SMALL_PERIOD_NS);
  clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &t0, NULL);
}

   ...
help_thread(void){
  sem_get(&sem_help_me);
  /* sound the alarm */
       ...
}
```

As a second example, consider a producer and consumer on an asynchronous non-blocking queue where we would like the consumer to block if there is no data.

Example 7 *Producer/Consumer 2*

```
producer() {
        pq_enq(&q,newpacket());
        sem_post(&q_sem);
}
// we assume here there is never more than one consumer task
consumer() {
        sem_wait(&q); // when we pass here we know deq will succeed.
        m = pq_deq(&tx);
}
```

We might do more sophisticated flow control and error handling by putting low level control into an interrupt handler and activating an error thread if space gets short.

3.4 Interrupt handlers

Interrupt handlers or interrupt service routines (ISRs) are functions that are activated asynchronously by hardware events. An interrupt handler[G7] version of example 3 is simpler than the thread.

Example 8 *Interrupt handler using FIFO.*

```
rtl_request_irq(A2DDEVICEIRQ,handler);
...
handler()
{
    collect_data_from_a2d_device(&data);
    write(fd,&D,sizeof(D)); /* async write to rt-fifo */
    return;
}
```

RTCore, following the UNIX tradition, runs interrupt handlers in the context of whatever thread was executing when the interrupt is caught. The motivation for this approach is to avoid the cost of switching to a handler thread when that is not necessary.

Rule 6 ***Interrupt service overhead minimization***. *The least possible context should be saved and restored during an interrupt service operation.*

When a handler starts in RTCore further interrupts are disabled and the current interrupt is masked. When the handler is ready, it must unmask the interrupt but the handler should never re-enable interrupts directly. The RTCore kernel re-enables interrupts when it restores the context of the interrupted thread.

Example 9 *Interrupt Handler*

```
rtl_request_irq(A2DDEVICEIRQ,handler);
fd = fd_normal;
..
handler(){
collect_data_from_a2d_device(data);
  reenable_the_device(); // but interrupts are still disabled on this cpu
  if (write(fd,data,DATA_SIZE) != DATA_SIZE) {
    if (!overflow) {
       write(fd_error,DATA_SIZE));
       overflow = 1;
       fd = fd_overflow;
       sem_post(&failure_sem);
     } // else drop the data
    return;
  }
}
```

The basic idea here should be clear. The second thread is scheduled *just-in-time* by a semaphore post operation and this is a common and invaluable technique. In many cases where a static just-in-time schedule is impossible to design, it is possible to schedule real-time threads using semaphores. What does the thread waiting on the semaphore do when it wakes up? This depends on the data semantics. Possibilities include turning on an error indicator, some flow control, interpolating data and increasing the priority of the UNIX process at the other end.

Suppose, that the producer is interrupt driven as above, but the consumer is also a real-time task. For example, the second task may wait for a counting semaphore and a post operation from the first task may be the correct way to wake it up. In this case, the schedule adapts to data arrival times and both criteria of the just-in-time rule are met.

3.4.1 Interrupts and semaphores

It's worth looking at semaphore wakeups in a little more detail. Rule 1 determines much of the design of RTCore semaphores. A sem_post operation has the following semantics

Example 10 *Semaphore semantics:*

```
sem_post(s){
     atomically{ increment s
                 if there are waiters wake the highest priority one
 }
     if we woke a higher priority thread, on our CPU, switch.
}
```

A sem_post operation will reschedule the system so that if a higher priority thread is released by a post operation, the time that the higher priority thread waits to run is minimized. This may seem to be obviously good, but what what happens when an interrupt service routine calls sem_post? If high priority thread T′ is waiting on a semaphore and thread T is the running thread and an interrupt causes an ISR to run in the context of T and call sem_post then a thread switch will preempt T.

T	running	ISR	sem_post	...	resumes	ISR returns	runs
T'	waiting		wake	runs ...	pre-empted		

As a result, ISR code must allow for possible long delays between a sem_post and the completion of the ISR. It's generally wrong to have code that re-enables a hardware interrupt after a semaphore post. This is an example of one of the places where real-time constraints expose system behaviors that can easily be hidden on non-real-time systems. In the average case, there will not be wakeups of higher priority threads so it is possible to keep good average latency by simply setting a flag in sem_post and calling the scheduler in the low level ISR return code. But average case does not help us. We have to immediately switch and pay the price of needing some more vigilance in ISR code development. In cases where this behavior is not wanted, RTCore provides both a `pthread_kill` to wake a waiting thread on a semaphore without an immediate re-schedule and the mutex operations are also specifically designed not to call the scheduler. Since POSIX semantics for mutexes are kind of complex already, we decided that an additional small delay in mutex_unlock would not be an issue.

3.5 Lock-free structures

Sometimes it is possible to design data structures so that even though they are shared, there is no need to synchronize access.

Rule 7 ***Lock free data structures*** *Where there are fast algorithms, use lock-free structures to avoid synchronization.*

In RTCore we have a couple of useful primitives for building lock-free data structures and a high speed lock-free queue. The POSIX sem_getvalue (atomic read) sem_trylock (atomic decrement if greater than 0, fail otherwise) operations which are exceptionally useful for atomic access to counters. RTCore also offers rtl_test_and_set which is passed a pointer and a bit position, and atomically sets the indicated bit and returns the *previous* value. On the x86 test and set bit uses a special instruction, on other machines it is more complex and some processors have other useful atomic primitives. But the programmer can just use rtl_test_and_set and ignore processor differences. To illustrate, consider a system for sharing a collection of buffers among a set of concurrent tasks. This algorithm works well as long as the search is not too long and it works just as well on multiprocessor machines as it does on uniprocessors. Furthermore, both the allocate and free functions are interrupt handler safe.

```
struct mybuffer { unsigned int flags; char data[DATASIZE];};
struct mybuffer B[NUMBER_OF_BUFFERS] = {0, }; //zero it

struct mybuffer *allocate_mybuffer(void){
        int i;
        for(i=0; i< NUMBER_OF_BUFFERS; i++)
             if(!rtl_test_bit_and_set(0,&B[i].flags))
                   return &B[i];

        return (struct mybuffer *)0;
}

void free_mybuffer (struct mybuffer *b){
           b->flags = 0; // store is atomic
           return;
}
```

Test and set is not free, however and SMP programmers need to be aware of the problem of cache ping-pong[G9].

The RTCore one-way-queues do not need locks or even test-and-set operations. The one-way-queues work when there is a single reader and a single writer. I'll illustrate with a packet routing system. The rx_handler could be called on interrupt, get a message and queue it for a thread that could modify and then queue the modified messages for output. The tx_handler would then be called on interrupts to recycle storage of transmitted messages.

Example 11 `#include <onewayq.h>`

```
typedef struct {int flags; unsigned char d[SZ];} *mypacket_t;
BUILD_OWQ(pq_t,128,mypacket_t,-1);
pq_t txq,modifyq;

rx_handler() {
    if(norxerror()){
        if(-1 == pq_enq(&modifyq,newpacket())) fail();
    }
}
tx_handler() {
   /* one transmit done. throw away sent packet */
   m= pq_deq(&txq);
   free_message(m);
}

...

modify_thread(){

    while(!stop){
        while( (m = pq_deq(&modifyq)){
            process(m);
            pq_enq(&txq,m);
        }
        timespec_add_ns(&t0,DELAY_NS);
        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &t0, NULL);
   }
}
```

In this example, violations of the just-in-time rule don't cause a problem. Even if the two interrupt driven tasks and the periodic thread all run in parallel on a shared memory multi-processor they can safely share the queues. The RTCore one-way-queues take advantage of the fact that for a circular array based queue, the head pointer and tail pointer are not shared – the consumer needs the head, and the producer needs the tail. And a very large number of shared data structures can be treated as single-producer/single-consumer queues. Note that we defeat concurrency problems here without explicit locks, simply by designing the system to avoid a contention that, at first look, seems unavoidable.

How do we cope with multiple consumers or multiple producers? In that case, one answer is to use locks. But note that we can have one lock for consumers and one lock for producers — effectively decoupling two groups of threads in accordance with rule 7.

4 Unavoidable synchronization

There are systems where scheduling conflicts are inescapable, but not as many as are generally believed. Unfortunately, sometimes we cannot guarantee that tasks obey the just-in-time rule. Before we discuss how to synchronize, consider why we need to synchronize. When careful design analysis, or more likely, software based on legacy architecture, cannot be designed just-in-time, synchronization is required.

I want to distinguish three types of synchronization that are often used in combination.

1. Atomic code blocks. Using primitives to prevent protecting blocks of instructions so that they are executed as a single block — as if they were a single atomic operation.

2. Synchronization via pre-emptible critical regions. This is the standard mutex-protected critical region.

3. Synchronization via server centralization.

4.1 Atomic code block

"Atomic" is a relative term and needs to be qualified. A block of code between mutexes is atomic with respect to thread switching: once the mutex is locked, no other thread can enter the code region. A block of code between interrupt disable and interrupt enable operations is atomic with respect to uniprocessor computations: no other computation on this processor can start until interrupts are re-enabled (unless the code causes an exception like a divide-y-zero or page fault). A block of of code between

Almost every modern processor provides methods of guarding a block of instructions so it can run atomically. On a uniprocessor, we can usually just disable hardware interrupts which prevents interrupts or preemption, on a multiprocessor we can use spin locks to prevent parallel computation. These primitive operations are the basis for all other blocking synchronization methods: including semaphores[2]

On a uniprocessor, any section of code can be made atomic by simply disabling interrupts. RTCore offers methods that are similar to those found on nearly every operating system. Here's a low level method, not recommended, but shown for understanding.

```
/* DANGER! don't do this use pthread_spin_lock instead */
rtl_stop_interrupts();
do some work // no preemption, no interrupts
...

rtl_allow_interrupts();
```

There are no fewer than four things to worry about in this simple example:

[2] Unfortunately, it is no longer possible to always rely on the atomicity of primitive atomic operations. Chip designers and firmware designers keep wanting to interfere with operating systems, mainly so they can "transparently" compensate for buggy hardware. In this cause, x86 processors have SMI and other interrupts that ignore operating system directives and advance anyway! This is really a tough one and can require careful selection of motherboards and BIOS software.

1. What if interrupts were disabled already when we entered this code? In that case, we enable interrupts at the end when we should not. This is easily solved. See below.

2. How long does do some work take? It had better be a small, simple and fast section of code because disabling interrupts strikes at the heart of our response time.

3. What about a multiprocessor system? Disabling interrupts means that this task won't be interrupted or preempted. On an SMP system, it is still possible for another processor to ruin the atomicity of the code. The answer here is spin locks - see below.

To solve the problem with multiprocessor systems, we need spin locks. The idea with a spin lock is that a task loops trying test and set until it succeeds and the task gets the lock. The task *spins* until it gets the lock. The POSIX defined spin locks are pthread_spin_lock and pthread_spin_unlock. In RTCore, the locking call disables interrupts before the lock is set, and unlocking restores the interrupt In a uniprocessor system setting and clearing the lock is omitted as an optimization (Linux spin-lock does this too). Note that once a thread has acquired a spin lock, it cannot be interrupted or preempted until it releases the lock or calls a blocking function of some sort (VERY inadvisable). Code within a spin lock should be fast and deterministic. Don't try to do complex computations with spin locks held.

The implementation of POSIX spin locks in RTCore is designed to prevent one of the standard spin lock errors. If you acquire a spin lock without disabling interrupts, it is possible to deadlock when a signal handler or preempting thread gets stuck on the same lock. Since the thread holding the lock is not able to run, the attempt to get the lock results in deadlock. In RTCore, pthread_spin_lock disables interrupts first and the unlock operation restores the previous interrupt state. This also solves our problem of re-enabling interrupts by mistake. That's why we recommend using the POSIX spin lock operations on both SMP and uniprocessor systems. The ugly parts are hidden in the call, the interface is a POSIX standard, and you won't get a nasty surprise 3 years later when someone runs your code on an SMP machine!

```
pthread_spin_lock(&myspin);
critical section // no preemption, no interrupts
...

pthread_spin_unlock(&myspin); // in a uniprocessor myspin is not locked
```

Suppose we altered our example above to allow for two receive devices sharing the same modify queue (we could perhaps do better by using multiple modify queues and having the thread loop through them, but let's suppose we had no choice.) In this case we would have two producers, needing a spin lock. The first rx handler could then be rewritten as follows.

```
rx_handler() {
   if(norxerror()){
      pthread_spin_lock(&modifylock);
      error = myq_enq(&tx,newpacket());
      pthread_spin_unlock(&modifylock);
      if(error)fail();
   }
}
```

Note that the consumer of the modify queue does not need the spin lock since there is still only a single consumer.

It's possible to construct much more elaborate lock free data structures and many very clever ones have been invented. [2] is a good start on the subject.

Just for completeness, here's how you would implement a spin lock from test and set lock. Note the optimization of not using the expensive test and set operation while we loop. The volatile key word is used to prevent the compiler from deciding it can read the memory location once, put it in a register and loop until the register value changes[3].

```
void pthread_spin_lock(rtl_spin_t *s){
    while(rtl_test_bit_and_set(0,&s->lock)){
           // don't use the expensive test and set,
           while( (volatile)s->lock & 1);
     }
}
```

Cache ping-pong is also a problem for spin locks.

For comparison, let's do a quick analysis of semaphore costs and it's here that the difference between real-time and non-real-time becomes painfully obvious. On Linux, where semaphores have been obsessively optimized the code for a IA32 processor is a macro that looks something like this:

```
down_semaphore:
     atomic_decrement_instruction address_of_counter
     jump_if_nonzero down_failed_code
```

The idea is that in the *common case* there will be no contention and down_semaphore takes one atomic decrement instruction and one failed branch and just falls through. On a fail, we jump to a "C" function since the semaphore blocking code is so costly anyways. For Linux, if 99.99% semaphore down operations have no contention, the cost of the failure case is not significant and if there are many failure cases, the problem should be solved by reducing contention. For a real-time programmer, this solution is a partial solution but we need to always consider worst case.

See Vahalia [5] for an excellent textbook coverage of locking.

4.2 Guarded critical sections

Semaphores were used in section 3.3 as signaling mechanisms. They can also be used to guard critical sections of code. A binary semaphore (mutex) may be locked as a thread enters a critical section and unlocked as the thread leaves the critical section. In theory, semaphores have an advantage over disabling interrupts and setting spin locks by imposing less delay for more important tasks and allowing more efficient use of the processor. In practice, semaphore guarded critical regions need to be used with care and are often much less efficient than the alternatives.

When they are correctly used, semaphore guarded critical regions are fine. I particularly recommend use of the POSIX mutex conditional waits to avoid the standard pitfall of missed wake-ups[G10].

[3] Not all "C" compilers do volatile properly. Sometimes you want to use an assembler escape to force correct behavior.

4.2.1 Priority and semaphores

The biggest problem with semaphore guarded critical regions is that priority scheduling interacts with semaphores in a very ugly way. Let's first look at the simplest case.

A high priority task simply preempts a lower one, whether the lower priority task holds a lock or not. If the time required for *spin lock; critical; spin unlock* is more than the time required for *semaphore wait*, then a high priority task will be delayed less by a semaphore protected critical section than a spin-lock protected region. But, this supposes that the critical region is relatively expensive and that the high priority task does not need the semaphore. If the critical section cost is small, then spin locks work better. One of the nice things about spin lock protected critical sections is that they are totally modular: nobody waits more than the computation time of the slowest critical section.

Rule 8 *If you are tempted to put a critical section within semaphores in order to allow preemption during the section, try shortening your critical section.*

If semaphores used to guard critical regions are shared between threads that are not all the same priority, the system will encounter *priority inversion*. Suppose that task High and task Low share a resource guarded by a semaphore or some other blocking synchronization primitive. The standard worst case is:

1. Low acquires the lock;
2. High preempts and there is a context switch;
3. High blocks on the lock, runs the scheduler;
4. Low restarts after a second context switch and then completes the protected operation, releases the lock,and triggers scheduler.
5. High restarts after a third context switch and completes the protected operation and releases the lock

To make this a little more concrete, if over-optimistic, suppose that context switches take 200 cycles, successful lock operations take 10 cycles, blocking lock operations take 100 cycles and the scheduler takes 50 cycles. Then if the computation guarded by the semaphore takes t cycles, High waits $100 + 50 + 200 + t + 10 + 100 + 200 = 660 + t$ cycles for Low to complete. Note that if Low used a RTCore spin-lock, then the total wait time for High would be $t + 10 = 210$ (in both cases I'm ignoring the cost of the first context switch to run High since we have to pay it anyways.) The advantage of the semaphore guarded method is that SuperHigh can pre-empt both High and Low as long as SuperHigh does not also need the same semaphore.

The analysis above is optimistic for a couple of reasons, most notably the assumption that Low will get to run uninterrupted while High waits. Many multi-tasking systems have a wonderful scheduling property called "liveness"which guarantees that every task will get to run. In traditional UNIX systems, the longer a process waits to run, the higher its priority, and the longer a task uses the CPU, the lower its priority. One result is that a task that holds a lock will eventually get to advance and release the lock. Most real-time systems are not "live". A low priority task may be delayed, indefinitely by higher priority tasks. If the low priority task is indefinitely delayed while it holds a lock that a higher priority task is waiting for, we have *unbounded priority inversion*.

It is widely, but incorrectly, believed the *priority ceiling* and *priority inherit* mutexes solve priority inversion. The actual purpose of these methods is to solve only *unbounded* priority inversion and they do that at a cost. Priority ceiling associates a mutex with a priority so that if a lower priority thread locks the mutex, it gets the higher priority while it holds the mutex, and if a higher priority thread tries to lock the mutex, there is a failure. The cost is a priority switch on a mutex acquisition. This may or may not be a big deal, depending on how the operating system is implemented — but think about what happens to a priority ordered run-queue when a thread acquires a priority ceiling mutex. Priority inheritance is a dynamic version of the algorithm that promotes a thread that holds a mutex to the highest priority of any thread waiting for the mutex. See [6] for a detailed analysis of what is wrong with priority inheritance, but note the intractability of the problem. In the example above, the high priority thread is required to wait for the low priority thread, by design. Even if the various priority juggling algorithms worked well (and they do not), they could not cure the design error of making the high priority thread wait for the low priority thread that owns the semaphore. And the complexity of these algorithms increases the cost of avoiding the spin locks. Unfortunately, we can often create such situations unthinkingly by overuse of mutex or semaphore guarded critical regions. In many cases, the purported *gain* of permitting preemption during the critical section is not worth the cost of synchronization.

I wrote a completely unoptimized enqueue function, ran it through the gcc compiler with optimization turned off and counted a total of under 20 instructions in the entire routine - the code path is even shorter. Suppose we multiply by 5 to get 100 instruction cycles. You can see that enabling preemption during execution of this operation gains us very little — and it can make the worst case much worse.

4.3 Avoiding priority inversion

Rule 9 ***Semaphores guarding critical regions should never be shared by tasks of different priorities****. There is never a need to violate this rule.*

Suppose we have a critical section that seems to contradict rule 9.

1. Make sure that there are no just-in-time solutions, the slow operations cannot be delegated to the non-real-time side, and there are no lock-free solutions. If this fails, go to the next step.

2. Apply rule 8 and make the critical section so fast that spin locks work better. All critical sections that refer to the same shared data need to be optimized in this way in any event.

3. If we have concluded that there is an absolute requirement for a slow atomic operation on the data structure, design the system to make sure only tasks with the equal priority access this data structure. If we cannot do that, it means that we have designed a system so that some task A is more important than some task B but B still has the ability to block A. Try very hard to remove this contradiction. If this fails, go to the next step.

4. Essentially, priority ceiling is an indirect method of building a server. If you need a server, build one directly don't make an obscurely hidden one that can cause surprises later. Threads should be cheap in the OS (they are in RTCore), make use of them. Servers are easy - see the example below.

Suppose we have a data structure D that is accessed by methods $m_1, \ldots m_n$ (these are all the operations on this data structure). After much analysis, we see that D must be shared among tasks of unequal priorities, and that just-in-time and lock-free cannot be applied. Furthermore at least one of the methods takes too much time to permit a spin-lock guard and there is no way to shorten it. Just to make things interesting, say that there are many possible tasks that need to access D and that we cannot use reader/writer locks or otherwise structure access. There are several alternative methods here, but one useful one is to use RT-FIFOs and semaphores to construct a server. If the RT-FIFOs turn out to be too slow, then we were probably wrong about how computationally expensive the methods are, but we could replace the fifos with one-way queues if needed. The server will serialize all operations on D. No other task will directly operate on D, instead the tasks will make requests to the server. The method is simple. Create a FIFO. Size the FIFO to be $k * r$ where k is the size of a request structure and r is the max number of requests that can be outstanding. The request structure may look like

```
struct myrequest{
              sem_t *wakeme;
              enum method_t method;
              int priority;
              int arg1,arg2,arg3;
              }
```

A task makes a request by filling in a myrequest structure with a pointer to a semaphore that is private to that task, a method identifier naming the operation to be executed, and some arguments. The task then makes the request and waits.

```
r.wakeme = &mysem;
r.method = SortTheDatabase;
r.arg1 = 0;
r.arg2 = MAXROW;
write(req_fd,&r, sizeof(r));
sem_post(&server_wake);
sem_wait(&mysem);
```

The server looks something like the following.

```
while(1){
      sem_wait(&server_wake);
      while(read(req_fd,&req[next_free()],sizeof(struct myrequest)))
         == sizeof(struct myrequest)
            mark_used(next_free());
      if(stuff_2_do()){
            i = most_important();
            do_request(i);
            sem_post(req[i].wakeme);
            mark_done(i);
      }
}
```

The priority of the server should be set to depend on the importance of operations on D. One guide will the highest priority thread to touch D. A more sophisticated server could abort long running low priority requests when higher priority ones showed up.

A Glossary

Glossary Definition 1 *Deadlock* is the condition where some task waits for another task to release a resource and the second task is waiting (maybe indirectly) for the first task to release a resource: so there is no forward path. For example if task A has a code sequence $locksem_1; locksem_2; work; releasesem_2; releasesem_1$ and task B has a code sequence $locksem_2; locksem_1; work; releasesem_1; releasesem_2$, then if A is locks 1 and then B locks 2, neither will be able to progress to release. Another classical deadlock case is cause by taking a spin-lock without disabling interrupts. Suppose that task A sets spin-lock L and then is interrupted by a service routine that spins on L - there is no way to advance.

To prevent deadlock, there are two answers. First, use atomic or safe operations whenever possible instead of building your own synchronizing systems. The RTCore RT-FIFOs offer a safe and efficient method of exchanging information between threads that has been well debugged and validated by years of use. Second, *KEEP IT SIMPLE*. The times in which it is a good idea to acquire multiple locks are exceptionally rare and it is almost always the case that a design of that form is just plain wrong. Never acquire a lock in one function and release it in a second — locks and unlock should be close together in the code text (unless the lock is a signaling lock). One of my cautions about using mutexes anywhere is that it is easy to create mutex chains without thinking about it. If function f gets mutex a and calls function g which gets mutex b and we modify g to call h which also gets a, we have created a deadlock. Some people advocate recursive mutexes as a solution — based on the theory that it is ok to write software without understanding what components lock which mutexes.

Glossary Definition 2 *Synchronization* Even something as simple as $x = x+1$ is a three step operation on most processors: (1) fetch, (2) increment, and (3) store the result. On a multiprocessor computer, as one processor goes through these three steps, it is possible that another processor will read x or store a new value in x *in parallel*. For example, suppose that x contains the number 100 and processor A and B both start to increment at the same time. If A and B complete 1 and 2 at the same time, then both will store the value 101 to x in step 3 — while the correct value should be 102. The same problem occurs in single processor computers due to input/output operations and interrupt processing. Input/output devices capable of direct memory access (DMA) can change memory in parallel with the processor. Interrupts cause the processor to jump to an interrupt handler, which may call a scheduler. In such a situation, task A may be interrupted, say, after step 1, and then if task B is run and modifies x, when task A is resumed it will store the wrong value.

Task A	fetch x				store x
Task B		fetch x	add 1	store x	

Glossary Definition 3 *Concurrent* is used to refer to either actual parallel computation or pseudo parallel computation caused by interrupts and preemption (when the running task is forced to stop to let a second task run). In either case, competing operations on the same data can result in data corruption.

Synchronization operations act as gateways to ensure that only one task can use a shared resource at one moment.

Glossary Definition 4 *Real-time* Definition of real-time software

Glossary Definition 5 *Deadlocks* A deadlock is when no task can progress because each is waiting for another one to release some resource. For example, if we have 2 buffers and task A and B both need to allocate 2 buffers, and each allocates 1 and then blocks waiting for the other to release the other buffer. or live-lock[4]

Glossary Definition 6 *Semaphores* Semaphores are perfect for event driven programming although they certainly were not designed with real-time in mind. In Dijkstra's famous paper on the *THE* operating system introduced semaphores [1], he notes specifically that the *absence* of timing constraints on processes was a key design consideration. As Parnas notes, THE was not a particularly fast system even in non-real-time terms [3].

Glossary Definition 7 *Interrupt handler*.An interrupt handler is a function that is called in response to a hardware interrupt. For example, a data acquisition device may signal the processor that it has collected a sample. When hard interrupts are enabled and such a signal is asserted, the processor will save a small subset of processor state and jump to low level operating system code that will, in turn, call any interrupt handlers that users have installed. In RTCore, real-time interrupt handlers can be installed via the rtl_request_irq operation or via the POSIX sigaction call. Interrupt handlers run effectively at the highest priority level, pre-empting any running task but they run at the priority of the interrupted thread.

One of the key implementation differences between UNIX and VMS operating system was that VMS ran interrupt service routines as processes, but UNIX ran interrupt service routines in the the context of the running process. The rationale for the UNIX implementation is to make interrupt service more efficient by eliminating one context switch and some scheduler overhead. RTCore runs interrupt handlers in the context of the current thread for the same reason. The trade-off for running interrupts in the context of the current task is that system state is harder to understand. Until you get used to the idea, running an interrupt handler in any random executing task is counter-intuitive. Remember that interrupt handlers run outside of the normal priority scheme: handlers are always higher priority than the current thread. To take advantage of the low call overhead, interrupt service routines should be small and fast and should activate tasks for additional work if needed.

Glossary Definition 8 . According Glenn Reeves[4] the problem was buried inside the VxWorks I/O subsystem. A low priority task (the `ASI/MET` task) was connected to the high priority `bc_dist` task via a VxWorks IPC mechanism. The `ASI/MET` task made a `select` call that, within VxWorks, invoked a semaphore protecting the file descriptors under the select. One of these file descriptors was for the IPC pipe between `bc_dist` and `ASI/MET`. Before the semaphore could be released, the `ASI/MET` task was preempted by some medium priority tasks. The next invocation of the higher priority `bc_dist` task then stalled when it attempted to send more data on the IPC pipe. Classical unbounded priority inversion.

From Reeves summary, it appears as if the ASI/MET task broke the conventions of the system by making use of the pipe mechanism in place of the double-buffered shared memory used for by all other tasks. So the ASI/MET task made use of VxWorks IPC, and VxWorks IPC made use of a semaphore. This, in my humble opinion, amply illustrates the dangers of mutexes, buried in lower level code and

[4]When a task is ready to run but other tasks use up all of the processor time.

producing a critical path as a side effect. Note that double-buffered shared memory, or a lock-free queue, or even a queue with atomic `enq` and `deq` operations would have avoided this error. To "fix" the error, NASA programmers enabled VxWorks priority inheritance for the `select` code. The fix worked for two reasons: the system generally avoided semaphores and the engineers had good luck.

Black boxes that contain mutexes are dangerous.

Glossary Definition 9 *Cache ping-pong* .

In a shared memory multiprocessor test and set is not necessarily a cheap operation. I mentioned cache ping-pong above. When a processor does test and set operation on some memory location, it must instruct all other processors to invalidate their cached copies of that memory. Now suppose that processor 1 test and sets location a, processor 2 then tests and sets a and so on. Each test and set will include a global invalidate and a cache miss. Cache ping-pong can also be caused by shared locations that are on the same cache line. For example, suppose the structure

```
struct { unsigned int flags; int count; }G;
```

was used so that G.count was read all over the place and only changed by the owner - selected by using a test and set operation that ran often. In that case a processor that referenced G.count would almost certainly have a cache miss, since the test and set would cause all processors to discard the entire cache line containing G.flags.

Glossary Definition 10 *Missed wakeups.* Suppose that task A executes the code $\mathtt{if}(x == 0)wait\_on(\&q)$ and after the comparison, but before the mutex runs, an interrupt handler executes the code $x = 1; wake(q)$. The handler will find nobody to wake, but task A will resume and then wait, maybe forever because it missed the wakeup. Semaphores and mutexes are designed to minimize this problem.

Glossary Definition 11 *Atomic primitives.* Some processors have the ability to execute multi-step operations atomically by locking out interrupts and other processors. For example, in the x86 processors lock inc atomically increments a memory location, using a so-called read-modify-write cycle to prevent any other processor from modifying the location. The x86 also has a very useful bit test and set instruction that atomically tests a bit and sets the bit to 1.

To avoid missed wakeups, there are several good answers. The standard POSIX threads answer is the condition wait and that's often a good method. POSIX condition wait atomically releases a mutex and puts a thread on the condition wait queue. A wakeup of a thread on the condition wait queue atomically re-acquires the mutex and wakes the thread. This method is excellent for code of the form

```
lock mutex
while condition is false
    conditional wait
unlock mutex
```

It's also possible to use timeouts — and with care this can be the best solution. The most useful cases are where the thread that can miss the wakeups is not critical. Here we can reduce synchronization overhead by simply ignoring the missed wakeup and arranging to timeout.

```
while condition is false and time delay not expired
    conditional wait
```

Glossary Definition 12 *Threads.* The POSIX standard defines a thread as follows.

> A single flow of control within a process. Each thread has its own thread ID, scheduling priority and policy, errno value, thread-specific key/value bindings, and the required system resources to support a flow of control.

References

[1] Edsger W. Dijkstra. The structure of the "THE"-multiprogramming system. *Comm. ACM*, 11(5):341–346, 1968.

[2] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared — memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[3] David Lorge Parnas. Why software jewels are rare. *IEEE Computer*, 29(2):57–60, 1996.

[4] Glenn E. Reeves. What really happened on mars. In *RISKS Forum (19.54)*, risks@sri.com, jan 1998.

[5] Uresh Vahalia. *Unix Internals: The new frontiers.* Prentice-Hall, 1996.

[6] Victor Yodaiken. Priority inheritance is a non-solution to the wrong problem. Technical report, FSMLabs, 2002.