Solving the Priority Inversion Problem in legOS

Michael Haugaard Pedersen, Morten Klitgaard Christiansen & Thomas Glæsner
Computer Science
University of Aalborg
{willow, kc, glaesner}@cs.auc.dk
http://www.cs.auc.dk/˜willow/legOS

22nd May 2000

Abstract

legOS is currently the most powerful development tool available for the LEGO® Mindstorms™ RCX unit. Besides being a powerful development tool it also comes close to being a full scale operating system that supports prioritized threads, synchronization with semaphores and multitasking. legOS however suffers from some serious problems with respect to being a real-time operating system, and one of these problems is the problem of priority inversion. The actual existence of the priority inversion problem in the legOS operating system is illustrated through a test case and as a solution we have implemented the Priority Ceiling Protocol. Besides preventing priority inversion, the Priority Ceiling Protocol also prevents deadlock and chained blocking, and the correctness of the implementation is therefore documented through various tests cases, illustrating how priority inversion, deadlock and chained blocking are prevented.

1 Introduction

In real-time systems the correctness of the system is not only dependent on the logical result of a computation as in conventional systems, but also on the time at which results are produced. Therefore a real-time operating system should guarantee, that when a high priority process or task arrives at the task manager, it should be processed and executed as fast as possible. It should not be possible for an independent lower priority process to preempt or cause any delay in the execution of the higher priority process. With independent we mean that the two processes do not share any resource.

A real-time operating system can be characterized by the following requirements [Stallings, 1998]:

- Determinism
- Responsiveness
- Reliability
- Fail-soft operation
- User control

First of all the operating system has to behave in a deterministic way. For instance a higher priority process is always expected to finish before a lower priority process, if these do not share any resource.

If these two processes do in fact share a resource, the responsiveness requirement tells us, that it is always possible to calculate the worst case delay of the execution of the higher priority process.

The reliability issue is very important when working with real-time operating systems. A failure in a real-time operating system can have serious consequences for the users of the systems. For instance failure in a real-time system controlling traffic lights could cause car crashes and in some situations injuries and death.

Fail-soft of operation secures that the system does not just crash at the moment it experiences a serious error. The fail-soft mechanism sees to, that the system can continue running with as much capability as possible.

In real-time operating systems users have a more fine grained control of both the task manager and the priority of processes. Thus user control is the final point we expect a real-time operating system to fulfill.

Examples of current applications of real-time systems include control of laboratory experiments, process control plants, robotics, air traffic control, telecommunications, and military command and control systems, basically systems that are dependent on the time at which results are produced.

When working with real-time operating systems it is in fact possible for a higher priority process to be blocked by lower priority processes. This can occur if the higher priority process is waiting for a resource (guarded by a mutex lock or a semaphore) that a lower priority process already possesses (or locks). As long as the resource is locked it is possible for lower priority processes to indirectly preempt the higher priority process, thus postponing the execution of it.

As a real life example of the priority inversion problem we refer to the problems NASA experienced during the Mars Pathfinder mission. To keep it short the Pathfinder spacecraft experienced the priority inversion problem simply because the components of the spacecraft were using a common resource (an information bus) to communicate with each other and access to the information bus was synchronized through the use of mutex locks. For more information we refer to the web-page describing "What happened on Mars?"[1].

It is the intention of this paper to document our work in making legOS more more deterministic. By deterministic we mean that the worst case running time of a process from start to finish can be calculated prior to execution and regardless of the lower priority processes in the system. Also by preventing deadlock and chained blocking the operating system should be more responsive.

2 What is legOS?

In 1998 LEGO® released the Mindstorms™ robot development kit[2] containing the RCX, a programmable 16MHz Hitachi H8 micro-controller. The RCX firmware, which is supplied with the RCX makes it possible for developers to program the RCX in RCX code, a very simple and limited development tool. After the release of LEGO® Mindstorms™ several independent development tools for the RCX such as *NQC*, *pbFORTH* and *legOS*[3] have been made available for the public.

legOS is actually not only the most powerful development tool available but also an open-source embedded operating system for the RCX. It is a replacement firmware that completely replaces the default RCX firmware delivered with the standard LEGO® Mindstorm™ packet. *legOS* offers program developers the ability to develop programs to run on the RCX in assembly language, C, or C++. With *legOS* the developer has direct control of a wide variety of sensors and output devices such as, display, IR port, motor(s) and the memory of the RCX [Knudsen, 1999].

The *legOS* operating system supports prioritized multitasking, the ability to work with processes and the ability of processes to synchronize through semaphores. When all this is said *legOS* is far from being a real operating system especially when looking at it from a real-time perspective[4]. For instance, interrupts are only used by the operating system to increment the system timer, one time per millisecond and after about 20 milliseconds a task shift is initiated. The only time, the operating system checks whether a process that is waiting for an event (i.e. a sensor reading) can continue to run, is when a task shift is initiated. Thus, the value of the sensor could have changed many times in between the task shift. If instead interrupts had been used to directly signal waiting processes fewer sensor readings would be lost[5].

The above mentioned problem is not the only problem with *legOS* and we will throughout this paper point out several more weaknesses in *legOS*. *legOS* was conceived by Markus L. Noga in October 1998, who still remains the driving force behind the project [Noga, 1999].

3 The Priority Inversion Problem in legOS

Priority inversion is when a higher priority process is blocked by one or more lower priority processes. This can occur when a resource is shared

[1] http://www.cs.cmu.edu/afs/cs/user/raj/www/mars.html

[2] http://www.mindstorms.com

[3] http://www.noga.de/legOS

[4] Actually it has probably never been the intension that the *legOS* operating system should be a real-time operating system, but never the less we will still try to look at it with real-time glasses.

[5] A solution to this problem can be found in [Pedersen et al., 2000]

between processes of different priorities and access to the resource is synchronized by mutex locks or semaphores.

To show that the priority inversion problem truly exist in the *legOS* operating system[6] we present the following test case. The test case consists of a LEGO® car controlled by three processes P_1, P_2 and P_3, where 1, 2 and 3 denotes the priority of the processes[7]. The processes have the following behavior and relations:

Processes:

- P_3 Stop the engine.
- P_2 Blinking front light, which can be triggered by a touch-sensor.
- P_1 Drive (start engine, set speed, set direction (fwd)).

Relations:

- P_1 and P_3 share the *engine* resource.

Figure 1 describes how the test case evolves over time.

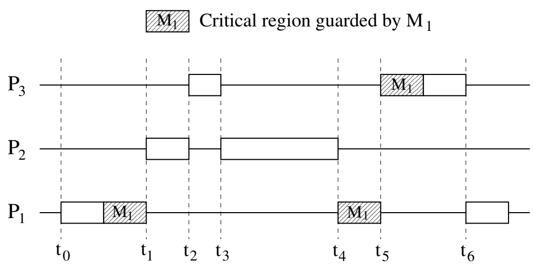

Figure 1: P_3 experiences priority inversion

As the figure shows P_1 is started at time t_0. At time t_1 P_2 is started and preempts P_1. Next, P_3 is started at time t_2, but it is suspended at time t_3, where it needs to lock M_1. Then P_2 continues and blocks P_1 until it is finished at time t_4. Here P_1 can leave its critical region and release its lock on M_1. Finally, at time t_5 P_3 can lock M_1 and enter its critical region.

It is important to notice that between t_3 and t_5 P_3 experiences priority inversion, and had it been the case that other processes with priority between "1" and "3" had been initialized in this period, for example if several P_2 processes had been initialized, then P_3 would have been blocked until these too were finished. Actually the worst case scenario is that this could result in P_3 being blocked forever.

In practice the car starts by driving forward and thereby locking the engine resource. Shortly after the drive process will be preempted by the front light process. When the front light process has finished running, the drive process will continue unless the front light process has been triggered again while running, then it will continue blinking. Only when both the drive and the front light processes have run to completion, the stop process is able to start and thereby stop the engine. Clearly it is not acceptable that the stop process is postponed because the car has to blink.

As a solution to this problem we will implement the Priority Ceiling Protocol in *legOS*, which reduces the worst case blocking time of a process (by lower priority processes) to at most the duration of the execution of a single critical region of one lower priority process [Sha et al., 1990].

4 The Priority Ceiling Protocol

The goal of the Priority Ceiling Protocol is to solve the priority inversion problem as well as preventing the formation of deadlock and chained blocking.

The idea is that all mutexes are assigned a "ceiling priority", which is equal to the priority of the highest priority process that can lock the mutex. When a process P_j, as the only one, locks a mutex it runs with its own priority, and can be preempted as normal. In order for a higher priority process P_i to lock a mutex, P_i has to have a priority higher than the ceiling priority of any of the already locked mutexes (if no other mutexes are locked, then P_i is allowed to lock the mutex regardless of its priority). When P_i has a higher priority, it means that P_i is not going to use the same resource as the one P_j is currently using, so P_i can preempt P_j. If P_i does not have a higher priority than the ceiling priority, then P_j in-

[6]In the rest of the paper we will refer to the *legOS* operating system as just *legOS*.

[7]In *legOS* the priority of a process can be any number from 0 to 20, where 20 is the highest priority and 0 is the priority of the idle process. This priority numbering will be used throughout the paper.

herits the priority of P_i, which becomes blocked and P_j continues running. This approach ensures that when a process P_i preempts a process P_j that is running in its critical region, and then enters its own critical region, then process P_i is guaranteed to have a higher priority than all of the preempted processes [Sha et al., 1990], [Jensen, 1999].

4.1 Preventing Priority Inversion

We will now, using the test case presented in section 3, illustrate how priority inversion is prevented with the Priority Ceiling Protocol. Recall that:

- P_1 and P_3 share the *engine* resource guarded by the mutex M_1, giving M_1 the ceiling priority 3.
- P_2 uses no shared resources.

Figure 2 depicts how the processes in the test case evolves over time.

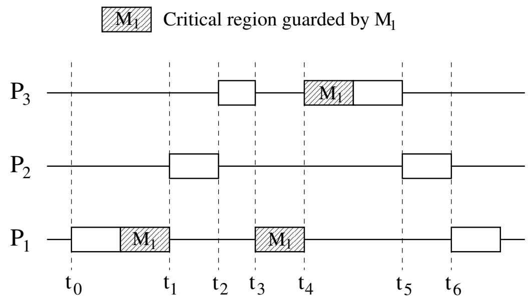

Figure 2: Preventing priority inversion

If we compare the two situations by looking at figure 1 and 2, the difference is that at time t_3 when P_3 tries to lock the mutex M_1 guarding the *engine* resource, P_3 becomes suspended and P_1 inherits P_3's priority ("3"). P_1 then runs to the end of its critical region, and when it releases the *engine* resource at time t_4 it obtains its original priority "1". P_3 can now lock M_1 and enter its critical region. At time t_5 when P_3 finishes, that is stopping the engine, P_2 has the highest priority and completes its computation. Finally, at t_6 P_1 can run to the end.

4.2 Deadlock

Deadlock can occur when two processes P_i and P_j share two resources A and B. If for instance P_i locks the mutex guarding A and P_j locks the one guarding B. Then deadlock occurs if P_j needs A while using B and P_i needs B while using A, because the processes then will be blocked waiting for each other to release its resource. As mentioned above, deadlock is prevented by the Priority Ceiling Protocol, but how does it prevent it?

4.3 Preventing Deadlock

When a process P_j tries to lock a mutex M_1, and a process P_i is running in its critical region having locked the mutex M_2, then a check is made to see if P_j has a higher priority than the ceiling priority of all locked mutexes. If this is the case then P_j locks M_1 and preempts P_i. If not then P_j is not allowed to lock the mutex M_1, and P_i inherits the priority of P_j and P_j is suspended. By blocking P_j outside its critical region a possible deadlock has been avoided.

Figure 3 illustrates the test case, which shows the prevention of a possible deadlock. The test case consists of the following processes and mutexes.

Processes:

- P_3 Blinking light
- P_2 Drive forward at slow speed, then speeding up and down and stop.
- P_1 Drive fast backwards, then slowing down, speeding up and stop.

Mutexes:

- M_3 is the mutex guarding the *blink* resource.
- M_2 is guarding the *slow speed* resource.
- M_1 is guarding the *fast speed* resource.

Relations:

- Processes P_1 and P_2 share the two resources guarded by mutex M_1 and M_2.
- Process P_3 needs a resource guarded by mutex M_3.

At time t_2 process P_2 tries to lock M_2, but since the ceiling priority of the locked mutex M_1 is not lower

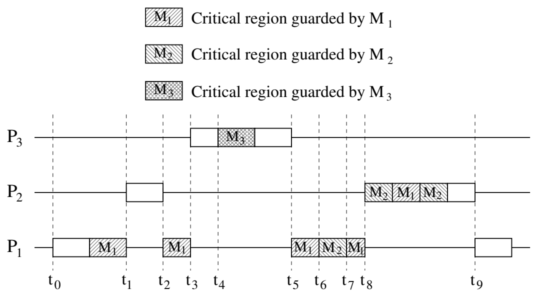

Figure 3: Preventing deadlock

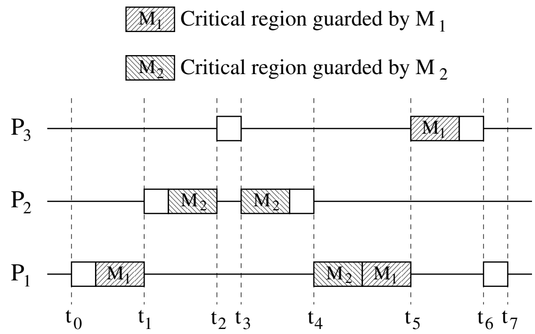

Figure 4: P_3 is blocked for the duration of the critical region computation of P_1 and P_2

than the priority of P_2 then P_2 is blocked outside its critical region, and P_1 inherits the priority of P_2 and continues running in its critical region. At time t_3 P_3 preempts P_1 and enters its critical region at time t_4, because it has a higher priority than the ceiling priority of M_1. At time t_5, P_3 finishes and P_1 wants to lock M_2. Since P_1 is running in its critical region and M_2 is not already locked, P_1 is granted the lock on M_2. Next, everything runs to completion and the possible deadlock, which could have occurred approximately at time t_3, if P_2 had been allowed to lock M_2 at time t_2, has been avoided.

In practice the car starts by driving fast backwards and is then preempted by the blinking process and it will remain at the current speed for as long as the blinking process runs plus the rest of the time the fast backwards process was set to run. Then the car decreases and increases the speed one more time, after which it changes direction driving slow forward, speeds up and down and stops. Without the Priority Ceiling Protocol the car would have deadlocked in the situation where it drives fast forward and never stop (at some point it would probably crash!).

4.4 Chained Blocking

Chained blocking is when a higher priority process is blocked for the duration of the critical region of several lower priority processes. An example of this problem is shown on figure 4.

As the figure shows, P_3 is blocked for the duration of the critical region of both P_1 and P_2, which is between t_3 and t_5.

4.5 Preventing Chained Blocking

The prevention of chained blocking is solved by the ceiling priority and by priority inheritance. So by using both of these properties a higher priority process will never be blocked for more than the duration of one critical region computation of a lower priority process. An example of this is shown in figure 5.

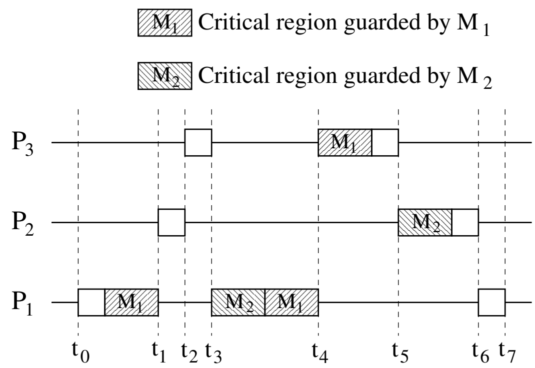

Figure 5: P_3 is only blocked for the duration of the critical region computation of P_1

As it shows P_2 can not lock M_2 because it has a lower priority than the ceiling priority of M_1 and therefore P_3 is blocked for the duration of the critical region computation of only one lower priority process, in this case P_1.

5 The legOS Operating System

In this section we will give an outline of how the processes are stored in *legOS* and how *legOS* go about getting the the right process to run, sleep or wait for an event. It is important to understand how *legOS* handles the process management in order to understand the design and implementation of the Priority Ceiling Protocol.

5.1 Organization of the Processes

Figure 6 shows how the processes are organized in *legOS* and some of the attributes in both the process and priority level structure. The boxes represent priority levels, which are grouped together to form an ordered prioritized chain, where 20 is the highest priority and zero is the lowest priority. Each priority level contains a pointer *cpid* to a chain of one or more processes (a process is visualized by a circle). The process pointed to by *cpid* is the process in the process chain that is currently executing or has been executed most recently. A priority level will always contain at least one task otherwise it will be non-existing. To access the priority chain, a pointer *priority_head*, which points to the highest priority level, is used. Each priority level contains a pointer *prev* (in the figure visualized as *p*) to the previous priority level and a pointer *next* (in the figure visualized as *n*) to the next priority level.

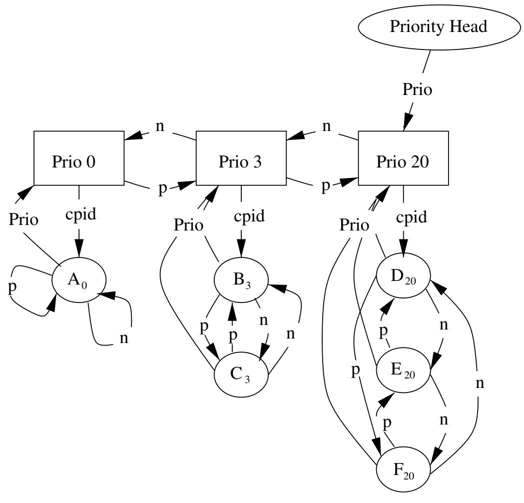

Figure 6: The organization of processes in *legOS*

The process chain pointed to at each priority level is a chained list of processes. A process can either be in a state of sleeping, running or waiting for an event. The processes are connected to each other by the pointers *next* and *previous* (visualized in figure 6 by *n* and *p*), also each process in the chain contains a pointer *priority* (visualized in figure 6 as *prio*). This pointer is used whenever the priority level of a process is needed.

Initially, before any user program is started the operating system will only contain two processes, one with priority zero and one with priority 20. The zero priority process is used to make the RCX unit go into sleep mode when not used, and the 20 priority process is used to receive data from the IR-tower. Also, this is the place where *legOS* checks whether the start program button or the power off button has been pushed. Since the priority level of processes are created while *legOS* is running, *legOS* can itself run out of memory. Thus, *legOS* does not satisfy the the reliability requirement stated in section 1.

5.2 Task Scheduling

The task manager uses the above mentioned data structure to find out which process is the next to run (in *legOS* a process that is ready to run is marked as sleeping). Basically, what the task manager does is to traverse the process chains in a round-robin fashion starting at the process chain located at the highest priority level searching for a process ready to run. This procedure is repeated every time a task shift is initiated.

5.3 Event Handling

Finally, we will just mention how *legOS* deals with events. If the task manager, while looking for a process to run, finds a process that is waiting for an event, the task manager will test if this event have occurred by running a special wake up function. This function is defined inside the user program and takes one argument. In order for the task manager to be able to get hold of the wake up function and its argument, both the argument and the address of the function is stored as attributes in the process data structure. If the event has occurred the task manager will execute the process right away, otherwise it will look for another process leaving the state of the process unchanged.

6 Design & Implementation

In this section we will explain how we have designed and implemented the Priority Ceiling Protocol in *legOS*. The design and implementation have been divided into two parts - one part dealing with priority inheritance of processes and a second part dealing with deadlocks and chained blocking. This section also contains a description of a memory fault we have encountered during our work with *legOS*.

Note that since *legOS* does not directly support mutexes, we have implemented mutex facilities according to the POSIX standard [Butenhof, 1997]. The implementation of the mutex facilities and the Priority Ceiling Protocol can be found in the files *pthread.h* and *pthread.c*. The following procedures and functions have been implemented:

int pthread_mutex_init(pthread_mutex_t *,
const pthread_mutexattr_t *),
int pthread_mutex_lock(pthread_mutex_t *),
int pthread_mutex_unlock(pthread_mutex_t *),
int pthread_mutex_trylock(pthread_mutex_t *),
wakeup_t pthread_mutex_event_wait(wakeup_t),
int pthread_mutex_setprioceiling(pthread_mutex_t *,
int, int *),
int pthread_mutex_destroy(pthread_mutex_t *),
void boost_pprio(pthread_mutex_t *, pdata_t *),
void deboost_pprio(pthread_mutex_t *, pdata_t *),
void add_to_mutex_list(pthread_mutex_t *) and
void remove_from_mutex_list(pthread_mutex_t *).

The interface of all the functions starting with *pthread_mutex* (except for the function *pthread_mutex_event_wait(wakeup_t)*) are implemented according to the POSIX standard. The rest are procedures that are called within the POSIX standard functions. Finally, it should be noted that the mutexes, as currently implemented, can only be used when a ceiling priority is given.

6.1 Priority Inheritance

The first thing we have to take into account is how to boost the priority of a process if this process has locked a mutex which is needed by a higher priority process. Consider figure 7(a); let process A_3 run inside its critical region, which is guarded by a mutex M. Then imagine that process D_7 needs M in order to finish execution. What we want to do is to boost the priority of A_3 to the same priority as D_7 until A_3 releases M.

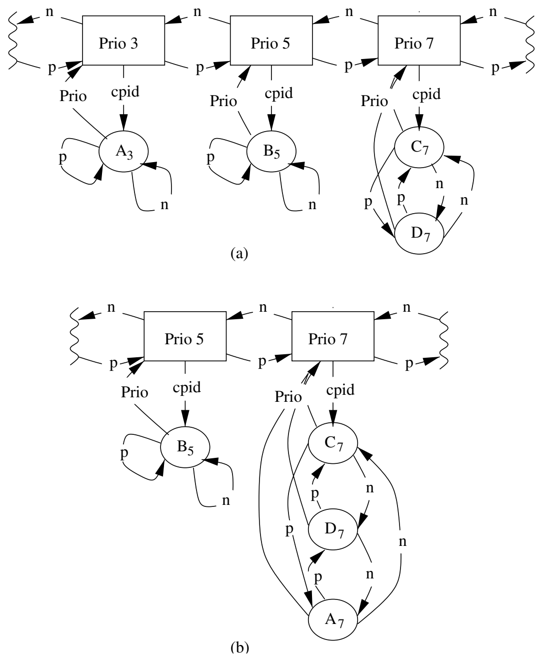

Figure 7: (a) Before boosting the priority of process A_3 to priority level ”7”. (b) After the process A_3 has been boosted to run at priority ”7”

The part of boosting the priority of a process in *legOS* boils down to moving the process from its original priority level to the new priority level. If the process is the only process at its original priority level we just remove the priority level and then create it again when the process releases the mutex. Figure 7(b) shows the data structure of the task manager after A_3 has been boosted to priority level ”7”. When A_7 (the boosted A_3) releases the mutex the process will be moved back to its original priority level and in case this level does not exist we have to create it again. In order to be able to recreate the original priority level we store the original priority of the process in an extra attribute at the time of process creation. Also, when a process locks a mutex we need to store a pointer to the process locking the mutex (the owner) in the mutex structure. By doing this we achieve that when a higher priority process needs to lock an already locked mutex the owner can easily be found and (maybe) boosted. This is also here the main difference between mutexes and semaphores shows. It would not be possible to implement the Priority Ceiling Protocol on a semaphore object since a semaphore object does not contain any information about the owner. To even talk about an owner of a semaphore object does not make sense, since semaphores do not necessarily need to be released (or decreased) by the same process.

The possibility of boosting the priority of a process takes place at the moment a process tries to lock an already locked mutex. Thus, we have implemented the boosting of a process in *pthread_mutex_lock(pthread_mutex_t *)*. This method takes a pointer to a mutex object or structure as argument and tests if the mutex is not already locked by using *pthread_mutex_trylock(pthread_mutex_t *)*. If the mutex is already locked the priority of the owner will be boosted by executing the method *boost_pprio(pthread_mutex_t *, pdata_t *)*. After this we can only wait for the owner to release the mutex, this is done by the wait event function *wait_event(pthread_mutex_event_wait, pdata_t *)*, which takes as argument a wake up function and its argument (the reason why the wakeup function takes as argument a process and not a mutex, will be explained later). If the mutex is not locked, a process is granted a lock on it right away. After the mutex has been locked we just need to set the new owner of the mutex. Figure 8 shows a flow diagram for the locking function.

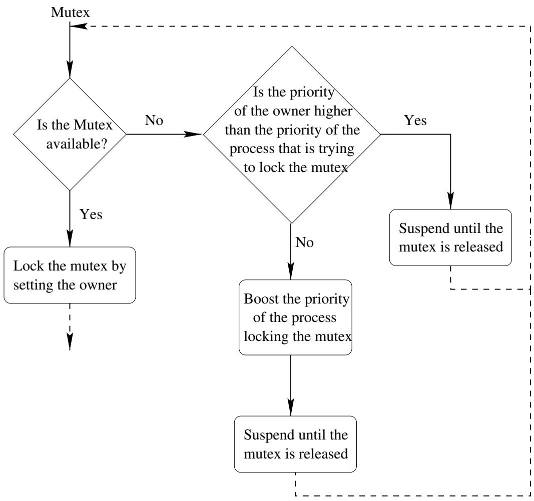

Figure 8: Flow diagram for the locking function

A mutex is unlocked using the function *pthread_mutex_unlock(pthread_mutex_t *)*. Thus, if the process that is unlocking the mutex has been boosted, we need to restore it to its original priority. We can check if a process has been boosted by comparing the original priority with the current priority level that the process is in. If these to levels are the same we know that the process has not been boosted. To restore a process to its original priority level we have used the function *deboost_pprio(pthread_mutex_t *, pdata_t *)*. This function have the following behavior. First, it needs to check whether the original priority level still exists. If not, a new level is created by using the original priority attribute and added to the priority chain. Then the process needs to be moved back to its original level and all pointers should be restored to their respective targets. The flow diagram in figure 9 illustrates the general idea of the design and the implementation of the unlocking function.

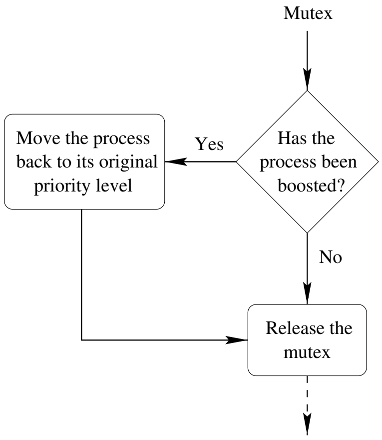

Figure 9: Flow diagram for the unlocking function

6.2 Preventing Deadlock and Chained Blocking

As you might recall to prevent deadlock and chained blocking we need to compare the priority of a process that wants to lock a mutex to the highest ceiling priority of all of the already locked mutexes. The process is only granted a lock if none of the locked mutexes have a higher ceiling priority than the priority of the process. Thus, the locking function has now been extended to first of all check for this situation before we allow a lock on the mutex. To be able to make this comparison we need to have an ordered list of all locked mutexes and an entry to the list pointing to the mutex with the highest ceiling priority. If the process is not granted a lock it will be suspended until it has a higher priority than the ceiling priority of any of the locked mutexes. Thus we need to extend the wake up function, *pthread_mutex_event_wait(pdata_t *)* to perform this check. The task manager can now use it to decide whether a process should wake up or continue to be suspended.

We need to consider just one more thing before we are finished with *pthread_mutex_event_wait(pdata_t *)*. Consider the scenario depicted in figure 10; at time t_1 a process running at priority level one locks the mutex M_1 and at time t_2 the process also acquires a lock on mutex M_2. At time t_3 the priority of the process is boosted to a higher priority level and at time t_4 the process releases M_1 again.

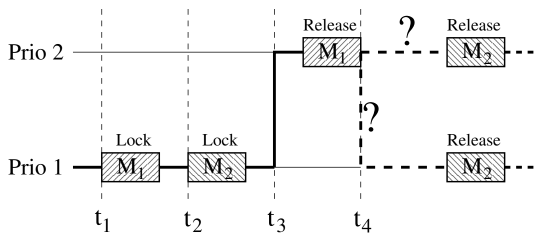

Figure 10: Boosting problem

The question is now, at what priority level should the process continue its execution in? Of course we would not allow the process to continue running with the boosted priority, if the reason for boosting the process was that the higher priority process just needed to lock M_1. If we allow this we would just face a new situation of priority inversion - the mere problem we where trying to solve! The way we have solved this problem, is to deboost every time a mutex is unlocked, unless the unlocking process has not been boosted at all. When we deboost every time a process unlocks a mutex, we need to boost it again if it still locks a mutex needed by a higher priority process. Thus we need to extend the wake up function *pthread_mutex_event_wait(pdata_t *)* again, so that this function can also boost the priority of a process. The flow diagram shown on figure 11 illustrates the extended version of *pthread_mutex_event_wait(pdata_t *)*.

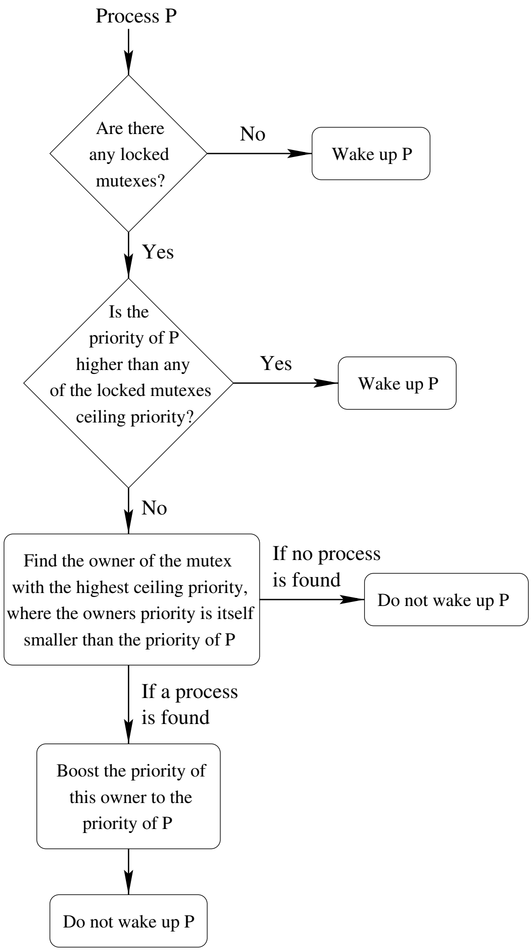

Figure 11: The extended version of *pthread_mutex_event_wait(pdata_t *)*

6.3 Memory Fault

Throughout the implementation and testing of the Priority Ceiling Protocol we encountered some internal RCX memory faults. These problems appeared when a program was allocating memory to a new priority level using the function *malloc(size_t)*.

As discussed in the *legOS* group at LUGnet[8] it has been found that memory addresses from 0xfb80 to 0xfd7f in the RCX is not writable, or more exactly, data ”stored” in this address space is always read as 0xff. Although, this bug has been fixed in later versions of legOS, it is still not known whether other ”bad” regions in the memory exists. Our way of fixing this problem was to download our programs to the second program region (as program number 1) of the RCX, and by doing this avoiding the ”bad” memory regions present in the first program region (program number 0). This apparently fixed our memory problem, but a complete check for ”bad” regions in the RCX is recommended to completely avoid any further memory problems.

7 Testing the Implementation

In order to see how expensive the implementation is with respect to time, we have compared the execution time of two test cases by first running them using mutexes and secondly running them using semaphores, that is without the Priority Ceiling Protocol. As the first test case we have chosen the test case described in section 3 and illustrated in figure 1 and 2 and as the second test case we have used a slightly modified version where the locking order ensures that no processes are boosted.

The tests has been performed by running the respective test cases five consecutive times registering the start and finish time of the entire run-through. It should be noted that we have not been able to use the system time to measure the difference since this is based on interrupts, and in order to avoid task shifts, when the mutex facilities are used, interrupts are disabled while executing the lock and unlock functions. As a result time has been measured by hand (with a stop watch). The results after three test runs of the original test case are shown in table 1 and the results after three test runs of the modified test case are shown in table 2.

As table 1 shows there is a difference between using mutexes and semaphores of approximately 340 milliseconds, which yields an average of 68 milliseconds per run-through. Table 2 shows that the difference between using mutexes and semaphores when omitting boosting is only about 76 milliseconds, which yields an average of 15 milliseconds per run-through. From these test results we can see that the boosting part takes approximately 53 milliseconds.

[8] LEGO® User Group Network, http://www.lugnet.com/robotics/rcx/legos/

Execution time using mutexes	
Difference [msec]	*Average [msec]*
114660	
114650	114567
114420	

Execution time using semaphores	
Difference [msec]	*Average [msec]*
114200	
114280	114227
114220	

Table 1: Time used to execute test case with boosting

Execution time using mutexes	
Difference [msec]	*Average [msec]*
114240	
114110	114123
114020	

Execution time using semaphores	
Difference [msec]	*Average [msec]*
114020	
114080	114047
114040	

Table 2: Time used to execute test case without boosting

8 Conclusion

Even though the implementation of the Priority Ceiling Protocol does in fact make the *legOS* operating system more deterministic and responsive there is still room for improvements. As you will see in section 9, priority inversion still occur in the form of ceiling blocking.

Also, *legOS* still does not support interrupts for anything else than the system clock, which means that external devices can not signal waiting processes directly, and the operating system then has to poll special registers in order to register signals (i.e. interrupts) from external devices. Because of this polling scheme the *legOS* operating system is still not as responsive as real-time developers could have whished for (i.e. interrups can get lost due to the fact that new interrupts overwrites older interrupts).

On the basis of our work with *legOS* we must conclude that *legOS* is far from being a real-time operating system. Even though the implementation of the Priority Ceiling Protocol has made the operating system more deterministic and responsive it still needs a lot of work before it satisfies all the requirements for a real-time operating system as stated in the introduction.

With respect to the implementation, the test results shows that there is only little overhead when using mutexes as opposed to using semaphores. Most of the overhead is probably due to the priority boosting of processes.

9 Future Work

The Priority Ceiling Protocol is an effective real-time protocol, but there is still room for improvements. Even though the protocol prevents priority inversion and deadlock, it introduces a new kind of priority inversion, called ceiling blocking, where a higher priority process is blocked by a lower priority process, even though the low priority process is not going to use the same resource as the high priority process.

Also, as mentioned in section 6, the mutex facilities have only been implemented to support the Priority Ceiling Protocol. As the system is working now users can not use mutexes without also using the Priority Ceiling Protocol. Thus, in order to make the system more flexible, the users have to be able to use the mutexes without the ceiling priority and thereby not using the Priority Ceiling Protocol. Also, some people would say that deadlock is a feature rather than a problem, and in order to please those people the Priority Inheritance Protocol should also be implemented in *legOS* as a stand alone option.

References

[Butenhof, 1997] Butenhof, D. R. (1997). *Programming with POSIX Threads*. Addison-Wesley.

[Jensen, 1999] Jensen, P. K. (1999). *Reliable Real-Time Applications, and how to use tests to model and understand*. PhD thesis, Aalborg University. Unpublished Ph.D. Thesis Submitted to Institute for Computer Science.

[Knudsen, 1999] Knudsen, J. B. (1999). *The Unofficial Guide to Lego Mindstorms Robots*. Beijing-Farnham : O'Reilly.

[Noga, 1999] Noga, M. L. (1999). Designing the legos multitasking operating system. *Dr. Doob's Journal*.

[Pedersen et al., 2000] Pedersen, S. T., Christensen, L., and Rasmussen, E. B. (2000). Prioritized interrupts in legos. Unpublished Project Paper Submitted to Institute for Computer Science at Aalborg University.

[Sha et al., 1990] Sha, L., Rajkumar, R., and Lehoczky, J. P. (1990). Priority inheritance protocols: An approach to real-time stynchronization. *IEEE Transactions on Computers*, 39(9):1175–1185.

[Stallings, 1998] Stallings, W. (1998). *Operating Systems: internals and design principles*. Prentice-Hall, Inc., 3rd edition.