An Introduction to Real Time Operating Systems and Scheduling

Ambuj Tewari
Utkarsh Srivastava

April 25, 2001

Submitted in partial fulfillment of the requirements of the course CS 397
Supervisor: Dr. Deepak Gupta

Indian Institute of Technology, Kanpur

Contents

1 Introduction

An realtime operating system may be defined as one that has the ability to provide a required level of service along with a bounded response time. They can be categorized into hard and soft real time systems. Soft real time systems are one in which timing requirements are statistically defined. Video conferencing is a classic example: it is desirable that frames be not missed but it is acceptable if a frame or two is occasionally skipped. However in a hard real time system deadlines must be guaranteed. For example a rocket fuel injection procedure *must* be completed in time.

In order to achieve a bounded response time, careful planning has to go into the design of the scheduling policy. Real time scheduling is consequently the most widely studied aspect of real time operating systems. Here we shall touch upon a few results on real time scheduling for uniprocessors as well as multiprocessors for independent tasks. The uniprocessor scheduling problem has been thoroughly studied and a number of results derived, but for multiprocessors most of the interesting problems have been proved to be NP complete.

However it is not uncommon to have interacting real time tasks, for example a periodic data acquisition task which sets off a robot motion task on a particular data pattern. In such cases resource sharing has to be done between the real time tasks in a mutually exclusive manner. Here we shall also look at a few approaches to real time synchronization including the simple priority inheritance protocol and the priority ceiling protocol. These approaches mainly differ in the way in which they curb the problem of priority inversion which arises under any lock-based protocol. We also look upon a lock free protocol whichdoes not suffer from priority inversion but at the cost of possibly extra computation.

Earlier real time systems tended to be extremely simple machines dedicated to executing real time tasks. But over the years real time systems have become more sophisticated, requiring services such as Graphical User Interface and networking (for example to display the results of a data acquisition task using bar charts). Such services are conventionally provided by time sharing systems and hence several attempts have been made to club the two together. Here we shall look at the approach taken by RT Linux (A Hard Real Time variant of Linux). The intention is to make use of the sophisticated services and highly optimized average case behavior of a standard time-shared operating system (Linux) while still permitting real-time functions to operate in a predictable and low-latency environment.In RT Linux a simple real-time executive runs a non-real-time kernel as its lowest priority task, using a virtual machine layer to make the standard kernel fully pre-emptable. Lastly we look at the implementation of the basic priority inheritance protocol that we carried out in the RT Linux mutex implementation.

2 Real Time Scheduling on Uniprocessor Systems

2.1 Scheduling paradigms

A variety of strategies have been adopted by researchers for scheduling real time tasks. Most classical scheduling theory deals with static scheduling. In static scheduling, we have complete knowledge of the task set and its constraints, such as deadlines, computation times and release times. We can either resort to building a complete schedule beforehand or we can assign static priorities to all tasks and always choose the highest priority task to be run on the processor. The former technique is referred to as *Static Table Driven Approach.* The latter may be termed as *Static Priority Based Approach.* In contrast to the above two approaches, we may have a *dynamic* scheduling algorithm. Although there is no agreement on the use of the term *dynamic* as applied to scheduling algorithms, a dynamic approach is one where the decision of choosing the task to run, is made at run time. This may either involve assigning dynamic priorities to tasks or doing a schedulability analysis when a new task arrives. Also, we call a scheduling algorithm *online*, if it has the ability to schedule tasks which are added dynamically. We will not discuss this issue of terminology any further. For us, unless specified otherwise, *dynamic* would mean something which does not involve preparing a schedule offline or assigning static priorities to tasks. Throughout our discussion it will be assumed, unless specified otherwise, that preemptions of tasks take place at integral time boundaries. This is often referred to as the *Integer Boundary Constraint(IBC).*

2.2 The Rate Monotonic Algorithm

The rate monotonic(RM) algorithm is a famous one, and is applied to schedule a set of periodic tasks, where task i has a time period of T_i and a computation requirement of C_i in every time period. This algorithm was first discussed by Liu and Layland [LIU73] in their classic paper on real time scheduling. The RM algorithm assigns priorities to tasks according to their rates, i.e. inverse of period. So the task with the shortest period is assigned the highest (static) priority. It may be noted that we are here assuming the deadlines of tasks to be equal to their respective periods. Now, it is easy to see that a necessary condition for scheduling the tasks on a single processor is that the computation requirements of the entire task set should not exceed that amount of processor time available. So we have

$$\sum(C_i/T_i) \quad \leq \quad 1 \qquad (1.2.1)$$

The left hand side of this inequality is referred to as the *utilization factor(U)* of the task set. This is, however, not a sufficient condition for a task set to be schedulable under static priority assignment. Liu and Layland proved that if the task set is schedulable under *any* static priority assignment then they must be so under RM. Thus, RM is optimal among all static priority algorithms. As a consequence of this strong result, we only need to establish a sufficient condition for schedulability under RM. In the same paper it was established that RM always succeeds in scheduling a task set, if the following condition holds

$$\sum(C_i/T_i) \quad \leq \quad n(2^{1/n} - 1) \qquad (1.2.2)$$

where n is the number of tasks in the task set. The quantity on the right side of the inequality above decreases monotonically from .83 when $n = 2$ to $\ln 2 = .69$ as $n \to \infty$. The result shows that any periodic task set of any size will be able to meet all deadlines if the RM algorithm is used and the total utilization is less than 70%. It may also be noted that RM is not optimal if deadlines are not equal to periods. In that case, the deadline monotonic(DM) algorithm is optimal. It assigns highest priority to the task having shortest deadline period.

2.3 Exact Analysis using Response Time

It has been observed that RM succeeds in scheduling task sets with over 90% utilization [LHC89]. It has also been demonstrated that the average case behaviour of the RM algorithm is much better than the worst case. Here, we briefly describe a simple procedure to determine feasibility of a given task set under some static priority assignment. This can be used to find out whether RM will succeed in scheduling the task set, when the sufficient condition (1.2.2) does not hold. The idea is to calculate the response time(R_i) of each task. The response time is the time which the given task will take to execute in the worst case, in the presence of higher priority tasks. So, for feasibility we have the condition

$$R_i \quad \leq \quad T_i \qquad (1.3.1)$$

Let us now see, how can we calculate R_i. Observe that R_i has to satisfy the recurrence

$$R_i \quad = \quad C_i \quad + \quad \sum_{j \epsilon hp(i)} \lceil R_i/T_j \rceil * C_j \qquad (1.3.2)$$

where $hp(i)$ is the set of tasks having higher priority than task i.This is because $\lceil R_i/T_j \rceil$ is the maximum number of invocations of task j during the response time and each invocation requires C_j units of processor time. The response time should be equal to the sum of computation time of task i and the time spent in serving the higher priority tasks. We can solve the recurrence (1.3.2) iteratively as follows

$$R_i^0 = C_i \qquad (1.3.3)$$

$$R_i^{n+1} = C_i \quad + \quad \sum_{j \epsilon hp(i)} \lceil R_i^n/T_j \rceil * C_j \qquad (1.3.4)$$

In (1.3.3), we set our initial approximation of R_i to be C_i. Subsequent approximations are calculated using (1.3.4). In this way we can compute the response time to the desired degree of accuracy.

2.4 The Earliest Deadline First Algorithm

We saw in a previous section (1.2) that no static priority algorithm can guarantee to schedule task sets which have full (100%) processor utilization. It is a natural question then, to ask whether condition (1.2.1) is sufficient for schedulability or not. In the above mentioned paper of Liu and Layland, they not only answered this question in the affirmative but also proved that a simple algorithm, called the *Earliest Deadline First(EDF) algorithm* ,can schedule any (periodic) task set satisfying (1.2.1). The EDF scheduler simply chooses that task which has the nearest deadline. Later, it was shown by Dertouzos [DRT74] that the scheduler is optimal for arbitrary task sets, not just periodic ones. The optimality proof depends on the fact that it is always possible to transform a feasible schedule into one which follows the EDF algorithm. This so because if at any time the processor executes some task other than the one which has closest deadline, then it is possible to interchange the order of execution of these two tasks. Since, the sacrificed task has a more distant deadline, making up for its one unit of lost processor time before closest deadline certainly does not violate its own deadline. It is worthwhile to mention here that the *Least Laxity First(LLF) algorithm* is also optimal in the same way as EDF is. The LLF scheduler schedules the task with the least laxity where laxity is the difference between time before a deadline arrives and the remaining computation time. A task with zero laxity *must* be scheduled or it will miss its deadline.

3 Real Time Scheduling on Multiprocessor Systems

3.1 Problems in the Multiprocessor case

We have seen that the uniprocessor case is well understood and we have a necessary and sufficient condition for schedulability of periodic tasks on a uniprocessor. A natural extension of the scheduling problem is to introduce more processors. Clearly, the modified version of condition (1.2.1) is

$$\sum(C_i/T_i) \quad \leq \quad m \qquad (2.1.1)$$

where m is the number of processors. (2.1.1) is clearly necessary for feasibility but we should pause before we claim that it is sufficient too. Any multiprocessor schedule must satisfy two constraints. First, a processor must not be allocated to more than one task at a time. Second, a task must not be scheduled on more than one processor at the same time. These constraints introduce subtle differences between the uniprocessor case and the multiprocessor case and it so turns out that the optimal scheduling algorithms (like EDF and LLF) of the uniprocessor case cease to be so when there are more than one processors.

3.2 The Insufficient Knowledge Problem

The proofs of optimality of EDF and LLF break down because of the additional second constraints mentioned in the previous section (2.1). Since EDF is so simple both conceptually and implementation wise, it is natural to ask for an equally simple and efficient algorithm for scheduling tasks on m processors. We also desire to have an optimal algorithm, so that we can find a schedule if one exists. Dertouzos and Mok [DRT89] however answered the question in the negative for a wide class of schedulers. In particular, they proved that if we do not have *a priori* knowledge of any of the following parameters: 1) deadlines, 2) computation time, or 3) the start times, then for *any* algorithm one might propose, one can always find a set of tasks which cannot be scheduled by the proposed algorithm but which can be scheduled by another algorithm. They established their claim by "adversary" arguments. The interested reader is requested to refer to their paper for detailed proofs. The RM algorithm does not care about computation times at all, so we cannot expect it to be optimal, and it is indeed the case that RM is not an optimal priority assignment algorithm for the multiprocessor case.

Dertouzos and Mok mention two conditions under which task sets may be scheduled optimally at run time. One is the simple case when each task has unit computation time. In this case the optimality proof of EDF does not break down and so the task set is schedulable optimally by an EDF scheduler. Second is the case when the set of tasks can be scheduled if they all start at the same time. If this property is satisfied by a task set then it can be scheduled even when the tasks do not start at the same time. The optimal scheduler in this case is the LLF scheduler.

3.3 Pfair schedules

We have, so far, not addressed the question whether $U \leq m$ (U is the utilization) is a sufficient condition for schedulability of periodic real time tasks on m multiprocessors, in the presence of IBC[1]. The question remained open for many years though a partial answer was provided in [DRT89]. They proved that if $T = GCD(T_1, T_2, \dots T_n)$, and $T * C_i/T_i$ is an integer for all i , then it is possible to schedule these n tasks on m processors provided $U \leq m$. The idea was to allocate $T * C_i/T_i$ time units to task i in each slot of T time units. A task has to wait for T_i/T slots before it is allotted C_i time units, and the total duration of these slots is exactly equal to the period of the task, so no deadlines are missed. This condition was improved slightly by Khemka and Shyamsundar [KHM92] but they came nowhere close to the complete answer. Baruah et al. [BRH96] proved that $U \leq m$ is indeed sufficient for scheduling n task on m identical multiprocessors.

[1]recall that IBC is the integer boundary constraint

In fact, they construct a schedule that is not only feasible but also *pfair*. The notion of Pfairness is defined and discussed in the next section.

3.3.1 The notion of Pfairness

Define the weight W_i of task i to be C_i/T_i . Instead of viewing task i as a task with period T_i and computation time C_i, pfair schedulers view it as a task with period $T_i/C_i = 1/W_i$ and computation time unity. Now, because of IBC, we cannot demand that task i be allotted $W_i * t$ time units till time t, for all $t \geq 0$. So, we require in our schedule that (for all t), task i has been alloted either $\lceil W_i * t \rceil$ or $\lfloor W_i * t \rfloor$ time units at time t. Thus pfairness is a stricter requirement than feasibility. This is because any pfair schedule is a feasible schedule but the converse is not true. Now, it is remarkable that the condition (2.1.1) actually is sufficient for the existence of a *pfair* schedule, not just a feasible schedule.

3.3.2 A Pfair scheduling algorithm

Before, we present the algorithm of Baruah et. al., some notations have to be defined. S denotes a schedule.

$S(x,t) = 0$ if task x is scheduled at time t otherwise it is 1.
$lag(S,x,t) = W_x * t - \sum_{i=0}^{t-1} S(x,i)$
S is pfair if and only if the following inequality holds for all tasks at all times.
$-1 < lag(S,x,t) < +1$
A task is *ahead* if lag is less than zero, is *behind* if lag is more than zero, and is *punctual* otherwise.

The *characteristic string* of task x is defined as follows
$\alpha_t(x) = sign(W_x * (t+1) \ - \ \lfloor W_x * t \rfloor \ - \ 1)$
Here $\alpha_t(x)$ represents the tth character of $\alpha(x)$. Hence $\alpha(x)$ is a string over the alphabet $\{-, 0, +\}$ and $- < 0 < +$ lexicographically.

The *characteristic substring* of task x at time t is defined to be
$\alpha(x,t) = \alpha_{t+1}(x)\alpha_{t+2}(x)\ldots\alpha_u(x)$
where $u = min\, i : \ i > t \wedge \alpha_i(x) = 0$

A task x is *tnegru* at time t if x is ahead and $\alpha_t(x) \neq +$
A task x is *urgent* at time t if x is behind and $\alpha_t(x) \neq -$
Otherwise it is contending.

It can be proved that if a task is *tnegru* then a pfair scheduler will not schedule it and if it is *urgent* then it will certainly schedule it at time t. So, the

algorithm (hereafter referred to as PF) selects all *urgent* tasks immediately. Then it imposes a total order $\succeq$on the contending task as follows
$x \succeq y \quad \Leftrightarrow \quad \alpha(x,t) \geq \alpha(y,t)$, where the comparison is lexicographic.
Then it chooses as many contending tasks as required to make all processors busy.

$\underline{AlgorithmPF}$

1) Select all *urgent* tasks (n_0).
2) Impose total order $\succeq$on the contending tasks, as defined above
3) Select $m - n_0$ highest priority contending tasks.
4) Schedule the selected tasks.

3.3.3 Proof Idea

Baruah et. al. first prove, using a network flow argument, that a pfair schedule exists, given condition (2.1.1). Then they prove by induction on t that if algorithm PF's schedule(S_{PF}) is pfair at time t then it is so at time $t+1$ also. It is here that pfairness come in handy. If your induction hypothesis is strong enough you will be able to prove strong results. So by assuming pfairness to hold till time t they succeed in proving it at time $t+1$, while people working on simple feasibility were unable to prove the existence of feasible schedules given the same condition (2.1.1). The interested reader should refer to [BRH96] for the detailed proof and a sample run of the Algorithm PF.

3.3.4 A Static Priority Pfair Scheduling Algorithm

Baruah's notion of pfairness inspired researchers to study pfair scheduling in detail. Baruah [BRH??] has provided results for the case when the m processors are no longer identical. Ramamurthy and Moir [RMM00] present a static priority algorithm for m multiprocessors and derive a sufficient condition under which it succeeds in scheduling the task set. Their algorithm is a simple *weight monotonic algorithm(WM).* It always chooses the m tasks having highest weights. If for every $x \in T$ there exists a $t \in (0, \lfloor 1/W_x \rfloor)$ such that $\sum_{y<x} \lceil W_y * t \rceil \quad < \quad m*t$, then T is scheduled in a pfair manner by WM. They derive this condition by proving that if the generated schedule in not pfair then the above condition does not hold.

4 Priority Inheritance Protocols

Real time systems often suffer from the well-known problem of *priority inversion.* This problem was first discussed by Lamport and Redell [LMP80] in the context of monitors[2].This is caused when a high priority task,T_1, tries to acquire a resource currently held by a low priority task,T_3, and blocks on it. In such a scenario, if there is a task with intermediate priority, T_2, it can preempt T_3 and start running thereby delaying the execution of T_1 by an arbitrary amount of time. Such a situation is clearly not desirable in real time systems. The use of *priority inheritance protocols* is one approach that Sha et. al. [SHA90] suggested to rectify the priority inversion problem.

4.1 The Basic Priority Inheritance Protocol

4.1.1 The protocol

1. Highest priority task T, is assigned the processor. It will be blocked if it attempts to lock an already locked semaphore S. When T exits its critical section, in case it did acquire S, the highest priority task, if any, blocked by task T will be awakened.

2. A task T uses assigned priority unless it is in a critical section and blocks higher priority tasks. It inherits the priority of the highest priority task blocked by it. On leaving the critical section, it resumes its original priority.

3. Priority Inheritance is transitive - if T_3 blocks T_2 and T_2 blocks T_1 then T_3 inherits P_1, the priority of T_1.

4. Task T can preempt T_L, a lower priority task, if T is not blocked and T_L's current priority is less than T's current priority.

This protocol ensures the following property:

If there are n lower priority tasks that could block T and m is the number of semaphores that can be used to block T, then T can be blocked for at most the duration of $min(n, m)$ critical sections.

It is important to note that under the basic priority inheritance protocol, a high priority task can be blocked by a low priority task in one of two situations . First there is the *direct blocking*: a situation in which a higher priority task attempts to lock a locked semaphore. Direct blocking is necessary to ensure the consistency of shared data. Second, a medium priority task T_2 can be blocked by a low priority task T_3, which inherits the priority of a high priority task T_1. We call this form of blocking *push-through blocking.* This is necessary to avoid having a high priority task being indirectly preempted by the execution of a medium priority task.

[2]Monitors are a mechanism to achieve mutual exclusion

4.1.2 Problems with the protocol

The basic priority inheritance protocol has the following two problems. First, the basic protocol, by itself, does not prevent deadlocks. For example, suppose that at time t_1, task T_2 locks semaphore S_2 and enters its critical section. At time t_2, task T_2 attempts to make a nested access to lock semaphore S_1. However, task T_1, a higher priority task, is ready at this time. Task T_1 preempts task T_2 and locks semaphore S_1. Next, if task T_1 tries to lock semaphore S_2, a deadlock is formed. Second, the blocking duration for a task, though bounded, can still be substantial.

4.2 The Priority Ceiling Protocol

4.2.1 Goals of the protocol

The protocol is designed to ensure that when a task T preempts the critical section of another task and executes its own critical section Z, the priority at which this new critical section Z will execute is guaranteed to be higher than the inherited priorities of all the preempted critical sections.

4.2.2 Priority Ceiling of a Semaphore

The *priority ceiling* of a semaphore is defined to be the priority of the highest priority task that may attempt to acquire this semaphore.

4.2.3 The protocol

1. Task T, which has the highest priority among the tasks ready to run, is assigned the processor and let S^*be the semaphore with highest priority ceiling of all semaphores locked by other tasks. Task T will block if it tries to lock a semaphore and its priority is not higher than the ceiling of S^*. In this case, T is said to be blocked by the task holding S^*.

2. If task T blocks higher priority tasks, T inherits P_H, the highest of the priorities of tasks blocked by T. Inheritance is transitive.

3. A task T, when it does not attempt to enter a critical section, can preempt another task T_L if its priority is higher than the priority, inherited or assigned, at which task T_L is running.

This protocol ensures the following properties:

A task T can be blocked by a lower priority task T_L, only if priority of task T is no higher than the highest priority ceiling of all the semaphores that are locked by all lower priority tasks when T is initiated. Suppose critical section Z of T_j

is preempted by task T_i which enters its critical section Z'. Under this protocol, task T_j cannot inherit a priority level which is higher than or equal to that of task T_i, until task T_i completes. Also, transitive blocking is not possible under this protocol. The protocol ensures that the period, during which a high priority task is blocked by a low priority task, is bounded. A task T can be blocked for at most the duration of one critical section, among all those critical sections of lower priority tasks which can cause blocking of T.

4.3 Immediate Ceiling Priority Inheritance Protocol

This protocol is a simple variation of the Priority Ceiling Protocol mentioned above. Under this protocol, as soon as a task acquires a semaphore, its priority is raised to the ceiling priority of the semaphore. The protocol ensures that a task is blocked by only a single lower priority task, and that too when it is released. This implies that once a task starts executing, no further low priority tasks can cause blocking of this task.

5 Lock Free Shared Objects

5.1 Motivation

Priority Inheritance Protocols do away with the problem of priority inversion but they lead to additional overheads inside the operating system. This is because the OS has to keep track of ceiling priorities and inherited priorities. Moreover, calculating ceiling priorities requires one to know exactly which tasks access which objects. Therefore, tasks cannot be dynamically added easily. This led Anderson et. al. [AND97] to consider “lock free” ways of accessing shared objects.

5.2 Features of Lock Free Objects

As the name suggests, these objects are accessed without locking. They do not lead to priority inversion, because no locking is involved, but individual tasks may interfere with each other. The number of interferences can be bounded by appropriate scheduling. This mechanism of implementing shared objects pertains to periodic hard real-time tasks that share lock-free objects on a *uniprocessor.*

5.3 Implementation

Lock free objects are implemented using retry loops:

```
typedef struct {
  valtype data;
  Qtype *next;
} Qtype;

Qtype *head, *tail;

Enqueue(valtype input) {
  Qtype *old, *new;
  Qtype **addr;
  *new = (input, NULL);
  repeat {
    old = tail;
    if (old != NULL)
      addr = &(old->next);
    else
      addr = &head;
  } until (CAS2(&tail,addr,old,NULL,new,new));
}
```

Two word Compare and Swap (CAS2) instruction is used to atomically carry out the update. The "repeat-until" loop is executed repeatedly until CAS2 succeeds. Note that the queue is not actually locked by any process.
Interference is said to occur when a task is preempted in the middle of the loop and CAS2 fails. The retry loop is potentially unbounded. If it is somehow ensured that the retry loop is bounded then the implementation is termed as *wait-free.*

5.4 Intuition behind lock free objects

If it is indeed the case that retry loops are unbounded, then how can we guarantee that deadlines will be met. The primary intuition behind using lock free objects is that if a task T_i causes task T_j to experience a failed update, then T_j cannot cause T_i to experience a failed update. This is because only one of them can have higher priority.
A set of tasks using shared lock free objects is schedulable only if there is enough free processor time to accommodate the failed updates. Although lock free objects hog more CPU time (since computation is repeated), still it can win over lock based schemes since it gives little hindrance to high priority tasks - which are more likely to miss deadlines under lock based schemes.

5.5 Universal Construction

Any kind of shared object can be dealt with by the following *Universal Construction:*

- Numerous copies of the implemented object are kept.
- Current copy is indicated by a shared object pointer.
- Retry loop consists of

  - shared object pointer is read.
  - local copy of the object is made.
  - desired operation is done on the local copy.
  - Attempt is made(using CAS) to make the shared pointer point to the local copy

We can modify the above universal construction slightly to obtain a wait free implementation. Each task can be required to announce that it has a pending shared-object access operation. Using this information, higher priority tasks can help lower priority tasks to complete those pending operations. This again leads to the problem of priority inversion since a medium priority task will have to wait for a high priority task helping a lower priority task. But when using lock free objects, time to complete an object access decreases with increasing priority. Thus certain task sets are schedulable under lock free scheme but not under wait free.

6 RT Linux

6.1 Incompatibility of time sharing and real time systems

Conventional real time systems have several problems preventing them from being used as hard real time operating systems. The root of these problems is that time sharing operating systems try to optimize the average case but real time systems have to consider the worst case in order to guarantee deadlines. Conventional time sharing operating systems and real time systems have contradictory design goals e.g virtual memory is highly desirable in the former but undesirable in the latter (unless some provision is made for locking of pages in memory for real time tasks).

Linux can't be used as a real time system for a variety of reasons, most notably the presence of a non-preemptible kernel and the disabling of interrupts as a means for coarse grained synchronization. Promiscuous use of disabling and enabling interrupts inflicts unpredictable interrupt dispatch latency. It is difficult to bound the time spent with interrupts disabled and even if such a bound is obtained it may be too large to be useful.

6.2 The RT Linux solution

RT Linux slips a small, simple real time operating system underneath Linux. Linux becomes the idle task for this kernel and we preempt Linux whenever a real time task needs the processor. The disabling of interrupts for long periods of time is prevented by putting a layer of emulation software between the Linux kernel and the interrupt controller hardware, a technique similar to that described in [STO93]. Any attempt by the Linux kernel to disable interrupts is caught by the emulator. Also all hardware interrupts are directly caught by the emulator and immediately serviced. Using this approach the interrupt dispatch latency has been brought down close to the hardware limit of well under $30\mu sec$. The emulation is achieved by replacing all occurrences of cli (instruction for disabling interrupts) and sti (instruction for enabling interrupts) in the linux kernel, by their corresponding emulated versions S_CLI and S_STI. The idea of emulating cli and sti to achieve real time performance is due to Victor Yodaiken [BAR96].

S_CLI merely marks a bit to signify that Linux has disabled interrupts. When an interrupt occurs, the emulator first sees if there is a real time handler for this interrupt. If yes, it is immediately invoked. Next, if the real time handler is willing to share the interrupt with Linux,the emulator checks to see whether linux has interrupts enabled. If yes, the interrupt is passed on to Linux else it is just marked as pending. Later when Linux enables its interrupts (using S_STI) all the pending and unmasked interrupts are serviced. Thus, the amount of time spent with interrupts actually disabled is very less which accounts for the extremely low interrupt latency.

6.3 Real time tasks in RT Linux

In the initial design, each real time task used to run in its own address space. But this leads to the associated overhead of system calls and frequent TLB (Translation Lookaside Buffer) invalidations. This leads to greater security but is clearly a luxury when processor time is at a premium. Hence this design has been discarded and now all real time tasks run in the same address space as that of the kernel and at the maximum privilege level. This is highly error prone and a bug in a single real time application can wipe out the entire system. But it is expected that real time systems are deployed after much testing and debugging and hence this should not be much of a concern. Further optimization is done by not using the hardware context switch mechanism provided by x86 but by switching context through software. This mechanism saves a minimum of state information as opposed to the hardware mechanism which saves too much state information, for example real time tasks by default have only integer context, hence the floating point context is switched only if the process to be switched in requires it.

RT Linux is module based. The tasks are loaded as kernel modules and hence can be dynamically added. The scheduler itself is a loadable kernel module. Hence it is possible to change the scheduler without having to recompile the kernel. The default RT Linux scheduler is a simple preemptive static priority scheduler. Alternate policies such as the Earliest Deadline First (EDF) or the Rate Monotonic (RM) have also been implemented.

6.4 Timer Resolution

One of the problems with Linux is that its timer resolution is coarse (10 ms). Hence a task can't be scheduled at a precision of more than 10ms. Scheduling precision is measured in terms of *task release jitte*r which is defined as the difference between the exact time when the task was to be scheduled and the time when it was actually scheduled. Jitter can be lowered by increasing the clock frequency but then more time is spent in handling clock interrupts.

In RT Linux this tradeoff is avoided by providing timer operation in two distinct modes. The normal periodic mode is used when a high degree of scheduling precision is not required. When the jitter has to be minimized, the timer can be operated in oneshot mode. Here, on every interrupt, the clock is reprogrammed to interrupt the CPU at exactly the time when the next task has to be scheduled. Global time is kept track of by summing up all intervals between interrupts or (on Pentium onwards) by using the on-chip timestamp counter. Periodic interrupts are simulated for Linux. But this mode has the associated overhead of reprogramming the clock everytime which is costly in terms of time.

6.5 Inter-process Communication

An upshot of the RT Linux interrupt emulation is that a linux task can be preempted by a real time task even while the former is inside a system call and the linux kernel data structures are in an inconsistent state. Hence no linux kernel function can be safely called from a real time task. This seems to defeat the original purpose of providing sophisticated services to a real time application. But RT Linux assumes that a real time application can be split into two distinct parts: one with hard real time constraints but which is very simple and does not require any sophisticated services and one which does more sophisticated processing such as GUI, disk I/O etc. but which has no real time constraints. A typical example is a data acquisition application which consists of a real time part which collects data, and a non real time part which takes care of data logging and display.

The real time component of the application is loaded as a kernel module while the non real time component runs as a normal Linux task. RT Linux provides two means of communication between the real time tasks and the linux processes : RT fifos and shared memory. An RT fifo is a first in first out byte stream which does not preserve message boundaries and can be used for unidirectional transfer. On the RT side, RT fifos export non blocking lock free functions to read and write while on the linux side they export a standard character device interface. Real time tasks may also set up a handler corresponding to an RT fifo which is invoked whenever a Linux task writes to it. Thus real time tasks need not poll the fifo for arrival of data. For communication using shared memory, a shared memory pool is set aside. A real time process may directly read/write to this pool. The linux process may map this portion of memory to its own address apace and then access it in the usual way.

7 Implementation of Simple Priority Inheritance protocol in RT Linux

We have modified the standard RT Linux mutex package to implement the simple priority inheritance protocol as defined in section 3.1.

7.1 Design

The basic data structure associated with each task is a doubly linked list of mutexes that it owns. Every task T_j has a base priority P_j and each mutex M has a priority field P_M. Let us denote the list of task T_j by L_j. The dispatch priority D_j of T_j is defined as $D_j = max(\{P_j\} \cup \{P_M : M \in L_j\})$. We seek to maintain the following invariants:

1. T_j owns M $\Rightarrow P_M = max(\{P_j\} \cup \{D_k : T_k$ is directly or indirectly blocked on M$\})$.
2. M $\in L_j \Leftrightarrow T_j$ owns M
3. The list L_j is sorted on the priority field of the mutexes.

It is clear that if the above invariants are maintained and the scheduler chooses the task with the highest dispatch priority, the conditions of the priority inheritance protocol shall be satisfied. To maintain these the mutex operations are carried out as follows:

1. Mutex Locking: When a task T_j acquires a mutex M, we set $P_M = P_j$ and insert M at the head of L_j. Since no task is currently blocked on M, (1) is satisfied. (2) is trivially satisfied. Since all mutexes in L_j have priority at least P_j, (3) is also satisfied.

2. Blocking on a mutex: When a task T_j blocks on a mutex M_k owned by T_i , do the following:

    (a) Set $P_{M_k} = max(P_{M_k}, D_j)$.

    (b) Move M_k in L_j so as to keep L_j sorted.

    (c) If T_i is itself blocked on M_l owned by T_n, repeat the above step with j=i, k=l and i=n.

    When T_j blocks on M_k, we first update P_{M_k} so as to satisfy (1). We traverse the entire blocking chain (by step 3) and hence all P_M such that T_j is directly or indirectly blocked on M get updated. Thus (1) is satisfied. We are not changing the list of any task, hence (2) is satisfied. Step 2 ensures that each list remains sorted.

3. Mutex Unlocking: When a task T_j releases a mutex M, we simply remove M from L_j. Now M is not owned by any task hence (1) is trivially satisfied. Clearly (2) and (3) are also satisfied.

7.2 Implementation and Running Time

A few implementation details need to be pointed out. The field that holds base priority in the normal implementation, has been modified to hold the dispatch priority and a new field has been added to hold the base priority, so that the scheduler does not have to be modified. Since the mutex operations involve list operations, they have to be synchronized. For this, a spinlock is associated with the list of each task. This spinlock has to be acquired before doing any operation on that list.

Locking and unlocking a mutex are still O(1). Since the mutex can be added/removed from the list and the dispatch priority updated in O(1) time. For this, it is important that the list be sorted. Blocking on a mutex has a worst case bound of O(m+n) where m is the number of tasks and n is the number of mutexes.
This is because all the tasks may be blocked in a chain and the list of each task may have to be traversed in order to keep the list sorted.

8 Conclusions and Future Work

In this course we have looked at various aspects of real time systems ranging from the theoretical foundations of real time scheduling to the implementation of a real time kernel in a freely available operating system. The scheduling problem has been deeply studied and a wealth of literature is available. For uniprocessors efficient optimal solutions are available both under the static priority (rate monotonic) and the dynamic priority (Earliest Deadline First, Least Laxity First) category. For multiprocessors however, no optimal static priority algorithm is known. However as long as the utilization factor is less than or equal to the number of processors, the tasks can still be scheduled in a p-fair manner by following the dynamic priority algorithm due to Baruah et al. P-fair scheduling is a relatively new concept and a natural direction of future work is to investigate p-fair schedulability under precedence constraints and resource sharing.

But static priority algorithms are significantly simpler to implement and imply less run time scheduling overhead, hence the existence of better static priority algorithms is worth investigating. Another approach has been to partition the jobs among processors so as to avoid the overhead of task migration. The well known uniprocessor algorithms can then be applied on the individual processors. To find an optimal assignment of jobs to processors has been proven to be NP complete. Heuristic algorithms similar to those for bin-packing have been proposed for the same [DVR86].

We also looked at the simple priority inheritance protocol to curb the problem of priority inversion. The Priority Ceiling Protocol, a better but significantly more complex protocol was also studied. A lock free protocol was also studied which does away with the problem of priority inversion. It would be an interesting exercise to make a quantitative comparison of the two approaches.

Lastly we looked at the implementation of a real time system in a widely available non-real time operating system (Linux). RT Linux has been designed to require minimal changes to the Linux kernel, essentially the changes are limited to the low level interrupt wrappers and the routines to disable and enable interrupts. So here we have an off the shelf general purpose as well as hard real time operating system that can ride the wave of the main Linux development effort. In all, RT Linux provides a viable platform for hard real time applications which also require sophisticated services such as Xwindows, networking etc. It would be indeed interesting to implement the various scheduling schemes studied above in the RT Linux kernel.

References

[AND97] J.H. ANDERSON,S. RAMAMURTHY,K. JEFFAY,Real-Time Computing with Lock-Free Shared Objects,*ACM Trans. on Comp. Sys.,vol. 15.,no. 2,1997*

[BAR96] M. BARABANOV,V. YODAIKEN,Real-Time Linux,*Linux Journal,1996*

[BRH96] S. BARUAH,N. COHEN,G. PLAXTON,D. VARVEL,Proportionate Progress:A Notion of Fairness in Resource Allocation,*Algorithmica 1996*

[BRH??] S. BARUAH,Scheduling Periodic Tasks on Uniform Multiprocessors,*Accepted for publication in Inform. Proc. Letters*

[DRT74] M. DERTOUZOS,Control Robotics:The Procedural Control of Physical Processes,*Proc. IFIP Cong,1974*

[DRT89] M. L. DERTOUZOS,A.K. MOK,Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks,*IEEE Trans. on Soft. Engg.,vol. 15,no. 12,1989*

[DVR86] S. DAVARI,S.K. DHALL,An On-Line Algorithm for Real-Time Tasks Allocation,*Proc. of the 7th IEEE RTSS,1986*

[KHM92] A. KHEMKA,R.K.SHYAMSUNDAR,Multiprocessor Scheduling of Periodic Tasks in Hard Real-Time Environment,*IPPS 1992*

[LHC89] J. LEHOCZKY,L. SHA,Y. DING,The Rate Monotonic Scheduling Algorithm:Exact Characterization and Average Case Behaviour,*Proc. of the 10th IEEE RTSS,1989*

[LIU73] C.L. LIU,J.W. LAYLAND, Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment,*JACM,vol. 20,no. 1,1973*

[RMM00] S. RAMAMURTHY, M. MOIR, Static Priority Periodic Scheduling on Multiprocessors,*Proc. of the 21st IEEE RTSS,2000*

[SHA90] L. SHA,R. RAKUMAR,J.P. LEHOCZKY,Priority Inheritance Protocols: An Approach to Real-Time Synchronization,*IEEE Trans. on Comp.,vol. 39,no. 9,1990*

[STO93] D. STODOLSKY,J.B. CHEN,B. BERSHAD,Fast Interrupt Priority Management in OS kernels,*Proc. of the 2nd USENIX Symp.,1993*