NANYANG TECHNOLOGICAL UNIVERSITY

SCE13-0026

Micrium μC/OS-III++

Submitted in Partial Fulfillment of the Requirements for the Degree of Bachelor of Computer Engineering of the Nanyang Technological University

by

Do Nhat Minh

School of Computer Engineering

April 5, 2014

Abstract

Real-time operating systems play an important role in time and safety-critical software systems used in many fields, such as avionics, automotives and defense applications. μC/OS-III is an open-source real-time operating system which aims to be used on embedded devices with restricted resources. μC/OS-III has many useful features; however, it does not have better algorithms proven in more recent advances in real-time systems research.

In this thesis, a hybrid scheduler where Earliest Deadline First scheduling runs on top of Fixed-Priority scheduling and Priority Ceiling Protocol for time-guaranteed resource sharing are implemented to enhance μC/OS-III.

Acknowledgements

I am particularly grateful to my supervisor, Dr. Arvind Easwaran from the School of Computer Engineering, for being extremely supportive of my project choice and for giving invaluable advice during the course of this final year project.

I am grateful to Mr. Muhamed Fauzi Bin Abbas, whose advice has helped me overcome hurdles encountered along the way.

I would also like to thank Tran Ngoc Khanh Thy for the continued emotional support during the course of this final year project.

Contents

List of Figures

Listings

Chapter 1

Introduction

An operating system (OS) is a collection of software, or software components, which can be characterized as serving the following purposes, (1) interfacing with the underlying hardware to provide convenient abstractions for application programmers, and (2) managing the programs running on the system so that misbehaving programs do not impede others (Witchel, E. , 2009).

A real-time operating system is an OS which must adhere to a real-time constraint. In such a system, the timeliness of the results from programs are as important as the correctness of such solutions. The system risks catastrophic failures if deadlines are missed. Some important applications for real-time operating systems are in avionics and automotives, where missing task deadlines leads to lives lost.

μC/OS-III is an open-source priority-based preemptive real-time multitasking operating system for embedded systems. It has many useful features which aims to cut development time (Labrosse, J. J. , 2010, pp. 27-31).

However, μC/OS-III does not implement deadline management but defer this task to the programmer. Moreover, uC/OS-III has no facility to specify nor keep track of the deadlines for running tasks of the system.

1.1 Goal

The goal of this thesis is to implement better algorithms for μC/OS-III. For task scheduling, a hybrid scheduler where Earliest Deadline First scheduling runs on top of Fixed-Priority Scheduling is implemented. For resource sharing with mutexes, Priority Ceiling Protocol is implemented. In addition, a new type of tasks, namely recurrent tasks, is introduced as a complement to the new scheduler.

1.2 Overview

This thesis is divided into the following.

Background explains concepts of scheduling algorithms and protocols for mutex acquisition and release.

Literature Review summarizes how Earliest Deadline First and Priority Ceiling Protocol is implemented for Ada 2005.

Overview of μC/OS-III provides a brief introduction to the current state of μC/OS-III regarding the scheduler and mutex operations.

Enhancements to μC/OS-III discusses how several enhancements to μC/OS-III are implemented.

Experiment introduces a sample application developed to test the new capabilities of μC/OS-III.

Results shows benchmarks for the new code, as well as evaluation for these benchmarks.

Conclusion summarizes the thesis and discusses future directions.

Chapter 2

Background

2.1 Scheduling Algorithms

A scheduling algorithm is an algorithm to determine the ordering of task executions in order to maximize resource utilization, while satisfying safety and correctness (Liu, C. L. and Layland, J. W. , 1973).

A real-time system consists of a number of tasks, each of which has a deadline and a period. Tasks must be completed before their deadlines or risk catastrophic consequences, e.g. a plane might crash if the task that sends sensor failure status misses deadlines. Task period is the interval between release times of instances of a task. Request rate is the reciprocal of task period. A task set is a collection of tasks to be scheduled. A task set is feasible when there exists an ordering where no deadline is missed.

2.1.1 Rate Monotonic Scheduling

Rate Monotonic Scheduling belongs to the class of fixed priority scheduling algorithms. A fixed priority scheduling algorithm is a scheduling algorithm where task orderings are based on statically assigned priorities of the tasks. The rate monotonic priority assignment assumes that a task with higher request rate is assigned a higher priority. C. L. Liu and J. W. Layland in their seminal paper have proven that for fixed priority scheduling, if there exists a feasible assignment, the rate monotonic assignment is also feasible (Liu, C. L. and Layland, J. W. , 1973). In other words, rate monotonic scheduling algorithm (RMS) is optimal.

2.1.2 Utilization Analysis

CPU utilization for a task is the ratio between the time spent in execution and its period. CPU utilization for a task set is the summation of CPU utilization of each task in the set.

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

where C_i is execution time and T_i is period for task i, n is the number of tasks in the task set and U is CPU utilization.

2.1.3 Utilization Analysis for RMS

It is proven that CPU utilization for a task set in RMS must be kept below an upper bound in order to guarantee feasibility. This upper bound is

$$U_{RMS} \leq n(2^{\frac{1}{n}} - 1)$$

where n is the number of tasks in that task set and U_{RMS} is CPU utilization (Liu, C. L. and Layland, J. W. , 1973).

As n tends towards infinity, this expression will tend towards

$$\lim_{n \to \infty} n(2^{\frac{1}{n}} - 1) = \ln 2 \approx 0.693147\ldots$$

As a rule of thumb, a task set is feasible under fixed priority scheduling when its CPU utilization is below 69.3%. The other 30.7% of CPU time can be reserved for other non real-time tasks.

However, this upper bound is pessimistic. It has been shown that a randomly generated task set will meet all deadlines when utilization is below 85% if exact task deadlines and periods are known, which might be difficult to achieve (Lehoczky, J.; Sha, L. and Ding, Y. , 1989).

2.1.4 Earliest Deadline First Scheduling

Earliest Deadline First Scheduling (EDF) is a scheduling algorithm where task orderings are based on deadlines of task instances. Under this scheme, an instance of a task is assigned highest priority if its deadline is nearest, while an instance of a task with a deadline that is farthest is assigned the lowest priority.

2.1.5 Utilization Analysis for EDF

The utilization bound for EDF is

$$U_{EDF} \leq 1$$

where U_{EDF} is the CPU utilization (Liu, C. L. and Layland, J. W. , 1973).

From the utilization bounds for RMS and EDF, it is trivial to see that EDF guarantees all deadlines of a task set at a higher load than RMS. Therefore, EDF is more desirable from a resource utilization standpoint.

2.2 Resource Sharing with Mutual Exclusion

In order to maximize utility of resources in computer systems, resources are shared among the tasks that needs them. These resources include hardware such as printers or software such as a region of computer memory. However, some resources must only be accessed by one task at a time. Examples include printers where unprotected, concurrent access to the print queue will result in sentences from different documents interleaving each other.

Mutual exclusion, or mutex, is a technique to protect such resources from concurrent access. Mutex was first identified and an implementation for which was first introduced by Dijkstra in his seminal 1965 paper (Dijkstra, E. W. , 1965).

2.2.1 Priority Inversion

In the context of real-time systems, there is a problem of priority inversion in the use of mutexes for resource sharing. Priority inversion occurs when a high

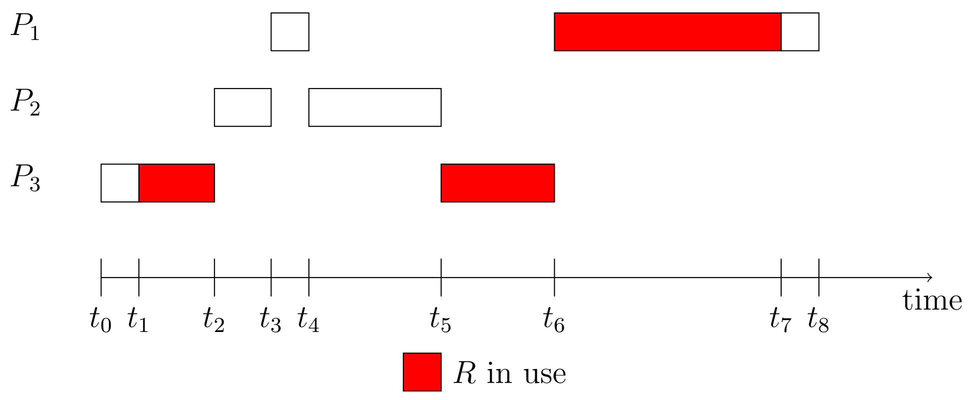

Figure 2.1: Priority Inversion

priority task wants to access a shared resource and is blocked by a lower priority task holding the mutex for that resource.

Figure 2.1 shows an example of priority inversion. In this example, P_1 has highest priority, while P_3's priority is the lowest. At t_0, P_3 begins execution. At t_1, P_3 accquires the mutex R. At t_2, P_3 is preempted by P_2. At t_3, P_1 preempts P_2 and then at t_4 tries to acquire R which blocks P_1 and P_2 resumes execution. P_2 finishes execution at t_5. P_3 then resumes execution and releases R and finishes execution at t_6. P_1 then resumes execution, having successfully acquired mutex for R. At t_7, P_1 releases R and finishes execution at t_8.

In the above example, the highest priority task, namely P_1, is preempted by a lower priority task, namely P_2. It is easy to see that this example can be expanded to include multiple medium priority tasks, where the highest priority task is preempted consecutively by those tasks. In that case, the highest priority task may even miss its deadline because of the chain blocking of those lower priority tasks.

A real-life example of priority inversion is the incident of Mars Pathfinder spacecraft. There were many software tasks running on the Mars Pathfinder's VxWorks real-time operating system. The high priority task `bc_dist` was blocked by the much lower priority task `ASI/MET`, which in turn was blocked by other medium priority tasks. `bc_dist` therefore missed its deadline, which was before the execution of the `bc_sched` task. The software on Mars Pathfinder dealt with

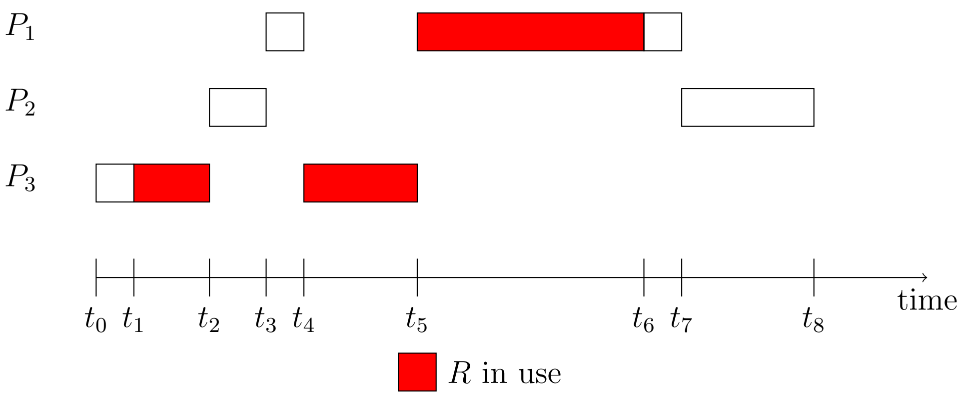

Figure 2.2: Priority Inheritance Protocol

this missed deadline by rebooting itself. Although no catastrophe resulted, the rest of the activities for that day was only accomplished the next day (Reeves, G. E. , 1997).

2.2.2 Priority Inheritance Protocol

In order to solve the problem of Priority Inversion, Lui S. et. al. proposed a class of Priority Inheritance Protocols in their seminal 1990 paper (Lui S.; Rajkumar, R.; Lehoczky, J.P. , 1990). The basic idea for Priority Inheritance Protocols is when a lower priority task blocks a higher priority task inside its critical section, it is promoted to the same priority as that higher priority for the duration of its critical section.

Figure 2.2 shows how priority inheritance can help mitigate the problem of priority inversion. The task set illustrated is the same as in figure 2.1. The tasks are scheduled the same way as in figure 2.1 from t_0 to t_4. However, when P_1 tries to take the mutex R and is blocked by P_3, P_3's priority is raised to be the same as P_1's. Thus, P_2 is no longer able to preempt P_3 and P_3 can run until it releases R and finishes at t_5. As P_1's priority is higher than P_2's, it is scheduled to run at t_5. P_1 releases R at t_6 and finishes at t_7, as which point P_2 is scheduled to run till finish.

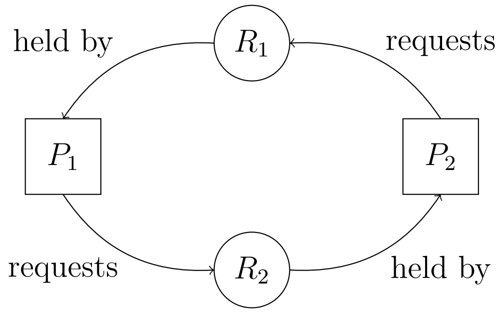

Figure 2.3: Deadlock - Resource Allocation Graph

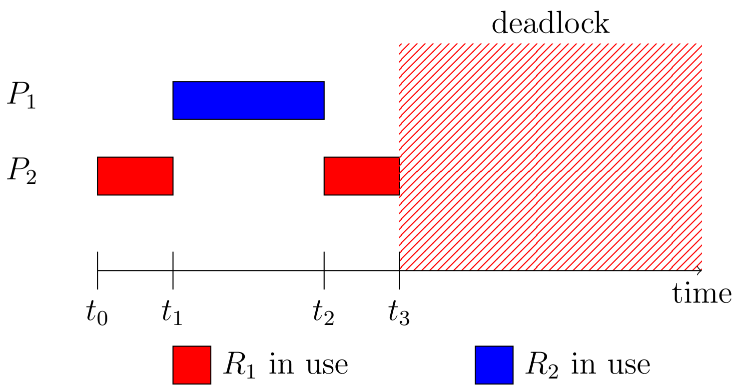

Figure 2.4: Deadlock

2.2.3 Deadlock

In the use of mutexes for resource sharing, a deadlock occurs when a task waits on a mutex to a resource held by another task, which in turn is waiting for another mutex held by the first task, as illustrated in figure 2.3.

Although Priority Inheritance Protocol is well-suited to handle the problem of priority inversion, it does not help to prevent deadlocks, as illustrated in figure 2.4. In this example, P_1 has higher priority than P_2. At t_0, P_2 starts executing and acquires the mutex R_1. At t_1, P_1 becomes ready and starts executing immediately, as it has higher priority than P_2. P_1 acquires the mutex R_2 at the start of its execution. At t_2, P_1 tries to take R_1 but is blocked by P_2, which holds that mutex. With P_1 blocked, P_2 then resumes execution and at t_3 tries to take R_2, which is currently held by P_1. As P_1 and P_2 are now waiting for each other, no progress can be made and a deadlock occurs.

Necessary Conditions for Deadlocks

Necessary conditions for deadlocks to arise, also known as the Coffman conditions, are as follows. If in a system, any one of the four conditions is not met, deadlocks will not occur (Coffman, E.G. Jr.; Elphick, M.; Shoshani, A. , 1971):

Mutual Exclusion

At most one task can have access to a resource at any instant.

Hold and Wait

After gaining access to the resource, the task has to wait before it can proceed; e.g. it has to wait for other resources to become available.

No Preemption

The OS cannot take away a resource from the task once it has successfully gained access to the resource.

Circular Wait

There exists a set of resouces $P = \{P_1, P_2, \ldots, P_n\}$ where P_1 is waiting for a resouce held by P_2, which is waiting for a resource held by P_3, $\ldots$, and so on, with P_n waiting for a resource held by P_1.

Deadlock Handling

There exist approaches to handling deadlocks, namely deadlock detection, deadlock prevention and deadlock avoidance.

Under deadlock detection, deadlocks are allowed to happen. On the other hand, the OS reserves the right to kill and restart tasks involved in the deadlock and the acquired resources are preempted to give to other tasks in an attempt to break the deadlock. However, restarting tasks and preempting resources could lead to inconsistent state for the resource under protection.

Deadlock prevention is implemented by breaking one of the necessary conditions for deadlocks to occur. Breaking mutual exclusion is imposible for resources which must be accessed by only one task. Resource holding or hold and wait can be prevented by making all task acquiring all resources before they can proceed;

however, this is very inefficient use of resources. Resource preemption is infeasible for certain resources. Finally, circular wait can be prevented by giving specific ordering for all resources and requiring all tasks adhere to this ordering for resource acquisition.

Under deadlock avoidance, the system is prevented from ever reaching an unsafe state. This requires the OS to have some information about resource usage of each task. One algorithm for deadlock avoidance is Priority Ceiling Protocol, which is introduced in the next section.

2.2.4 Priority Ceiling Protocol

Priority Ceiling Protocol (PCP) extends Priority Inheritance Protocol by adding more restrictions on mutex acquisition so as to implement deadlock avoidance. Under this scheme, each mutex is assigned a priority ceiling which is the same priority as the highest priority task that will use that mutex. A task is allowed to acquire a mutex when its priority is higher than the system ceiling, which is equal to the highest ceiling among the ceilings of mutexes currently locked by tasks other than the current task. When a low priority task blocks higher priority tasks from acquiring a mutex, the priority of the low priority task is raised to the highest priority among the higher priority tasks, so in this regard, it is the same as Priority Inheritance Protocol (Lui S.; Rajkumar, R.; Lehoczky, J.P. , 1990).

Figure 2.5 shows how PCP can handle deadlocks and still prevent chained blocking like Priority Inheritance Protocol. In this example, P_1 has the highest priority, followed by P_2 and then P_3. Mutexes R_1 and R_2 are both used by P_1 and P_2 and thus, under PCP, are assigned priority ceilings which are at the same priority as P_1's. At t_0, P_3 begins execution and acquire mutex R_1. The system ceiling is raised to be the priority ceiling of R_1, which is the same as P_1's priority. At t_1, P_2 starts executing, as it has higher priority than P_3. At t_3, P_1 preempts P_2 and begins execution. At t_4, P_1 tries to acquire R_2, but is denied and put into pending state, as its priority is not higher than the current system ceiling. P_3 inherits P_1's priority for blocking P_1 and is scheduled to run immediately. It then acquires successfully R_2 at t_5, as there is no other active mutex acquired by

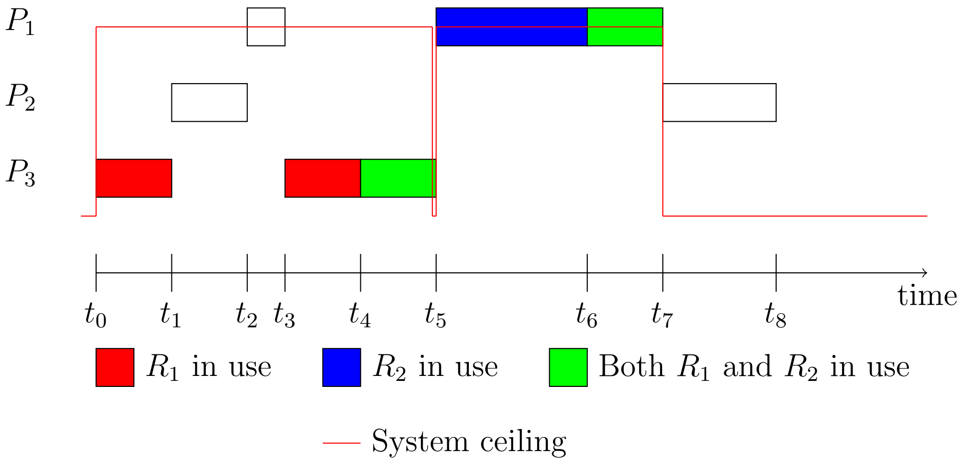

Figure 2.5: Priority Ceiling Protocol

any task other than P_3, and runs till completion at t_5, at which point it releases both R_1 and R_2. P_1 is then woken up, acquires R_2, runs until t_6 and acquires successfully R_3. P_1 completes its run at t_7, at which point it releases both R_1 and R_2. The system ceiling is reset to be below P_3's priority. P_2 is then scheduled to run till completion at t_8.

Chapter 3

Literature Review

Burns et al (Burns, A.; Wellings, A.J.; Zhang, F. , 2009) proposed implementing Earliest Deadline First Scheduling on top of Fixed Priority Scheduling, also known as Rate Monotonic Scheduling, as the hybrid scheduling algorithm for Ada 2005. The system model is that a small number of tasks at a low priority are scheduled with EDF, while other higher priority tasks are scheduled with RMS. In this system, the user can specify which priority levels are scheduled with EDF and which higher priority levels are scheduled with RMS.

Cheng et al (Cheng, A.M.K.; and Ras, J. , 2007) proposed an implementation of Priority Ceiling Protocol for Ada 2005. The ceiling priorities of the mutexes are saved into an array, while the locking statuses of them are saved into another. Upon successful locking of a mutex, the task identifier and original priority of the task is saved into an array private to the scheduler. If a task requests for a lock on a mutex but was unsuccessful, it will be queued for future attempt, while the task holding the mutex will be raised in priority to be the same as that task if applicable. On release of a mutex, if there are other tasks waiting on that mutex, the task with highest priority will be woken up and take control of the mutex while the original owner of the mutex will be reset to the original priority.

Chapter 4

Overview of Micrium μC/OS-III

Micrium μC/OS-III is a preemptive multitasking real-time OS targeting embedded devices. In this chapter, the inner working of this OS is examined.

4.1 Task Model

Tasks in μC/OS-III are implemented as normal C functions with their own accompanying stacks and Task Control Blocks. However, unlike normal C functions, tasks are not allowed to return (Labrosse, J. J. , 2010, p. 83).

4.1.1 Types of Tasks

There are two type of tasks, namely run-to-completion and infinite loop.

Run-to-completion tasks must call `OSTaskDel()` at the end of the function (Labrosse, J. J. , 2010, p. 84).

Listing 4.1: Run-to-completion Task

```
void RunToCompletionTask(void* p_arg) {
    OS_ERR err;

    /* do work */

    /* a NULL pointer indicates the current task
     * should be deleted
     */
```

```
    OSTaskDel((OS_TCB*) 0, &err);
}
```

Infinite-loop tasks do not need to call `OSTaskDel()`; however, they must make calls to services in the OS inside the infinite loop in order to yield control of the CPU to other tasks (Labrosse, J. J. , 2010, p. 85).

Listing 4.2: Infinite-loop Task

```
void RunToCompletionTask(void* p_arg) {
    OS_ERR err;

    /* initialization */
    while (1) {              /* or for(;;) */
        /* do work */
        /* must call one of the following
         *     OSFlagPend()
         *     OSMutexPend()
         *     OSPendMulti()
         *     OSQPend()
         *     OSSemPend()
         *     OSTimeDly()
         *     OSTimeDlyHMSM()
         *     OSTaskQPend()
         *     OSTaskSemPend()
         *     OSTaskSuspend()
         *     OSTaskDel()
         */
    }
}
```

4.1.2 Task Creation

In order to create a new task, the user must provide an allocated Task Control Block (`OS_TCB`) and other arguments to `OSTaskCreate()`.

Listing 4.3: `OSTaskCreate()`

```
void OSTaskCreate (OS_TCB        *p_tcb,
                   OS_CHAR       *p_char,
                   OS_TASK_PTR    p_task,
                   void          *p_arg,
```

```
                     OS_PRIO          prio,
                     CPU_STK         *p_stk_base,
                     CPU_STK_SIZE     stk_limit,
                     CPU_STK_SIZE     stk_size,
                     OS_MSG_QTY       q_size,
                     OS_TICK          time_slice,
                     void            *p_ext,
                     OS_OPT           opt,
                     OS_ERR          *p_err);
```

Task Control Block (`OS_TCB`) is a C `struct` containing necessary task-related information on which the whole of μC/OS-III depends for proper functioning. The information contained in a TCB includes a pointer to the top of stack, a pointer to the C function underlying this task, the current state of this task, the priority of this task and many more.

The bare mininum arguments which the user must provide to create a task are a task control block (`OS_TCB*`), a C function implementing that task (`OS_TASK_PTR` which is a `typedef` of `void (*)(void*)`), a positive integer as priority (`OS_PRIO`), an allocated array as the task stack (`CPU_STK*`) and an `OS_ERR*` for error reporting.

4.2 Scheduling Algorithm

μC/OS-III has a priority-based, preemptive scheduler implementing Fixed-Priority Scheduling (Labrosse, J. J. , 2010, p. 141).

Priority-based

Each task are assigned a static priority. The OS schedule them based on their priorities.

Preemptive

Higher priority tasks can preempt lower priority tasks, which means that during execution of a low priority task, if a high priority task is ready, the low priority task may be suspended so as to give CPU time to the high priority task.

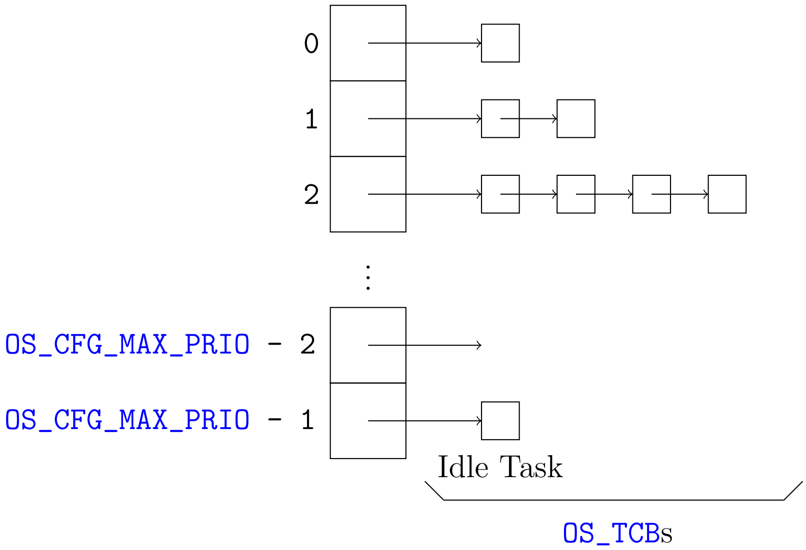

Figure 4.1: Ready List

In order to schedule the tasks, the OS keeps track of ready tasks in ready lists. Tasks with the same priority are put in the same ready list. `OSRdyList` is an array of singly linked lists, where each node is an `OS_TCB`, as shown in figure 4.1.

When a task yields the CPU, the scheduler finds a non-empty linked list with highest priority in `OSRdyList`, removes the first `OS_TCB` from the list and context switches to the task described by that TCB.

4.3 Resource Sharing with Mutexes

μC/OS-III implements Priority Inheritance Protocol for mutexes. There are three basic operations on a mutex, namely mutex creation, acquisition and release. μC/OS-III allows a task to acquire a mutex multiple times (also known as nesting); however, that task must release the mutex an equal number of times.

4.3.1 Mutex Creation

In order to create a mutex, the user must call `OSMutexCreate()` passing in an allocated `OS_MUTEX`*, an optional `OS_CHAR`* as the name of the mutex, and an allocated `OS_ERR`* for error reporting.

Listing 4.4: OSMutexCreate()

```
void OSMutexCreate(OS_MUTEX *p_mutex,
                   OS_CHAR   *p_char,
                   OS_ERR    *p_err);
```

4.3.2 Mutex Acquisition

In order to acquire a mutex, the user must call `OSMutexPend()` passing in a created `OS_MUTEX*`, an optional `OS_TICK` as timeout (`0` for no timeout), an `OS_OPT` which are options or'd together, an optional `OS_TS*` to save the timestamp when the mutex is released with `OSMutexPost()`, and an `OS_ERR*` for error reporting. Below is the signature of the function, along with the pseudocode explaining how it works.

Listing 4.5: OSMutexPend()

```
void OSMutexPend(OS_MUTEX *p_mutex,
                 OS_TICK    timeout,
                 OS_OPT     opt,
                 OS_TS     *p_ts,
                 OS_ERR    *p_err)
{
    check if the arguments passed in are valid;
    if the mutex does not have an owner {
        set nesting level to 1;
        assign the current task as the owner;
        return;
    }

    if the owner of the mutex is the current task {
        increase the nesting level;
        return;
    }

    /* here we know that the mutex already has an
     * owner, which is not the current task.
     *
     *
     * the next if statement implements Priority
     * Inheritance Protocol
```

```
     */
    if the current task has a higher priority
        than the owner {
        change the active priority of the owner
            to be the same as the current task;
    }

    block current task and give control back
        to the OS;

    /* the task resumes here when the OS has given
     * the current mutex to it; i.e. acquisition is
     * successful.
     */
    return success;
}
```

4.3.3 Mutex Release

In order to release a mutex, the user must call `OSMutexPost()` passing in a created `OS_MUTEX`*, an `OS_OPT` for options, and an `OS_ERR`* for error reporting. Below is the signature of the function, along with pseudocode explaining how it works.

Listing 4.6: `OSMutexPost()`

```
void OSMutexPost(OS_MUTEX *p_mutex,
                 OS_OPT    opt,
                 OS_ERR   *p_err)
{
    check if the arguments passed in are valid;
    if the current task is not the owner {
        return failure;
    }

    decrease the nesting level on mutex;
    if the nesting level is not zero {
        report via p_err that the task still nests;
        return;
    }

    /* here we know that the current task has
     * released the mutex thoroughly with no more
     * nesting.
```

```
    */

    if there is no more task waiting on this mutex {
        return success;
    }

    adjust the priority of the current task back to
        its original level;
    give ownership of this mutex to the task waiting
        with highest priority;

    return success;
}
```

Chapter 5

Enhancements to μC/OS-III

In this chapter, several enhancements to μC/OS-III are discussed. A new kind of tasks is introduced, namely recurrent tasks. In addition, Earliest Deadline First is implemented as a new scheduler for μC/OS-III, while Priority Ceiling Protocol is implemented for mutex operations.

5.1 Earliest Deadline First Scheduling

Earliest Deadline First scheduling is a dynamic priority scheduling scheme. The priority of a task in EDF does not depends only on a static priority assigned at creation, but depends on its runtime deadline.

A new field is added to the `struct OS_TCB`, namely `AbsDeadline` of type `OS_TICK`, to represent the absolute deadline in milliseconds since the start of μC/OS-III. By default, this field is set to be the maximum value `OS_TICK` can take, which is $2^{64} - 1$ milliseconds $\approx 5.84842 \times 10^6$ centuries, guaranteeing that deployed devices running this modified μC/OS-III reach their end-of-life long before this time stamp is reached.

When inserting tasks into the ready list, the tasks are sorted in ascending order of absolute deadlines at each priority level. The implementation for the scheduler is still the same as the μC/OS-III's previous implementation.

5.2 Recurrent Tasks

A recurrent task is a run-to-completion task which is created repeatedly at regular intervals. The interval between two instances of a recurrent task is its period. Introduction of this type of tasks complements Earliest Deadline First.

In order to implement this kind of tasks, the OS must keep track of the period as well as the relative deadline of each task. A new type is introduced to keep tract of these information, namely `OS_TD`, short for *task descriptor*.

Listing 5.1: `OS_TD`

```
typedef struct os_td OS_TD;

struct os_td {
    OS_TD       *PrevPtr;
    OS_TD       *NextPtr;
    OS_TICK      Period;
    OS_TICK      RelDeadline;
    OS_TICK      AbsSpawnTime;
    CPU_CHAR    *NamePtr;
    OS_TASK_PTR  TaskEntryAddr;
    void        *TaskEntryArg;
};
```

In order to create a recurrent task, the user must call `OSRecTaskCreate()`, the signature of which is provided below. The minimal required arguments which the user must provide are an allocated `OS_TD*`, the relative deadline of the task (`OS_TICK`), the period of the task (`OS_TICK`), the C function implementing the task (`OS_TASK_PTR`), an allocated `OS_ERR*` for error reporting.

Listing 5.2: `OSRecTaskCreate`

```
void OSRecTaskCreate (OS_TD                 *p_td,
                      OS_TICK                rel_deadline,
                      OS_TICK                period,
                      OS_TICK                delay,
                      CPU_CHAR              *p_name,
                      OS_TASK_PTR            p_task,
                      void                  *p_arg,
                      OS_ERR                *p_err);
```

Below is a skeleton for a C function implementing a recurrent task.

Listing 5.3: Recurrent Task

```
void RecurrentTask (void* p_arg)
{
    OS_ERR err;

    /* initialization */

    /* do work */

    /* there is no need to call the any services by
     * the OS like in the case of infinite-loop task,
     * or call OSTaskDel() at the end like in the
     * case of run-to-completion task
     */
}
```

Each OS_TD is a node for a doubly linked list, named OSTaskList, maintained by the OS to be in ascending order of absolute spawn time.

A dedicated task, named OSTaskSpawner, is created at OS initialization time to create tasks from these task descriptors. Below is the pseudocode explaining how this task is implemented. As μC/OS-III requires a task to have a task control block (OS_TCB) and a stack, the new addition to the code asks the user to provide the number of tasks dynamically created at runtime on behalf of the user via a # define. At compilation time, a chunk of memory with size equal to the maximum number of OS_TCB, each accompanied with its own stack, is reserved. At runtime, OSTaskSpawner takes from that chunk of memory to create the required OS_TCB and stack for each recurrent task.

Listing 5.4: OS_TaskSpawnerTask

```
void OS_TaskSpawnerTask (void *p_arg)
{
    loop forever {
        if there is no task descriptor in
            the task list
        {
            sleep for 10 milliseconds;
```

```
            /* this task can be woken up mid-sleep if
             * a task descriptor is added to the task
             * list
             */
        } else {
            sleep until the absolute spawn time
                of the first task descriptor in
                the task list;

            extract the first task descriptor in
                the task list;

            create a task from that task descriptor;

            update the absolute spawn time of that
                task descriptor to be the next
                spawn time;

            reinsert that task descriptor into
                the task list;
        }
    }
}
```

5.3 Priority Ceiling Protocol

Chapter 6

Experiment

Chapter 7

Results

Chapter 8

Conclusion

Bibliography

Witchel, E. (2009) *CS372 Operating Systems*. Retrieved from http://www.cs.utexas.edu/users/witchel/372/lectures/01.OSHistory.pdf on 2014/03/10.

Labrosse, J. J. (2010) *μC/OS-III: The Real-Time Kernel*. Retrieved from http://micrium.com/books/ucosiii/ti-lm3s9b92/.

Liu, C. L. and Layland, J. W. (1973) *Scheduling Algorithm for multiprogramming in a Hard-Real-Time Environment*. Journal of ACM, 1973, vol 20, no 1, pp. 46-61.

Lehoczky, J.; Sha, L. and Ding, Y. (1989) *The rate monotonic scheduling algorithm: exact characterization and average case behavior*. IEEE Real-Time Systems Symposium, pp. 166-171.

Dijkstra, E. W. (1965) *Solution of a Problem in Concurrent Programming Control*. Communications of the ACM, Volume 8, Issue 9, p. 569.

Reeves, G. E. (1997) *What really happened on Mars?*. http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html, accessed on 2014/03/17.

Lui S.; Rajkumar, R.; Lehoczky, J.P. (1990) *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*. IEEE Transactions on Computer, Volume 39, Issue 9, pp. 1175-1185.

Coffman, E.G. Jr.; Elphick, M.; Shoshani, A. (1971) *System Deadlocks*. Computing Surveys, 2 (1971), pp. 67-78.

Cheng, A.M.K.; and Ras, J. (2007) *The Implementation of the Priority Ceiling Protocol in Ada-2005*. Ada Letters, Volume XXVII, Number 1, April 2007.

Burns, A.; Wellings, A.J.; Zhang, F. (2009) *Combining EDF and FP Scheduling: Analysis and Implementation in Ada 2005*.