CSC 714

REAL TIME COMPUTER SYSTEMS

FINAL PROJECT REPORT

IMPLEMENTATION OF EDF, PCEP AND PIP IN RT-LINUX

Under the guidance of Dr. Mueller

Amit Choudhary (achoudh4@unity.ncsu.edu)
Nitin Shrivastav (nshriva@unity.ncsu.edu)
Ramnath Venugopalan (rvenugo@unity.ncsu.edu)

12/02/02

Table of contents:

INTRODUCTION

Problem Description:

Our objective is to add support for **Earliest Deadline First**, **Priority Inheritance Protocol** and **Priority Ceiling Emulation Protocol** in RT-Linux. The default scheduler in RT-Linux is a fixed-priority scheduler. There is no deadline monitoring or resource management (e.g. support for priority inheritance) functionality with the distribution we obtained from *ftp.fsmlabs.com*.

Solved issues:

Verified and implemented EDF, PIP, PCEP. The Priority inheritance protocol and Priority ceiling emulation protocol are used with fixed-priority scheduling mechanism.

Task division:

All - Verification & implementation of EDF, PIP, PCEP

Test Cases: Amit choudhary

Final report: Nitin Shrivastav

BACKGROUND [8]

Earliest deadline first algorithm:

The earliest deadline first algorithm is an example of dynamic priority scheduling. At every scheduling point, the algorithm finds the job with the earliest deadline and schedules this task. EDF is dynamic, as different jobs will have different priorities at different point in time.

Priority Inheritance Protocol:

PIP is a resource access control protocol used for allocating resources that control priority inversions in Real-time systems. Priority inversion occurs when a higher priority job is blocked on a resource acquired by a lower priority job. This scenario is not ideal for Real-time systems where such inversions may cause some jobs to miss their deadlines. Definition of PIP is given below [Mueller, class slides]

PIP Definition

Each job J has an **assigned priority** (e.g., RM priority) and a **current priority.**

1. Scheduling Rule: Ready jobs are scheduled on the processor preemptively in a priority-driven manner according to their current priorities. At its release time t, the current priority of every job is equal to its assigned priority. The job remains at this priority except under the condition stated in rule 3.

2. Allocation Rule: When a job J requests a resource R at time t,

- if R is free, R is allocated to J until J releases it, and
- if R is not free, the request is denied and J is blocked.

3. Priority-Inheritance Rule: When the requesting job J becomes blocked, the job k that blocks J inherits the current priority of J. The job k executes at its inherited priority until it releases R (or until it inherits an even higher priority); the priority of k returns to the priority it had when it

acquired the resource R.

PIP doesn't prevent deadlocks.

Priority ceiling emulation protocol:

PCEP is a simplification of Priority ceiling protocol. The definition of PCEP is given below:

PCEP definition:

Each job has an assigned priority and a current priority. System ceiling variable is global to the task set. Each lock has a ceiling priority that denotes the priority of the maximum priority job that will be acquiring it.

1. Scheduling Rule:

- At its release time t, the current priority p(t) of every job J equals its assigned priority. The job remains at this priority except under the conditions of rule 2.
- Every ready job J is scheduled preemptively and in a priority-driven manner at its current priority p(t).

2. Allocation Rule: Whenever a job J requests a resource R at time t, one of the following two conditions occur:

- R is held by another job. J's request fails and J becomes blocked.
- R is free.
    - If J's priority is higher than the current system ceiling, R is allocated to J, J is assigned the ceiling priority of mutex and System ceiling is assigned J's updated priority.
    - If J's priority is not higher than the system ceiling, R is allocated to J only if the system ceiling is due to J's priority; otherwise, J's request is denied and J becomes blocked.

RT-Linux Basics:

"RTLinux was developed at the department of computer science at the New Mexico Institute of Technology, Socorro NM, by Victor Yodaiken and Michael Barabanov.

RTLinux (***R**eal-**T**ime **L**inux*) is an extension to the Linux operating system designed to handle time-critical tasks. In essence RTLinux is a small real-time executive that runs the Linux OS as just another task (as its lowest priority task).

Linux is made completely preemptable and can, if necessary, be disabled entirely in order to dedicate the CPU to the more important real-time tasks. This approach makes it possible to make use of all the facilities of standard Linux, such as X-window, networking, development environment etc.

Design & Implementation

Most operating systems, including Linux, disable interrupts in critical sections of the kernel to, amongst other things, achieve synchronization. This means that the kernel is non-preemptive and tasks running in kernel mode cannot be forced to give up the processor to another task, no matter the priority. To be able to guarantee the temporal behavior of real-time tasks, the designers of RTLinux simulate this interrupt disable/enable feature and are thereby fooling the Linux kernel

into becoming completely preemptable.

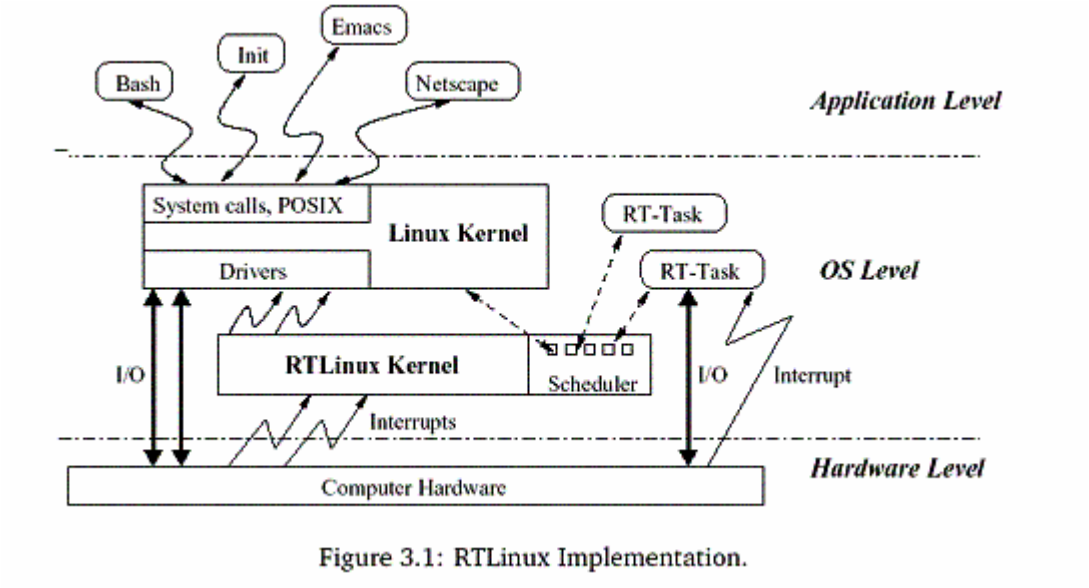


Figure 3.1: RTLinux Implementation.

Since the standard Linux kernel runs as the lowest priority task it can freeze up for long periods of time. This is because the CPU is busy handling the real-time kernel tasks. All interrupts are caught by the real-time kernel and passed on to the Linux kernel when no real-time tasks are scheduled to execute. Therefore the interrupts are always available to the real-time kernel while still allowing Linux to disable them as needed. However, the real-time kernel in itself is not preemptable, but its routines are supposedly small and fast enough to not cause any significant delays in themselves. The default scheduler supplied with RTLinux is a preemptive fixed priority scheduler, for previous versions of RTLinux there exist other scheduling modules, such as Earliest Deadline First and Rate Monotonic. This modularity makes RTLinux flexible enough to support both the time and event triggered paradigms.

Characteristics & Features

- Support for POSIX 1003.13 "single process/minimal real-time system" profile.
- Dynamic exchange of scheduler at runtime.
- IPC, through Shared memory.
- IPC, through dedicated Linux devices, RT-FIFOs.
- Improved timer resolution.
- Full support for SMP based systems.

Real-Time tasks cannot communicate directly with the Linux processes or make use of the Linux kernel services, since this would impose unbounded timing constraints, instead RTLinux provides functions for both shared memory and FIFO queues as a means of communication and synchronization. The FIFO queues are implemented as standard unidirectional Linux devices accessible by any Linux process. The real-time tasks, as well as all kernel-contained data, are never swapped to disk, which eliminates the problem of unbounded response times. This is provided in RTLinux through the reprogramming of the separate timer chip4, which exist in some form in all Intel x86-based computers. This provides both periodic as well as one-shot timers." [9]

EDF IMPLEMENTATION

In RT-Linux, we can add new scheduler modules to implement different scheduling policies. We added a new scheduler module to implement edf algorithm. This module needs to be loaded at run-time instead of the default fixed-priority scheduler. In this section, we first discuss the structure of existing fixed priority scheduler module in RT-Linux, then we discuss the new data structures added to implement the EDF functionality, and finally the pseudo code for implementing the EDF functionality.

Existing fixed-priority scheduler in RT-Linux:

The scheduler functionality is implemented in "rtl_schedule" function in rt-linux/schedulers/rtl_sched.c. The pseudo code for this function is described below:

```
/* Pseudo code for rtl_schedule function */

int rtl_schedule(void)
{
        Disable the interrupts

        for (all the tasks in the task-list) /* Task list is a linked list containing information about
                                                every task*/
        {
                if (task is due to be released for next period)  /* RTL_THREAD_TIMERARMED
                                                                    flag is used to check this */
                {
                        if (current-time > resume time for this task)  /* Resume time for the task
                                                                          denotes the next time at which
                                                                          this task is to be released */
                        {
                                Mark the task as ready for execution /* RTL_SIGNAL_TIMER &
                                        RTL_SIGNAL_READY are used to indicate this */

                                Mark the task as released /* RTL_THREAD_TIMERARMED flag
                                                                                    */
                                Update the next resume time if the task is periodic
                        }
                }

                If this task can be executed (i.e. is not suspended) and has highest priority
                among all the tasks seen so far, then store the task-id in 'next-task' variable.
        }

        if (clock-mode = ONE-SHOT) then call function find_preemptor to find the next
        scheduling point

        set the clock interrupt at next scheduling point (for PERIODIC clock-mode, the next
        scheduling point is after a fixed amount of time)

        If (no task has been selected in the above for loop), then select Linux task to run.

        Do the context switch.
```

```
        enable the interrupts.

}

/* Pseudo code for find_preemptor function for ONE-SHOT clock-mode */
/* This function finds the next preemption point in ONE-SHOT clock mode */
task-list * find_preemptor (task chosen) /* 'chosen' task is the next task to be run */
{
        set preemptor to NULL;
        for (all the tasks in the task-list)
        {
                if (task is to be released in future) /* Checked through
                                        RTL_THREAD_TIMERARMED */
                {
                        if (task-> priority > chosen->priority)
                        {
                                if ((! preemptor) && (task->resume-time <
                                                                chosen->resume-time))
                                {
                                        set preemptor to task
                                }
                        }
                }
        }
        return preemptor;
}
```

EDF Implementation details:

Data-structures modified for supporting EDF functionality:

File - **include/rtl_sched.h**:

a. Added following structure for EDF parameters

```
struct EDF {
        hrtime_t orig_deadline;  /*The actual abs. deadline when there is no
                                   inheritance*/
        hrtime_t rl_deadline;    /*The relative deadline of the thread*/
        };
```

We are using deadline as priority, the earlier the deadline, the higher the priority. There is no priority field as such.

b. Added a pointer to EDF structure defined above in rtl_thread_struct structure. This structure stores all the thread states.

We made a copy of the existing rtl_sched.c file which implements scheduler functionality. Our EDF implementation monitors deadline misses and simply reports it to user (prints on the screen). Also, now the developers need to specify the relative deadlines for each task. We have added a new API new_pthread_make_periodic which allows users to specify deadlines along with other parameters. In all our implementations , we have added our own APIs when needed instead of modifying any existing APIs.

To implement EDF, we made changes in some functions. The new pseudo code for these functions

are listed below:

a. rtl_schedule( ): As discussed above, this is the main function implementing scheduling. So, major changes are done in this function. The new pseudo code along with our changes is listed below:

```
/* Pseudo code for rtl_schedule function */

int rtl_schedule(void)
{
        Disable the interrupts

        for (all the tasks in the task-list) /* Task list is a linked list containing information about
                                                every task*/
        {
                /* New changes */
                if (task is periodic && task has already been released for this period && current-
                time > deadline of the task)
                        log_the_event (task has missed its deadline);
                /* End new changes */
                if (task is due to be released for next period)  /* RTL_THREAD_TIMERARMED
                                                                    flag is used to check this */
                {
                        if (current-time > resume time for this task)  /* Resume time for the task
                                denotes the next time at which this task is to be released */
                        {
                                Mark the task as ready for execution /* RTL_SIGNAL_TIMER &
                                        RTL_SIGNAL_READY are used to indicate this */
                                Mark the task as released     /* RTL_THREAD_TIMERARMED
                                                                 flag */
                                Update the next resume time if the task is periodic
                                /* New changes */
                                Set the deadline for this task if the task is periodic
                                /* End changes */
                        }
                }
                /* New changes */
                If this task can be executed (i.e. is not suspended) and has earliest deadline
                among all the task seen so far, then store the task-id in 'next-task' variable.
                else if this task has same deadline as the earliest deadline seen so far, then
                select the task which was released earlier
                /* End changes */
        }

        if (clock-mode = ONE-SHOT) then call function find_preemptor to find the next
        scheduling point

        set the clock interrupt at next scheduling point (for PERIODIC clock-mode, the next
        scheduling point is after a fixed amount of time)

        If (no task has been selected in the above for loop), then select Linux task to run.

        Do the context switch.

        enable the interrupts
```

```
}
```

b. find_preemptor( ): This function finds the next preemption point in ONE-SHOT clock mode.

```
/* Pseudo code for find_preemptor function for ONE-SHOT clock-mode */

task-list * find_preemptor( task chosen) /* 'chosen' task is the next task to be run */
{
        set preemptor to NULL;
        for (all the tasks in the task-list)
        {
                if (task is to be released in future) /* Checked through
                                RTL_THREAD_TIMERARMED */
                {
                        /* New changes */
                        if (task-> resume-time < chosen->deadline)
                        {
                                if ((! preemptor) &&
                                        (task->resume-time < preemptor->resume-time))
                                {
                                        set preemptor to task
                                }
                                else if (preemptor
                                         && task->resume-time == preemptor->resume-time)
                                {
                                        select the task which was released earlier
                                }
                        }
                        /* End changes */
                }
        }
        return preemptor;
}
```

c. new_pthread_make_periodic_np(): New API to support specification of relative deadlines from application programs. This function is same as pthread_make_periodic_np function of rt-linux. We have added code to initialize EDF structure in the new function. The pseudo code for the API is listed below.

```
int new_pthread_make_periodic_np(thread, start-time, period, relative-deadline)
{
        disable the interrupts;
        update period in the task structure;
        set resume-time to start-time
        /* New changes */
        set relative-deadline in task structure
        set the absolute deadline for the first period /* All the deadlines for later jobs of this task
        will be set in rtl_schedule( ) function */
        /* End changes */

        call rtl_reschedule_threads
        enable interrupts
}
```

d. pthread_create(): Added code for allocating memory for EDF structure. This is a long function and so we only show our changes made in the function.

```
{

        Allocate memory for EDF structure;
        set resume-time to 0;
        set deadline to 0;
        set priority to 0;

}
```

e. init_module(): Linux thread is handled separately by rt-linux. Its memory is assigned in init_module() of rtl_sched_edf.c. So, added code to allocate memory for EDF structure, and made the deadline of this thread as HRINFINITY. This will ensure that Linux thread will have lowest priority as we are using deadline field as the effective priority for scheduling.

Verification: We have verified our EDF functionality. Two sample test cases along with their outputs are in Appendix.

PCEP IMPLEMENTATION

We decided to implement this resource access protocol on mutexes provided in RT-Linux. Semaphores were not considered as they can have multiple instances. There are 2 types of mutexes currently supported in RT-Linux - PTHREAD_MUTEX_SPINLOCK_NP & PTHREAD_MUTEX_NORMAL. We have added a new mutex-type PTHREAD_MUTEX_PIP_PCEP to implement PCEP and PIP policies. The new type is similar to PTHREAD_MUTEX_NORMAL mutex type (blocking). "ifdef" is used to distinguish between the two resource control policies namely PIP & PCEP. So, user needs to run the "make config" program provided with RT-Linux, and select the resource policy(PIP or PCEP) to be used.

Data structures added to support PCEP functionality:

1. **include/rtl_mutex.h**:

Modifications:

a. Added a new pointer field in the pthread_mutex_t structure to point to the thread which has locked this mutex.

```
#ifdef _RTL_POSIX_THREAD_PRIO_PROTECT
struct rtl_thread_struct* owner_thread;
#endif
```

b. Added a new structure for implementing a linked list of all the mutexes in the system. This list is used to update
the system ceiling and priorities of threads on locking/unlocking of mutexes. MAXPRIO is the initial system ceiling which is lower than the priority of lowest thread.

```
#ifdef _RTL_POSIX_THREAD_PRIO_PROTECT
typedef struct mutex_list* list_ptr;
struct mutex_list {
        pthread_mutex_t * mutex_element;
        list_ptr next;
};
#define MAXPRIO 0
#endif
```

c. A mutex of type PTHREAD_MUTEX_NORMAL is used to synchronize access to the global data-structure (the list of mutexes in 1b & system ceiling structure in 2a

2. **include/rtl_sched.h**

Modifications:

a. Added a structure for maintaining system ceiling and the thread id because of which the system ceiling
is at the present level.

```
struct ceiling_t {
        int sys_ceiling; /*The system ceiling for PCEP*/
        struct rtl_thread_struct* ceiling_thread;
};
```

b. Added a new field called base_sched_priority in the structure rtl_thread_struct for storing the original priorities the thread priority will change as and when it locks/unlocks a mutex.

c. Added a line in the function pthread_setschedparam() to initialize the base_sched_priority of a thread.

Implementation details:

The developer needs to specify the ceilings of every mutex using an API. In the init_module if the application program, pcep_init( ) needs to be called which creates the global mutex to synchronize access to global data-structures.

The mutex functionality is supported in RT-Linux in rt-linux/schedulers/rtl_mutex.c file. The implementation details are examined function by function

a. pthread_mutex_create( ): The pseudo code for this function is shown along with our changes:

```
int pthread_mutex_lock(pthread_mutex_t * mutex)
{
        switch (mutex->type) {

        Case (PTHREAD_MUTEX_SPINLOCK_NP):
                .....
                No changes
                .....
               break;

        Case (PTHREAD_MUTEX_NORMAL):
                .....
                No changes
                .....
                break;

        /* New changes */
        Case (PTHREAD_MUTEX_PIP_PCEP):

                if (!mutex-> valid) return EINVAL; /* Indicates that mutex has been deleted */

                rtl_spin_lock_irqsave(& mutex->lock, flags) ; /* standard function */

                while ((ret = __pip_pcep_trylock(mutex)) {

                if (ret is EINVAL) then (rtl_spin_lock_irqrestore, return Error);

                call rtl_wait_sleep to insert to insert this thread in the wait-queue of the mutex.

                rtl_spin_lock(& mutex->lock)
                }

                rtl_spin_unlock_irqrestore(&mutex->lock, flags);
                return 0;

        /* End changes */
}
```

b. __pip_pcep_trylock( ): This is the function in which major changes have been done for implementing PCEP. The pseudo code is shown below.

```
int __pip_pcep_trylock(pthread_mutex_t * mutex)
{
        #ifdef _RTL_POSIX_PRIO_PROTECT
                self = pthread_self( ); /* store thread-id of thread trying to lock the mutex */

                pthread_mutex_lock(& global_mutex); /* Lock the global mutex */

                if ( self-> priority is less than current system ceiling)
                {
                        unlock the global mutex;
                        return EBUSY; /* EBUSY is 1, so in the pthread_mutex_lock function, the
                                calling thread will be put in the wait queue of this mutex */
                }
                else if (self->priority == current system ceiling)
                {
                        if (system->ceiling_thread != self) /* The system ceiling was not raised
                                                              because of this thread */
                        {
                                unlock the global mutex;
                                return EBUSY;
                        }
                }

                if (test_and_set_bit(0, &mutex->busy))  /* Mutex is already locked by some other
                                                          thread */
                {

                        unlock the global mutex;
                        return EBUSY:
                }

                if (self-> priority < mutex->ceiling_priority) self->priority = mutex->ceiling_priority;

                if (mutex->ceiling_priority is greater than current system ceiling)
                {
                        set system ceiling to mutex->ceiling_priority;
                        store id of the current thread in system_thread which indicates that this
                        thread has caused the current ceiling to be raised
                }

                update mutex->owner as self;
                unlock global mutex;
        #endif _RTL_POSIX_PRIO_PROTECT
}
```

c. int pthread_mutex_trylock(pthread_mutex_t * mutex)

```
{
Case (PTHREAD_MUTEX_SPINLOCK_NP):
                .....
                No changes
```

```
                .....
            break;

        Case (PTHREAD_MUTEX_NORMAL):
                .....
                No changes
                .....
                break;

        /* New changes */
        Case (PTHREAD_MUTEX_PIP_PCEP):

                if (!mutex-> valid) return EINVAL; /* Indicates that mutex has been deleted */

                rtl_spin_lock_irqsave(& mutex->lock, flags) ; /* standard function */

                ret = __pip_pcep_trylock(mutex);

                if (ret is EINVAL) then (rtl_spin_lock_irqrestore, return Error);

                rtl_spin_lock_irqrestore(&mutex->lock, flags);

                }

                return 0;

        /* End changes */
}
```

d. pthread_mutex_unlock(): There are lots of changes in this function to implement PCEP functionality. The functionality to implement the PCEP can be expressed in following statements:

- Iterate through the linked list of mutexes to find the next highest priority locked mutex. The ceiling of this mutex will be the new system ceiling.
- Wake-up all the threads that were blocked as they were not eligible due to their ceiling priorities.
- Iterate through the linked list of mutexes to find the next highest priority mutex locked by the current thread. The priority ceiling of this thread will be the new priority of the thread if it is greater than sched_base_priority.If no such mutexes exist, then thread will return back to its base sched priority.

```
int pthread_mutex_unlock(pthread_mutex_t * mutex)
{
        Case (PTHREAD_MUTEX_SPINLOCK_NP):
                .....
                No changes
                .....
            break;

        Case (PTHREAD_MUTEX_NORMAL):
                .....
                No changes
```

```
        .....
        break;

/* New changes */
Case (PTHREAD_MUTEX_PIP_PCEP):

        #ifdef _RTL_POSIX_PRIO_PROTECT

        pthread_mutex_t * highest_prio_mutex = NULL;
        int move_prio = 0;

        set
        if (!mutex-> valid) return EINVAL; /* Indicates that mutex has been deleted */

        rtl_spin_lock_irqsave(& mutex->lock, flags) ; /* standard function */

        set mutex->owner_thread to NULL;

        for (all the mutexes in the global mutex-list)
        {
                let 'next' be the next mutex in the global list;

        /* The code below implements the following functionality  */
        /* Iterate through the linked list of mutexes to find the next highest priority
        locked mutex.The ceiling of this mutex will be the new system ceiling.*/

        if ((next->owner_thread) &&  /* The next mutex is locked by some thread */
          (!highest_prio_mutex ||
                 highest_prio_mutex->ceiling_priority < next->ceiling_priority)
                        {
                                highest_prio_mutex = next;
                        }

                /* The code below implements the following functionality  */
                /* Wake-up all the threads that were blocked as they  were not eligible
                due to their ceiling priorities.*/

                if (!next->owner_thread)
                        rtl_wait_wakeup(&next);  /* wakeup all the tasks that were
                blocked because their ceiling priority was less than the system ceiling */

/* The code below implements the following functionality */
/*  Iterate through the linked list of mutexes to find the next highest priority mutex locked
by the current thread. The priority ceiling of this thread will be the new priority of the
thread if it is greater than sched_base_priority. If no such mutexes exist, then thread will
return back to its base sched priority.*/



        if ((next->owner_thread == pthread_self( ) ) && (next->ceiling_prio > move_prio))
                        move_prio = next->ceiling_prio;

        }

        update the system ceiling;
```

```
                update the threads priority; /* Revert it back to its base priority if no other mutex
                is locked */

                unlock the global_mutex;

                /* End changes */
                unlock the mutex;

                return 0;
}
```

e. pthread_mutex_init( ): The changes in this function are listed in following pseudo-code:

```
void pthread_mutex_init( ) {
        .....
        No changes
        ....
        lock the global mutex;
        Add the mutex to global list of mutex;
        unlock the global mutex;
}
```

f. pthread_mutex_destroy( ) : The changes in this function are listed in following pseudo-code:

```
void pthread_mutex_destroy( ) {
        .....
        No changes
        ....
        lock the global mutex
        Remove the mutex from global list of mutex
        unlock the mutex
}
```

Verification: We have verified our PCEP functionality. A sample test case along with its output is in Appendix.

PIP IMPLEMENTATION

We decided to implement this resource access protocol on mutexes provided in RT-Linux. Semaphores were not considered as they can have multiple instances. There are 2 types of mutexes currently supported in RT-Linux - PTHREAD_MUTEX_SPINLOCK_NP & PTHREAD_MUTEX_NORMAL. We have added a new mutex-type PTHREAD_MUTEX_PIP_PCEP to implement PCEP and PIP policies. The new type is similar to PTHREAD_MUTEX_NORMAL mutex type (blocking). "ifdef" is used to distinguish between the two resource control policies namely PIP & PCEP. So, user needs to run the "make config" program provided with RT-Linux, and select the resource policy(PIP or PCEP) to be used.

We implement the WAIT-FOR graph using adjacency matrix representation. To detect deadlock in the graph , we use topological sorting algorithm to detect cycles in the graph. Whenever a new thread or mutex is created , it is assigned a vertex number in the graph. This is implemented by calling a new API 'pip_register'. The graph edges are properly updated as thread acquires or blocks on a mutex or releases a mutex.

The deadlock-detection algorithm is run when a thread tries to lock a mutex. No allocation is done if this allocation may result in deadlock. An error value, EDDLK is returned to application program indicating that deadlock might occur if it acquires this mutex. It is left to the application to handle the deadlocks.

Data structures modified:

1. **include/rtl_mutex.h**

Modifications:

- Added following two fields in pthread_mutex_t structure

```
#ifdef _RTL_PRIO_PIP
        int vnum; /* The vertex number of this mutex in the wait-
                        for graph */
        struct rtl_thread_struct* owner_thread; /* The thread
                        which locks this mutex */
#endif
```

-
```
#ifdef _RTL_PRIO_PIP
#define EDDLK 77


/* Declaration for adjacency matrix. This is used to represent the wait-for graph */
typedef struct adjacency_matrix * matrix;

struct adjacency_matrix {
        int     matrix_element;
};

int * indegree; /* Array to store indegrees of each vertex */
```

- A mutex of type PTHREAD_MUTEX_NORMAL to synchronize access to adjacency matrix

2. **include/rtl_sched.h**

Modifications:

- Added a new field called base_sched_priority in the structure rtl_thread_struct for

storing the original priority as the thread priority will change as and when it locks/unlocks a mutex.

- Added a line in the function pthread_setschedparam()to initialize the base_sched_priority of a thread.

Implementation details:

We discus the implementation details function by function. The application will call API pip_init to initialize the data-structures for PIP. The pseudo-code is shown below:

a. void pip_init( )

```
{
        Allocate memory for adjacent matrix;
        Create global-mutex;
}
```

Then, when each mutex or thread is created, function pip_register is called to add this to wait-for graph.

b. void pip_register( )

```
{
        lock the global-mutex;
        allocate a vertex number to this mutex;
        unlock the mutex;
}
```

c. pthread_mutex_lock(): This is same as pthread_mutex_lock for PCEP as we don't distinguish between PIP and PCEP in this function. The distinction is made in pip_pcep_trylock( ) based on #ifdef. For PIP #ifdef _RTL_PRIO_PIP is used.

d. pcep_pip_trylock( ): This function implements most of the functionality of PIP. The pseudo-code is explained in detail below:

```
int pcep_pip_trylock(pthread_mutex_t * mutex)
{
        #ifdef _RTL_PRIO_PIP


        if (test_and_set(&mutex->busy,0)
        { /* Mutex is locked by someone else */

                Add_edge(pthread_self( ), mutex); /* Add the request edge in the
                                                        wait-for graph */

                status = check_deadlock( ); /* Check if deadlock can occur if we block for this
                                                mutex*/

                if (status ==  DEADLOCK)
                {
                        Remove_edge(pthread_self( ), mutex);
                        unlock global_mutex;
                        return EDDL;
```

```
                }
                else
                {
                        inherit_priority(pthread_self(), mutex); /* This function will increase the
                        priority of the owner thread of 'mutex' to the thread which is blocking on
                        this mutex */

                        unlock global_mutex;
                        return EBUSY;
                }
        }
        else
        {
                Add_edge(mutex, pthread_self());

                status =  check_deadlock( );

                if (staus == DEADLOCK)
                {
                        Remove_edge(mutex,pthread_self());
                        clear_bit(0, &mutex->busy);
                        unlock global_mutex;
                        return EDDL;
                }

        }

        update the owner thread of this mutex;
}
```

e. void inherit_priority(thread t, mutex m)

```
        {
                Find the thread which has acquired mutex m.Let p denote this thread;

                if (p ->priority < t->priority) p->priority = t->priority;
        }
```

f. pthread_mutex_unlock( ): This function also has many changes for PIP implementation. The detailed pseudo-code is given below.

```
int pthread_mutex_unlock(pthread_mutex_t * mutex)
{

switch(mutex->type):

        Case (PTHREAD_MUTEX_SPINLOCK_NP):
                .........
                No changes
                .........
               break;

        Case (PTHREAD_MUTEX_NORMAL):
                .........
                No changes
                ........
```

```
                break;

        /* New changes */
        Case (PTHREAD_MUTEX_PIP_PCEP):

                if (!mutex-> valid) return EINVAL; /* Indicates that mutex has been deleted */

                rtl_spin_lock_irqsave(& mutex->lock, flags) ; /* standard function */

                lock the global mutex

                Remove_edge(mutex, pthread_self());

                for (all resources locked by this thread)
                {
                        find the highest priority thread blocking on this resource; Call this thread t
                        if (t->priority > pthread_self()->priority) inherit_priority( );
                }

                if (t == NULL) then move
                        pthread_self->priority to pthread_self->base_sched_priority;
                unlock the global mutex;

                unlock the mutex;

        /* End changes */
}
```

g. pthread_mutex_trylock( ): This is same as for PCEP.

h. check_deadlock( ): This function runs topological sorting algorithm to detect cycles in the Adjacency matrix.

```
int check_deadlock( )
{

        for (all the vertexes in the vertex list)
        {
                v = find_new_vertex_indegree_zero( );
                if (v == -1)
                        return EDDLK;

                for (all vertex w adjacent to v)
                        indegree[w]--;
        }
        return NODDLK;
}
```

Verification:

PIP functionality has been verified. Two test cases are listed in Appendix. One test case is for testing deadlock condition and other test case is a complex one with multiple priority inheritance.

OTHER ISSUES

These were some of the issues we faced during the implementation:

- Installation issues: The 'gcc' version was causing trouble in installation. "kgcc" is required for RT-Linux 3.1
- De-bugging issues: The debugging was not easy as we were not able to use GDB. So, whenever there was some bug in the code, kernel crashed and required a re-boot.

REFERENCES

1. Official webpage for RT-Linux http://www.fsmlabs.com

2. Getting started with RT-Linux http://www.fsmlabs.com/developers/docs/pdf/GettingStarted.pdf

3. RT-Linux installation http://fsmlabs.com/developers/docs/pdf/Installation.pdf

4. RT-Linux tutorial
http://216.239.39.120/translate_c?hl=en&u=http://www.linuxfocus.org/English/May1998/article44.html

5. Man pages http://www.fsmlabs.com/developers/man_pages

6. Two approaches to RT-Linux http://www.linuxdevices.com/articles/AT7005360270.html

7. Some white papers http://www.fsmlabs.com/developers/white_papers

8. Course-notes, CSC 714, Real-time computer systems, http://courses.ncsu.edu/csc714

9. Modular scheduling in RT-Linux http://www.vmlinux.org/rtl/

APPENDIX

In all the test cases described below, the words “task” and “thread” mean the same thing. All our resources are mutexes.

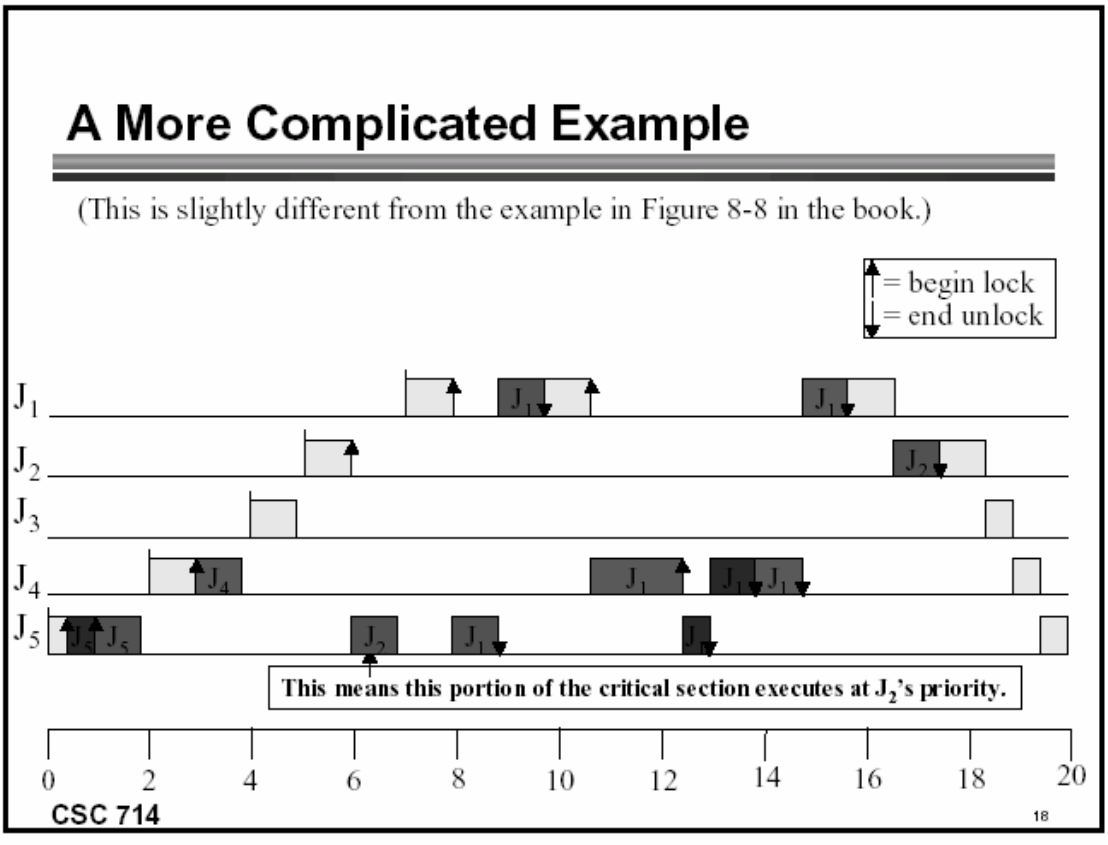
Test case for PIP:

There are two test cases. The first one had been discussed by Dr. Mueller in class and the second is a simple test case showing deadlock avoidance.

Test Case 1

The figure of this test case as discussed by Dr. Mueller is listed below and our output matches the example. The priorities are listed below. Priority 5 has highest priority.

Task No.	Priority
1	5
2	4
3	3
4	2
5	1



We will denote J5 as task 5 and so on.
Initially task 5 runs and acquires resources (mutexes) 1 and 2. Now, task 4 runs and acquires mutex 3. Task 2 and 1 block on mutex 2. Task 5 runs at Task 2's priority and then at Task 1's priority and so on. The priority change is reflected in the output.

The output is listed below.

RTLinux PIP test starts on CPU0
Thread 5 running at priority 1.
Thread 5 abt to lock mutex 1 ...
Thread 5 mutex1 lock returned 0
Thread 5 abt to lock mutex 2 ...
Thread 5 mutex2 lock returned 0
Thread 5 is about to sleep for 500000 ns
Thread 4 running at priority 2.
Thread 4 abt to lock mutex 3 ...
Thread 4 mutex3 lock returned 0
Thread 4 is about to sleep for 500000 ns
Thread 3 running at priority 3.
Thread 3 is about to sleep for 500000 ns
Thread 2 running at priority 4.
Thread 2 abt to lock mutex 2 ...
Thread 1 running at priority 5.
Thread 1 abt to lock mutex2...
Thread 5 running at priority 5.
Thread 5 abt to unlock mutex 2...
Thread 5 mutex2 unlock returned 0
Thread 1 mutex2 lock returned 0
Thread 1 abt to unlock mutex 2...
Thread 1 mutex2 unlock returned 0
Thread 1 abt to lock mutex3...
Thread 4 running at priority 5.
Thread 4 abt to lock mutex 1 ...
Thread 5 abt to unlock mutex 1...
Thread 5 mutex1 unlock returned 0
Thread 4 mutex1 lock returned 0
Thread 4 abt to unlock mutex 1...
Thread 4 mutex1 unlock returned 0
Thread 4 abt to unlock mutex 3...
Thread 4 mutex3 unlock returned 0
Thread 1 mutex3 lock returned 0
Thread 1 abt to unlock mutex 3...
Thread 1 mutex3 unlock returned 0
Thread 1 ends.
Thread 2 mutex2 lock returned 0
Thread 2 abt to unlock mutex 2...
Thread 2 mutex2 unlock returned 0
Thread 2 ends.
Thread 3 running
Thread 3 ends.
Thread 4 running at priority 2. (returned to base priority)
Thread 4 ends.
Thread 5 running at priority 1. (returned to base priority)
Thread 5 ends.

Test Case 2

It is a simple test case which exhibits deadlock avoidance. Task 1 acquires resource (mutex) 3. Task 2 acquires mutex 1. Task 3 acquires mutex 2. Now, task 2 blocks on mutex 2 and task 3 blocks on mutex 3. Now, task 1 wants to acquire mutex 1 and if it blocks on mutex 1, we have a deadlock. So, task 1 is not allowed to block on mutex 1.

The output is listed below.

RTLinux deadlock test starts on CPU0
Thread 1 abt to lock mutex3...
Thread 1 mutex3 lock returned 0
Thread 1 is about to sleep for 500000000 ns
Thread 2 abt to lock mutex 1...
Thread 2 mutex1 lock returned 0
Thread 2 is about to sleep for 5000000 ns
Thread 3 abt to lock mutex 2...
Thread 3 mutex2 lock returned 0
Thread 3 is about to sleep for 5000000 ns
Thread 2 abt to lock mutex 2 ...
Thread 3 abt to lock mutex 3 ...
Thread 1 abt to lock mutex1...
Thread 1 mutex1 lock returned 77
Locking/Blocking of mutex1 by Thread 1 will result in DEADLOCK
So, the mutex was not granted/not allowed to block.
Thread 1 abt to unlock mutex 3...
Thread 1 mutex3 unlock returned 0
Thread 1 ends.
Thread 3 mutex3 lock returned 0
Thread 3 abt to unlock mutex 2...
Thread 3 mutex2 unlock returned 0
Thread 3 abt to unlock mutex 3...
Thread 3 mutex3 unlock returned 0
Thread 3 ends.
Thread 2 mutex2 lock returned 0
Thread 2 abt to unlock mutex 1...
Thread 2 mutex1 unlock returned 0
Thread 2 abt to unlock mutex 2...
Thread 2 mutex2 unlock returned 0
Thread 2 ends.

Test case for EDF:

There are two test cases for EDF. The first task set is schedulable under edf. The second task set has utilization above 1 and hence there are deadline misses.

Test Case 1

Task set: All times are in nanoseconds. The reason for the phases being different is that RT-Linux doesn't execute a task in its first period. This is because as soon as you enter the while(1) loop, you do a pthread_wait_np(). This suspends the thread till its next release time. So, effectively even if all the tasks are released at the same time, it appears as though they were released at different times. That is, at the start of the second period.

Task 1: (500000000, 100000000, 500000000, 500000000)

Task 2: (800000000, 100000000, 800000000, 800000000)
Task 3: (1000000000, 100000000, 1000000000, 1000000000)

The output is listed below. 5 means that the Task with period 500000000 is running and so on.

```
5  8  5  10
5  8  5  10
8  5  5  10
8  5  5  8  10 /*End of a hyperperiod*/
5  8  5  10
5  8  5  10
8  5  5  10
8  5  5  8  10 /*End of a hyperperiod*/
5  8  5  10
5  8  5  10
8  5  5  10
8  5  5  8  10 /*End of a hyperperiod*/
5  8  5  10
5  8  5  10
8  5  5  10
8  5  5  8  10 /*End of a hyperperiod*/
```

Test Case 2

Task set: All times are in nanoseconds. Here we show deadline misses. Since there are three tasks with period 200000000 ns and execution time as 100000000 ns, the total utilization is more than 1. So, this task set is not schedulable and hence we see deadline misses.

Task 1: (200000000, 100000000, 200000000, 200000000)
Task 2: (200000000, 100000000, 200000000, 200000000)
Task 3: (200000000, 100000000, 200000000, 200000000)

The output is listed below. Here the tasks are identified by their task ids. We can see that while the Task c0ea8000 was running, it missed its deadline and the scheduler reported it.

```
Task c0d18000 running
Task c0d18000 finishes
Task c0a88000 running
Task c0a88000 finishes
Task c0ea8000 running
Task c0ea8000 missed its deadline
```

Test case for PCEP:

There are 4 tasks and 4 resources in this test case. The resources are mutexes. Priority 4 has highest priority.

Task No.	Priority
1	4
2	3
3	2
4	1

Mutex No.	Ceiling
1	4
2	4
3	3
4	2

First task 4 runs and acquires mutexes 4 and goes to sleep. The system ceiling and task 4's priority is raised to 2. Now, task 3 runs but it blocks on mutex 3 as its priority is equal to the system ceiling and it is not because of this task.
Next task 2 runs and acquires mutex 2 and goes to sleep. The system ceiling and task 2's priority is raised to 4. Now, task 1 runs but it blocks on mutex 1 as its priority is equal to the system ceiling and it is not because of this task.
Next task 2 runs and locks mutex 3 successfully. This is because mutex 3 is free and the system ceiling is 4 which is also equal to the inherited priority of task 2. And task 2 was the task which raised the system ceiling.

So, we see the proper PCEP functionality. The other parts of the output are not as significant as the initial part which has been explained above.

The whole output is listed below.

```
RTLinux PCEP test starts on CPU0
Thread 4 abt to lock mutex 4 ...
Thread 4 mutex4 lock returned 0
Thread 4 is about to sleep for 900000 ns
Thread 3 abt to lock mutex 3 ...
Thread 2 abt to lock mutex 2 ...
Thread 2 mutex2 lock returned 0
Thread 2 is about to sleep for 300000 ns
Thread 1 abt to lock mutex1...
Thread 2 abt to lock mutex 3 ...
Thread 2 mutex3 lock returned 0
Thread 2 is about to sleep for 300000 ns
Thread 2 abt to unlock mutex 3...
Thread 2 mutex3 unlock returned 0
Thread 2 abt to unlock mutex 2...
Thread 2 mutex2 unlock returned 0
Thread 2 ends.
Thread 1 mutex1 lock returned 0
Thread 1 abt to unlock mutex 1...
Thread 1 mutex1 unlock returned 0
Thread 1 abt to lock mutex2...
Thread 1 mutex2 lock returned 0
Thread 1 abt to unlock mutex 2...
Thread 1 mutex2 unlock returned 0
Thread 1 ends.
Thread 4 abt to unlock mutex 4...
Thread 4 mutex4 unlock returned 0
Thread 3 mutex3 lock returned 0
Thread 3 abt to unlock mutex 3...
Thread 3 mutex3 unlock returned 0
Thread 3 abt to lock mutex 4 ...
Thread 3 mutex4 lock returned 0
Thread 3 is about to sleep for 500000 ns
Thread 3 abt to unlock mutex 4...
```

```
Thread 3 mutex4 unlock returned 0
Thread 3 ends.
Thread 4 ends.
```