CSC 714
Real Time Computer Systems

Project Report

Implementation of EDF, PIP, PCEP in BrickOS

Sushil Pai
spai@ncsu.edu

Project URL: http://www4.ncsu.edu/~spai/csc714

OBJECTIVE: **3**

INTRODUCTION: **3**

IMPLEMENTATION **5**

BASIC IMPLEMENTATION: **5**
EARLIEST DEADLINE FIRST **6**
PSEUDO CODE 6
MOD LOG 6
TEST RESULTS 6
PRIORITY INHERITANCE PROTOCOL **9**
DEFINITION 9
PSEUDO CODE 9
MOD LOG 10
TEST RESULTS 10
PRIORITY CEILING EMULATION PROTOCOL **12**
DEFINITION 12
PSEUDO CODE 12

CHALLENGES FACED **14**

REFERENCES **14**

Objective:

Add support for EDF (Earliest Deadline First), PIP (Priority Inheritance Protocol) and PCEP (Priority Ceiling Emulation Protocol) in Brick OS. Currently Brick OS supports only static priority scheduling which does not perform any kind of deadline monitoring and resource management.

Introduction:

BrickOS (previously known as LegOS) is an open source embedded operating system, featuring preemptive multitasking, dynamic memory management and IR networking. It is designed to run on a Lego Mindstorm RCX brick based on the Hitachi H8/3292 microcontroller. It was started by Markus Noga in October 1998. The default scheduler that BrickOS supports is Static priority preemptive scheduling. This scheduling does not perform any kind of deadline monitoring and resource scheduling.

The **Earliest Deadline First (EDF)** algorithm is a dynamic priority-scheduling algorithm in which the priorities of individual jobs are based on their absolute deadlines. An EDF algorithm can generate a feasible schedule for a system of N independent, pre-emptable tasks as long as the total density of the system is less than 1. Hence EDF is an optimal scheduling algorithm.

In the **Priority Inheritance Protocol** (**PIP**), the resource holder inherits the priority of the highest priority blocked process. When a thread tries to lock a resource using this protocol and is blocked, the resource owner temporarily receives the blocked thread's priority, if that priority is higher than the owner's. It recovers its original priority when it unlocks the resource.

Priority Ceiling means that while a process owns the resource lock it runs at a priority higher than any other process that may acquire the resource. In the priority ceiling solution each shared resource is initialized to a priority ceiling. Whenever a process locks this resource, the priority of the process is raised to the priority ceiling. This works as long as the priority ceiling is greater than the priorities of any process that may lock the resource.

Implementing a scheduler based on EDF/PIP/PCEP in the scheduler would take care of the above mentioned problems (deadline monitoring and resource management).

LNP stands for **LegOS Network Protocol**. It allows for communication between brickOS powered RCX, and host computers. This is required for verification of the scheduler operation.

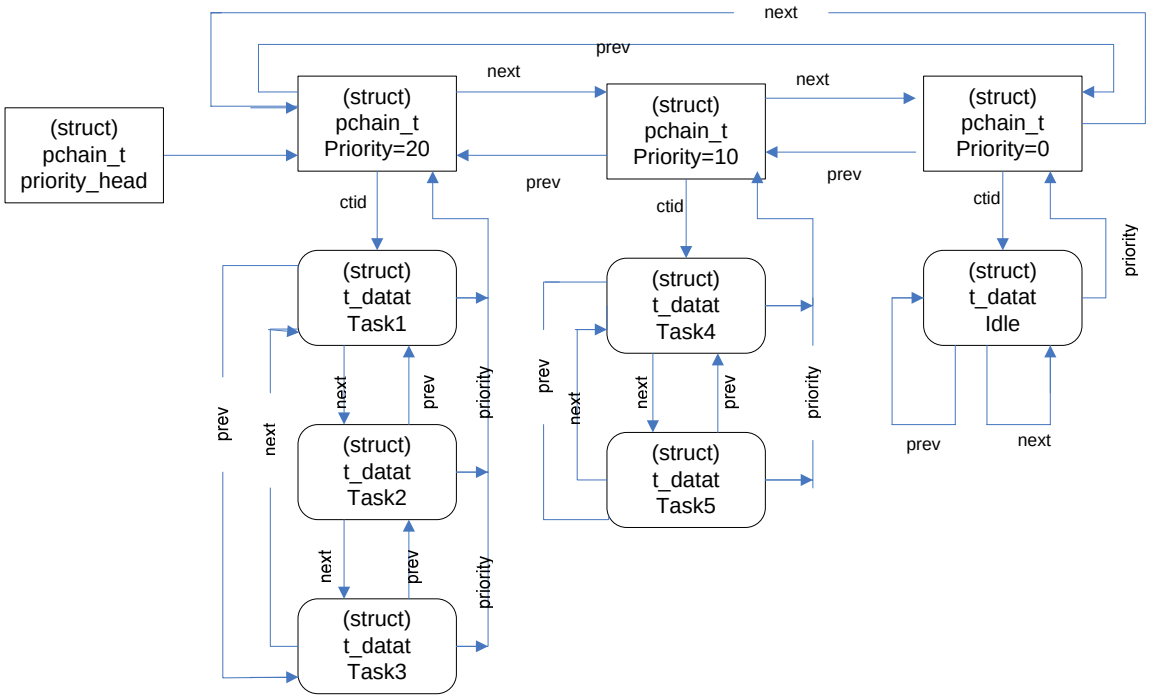

Figure: Implementation of tasks and priorities in BrickOS

Implementation

Basic Implementation:

1. Phase, period and deadlines: These are the 3 main characteristics of a task in a real time system. Due to the need for some information like priority ceiling of the task and the deadline of tasks to be present a priori, the system first needs to be initialized. Incase of EDF, the deadline of the task was provided to the scheduler as an input parameter in the execi() function. Implementing of phase and period had to be done using while loop and msleep()
2. Implementing resources: Resources were implemented using semaphores. Semaphores were added into the resource structure along with other information like priority ceiling of the resource and the link to the task that was currently holding the resource. A list of these resources has also got to be maintained using a linked list.
3. LNP for debugging: Inorder to debug the code and to verify the correct implementation of the code, LNP was used. The tasks would transmit some data to the PC over IR at specific locations during their execution to indicate the state of the system. No LNP related code was added within the code of the scheduler as it seemed to interfere with the work of the scheduler.
4. Content of the tasks: The tasks consist of multiple 'for' loops and LNP related data at the end of each loop to indicate the state of the task.
5. Priorities in EDF: It was assumed that the user would give the same priority to all the tasks that need to be run under EDF scheduling. Any task having higher/lower priority would run based on the default fixed priority scheduling.
6. Time: The get_system_up_time function was used to compute the time. The system up time when subtracted from the time the first task was released and then divided by 20 (20 milliseconds is the default size of each time slice) was provided in the output below as the timeslot.

Earliest Deadline First

Pseudo Code

Scheduling of a task:
Assign a task block to the new task using malloc function
Calculate absolute deadline. Absolute deadline= system up time of the RCX+relative deadline
Traverse through the priority chain till current priority = priority of new task
If absolute deadline of new task < absolute deadline of first task
    add the new task to the top of the priority queue
    update the priority block to point to the new task
If absolute deadline of new task < absolute deadline of last task
    add the new task to the bottom of the priority queue
Traverse through the tasks under the priority
    If absolute deadline of new task < absolute deadline of current task
        Break
add task above the current task

Deadline Monitoring:
(done at every timeslot)
get the system up time of the RCX
traverse through the list of tasks under the priority level
    If absolute deadline of the task < system up time
        kill the task using the kill function

Note: as the linked list used is a double linked list, the nearby nodes also need to be updated.

Mod Log

    execi() in kernel/tm.c
    tm_scheduler() in kernel/tm.c
    struct _tdata_t in include/tm.h

Test Results

1. System with 2 tasks

Task details:

Task	Phase	Period	Deadline	Priority
Task1	0	10s	5000 ms	1
Task2	100 ms	10s	1000 ms	1

Output:

TimeSlot Task Event

1679 Task0

Setting system time to 1679

13 Task1 Release

26 Task1 Start

39

Info: Deadline of Task1=39114

54 Task1 Stage1

31 Task2 Release

82 Task2 Start

94

Info: Deadline of Task2=36201

123 Task2 Stage1

150 Task2 Stage2

177 Task2 Over

266 Task1 Stage2

453 Task1 Over

581 Task1 Release

2. System with 3 tasks

Task details:

Task	Phase	Period	Deadline	Priority
Task1	0	20s	5s	1
Task2	100 ms	20s	1s	1
Task3	100ms	20s	4s	1

Output:

TimeSlot Task Event

---

271 Task0

Setting system time to 271

14 Task1 Release

27 Task1 Start

40

50 Task1 Stage1

32 Task2 Release

78 Task2 Start

90

Info: Deadline of Task2=7973

82 Task3 Release

119 Task3 Start

133

Info: Deadline of Task3=12298

120 Task2 Stage1
188 Task2 Stage2
227 Task2 Over
366 Task1 Stage2
376 Task3 Stage1
617 Task3 Stage2
703 Task1 Over
779 Task3 Over

Priority Inheritance Protocol

Definition

Each job Jk has an **assigned priority** (e.g., RM priority) and a **current priority** $\pi k(t)$.

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,
(a) if R is free, R is allocated to J until J releases it, and
(b) 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 J*l* that blocks J inherits the current priority of J. The job J*l* executes at its inherited priority until it releases R (or until it inherits an even higher priority); the priority of J*l* returns to its priority $\pi l(t')$ that it had at the time t' when it acquired the resource R.

Notice: If resources not properly nested, then need to recomputed prior on release if another resource is still held.

Pseudo Code

Initialization:
- create a pointer to access the first resource in a linked list of resource
- add each resource to the beginning of the resource queue
note: the position of the resource in the linked list does not matter. Adding to the beginning of the linked list is faster than adding it to the end.

Allocation & Priority Inheritance:

```
Check if the resource is free using semtrywait function
if the resource is free
        update the resource data structure
        grab the resource and return to execution
if the resource is not free
        if the task holding the resource is the only task under its priority
                delete the corresponding priority block
        else
                delink the holding task from its priority queue
        if the current task is the only task under its priority
                update the current task
        else
                update the current task and the last task under its priority
        add the holding task to the top of the priority chain of the current task
```

releasing of resource:

release the resource semaphore
decrease the priority of the task to its original priority

Deadline Monitoring:
traverse through the priority linked list
    traverse through the tasks under that priority
        if(tasknum of current task = tasknum of new task)
            kill(current task)

Mod Log

get_resource() in kernel/tm.c and include/sys/tm.h
release_resource() in kernel.tm.c and include/sys/tm.h
init_resource() in kernel.tm.c and include/sys/tm.h
execi() in kernel/tm.c
tm_scheduler() in kernel/tm.c
struct _tdata_t in include/tm.h
struct resource in include/tm.h

Test Results

1. System with 2 tasks

Task details:

Task	Phase	Period	Deadline	Priority
Task1	0	10s	10 s	1
Task2	100 ms	10s	10 s	6

Output:
TimeSlot Task Event

2374 Task0
Setting system time to 2374
13 Task1 Release
26 Task1 Start
39 Task1 Res2 held
53 Task1 Stage1
31 Task2 Release
80 Task2 Start
85 Task3 Release
99 Task2 Stage1
251 Task5
Info: New Priority=6
264 Task1 Res2 released
278 Task1
Info: Old Priority=1
264 Task2 Res2 held

313 Task2 Stage2
325 Task2 Res2 released
346 Task2 Over
105 Task3 Start
377 Task3 Stage1
396 Task3 Stage2
416 Task3 Over
506 Task1 Stage2
693 Task1 Over

2. System with 3 tasks

Task details:

Task	Phase	Period	Deadline	Priority
Task1	0	10s	10s	1
Task2	100 ms	10s	10s	5
Task3	100ms	10s	10s	3

Output:
TimeSlot Task Event
---
271 Task0
Setting system time to 271
14 Task1 Release
27 Task1 Start
40
50 Task1 Stage1
32 Task2 Release
78 Task2 Start
90
Info: Deadline of Task2=7973
82 Task3 Release
119 Task3 Start
133
Info: Deadline of Task3=12298
120 Task2 Stage1
188 Task2 Stage2
227 Task2 Over
366 Task1 Stage2
376 Task3 Stage1
617 Task3 Stage2
703 Task1 Over
779 Task3 Over

Priority Ceiling Emulation Protocol

Definition

1. Scheduling Rule: At all times, jobs are scheduled on the processor in priority-driven, preemptive manner according to their current priorities.

2. Allocation Rule: If a job requests a resource
    1. and the resource is free, it is allocated the resource and the current priority is raised to the ceiling of the resource.
    2. and the resource is busy, it is block until the resource becomes available and the job has the highest current priority.

Upon releasing the resource, the current priority is lowered to the maximum of the assigned priority and the priority ceiling of any resource being held.

Pseudo Code

Initialization:
- create a pointer to access the first resource in a linked list of resource
- add each resource to the beginning of the resource queue

Note: the position of the resource in the linked list does not matter. Adding to the beginning of the linked list is faster than adding it to the end.

Allocation & Priority Inheritance:

```
Check if the resource is free using semtrywait function
if the resource is free
        update the resource data structure
        if the task holding the resource is the only task under its priority
                delete the corresponding priority block
        else
                delink the holding task from its priority queue
        traverse through the priority linked list
                if priority of the linked list > ceiling priority of the resource
                        break
        if the current priority = ceiling priority of the resource
                if the current priority has only 1 task under it
                        update the first task
                else
                        update the first task and the last task under the priority
                add the holding task to the top of the priority chain of the current task
        else
                create a new priority level
                update the priority levels around it
                add the current task to the top of the new priority level

        add the holding task to the top of the priority chain of the current task
        grab the resource and return to execution
if the resource is not free
        put task in blocked state
```

yield the rest of the timeslice

Deadline Monitoring:
traverse through the priority linked list
    traverse through the tasks under that priority
        if(tasknum of current task = tasknum of new task)
            kill(current task)

releasing of resource:
release the resource semaphore
decrease the priority of the task to its original priority

Mod log

get_resource() in kernel/tm.c and include/sys/tm.h
release_resource() in kernel.tm.c and include/sys/tm.h
init_resource() in kernel.tm.c and include/sys/tm.h
execi() in kernel/tm.c
tm_scheduler() in kernel/tm.c
struct _tdata_t in include/tm.h
struct resource in include/tm.h

Note: This code is currently having some problems and hence is not provided here

Challenges faced

- Installing BrickOS: As i am more comfortable with the windows environment, I tried installing BrickOS on cygwin. I was able to compile my code there, but was not able to use firmdl3 and dll to upload the firmware image on the RCX. I then tried to do it using DJGPP. I again faced the same problem. I then installed ubuntu Linux on my desktop. The installation of BrickOS on Linux did not have any problems.
- Difficulties with LNP and pyLNP: My first attempt at the LNP based communication between the RCX and the PC using C did not work well, I explored the option of using pyLNP. I encountered some problems with importing the LNP related settings. I then realized that there was a c based program already present in the lnp installation, I used it and found it to work well.
- Need to reboot PC each time the program is loaded into the RCX: After using the LNP daemon, I am unable to load the firmware/program onto the RCX. Killing the daemon process does not resolve the issue.
- Loading of firmware stops at 98%: I have often encountered this problem that the loading of the firmware stops at 98 or 99%. I initially thought that the problem was with the size of the firmware and hence disabled a few of the functionalities on the RCX (dmotor, dsound). When this did not help, i tried making a few changes in my code. Surprising, things like changing 'for' loop into 'while' loop, and traversing a linked list based loop from the 2nd to the first element instead of 1st element to the last element, seemed to get rid of this problem.

References

- BrickOS Kernel Documentation
  http://legos.sourceforge.net/docs/kerneldoc.pdf
- LegOS Documentation
  http://user.it.uu.se/~tobiasa/RT-TF01/legosdoc/index.html
- Hitachi 8/300 Processor architecture and instruction set
  http://moss.csc.ncsu.edu/~mueller/rt/mindstorm/h3314.pdf
- LegOS Network Protocol
  http://legos.sourceforge.net/files/linux/LNPD/
- CSC714 Lecture Slides by Dr. Mueller
  http://www.ncsu.edu/csc714