CSC 714
Real Time Computer Systems

Progress Report 1
(On 31st October, 2005)

Implementation of EDF, PIP, PCEP in BrickOS

Sushil Pai
spai@ncsu.edu

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

Current Progress and next steps:

S. No.	Task Description	Status
1.	Familiarize with the existing scheduling mechanism of Brick OS	Completed
2.	Identifying the data-structures and functions associated with tasks and resources that need to be changed for implementing the scheduling mechanisms - The objective was to make as less changes to the existing data structures as possible. For this, semaphores will be implemented as resources. - The new data structures (for resources, priority ceiling) and the modified data structures have been provided below in the pseudo code. - For simplicity, currently the period of the task will be considered as its deadline.	Completed
3.	Implement communication between the RCX and a PC using LNP	
4.	PCEP Implementation and verification - Pseudo code for the PCEP was completed - Next Step: implement the pseudo code and test it	Pseudo-code Completed
5.	PIP Implementation and verification	Pseudo-code In Progress
6.	EDF Implementation and verification	
7.	PCEP+EDF Implementation and verification	
8.	PIP+EDF Implementation and verification	

Pseudo Code for PCEP implementation:

(The changes have been marked in blue)

1. Data structures:

```
struct _pchain_t
{
        priority_t priority;            //!< numeric priority level
        struct _pchain_t *next;         //!< lower priority chain
        struct _pchain_t *prev;         //!< higher priority chain
        struct _pdata_t  *cpid;         //!< current process in chain
};

struct _pdata_t
{
        //existing variables
        size_t *sp_save;                //!< saved stack pointer
        pstate_t pstate;                //!< process state
        pflags_t pflags;                //!< process flags
        pchain_t *priority;             //!< priority chain      //current priority
        struct _pdata_t *next;          //!< next process in queue
        struct _pdata_t *prev;          //!< previous process in queue
        struct _pdata_t *parent;        //!< parent process
        size_t *stack_base;             //!< lower stack boundary

        //New variables
        pchain_t *init_priority;        //!< initial priority    [needed for PIP/EDF]

        //existing functions
        wakeup_t(*wakeup) (wakeup_t);       //!< event wakeup function
        wakeup_t wakeup_data;               //!< user data for wakeup fn

};


// This data structure is required to maintain a list of resources. It is also required to recomputed the priority ceiling of the system
typedef struct resource                 [needed for PCEP/PIP]
{
        sem_t *sempahore;               // semaphore associated with the resource
        int priority_ceiling;           // priority ceiling of the resource
        pdata_t holding_task;           // task that is currently operating on the resource
        resource *next;                     // pointer to the next resource
};

//Pointer to the first resource of the system
```

```
resource *resource_list;        [needed for PCEP/PIP]

//Anonymous class that maintains the priority ceiling of the system
struct                          [needed for PCEP/PIP]
{
        int ceiling_priority=0;         // priority ceiling of the system
        pdata_t ceiling_task;           // task responsible for the current priority ceiling ????
}ceiling;
```

2. Functions:

```
void init_resource(resource *curr_resource, int priority_ceiling)
{
        //initialize the semaphore of the resource
        sem_init(curr_resource->sempahore, 0, 1);
        curr_resource->priority_ceiling = priority_ceiling;

        //add the resource to the list of resources
        curr_resource->next points to NULL

        if( resource_list is empty)
        {
                resource_list points to curr_resource
        }
        else    // there are resources in the list
        {
                curr_resource->next points to the resource pointed by resource_list
                resource_list points to curr_resource
        }
}


int get_resource(resource *curr_resource)
{
        sem_trywait(curr_resource->sempahore)
        increase ceiling->ceiling_priority to curr_resource->priority_ceiling
        ceiling->ceiling_task points to cpid
        return successful
}

//This function is called when a
int release_resource( sem_t *semResource)
{
        if( sem_trywait(&task_sem) )
                return unsuccessful
```

```
        sem_post(semResource)
        ceiling->ceiling_priority = recompute_ceiling()

        sem_post(&task_sem)
}


size_t *tm_scheduler(size_t *old_stack_pointer)
{
        pdata_t  *next // next process to execute
        pchain_t *curr_priority
        wakeup_t tmp


        // if the task semaphore is blocked, do not perform any scheduling
        if( sem_trywait(&task_sem) )
        {
                return old_stack_pointer
        }

        // take care of the task that is currenly running
        // if necessary remove the task from the list
        switch(cpid->pstate)
        {
                case P_ZOMBIE:
                                if(cpid->next!=cpid)
                                {
                                        //remove task from chain for this priority level
                                        priority->cpid is cpid->prev
                                        cpid->next->prev is cpid->prev
                                        cpid->prev->next is cpid->next
                                }
                                else
                                {
                                        // remove priority chain for this priority level
                                        if(priority->next)
                                                  priority->next->prev = priority->prev;
                                        if(priority->prev)
                                                priority->prev->next = priority->next;
                                        else
                                                priority_head = priority->next
                                        free(priority)
                                }

                                free cpid->stack_base                   // free stack
```

```
                        free cpid                    // free process data

                        switch(--nb_tasks)
                        {
                                case 1:
                                        // only the idle process remains
                                        if(priority_head->cpid!=pd_idle)
                                        {
                                                fatal("ERR00")
                                        }
                                        *((pd_idle->sp_save) + SP_RETURN_OFFSET ) = (size_t) &exit
                                        pd_idle->pstate=P_SLEEPING
                                        break

                                case 0:
                                        // the last process has been removed
                                        // -> stop switcher, go single tasking
                                        systime_set_switcher(&rom_dummy_handler)
                                        cpid=&pd_single
                                        sem_post(&task_sem)
                                        return cpid->sp_save
                        }
                        break
   case P_RUNNING:
                        cpid->pstate=P_SLEEPING
                        // no break

                        case P_WAITING:
                        cpid->sp_save=old_stack_pointer
}

// find next process willing to run by traversing through the list of tasks that need to be executed

curr_priority points to priority_head
next points to curr_priority->cpid->next


repeat forever
{
        if(next->pstate == P_SLEEPING)
                break

        if(next->pstate == P_WAITING)
        {
                // this is the highest priority task that can be executed
                // check if the task has higher priority than priotity ceiling of the system
```

```
                if( next->parent->priority > ceiling.ceiling_priority )
                {
                    tmp = next->wakeup(next->wakeup_data)
                    if( tmp != 0)
                    {
                        next->wakeup_data = tmp
                        goto NEXT_FOUND
                    }
                }
            }


            if(next == priority->cpid)
            {
                // go to next priority level
                if(priority->next != NULL)
                    priority = priority->next
                else
                {
                    // FIXME: idle task has died
                    //      this is a severe error.
                    fatal("ERR01")
                }
                next=priority->cpid->next
            }
            else
                next=next->next
        }

NEXT_FOUND:
        cpid=next->priority->cpid=next      // execute next process
        cpid->pstate=P_RUNNING

        sem_post(&task_sem)
        return cpid->sp_save
}

//This function is called when the current task releases a resource
int recompute_ceiling()
{
        //initialize
        ceiling.ceiling_priority is 0
        curr_resource points to first resource of the resource_list
        while( end of the resource list)
        {
            if( curr_resource->holding_task is not NULL)
```

```
        {
            if( curr_resource->priority_ceiling > ceiling.ceiling_priority )
                curr_resource->priority_ceiling = ceiling.ceiling_priority
        }
        curr_resource points to the next resource in the resource_list
    }
}


pid_t execi(int (*code_start)(int,char**),int argc, char **argv,priority_t priority,size_t stack_size)
{
    pdata_t *pd
    pdata_t *pdata              // for traversing
    pchain_t *pchain, *ppchain  // for traversing priority chain
    pchain_t *newpchain         // for allocating new priority chain
    size_t *sp
    int freepchain=0

    // get memory
    // task & stack area belong to parent process
    // they aren't freed by mm_reaper()
    if((pd=malloc(sizeof(pdata_t)))==NULL)
        return -1
    if((sp=malloc(stack_size))==NULL)
    {
        free(pd)
        return -1
    }
    // avoid deadlock of memory and task semaphores
    // by preallocation.
    if((newpchain=malloc(sizeof(pchain_t)))==NULL)
    {
        free(pd)
        free(sp)
        return -1
    }

    pd->pflags = 0
    if ((size_t)code_start < (size_t)&mm_start)
        pd->pflags |= T_KERNEL

    pd->stack_base=sp           // these we know already.
    sp+=(stack_size>>1)         // setup initial stack

    // when main() returns a value, it passes it in r0
    // as r0 is also the register to pass single int arguments by
```

```
        // gcc convention, we can just put the address of exit on the stack.
        *(--sp)=(size_t) &exit

        // we have to construct a stack stub so tm_switcher,
        // systime_handler and the ROM routine can fill the
        // right values on startup.

        *(--sp)=(size_t) code_start     // entry point    < these two are for
        *(--sp)=0                       // ccr            < rte in ROM
        *(--sp)=0                       // r6             < pop r6 in ROM
        *(--sp)=(size_t)
        &rom_ocia_return                // ROM return     < rts in systime_handler

        *(--sp)=(size_t) argc           // r0             < pop r0 in systime handler
        *(--sp)=(size_t)
        &systime_tm_return                      // systime return < rts in tm_switcher

        *(--sp)=(size_t) argv           // r1..r5  < pop r1..r5 in tm_switcher
        *(--sp)=0
        *(--sp)=0
        *(--sp)=0
        *(--sp)=0

        pd->sp_save=sp                          // save sp for tm_switcher
        pd->pstate=P_SLEEPING                   // task is waiting for execution
        pd->parent=cpid                         // set parent

        sem_wait(&task_sem)

        ppchain=NULL

        for(pchain = priority_head; pchain != NULL && (pchain->priority) > priority; ppchain = pchain, pchain
= pchain->next);

        if(pchain==NULL || pchain->priority!=priority)
        {
                // make new chain

                newpchain->priority=priority
                newpchain->cpid=pd

                newpchain->next=pchain
                if(pchain)
                        pchain->prev =newpchain
                newpchain->prev=ppchain
                if(ppchain)
```

```
                ppchain->next=newpchain
        else
                priority_head=newpchain

        // initial queue setup
        pd->prev=pd->next=pd
        pd->priority=newpchain
        //store the initial priority of the task for PIP
        pd->init_priority = newpchain
    }
    else
    {
        //check if the process is already present in the priority queue
        for(pdata=pchain->cpid; *(pd->sp_save+11)!=*(pdata->sp_save+11); pdata=pchain->cpid->next
);

        if( *(pd->sp_save+11) == *(pdata->sp_save+11) )
                kill (pdata)

        // add the new task to the back of queue
        pd->priority=pchain
        //store the initial priority of the task for PIP
        pd->init_priority = pchain
        pd->prev=pchain->cpid->prev
        pd->next=pchain->cpid
        pd->next->prev=pd->prev->next=pd
        freepchain=1    // free superfluous pchain.
    }

    nb_tasks++

    sem_post(&task_sem)
    if(freepchain)
        free(newpchain)
    return (pid_t) pd
}
```

3. Sample Program:

```
//sample program to be used to check the scheduler
resource *resource1;
resource *resource2;
resource *resource3;

int main(int argc, char *argv[])
{
    //Start the task manager
```

```
        tm_start();
        ceiling.ceiling_priority = 0;

        //Initialize 3 resources
        resource_init(resource1,08);
        resource_init(resource2,12);
        resource_init(resource3,12);


        while(1)
        {
                execi(&task1, 0, NULL, 10, DEFAULT_STACK_SIZE);
                sleep(20);
                execi(&task2, 0, NULL, 12, DEFAULT_STACK_SIZE);
                sleep(20);
                execi(&task3, 0, NULL, 08, DEFAULT_STACK_SIZE);
                sleep(60);
        }
}

void task1(int argc, char *argv[])
{
        <instructions_independent_of_resources>

        get_resource(resource2);

        <instructions_that_require_resource2>

        release_resource(resource2);
}

void task2(int argc, char *argv[])
{
        <instructions_independent_of_resources>

        get_resource(resource3);

        <instructions_that_require_resource3>

        get_resource(resource2);

        <instructions_that_require_resource2>

        release_resource(resource2);

        release_resource(resource3);
```

```
}

void task3(int argc, char *argv[])
{
	<instructions_independent_of_resources>

	get_resource(resource1);

	<instructions_that_require_resource1>

	release_resource(resource1);

	<instructions_independent_of_resources>

	get_resource(resource2);

	<instructions_that_require_resource2>

	release_resource(resource2);
}
```