CSC 714
Real Time Computer Systems

Progress Report 2
(On 15th November, 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 fewer 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 - Two techniques were investigated: using pyLNP (which uses python) or writing a C Program. Of these 2, the C program is currently being used - Functions that have been implemented: o On RCX: transmits text (task name/additional details) along with the current system time. o On PC: read the data transmitted by the RCX and display it	Completed
4.	PCEP Implementation and verification - Pseudo code for the PCEP was completed - Code has been implemented and is currently being tested	In Progress
5.	PIP Implementation and verification - Pseudo code for PIP was completed	Pseudo-code completed

	- Next Step: Implement the code on the RCX	
6.	EDF Implementation and verification	
7.	PCEP+EDF Implementation and verification	
8.	PIP+EDF Implementation and verification	

Problems faced:

1. Difficulties with pyLNP: Due to lack of adequate documentation on pyLNP, there were some difficulties in importing the LNP related setting. So, instead the c program approach was tried.
2. 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.

Pseudo-code:

The following is the pseudo code of the two protocols. In order to make this simpler to comprehend, only the additions and the modifications to the existing code are provided. All the changes are associated to the respective policies of the protocols:

1. Priority Ceiling Emulation Protocol:

1. Initialization of the system/ Updating of system ceiling

Whenever all the resources are free, the ceiling of the system is Ω. The ceiling ΄Π(t) is updated each time a resource is allocated or freed.

1.1 New structure for resource and a pointer to maintain a list of the resources

```
typedef struct resource
{
    sem_t *semaphore;            // semaphore associated with the resource
    int priority_ceiling;        // priority ceiling of the resource
    tdata_t holding_task;        // task that is currently operating on the resource
    resource *next;                    // pointer to the next resource
     int waiting_process_flag=0; // is 1 if there is a higher priority process waiting for this resource
};

resource *resource_list;
```

The structure will be used for PIP also. In order to use the existing semaphore functionalities provided by BrickOS, a semaphore variable has been used as member of the structure.

1.2 void init_resource (resource *curr_resource, int priority_ceiling)

This function initializes a resource. The function will be used in PIP also.

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

 //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
 }
}
```

1.3 int recompute_system_ceiling()

This function is called when a task releases a resource. In the case when a task acquires a resource, this function is not called; the system ceiling will be computed based on the priority of the task acquiring the resource. But, when a resource is released, we need to go through all the resource that are currently being utilized and compute the system ceiling. This function is not used by PIP.

```
int recompute_system_ceiling()
{
 ceiling.ceiling_priority is initialized to 0
 curr_resource points to first resource of the resource_list
 traverse through the list of resources
 {
  if( curr_resource is being held by a task)
  {
   if( curr_resource->priority_ceiling is greater than the ceiling.ceiling_priority )
   curr_resource->priority_ceiling = ceiling.ceiling_priority
  }
  go to the next resource in the resource list
 }
}
```

1.4 Changes to pid_t execi(int (*code_start)(int,char**),int argc, char **argv,priority_t priority,size_t stack_size)

An assumption made in the project is that the deadline of a task is equal to the period of the task. As a result, at the end of the period, if the current task has not met its deadline, it is removed from the ready queue.

```
…
 #ifdef Impl _PCEP_PIP
   //check if the process is already present in the priority queue
   for(pdata=pchain->tpid; *(pd->sp_save+11)!=*(pdata->sp_save+11); pdata=pchain->tpid->next );
   if( *(pd->sp_save+11) == *(pdata->sp_save+11) )
     kill (pdata)
 #endif
 // add the new task to the back of queue
 pd->priority=pchain
 ...
```

2. Scheduling Rule: After a job is released, it is blocked from starting executing until its assigned priority is higher than the current ceiling ï ï‚¢(t) of the system. At all times, jobs that are not blocked are scheduled on the processor in priority-driven, preemptive manner according to their assigned priorities.

2.1 changes to size_t *tm_scheduler(size_t *old_stack_pointer)
A new check that needs to be performed here is that when a new higher priority task wants to execute, check if the priority of the task is higher than the system ceiling. Only if it is true, let the task execute, else, execute the current task.

```
  ...
 repeat forever
 {
   if(next->pstate == P_SLEEPING)
     break
   if(next->pstate == P_WAITING)
   {
     // this is the highest priority task that can be executed
     #ifdef Impl _PCEP
     // check if the task has higher priority than priority ceiling of the system
     if( next->parent->priority > ceiling.ceiling_priority )
     {
     #endif
       if ((next->tflags & T_SHUTDOWN) != 0)
     {
       next->wakeup_data = 0;
       break;
     }
     ctid = next;
     tmp = next->wakeup(next->wakeup_data);
     if (tmp != 0)
     {
       next->wakeup_data = tmp;
       break;
```

```
    }
    #ifdef Impl _PCEP
    }
    #endif
  }
 }
 ...
```

3. 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.

3.1 int get_resource(resource *curr_resource)

This function is called when a task wants to grab a resource. When a task wants to use a resource, the resource is allocated to the task if the resource is free. After that we have to recompute the system ceiling.

```
int get_resource(resource *curr_resource)
{
 //The following condition will occur only for PIP
 if( sem_trywait(curr_resource->semaphore) != 0 )
 {
  #ifdef Impl_PIP
    // the priority of the current task is inherited by the task holding the resource
    update the waiting_process_flag of the resource to 1
    call function for curr_resource->holding_task to inherit the priority of the current task (inherit_priority)
    yield the rest of the time slot
  #endif
 }

 #ifdef Impl_PCEP
  increase ceiling->ceiling_priority to curr_resource->priority_ceiling
  ceiling->ceiling_task points to tpid
 #endif
 return successful
}
```

3.2 int release_resource(resource *res)

This function is called when a task releases a resource. We need to recompute the system ceiling once the resource is released.

```
int release_resource(resource *res)
{
 sem_post(res->semaphore)

 #ifdef Impl_PIP
   if( the waiting_process_flag is 1)
     decrease the priority of the current task to its original priority
     yield the rest of the time slice // so that the higher priority task can resume execution
   else
    let the current process continue execution
 #endif

 #ifdef Impl_PCEP
   //In case of PCEP, the new system ceiling needs to be recomputed
   ceiling->ceiling_priority = recompute_ceiling()
 #endif
}
```

2. Priority Inheritance Protocol:

1. Initialization of the system

1.1 New element added to the task structure (struct _tdata_t)
pchain_t *init_priority

This pointer points to a node of the pchain_t linked list corresponding to the initial priority of the task.

1.2 New structure for resource and a pointer to maintain a list of the resources

```
typedef struct resource
{
    sem_t *semaphore;              // semaphore associated with the resource
    int priority_ceiling;          // priority ceiling of the resource
    tdata_t holding_task;          // task that is currently operating on the resource
    resource *next;                        // pointer to the next resource
      int waiting_process_flag=0; // is 1 if there is a higher priority process waiting for this resource
};

resource *resource_list;
```

This structure is same as the structure used for PCEP. In order to use the existing semaphore functionalities provided by BrickOS, a semaphore variable has been used as member of the structure.

1.3 void init_resource(resource *curr_resource, int priority_ceiling)

This function initializes a resource. It is same as the function used in PCEP. As PIP does not require a priority ceiling, it has been ignored.

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

  //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
  }
}
```

1.4 Changes to pid_t execi(int (*code_start)(int,char**),int argc, char **argv,priority_t priority,size_t stack_size)

As we need to remember the initial priority of the task, the new variable (init_priority) is initialized with the

```
  pd->priority=newpchain;
  #ifdef Impl _PIP
    pd->init_priority=newchain;
  #endif
```

2. 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.

3. 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.

3.1 int get_resource(resource *curr_resource)

This function is called when a task wants to grab a resource. In the case of PCEP a resource is always allocated to the task, but in the case of PIP, we need to check the availability of the resource before allocating it to a task. Unlike PCEP, we do not have to recompute the system ceiling.

```
int get_resource(resource *curr_resource)
{
  //The following condition will occur only for PIP
  if( sem_trywait(curr_resource->semaphore) != 0 )
```

```
 {
  #ifdef Impl_PCEP
    this is an error
  #endif

  #ifdef Impl_PIP
    // the priority of the current task is inherited by the task holding the resource
    update the waiting_process_flag of the resource to 1
    call function for curr_resource->holding_task to inherit the priority of the current task (inherit_priority)
    yield the rest of the time slot
  #endif
 }

 #ifdef Impl_PCEP
  increase ceiling->ceiling_priority to curr_resource->priority_ceiling
  ceiling->ceiling_task points to tpid
 #endif
 return successful
}
```

3.2 int release_resource(resource *res)
This function is called when a task releases a resource. In the case of PCEP, we recompute the system ceiling, which we do not have to do in the case of PIP. The new condition that we need to handle is that we need to check if a higher priority task is waiting for the resource being released. If so, then the higher priority process should resume execution and not the current process.

```
int release_resource(resource *res)
{
 sem_post(res->semaphore)

 #ifdef Impl_PIP
  if( the waiting_process_flag is 1)
    decrease the priority of the current task to its original priority
    yield the rest of the time slice // so that the higher priority task can resume execution
  else
   let the current process continue execution
 #endif

 #ifdef Impl_PCEP
  //In case of PCEP, the new system ceiling needs to be recomputed
  ceiling->ceiling_priority = recompute_ceiling()
 #endif
}
```

4. 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.

4.1 void inherit_priority(tdata_t* curr_task, resource* res)

As the task are put in a double-linked list based queue, the function would have to change the next and previous pointers of the current task, the task holding the resource and the tasks before and after the 2 tasks in the task queue.

```
void inherit_priority(tdata_t* curr_task, resource* res)
{
  res->holding_task is the task that is currently holding the resource
  update the next, prev pointers and the priority pointer of res->holding_task to increase the priority of the holding task
  the holding task is placed above the current task in the queue
}
```