**ECE 354: Real Time Operating System**

**Project 2**

**Documentation**

**Context Switching**

Context switching saves the current process ID and the loads the loads the next process’ PID. This is why two methods were created for context switching. One of them is save\_context, which saves the current process’ stack pointer and the other method, load\_context, stores the next process’ stack pointer. The following is the pseudocode for context switching

Void save\_context()

set up a process pointer to point to the current process

current process = get\_proc(process id)

push data and address registers into the stack

move the stack pointer in the stack to the current process’ stack pointer

Void load\_context()

set up a process pointer to point to the next process

next process = get\_proc(process id)

store stack pointer (sp) to equal a pointer for the next process’ current stack pointer

if (sp != NULL)

load A7 with the stack pointer

if(next process’ state == READY)

pop data and address registers into the stack

move the stack pointer in the stack to the next process’ stack pointer

**Scheduler**

First initialize scheduler by creating ready and blocked queues and setting the running process to null

Void scheduler\_init()  
set the size of the ready queue based on the size of the malloc declared in malloc.c

Initialize ready queue

set the size of the blocked queue based on the size of the malloc declared in malloc.c

Initialize blocked queue

Set the running process to null

The function scheduler\_run selects a process to run based on its priority

Void scheduler\_run()

If(ready queue’s head is null || running process’ priority <= ready queue’s head priority)

Return

If(running process != null)

Enqueue running process

Set running process state to READY

Dequeue new running process

Set running process state to RUNNING

**Null Process**

The null process is the lowest priority process (Priority #4) that enables the RTOS to keep on running if there are no processes ready to run or are in the ready queue. The null process has an infinite loop. The release processor function will be called in this infinite loop.

Void null\_process()

while(true)

Release processor

**Process Control Block**

The process control block (PCB) is the data structure that contains all the necessary information required for a process. The data structure contains the following fields

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Description** |
| ID | Integer | ID of a process (unique for each process) |
| Priority | Integer | Priority for each process |
| State | Integer | Process’ current state |
| Block\_Type | Integer | Type of block for each process |
| Void (\*entry) () | Void function | Points to the starting address of the process |
| Curr\_sp | Integer \* | Pointer to the current stack pointer |
| Sz\_stack | Integer | Size of the stack |

**Trap function handling**

The functions release\_processor, set\_process\_priority are kernel level funtions, which need TRAP function in order for the user to change from the user mode to the kernel mode.

|  |
| --- |
| /\*Process Management\*/  int release\_processor() {     rtx\_dbug\_outs((CHAR \*)"rtx: release\_processor \r\n");         setpr(4);         asm( "TRAP #10" );     return 0; } |

The trap handler declared in trap.c and trap.h is called by the functions declared in rtx.c. The trap handler declared in the trap.c calls the kernel level implementations of these functions, which are re-named and called as k\_”function names”

|  |
| --- |
| int CURR\_TRAP = 0; void setpr(int value) {         CURR\_TRAP = value; }  VOID c\_trap\_handler( VOID ) {  //initialize variables  switch (CURR\_TRAP) {  //cases for other functions                  case 4:                         k\_release\_processor();                         break;  ………..  } |

The c\_trap\_handler makes switches to kernel level functions from the specified user level functions.

**Release Processor**

The release processor function declared in process.c , swaps the current process one with the next one in the ready queue, there by releasing the current running process.

|  |
| --- |
| if(running\_process->state == STATE\_RUNNING){                 running\_process->state = STATE\_READY;                 enqueue(ready\_queue, running\_process->ID, running\_process->priority);                  running\_process = NULL;  }  scheduler\_run(); |

It function first saves the values of the current process in the data registers and then carries out the switching between processes, this functionality is carried out in context switching function, while referring to the process ID.

save\_context(running\_process->ID);

When the currently running process is detected, by using the parameter state = STATE\_RUNNING then the running process in enqueued in the ready queue.

After which the scheduler is run in order to detect any pre-emption. A Success is returned when the process is successfully finished.

**RTX initialization**

The RTX initialization is declared in the init.c file . The file has three functions load\_null\_process(), the load\_test\_processes() and the init\_pcb().

The init\_pcb() function initializes the PCB structure, thereby allowing us to define the actual structure by which processes will be loaded and called. It first calls the load\_null\_process() and then the load\_test\_process().

The load\_null\_process() loads the null process in the beginning. It allows us to define the starting of the stack pointer from where all processes will be handled & carries out the saving of the exception stack pointer on top of it.

The load\_test\_process() saves all the given processes and their priorities. Puts the exception stack on top of it and makes the stack pointer point to the new starting point.

|  |
| --- |
| void load\_null\_process() {  all\_processes[0].ID = 0;  all\_processes[0].priority = 4;  ……..  // Increment the current\_sp as this will the starting point of the stack of the next process  current\_sp = malloc(all\_processes[0].sz\_stack) + all\_processes[0].sz\_stack;//current\_sp+all\_processes[i].sz\_stack ;  all\_processes[0].curr\_SP = current\_sp;    // Save the Exceprtion Stack Frame  \*all\_processes[0].curr\_SP = all\_processes[0].entry;  ……….  }  void load\_test\_processes() {  int i;  /\* get the third party test proc initialization info \*/  \_\_REGISTER\_TEST\_PROCS\_ENTRY\_\_();  // Save Test Processes  for (i =1; i< NUM\_TEST\_PROCS+1; i++ ) {  all\_processes[i].ID = g\_test\_proc[i].pid;  all\_processes[i].priority = g\_test\_proc[i].priority;  …..  // Increment the current\_sp as this will the starting point of the stack  current\_sp = current\_sp+all\_processes[i].sz\_stack ;  all\_processes[i].curr\_SP = current\_sp;  ……  }  }  void init\_pcb()  {  load\_null\_process();  load\_test\_processes();  } |

**Set\_ Priority**

The set process priority is defied as a kernel level function, which allows us to change the priority of the current processes. It takes process ID and priority as its function parameters and is called by the trap handler.

A return value of 0 is given for success and -1 otherwise.

If the process is currently running, then we just simply change it’s priority, otherwise If the process is in a blocked queue or ready queue then we remove the process from the queue, change it’s priority and enqueue the process back again in the same queue. After this we run the scheduler again in order to determine the new sequence in which the process should be called, just in case there is pre-emption.

|  |
| --- |
| int k\_set\_process\_priority(int process\_ID, int priority)  {  …….  if (state == STATE\_RUNNING)  {  get\_proc(process\_ID)->priority = priority;  }  else if (state == STATE\_BLOCKED)  {  remove(blocked\_queue, process\_ID);  get\_proc(process\_ID)->priority = priority;  enqueue(blocked\_queue, process\_ID, priority);  }  else  {  remove(ready\_queue, process\_ID);  get\_proc(process\_ID)->priority = priority;  enqueue(ready\_queue, process\_ID, priority);  }  /\* Run scheduler in case a process needs to be pre-empted \*/  scheduler\_run();  return RTX\_SUCCESS;  } |

**Get\_ Priority**

It returns the priority of a given process by using it’s process\_ID. The kern el level defined funtion, when initiated by the trap handler calls the get\_proc function with the process\_ID, which in turn returns the pointing value of the all\_process[process\_ID], which indeed is the priority of the process.

The function returns the priority if success, else a -1 if not successful.

|  |
| --- |
| int k\_get\_process\_priority(int process\_ID)  {  return get\_proc(process\_ID)->priority;  }  struct process \* get\_proc(int process\_ID)  {  return &(all\_processes[process\_ID]);  } |