ECE 354: Project 2 Part A

Group: ECE.354.S11-G031
Members: Ben Ridder (brridder), Casey Banner (cccbanne), David Janssen (dajjanss)

This document is concerned with the high level implementation of the specified primatives. As such, most of the low level back end is omitted for simplicity. Any kernel structures required to understand the primitive are described when the primitive is discussed.

RTX Initialization

The RTX initialization happens in three phases: process, priority queues, and interrupts. Pseudo code for all of these phases can be seen in Listing 1.

The process initialization happens in `init_processes()`. For each test process, a PCB is created based on the process table supplied by the test cases. This PCB is then added to the kernel's own process list but not added to a priority queue yet. Then, for each process in the kernel's process list, a stack pointer is setup and a exception frame is added to the process' new stack. The format/vector word portion of the exception frame is set to 0x4000 to represent a 4 byte aligned stack and the SR portion is 0x0000. The program counter in this frame is set to the entry point of the process.

Next, priority queues are initialized. The data structures are already allocated but are not initialized. To ensure proper operation, the queues are put into a consistent empty state. After the queues are setup, the processes are added to appropriate queue.

Finally, the interrupts are setup. First, the vector base register (VBR) for the interrupts is set to memory location 0x10000000. Next, the soft interrupt for `system_call()` is installed to the VBR at vector 0.

Listing 1: Pseudo code for RTX initialization

```
void init() {
    init_processes()
    init_priority_queues()
    init_interrupts()
}

void init_processes() {
    init_test_processes()
    add null process to process list
    for each process in process list:
        initialize stack pointer
        push PC portion of exception frame onto the stack
        push F/V and SR portions of exception frame onto
            the stack
```

```
}

void init_test_processes() {
    for each test process:
        setup PCB
        add to process list
}

void init_priority_queues() {
    for each process:
        add process to correct priority queue
}

void init_interrupts() {
    initialize VBR to 0x10000000
    install system call ISR at vector 0
}
```

Release Processor

The release processor primitive is used to make a scheduling decision. Currently, this happens when the currently running process calls `release_processor()`. After making a mode switch, the kernel makes a decision on the next process to execute. When this decision is made, a context switch is made to the selected process. See Listing 2 for a high level pseudo code representation of this primitive

The kernel process decision currently uses several priority queues, four priority levels with two sub-queues. Every priority level has a `READY` and a `DONE` queue. Initially, all process in a priority level are in the priority level's `READY` queue and after execution they are moved to the `DONE` queue. In order to decide which process next, the kernel dequeues the next process on the highest non-empty `READY` priority queue. If all `READY` queues are empty, the `DONE` queues are moved to the `READY` queues and the kernel tries to select a process again.

A context switch is made after the kernel selects the next process. If there is a process running, the registers are saved on the stack and it's stack pointer is moved to the PCB. It's state is marked as `STATE_READY` to indicate it's state has been saved and is capable of executing again. Next, the selected process' stack pointer is restored. If the process is in the state `STATE_STOPPED`, meaning it has never been run before, the kernel changes the state to `STATE_RUNNING` and executes the process by returning from the exception. On the other hand, if the process has been run before, meaning that it is in the ready state, the data and address registers are restored from the stack. In this case, the method returns normally. Execution continues at the same it left off.

Listing 2: Pseudo code for `release_processor()`

```
int release_processor() {
    mode switch to kernel mode using system call
        exception

    if all queues are empty:
        move DONE queues to READY queues

    dequeue next process from highest non-empty priority
        queue

    if there is a currently running process:
        save all of the registers on to the process stack
        save stack pointer to PCB
        set state to STATE_READY

    restore next process stack pointer
    if next process state is STOPPED:
        return from exception using the next process
            exception frame
    else if state is READY:
        restore registers from stack

    mode switch to user mode using system call exception
        frame
    return RTX_SUCCESS
}
```

Priority Set and Get

The `set_process_priority(...)` and `get_process_priority(...)` primitives are used to set and get, respectively, the priorities of the specified process ID.

Setting the process priority is shown as pseudo code in Listing 3. First, the values of the process ID and the priority are saved to registers as a mode switch to kernel mode needs to be performed. These values are retrieved after the mode switch. After the mode switch, a check is performed to ensure that the ID and priority is valid. If it isn't, `RTX_ERROR` is returned. Otherwise, another check is made to see if the ID belongs to the currently running process and that there is a running process. If this is true, the priority of the running process is simply changed and `RTX_SUCCESS` is returned. If it is not the currently running process, the process is first removed from the appropriate queue based on its current priority level. Then the priority is updated to the new one and the process is enqueued onto the queue associated with the new priority with the same queue type as the one it was removed from. Finally, the function returns

RTX_SUCCESS to indicate successful completion.

Listing 3: Pseudo code for setting the process priority

```
int set_process_priority(int pid, int priority) {
    move pid and priority into separate data registers
    mode switch to kernel mode

    retrieve pid and priority from the data registers

    if invalid PID or invalid priority:
        return RTX_ERROR

    if running process is the process to change:
        update running process to the new priority
    else:
        dequeue process from appropriate queue
        update priority of that process
        enqueue process to its new priority queue

    mode switch to user mode

    return RTX_SUCCESS
}
```

Compared to setting the priority level, getting a process' priority is straight forward as can be seen in Listing 4. Similar to setting the priority, the process ID is moved into a data register and system is switched into kernel mode. After retrieving the process ID, a check is made to ensure that the process id is valid. If it is invalid RTX_ERROR is returned. The calling process should perform a check of its own to ensure that the returned value is not a negative which indicates an error. Finally, the process is looked up on the process table and the priority is returned from that.

Listing 4: Pseudo code for getting the process priority

```
int get_process_priority(int pid) {
    move pid into a data register
    mode switch to kernel mode

    retrieve pid from data register

    if the pid is invalid:
        return RTX_ERROR

    get process from the process table using pid

    mode switch to user mode
```

```
    return process priority
}
```

Null Process

The null process was implemented to adhere to the project specifications. It has a priority of four and a process ID of zero. The process method is defined as an infinite loop that constantly calls `release_processor()`. It is added to the process table through a call to `init_processes(...)`. As this process is required to always be at process ID zero, care should be taken to ensure that this is not overwritten by other processes in the initialization methods. This process is outlined in Listing 5.

Listing 5: Pseudo code for the null process

```
void process_null() {
    while (true):
        release_processor();
}
```