RTX Operating System Report

Tyler Babaran

insert student number here
EMAIL@uwaterloo.ca

Kelly McBride

insert student number here
EMAIL@waterloo.ca

Peter Socha

20484453
psocha@uwaterloo.ca

April 6, 2015

Contents

List of Algorithms

List of Figures

Chapter 1

Introduction

Kelly

Chapter 2

Design Description

2.1 Global Variables and Structures

Kelly

2.2 Memory Management

2.2.1 Memory Structure

Tyler

2.2.2 Requesting Memory Blocks

Tyler

2.2.3 Releasing Memory Blocks

Tyler

2.3 Processor Management

2.3.1 Process Control Structures

Kelly

2.3.2 Process Queues

Tyler

2.3.3 Process Scheduling

Kelly

2.4 Process Priority Management

2.4.1 Get Process Priority

Peter

```
int k_get_process_priority(int process_id);
```

The k_get_process_priority primitive is used to get the priority of a process. It takes a process ID as an input parameter. It outputs one of two things:

- The process's m_priority member if the process id is valid
- RTX_ERR (equal to -1) if the process id is invalid

k_get_process_priority never modifies any process and never modifies any process queues.

2.4.2 Set Process Priority

Tyler

2.5 Interprocess Communication

2.5.1 Message Structure

Kelly

2.5.2 Sending Messages

Tyler

2.5.3 Receiving Messages

Tyler

2.5.4 Delayed Send

Peter

```
int k_delayed_send(int process_id, void *message_envelope, int delay);
```

The purpose of delayed sending is so that messages are not received immediately. A message will not end up in the message queue of the receiving process until after a delay period has passed. `k_delayed_send` is the kernel primitive used to facilitate such message-passing.

The procedure begins by verifying its parameters, returning `RTX_ERR` if any of the values are unacceptable. It then creates a `message` structure using `message_new` with a delay equal to the `delay` argument given. In regular message-sending, the `delay` variable is set to zero. The message is finally enqueued on the message queue of the timer i-process. Regardless of the message's intended destination, it is given first to the timer i-process and all later handling is done by the timer i-process.

Algorithm 1 The delayed send function

```
procedure DELAYED_SEND(processID, message, delay)
    if delay is negative then
        return RTX_ERR
    end if
    if No process has id processID then
        return RTX_ERR
    end if
    m = create message object with expiry time of now + delay
    enqueue m on the timer iprocess's message queue
    return RTX_OK
end procedure
```

2.6 Interrupts and I-Processes

2.6.1 UART I-Process

Peter

The UART i-process has two functions:

- Read characters entered through the keyboard
- Output characters to the screen

The uart iprocess sometimes sends messages to the KCD and CRT processes. If this happens, a `g_uart_flag` variable is set to 1. When the iprocess completes, the asssembly routine that handles the interrupt checks the value of `g_uart_flag`. If the flag is true, it will call `k_release_processor()` to give KCD and CRT a chance to immediately run.

Algorithm 2 The uart iprocess

```
procedure UART_I_PROCESS
    g_uart_flag = 0
    if receive data available then
        read g_char_in from register
        if _DEBUG_HOTKEYS is enabled then
            PROCESS_HOT_KEY(g_char_in)
        end if
        if heap space is available then
            m = K_REQUEST_MEMORY_BLOCK
            Set the mtype of m to CRT_DISP
            Set the mtext of m to g_char_in
            Send m as a message to the CRT process
            g_uart_flag = 1
        end if
        if g_char_in is a carriage return then
            if heap space is available then
                m = K_REQUEST_MEMORY_BLOCK
                Set the mtype of m to DEFAULT
                Set the mtext of m to g_input_buffer
                Send m as a message to the KCD process
                g_uart_flag = 1
            end if
            reset the g_input_buffer
        else
            append g_char_in to g_input_buffer
        end if
    else if output data available then
        Receive the message m
        Output the mtext of m until the null terminator is reached
        K_RELEASE_MEMORY_BLOCK(m)
    end if
end procedure
```

The i-process's functionality requires it to request memory in order to make messages. In order to prevent the i-process from ever getting blocked, it never allocates memory if there is no free space available.

When the `_DEBUG_HOTKEYS` flag is enabled, a set of characters has special status. When one of the characters below is typed, a message is printed to the screen using uart1. The messages consist of lists of processes, with the process id and priority stated line by line. These characters can be pressed at any time are not counted towards strings used for command-passing.

- `!` prints the processes on the ready queue
- `@` prints the processes on the blocked-on-memory queue
- `#` prints the processes on the blocked-on-receive queue
- `$` prints the process that is currently running

2.6.2 Timer I-Process

Peter

The timer i-process is called by the timer interrupt handler, which runs 1000 times each second. The purpose of the iprocess is to send delayed messages and to update the global timer count `g_timer_count`.

The timer i-process treats its message queue differently than the other processes in the RTX. Instead of popping messages off of its queue one at a time, it scans the entire queue each time it runs. The expiry time of each message is compared with the current time and the message is sent to the destination process if the expiry time has passed. A special non-preemptive message-sending procedure is used so that the i-process is not pre-empted before it has finished scanning the queue. This non-preemptive sender places the receiving process on the ready queue but does not run it. Instead, a flag variable `g_timer_flag` is set that will cause `k_release_processor()` to run when the i-process is finished (this call is not made in the process, but in the assembly wrapper that handles the interrupt).

The timer i-process does not use `receive_message()` to read its queue and therefore never gets blocked.

Algorithm 3 The Timer iprocess

```
procedure TIMER_I_PROCESS
    disable interrupts
    increment g_timer_count
    g_timer_flag = 0
    for message m in the timer iprocess's message queue do
        if m's expiry_time is less than the present time then
            g_timer_flag = 1
            remove m from the timer iprocess's message queue
            send the message to its desintation process without preempting
        end if
    end for
    enable interrupts
end procedure
```

2.7 System Processes

2.7.1 Null Process

Peter

The null process has the lowest priority of any process in the operating system. It runs only when there are no ready processes to be run. When it runs, all it does is invoke `k_release_processor()` so that the kernel can check if there is a ready process to be run.

Algorithm 4 The null system process

```
procedure NULL_PROCESS
    while true do
        K_RELEASE_PROCESSOR()
    end while
end procedure
```

2.7.2 KCD Process

Peter

The Keyboard Command Decoder process exists so that users can send console commands to the system at runtime. A command can be registered by sending the KCD process a `KCD_REG` type message. The KCD maintains a list of registered commands inside an array. When a `DEFAULT` command is sent to the KCD, the KCD will try to recognize the command. If the command is found, the KCD will send a message to the process that registered the command; the message will have the `KCD_DISPATCH` type. In both cases, the contents of the command are stored inside the message's `mtext`. The KCD process is an intermediary between the UART i-process (which registers the keystrokes) and the eventual receiving message (which executes the command).

The KCD process assumes that command strings contain no whitespace. It assumes that any information between the first space and the end of the line is supplementary.

Algorithm 5 The KCD System Process

```
procedure KCD_PROCESS
    while true do
        message = RECEIVE_MESSAGE()
        if message is of type DEFAULT then
            Read the mtext up to first whitespace or newline
            Try finding the mtext in the command array
            if command is found then
                Send KCD_DISPATCH message to the process that registered the command. Send the entire mtext as contents.
            end if
        else if message is of type KCD_REG then
            Read the mtext and sending process
            Add the command to the command array
        end if
        RELEASE_MEMORY_BLOCK(message)
    end while
end procedure
```

2.7.3 CRT Process

Peter

The CRT process is used to print text to the system console. The process waits for messages of type `CRT_DISP`. If it receives such a message, it will send it to the UART i-process and modify the `IER` register so that the UART treats the message as an output message. The UART is interrupted and therefore the UART i-process will start to run immediately.

Algorithm 6 The CRT Process

```
procedure CRT_PROCESS
    while true do
        message = RECEIVE_MESSAGE()
        if message is of type CRT_DISPLAY then
            Send message to UART iprocess
            Set interrupt bits
        else
            RELEASE_MEMORY_BLOCK(message)
        end if
    end while
end procedure
```

2.8 User Processes

2.8.1 Wall Clock Process

Peter

The wall clock process is used to display the time in 24-hour format. If the clock is on, it will print the time on the screen each second by sending messages to the CRT process. The process maintains second-by-second timing by sending itself delayed messages with a delay of 1000 milliseconds.

The wall clock process registers three commands with the KCD when it initializes. By typing commands into the console, the user can control the wall clock's behaviour. When the process launches, the wall clock starts as inactive.

- `%WR` - Sets the clock time to 00:00:00 and sets the clock to active.
- `%WT` - Sets the clock to inactive. Cancels any scheduled future clock ticks.
- `%WS hh:mm:ss` - Sets the clock to active with the given time in 24-hour format.

The wall clock process relies on memory blocks to send display messages to the CRT process and to send itself delayed messages to re-awaken itself. If system memory has been depleted, it will be unable to properly funcion.

2.8.2 Set Priority Process

Peter

```
%C process_id new_priority
```

The `set_process_priority()` primitive described earlier can be used to programmatically change the priority of any process that is not a system process. It is, however, a programmatic call that must be set in user code in advance. The Set Priority process allows users to change the priority of a process at runtime using the `%C` command.

The priority change takes effect immediately. If the user enters invalid parameters, "Error" will be printed to the screen and the command will be ignored.

2.8.3 Stress Test Processes

Peter

Algorithm 7 The Set Priority Process

```
procedure SET_PRIORITY_PROC
    register with KCD as %C command
    while true do
        message = RECEIVE_MESSAGE()
        parse message mtext to get a process_id and new_priority
        if Setting process_id to new_priority is a valid operation then
            SET_PROCESS_PRIORITY(process_id)new_priority
        else
            make and send an "Error" message to CRT
        end if
        RELEASE_MEMORY_BLOCK(message)
    end while
end procedure
```

The stress test processes are a collection of three user processes called A, B, and C. The three of them are used to test how the system copes with the depletion of heap blocks in memory.

Process A waits until it receives a `%Z` command, after which it will repeatedly request memory, make messages, and send those messages to Process B.

Process B receives messages from A and sends them to Process C.

Process C receives messages from Process B. Every 20th message, it prints "Process C" to the screen by modifying B's message and passing it to the CRT. Every 20th message, it will then request a memory block and send itself a delayed `WAKEUP10` message to be received in 10 seconds. During those 10 seconds, it goes into a hibernation state, receiving messages from B and putting them on its local queue, but otherwise not doing anything.

In general, Process A requests memory blocks and Process C ends up releasing them. If Process C's priority is too low, memory blocks may end up never being released and we may end up in deadlock.

2.9 Initialization

Kelly

2.10 Testing

Tyler

Chapter 3

Major Design Changes

Add more sections as appropriate

3.1 Structure of Process Queue

Tyler

Chapter 4

Lessons Learned

Everyone contribute something

4.1 Source Control and Code Management

We used GitHub as a repository for our code, which proved to be very helpful. However, we never developed any systematic protocols for using GitHub and we did not take advantage of many of its features.

Nearly all development was done on the master branch and was pushed directly to the master branch. We rarely coded on the same module at the same time so conflicts were surprisingly rare. However, we did not have a systematic code review process. While this saved us time in the short run, it meant that team members did not have much of a chance to learn about the code that the other team members were writing. The team became overly specialized; many of the modules in the OS were well-understood by only one team member. A more systematic review process may have helped keep all members on the team well-grounded on all aspects of the OS.

4.2 Team Dynamics and Scheduling

There were no major conflicts between any of the team members, which proved beneficial for all of us.

We did not have a systematic process for allocation and scheduling of tasks. Usually a release cycle would begin as a free-for-all with members choosing parts they wanted to work on. Later on in the cycle, an allocation system would be determined, but it was informal, frequently did not go according to plan, and rarely carried any concrete deadlines for individual group members.

With P1 and P2 in particular, we encountered time trouble and needed to use a late day for each. We could have avoided this problem by allocating responsibilities more precisely and maintaining deadlines for the main milestones.

Chapter 5

Timing and Analysis

Peter

In order to do timing, a second timer (called "timer 1") was programmed. Whereas timer0 interrupted once every millisecond, timer1 never interrupts and never does anything other than maintain a count. Like timer0, timer1 operated at a speed of 100MHz.

The timer test code was added to the user test processes, with the tests conducted in `proc5()` after all the regular tests had been completed. The three primitives that had their time measured were `k_request_memory_block()`, `k_send_message()`, and `k_receive_message()`. The test was designed so that there would be no blocking or pre-emption during any of these calls. Each of these functions was called ten times inside a loop and the elapsed time was measured using the timer's `TC` register. By writing certain values to the timer's `TCR` register, we were able to programmatically start, stop, and reset the timer during and between tests.

The raw data proved to be very consistent and exhibited zero variance in 24 total runs. Since the timer ticked at 100MHz, it meant that each clock tick represented 10ns in time. Since each time value was obtained on a sample of 10 calls made consecutively, we divide by 10 to get the time elapsed per individual call. Here is the data, represented as ns per call.

	`k_request_memory_block`	`k_send_message`	`k_receive_message`
Time (ns)	420	1028	866

Figure 5.1: Timing Test Results

Judging by the high consistency of the results, the hardware always executed the code in the same predictable manner without any optimizations or stalls.

The primitives for message-handling are more expensive than the primitive for memory. This difference is likely due to the fact that the message primitives work with the `message` object type, which is larger and more complex than the `heap_blk` object type used to manage heap blocks.