RTX Operating System Report

Tyler Babaran

20457511
tbabaran@uwaterloo.ca

Kelly McBride

20479819
ke2mcbri@waterloo.ca

Peter Socha

20484453
psocha@uwaterloo.ca

April 6, 2015

Contents

List of Algorithms

List of Figures

Chapter 1

Introduction

Kelly

This report is a design document outlining the RTX operating system written by Tyler Babaran, Kelly McBride, and Peter Socha as part of the SE 350 course at the University of Waterloo. The OS is written for an MCB1700 board with an LPC1768 microcontroller.

The purpose of this report is to provide documentation on the operating system. It will outline the kernel API available to anyone programming processes for the OS. The provided system and user processes are described in detail. Information on initialization and interrupts it also included. Finally, there is a section describing the time performance of the some of the key kernel primitives.

The ruling concept of the RTX detailed here is simplicity. Some of the methods used to achieve 'simplicity' are unconventional, but the idea of simplicity still holds true. The priority queues are implemented as sorted linked lists where high priority processes are placed first, the memory heap is implemented in a way that might take a few explanations, and the message-envelope structure may seem unconventional, but each part is simple enough and they are separate concerns so that when the RTX functions, every part functions together as a whole in a simple and expected way.

Chapter 2

Design Description

2.1 Global Variables and Structures

Kelly
The global variables are structures used in the RTX are primarily for initialization, though some are core to its functionality.

- `gp_stack`: points to the last allocated stack address, used when allocating process stacks
- `g_test_procs`: an array of `PROC_INIT` structs, used for initializing user processes
- `g_proc_table`: an array of `PROC_INIT` structs, containing both system and user processes for initialization
- `gp_pcbs`: the array containing the PCB for every process, used during context switches constantly
- `gp_current_process`: a pointer to the current process' PCB, also used constantly
- Process Queues: `g_ready_queue`, `g_blocked_on_memory_queue`, `g_blocked_on_receive_queue`, the three process queues are global and used constantly as well, to place each process onto the correct queue based on its outstanding requests and messages.

In terms of global structures, both `heap_blk` and `pcb` are global and widely used, `heap_blk` for allocating memory blocks in the heap, and `pcb` for the process control blocks. Both are detailed further on.

2.2 Memory Management

2.2.1 Memory Structure

Tyler
The 'heap' that the processes request memory blocks from is maintained as a linked list. The length of the heap is exactly enough space for thirty memory blocks of size 0x80 (128 bytes).

Free space nodes keep track of the length of free space following this address and a pointer to the next chunk of unclaimed blocks. These headers are written directly into the blocks as they are

returned to the heap. They are free to be overwritten, the only safe header is the starting address of the heap. The heap is shifted by four bytes to protect the last block from ever being overwritten.

The request and release methods maintain the optimal structure of the list, deleting and overwriting excess or useless nodes.

2.2.2 Requesting Memory Blocks

Tyler

```
int k_request_memory_block(void);
```

First, the procedure checks if all memory blocks in the system heap have been reserved already. If so, this process becomes blocked and will be unblocked once another process releases one of the memory blocks.
Once a memory block is free the procedure iterates through the free space nodes like a linked list, searching for the first chunk of free space that is large enough to satisfy the request. Then the procedure passes a pointer to the first address of the block to the process and updates the heap to account for this space being reserved.

Algorithm 1 The memory request function

```
1: procedure REQUEST_MEMORY_BLOCK
2:     while heap is full do
3:         block the current process
4:     end while
5:     update the free space list
6:     return the address of the top of the block
7: end procedure
```

2.2.3 Releasing Memory Blocks

Tyler

```
int k_release_memory_block(void* memory_block);
```

Releasing memory blocks requires a pointer to a memory block. The procedure focuses on optimally returning the block to the heap list. Then it unblocks the next process waiting for a memory block, if any.

2.3 Processor Management

2.3.1 Process Control Structures

The Process Control Block (PCB) is the primary process control structure. It contains the `m_pid`, `m_priority`, and `m_state` as integers, and `mp_sp`, the process' stack pointer, `mp_next`, a PCB pointer used for our process queues, and `m_queue`, the pointer to that process' message queue.

Algorithm 2 The memory release function

```
1: procedure RELEASE_MEMORY_BLOCK(*memory_block)
2:     if this block is the top block of the heap then
3:         modify heap header node (never gets overwritten)
4:     end if
5:     if there is free space immediately beneath this block then
6:         combine them by increasing this block's length
7:     else this block becomes a new block node, is added to the list
8:     end if
9:     if there is free space immediately beneath this block then
10:        combine them by increasing this block's length
11:    end if
12:    if a process is blocked on memory then
13:        unblock that process, release the processor
14:    end if
15: end procedure
```

This is all of the information necessary for context switching from one process to the next. Users cannot view nor modify any process' PCB directly.

2.3.2 Process Queues

Tyler

The process queues are implemented to use one queue for all priorities, instead of a separate queue for each priority. Each `PCB` contains a `PCB` pointer, allowing the `PCB`s themselves to be the queue nodes.

By using generic methods that use pointers of pointers as parameters, the same four methods are used for all process queues:

```
void enqueue(pcb** targetQueue, pcb* element);
pcb* dequeue(pcb** targetQueue);
void remove_queue_node(pcb** targetQueue, pcb* element);
U32 is_empty(pcb* targetQueue);
```

The `enqueue` method compares the priorities of each process in the queue and inserts the element beneath all processes of equal value, ensuring FIFO ordering.

`dequeue` simply pops the highest priority element off the front of the queue.

The `remove_queue_node` procedure is used when processes change priority and need to be removed and reinserted back into the queue. This searches the queue for a particular `PCB`, based on process ID.

`is_empty` is purely a helper function for the other three.

2.3.3 Process Scheduling

Kelly

The scheduler is pre-emptive on I/O and inter-process communication. The scheduling algorithm itself is very simple: take the process from the top of the ready queue. The ready queue is sorted based on priority and then by order of arrival. The null process is always the last process on the

ready queue and it is chosen when there are no ready processes. The enqueuing and dequeueing procedures ensure that the processes are at the correct location, keeping both priority and order in mind. This scheduler does not use a time quantum nor pre-emption on a time quantum.

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

```
int k_set_process_priority(int process_id, int priority);
```

Process priority changing is all done through this method. After checking that the process is allowed to have its priority changed and the priority it is being set to is legal, the priority value is set and the process is removed and re-enqueued to the queue matching the process's current state. The processor is then released to allow for higher priority processes to take over.

Algorithm 3 The process priority changing function

```
1: procedure SET_PROCESS_PRIORITY(processID, targetPriority)
2:     if processID or targetPriority are not legal then
3:         return RTX_ERR
4:     end if
5:     set processID's priority to targetPriority
6:     determine state
7:     remove queue node from its current queue
8:     enqueue processID's PCB into the correct queue
9: end procedure
```

2.5 Interprocess Communication

2.5.1 Message Structure

Kelly

There are two structures used for message passing in the RTX, `msgbuf` and `message`.
The `msgbuf` structure is the general structure that both users and the kernel know about. It is typically placed at the start of an allocated memory block and contains integer `mtype` which describes the purpose of the message, be it `KCD_REG` to register a new command type, or `DEFAULT` for a general message, or any other of the defined types. The only other member of the `msgbuf` structure is then `mbody`, a `char` array of size 10 to contain the message contents.
The `message` structure is used only by the kernel and contains `message_envelope`, a pointer to the `msgbuf` within the allocated memory block, integers for the `sender_id`, `destination_id`, `expiry_time`, and a pointer to another message structure `mp_next` for use in the message queues.
Typically the `msgbuf` structure is allocated at the start of a memory block, and the `message` structure is allocated in the space right after, still before the 64-byte mark.

2.5.2 Sending Messages

Tyler

```
int k_send_message(int process_id, void* message_envelope);
```

Message passing is implemented using a queue stucture very similar to the process queues. Processes can call this method after creating a typical `message_envelope` containing a message type and a char array of the message's contents. The function call also requires a destination process that will receive this message. `k_send_message` is the kernel primitive used to facilitate message sending.

The procedure begins by using `message_new` to create a `message`, our kernel level wrapper for the `message_envelope` containing additional necessary information such as the sender process's ID and the pointer necessary for message queue construction. The `message` is then enqueued on the message queue of the destination process. If the destination process is `BLOCKED_ON_RECEIVE`, we unblock and switch, releasing the processor to determine which process should be running next.

Algorithm 4 The send message function

```
1: procedure SEND_MESSAGE(destinationID, message envelope)
2:     m = create message object from message envelope
3:     enqueue m on the destination process's message queue
4:     if destination process is waiting for a message then
5:         move destination process to READY state
6:         pre-empt the processor, switching to the waiting process
7:     end if
8:     return RTX_OK
9: end procedure
```

2.5.3 Receiving Messages

Tyler

```
int k_receive_message(int* sender_id);
```

Each process maintains its own message queue in the PCB. `k_receive_message` is the kernel primitive used to facilitate message receiving. When a process tries to receive a message, first the procedure checks if the message queue is empty. If so, this process becomes `BLOCKED_ON_RECEIVE` and will become unblocked by the `send_message` procedure.

Once a message is in the queue, the procedure dequeues the first message and returns the message envelope data containing the message type and contents. The `sender_id` pointer parameter is set as the `sender_id` from the `message` wrapper for the process to cross reference and confirm this is the message it was waiting for.

Algorithm 5 The receive message function

```
1: procedure RECEIVE_MESSAGE(*sender_id)
2:     while message queue is empty do
3:         block this process
4:     end while
5:     m = dequeue the top message
6:     set sender_id as m's sender_id
7:     return m's message_envelope
8: end procedure
```

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 6 The delayed send function

```
1: procedure DELAYED_SEND(processID, message, delay)
2:     if delay is negative then
3:         return RTX_ERR
4:     end if
5:     if no process has id processID then
6:         return RTX_ERR
7:     end if
8:     m = create message object with expiry time of now + delay
9:     enqueue m on the timer iprocess's message queue
10:    return RTX_OK
11: 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.

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

Algorithm 7 The uart iprocess

```
1: procedure UART_I_PROCESS
2:     g_uart_flag = 0
3:     if receive data available then
4:         read g_char_in from register
5:         if _DEBUG_HOTKEYS is enabled then
6:             PROCESS_HOT_KEY(g_char_in)
7:         end if
8:         if heap space is available then
9:             m = K_REQUEST_MEMORY_BLOCK
10:            Set the mtype of m to CRT_DISP
11:            Set the mtext of m to g_char_in
12:            Send m as a message to the CRT process
13:            g_uart_flag = 1
14:        end if
15:        if g_char_in is a carriage return then
16:            if heap space is available then
17:                m = K_REQUEST_MEMORY_BLOCK
18:                Set the mtype of m to DEFAULT
19:                Set the mtext of m to g_input_buffer
20:                Send m as a message to the KCD process
21:                g_uart_flag = 1
22:            end if
23:            reset the g_input_buffer
24:        else
25:            append g_char_in to g_input_buffer
26:        end if
27:    else if output data available then
28:        Receive the message m
29:        Output the mtext of m until the null terminator is reached
30:        K_RELEASE_MEMORY_BLOCK(m)
31:    end if
32: end procedure
```

- `$` 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 8 The Timer iprocess

```
1:  procedure TIMER_I_PROCESS
2:      disable interrupts
3:      increment g_timer_count
4:      g_timer_flag = 0
5:      for message m in the timer iprocess's message queue do
6:          if m's expiry_time is less than the present time then
7:              g_timer_flag = 1
8:              remove m from the timer iprocess's message queue
9:              send the message to its desintation process without preempting
10:         end if
11:     end for
12:     enable interrupts
13: 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 9 The null system process

```
1: procedure NULL_PROCESS
2:     while true do
3:         K_RELEASE_PROCESSOR()
4:     end while
5: 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).

Algorithm 10 The KCD System Process

```
1: procedure KCD_PROCESS
2:     while true do
3:         message = RECEIVE_MESSAGE()
4:         if message is of type DEFAULT then
5:             Read the mtext up to first whitespace or newline
6:             Try finding the mtext in the command array
7:             if command is found then
8:                 Send KCD_DISPATCH message to the process that registered the command. Send the entire mtext as contents.
9:             end if
10:        else if message is of type KCD_REG then
11:            Read the mtext and sending process
12:            Add the command to the command array
13:        end if
14:        RELEASE_MEMORY_BLOCK(message)
15:    end while
16: end procedure
```

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.

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 11 The CRT Process

```
1:  procedure CRT_PROCESS
2:      while true do
3:          message = RECEIVE_MESSAGE()
4:          if message is of type CRT_DISPLAY then
5:              Send message to UART iprocess
6:              Set interrupt bits
7:          else
8:              RELEASE_MEMORY_BLOCK(message)
9:          end if
10:     end while
11: 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, Ërrorẅill be printed to the screen and the command will be ignored.

Algorithm 12 The Set Priority Process

```
 1: procedure SET_PRIORITY_PROC
 2:     register with KCD as %C command
 3:     while true do
 4:         message = RECEIVE_MESSAGE()
 5:         parse message mtext to get a process_id and new_priority
 6:         if Setting process_id to new_priority is a valid operation then
 7:             SET_PROCESS_PRIORITY(process_id)new_priority
 8:         else
 9:             make and send an Ërrorm̈essage to CRT
10:         end if
11:         RELEASE_MEMORY_BLOCK(message)
12:     end while
13: end procedure
```

2.8.3 Stress Test Processes

Peter

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 P̈rocess C̈to 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

Process Initialization is comprised of four parts:

- Process table setup: each process' data (`m_pid`, `m_stack_size`, `mpf_start_pc`, `m_priority`) is entered in `g_proc_table` for further initialization and housekeeping.
- PCB intialization and stack allocation: the data from `g_proc_table` is used to populate the PCBs in `gp_pcbs` and each process is allocated a stack of size `m_stack_size`.
- PCB enqueueing: every PCB is enqueued in the ready queue, `g_ready_queue`.
- Messages to set up some processes: processes which register a `KCD` command often do so as their first action so that the command can be used as soon as possible.

Once these steps are complete, the scheduler selects the first process to run from the ready queue and execution begins.

2.10 Testing

Tyler

We used the user processes to test the functionality of our system. We implemented a dynamic test polling system, where the user processes updated a list of execution ordering as they switched between each other.
This list is constantly compared with a preset expected execution order, and discrepencies cause the TEST_PASSED flag to flip. Since the texts printed in order, it became easier to identify at what point in the test execution the unexpected behaviour occurred.

Towards the end of the project, we implemented a second system timer to observe how long executions would take to execute. This allowed us to investigate inefficiencies and determine where our code base could be improved.

Chapter 3

Major Design Changes

Add more sections as appropriate

3.1 Structure of Process Queue

Tyler
By the end of P1 we had the blocked and ready queue each use a separate set of queueing functions. The introduction of the b̈locked on receiveq̈ueues forced us to rework the queueing methods to operate generically, passing a reference to the queue head along with each function call.
This cleaned and simplified a large section of the code base, greatly reducing our repeated code.

3.2 Heap Faults

Through stress tests we encountered a unique edge case in our memory heap assignment. The order that the KCD and Wall Clock requested blocks lead us to a chain of memory leaks that would have otherwise gone unnoticed. These leaks were caused by incorrectly calculating the shift required to protect the head heap node, causing it to be overwritten and the entire heap structure would be lost.

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.