RTX Software Design Report

Clement Hoang

20531116
c8hoang@uwaterloo.ca

David Su

20516776
dysu@uwaterloo.ca

Cole Vander Veen

20503626
cgvander@waterloo.ca

Peter Li

20522308
y648li@uwaterloo.ca

Winter 2016

Contents

List of Algorithms

List of Figures

Chapter 1

Introduction

The purpose of this report is to outline the design and implementation of the RTX code written by the group members, Clement Hoang, David Su, Peter Li, and Cole Vander Veen, as part of the SE350 course at the University of Waterloo. The OS is designed for a Keil MCB1700 Cortex-M3 board, with a LPC1768 microcontroller.

The report aims to provide documentation for the operating system, in order to facilitate the use and understanding for anyone interested in programming for the OS. As such, this report outlines the global variables used in the OS, then moves on to describing the kernel API in a modular and chronological way. Finally, the report closes with some analysis on the OS, and challenges that the group faced for the duration of the lab.

Chapter 2

Design Description

2.1 Global Variables and Data Structures

- `memQueue`: A data structure that models the free physical memory in the OS, by splitting the heap into blocks of equal size. It is represented by a `memQueue` data structure, which is a linked list of `MemBlock` nodes of size `BLOCK_SIZE` (`BLOCK_SIZE` is a constant defined in the OS). When a process requests memory using the kernel API call `request_memory_block`, a block of memory is dequeued from the `memQueue` and given to the process. When a process releases memory, it is enqueued into `memQueue` again.
  - `MemBlock`: To expand, the `MemBlock` is a C-struct representing a single block of memory that holds a pointer to the next `MemBlock` in the queue. There is reserved space at the front of each memory block in case the block needs to hold an `envelope`. This space is only used if the memory block is being used to send messages. The `envelope` contains the following fields:
    - `next`: a pointer to the next envelope in a message queue
    - `sender_id`: the process ID of the message sender
    - `recv_id`: the process ID of the message receiver
    - `send_time`: the time the message was originally sent at
- `gp_pcbs`: A pointer to an array of `PCB` (Process Control Block) structs. It holds the state of every process control block in the OS, and is interacted with by functions that need to access the `PCB`s. For example, setting the process priority or getting the process priority uses `gp_pcbs` to access the priority of a specific `PCB`.
  - `PCB`: a model of a process and its state. The `PCB` contains the following fields:
    - `mp_sp`: stack pointer of the process
    - `m_pid`: ID of the process
    - `m_priority`: priority of the process
    - `m_state`: state of the process
    - `nextPCB`: pointer to the next `PCB`, if it is in a queue
    - `msgHead`: beginning of the message queue
    - `msgTail`: end of the message queue

- g_proc_table: an array of PROC_INIT structs that contains process information for initialization. The PROC_INIT contains the following:
  - m_id: ID of the process
  - m_priority: initial priority of the process
  - m_stack_size: size to allocate for the process in words

  The process table is where the initial PCB data for each user process is defined. This data is used by the process_init function to create the initialize the PCB for each process.
- gp_stack: a pointer to the end of the stack space, which is also the top of the memQueue. When initializing the heap in the function heap_init, no memory past this address may be used as a free memory block.
- p_end: a pointer to the beginning of the memory heap, which is also conceptually the first memory block in memQueue. It is calculated by starting at &Image$$RW_IRAM1$$ZI$$Limit, then incrementing it upon PCB initialization in memory_init to account for the extra space for the PCBs.
- numOfBlocks: the number of blocks of MemBlock allocated in the heap_init function call.
- g_switch_flag: a boolean flag that indicates whether to continue to run the process before the UART receives an interrupt. If 1, then it will immediately switch, and if it's 0, then it will continue to run the process.
- gp_current_process: a pointer to the PCB of the currently running process. It is used in process_switch among other functions that deal with the current process.
- ReadyPQ: a linked list of queues (the queues are also implemented as linked lists) of PCB structures. It represents a priority queue of process blocks in the ready queue. This is used for scheduling, for example inside the scheduler function.
- BlockPQ: a linked list of queues of PCB structures. It represents a priority queue of process blocks in the blocked queue. This is used when releasing memory to determine which process should be the first to receive memory. This is used inside the k_release_memory function.
- NUM_OF_PRIORITIES: a constant that represents the number of priorities in the OS (default = 5). 0-3 represent the normal priorities from highest to lowest, whereas 4 is reserved for the null process.
- g_buffer: a circular buffer that keeps track of user input and wraps around at a certain size. It is used in kcdProc and other functions that deal with uart input.
- gp_buffer: a pointer to the next location g_buffer. It is used in c_UART0_IRQHandler.
- g_buffer_end: a pointer to the last location in g_buffer. It is helpful for calculations in kcdProc.
- PROC_STATE_E: An enum which consists of the states:
  - NEW: process was just created

  - RDY: process is on the ready queue
  - RUN: process is currently running
  - BLK: process is blocked on memory
  - WAIT: process is waiting to receive a message

- MSG_BUF: This represents the parts of a message that user processes have access to, as opposed to an envelope, which the users do not have access to. A memory block can be treated as a message by casting it to MSG_BUF. It consists of the following fields:
  - mtype: a user defined message type
  - mtext: a char array which contains the body of the message

2.2 Memory Management

2.2.1 Memory Overview

By keeping track of the gp_stack and gp_pcbs pointers and keeping track of how much memory the PCBs use, then it is possible to do some pointer arithmetic to determine the starting and ending addresses of the heap (the physical memory available to allocate in the RTX).
The implementation of memory management in this RTX involves maintaining a linked list that keeps track of all the available MemBlocks on the heap. Upon process consumption of memory, MemBlocks allocated are removed from the linked list and a reference to the block is kept in the process. Upon release, the MemBlock is appended to the linked list again. This means that the runtime is $O(1)$ for deletion and insertion of blocks.
The MemBlocks have a user-defined size called BLOCK_SIZE. The system also reserves some space in the beginning of MemBlocks for envelopes. By default, the BLOCK_SIZE is set to 128 bytes.

2.2.2 Requesting Memory Blocks

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

This function returns the address of an available MemBlock. The algorithm is analogous to the dequeue function of a queue. It takes the previous head of memQueue, returns it, then updates the linked list accordingly. It also sets the status of a process to BLK and enqueues the process into the blocked queue if there is no more memory for the process to use.

Algorithm 1 Requesting memory 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
```

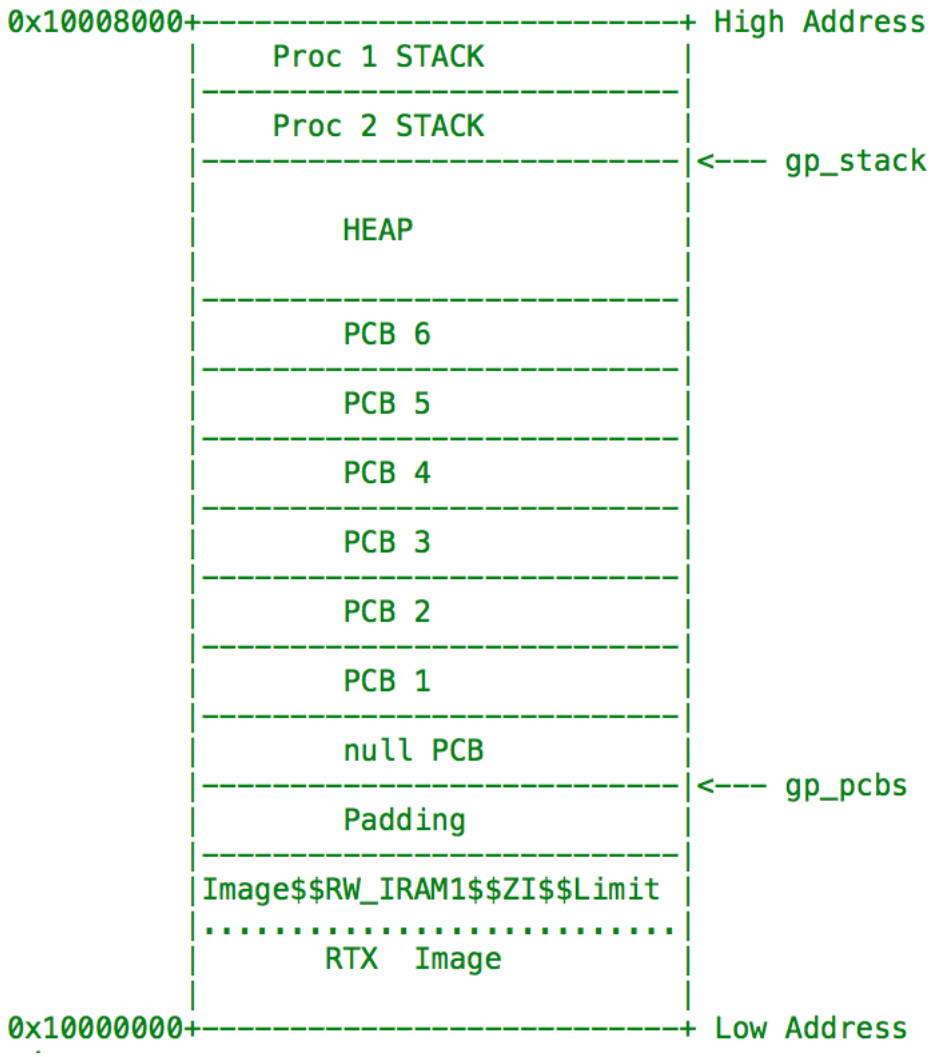

Figure 2.1: Memory Layout

2.2.3 Releasing Memory Blocks

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

This function takes a `MemBlock` parameter. This parameter is a reference to the `MemBlock` that will be freed. It also returns a flag on whether that operation was a success or not - `RTX_ERR` or `RTX_OK`. The algorithm is analogous to the enqueue function of a queue. After some validation logic, it attached the freed memory block to the end of the linked list. It also moves processes into the ready queue if a process is currently blocked.

Algorithm 2 Releasing memory 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
```

2.3 Processor Management

2.3.1 Releasing The Processor

When the processor is released by a process using `k_release_processor`, the `scheduler` is called to determine the next process to run. The `scheduler` will never choose a blocked process to run next. The `scheduler` will choose a process at the highest priority level to run at all times. The processes at the same priority level will be scheduled in a round robin fashion.
After the next process is chosen, a context switch must be performed to safely switch between processes. The state of the current process must be saved before beginning to execute the following process.

Algorithm 3 Releasing processor function

```
1: procedure RELEASE_PROCESSOR
2:     call the scheduler to determine the new process
3:     context switch from the old process to the new process
4: end procedure
```

The algorithm for `k_release_processor` is very simple, as the scheduler and the context switching algorithm implement the heavy functionality.

2.3.2 Scheduler

The scheduler is implemented using the priority queue `ReadyPQ`. The process that is dequeued from a priority queue will always be the process with the highest priority. Blocked `PCB`s are never added to the `ReadyPQ`, so the scheduler will never choose a blocked process to run. Because the most recently enqueued process is always shifted to the back of `ReadyPQ`, it will run later than the other processes in `ReadyPQ`, giving us a round robin scheduling system. This means that the priority queue aligns with the design mentioned above.

Algorithm 4 Scheduler

```
1: procedure SCHEDULER
2:     if a process was previously running then
3:         if the previous process is blocked then
4:             enqueue the previous process into BlockPQ
5:         end if
6:         if the previous process is ready then
7:             enqueue the previous process into ReadyPQ
8:         end if
9:     end if
10:    dequeue the next process from ReadyPQ and return it
11: end procedure
```

2.3.3 Context Switching

The context switch procedure has three main purposes. It must set the states of the old process and the new process to appropriate values. It must also save the stack pointer of the old process,

and restore the stack pointer of the new process. Finally, if the new process is in state NEW, it must pop the exception stack frame from the stack for the new process.

Algorithm 5 Context Switcher

```
1: procedure PROCESS_SWITCH
2:     if the new process has a state of NEW then
3:         if the new process is different from the old process then
4:             if the old process does not have a state of NEW then
5:                 if the old process is not blocked then
6:                     set the old process' state to RDY
7:                 end if
8:                 save the old process' stack pointer
9:             end if
10:        end if
11:        set the new process' state to RUN
12:        switch to new process' stack
13:        pop exception frame from the stack for the new process
14:    end if
15:    if the new process is different from the old process then
16:        if the old process is not blocked then
17:            set the old process' state to RDY
18:        end if
19:        save the old process' stack pointer
20:        set the new process' state to RUN
21:        switch to new process' stack
22:    end if
23: end procedure
```

2.4 Process Priority Management

2.4.1 Get Process Priority

The get process priority function takes in a process id as a parameter and returns its priority. The function returns -1 if the process is not found. To find the process by its id, we loop through all the PCBs which are stored in `gp_pcbs`, and compare the process_id. If a matching PCB is found, then we retrieve its priority and return. Otherwise, the loop will end without ever finding a process and will return -1.

Algorithm 6 Get Process Priority

```
1: function K_GET_PROCESS_PRIORITY(process_id)
2:     for every process in PCB array do
3:         if id of this process matches the process_id parameter then
4:             return priority of process
5:         end if
6:     end for
7:     return priority, or -1 if process is not found
8: end function
```

2.4.2 Set Process Priority

The set process priority function takes in the process id and a priority as the parameters and returns if the function has completed successfully. First we loop through all the process ids to see if we can find a process that has matching ids. Then if we have found the process, we check to see if the priority that is being set to is different from the priority it's currently at. Now at this stage we store the old priority and set the new one if its not currently executing, otherwise we would undo the priority change if the priority change would somehow fail. Finally, we release the processor since we need to immediately move the higher priority process because of the change.

Algorithm 7 Set Process Priority

```
1: function K_SET_PROCESS_PRIORITY(int process_id, int priority)
2:     if priority is within valid range then
3:         for every user process do
4:             if id of this process matches the process_id parameter then
5:                 if new priority is different then
6:                     if process is in wrong priority of ReadyPQ then
7:                         move process to correct priority
8:                     end if
9:                     release the processor (for pre-emption)
10:                end if
11:            end if
12:        end for
13:    end if
14: end function
```

2.5 Interprocess Communication

2.5.1 Message Structure

Every message uses a memory block as its underlying storage component, and has 2 parts: the part visible to the user (the struct `MSG_BUF`), and the part exclusive to the kernel (the struct `envelope`).

Every memory block created in the operating system has the first `sizeof(envelope)` bytes reserved, in case that memory block is used as a message. The envelope stores metadata about the message, such as the sender, receiver, the send time, etc.

Every process' `PCB` has a "mailbox", with a linked list of messages sent to that process that have not yet been read. Thus, the envelope also has a pointer pointing to the next envelope in that linked list.

Then, there is the `MSG_BUF` struct, which is directly accessible by the user processes. The `MSG_BUF` contains 2 pieces of data: the type of the message (often `#defined` as a constant), and the actual body of the message. The body of the message can take up the rest of the memory block (i.e. `BLOCK_SIZE - sizeof(envelope) - sizeof(MSG_BUF.mtype)`).

2.5.2 Sending Messages

The send message primitive receives as parameters the intended recipient of the message, and a pointer to the memory block containing the message. First, a few pieces of metadata is set on the envelope. Then, it merely places that message in the mailbox, located in the `PCB` of the recipient. Finally, if that message is blocked waiting for a message, it puts that message on the ready queue. Additionally, if that message has higher priority than the current running process, it releases the processor.

Algorithm 8 Send Message

```
1:  function SEND_MESSAGE(receiver_id, message)
2:      message.sender = current process
3:      message.receiver = receiver_id
4:      enqueue message in the the receiver's mailbox
5:      if receiver was blocked waiting for a message then
6:          put receiver on the ready queue, setting its state to Ready
7:          if receiver has higher priority than the current process then
8:              release the processor
9:          end if
10:     end if
11: end function
```

In addition to the `send_message` function specified above, we needed another version of the function that would not preempt under any circumstances. This was necessary for the case when the UART i-process needed to send a message. I-processes must not be preempted under any circumstances.

Algorithm 9 Send Message (Non-preemptive)

```
1:  function SEND_MESSAGE_NON_PREEMPT(receiver_id, message)
2:      message.sender = current process
3:      message.receiver = receiver_id
4:      enqueue message in the the receiver's mailbox
5:      if receiver was blocked waiting for a message then
6:          put receiver on the ready queue
7:          set receiver's state to Ready
8:      end if
9:  end function
```

Finally, we created another version of `send_message` specific to the timer i-process. The timer i-process must send messages to other processes once the timeout expires, if the message was initially sent with `delayed_send`. However, the sender of that message is not the timer i-process, but rather the original sender. This necessitated the creation of a new function. Again, this version is non-preemptive, since it is also called exclusively during an interruption.

Algorithm 10 Send Message (Timer version)

```
1: function TIMER_SEND_MESSAGE(message)
2:     receiver = message.receiver
3:     enqueue message in the the receiver's mailbox
4:     if receiver was blocked waiting for a message then
5:         put receiver on the ready queue, setting its state to Ready
6:     end if
7: end function
```

2.5.3 Receiving Messages

Receive message merely involves checking the current running process' mailbox, and checking if there are any messages there. If there is a message, dequeue it and return it immediately. Otherwise, set the current process' state to blocked on receive (`WAIT`), and release the processor. Note that processes that are blocked on receive are not in any queue; they are simply floating. When the message is sent to them they are accessed by their process ID and enqueued back into `ReadyPQ`.

Algorithm 11 Receive Message

```
1: function RECEIVE_MESSAGE
2:     while there are no messages in the current process' mailbox do
3:         set current process' state to WAIT
4:         release the processor
5:     end while
6:     dequeue the first message in the mailbox
7:     return message, sender_id
8: end function
```

Similar to sending messages, receiving messages also needs a non_blocking version, to be used by the i-processes. If there are no messages in an i-process' mailbox, it should never block. Instead, it should return `NULL` to indicate that there are no messages. This function is mainly used by the timer i-process.

Algorithm 12 Receive Message (non-blocking)

```
1: function RECEIVE_MESSAGE_NON_BLOCKING(process)
2:     if there are no messages in the process' mailbox then
3:         return NULL
4:     end if
5:     dequeue the first message in the mailbox
6:     return message, sender_id
7: end function
```

2.5.4 Delayed Send

Delayed send is implemented by sending a message to the timer i-process specifying the time when the message should actually be sent to the intended recipient.

Algorithm 13 Delayed Send

```
1: function DELAYED_SEND(receiver, message, delay)
2:     message.sender = current process
3:     message.receiver = receiver
4:     message.send_time = current_time + delay
5:     enqueue the message in the timer i-process's mailbox
6: end function
```

2.6 Interrupts and I-Processes

2.6.1 UART I-Process

The UART interrupt is enabled to send output to a display, and to receive input from the user. The OS handles those interrupts by registering a `UART0_IRQHandler` function that will be called whenever a UART interrupt occurs.
The file `uart_irq.c` is initialized by calling the function `uart_irq_init`. This function initializes the UART interrupts by setting the appropriate flags and choosing the correct UART port. Below is the interrupt handling pseudocode, starting with UART0_IRQHandler:

Algorithm 14 UART Interrupt Handler function (assembly)

```
1: function UART0_IRQHANDLER
2:     push registers onto stack
3:     call c_UART0_IRQHandler_wrapper()
4:     pop registers off stack
5: end function
```

Algorithm 15 UART Interrupt Handler function wrapper (C)

```
1: function C_UART0_IRQHANDLER_WRAPPER
2:     call c_UART0_IRQHandler()
3:     if there is another ready process with higher priority than the current process then
4:         release the processor
5:     end if
6: end function
```

Algorithm 16 Main UART Interrupt Handler function (C)

```
1: function C_UART0_IRQHandler
2:     if receive data available then
3:         read a character of user input to g_char_in
4:         if g_char_in is the null character then
5:             return
6:         end if
7:         echo_msg = get memory block (non blocking)
8:         if memory was received for echo_msg then
9:             set type of echo_msg to ECHO
10:            set text of echo_msg to a copy of the user input
11:            send echo_msg (no preemption) to KCD                ▷ Echo input back to UART
12:        end if
13:        if cur_msg is NULL (has not been created) then
14:            request a memory block for cur_msg
15:            if there is no more memory to create cur_msg then
16:                return
17:            end if
18:        end if
19:        if user input was a newline or message about to overflow memory block then
20:            set message type to DEFAULT
21:            send cur_msg (no preemption) to KCD
22:            reset cur_msg to NULL
23:        else
24:            append user input character to the end of cur_msg
25:        end if
26:    else if transmit interrupt enabled then
27:        if *gp_buffer != null character then
28:            send *gp_buffer
29:            gp_buffer = next location in circular buffer
30:        else
31:            disable transmit interrupt
32:            send null character
33:            reset gp_buffer and g_buffer_end
34:        end if
35:    end if
36: end function
```

If there is an incoming character, immediately forward it to the KCD (assuming there is free memory) to echo back to the user. Also, add that character to a buffer. Once a newline is encountered, send the entire buffer to the KCD to decode

If there are still messages in the transmit buffer, send the next character. If the character is a null character, disable the transmit interrupt as the transmission is finished.

Also, there is an `enable_UART_transmit` function that sets the appropriate flags to enable the interrupt for outputting characters. This function is called by the CRT to begin outputting characters.

2.6.2 Timer I-Process

The Timer interrupts are enabled to increment the global timer of the OS and to send messages originally sent by `delayed_send` at the appropriate times. The OS handles these interrupts by registering a `TIMER0_IRQHANDLER` that will be called whenever a timer interrupt occurs. A timer interrupt will be called every 12500 clock cycles, which takes 1ms to occur. The timer is initialized by calling the function `timer_init`. This function initializes the timer by setting several flags, including setting the PR field to 12500, which controls how often timer interrupts occur. This function also initializes other aspects of the timer, including the queue. Below is the interrupt handling pseudocode.

Algorithm 17 Timer Interrupt Handler function (assembly)

```
1: function TIMER0_IRQHandler
2:     push registers onto stack
3:     call c_TIMER0_IRQHandler
4:     pop registers off stack
5: end function
```

The main function of `TIMER0_IRQHANDLER` is to save the registers of the previously running process and call the `c_TIMER0_IRQHANDLER`, which does the majority of the computations. Then it restores the registers to jump back to the process that was running before the interrupt was called.

Algorithm 18 Timer Interrupt Handler function (C)

```
1: function c_TIMER0_IRQHANDLER
2:     increment global timer count
3:     while loop forever do
4:         receive message (non-blocking)
5:         if there are no more messages then
6:             break from loop
7:         end if
8:         insert message into Q, sorted by start time
9:     end while
10:    while there are message(s) in Q that have expired do
11:        dequeue message from Q
12:        send the message to its recipient
13:    end while
14:    if there exists a higher priority ready process then
15:        release the processor
16:    end if
17: end function
```

This C code increments the timer count and processes the messages the timer receives via `delayed_send`. The messages that need to be sent after a certain time period are stored in the local queue Q, which is sorted by the time that the messages should be sent to their recipients. This means that the message at the front of the queue is always the message that needs to be sent sooner than the rest. This algorithm first receives all the messages stored in the timer interrupt's `PCB` and inserts them into Q. Then it checks to see which messages need to be sent at the current `g_timer_count`

and sends each of these messages. This interrupt will only release the processor if there is currently a process in the `ReadyPQ` with a higher priority than the current running process, allowing the higher priority process to preempt the currently running process.

2.7 System Processes

2.7.1 Null Process

The null process simply releases the processor in an infinite loop. It should only be run if there are no other processes in the Ready state. It is necessary only because the processor should be running at all times.

Algorithm 19 Null Process

```
1: function nullProc
2:     while true do
3:         k_release_processor()
4:     end while
5: end function
```

2.7.2 CRT Process

The CRT's sole responsibility is to output the contents of the messages they receive to the UART.

Algorithm 20 CRT Process

```
1: function crtProc
2:     while true do
3:         msg = receive_message()
4:         copy text of msg to output buffer
5:         enable_UART_transmit()
6:         release_memory_block(msg)
7:     end while
8: end function
```

Algorithm 21 Copy string to circular buffer helper function

```
1: function copyToBuffer(str)
2:     for each char in str do
3:         g_buffer[g_buffer_end] = char
4:         g_buffer_end = (g_buffer_end + 1) mod BUFFER_SIZE
5:     end for
6: end function
```

2.7.3 KCD Process

The KCD's (Keyboard Command Decoder) responsibility is to decode messages that it receives, and forward them to the appropriate process that has registered itself to handle these types of messages. It must handle a variety of message types:

- Messages to register a process with a unique identifier
- Messages that should be echoed back to the UART
- Messages that begin with an identifier
- Messages that begin with a debug hotkey. There are three debug hotkeys:
  - ! will display all processes in the `RDY` state.
  - @ will display all processes in the `BLK` state
  - # will display all processes in the `WAIT` state

Algorithm 22 KCD Process

```
 1: function KCDPROC
 2:     while loop forever do
 3:         msg = receive_message()
 4:         if type of msg is KCD_REG (a regstration message) then
 5:             assign the identifier to the process that sent the message
 6:             release_message_block(msg)
 7:         else if type of message is ECHO (echoes user input) then
 8:             send msg to the CRT (echo it to the UART)
 9:         else
10:             if text of msg starts with '%' then
11:                 search for a process with the identifier in the text of msg
12:             end if
13:             if no process has this identifier then
14:                 release_memory_block(msg)
15:             else
16:                 send the message to this process
17:             end if
18:             if text of message begins with '!' then
19:                 search through PCBs for processes in ready state
20:                 send a message to the CRT to display these processes
21:             else if text of message begins with '@' then
22:                 search through PCBs for processes in blocked state
23:                 send a message to the CRT to display these processes
24:             else if text of message begins with '#' then
25:                 search through PCBs for processes in blocked on message state
26:                 send a message to the CRT to display these processes
27:             end if
28:         end if
29:     end while
30: end function
```

This processes repeatedly receives messages. If it is a keyboard command registration, the process saves the identifier along with the process that is registered to handle those commands. If it is an echo command, the message is merely forwarded to the CRT. Otherwise, if the message is an actual

command (i.e. it starts with a '%'), the message is forwarded to the process that is registered to handle it, if it exists. Finally, if the message begins with any debug hotkey, the corresponding debug output is sent to the CRT.

2.8 User Processes

2.8.1 Wall Clock Process

This process receives messages to reset, start at a specified time, and stop the wall clock. Once started, the clock will print the elapsed time every second.

Algorithm 23 Wall Clock Process

```
 1: function WALLCLOCKPROC
 2:     register itself with KCD to handle '%W' commands
 3:     id = 0
 4:     while loop forever do
 5:         msg = receive_message()
 6:         if msg contains reset command then
 7:             id++
 8:             time = 00:00:00
 9:             send increment time command to itself, delayed 1 second
10:             construct time string, send to CRT to print
11:         else if msg contains increment command then
12:             if msg's id == id then
13:                 time++
14:                 send increment time command to itself, delayed 1 second
15:                 construct time string, send to CRT to print
16:             else
17:                 release_memory_block(msg)
18:             end if
19:         else if msg contains a start at specified time command then
20:             id++
21:             time = parseTime(msg)
22:             send increment time command to itself, delayed 1 second
23:             construct time string, send to CRT to print
24:         else if msg contains stop command then
25:             id++
26:             release_memory_block(msg)
27:         else
28:             release_memory_block(msg)
29:         end if
30:     end while
31: end function
```

The local parameter "id" is used to ensure that messages containing the increment command are released when the clock is reset or terminated. When the clock is reset or terminated, its ID is

updated to ensure that increment messages sent previously (containing a different ID) are recognized as being deprecated.

2.8.2 Set Priority Process

This process registers itself with the KCD to handle "%C" commands. It accepts as parameters a process id, and the new priority for that process.

Algorithm 24 Set Priority Process

```
 1: function setPriorityProc
 2:     send message to KCD to register itself to handle "%C" commands
 3:     while true do
 4:         msg = receive_message()
 5:         parse msg's string to get the process_id and new priority to change to
 6:         if error parsing string then
 7:             print error message
 8:             continue
 9:         end if
10:         set_process_priority(process_id, new_priority)
11:         if set_process_priority returned error code then
12:             print error message
13:         end if
14:     end while
15: end function
```

2.9 Initialization

We initialize the kernel in `k_rtx_init.c`. In the initialization function, we initialize each of:

1. timer 0
2. timer 1
3. UART0 interrupt driven
4. UART1 polling
5. memory
6. PCBs
7. heap

The timers are initialized by setting appropriate configuration parameters such as the frequency. The UART is initialized by enabling UART interrupts and setting configuration parameters. The memory is initialized by allocating space for the each of the `PCBs`. Then, the fields in `PCBs` are populated from data in the process table, such as each process's priority, stack size, etc. Afterwards, stack space is allocated in memory based on how much each process requested. Then, all `PCBs` that require scheduling are placed on the ready queue. Finally, after the `PCBs` are allocated in the lower

memory addresses, and the stacks are allocated in the higher memory addresses, all remaining memory in between is split up into equally sized blocks, to be used as the heap.
One possible configuration parameter is the `BLOCK_SIZE` of the heap memory blocks. By default, it is 128 bytes. However, it can be tuned to smaller and larger values to fit the specific use case of the operating system, in order to reduce internal fragmentation.
In addition, user processes can tune their relative priorities to each other, to affect how they are scheduled, and they can request whatever stack size they require to properly function.

2.10 Testing

For P1 and P2, there were two phases to our testing procedure. At first, the group wrote many lightweight processes to test a large variety of boundary cases. Often one test was run at a time to minimize conflicts between tests and make debugging easier. By doing this we could catch the vast majority of bugs quickly and evaluate our OS in a controlled environment.
After the first phase of testing, we wrote more formal test cases to display the functionality of our OS for the demo. During our formal testing we dedicated `proc1` to displaying the output before our tests and the final results. Each individual test had one or more processes dedicated to running and verifying the results of the test. While an individual test is being run, every process that isn't dedicated to running the test simply releases the processor. Again, this minimizes the possibility that our tests will conflict and cause unexpected results.

2.10.1 P1

While testing P1, our informal tests were devoted to testing the memory management, the functions to set/get the priority of processes, and the scheduler. When testing boundary cases, we attempted to achieve node coverage of our code by testing every every permutation of removing/adding items to our queues. For example, we tried tests where every process became blocked to ensure that the `ReadyPQ` would function correctly when removing all `PCB`s (except the null `PCB`) from it.
Here are our formal test cases:

- `1`: A test to check if `get_process_priority` and `set_process_priority` function correctly. The process dedicated to this test:
  - Runs `get_process_priority` on itself
  - Sets its own priority to a different value using `set_process_priority`, then runs `get_process_priority` on itself again
  - Runs `get_process_priority` on a different process
  - Sets the priority of another process to a different value using `set_process_priority`, then runs `get_process_priority` on that process.

  If `get_process_priority` returns the correct priority level all four times, the test passes. Otherwise it fails.
- `2`: A test to check if the scheduler functions correctly with regard to differing priority levels (ie. it will not run a process if another process has a higher priority level). The process dedicated to this test first sets its own priority higher than the priority of every other process. Then it releases the processor. Since it is the highest priority process, the process should be

scheduled to run again, and no other processes should run. If any other processes run during this time, the test fails. Otherwise the test passes.

- `3`: A test to determine whether memory allocation functions correctly. The process dedicated to this test first requests several memory blocks. It then writes data to these memory blocks. Finally, the process reads the data back from the memory blocks. If the data is correct, the test passes. Otherwise the test fails.

- `4`: A test to determine if the OS behaves correctly when all memory is used up. The first process dedicated to this test requests one block of memory. Then the second process runs, requesting memory blocks until the system runs out of memory and the process becomes blocked on memory. After that, the first process runs again and releases its block of memory. If everything functions correctly, the test passes.

2.10.2 P2

For P2, our informal testing mainly focused on the message passing algorithms, the wall clock, and command registration. The majority of our informal tests were centered around how the wall clock and command registration would interact with the rest of our OS design. Normally we would run the wall clock while running our tests to ensure that they did not interfere with the wall clock's operation. We also tested our functions from P1 to ensure that they still functioned correctly in an environment with interrupts enabled.
Here are our formal test cases:

- `1`: A test to evaluate `send_message` and `receive_message`. The first process dedicated to this test sends the message `"hi"` using `send_message` to the second process. The second process receives the message using `receive_message`, and verifies that the sender is the first process. It also verifies that the message is correct. Afterwards, the second process sends the message `"bye"` back to the first process, which verifies the message in the same way. If both messages are correct, the test passes. Otherwise the test fails.

- `2`: A test to verify that `delayed_send` is functioning correctly. The process dedicated to this test sends four messages to itself with varying delays, in the following order:

  - The message `"200"` with a delay of 200ms
  - The message `"100"` with a delay of 100ms
  - The message `"50"` with a delay of 50ms
  - The message `"201"` with a delay of 201ms

  The process then receives all four messages. It confirms that each message was received in the proper order. If the messages are correct, the test passes, otherwise it fails.

- `3`: A test of the command registration. The process dedicated to this test first registers itself by sending the command `"%A"` to the KCD. Then it sends the message `"%A ABCD"` to the KCD, which should eventually be sent back to the process itself by the KCD. Finally, it receives the message and checks its validity. If the correct message is received, the test passes, otherwise it fails.

2.10.3 P3

The formal test cases for P3 were processes A, B and C as outlined in the lab manual. We tested many different combinations of priority levels for these three processes. On some priority levels (such as HIGH, MEDIUM, LOW for A, B and C respectively), there was a deadlock during the OS' operation, due to process C being unable to request a memory block to allow itself to hibernate. We concluded that while our OS' design is not capable of preventing this type of deadlock, this is not outside the standard operation for our OS.

2.11 Major Design Changes

One major change that occurred between P1 and P2 was how the memory blocks were structured. In P1, we returned a pointer to the very first byte in the memory block whenever memory was requested, giving user processes complete access to every byte in the memory block. However, in P2, we discovered that this implementation was no longer feasible. When messages get sent using a memory block, there needs to be space in the block for the OS to store metadata about that message, such as the sender, receiver, etc. Therefore, the major design change was to always leave x bytes at the beginning of each memory block for the kernel to store the metadata, and return a pointer to the beginning of the block + x bytes. This required changing code that assumed it had the entire memory block available to use.

A major stumbling block was a hardfault we were getting in P1. At that time, we did not understand the difference between the difference between `k_` prefixed functions and non-`k_` prefixed functions. It took many hours of debugging to realize that the non-`k_` prefixed functions invoke the SVC Handler, which changed the processor into kernel mode, allowing us to execute privileged instructions. When calling the `k_` prefixed functions from a user process, it immediately hardfaulted since it was trying to execute privileged instructions.

Another issue we ran into in P2 was a interrupt issue where our kernel primitives were being interrupted by the timer and UART i-processes, causing undefined behaviour. This problem was fixed by calling `__disable_irq` and `__enable_irq` to disable and re-enable interrupts at the beginning and end of kernel primitives.

If we were to start this project again, we would like to further improve the modularity of the code by furthering breaking it up into different files. Additionally, our naming conventions are quite inconsistent, and we would like to improve that as well. This would make our code much cleaner and more elegant.

Chapter 3

Lessons Learned

Upon the completion of this project, the group has gained many valuable insights. This includes things like code quality, source control, and team dynamics.

3.1 Version Control

For our codebase, we used Git as version control. At first, it was a bit painful because we were mostly committing to the master branch and we haven't found out which files to add to the .gitignore yet. This led to a lot of merge conflicts from differing binary files. However, when the project was in a more stable state, we later progressed to branching out for different tasks and modules. This proved to be quite helpful because it allowed us to separate concerns and work on tasks independently. One way we could have improved this is by having a systematic review process. Because of the team's lack of periodic code reviews and the fact that we divided the work up, the team did not have the opportunity to learn about work that other team members worked on, resulting in a team whose knowledge was extremely specialized and exclusive for each individual member. This meant that bugs relating to a certain module can only be understood by some members of the team, which sometimes slowed progress down. To improve on this, the team could have periodic code reviews to keep everyone up to date and knowledgeable in all parts of the codebase.

3.2 Code Management

When the group first started off coding, the code was extremely unmodular, files were monolithic, comments were lacking, and global variables were overly used. This made the code quite messy to look at, and as we progressed through the different parts of the lab, we found it increasingly difficult to debug and navigate through our codebase. Somewhere between P2 and P3, we hit a wall where our RTX had some fundamental bugs with regard to scheduling. However, it was very hard to trace this bug down with our style of code. We dedicated a few days to refactoring the code base; we added comments on hard to understand expressions, we tried to reduce the use of global variables if they were not needed, and we separated long functions into smaller re-usable components. The end result was that unit testing was a lot easier.

3.3 Logging and Output

For some memory issues, it was extremely hard to debug. Even when tracing through the debugger, looking at pointer values is not always the most intuitive. In order to make development easier, we decided to add scoped debug flags in our RTX in order to trace bugs without making the output more convoluted. For example, we had a `MEM_DEBUG` flag which could be enabled to output the available amount of memory blocks after each transaction in order to debug a memory leak, and have the default `DEBUG` flag enabled for less specialized debugging. This allowed us to more intuitively track errors down.

Chapter 4

Team Dynamics and Individual Responsibilities

The team members did not enforce a strict task allocation process for deciding each members' responsibilities. Generally however, the scheduling process is similar to the following scenario: Upon the initial lab introduction, all four of the members would be present, in order to discuss the material and areas of interest. Then, the members would split into two pairs to work on the parts they chose respectively, using pair programming concepts, and communication both online and offline. If one pair is finished earlier than the other, then they would usually group up with the remaining pair to work through the problem. Otherwise, they would collaboratively merge the parts they've worked on into a final working product before the deadline.

Chapter 5

Timing Analysis

In order to time various functions of our OS, we implemented a second timer. The original timer for our OS, used to implement the `delayed_send` function, triggers an interrupt every 12500 clock cycles, which is equivalent to 1ms. However, the functions `request_memory_block`, `send_message` and `receive_message` all perform their function in far less than 1ms, so this timer is useless for timing them.

In response to this, we implemented a timer that interrupts once every 13 clock cycles. The formula for calculating the period of time between interrupts was found in the timer.c file. It is $2 * (\#\text{ clock cycles per interrupt}) * (\frac{1}{25}) * 10^{-6}s = \text{period between interrupts}$. Using this formula, we calculated the period between interrupts for our new timer to be 1.04 microseconds.

With this timer we could time our functions meaningfully. For each kernel function that we wanted to test, we wrote a process that checks the time on this timer and runs the kernel function we wanted to test 70 times consecutively. After the test has been run, we record the new time on the timer and compare to the previous value to determine how long the operation of the kernel functions took. The differences in timing due to the incrementing of our timers and from the processing of the while loop surrounding our functions calls was deemed to be negligible and thus they are not analyzed.

In the time it took for `request_memory_block` to be called 70 times, there were 136 interrupts from our timer. At 1.04 microseconds per interrupt, this takes 141.44 microseconds total for all 70 function calls. This means that a single functions call required approximately 2.02 microseconds.

Similarly, in the time it took for `send_message` to be called 70 times, there were 82 interrupts from our timer. This translates to 1.22 microseconds per function call.

Finally, we determined that 116 interrupts occurred while `receive_message` was called 70 times. This means that each function call required 1.72 microseconds. Note that 70 messages were sent to our process before we began testing, so no time was spent in a WAIT state.