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 of the RTX 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.
It is aimed 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, and then moves on to describing the kernel API in a modular and chronological way, from when we implemented it. 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`. It is used by the kernel API when releasing and requesting memory, by popping a block when it is used by a process, and pushing it back in when it is released.
  - `MemBlock`: To expand, the `MemBlock` is a C-struct that holds a pointer to the next `MemBlock` in the queue. It also has reserved space in the front in case the block needs to hold an `envelope`. The `envelope` contains the following fields:
    * `next`: a pointer to the next envelope in the queue
    * `sender_id`: the process ID of the sender
    * `recv_id`: the process ID of the receiver
    * `send_time`: the time it was originally sent at
- `gp_pcbs`: A pointer to an array of `PCB` structs. It holds the state of all the process control blocks that are in the OS, and is interacted with by functions that change and read PCB states. 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` looks like 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 used often during the process_init() function.

- gp_stack: a pointer to the top of the memory heap, which is also the last conceptual memory block in memQueue. It is used during heap_init().
- p_end: a pointer to the beginning of the memory heap, which is also the first conceptual memory block in memQueue. It is calculated by starting at &Image$$RW_IRAM1$$ZI$$Limit, then incrementing it upon PCB initialization in memory_init().
- numOfBlocks: the number of blocks of MemBlock allocated in the heap_init()
- 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 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 linked lists of PCB structures. It represents a priority queue of process blocks in the blocked queue. This is used for scheduling, for example inside the scheduler() 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 the priority of the null process.
- gp_buffer: a pointer to the next location g_buffer. It is used in c_UART0_IRQHandler().
- g_buffer: a circular buffer that keeps track of the input and wraps around at a certain size. It is used in kcdProc() and other functions that deal with uart input.
- 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: It represents a message buffer and is a part of an envelope, with some space in memory blocks reserved for it. 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 `PCB`s 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 maintaing 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 has performance implications of $O(1)$ 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 doesn't take any parameters, and returns the address of an available `MemBlock`. The algorithm is basically a queue dequeue function. It takes the previous head of `memQueue`, returns it, then updates the linked list accordingly.

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
```

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 basically a queue enqueue function. 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 possible.

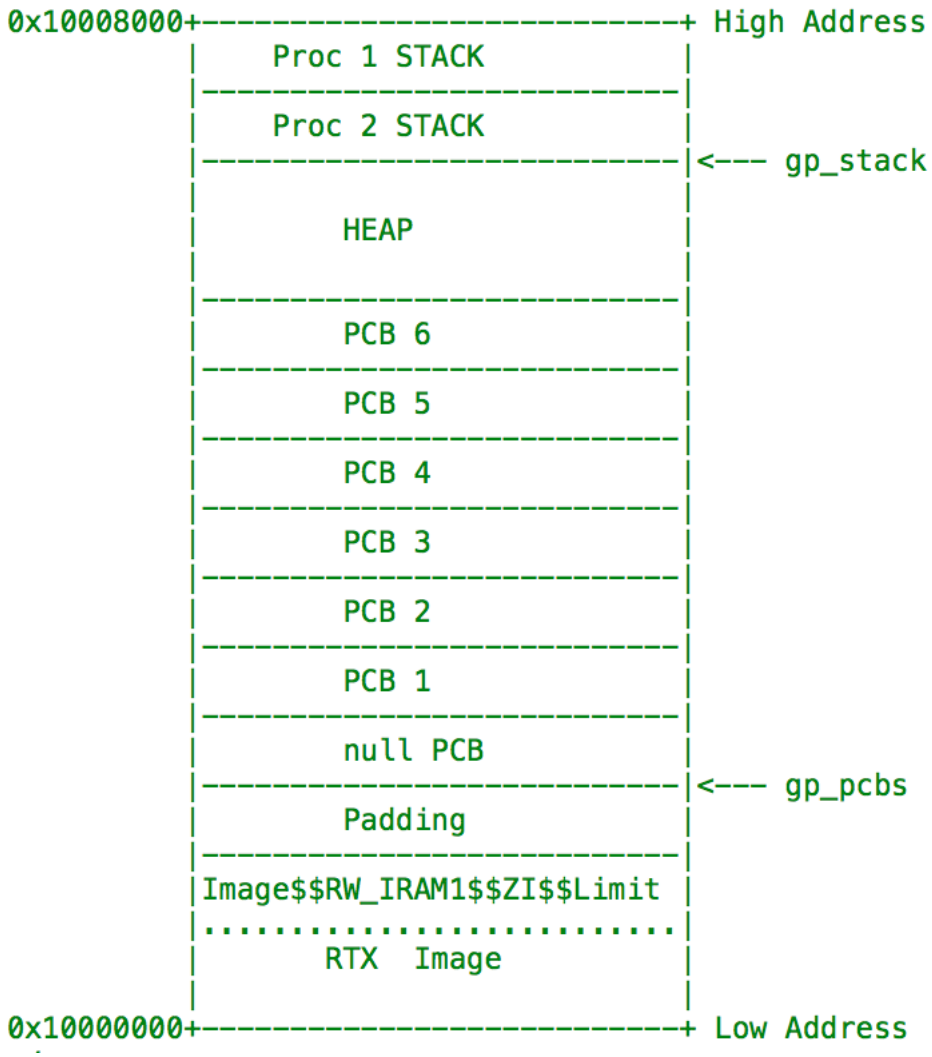

Figure 2.1: Memory Layout

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 Process Control Structures

DFASFAFD

2.3.2 Process Queues

fsadfasdfadsf

2.3.3 Process Scheduling

sdfasdfasdfdasf

2.4 Process Priority Management

2.4.1 Get Process Priority

asdfadsfasf

2.4.2 Set Process Priority

dsfasdfasfsfdf

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 with the actual 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 3 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
7:         set receiver's state to Ready
8:         if receiver has higher priority than the current process then
9:             release the processor
10:        end if
11:    end if
12: 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 preempt under any circumstances.

Algorithm 4 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. However, the sender of that message is not the timer i-process, but rather the original sender. Thus, this necessitated the creation of a new function. Again, this version is non-preemptive for the same reasons as `send_message_non_preempt()`.

Algorithm 5 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
6:         set receiver's state to Ready
7:     end if
8: end function
```

2.5.3 Receiving Messages

Receive message merely involves checking the current running process' mailbox, and seeing if there is any messages there. If there is, deque that message, and return it immediately. Otherwise, set the current process' state to blocked on receive, and release the processor.

Algorithm 6 Receive Message

```
1: function RECEIVE_MESSAGE
2:     while there are no messages in the current process' mailbox do
3:         current_process.state = BLOCKED_ON_RECEIVE
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 immediately, and indicate that there are no messages. This function is mainly used by the timer i-process.

Algorithm 7 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 8 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 9 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 10 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:         call k_release_processor()
5:     end if
6: end function
```

Algorithm 11 Main UART Interrupt Handler function (C)

```
1: function C_UART0_IRQHandler
2:     if receive data available then
3:         g_char_in = newly received char
4:         if g_char_in == null character then
5:             return
6:         end if
7:         echoMsg = get memory block (non blocking)
8:         if echoMsg is not null then
9:             echoMsg->mtype = ECHO
10:            if g_char_in == '\r' then
11:                echoMsg->mtext = "\n\r\0"
12:            else
13:                echoMsg->mtext = g_char_in + '\0'
14:            end if
15:            send echoMsg (no preemption) to KCD
16:        end if
17:        if cur_msg is null then
18:            cur_msg = get memory block (non blocking)
19:            if cur_msg is null (no more memory) then
20:                return
21:            end if
22:            msg_str_index = 0
23:        end if
24:        if g_char_in == '\r' or message about to overflow memory block then
25:            cur_msg->mtext += '\0'
26:            cur_msg->mtype = DEFAULT
27:            send cur_msg (no preemption) to KCD
28:            reset cur_msg, msg_str_index
29:        else
30:            cur_msg->mtext += g_char_in
31:        end if
32:    else if transmit interrupt enabled then
33:        if *gp_buffer != null character then
34:            send *gp_buffer
35:            gp_buffer = next location in circular buffer
36:        else
37:            disable transmit interrupt
38:            send null character
39:            reset gp_buffer and g_buffer_end
40:        end if
41:    end if
42: 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

sdfasfdafd

2.7 System Processes

2.7.1 Null Process

The null process simply releases the processor in an infinite loop.

Algorithm 12 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 13 CRT Process

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

Algorithm 14 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.

Algorithm 15 KCD Process

```
 1: function KCDPROC
 2:     identifiers = empty array
 3:     processes = empty array
 4:     numIdentifiers = 0
 5:     while true do
 6:         msg = receive_message()
 7:         if msg->mtype == KCD_REG then
 8:             add msg's identifier to identifiers
 9:             add msg's sender to processes
10:             numIdentifiers++
11:             release_message_block(msg)
12:         else if msg->mtype == ECHO then
13:             send_message(PID_CRT, msg)
14:         else
15:             if msg->mtext starts with '%' then
16:                 handler = search for handler in identifiers
17:             end if
18:             if no handler for this type of message then
19:                 release_memory_block(msg)
20:             else
21:                 send_message(handler, msg)
22:             end if
23:             if msg->mtext begins with '!' then
24:                 search through PCBs for processes in ready state
25:                 debugMsg = create message with those processes
26:                 send_message(PID_CRT, debugMsg)
27:             else if msg->mtext begins with '@' then
28:                 search through PCBs for processes in blocked state
29:                 debugMsg = create message with those processes
30:                 send_message(PID_CRT, debugMsg)
31:             else if msg->mtext begins with '#' then
32:                 search through PCBs for processes in blocked on message state
33:                 debugMsg = create message with those processes
34:                 send_message(PID_CRT, debugMsg)
35:             end if
36:         end if
37:     end while
38: 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 16 Wall Clock Process

```
 1: function wallClockProc
 2:     register itself with KCD to handle '%W' commands
 3:     id = 0
 4:     while true do
 5:         msg = receive_message()
 6:         if msg contains reset command then
 7:             id++
 8:             time = 0
 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
```

2.8.2 Set Priority Process

dsfasdfasdfadsf

2.8.3 Stress Test Processes

dfdasfasdfads

2.9 Initialization

dasfasfasfd

2.10 Testing

dfadsfasdf

2.11 Major Design Changes

dsfdafadsf

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 indepdently. 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 speciaized 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 knowledgable 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 globals 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 convoluting the output. 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 colloratively merge the parts they've worked on into a final workiing product before the deadline.

Chapter 5

Timing Analysis