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

XXXXXXXXX
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.
- `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
- `gp_stack`:
- `p_end`
- `numOfBlocks`

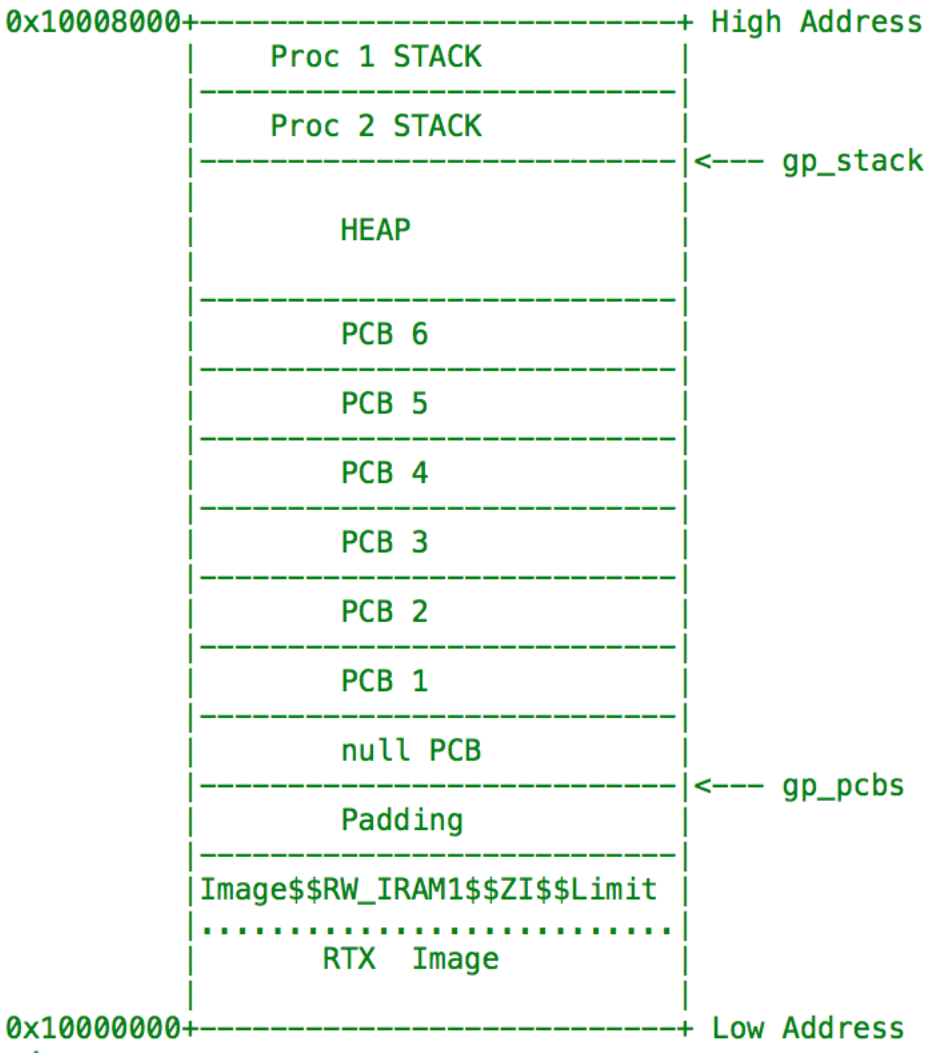

Figure 2.1: Memory Layout

2.2 Memory Management

2.2.1 Memory Structure

dsfdasfdsafdsafdsafsadfdsaf

2.2.2 Requesting Memory Blocks

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

describe input, output, effects

2.2.3 Releasing Memory Blocks

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

describe input, output, effects

Algorithm 1 k_request_memory_block

```
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
```

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
```

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

dsfadsfadsfdasfdafs

2.5.2 Sending Messages

adsfdsafasdfasf

2.5.3 Receiving Messages

dsfafasfdasf

2.5.4 Delayed Send

sdfasfasfd

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:

```
1: function UART0_IRQHandler
2:     push registers onto stack
3:     call c_UART0_IRQHandler_wrapper()
4:     pop registers off stack
5: end function
```

```
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
```

```
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
```

```
            send cur_msg (no preemption) to KCD
            reset cur_msg, msg_str_index
        else
            cur_msg->mtext += g_char_in
        end if
    else if transmit interrupt enabled then
        if *gp_buffer != null character then
            send *gp_buffer
            gp_buffer = next location in circular buffer
        else
            disable transmit interrupt
            send null character
            reset gp_buffer and g_buffer_end
        end if
    end if
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

sdfdasfafadsf

2.7.2 CRT Process

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

```
function crtProc
    while true do
        msg = receive_message()
        copy msg->mtext to output buffer
        enable_UART_transmit()
        release_memory_block(msg)
    end while
end 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.

```
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.

```
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

3.1 Source Control and Code Management

sdfdsafsadf

Chapter 4

Team Dynamics and Individual Responsibilities

4.1 adsfadsf

dfasfasdf

Chapter 5

Timing Analysis