SE 350 Lab Documentation

Group # : G106

Members:

Chuhan Xiong (20412726)

Chuqiao Shi (20444076)

Mengqi Liu (20408388)

Chunkai Yang (20409812)

08

**Fall**

Table of Contents

Preface 3

RTX Design 4

1 Processes 4

1.1 NULL process 4

1.2 User test processes (6) 4

1.3 Stress Test processes 5

1.4 System processes 5

1.5 Interrupt process (i-process ) 8

2 Data Structure (Global Variable) 10

2.1 Ready Queue 11

2.2 Memory Design 12

2.3 Blocked Queue 13

2.4 Envelope (message delivery) 14

2.5 Delayed send message & Delay Queue Structure 15

3 Procedures 16

3.1 Request Memory 16

3.2 Release Memory block 16

3.2 Context Switching 17

3.3 Send Message 17

3.5 Receive Message 18

3.6 Preemption 19

3.7 Set Process Priority 20

4 Kernel API 21

5 System initialization 23

5.1 svc\_main 23

5.2 k\_rtx\_init 23

5.3 memory\_init() 23

5.4 proc\_init() 24

5.5 UART0\_IQR\_init() and UART1\_init() 24

6.Testing 24

6.1 PART I 24

6.2 PART II 25

6.3 PART III 27

6.4 PART IV 27

BUGS & WAY TO FIX 28

7.1 Clear ‘Next’ when pop from ready queue 28

7.3 ERROR code: -1 - Wrong memory address of Heap 30

7.4 No process status check in process switch 32

7.5 Missed priority in Blocked Queue and generic Queue struct 33

7.6 Wrong heap allocation calculation in k\_memory.c 33

7.7 HARD FAULT in kernel function calls 34

Appendix A: 36

1 DATA STRUCTURE (BASIC ELEMENT): 36

A.1.1 READY\_Q 36

A.1.2 READY\_Q\_BLK 36

A.1.3 PCB 36

A.1.4 PROC\_INIT 36

A.1.5 MEM\_BLK 36

A.1.6 msgbuf 37

2 System initialization functions 37

A.2.1 svc\_main() 37

A.2.2 k\_rtx\_init() 37

A.2.3 memory\_init() 38

A.2.4 process\_init() 39

A.2.5 UART0\_IRQHandler() assembly 40

3 P4 timer testing data sheet 41

Appendix B: Hotkey illustration 42

**Table of Figures**

Figure 1 User Test output Figure 2 Wall Clock display 4

Figure 3 Wall Clock interruption 7

Figure 4 KCD Command Registration Table 7

Figure 5 UART interrupt procedure 8

Figure 6 Recommended Ready Queue Structure 11

Figure 7 Real Design of Ready Queue 12

Figure 8 Memory Model - Linked List 12

Figure 9 Wrong Blocked Queue Design 13

Figure 10 Message Envelope Structure 14

Figure 11 Message Queue Data Structure 15

Figure 12 Delay Queue Data Structure 15

Figure 13 Preemption Example 19

Figure 14 Illustration of Popping up PCB from Ready Queue 29

Figure 15 Illustration of infinite loop between Ready Queue and Blocked Queue 33

# Preface

This project is implementing a small real-time executive (RTX). The executive will provide a basic multiprogramming environment, with five priority levels, preemption, simple memory management, message-based inter-process communication, a basic timing service, system console I/O and debugging support. We have 4 parts in total:

Part I: Implementing memory management system including a linked list memory blocks and request/release memory, and process control blocks which store information about the process status.

Part II: Message queue data structure implemented. Msg send & receive functionality implemented. RTX system has the some hotkeys and wall clock now. Moreover, interrupted processes and preemption are expected.

Part III: Implements stress test processes (3 in total) and set\_prioirty\_process();

Part IV: Add a timer for speed test. Finishes up documentation.

# RTX Design

## 1 Processes

In our entire RTX project, there are 15 processes in total, including 1 NULL process (kernel), 6 user test processes, 3 stress test processes, 4 system processes (kernel) and 2 i-processes.

### 1.1 NULL process

NULL process is kernel level process. This process has the lowest priority which is 4 (Notice that NULL process is the only process that has level 4 priority and its unique PID = 0. It is particularly important that when CPU has nothing to do, i.e., all the ready queues are empty. CPU always needs to do something, so when there is no valid process in ready queue, NULL process gets to run to protect the system.

The following is how NULL process supposed to work:

void nullProc(){

while(1){

k\_release\_processor();

}

}

When nothing is in ready queue, just simply repeatedly run NULL proc which will call k\_release\_processor all time along.

### 1.2 User test processes (6)

PID 1 to 6 are corresponding to 6 different user defined processes. They are used to demonstrate the operations of RTX system. They have the following nature:

* Run at unprivileged level
* Do not assume any kernel level data structures
* Only call functions at in RTX API
* Have no idea of detailed internal design

The following diagram shows what the running result would be like on screen display should be:

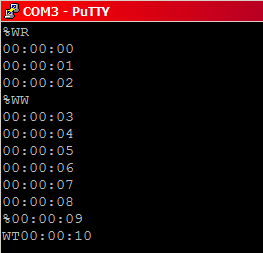
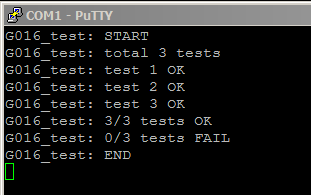


Figure User Test output Figure Wall Clock display

The left window gives the output from user process running result.

The right window shows the interrupt (show in future section) that triggers from user process.

### 1.3 Stress Test processes

The stress scenario being tested in Part III is depletion of memory blocks. Following the pseudocode provided in the manual, we have implemented process A, B, and C as the stress test processes.

Process A first register with KCD as the handler of  %Z commands. Then, when it receives a %Z command, a counter is initialized with 0 and process A enters the second loop.

When process A is scheduled to run, it requests a memory block to construct a message. The message type is set to be count\_report, and the content is the current counter value. The message is then sent to process B, and the counter is incremented by 1. Process A then releases the processor.

So everytime process A is running it sends a message to B to “report” the updated counter value.

Process B just keeps receiving messages and forward the messages to process C. It gets BLOCKED\_ON\_RECEIVE when there is no message available.

Process C first tries to receive a message. If the message type is count\_report, and the counter value is divisible by 20, then process C will “hibernate” for 10 seconds, i.e. not read/release any messages. Messages received while it hibernates will be stored in a local queue and will be processed when process C wakes up.

However, during process C’s hibernation, since no messages are read or released and A keeps requesting memory blocks, memory blocks will be depleted soon.

When memory blocks are depleted:

- A should be BLOCKED since there is no available memory block, and

- B should be BLOCKED\_ON\_RECEIVE, since A is blocked and no message is sent

- C should be BLOCKED\_ON\_RECEIVED too until the wakeup message is received. When C wakes up, it will process the messages stored in the local queue and release the message envelopes. Process A will then be scheduled to run once there is an available memory block.

### 1.4 System processes

We actually have 5 system (kernel) processes. However, NULL process is a patriotically special process that has been discussed in sec 1.1. Now let us focus on the other 4 important kernel processes: Set process priority process, Wall clock display process, KCD process, and CRT process.

#### 1.4.1 Set process priority process

The set priority command process allows user to set a process’s priority through **%C process\_id new\_priority** command.

Initially, the set priority command process registers the command %C in KCD by sending a registration message to KCD. The message has type KCD\_REG and the command as its content. Since the change in priority level should be immediate, the set priority command process has a priority of HIGH.

So when KCD receives a keyboard input starting with %C, a message contains the input command will then be sent to the set priority command process to handle.

Once the set priority command process receives a command message, it will validate the command format. If the command format follows %C int int, the process will call the kernel function k\_set\_process\_priority through SVC, and pass over the process\_id and new\_priority.

Part of pseudo code will be provided in section xxxxxx to show how this process works.

#### 1.4.2 Wall clock display process

The wall clock display process registers itself with the Keyboard Command Decoder (KCD) process as the command handler for the %W commands:

* The %WR command will reset the current wall clock time to 00:00:00, starts the clock update display time every second.
* The %WS hh:mm:ss command sets the current wall clock time to hh:mm:ss, starts the clock update display time every second.
* The %WT command will terminate the wall clock display.

To correctly handle these commands, the wall clock responds to two type of messages:

##### Keyboard command input(COMMAND) The command messages come from the KCD process. When the keyboard input string is %WS, %WR, or %WT, KCD will identify it as valid command identifier and will create a message with type COMMAND and the command as the message content, and send it to the wall clock process, which is the handler for the command.

In wall clock process, we have

- a variable “time” to keep track of the current display time in seconds.   
- a flag “terminate” to indicate whether the wall clock is terminated or not.

For commmands

%WS - time = the input time converted to seconds; terminate = 0; And start updating the display

%WR - time = 0; terminate = 0; And start updating the display

%WT - time = 0; terminate = 1; And stop updating the display

##### Wall clock update message (WALL\_CLK\_INCREMENT)

To update the wall clock display every second, starting from time 00:00:00 or the set time, wall clock process sends a 1000ms delayed message with type WALL\_CLK\_INCREMENT to itself every time the display is updated.

So when a message with type WALL\_CLK\_INCREMENT is received, the variable “time” will be incremented by 1, and a display message with the new time is sent to CRT display the updated time.

The following diagram illustrates the process:

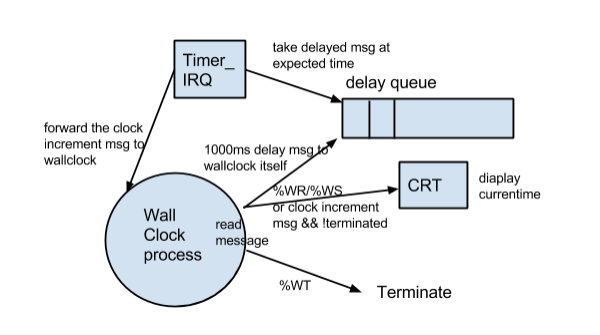


Figure Wall Clock interruption

#### 1.4.3 KCD (Keyboard Command Decoder) System Process

A keyboard command starts with the prompt character %, followed by a character string command identifier and possibly additional command data.

The command decoder process responds to two types of messages:

##### Command Registration (KCD\_REG)

Process that has input commands associated with needs to register the commands with KCD. So when the process is initially scheduled to run, it will send messages with type KCD\_REG to register the commands.

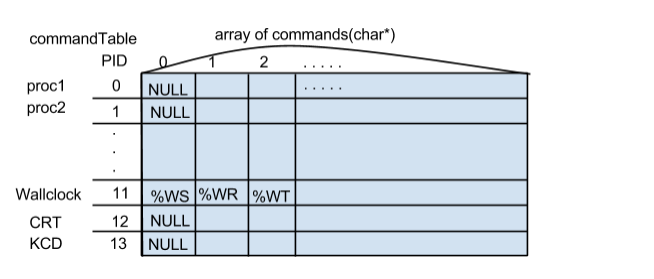
When KCD receives the registration message, it will store the command in a registered-command table. The command table is implemented using an 2-dimensional array, where the process ids are mapped to the indices of the array, and the registered commands are stored as an array of char\* at the corresponding process’s index.  


Figure KCD Command Registration Table

##### Keyboard Input (DEFAULT)

##### The string input is sent to CRT display for output.

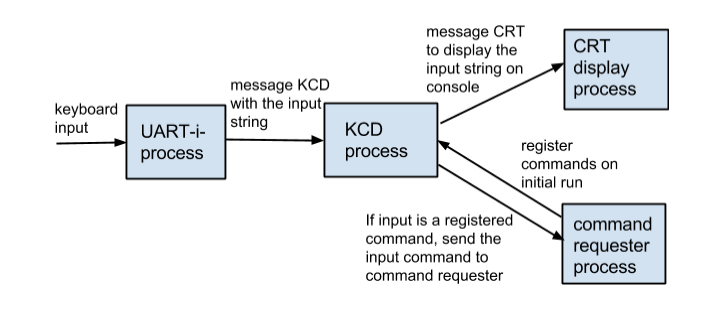


Figure UART interrupt procedure

Since UART-i-process handles keyboard input interrupt, every input character will cause an interrupt. So the UART-i-process sends the keyboard input to Keyboard Command Decoder (KCD) character by character.

KCD takes the char input and check: if it is a prompt command char ‘%’, KCD adds the char to a buffer that will hold further command input chars until the return button is hit, i.e. a newline ‘\n’ character is read.

When the return character ‘\n’ is received, KCD will process the buffer by searching through the command table. If it is a valid command identifier, the command will be forwarded to the command handler. A display message is sent to CRT every time to display the char on console.

Otherwise, KCD does not keep track of the input char. A display message is also sent to CRT.

#### 1.4.4 CRT process

The CRT display process responds to only one message type: a CRT display request.  The message type is set to CRT\_DIS, and the message body contains the character string to be displayed. The process will output the string to the console CRT. It uses the UART i-process to print to the console display. Any message received is freed using the release\_memory\_block primitive.

### 1.5 Interrupt process (i-process )

I-process is an aberration of interrupt processes. In RTX project, there are two required i-processes: Timer I-process and UART I-process.

#### 1.5.1 Timer I-process

The timer i-process works as what its name indicates. This interruption will be needed every time a hardware interruption occurs. Besides, a timer self-interruption needs to be triggered every 1 ms for handling the delivery of delayed send message after the required time expired. For example, process A has sent a message M to process B with delayed time T. All messages have delayed characteristic will be firstly sent to a “deleyed\_queue”. This queue is sorted by sent time. What timer will do is to do a self-trigger every 1 millisecond, then decide if the first element in delayed\_queue is time expired. If yes, the timer i-process will now take over the OS running routine – removes the time-expired message from deleayed\_queue and send it to its destination.

#### 1.5.2 UART I-process

UART I-process does a lot of things. The most important part of its functionality can be described as two subroutines: send UART to KCD && CDT with Wall Clock.

The second part, CDT and wall clock, is not necessary related to UART I-process interruption. However, this part is immediately triggered if any UART interruption exists in OS.

##### UART to KCD

UART\_i\_process is a kind of process that deals with user keyboard interrupt. Once user press a key on keyboard, an interruption occurs. Therefore, the state of UART\_i\_process is changed from waiting to running which means an interruption happens. Next, UART will call UART\_I(void) function (implemented in k\_process.c). This UART\_I function will do the following checking:

1. Check if the hotkeys\_debug\_flag set to be on or not.

2. Check if the input is 1 of 5 hotkeys. If it is a hot key and the flag above is turned on, this interruption will be treated as a predefined hotkey. If not, the RTX will enter the file k\_memory.c to do i\_request\_memory\_blk. This memory request function will only return null(no enough space) or a valid memory block. (Note that i\_request\_memory\_blk is a non\_blocking function, i.e, the current UART\_i\_process will never be blocked due to any memory issue. If a valid memory block is provided, it needs to be casted to a msgbuf type for further message sending. In the casted msgbuf, msg\_type is predefined and its body contains input char and “\0”(end of string). i\_process sends out this well-strutted message using i\_send\_msg to KCD.(Note that i\_send\_msg is has no preemption).

##### 1.5.2.1 Hotkey List

In our PTX P2 design, we have 5 hotkeys in total. This is the hotkey list and its corresponding functionality.

! (shift+1): Display current Ready Queue;

@ ( shift+2): Display current Blocked Queue;

# ( shift+3) : Display all the processes that are currently at the state blocked on receice. (Trying to read message but no message in its message queue yet);

$ ( shift+4): Display message send history, with PID, destination, time send, interrupt, type, and message content displaying together;

% (shift+5) : Display message receive history, with PID, destination, time send, interrupt, type, and message content displaying together.

The picture in Appendix B is a screen shot of pressing hotkey with some user-defined messages that are doing send & receive.

##### Wall clock with KCD and CRT

Frist of all, we need to identify the priorities of KCD, CRT and Wall Clock.

CRT – High priority

KCD – medium priority

Wall clock – low priority

At very beginning, CRT starts to run and tries to receive messages. However, no messages come yet and it becomes BLOCKED\_ON\_RECEIVE.

Then it is time for KCD to run. KCD will do two things: 1. It initializes a command registration table 2. KCD will also try to receive message as what CRT did, but it also gets BLOCKED\_ON\_RECEIVE.

Since wall clock process has higher priority than other user processes, it will be scheduled to run next. The process will send registration messages to KCD to register all its commands. Wall clock then gets preempted and KCD will process the registration messages, store them into the command table. After all registration messages are read, KCD will get BLOCKED\_ON\_RECEIVE again.

Now consider the case that we have a user input from keyboard. The entire procedure in 2.3.7 now should come into place. After KCD gets first char from UART\_i\_process, KCD should become unblocked and be moved to Ready Queue. Since CRT is still blocked, then it’s KCD’s turn to run (current highest priority process on Ready Queue, interrupts always have higher priority than any other processes).

KCD takes the char input and check: if it is a normal letter like ‘a’, KCD simply send a display message to CRT to display this char on console; otherwise, if it is a ‘%’ sign, KCD adds the char to a buffer that will hold further command input chars until the return button is hit, i.e. a newline ‘\n’ character is read.

When the return character ‘\n’ is received, KCD will process the buffer. If it’s a valid command identifier, the command will be forwarded to the command handler.

## 2 Data Structure (Global Variable)

In this part, we will briefly explain our design of data structures that are used as container.

### 2.1 Ready Queue

#### 2.1.1 Original Design based on tutorial slides

At the beginning, we tried to follow the pattern that TA offers in the tutorial lab slides, as illustrated in the figure below.

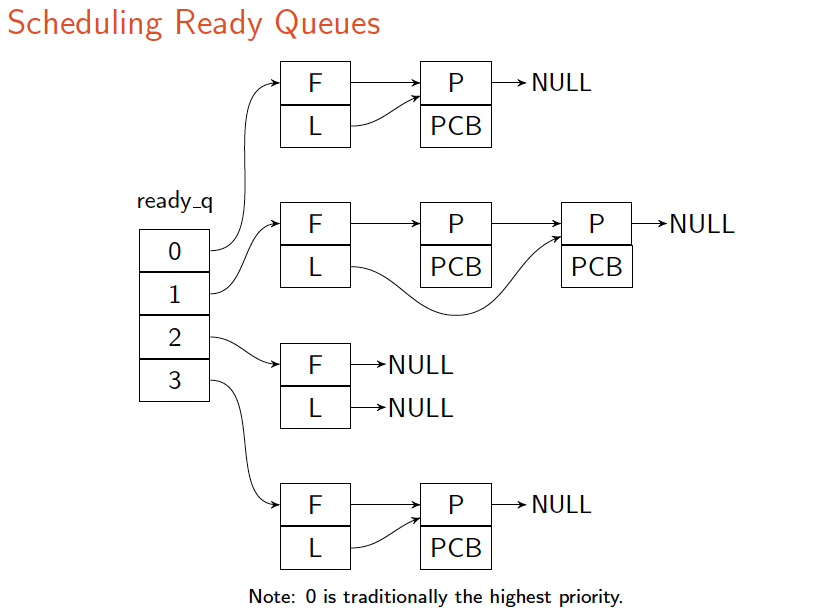


Figure Recommended Ready Queue Structure

In this case, we designed READY\_Q (Appendix A.1.1) ready\_q as an array with four elements in it, indexed as 0, 1, 2, 3. Each of these array elements is user-defined structure called READY\_Q\_BLK, this structure was originally defined in A.1.2.

By using this design, we can initialize the value in ready\_q as NULL before any processes needs to be placed into ready queue structure. Every time when a new PCB is inserted into ready\_q, the first step is to check the priority within the structure of PCB. The structure of PCB is defined as in Appendix A.1.3.

The PCBs will be inserted into ready\_q according to its priority. In this case, every array element in ready\_q is a linked list with pointer type READY\_Q\_BLK. In this linked list, every READY\_Q\_BLK node has a next pointer with same type points to the next element in the linked list.

Overall, the design stated above is tightly following the instructions from tutorial slides and diagram shown above.

#### 2.1.2 Real look

In our new Design, we decided to use an array of linked list to store all the information that we need for a priority queue.

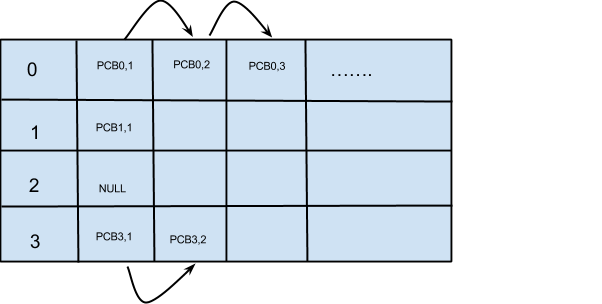


Figure Real Design of Ready Queue

The illustration above is our new design 1. This structure is basically a 2-D Array. The vertical indices indicate the priority. The horizontal ones store the PCBs according to their priority information in the struct. In this case, no linked list is required. However, we change the struct of PCB to what we have in Appendix A.1.3.

Compared to the previous structure that we had for PCB, we simply resume the field of:

struct pcb \*mp\_next;

So even though this 2-D array ready\_q is not implemented with linked list embedded, we can still use this \*mp\_next pointer points to next PCB component. That is, ALL PCBs are still linked together. Therefore, when scheduler is be triggered by process\_switch(PCB\*), and this functionality is being called from k\_process.c by k\_process\_release().

### 2.2 Memory Design

According to TA’s slides and MEM\_BLK data structure (Appendix A.1.5) which has a pointer that is able to point to the next element in linked list data structure, our basic design looks exactly the same as the following diagram:

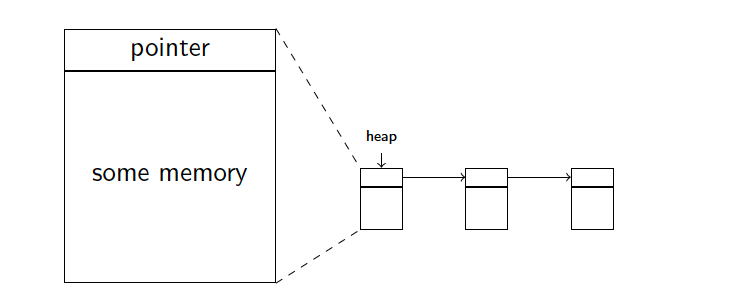


Figure Memory Model - Linked List

For this Linked List , we construct it as:

current = gp\_heap;

for (i = 0; i < MEM\_BLK\_NUM; i++){

curr\_mem\_blk = (MEM\_BLK\*)current;

curr\_mem\_blk->isUsed = 0;

current += sizeof(MEM\_BLK);

current += (MEM\_SIZE/4);

if (i != MEM\_BLK\_NUM - 1){

curr\_mem\_blk->next = (MEM\_BLK \*)current;

}

else{

curr\_mem\_blk->next = NULL;

}

}

We first declare that we could only have 32 fixed number of MEM\_BLK\_NUM

# define MEM\_BLK\_NUM 32

# define MEM\_SIZE 128

First, we need go back to the place of gp\_heap where the address of linked list of MEM\_BLK starts. Then we manually add 32 MEM\_BLKs and link them together by setting proper next pointer. The last element in this list HAS to be set to points to NULL.

### 2.3 Blocked Queue

#### 2.3.1 Wrong design without priority involved

Blocked queue (PCB\* blockedQueue) holds the processes that are currently blocked due to lack of memory blocks. It is a simple design with a singular linked list with each single element has a type of PCB\*.

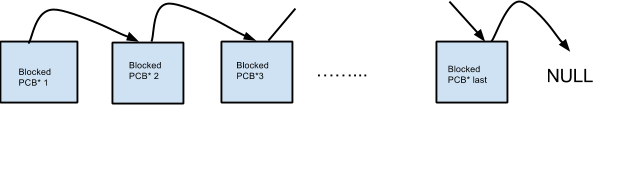


Figure Wrong Blocked Queue Design

When some available memory blocks has been detected, i.e. some other process (es) have(s) released the memory blocks that they have, the first element of this blockedQueue will be popped and insert to ready\_q.

As long as there is any available memory block, the first element will then be added back to ready\_q. If the number of available blocks is large enough to process the newly released one, then this process will be popped from ready\_q after finishing the procedure. If the number is not large enough, this process will be added to the last of blockedQueue again.

#### 2.3.2 Expected design with priority setting

As we state later, we remodel our blocked queue by using generic template same as ready queue. See more information in section 4.5.

Actually after we realize that ready\_queue, blocked\_queue, and message\_queue are all done by priority queue, we use a generic template which adapts for all structure that has priority involved.

Some more useful information might be found in section 7.5

### 2.4 Envelope (message delivery)

Messages are carried in “envelopes”. An envelope contains a header and a “msgbuf”. For part II, we defined the structure MEM\_BLK to be the fix-sized header.

So when there is a message, we request a memory block to store the msgbuf, which contains the message type and content. The header will store the next pointer, sender\_id, receiver\_id and the time for message to be sent.

Since only the msgbuf is being sent and received, in order to retrieve the header to set sender\_id, receiver\_id, send time and the next pointer, we defined a function get\_memory\_blk\_from\_message(\*msgbuf), which returns a pointer pointing to the beginning of the header..

The “envelope”, header, and the msgbuf structures are illustrated as the following:

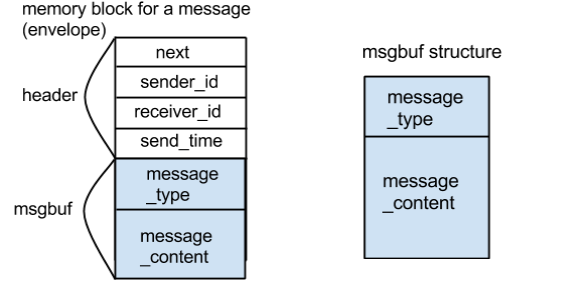


Figure Message Envelope Structure

Each receiver, a PCB has a **message queue** to store the messages sent to it. The message queue is implemented as a linked list of “envelopes”. Each envelope has its next pointer pointing to the next envelope added to the queue.

So a new field **msgqueue**, a pointer pointing to the head of the envelope linked list, is defined in the PCB structure.

The message queue linked list is illustrated as the following:

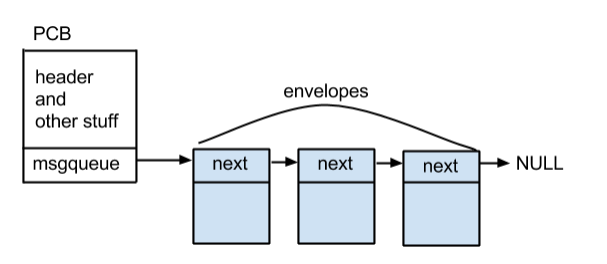


Figure Message Queue Data Structure

When the sender calls send\_message(sender\_id, \*msgbuf), the new message envelope is added to the end of the msgqueue. When the receiver calls receive\_message(), the first envelope in the msgqueue is dequeued, and the msgbuf in the envelope is returned to the receiver.

So the receiver only has access to the message type and message content, the header of the envelope (next pointer, sender\_id, receiver\_id, send time) will not be exposed to the receiver.

After the message is received, in order to release the entire memory block (the envelope), a function is defined. It returns a pointer to the memory block, which contains both the header and the msgbuf. Thus, release memory will release the entire memory block.

### 2.5 Delayed send message & Delay Queue Structure

The timer-i-process receives messages to deliver after a delay.

So when delayed\_send is used to send a message, the message envelope will be put into a delay queue. The timer-i-process will constantly check the delay queue: if a message’s delay time has expired, i.e. the expected sending time of the process is reached, the timer-i-process will then forward the message to the receiver.

The delay queue is implemented as a linked list of envelopes, sorted by the expected sending time of the messages (which is the send time of sender + delay time).

The delay queue structure is illustrated as the following:

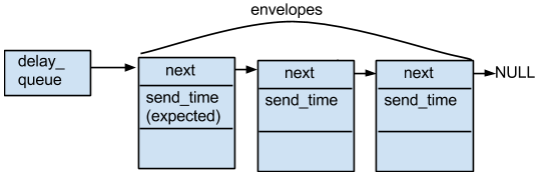


Figure Delay Queue Data Structure

When inserting a new delayed message envelope, we will find the proper position for the envelope to make the linked list remain sorted by the expected sending times of the messages.

While a message is delayed send to the receiver, if the receiving process has a higher priority than the sending process, then preemption happens. Otherwise the sender continues executing. Preemption is further explained in the preemption section.

## 3 Procedures

### 3.1 Request Memory

The following is the pseudo code for request memory blocks:

Int k\_request\_memory\_block(){

Atomic(on); //no interruption allowed

While (no memory block available){

Put pcb on blocked\_queue;

Set process to be in blocked state;

Release\_processor();

}

Int mem\_blk = next free blk;

Update heap pointer address;

Atomic(off);//interrupt enabled

Return mem\_blk;

}

### 3.2 Release Memory block

The following is the pseudo code for release memory block

Int k\_release\_memory\_block( void \*memory\_block){

Atomic(on);//no interruption allowed

If( memory block pointer is invalid){

Return error; //you cannot release any invalid memory block

}

Put memory\_block into heap;

Update heap pointer address;

If (if there are processes being blocked)

{

//release the last one in the blocked queue

handle\_process\_ready();// adds a pcb just removed from the blocked queue

push to ready queue;

release the processor;

}

Atomic(off);//external interruption (timer or hardware now acceptable)

}

### 3.3 Context Switching

Context switching switches out the current running process (pcb\_old) and run the new process (new\_process). The parameter pcb\_old is currently in state RUN. The new process is selected by the scheduler to run next.

The context switching procedure returns RTX\_OK upon success and RTX\_ERR upon failure.

int process\_switch(PCB \*pcb\_old)  {

if (new\_process->state is NEW) {

if (new\_process != pcb\_old && pcb\_old != NULL &&pcb\_old->state != NEW) {

save context of pcb\_old;

if (pcb\_old->state != BLOCKED && pcb\_old->state !=  BLOCKED\_ON\_RECEIVE) {

pcb\_old->m\_state = READY;

}

}

update current\_process = new\_process;s

new\_process->state = RUN;

restore context of current\_process;’

execute current\_process;

atomic\_off(); //adding atomic off here since rtc jumps straight to the new process without effectively return

pop exception stack frame from the stack for a new processes;

}

/\* The following will only execute if the if block above is FALSE \*/

if (new\_process != pcb\_old) {

if (new\_process->state == READY && pcb\_old != NULL){

save context of pcb\_old;

if (pcb\_old->state != BLOCKED && pcb\_old->state != BLOCKED\_ON\_RECEIVE){

set pcb\_old->state = READY;

}

new\_process->state = RUN;

restore context of current\_process;

execute current\_process;

}

else {

set current\_process = pcb\_old; // revert back to the old proc on error

return RTX\_ERR;

}

}

return RTX\_OK;

}

### 3.4 Send Message

The entire RTX supports a message-based interposes, that it, a process can send message to another process. All messaged are decorated in envelopes with a head each.

Data structure of envelope can be found in section 2.4

#### 3.4.1 Direct message sending

The direct message sending method send\_message(sender\_id, \*msgbuf) will the add new message to the msgqueue of the receiver process.

The pseudo code looks like the following:

void k\_send\_message(int process\_id, void \*message){

atomic\_on();

enqueue message onto the msg\_queue of receiving\_proc;

if(receiving\_proc->state  is  BLOCKED\_ON\_RECEIVE){

set receiving\_proc->state to READY;

push receiving\_proc to ready\_queue;

// if preemption happens

if (receiving\_proc->priority is higher than current\_process->priority){

k\_release\_processor();// current process is preempted

}

}

atomic\_off();

}

#### 3.4.2 Delayed message sending

The delayed\_send is used to send a delayed message. The message will be put into a delay queue, and the timer-i-process will constantly check the delay queue: if a message’s delay time has expired, i.e. the expected sending time of the process is reached, the timer-i-process will then forward the message to the receiver.

The pseudo code for delayed\_send is as the following:

void k\_delayed\_send(int process\_id, void \*message, int delay){

atomic\_on();

MEM\_BLK\* msg\_envelope = get\_msg\_envelope(message);

msg\_envelope->sender\_id = current\_process->pid;

msg\_envelope->destination\_id = process\_id;

msg\_envelope->expectedSendTime = systemTime + delay;

msg\_envelope->nextMsg = NULL;

add message to delay\_queue;

ato\_off();

}

### 3.5 Receive Message

When the receiver calls receive\_message(), the first envelope in the msgqueue is dequeued, and the msgbuf in the envelope is returned to the receiver.

So the receiver only has access to the message type and message content, the header of the envelope (next pointer, sender\_id, receiver\_id, send time) will not be exposed to the receiver.

void\* k\_receive\_message(int \*sender\_id){

atomic\_on();

PCB\* current\_process = getcurrentProcess();

while (current\_process->msg\_queue is NULL) {

set  current\_process->m\_state  to  BLOCKED\_ON\_RECEIVE;

k\_release\_processor();

}

//Dequeue

MSG\_BUF\* msg = dequeue the first msg from current\_process->msg\_queue;

ato\_off();

return msg;

}

### 3.6 Preemption

Generally speaking, preemption is a scenario that an interruption (might be caused by keyboard or any hardware input or timer) happens when the process A is running but when the interruption ends the program starts to run process B which must have a higher priority than A.

We can explain this procedure more precise by using the diagram below.

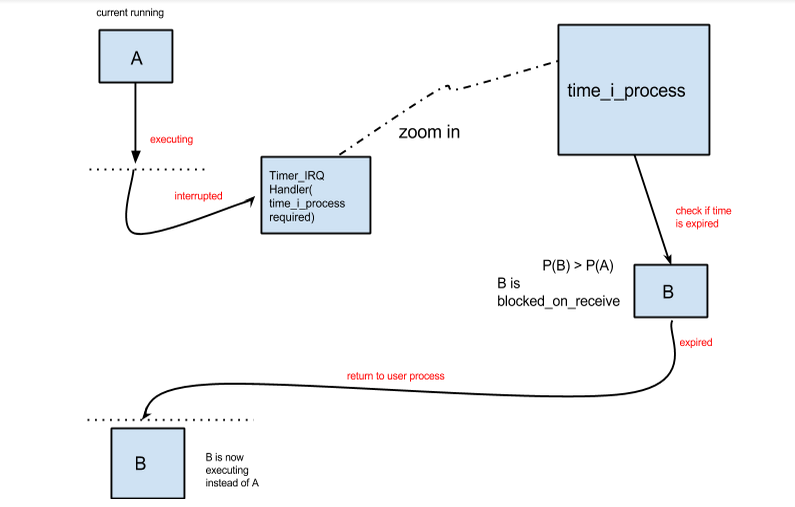


Figure Preemption Example

In the above diagram, we firstly have a current running process A with its priority labeled as P(A). When it is still executing, an interruption happens and the current routine is now switching to interruption routine. In interruption routine, Timer\_IQRHandler(in our diagram the interruption is caused by a timer) is handling the interruption.

Inside this Timer\_IRQHandler, a process called timer\_i\_process is required. To this i\_process, it will check if there is any other process that is current blocked\_on\_receive(might be as the result of lack of resource). If any, check the priority of them and compared with current process A. If process still have the highest priority, then none of those blocked\_on\_receive processes will be considered to run instead of A after the interruption routine. However, if any of those blocked\_on\_receive processes B has higher priority than A then the scheduler will be called and context\_switch will be involved. In this case, if the interruption finishes, process A should be stopped while process B starts running. We also call this as “ A is preempted by B” or “ B preempts A.

### 3.7 Set Process Priority

Most part of this procedure has been stated in section 1.4.1 since it will be called by set priority process. Particularly in our RTX, the kernel function k\_set\_process\_priority will first check if the process is not NULL process and the process\_id and new\_priority are valid, i.e. within range, then do the following:

int k\_set\_process\_priority(int process\_id, int priority){

atomic\_on();

if (process\_id == PID\_NULL && priority != LOWEST

|| process\_id != PID\_NULL && priority is out of range

|| (process\_id == PID\_TIMER\_IPROC

|| process\_id == PID\_UART\_IPROC)

){

ato\_off();

return RTX\_ERR;

}

if (process is in READY or NEW state) {

erase it from the ready\_queue

set the process priority to be new\_priority

insert it back into ready\_queue with the new priority set

}

else if (process is in BLOCKED state) {

erase it from the blocked\_queue

set the process priority to be new\_priority

insert it back into blocked\_queue with the new priority set

}

else if (process is in RUN or BLOCKED\_ON\_RECEIVE state) {

set the process priority to be new\_priority

}

k\_release\_processor();

atomic\_off();

return RTX\_OK;

}

Thus, the target process is set to the new priority, and it is placed in the correct place in ready\_queue or blocked\_queue based on its new priority.

After so, an immediate call to context\_switching() might be fatal, however, this depends on the priority that we just set. Context\_switching() is badly needed if pre-emption occurs. For example, we are currently have Process A in medium priority . Process B at this time is set to have a high priority. It appears that pre-emption happens and we need to pause A and get to run Process B. In this case, we badly need to make a call to context\_switching().

## 4 Kernel API

We have systematically well-designed API for all user functions to access any kernel functions in rtx.h file.

The following is part of is most important API:

/\* Processor Management \*/

extern int k\_release\_processor(void);

#define release\_processor() \_release\_processor((U32)k\_release\_processor)

extern int \_\_SVC\_0 \_release\_processor(U32 p\_func);

extern int k\_get\_process\_priority(int pid);

#define get\_process\_priority(pid) \_get\_process\_priority((U32)k\_get\_process\_priority, pid)

extern int \_get\_process\_priority(U32 p\_func, int pid) \_\_SVC\_0;

extern int k\_set\_process\_priority(int pid, int prio);

#define set\_process\_priority(pid, prio) \_set\_process\_priority((U32)k\_set\_process\_priority, pid, prio)

extern int \_set\_process\_priority(U32 p\_func, int pid, int prio) \_\_SVC\_0;

/\* Memeory Management \*/

extern void \*k\_request\_memory\_block(void);

#define request\_memory\_block() \_request\_memory\_block((U32)k\_request\_memory\_block)

extern void \*\_request\_memory\_block(U32 p\_func) \_\_SVC\_0;

extern int k\_release\_memory\_block(void \*);

#define release\_memory\_block(p\_mem\_blk) \_release\_memory\_block((U32)k\_release\_memory\_block, p\_mem\_blk)

extern int \_release\_memory\_block(U32 p\_func, void \*p\_mem\_blk) \_\_SVC\_0;

/\* IPC Management \*/

extern int k\_send\_message(int pid, void \*p\_msg);

#define send\_message(pid, p\_msg) \_send\_message((U32)k\_send\_message, pid, p\_msg)

extern int \_send\_message(U32 p\_func, int pid, void \*p\_msg) \_\_SVC\_0;

extern void \*k\_receive\_message(int \*p\_pid);

#define receive\_message(p\_pid) \_receive\_message((U32)k\_receive\_message, p\_pid)

extern void \*\_receive\_message(U32 p\_func, void \*p\_pid) \_\_SVC\_0;

/\* Timing Service \*/

extern int k\_delayed\_send(int pid, void \*p\_msg, int delay);

#define delayed\_send(pid, p\_msg, delay) \_delayed\_send((U32)k\_delayed\_send, pid, p\_msg, delay)

extern int \_delayed\_send(U32 p\_func, int pid, void \*p\_msg, int delay) \_\_SVC\_0;

K\_release\_processor: This function call leads to the calling process voluntarily releases the processor. We need to use this function every time when we want to release the processor, especially when deal with priority queue and context\_switching and now is the time to take the next process, k\_release\_processor will be called. Also, if our OS is idle and have a empty ready queue, k\_release\_processor will also be call to run NULL process to protect our system.

K\_get\_process\_priorty: This function call returns the process priority by taking PID as input parameter. Usually triggered by user processes.

K\_set\_process\_priority: This function call might be triggered by set\_process\_priority process. This function simply changes the current priority to input priority of input PID. This function will most likely lead to preemption.

K\_request\_memory\_block: This function call is made when any process is requesting any more memory blocks. One particular usage is doing process initialization for user process.

K\_release\_memory\_block: This function call is made when any process needs to ‘clean up’ itself. This has to be done manually. For this functionality, you may also want to clean up everything that used to store in the memory block which is releasing.

K\_send\_message: This function call is made when there is a message that need to be send to its destination. This functionality is using almost everywhere in UART-i-process and timer interruption since every time when there is an interruption (either from timer or hardware) detected, a message sending is required.

K\_receive\_message: This function call is made when a process needs to check if its received\_queue has any message there. If yes, this process will take the message info and deal with the message that obtained from receive\_queue; if not, the receiving process will continue on its current procedure. For this function, we need to remove the received message from received\_queue once receiving process gets it.

K\_delay\_send : This function call is made if any delay\_send is required. A delay send means that when process A send a message to its destination process B, the procedure is delayed for certain time (same as the number that specified in function parameter). Particularly, no delay when A is sending out the message, but one more delayed queue is required so if this is a delay send, all message will be send to delayed queue first. When the certain time expired, the message in delay queue with highest priority will then be sent to its destination. For this process, we need to remove the delayed message from delayed\_queue when its delay time expires.

## 5 System initialization

### 5.1 svc\_main

In order to trigger our RTX, we have an entry point in main\_svc.c file to start the system. All other variables or processes that need to be initialized will be triggered by possible subroutines.

For a successful svc\_main function call (see Appendix A.2.1), we will be directed to rtx\_init(); If any error occurs, function should return RTX\_ERR; After the execution of 5.2-5.5 is fully finished, a call to k\_release\_process() is made so the kernel function call to scheduler and context\_switching will be made soon. Therefore, the entire RTX should start.

### 5.2 k\_rtx\_init

As a result of function call from svc\_main: rtx()\_init, void k\_rtx\_init(void) in k\_rtx\_init.c (see Appendix B.2.2) will execute.

A global timer is set here for delayed queue message queue.

Since this is the most important part of system initialization, all external interrupt are disabled by \_\_disable\_irq().

In here we basically make a function call to memory initialization and process initialization respectively for further initialization. After everything is proper settled, k\_release\_processor() takes turn to run therefore a function call to scheduler and context\_switching() is made and entire RTX now starts to work.

### 5.3 memory\_init()

Memory\_init() has is a little bit longer but we can briefly group it into separate steps:

1. Create an array of PCB\* on main memory (using gb\_pcbs to find address)

2. Create separate spaces for each single PCB on main memory (using gp\_heap to changing the address, growing number)

3. Allocate everything that we need for each process on stack (using gp\_stack to store information and address keeps drowning downwards)

4. Allocate heaps on RAM to store our 32 memory blocks.

In this part, we need to worry about pointer arithmetic

### 5.4 proc\_init()

Immediately after memory initialization finishes, proc\_init() (code is available at Appendix 2.4) is ready to run.

In proc\_init() process, notice that we have available memory blocks from memory initialization. Here we have a global g\_proc\_table that holds all processes.

The following are basically we are doing in this function:

1. Set NULL process

2. Set test processes1-6

3. Set 3 stress tests

4. Set system process

5. Initialize exception frame stack for all process

6. Add/Push all processes to ready queue

So-called set means you need to clear contents in process, find proper memory address for it, set priority for each of them, etc,

### 5.5 UART0\_IQR\_init() and UART1\_init()

UART0\_IQR is an interrupt driven procedure that can interact with CRT and KCD while UART1 polling is designed for debugging only.

For UART0\_IQR\_init() the major part that we need to concern with is the hardware configuration since we need the capability to handle with hardware interrupt from ARM board.

IMPORTANT: UART0\_IQR handler should be implementation in assembly language. Therefore, accessing c symbol from assembly is also required.

An example of \_\_asmvoid UART0\_IRQhandler is provided in Appendix A.2.5.

For UART1\_init(), it also sets up a lot of system configuration. It supports the functionality of polling UART to send and receive data.

# 6.Testing

### 6.1 PART I

The following are the possible test cases that we tried for debugging purpose. Numbers might vary.

Priorities of the processes: Process 1 & 2 - HIGH, 3 & 4 = LOW, 5 & 6 = LOWEST

#### 6.1.1 Test 1: Process gets blocked when there is no enough memory blocks available

- Process 1 requests 20 blocks of memory (32 memory blocks in total) then release processor

- Process 2 requests 12 blocks of memory then release processor

- Process 1 requests 20 blocks of memory and gets blocked

- Process 2 requests 12 blocks of memory and gets blocked

- Process 3 gets processed

#### 6.1.2 Test 2: Blocked process gets ready when there are memory blocks released

Continued from test 1

- Process 1 then releases 1 blocks of memory

Since process 1 is the first process inserted into the block queue, when there are memory blocks available, it gets out of the block queue first and goes back into ready queue. But process 1 requests 20 blocks to run, still not enough available memory blocks, process 1 get blocked again. It should then be added to the end of blocked queue.

- Then process 1 requests 5 blocks of memory

The function k\_release\_memory\_blocks() will be called 5 times. And then the first 5 elements in blocked queue will be re-added to ready\_q based on their priorities. Scheduler will continuously deal with ready\_q and re-block the process that requires too much resources.

#### 6.1.3 Test 3: NULL process is executed when no other ready processes

- Process 1 requests all the 32 blocks of memory and then release processor

- Process 2 then gets blocked since there is no memory available. Following that process 1, 3, 4, 5 are all blocked

- When all processes get blocked, the NULL process is reached and keeps running until there are memory blocks released by process 1

- Program should not crush when only NULL process is running

### 6.2 PART II

#### 6.2.1 Test Data of memory address for PCB and envelope

For PART II, we have 16 PCBs in memory. The following data table is what we used to keep take in memory location.

Note that the specific memory location on different local machine may different from each other. However, we should note the difference between two nearby PID on memory should be fixed size.

|  |  |
| --- | --- |
| PID | PCB memory location |
| 0 | 0x0B64 |
| 1 | 0x0B80 |
| 2 | 0x0B9C |
| 3 | 0x0BB8 |
| 4 | 0x0BD4 |
| 5 | 0x0BF0 |
| 6 | 0x0C0C |
| 7 | 0x0C28 |
| 8 | 0x0C44 |
| 9 | 0x0C60 |
| 10 | 0x0C7C |
| 11 | 0x0C98 |
| 12 | 0x0CB4 |
| 13 | 0x0CD0 |
| 14 | 0x0CEC |
| 15 | 0x0C08 |

From this table, we should be aware of that the address difference between each nearby two PCB is 1C = 28 byte = the space of 7 pointers. In our PCB design for PART II, we have 7 fields in total, which perfectly make sense here.

If the project is switched to another machine which has a different start memory address, the difference should still be 7\*4 = 28 bytes = 1c as long as our PCB structure remains the same.

The second table is used for testing memory address of message buf(envelope) structure.

Mem: Stack = 0x01000800, number keeps decreasing if anything is pushed onto stack

Heap = 0x01000D28, number keeps increasing if anything is pushed onto heap

**Possible Memory Location**

|  |
| --- |
|  |

|  |  |
| --- | --- |
| MEM\_BLK\*(Message header) | Message body, 128byte space |
| 0x0D28 | 0x0D40 |
| 0x0DC0 | 0x0DD8 |
| 0x0E58 | 0x0E70 |
| 0x0EF0 | 0x0F08 |

Recall our envelope structure from Figure 5, we have a 24 byte message header with manually casted to type MEM\_BLK\* and a 128 byte message body.

First of all we notice that the difference between next MEM\_BLK\* and next message body is a FIXED SIZE 98 represents 152 bytes. This is the total size of one valid envelope. Next thing is that in the same iteration, difference between header and body is always fixed as 0x18 = 24 bytes which is the size of one valid header.  The difference between previous message body and next message header is always fixed as 0x80 which is 128 bytes. This is the size for one valid message body.

Note that the start address may vary from different local machine.

### 6.3 PART III

This part is focusing on stress test processes and information can be found in section 1.3

### 6.4 PART IV

No special test requires. After a new timer is introduced and all the tests from p1 to p3 all pass successfully, then this RTX should work perfectly.

For the timer testing in P4, we get the following data sheet (see Appendix 3) and summary.

From the calculations above, we can derive the average time cost for the primitives:

request\_memory\_block: 5.623 microseconds

send\_message: 8.895 microseconds

receive\_message: 5.357 microseconds

release\_memory\_block: 9.347 microseconds

get\_process\_priority: 1.580 microseconds

release\_processor: 4.670 microseconds

All the measurements are made with the following conditions to ensure accuracy:

1. The measuring process keeps running for the entirety of the measurement. No process switch or preemption takes place during measurement.
2. Uart0 interrupt is not triggered during measurement..
3. Uart1 polling is not called during measurement.
4. Timer0 can interrupt, but no preemption will take place. The delayed-message queue is always kept empty to minimize the run-time of timer0\_irq\_handler.
5. Timer1 only increases time when the measuring process is running the primitive being tested.
6. For those primitives whose performance can be impacted by the length of the linked lists, all cases where the length of the list is between 0 and 30 are covered.

We think the values measured are reasonable. We specifically measured the speed of the get\_process\_priority primitive as a reference. It is an extremely simple primitive with only a few lines of code, so its run-time should be dominated by the cost of the SVC, this gives us an approximation of the cost of SVC.

Request\_memory\_block, receive\_message, and release\_processor all cost around 5 microseconds on average. We find this value reasonable. The additional cost besides the cost of SVC call is resulted from probing the linked lists.

Interestingly, send\_message and release\_memory\_block both cost around 9 microseconds, significantly longer than other measured values. After reviewing our code, we found reasonable explanations. For send\_message, the additional cost comes from the inefficient linked-list probing. Our code needs to go through the message queue when adding a new message. Since the “next” pointer is stored in the memory block header instead of the message body, we need to alter and cast the message pointers every time, resulting in extra cost. For release\_memory\_block, we actually perform a safety-check on the given memory address by comparing it to a list of valid heap addresses, unfortunately this is an O(n) operation. We also query the blocked queue and clear the entire memory block when it is released, these add more cost.

## BUGS & WAY TO FIX

### 7.1 Clear ‘Next’ when pop from ready queue

Another issue we had was an infinite loop when pushing a new process to ready queue.

In our pushToReadyQueue method, we have a loop to look for the end of the linked list and add the PCB to the end:

PCB\* tmp\_blk = ready\_q[priority];

while (tmp\_blk != NULL) {

tmp\_blk = tmp\_blk->next;

}

tmp\_blk->next = pcb;

The while became an infinite loop, which means that the last element in the linked list has its next pointer still pointing to right things.

After debugging, we realized that the problem was in the popFromReadyQ method.

When the scheduler is fired, the process with highest priority will be selected from the ready queue and returned as the next process to be executed. After the process with highest priority is popped out, the linked list in the ready queue will be updated. Initially, we have the following code to pop the first PCB from the priority linked list of the ready queue:

PCB\* popFromReadyQWithPriority(int priority) {

PCB\* firstNode = ready\_q[priority];  
 if (ready\_q[priority] != NULL) {

ready\_q[priority] = ready\_q[priority] ->next;

}

return firstNode;

}

In this code, we take the first PCB in the linked list, update the next PCB to be the head of the linked list, and simply return the first PCB.

However, the returned first node still points to the next node in the linked list, which causes the infinite loop when inserting an PCB into the ready queue.

So the solution is to set the next pointer of the firstNode to NULL when it is popped from the linked list.

PCB\* popFromReadyQWithPriority(int priority) {

PCB\* firstNode = ready\_q[priority];  
 if (ready\_q[priority] != NULL) {

ready\_q[priority] = ready\_q[priority] ->next;

firstNode->next = NULL;

}

return firstNode;

}

The following diagram clearly explained what happened in our infinite loop.

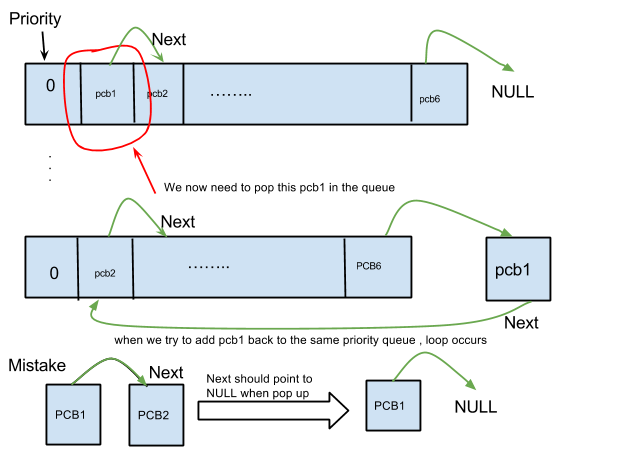


Figure Illustration of Popping up PCB from Ready Queue

### 7.3 ERROR code: -1 - Wrong memory address of Heap

In P1, we need to create a process that mimic the functionality of “maclloc” in C programming language.

When a process is running, say process A in the first place of in the ready queue with PRIORITY = 0, it may request 20 memory blocks. Next process to run is B, and it request for 14 memory blocks. However, we only have maximum 32 so the process B will be temporarily placed on to blocked queue until process A release its 20 memory blocks. This procedure involves k\_request\_memory\_block() and k\_release\_memory\_block (). The case is that we encountered error with ERROR code: -1 from compiler, which indicates we were pointing to wrong memory addresses during this request-release process of memory.

In our design for memory block request:

MEM\_BLK\* getAvailableMemBlk(){

int counter = 0;

int found = 0;

MEM\_BLK\* mem\_blk\_ptr = (MEM\_BLK\*)gp\_heap;

while(counter < MEM\_BLK\_NUM && found == 0){

if (mem\_blk\_ptr -> isUsed == 1){

mem\_blk\_ptr = (mem\_blk\_ptr+sizeof(MEM\_BLK)+(MEM\_SIZE/4));

counter++;

}

else{

found = 1;

}

}

if (found == 1){

return mem\_blk\_ptr;

}

else{

return NULL;

}

}

void \*k\_request\_memory\_block(void) {

MEM\_BLK\* availableBlock = getAvailableMemBlk();

//return (void\*)(gp\_heap+sizeof(MEM\_BLK));

#ifdef DEBUG\_0

printf("k\_request\_memory\_block: entering...\n");

#endif /\* ! DEBUG\_0 \*/

if (availableBlock == NULL){

//if no memory, push the current process to blocked queue once

pushToBlockedQ(gp\_current\_process);

}

while (availableBlock == NULL){

gp\_current\_process->m\_state = BLOCKED;

k\_release\_processor();

availableBlock = getAvailableMemBlk();

}

availableBlock->isUsed = 1;

availableBlock = availableBlock+sizeof(MEM\_BLK);

return (void\*)(availableBlock);

}

The noticeable fact is that in all the steps of address calculation of heap, we all use type of MEM\_BLK for variables to estimate total usage memory in this request process. First of all, we need to find an available memory block with desired size of MEM\_BLK(size of structure which contains a pointer to hold the links between MEM\_BLK as a linked list)+MEM\_SIZE/4(MEM\_SIZE is with type of U\*32, divided by 4 for conversion to byte)

The thing is that when we try to release what the process has been requested, we did not cast all the variable to type of MEM\_BLK.In our first trial, we had the following in our source code:

int k\_release\_memory\_block(void \*p\_mem\_blk) {

PCB\* pop\_blocked\_ptr = popFromBlockedQ();

(U32)\* mem\_blk = (U32\*)((MEM\_BLK\*)p\_mem\_blk - sizeof(MEM\_BLK));

if (is\_p\_block\_valid(mem\_blk) == 0){

return RTX\_ERR;

}

//set the block to be free

returnMemBlk(mem\_blk);

//if there are processes being blocked, release the last one in the blocked queue

if (pop\_blocked\_ptr != NULL){

//handle\_process\_ready adds a pcb just removed from the blocked queue to the ready queue

handle\_process\_ready\_from\_blocked(pop\_blocked\_ptr);

}

return RTX\_OK;

}

we chose the pointer type U32\* because the sizeof(MEM\_BLK) is a type of U32\*. But when we checked the address of heap in debugger, we got the following address information:

Address of available block returned: 1000470

Address that process has finished the MEM\_BLK release: 1000450

We actually were expecting that the position of first available address and the address of heap pointer returned after release\_mem\_block should be the same. For example, you borrowed a book from library and used it for some paper reference. You might be able to keep the book for 2 weeks, 2 months or even longer, however you still need to return the book to library at some later point of time. So the total number of books that you have does not change after you have returned book. Same idea here, processes ask for some available memory to do something, but later, in our case, after the function of k\_release\_memory\_block() has been called, heap should not have extra stuff on it. Those two numbers tell us that some parts of requested memory do not get released properly.

To fix this problem, we tried to cast our memory address to void\*:

(void)\* mem\_blk = (void\*)((MEM\_BLK\*)p\_mem\_blk-sizeof(MEM\_BLK));

This is the result that we get from debugger:

Address of available block returned: 1000470

Address that process has finished the MEM\_BLK release: 1000430

Apparently the problem has not be resolved, but our guess is correct - wrong variable type casting leads to the wrong address deallocation.

Finally, we changed our code to:

(MEM\_BLK)\*mem\_blk=(MEM\_BLK\*)((MEM\_BLK\*)p\_mem\_blk(sizeof(MEM\_BLK));

The reason that we cast address to MEM\_BLK\* type is as stated before - for the part of memory allocation, the memory address that we returned is of type MEM\_BLK\*. The evidence of correctness is:

Address of available block returned: 1000470

Address that process has finished the MEM\_BLK release: 1000470

Problem got fixed.

### 7.4 No process status check in process switch

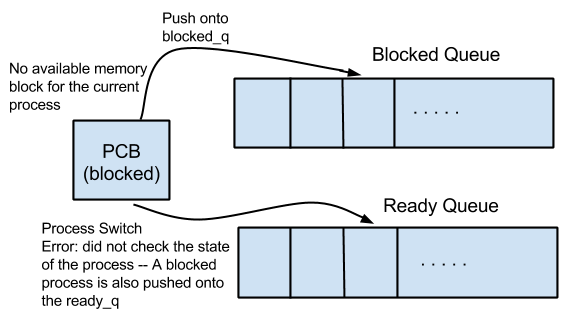
When a process gets blocked, it is added into the blocked\_q, and the process is marked as BLOCKED. However, after the processor is released, process switch is trying to switch from the current blocked process to another ready process. During the process switch, we did not check the state of the current process and simply add it to the ready queue. This causes the problem that even if the current process is blocked and is already in the blocked queue, it is still inserted back to the ready queue. Then an infinite loop is happening because a ready process is a blocked process.

Figure Illustration of infinite loop between Ready Queue and Blocked Queue

### 7.5 Missed priority in Blocked Queue and generic Queue struct

When we did our P1 demo, we were told that we should not just treat blocked queue as a simple FIFO container. Moreover, we also need to concern the priority in blocked queue as what we did in ready queue. Also, due to the reason that we also need a message queue in P2, we were suggested a generic queue struct.

In our generic queue, we have the following functionality that shared by all the structures.

void pushToQueue(PCB \*pcb, PCB\*\* queue);

void pushToQueueAtFront(PCB \*pcb, PCB\*\* queue);

void eraseFromQueue(PCB \*pcb, PCB\*\* queue);

PCB\* popFromQueue(PCB\*\* queue);

PCB\* popFromQueueWithPriority(int priority, PCB\*\* queue);

Now we have a queue with type of PCB \*\* which has the capability to hold an array of PCBs. One important thing to mention is that when we deal with message queue, variable type casting is needed since messages are not type of PCB.

### 7.6 Wrong heap allocation calculation in k\_memory.c

In our P2, we need to ensure that we don’t lose any part of memory from message since the pointers that links to the next message in message list is stored in envelope header.  However, we found that we were losing part of our linked list. This mean that we somehow did wrong arithmetic on heap with pointers. At first, we have

In this case, when we tried to find the heap address of next\_mem\_block, we thought this part of code **should** continuously updated the current heap pointer with an full memory block with an men\_blk struct size. However, next\_mem\_blk += sizeof(MEM\_BLK) is not going to only plus sizeof(MEM\_BLK) which is 128 us. This line, for compiler point of view, it will update the pointer with (struct of next\_mem\_blk)\*sizeof(MEM\_BLK) which is 24\*128.

This action resulted in an epic fail stuck overflow since we not only had used up everything in heap but also stuck as more processes were requesting memory.

The fix of this bug we did the following:

next\_mem\_blk = (MEM\_BLK\*)gp\_heap;

for (i = 0; i < MEM\_BLK\_NUM; i++){

curr\_mem\_blk = next\_mem\_blk;

curr\_mem\_blk->isUsed = 0;

curr\_mem\_blk->send\_id = 0;

curr\_mem\_blk->dest\_id = 0;

curr\_mem\_blk->next = NULL;

next\_mem\_blk++;

next\_mem\_blk = incrByBlockSize(next\_mem\_blk);

if (i != MEM\_BLK\_NUM - 1){

curr\_mem\_blk->next = next\_mem\_blk;

}

else{

curr\_mem\_blk->next = NULL;

}

}

MEM\_BLK\* incrByBlockSize(MEM\_BLK\* base){

U32\* temp\_cast\_median;

temp\_cast\_median = (U32\*)base;

temp\_cast\_median += (MEM\_SIZE/sizeof(U32));

base = (MEM\_BLK\*)temp\_cast\_median;

return base;

}

After applying the fix, next\_mem\_blk++ will only update the current heap pointer address value will be next\_mem\_blk+1\*sizeof(struct of(MEM\_BLK)=24), then add another size of mem\_blk which is 128. By doing this, we can actually go to the start of next available mem\_blk.

### 7.7 HARD FAULT in kernel function calls

First of all, we need to understand two function definitions in this part: ato\_on() and ato\_off();

ato\_on(): This function has been used to disable other IQR from this particular point. If somewhere in the code is set to be ato\_on(), this means that from this point of code, no other interruption is accepted anymore. Process or CPU must wait until this part of execution to be done then they are able to do other works. NO ANY OTHER INTERRUPTION IS ALLOWED HERE.

ato\_off(): This function has been used to enable other IQR from this particular point. From where ato\_off() is been called, the following part of code is no longer being protected therefore any interruption may happen.

In our previous design, a flaw that is called hard fault would have killed our PTX project. Consider the following scenario, if a process A for any reason has been released, i.e , k\_release\_processor has been called, the execution now goes to call function process\_switch. Process\_switch may call some functions to create new process.  This new process is going to call some other kernel functions with SVC instructions.

At beginning, we call ato\_on() before we call \_rte() when a new process is created. Therefore, now no more interruptions from anywhere will be accepted. Since \_\_rte() is an exception, the code execution exits from this point, so all of the parts in the function that follows it will not be executed. So ato\_off() here failed to be called. Then, if this new process is calling any kernel function, e.g, release\_process(),  the request will be sent to CPU therefore a SVC instruction will be involved into this process. However, SVC is expected to trigger another interrupt to CPU while our code line is still in the mode ato\_off(). NO INTERRUPTION CAN BE ALLOWED. As a result of these atomic functions, SVC will fail to execute, nothing will return to kernel. HARD FAULT caught.

Our defected code is :

….

ato\_on();//somewhere before process\_switch

….

gp\_current\_process->m\_state = RUN;

\_\_set\_MSP((U32) gp\_current\_process->mp\_sp);

\_\_rte();  // pop exception stack frame from the stack for a new processes

ato\_off();// This line will never be touched ever. No interruption allowed.

}

The fix of this problem is

….

ato\_on();//somewhere before process\_switch

….

gp\_current\_process->m\_state = RUN;

\_\_set\_MSP((U32) gp\_current\_process->mp\_sp);

ato\_off(); //adding atomic off here since rtc jumps straight to the new process without effectively return

\_\_rte();  // pop exception stack frame from the stack for a new processes

Note: All places that involve in process\_switch and k\_release\_processor has been added with critical section protection with ato\_on() and ato\_off()

By doing this, we first disable ALL possible interruption as usual but we enable all the interruption before any exception \_\_rte() is been called. Then we deal with SVC part in the newly created process, interruption is allowed now with no more HARD FAULTs.

## Appendix A:

## 1 DATA STRUCTURE (BASIC ELEMENT):

### A.1.1 READY\_Q

typedef struct ready\_q

{

READY\_Q\_BLK \*first;

READY\_Q\_BLK \*last;

} READY\_Q;

### A.1.2 READY\_Q\_BLK

typedef struct ready\_q\_blk

{

struct ready\_q\_blk \*next;

PCB \*next;

} READY\_Q\_BLK;

### A.1.3 PCB

typedef struct pcb

{

struct pcb \*mp\_next; /\* next pcb, not used in this example \*/

U32 \*mp\_sp; /\* stack pointer of the process \*/

U32 m\_pid; /\* process id \*/

PROC\_STATE\_E m\_state; /\* state of the process \*/\

int m\_priority;

} PCB;

### A.1.4 PROC\_INIT

typedef struct proc\_init

{

int m\_pid; /\* process id \*/

int m\_priority; /\* initial priority, not used in this example. \*/

int m\_stack\_size; /\* size of stack in words \*/

void (\*mpf\_start\_pc) ();/\* entry point of the process \*/

} PROC\_INIT;

### A.1.5 MEM\_BLK

typedef struct mem\_blk {

struct mem\_blk \*next;

int isUsed;

} MEM\_BLK;

### A.1.6 msgbuf

typedef struct msgbuf

{

int send\_id;

int dest\_id;

struct msgbuf \*next;

int mtype; /\* user defined message type \*/

char mtext[1]; /\* body of the message \*/

} MSG\_BUF;

## 2 System initialization functions

### A.2.1 svc\_main()

int main()

{

/\* CMSIS system initialization \*/

SystemInit();

#ifdef DEBUG\_0

init\_printf(NULL, putc);

#endif /\* DEBUG\_0 \*/

/\* start the RTX and built-in processes \*/

rtx\_init();

/\* We should never reach here!!! \*/

return RTX\_ERR;

}

### A.2.2 k\_rtx\_init()

void k\_rtx\_init(void)

{

volatile uint8\_t sec = 0;

\_\_disable\_irq();

timer\_init(0);

memory\_init();

process\_init();

uart0\_irq\_init(); // uart0 interrupt driven, for RTX console

uart1\_init(); // uart1 polling, for debugging

\_\_enable\_irq();

/\* start the first process \*/

k\_release\_processor();

}

### A.2.3 memory\_init()

void memory\_init(void)

{

U8 \*p\_end = (U8 \*)&Image$$RW\_IRAM1$$ZI$$Limit;

int i;

int totalLength;

U32\* current = gp\_heap;

MEM\_BLK\* next\_mem\_blk;

MEM\_BLK\* curr\_mem\_blk;

/\* 4 bytes padding \*/

p\_end += 4;

/\* allocate memory for pcb pointers \*/

gp\_pcbs = (PCB \*\*)p\_end;

p\_end += (NUM\_TOTAL\_PROCS) \* sizeof(PCB \*);

for ( i = 0; i < NUM\_TOTAL\_PROCS; i++ ) {

gp\_pcbs[i] = (PCB \*)p\_end;

p\_end += sizeof(PCB);

}

for ( i = 0; i < NUM\_TOTAL\_PROCS; i++ ) {

//printf("gp\_pcbs = 0x%x \n", gp\_pcbs[i]);

}

gp\_stack = (U32 \*)RAM\_END\_ADDR;

if ((U32)gp\_stack & 0x04) { /\* 8 bytes alignment \*/

--gp\_stack;

}

/\* allocate memory for heap, not implemented yet\*/

gp\_heap = (U32\*) p\_end;

if ((U32)gp\_heap & 0x04) {

++gp\_heap;

}

totalLength = (sizeof(MEM\_BLK)+(MEM\_SIZE/sizeof(int)))\*MEM\_BLK\_NUM;

current = gp\_heap;

for (i = 0; i < totalLength; i++){

\*current = 0;

current++;

}

next\_mem\_blk = (MEM\_BLK\*)gp\_heap;

for (i = 0; i < MEM\_BLK\_NUM; i++){

curr\_mem\_blk = next\_mem\_blk;

curr\_mem\_blk->isUsed = 0;

curr\_mem\_blk->send\_id = 0;

curr\_mem\_blk->dest\_id = 0;

curr\_mem\_blk->next = NULL;

next\_mem\_blk++;

next\_mem\_blk = incrByBlockSize(next\_mem\_blk);

if (i != MEM\_BLK\_NUM - 1){

curr\_mem\_blk->next = next\_mem\_blk;

}

else{

curr\_mem\_blk->next = NULL;

}

}

}

### A.2.4 process\_init()

void process\_init() {

int i = 0;

U32 \*sp;

/\* fill out the initialization table \*/

set\_test\_procs();

//define the null process

g\_proc\_table[0].m\_pid=(U32)(PID\_NULL);

//NULL process has priority 4

g\_proc\_table[0].m\_priority = 4;

g\_proc\_table[0].m\_stack\_size=0x100;

g\_proc\_table[0].mpf\_start\_pc = &nullProc;

for ( i = 0; i < NUM\_TEST\_PROCS; i++ ) {

g\_proc\_table[i+1].m\_pid = g\_test\_procs[i].m\_pid;

g\_proc\_table[i+1].m\_stack\_size = g\_test\_procs[i].m\_stack\_size;

g\_proc\_table[i+1].mpf\_start\_pc = g\_test\_procs[i].mpf\_start\_pc;

g\_proc\_table[i+1].m\_priority = g\_test\_procs[i].m\_priority;

}

initStressProcesses();

initSystemProcesses();

/\* initilize exception stack frame (i.e. initial context) for each process \*/

for ( i = 0; i < NUM\_TOTAL\_PROCS; i++ ) {

int j;

(gp\_pcbs[i])->m\_pid = (g\_proc\_table[i]).m\_pid;

(gp\_pcbs[i])->m\_state = NEW;

(gp\_pcbs[i])->msgqueue = NULL;

(gp\_pcbs[i])->wait = 0;

if ((g\_proc\_table[i]).m\_pid == PID\_TIMER\_IPROC || (g\_proc\_table[i]).m\_pid == PID\_UART\_IPROC){

(gp\_pcbs[i])->m\_state = WAITING;

}

sp = alloc\_stack((g\_proc\_table[i]).m\_stack\_size);

\*(--sp) = INITIAL\_xPSR; // user process initial xPSR

\*(--sp) = (U32)((g\_proc\_table[i]).mpf\_start\_pc); // PC contains the entry point of the process

for ( j = 0; j < 6; j++ ) { // R0-R3, R12 are cleared with 0

\*(--sp) = 0x0;

}

(gp\_pcbs[i])->mp\_sp = sp;

(gp\_pcbs[i])->m\_priority = (g\_proc\_table[i]).m\_priority;

if(i <= 6 || i == PID\_A || i == PID\_B || i == PID\_C || i == PID\_CLOCK || i == PID\_CRT || i == PID\_KCD || i == PID\_SET\_PRIO){

pushToReadyQ(gp\_pcbs[i]);

}

}

}

### A.2.5 UART0\_IRQHandler() assembly

\_\_asm void UART0\_IRQHandler(void)

{

PRESERVE8

IMPORT c\_UART0\_IRQHandler

IMPORT ato\_on

IMPORT ato\_off

PUSH{lr}

PUSH{r4-r11}

BL ato\_on

BL c\_UART0\_IRQHandler

POP{r4-r11}

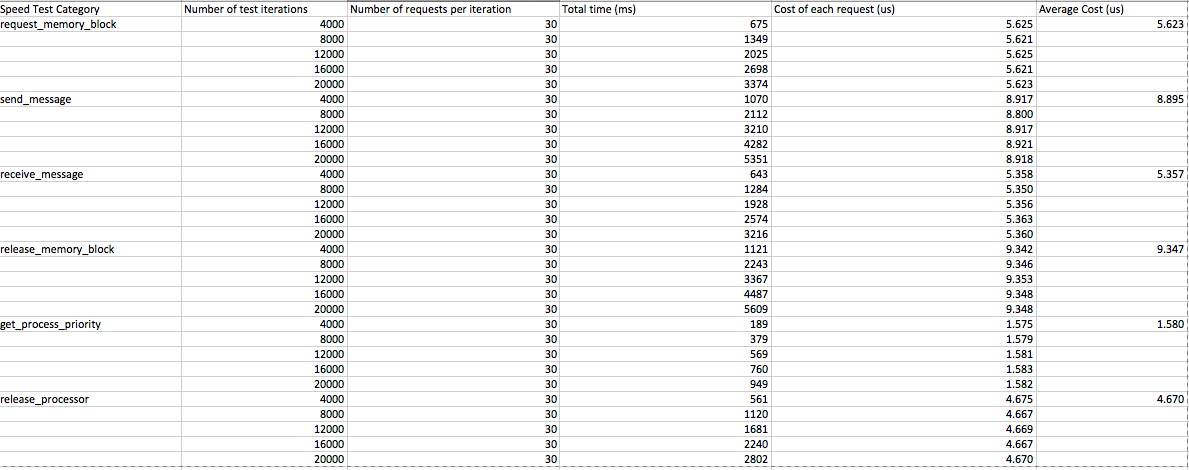
BL ato\_off

pop{pc}

}

## 3 P4 timer testing data sheet

You should be able to zoom in to see the details.



## Appendix B: Hotkey illustration

