

RTX Project Report

Xiang, Dian

20431601

`dxiang@uwaterloo.ca`

Justin McGirr

20413625

`jmcgirr@uwaterloo.ca`

Adrian Cheung

20421743

`a32cheun@uwaterloo.ca`

Aaron Morais

20413440

`aemorais@uwaterloo.ca`

April 4, 2014

Abstract

Contents

I	Introduction	5
II	Kernel API	6
1	RTX Primitives and Services	7
1.1	Memory Management	7
1.1.1	Description	7
1.1.2	Requesting Memory	8
1.1.3	Releasing Memory	9
1.2	Processor Management	9
1.3	Interprocess Communication	10
1.3.1	Send Messages	10
1.3.2	Receive Messages	11
1.3.3	Receive Messages	12
1.4	Description	13
1.5	Process Priority	13
1.6	Interrupts and Their Han	13
1.7	Description	13
1.7.1	Block Layer	13
1.7.2	Metadata Layer	13
1.8	Theoretical Analysis	13
1.9	Measurements	13
2	Interrupts and Their Handlers / Processes	14
2.1	Description	14
3	System and User Processes	15
3.1	Description	15

3.1.1	‘funProcess’	15
3.1.2	‘schizophrenicProcess’	15
3.1.3	‘fibProcess’	15
3.1.4	‘memoryMuncherProcess’	15
3.1.5	‘releaseProcess’	15
III	Initialization	16
IV	Testing	17
V	Timing	18
4	Acquiring Timings	19
5	Timing Analysis	20
VI	What We Learned	21
VII	Major Design Changes	22
A	Raw Measurement Data	23
A.1	Trial Information	23
A.2	Function Runtime Profiling	24

List of Algorithms

List of Figures

Part I

Introduction

Part II

Kernel API

Chapter 1

RTX Primitives and Services

1.1 Memory Management

1.1.1 Description

The RTX provides the utility of simple memory management of the heap. The main memory on the board (TODO: name the board) is divided into sections of the RTX Image, PCB data, the heap, and process stacks seen in figure blah. *** TODO insert figure blah here ***

Board LPC17xx (TODO insert the real name) does not have the hardware to support virtual memory. Thus, a fixed memory management scheme is used with variable block sizes. The OS kernel is always loaded into main memory. When the OS boots, a user stated number of PCBs and process stack are allocated. The OS also allocates memory for system processes such as the Keyboard Command Decoder Process (KCD) and the CRT Display Process. After allocation of PCBs and process stacks, the remaining memory is used for the heap, which is shared between all processes. This section will be focused on the memory management of the heap.

The heap is further divided into a variable number of blocks. Each block contains a header (HeapBlockHeader) and 128 bytes of data. Depending on the number of user and system processes, the number of available heap memory will vary. The size of the data can vary from 128 bytes but must be set at compile time. The OS supports two kernel calls that gives access to these memory blocks.

1.1.2 Requesting Memory

```
1 void *request_memory_block();
```

The first functionality supported by the OS is the ability to request memory. This call gives back a pointer to a memory block in the heap. The size of the memory block is defined by the constant `HEAP_BLOCK_SIZE`, which defaults to 128 bytes. These blocks are used for storing local variables or as envelopes for interprocess communication (described in section TODO). The user must cast the memory block to the proper type for his or her own use. An example of the usage is provided below, which stores the numbers 0-31 in an array of size 32.

```
1 void user_process() {  
    int size = 32;  
3  
    int* array = (int*)request_memory_block();  
5  
    for( int i = 0; i < size; i++ ) {  
7        array[i] = i;  
    }  
9  
    // ...  
11  
    release_memory_block( (void*)array );  
13 }
```

All processes will share the same heap memory pool. Thus, this primitive will block the process if the OS does not have anymore memory blocks to give out at the time, thus effecting the execution of the process. In that case, it will only be unblocked there is a new memory block available and if it has the highest priority on the list of processes waiting for a memory block. When using the memory block, the user must be aware of writing past the heap block size. The OS does not check for segmentation faulting. Thus, undefined behavior may occur. Also, the user must remember to release the memory block or memory leakage will occur.

1.1.3 Releasing Memory

```
1 int release_memory_block(void *memory_block);
```

The second functionality supported by the OS is the ability to release memory. This is a non-blocking call that returns the memory block back to the OS. This should be called when a message is received and not passed on or when the process is done using the memory block. If any process is blocked on memory, it will be unblocked and put to the ready queue. If the current process has lower priority than the unblocked process, then the current process will be preempted and the higher priority process will be executed instead. An example of this can be seen in figure CODE1 TODO on line 13.

1.2 Processor Management

```
1 int release_processor();
```

The OS manages processes as though it is on a uniprocessor. A priority based scheme with context switching is used for scheduling processes. A process can voluntarily release the processor to the OS at any time during its execution. If there are errors in the call, it will return an error without releasing the processor to the OS. If there are no errors and the invoking process is ready to execute, it is put to the end of the ready queue of its priority. If there are no other processes of equal or higher priority, the process will be chosen to execute again. However, if there is, another process may be chosen for execution. Below is an example which prints an increasing number and releases the processor at every turn.

```
1 void user_process() {  
    static num = 0;  
3    while(1) {  
        print(num++);  
5        release_processor();  
    }  
7 }
```

1.3 Interprocess Communication

Apart from heap memory, processes do not share information. Thus, communication between processes is done through message-based interprocess communication (IPC). Details of the internal layout of process mailboxes can be found in section TODO. The RTX gives three primitives to carry out this task, one for sending, one for delayed sending, and one for receiving.

1.3.1 Send Messages

```
1 int send_message(int process_id, void *message_envelope);
```

A process can compose a message envelope to be sent to another process. Memory for envelope message must be requested from the RTX using the request_memory() routine. The envelope consists of a type (mtype) and the message data (mtext) which must be filled in as seen below. The predefined message types are used for the KCD, CRT, and Wall Clock process, which are built into the RTX (see section TODO for more information). The message data has a predefined size which is a MessageType smaller than the HEAP_BLOCK_SIZE. Thus, any message that are longer will exhibit undefined behavior. The process ID of the receiving process must also be known ahead of time in order to use send_message.

```
1 typedef enum {
    MESSAGE_TYPE_KCD_KEYPRESS_EVENT      = 0,
3    MESSAGE_TYPE_KCD_COMMAND_REGISTRATION = 1,
    MESSAGE_TYPE_CRT_DISPLAY_REQUEST     = 2,
5    MESSAGE_TYPE_WALL_CLOCK              = 3,
    MESSAGE_TYPE_USER_DEFINED             = 4,
7
    MESSAGE_TYPE_NUM                      = 5,
9 } MessageType;

11 struct msgbuf {
    MessageType mtype;
13    char mtext[HEAP_BLOCK_SIZE - sizeof(MessageType)];
};
```

The primitive returns a status which validates the, message, receiver process ID, and ready queue. If the receiving process is currently blocked on receiving message, this call will add the message to the process' message box and put it back on the message queue. The current process continues to execute unless the receiving process has a higher priority. In that case, the current process will be preempted and put to the back of the ready queue. Thus, this primitive may effect the execution of the process.

1.3.2 Receive Messages

```
void *receive_message(int *sender_id);
```

A process can use this primitive to receive messages from other processes. Unless the sender_id is NULL, the sender of the message will be written to the sender_id parameter. The current process will check its mailbox for any incoming messages. If there are no messages in its mailbox, the routine will block the process and select another process for execution. Execution of the process will occur again if a message is sent from another process and this process has the highest priority in the ready queue. The message_envelope are heap memory underneath. Thus, it is required that unless the process passes the message onto another process, the final receiving process must release the message_envelope block after its usage. An example of send_message and receive_message is shown below.

```
1 void process_send() {
2     while(1) {
3         // Request one block for process_receive to use
4         struct msgbuf* message_envelope
5             = (struct msgbuf*)request_memory_block();
6         message_envelope->mtype = MESSAGE_TYPE_USER_DEFINED_1;
7         strcpy(message_envelope->mtext, "process message");
8         send_message(PROCESS_RECEIVE_ID, message_envelope);
9
10        // Send another block for process_receive to send to CRT
11        struct msgbuf* message_envelope
12            = (struct msgbuf*)request_memory_block();
13        message_envelope->mtype = MESSAGE_TYPE_USER_DEFINED_2;
14        strcpy(message_envelope->mtext, "print this");
15        send_message(PROCESS_RECEIVE_ID, message_envelope);
16    }
17 }
```

```

19 void process_receive() {
    while(1) {
21         struct msgbuf* message
            = (struct msgbuf*)receive_message(NULL);
23         if( message->mtype = MESSAGE_TYPE_USER_DEFINED_2 ) {
            // Send to CRT for printing; do not release memory
25             message->mtype = MESSAGE_TYPE_CRT_DISPLAY_REQUEST;
            send_message(PROCESS_ID_CRT, message);
27         } else {
            // Do something with the message then release it
29             release_memory_block( (void*)message );
        }
31     }
}

```

1.3.3 Receive Messages

```

int delayed_send(int process_id, void *message-envelope, int
    delay);

```

This primitive is very similar to the primitive of `send_message` in section [1.3.1](#) with the addition of a delay. Instead of sending the message immediately, the message will be sent delay number of milliseconds. The `message-envelope` is constructed in the same way. The `process_id` will be the receiving process. The execution of the process may be preempted after a the delay period if the receiving process was blocked on `receive_message` and

has a higher priority.

1.4 Description

1.5 Process Priority

1.6 Interrupts and Their Han

1.7 Description

1.7.1 Block Layer

1.7.2 Metadata Layer

1.8 Theoretical Analysis

1.9 Measurements

Chapter 2

Interrupts and Their Handlers / Processes

2.1 Description

Chapter 3

System and User Processes

3.1 Description

3.1.1 ‘funProcess’

3.1.2 ‘schizophrenicProcess’

3.1.3 ‘fibProcess’

3.1.4 ‘memoryMuncherProcess’

3.1.5 ‘releaseProcess’

Part III

Initialization

Part IV

Testing

Part V

Timing

Chapter 4

Acquiring Timings

Chapter 5

Timing Analysis

Part VI

What We Learned

Part VII

Major Design Changes

Appendix A

Raw Measurement Data

A.1 Trial Information

Trial	Total Runtime	Notes
1	4.219	Normal (no stress processes)
2	7.754	Wall clock
3	8.487	Normal (no stress processes)
4	6.5	No Memory Muncher or Release Process
5	30.988	Stress processes

A.2 Function Runtime Profiling

Function	Trial	Time (μs)	# of Calls	Average time / call (μs)
k_sendMessage	1	601.58	552	1.090
k_receiveMessage	1	408.22	565	0.723
k_acquireMemoryBlock	1	244.12	294	0.830
k_sendMessage	2	647.44	594	1.090
k_receiveMessage	2	437.78	606	0.722
k_acquireMemoryBlock	2	258.68	320	0.808
k_sendMessage	3	630.99	579	1.090
k_receiveMessage	3	426.83	591	0.722
k_acquireMemoryBlock	3	259.24	321	0.808
k_sendMessage	4	108.80	100	1.088
k_receiveMessage	4	74.44	110	0.677
k_acquireMemoryBlock	4	92.47	123	0.752
k_sendMessage	5	750.63	687	1.093
k_receiveMessage	5	497.09	693	0.717
k_acquireMemoryBlock	5	329.90	447	0.738