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

List of Algorithms

List of Figures

Part I

Introduction

Part II

Awesome RTX

Chapter 1

Kernel API

1.1 Description

This section describes the kernel API that is available to users in the RTX. It will only go into details of how to use each API function call and states of different scenarios. Details of the implementation can be found in section TODO or from the raw code.

1.2 Memory Management

1.2.1 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 alway 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.

1.2.2 Requesting Memory

```
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.

```
void user_process() {
    int size = 32;

    int* array = (int*)request_memory_block();

    for( int i = 0; i < size; i++ ) {
        array[i] = i;
    }

    // ...

    release_memory_block( (void*)array );
}
```

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.

1.2.3 Releasing Memory

```
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.

1.3 Processor Management

```
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.

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

1.4 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.

1.4.1 Send Messages

```c
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.

```c
typedef enum {
    MESSAGE_TYPE_KCD_KEYPRESS_EVENT          = 0,
    MESSAGE_TYPE_KCD_COMMAND_REGISTRATION = 1,
    MESSAGE_TYPE_CRT_DISPLAY_REQUEST        = 2,
    MESSAGE_TYPE_WALL_CLOCK                 = 3,
    MESSAGE_TYPE_USER_DEFINED               = 4,

    MESSAGE_TYPE_NUM                        = 5,
} MessageType;

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

The primitive returns a status which validates the, message, receiver process ID, and ready queue. User processes is allowed to send a message to any user processes or system processes such as the KCD. 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.

1.4.2 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.

```
void process_send() {
    while(1) {
        // Request one block for process_receive to use
        struct msgbuf* message_envelope
            = (struct msgbuf*)request_memory_block();
        message_envelope->mtype = MESSAGE_TYPE_USER_DEFINED_1;
        strcpy(message_envelope->mtext, "process message");
        send_message(PROCESS_RECEIVE_ID, message_envelope);

        // Send another block for process_receive to send to CRT
        struct msgbuf* message_envelope
            = (struct msgbuf*)request_memory_block();
```

```c
        message_envelope->mtype = MESSAGE_TYPE_USER_DEFINED_2;
        strcpy(message_envelope->mtext, "print this");
        send_message(PROCESS_RECEIVE_ID, message_envelope);
    }
}

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

1.4.3 Receive Messages

```c
int delayed_send(int process_id, void *message_envelope, int
    delay);
```

This primitive is very similar to the primitive of send_message in section 1.4.1 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.

1.5 Process Priority

The RTX schedules processes based on a fixed priority scheme. Thus, the RTX provides the utility to change priorities and to get the process priority of any user processes.

1.5.1 Set Process Priority

```c
int set_process_priority(int process_id, int priority);
```

This primitive allows user processes to set the priorities of other user processes. User processes are not allowed to set the priorities of any system processes. System processes, however, are allowed to set the priorities of other system or i-processes. The primitive validates if the current process is allowed to set the priority of the process with process_id and whether it's a valid process_id. It also checks the validity of the priority. An error status of RTX_ERR is given back if the validation does not pass. If properly set, a status of RTX_OK is returned. The caller of this primitive is not blocked but can be preempted if the priority of the set process is higher. Otherwise, the process continues to execute. An example usage is shown in section 1.5.2

1.5.2 Get Process Priority

```c
int get_process_priority(int process_id);
```

Given a process_id, this primitive gives back the priority of any processes including any system or i-processes to a user process. A invalid process_id will result in a RTX_ERR status. Otherwise, a RTX_OK is returned to the caller. An example usage of both get and set process priority is shown below.

```c
// Assume this process has a process ID of 3
// and there is another user process with ID of 1
void user_process() {
    int process_priority_3 = get_process_priority(3);
    if( process_priority_3 == USER_PROCESS_PRIORITY_MEDIUM ) {
        // Make process 1 have a higher priority
        // This call will preempt
        int status = set_process_priority(
                            1, USER_PROCESS_PRIORITY_HIGH );
        if( status == RTX_ERR ) {
            release_processor();
        }
    }
}
```

Chapter 2

Interrupts and Their Handler and Processes

2.1 Description

2.2 UART

2.3 TIMER

Chapter 3

System and User Processes

3.1 Description

3.2 System Processes

3.3 User Processes

3.3.1 ‘funProcess’

3.3.2 ‘schizophrenicProcess’

3.3.3 ‘fibProcess’

3.3.4 ‘memoryMuncherProcess’

3.3.5 ‘releaseProcess’

Chapter 4

Initialization

4.1 Description

4.2 Memory Layout

4.3 Process Mailbox

Chapter 5

Major Design Changes

5.1 Description

5.2 Heap

5.3 Scheduler

5.4 What We Learned

Part III

Testing and Analysis

Chapter 6

Testing

6.1 Description

6.2 Theoretical Analysis

6.3 Measurements

Chapter 7

Timing Analysis

7.1 Description

7.2 Acquiring Timings

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