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 3, 2014

Abstract

Contents

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 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.

1.1.2 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.

1.1.3 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.

1.2 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();
    }
}
```

1.3 Interprocess Communication

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

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