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

Global Variables

1.1 Description

There are various amounts of variables used throughout the RTX. This section describes the global variables that are used to accomplish tasks in the RTX. Other global variables such as the global process IDs can be seen in the appendix TODO. There are three major sections that required global variables and data structures: memory management in the heap, scheduler, and user processes.

1.2 Heap Data Structures

The RTX provides functionality to request and release memory from the heap, which is shared and stored in the RAM of the board. There is one main data structure that stores each memory block.

```
typedef struct HeapBlockHeader {
  int source_pid;
  int dest_pid;
  unsigned int send_time;
  struct HeapBlock* p_next;
  struct HeapBlock* p_next_usr;
} HeapBlockHeader;

#define HEAP_BLOCK_SIZE 128

typedef struct HeapBlock {
  HeapBlockHeader header;
```

```
13   byte              data[HEAP_BLOCK_SIZE];
} HeapBlock;
```

The HeapBlock structure stores a header and the content of the block. When a memory is requested, users are given the data with a adjustable size of HEAP_BLOCK_SIZE and does not have knowledge of the header. Helper functions in the RTX turn the user block back into kernel block by adjusting the pointer of the block. Each header contains information about message passing (if the block is used for this purpose) and a pointer to the next block (for kernel memory management purposes). This global data structure is used in the following sections:

1. **Memory Management** - this section uses this data structure to give and receive back heap memory blocks
2. **Process Message Passing** - this section uses this data structure pass messages between processes. Message envelope are implemented as HeapBlocks.
3. **Timer I-Process** - The process uses message passing to send delayed messages.
4. **UART I-Process** - The process uses message passing to process input and output characters.
5. **CRT I-Process** - The process receives messages and passes message for terminal output.
6. **KCD Process** - The KCD uses message passing to CRT for display and user processes for processing.

1.3 Process Scheduler Data Structures

The RTX has a fixed-priority based scheduler that acts as a uniprocessor system. Context switching is required and the following data structures are used for this purpose.

1.3.1 Ready and Blocked Priority Queues

On initialization, each in memory process is given a process control block (PCB). The PCB contains data on the process such as its process ID, stack pointer, process state, and priority that the kernel will use for scheduling. The blocked and ready priority queues of PCBs keep track of which processes are blocked and ready for execution. Each process is in one of the 5 states at all time listed in ProcessState. More details about the states can be found in section TODO. Processes that are in state PROCES_STATE_READY are on the ready queue. Processes in state PROCESS_STATE_BLOCKED are in the blocked queue. There is also an implicit queue for processes blocked on message but we won't go into that in this section.

```
typedef enum {
  PROCESS_STATE_NEW                 = 0,
  PROCESS_STATE_READY               = 1,
  PROCESS_STATE_RUNNING             = 2,
  PROCESS_STATE_BLOCKED             = 3,
  PROCESS_STATE_BLOCKED_ON_MESSAGE  = 4,
} ProcessState;

typedef struct PCB {
  // Stack pointer
  U32*            sp;

  ProcessID       pid;
  ProcessState    state;
  ProcessPriority priority;

  struct PCB*     p_next;
  // Incoming messages, waiting to be processed.
  HeapBlock*      message_queue;
} PCB;

PCB* g_ready_process_priority_queue[PROCESS_PRIORITY_NUM] = {
   NULL};
PCB* g_blocked_process_priority_queue[PROCESS_PRIORITY_NUM];
```

PCBs and the priority queues are used in the following sections:

1. **Memory Management** - When a process requests memory without available memory blocks, the memory management unit must add the process to a blocked queue. When a memory block is released, the pro-

cess with the highest priority in the blocked queue (if any) is moved into the ready queue. Thus, the memory management unit must have access to the PCBs, and blocked and ready queues in order to accomplish these task.

2. **Process Management** - The process management unit is the one that schedules processes and keep track of the process priorities. Ready processes are taken from the ready queue to be ran if the current process is blocked or finished its execution. The process management unit also takes care of updating the two priority queues when the priority of a process has been changed.

3. **UART Process** - The UART Process needs access to the PCB. The PCBs also contain a mailbox for messages. The UART displays any incoming messages to the terminal display. Thus, it needs to know the PCB structure in order to gain access to the mailbox.

1.3.2 Priorities

The RTX is based on a fixed-priority scheduler with 5 user process priorities and 2 system process priorities given below. Users are given priorities PROCESS_PRIORITY_HIGH to PROCESS_PRIORITY_LOWEST. PROCESS_PRIORITY_NULL_PROCESS is given to the null process and PROCESS_PRIORITY_SYSTEM_PROCESS are given to critical processes such as the KCD, CRT, timer I-process, and UART I-process.

```
typedef enum {
  PROCESS_PRIORITY_INVALID          = 0,
  PROCESS_PRIORITY_SYSTEM_PROCESS   = 1,
  PROCESS_PRIORITY_HIGH             = 2,
  PROCESS_PRIORITY_MEDIUM           = 3,
  PROCESS_PRIORITY_LOW              = 4,
  PROCESS_PRIORITY_LOWEST           = 5,
  PROCESS_PRIORITY_NULL_PROCESS     = 6,
  PROCESS_PRIORITY_UNSCHEDULABLE    = 7,

  PROCESS_PRIORITY_NUM              = 8
} ProcessPriority;

typedef enum {
  USER_PROCESS_PRIORITY_HIGH            = 0,
  USER_PROCESS_PRIORITY_MEDIUM          = 1,
```

```
  USER_PROCESS_PRIORITY_LOW                = 2,
  USER_PROCESS_PRIORITY_LOWEST             = 3,

  USER_PROCESS_PRIORITY_NUM                = 4,
} UserProcessPriority;
```

This priority structure is used by the process management unit to schedule and block processes based on their priorities. The RTX provides users with UserProcessPriority while keeping an internal structure of ProcessPriority.

1.3.3 Message Passing

Message passing is a way for interprocess communication provided by the RTX. Messages are created in the form of a msgbuf structure, which includes the type and content. Some processes such as the CRT and KCD will require a certain type of message to be sent before the message can be processed correctly.

```
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_COUNT_REPORT                = 4,
  MESSAGE_TYPE_WAKEUP_10                   = 5,

  MESSAGE_TYPE_NUM                         = 6,
} MessageType;

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

The message passing data structure is evident in the KCD, wall clock process, and all I-processes. It can also be used in all user processes.

Chapter 2

Kernel API

2.1 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.

2.2 Memory Management

2.2.1 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.

2.2.2 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.

```
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.

2.2.3 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.

2.3 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();
```

```
    }
}
```

2.4 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.

2.4.1 Send Messages

```
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.

```
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;
```

```
13      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.

2.4.2 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.

```
1  void process_send() {
       while(1) {
3          // Request one block for process_receive to use
           struct msgbuf* message_envelope
5              = (struct msgbuf*)request_memory_block();
           message_envelope->mtype = MESSAGE_TYPE_USER_DEFINED_1;
7          strcpy(message_envelope->mtext, "process message");
           send_message(PROCESS_RECEIVE_ID, message_envelope);
9
           // Send another block for process_receive to send to CRT
11         struct msgbuf* message_envelope
               = (struct msgbuf*)request_memory_block();
```

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

2.4.3 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 2.4.1 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.

2.5 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.

2.5.1 Set Process Priority

```
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 2.5.2

2.5.2 Get Process Priority

```
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.

```
// 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 3

Interrupts and Their Handler and Processes

3.1 Description

3.2 UART

3.3 TIMER

Chapter 4

System and User Processes

4.1 Description

4.2 System Processes

4.3 User Processes

4.3.1 'funProcess'

4.3.2 'schizophrenicProcess'

4.3.3 'fibProcess'

4.3.4 'memoryMuncherProcess'

4.3.5 'releaseProcess'

Chapter 5

Initialization

5.1 Description

5.2 Memory Layout

5.3 Process Mailbox

Chapter 6

Major Design Changes

6.1 Description

6.2 Heap

6.3 Scheduler

6.4 What We Learned

Part III

Testing and Analysis

Chapter 7

Testing

7.1 Description

7.2 Theoretical Analysis

7.3 Measurements

Chapter 8

Timing Analysis

8.1 Description

8.2 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