ECE354: RTX Project Final Report

Ben Ridder - brridder
Casey Banner - cccbanne
David Janssen - dajjanss

July 24, 2011

Contents

Chapter 1

Software Design

1.1 Introduction

1.2 Global Information

Within the RTX system, there are several pieces of information available globally within all contexts and several that are exposed only within kernel mode. Certain data structures, such as the message envelope, are used by user processes. Other structures are used strictly by the kernel. Several global variables are available for certain use cases. Constants are used to define priority levels, IDs, and more. Stack usage is limited to standard function uses, storing states before switching contexts, and for process switching. Memory mapping is handled with a free list paired with an allocation bit field for control.

1.2.1 Data Structures

The global data structures available have been split up into two categories: user mode and kernel mode structures. This has been done to differentiate between what is normally accessible in each context.

User Level

In user level mode there is one data structure available to the programmer which is also available in kernel mode. This is the message envelope used to send and receive messages between processes. It contains sender and receiver process ids, the message type, pointers to the next and previous messages when it is enqueued, values to control delay sending, and a data field defined at the last 64 bytes of the envelope. The message type is defined in Section 1.2.2. To the user the only fields that are modifiable in any meaningful way are the message type and data. All of the other fields are modified by the kernel when the message is sent to another process. If the user does not need to set the message type then this data structure is not necessary as the only knowledge required for message sending is that the data resides at an offset of 64 bits. In this case, the message envelope is strictly used by the kernel.

Table 1.1: User Level Data Structures

Structure	Definition

Message Envelope	`int sender_pid`
	`int receiver_pid`
	`enum message_type type`
	`struct message_envelope* next`
	`struct message_envelope* previous`
	`int delay`
	`int delay_start`
	`unsigned char padding[36]`
	`unsigned char data[64]`

Kernel Level

1.2.2 Constants

1.2.3 Memory

1.3 Primitives

1.3.1 Release Processor

The release processor primitive is used to make a scheduling decision. Currently, this happens when the currently running process calls `release_processor()`. After making a mode switch, the kernel makes a decision on the next process to execute. When this decision is made, a context switch is made to the selected process. See Listing 1.1 for a high level pseudo code representation of this primitive

The kernel process decision currently uses several priority queues, four priority levels with two sub-queues. Every priority level has a `READY` and a `DONE` queue. Initially, all process in a priority level are in the priority level's `READY` queue and after execution they are moved to the `DONE` queue. In order to decide which process next, the kernel dequeues the next process on the highest non-empty `READY` priority queue. If all `READY` queues are empty, the `DONE` queues are moved to the `READY` queues and the kernel tries to select a process again.

A context switch is made after the kernel selects the next process. If there is a process running, the registers are saved on the stack and it's stack pointer is moved to the PCB. It's state is marked as `STATE_READY` to indicate it's state has been saved and is capable of executing again. Next, the selected process' stack pointer is restored. If the process is in the state `STATE_STOPPED`, meaning it has never been run before, the kernel changes the state to `STATE_RUNNING` and executes the process by returning from the exception. On the other hand, if the process has been run before, meaning that it is in the ready state, the data and address registers are restored from the stack. In this case, the method returns normally. Execution continues at the same it left off.

Listing 1.1: Pseudo code for `release_processor()`

```
int release_processor() {
    mode switch to kernel mode using system call exception

    if all queues are empty:
        move DONE queues to READY queues

    dequeue next process from highest non-empty priority queue
    if there is a currently running process:
        save all of the registers on to the process stack
        save stack pointer to PCB
        set state to STATE_READY

    restore next process stack pointer
    if next process state is STOPPED:
        return from exception using the next process exception frame
    else if state is READY:
        restore registers from stack

    mode switch to user mode using system call exception frame
    return RTX_SUCCESS
}
```

1.3.2 Set Process Priority

The `set_process_priority(...)` and `get_process_priority(...)` primitives are used to set and get, respectively, the priorities of the specified process ID.

Setting the process priority is shown as pseudo code in Listing 1.2. First, the values of the process ID and the priority are saved to registers as a mode switch to kernel mode needs to be performed. These values are retrieved after the mode switch. After the mode switch, a check is performed to ensure that the ID and priority is valid. If it isn't, `RTX_ERROR` is returned. Otherwise, another check is made to see if the ID belongs to the currently running process and that there is a running process. If this is true, the priority of the running process is simply changed and `RTX_SUCCESS` is returned. If it is not the currently running process, the process is first removed from the appropriate queue based on its current priority level. Then the priority is updated to the new one and the process is enqueued onto the queue associated with the new priority with the same queue type as the one it was removed from. Finally, the function returns `RTX_SUCCESS` to indicate successful completion.

Listing 1.2: Pseudo code for setting the process priority

```
int set_process_priority(int pid, int priority) {
    move pid and priority into separate data registers
    mode switch to kernel mode

    retrieve pid and priority from the data registers

    if invalid PID or invalid priority:
        return RTX_ERROR

    if running process is the process to change:
        update running process to the new priority
    else:
        dequeue process from appropriate queue
        update priority of that process
        enqueue process to its new priority queue

    mode switch to user mode

    return RTX_SUCCESS
}
```

1.3.3 Get Process Priority

Compared to setting the priority level, getting a process' priority is straight forward as can be seen in Listing 1.3. Similar to setting the priority, the process ID is moved into a data register and system is switched into kernel mode. After retrieving the process ID, a check is made to ensure that the process id is valid. If it is invalid `RTX_ERROR` is returned. The calling process should perform a check of its own to ensure that the returned value is not a negative which indicates an error. Finally, the process is looked up on the process table and the priority is returned from that.

Listing 1.3: Pseudo code for getting the process priority

```
int get_process_priority(int pid) {
   move pid into a data register
   mode switch to kernel mode

   retrieve pid from data register

   if the pid is invalid:
      return RTX_ERROR

   get process from the process table using pid

   mode switch to user mode

   return process priority
}
```

1.3.4 Request Memory Block

The `request_memory_block()` and `release_memory_block()` primitives are used by the operating system to allocate and deallocate memory blocks for processes.

Requesting a memory block is shown in pseudo code in Listing 1.4. First a mode switch to kernel mode is made. After the mode switch, a check is made to see if any memory blocks are available. If there are no available memory blocks, the process is moved to a blocked state and the processor is released. If memory is available, the memory block is popped off of the free list. The block index is retrieved and the appropriate memory allocation bit is set high. Finally, the memory block address is returned.

Listing 1.4: Pseudo code for requesting memory blocks

```
void* request_memory_block(){
   mode switch to kernel mode

   if no available memory blocks:
      running process state = blocked on memory
      release_processor()
   else:
      pop block of memory off the free list
      get block index
      set corresponding bit in memory allocation field to 1

   return address of block
}
```

1.3.5 Release Memory Block

Releasing a memory block is shown in the pseudo code in Listing 1.5. First a system call is made using the address of the memory block as a parameter. After the mode switch is performed, a check is made to determine the index in the allocation field of the block. This index is used to see if this block is currently allocated. If it is not then the system returns `RTX_ERROR`. If the block has been allocated then the block is pushed back on to the free list and that bit in the memory allocation field is set back to 0. Next, a check is made to see if there are any processes currently blocked on memory. If any are found, the first blocked process has its state set to ready and `RTX_SUCCESS` is returned. If there are no processes blocked on memory then `RTX_SUCCESS` is returned right away.

Listing 1.5: Pseudo code for releasing memory blocks

```
int release_memory_block(void* memory_block){
    mode switch to kernel mode

    if memory block is not allocated:
        return RTX_ERROR
    else:
        push block back on to free list
        set correct bit in memory allocation field to 0

            if there are processes blocked on memory:
                set state of first blocked process to ready

        return RTX_SUCCESS
}
```

1.3.6 Send Messages

The primitives `send_message(...)` and `receive_message(...)` are used by processes to send and receive messages between them. Each process has its own message queue which is setup upon process initialization. When a process sends a message it is enqueued to the message queue of the receiving process. When a process attempts to receive a message it polls its own message queue.

Listing 1.6 describes the functionality of `send_message(...)`. First, a system call is made with the process id of the receiver and a pointer to the message envelope. The system call then switches into kernel mode. Once the mode switch is complete the message is then pushed on the end of the recipient's message queue. There is no size limit on the message queue so this will never cause an error. A check is then made to see if the recipient process is currently blocked on receiving messages. If this is the case the state of this process is set to ready. Then `RTX_SUCCESS` is returned.

Listing 1.6: Pseudo code for sending messages

```
int send_message(int process_id, void* message_envelope){
    mode switch to kernel mode

    push message on to end of message queue

    if receiving process is blocked on messages:
        set state of receiving process to ready

    return RTX_SUCCESS
}
```

1.3.7 Receive Messages

Pseudo code for `receive_message(...)` is shown in Listing 1.7. A system call is made to switch the system into kernel mode. In this system call a pointer for storing the process id of the sender is passed in as a parameter. Upon switching to kernel mode the running process checks to see if its message queue is empty. If the message queue is empty the process is now blocked on messages. The processes' state is changed to reflect this and release processor is called. If the queue is not empty the message is popped off the queue, the process id of the sender is written to the `sender_id` pointer, and the message is returned.

Listing 1.7: Pseudo code for receiving messages

```
void* receive_message(int* sender_id){
    mode switch to kernel mode

    if message queue for running process is empty:
        running process state = blocked on messages
        release_processor()
    else:
        pop message off queue
        *sender_id = process id of the message sender
        return message
}
```

1.3.8 Delayed Send

The `delayed_send(...)` primitive queues a message to be sent after a specified delay. Messages are put onto a internal queue that holds all delayed messages regardless of which process they belong to. Messages in the queue are ordered by send time, with the soonest at the front of the queue. This way, the check to see if a message should be sent can be run in $O(1)$ time.

This function takes three parameters: the receiving process ID, a pointer to a memory block containing the message, and the delay time in milliseconds. After a mode switch to kernel mode, the parameters are checked to make sure there is a valid receiver id and delay value; invalid parameters return an `RTX_ERROR`. If this check passes then the message is inserted into the correct spot in the delayed message queue.

Each time the timer i-process runs, it checks the head of the delayed message queue to see if a message should be sent. If so, the message is dequeued and forwarded to the correct process and the head is checked again. The pseudo code for the delayed send is in Listing 1.8. The pseudo code and implementation information for the timer forwarding is shown below in the timer i-process section.

Listing 1.8: Pseudo code for delayed send

```
int delayed_send(int process_id, void* message_envelope, int delay):
    switch to kernel mode

    if (receiver_id is invalid || delay is invalid):
        return RTX_ERROR
    else:
        set message sender id
        set message receiver id
        set delay information
        insert into delay queue

        return RTX_SUCCESS
```

1.4 Processes

1.4.1 Null Process

The null process was implemented to adhere to the project specifications. It has a priority of four and a process ID of zero. The process method is defined as an infinite loop that constantly calls `release_processor()`. It is added to the process table through a call to `init_processes(...)`. As this process is required to always be at process ID zero, care should be taken to ensure that this is not overwritten by other processes in the initialization methods. This process is outlined in Listing 1.9.

Listing 1.9: Pseudo code for the null process

```
void process_null() {
    while (true):
        release_processor();
}
```

1.4.2 KCD Process

The `process_kcd()` blocks until a message is received. Once it receives a message, the message type is checked to see if it is either key input or command registration. If it is neither, the message is released.

When the message type is key input, the message is checked to see if it matches to a previously registered command. If a match is found the message is forwarded to the registered process for that command.

When the message type is a command registration, the PID of the registering process is saved along with the command itself. The command character is passed in the message body. The pseudo code for the KCD can been seen in Listing 1.10.

Listing 1.10: Pseudo code for implementing the KCD Processes

```
void process_kcd():
    while(1):
        receive a message
        if (message type = key input):
            if (message body = registered command):
                send message to registered process
        else if (message type = command registration):
            save the PID and command to be registered
        else:
            release message
```

1.4.3 CRT Process

The `process_crt_display()` blocks until a message is received. If type is correct, then the message data is sent to the UART and the message is released. Pseudo code for the process can be found in Listing 1.11.

Listing 1.11: Pseudo code for the CRT process

```
void process_crt_display():
    while(1):
        receive a message
        if (message type = output || message type = key input):
            send message data to uart
        release message
    }
}
```

1.4.4 UART I-Process

The UART i-process is run in response nice to both transmit ready and receive ready UART interrupts. After either of these interrupts fires, a context switch is immediately made to the UART i-process. Once it runs, the UART status register is checked to determine whether a read or a write is to occur.

When reading in data, all characters are appended to a buffer until a carriage return is read in. At this point, a null terminating character is appended. A check is made to determine the first value in the string buffer. If it is a percent (%), a message is created with the string buffer copied into its data field. This message is sent to the KCD. If it is an exclamation mark (!), the string buffer is sent to the UART debug decoder. After this, the character read in is echoed out again through a transmit interrupt request and interrupts are re-enabled.

If it is a write, a character is written to serial port and transmit ready interrupt is masked. If a carriage return was written, then a newline character is also queued to be written. A special case is considered for when the message type is specified to not require any newlines. Pseudo code for this process is below as Listing 1.12.

Listing 1.12: Pseudo code for the UART I-Process

```
void i_process_uart():
   while(1):
      determine uart state

      if(state = read):
         read in character
         append character to string buffer
         if (character = carriage return):
            append null to string buffer

            if (buffer[0] = '%'):
               create a new message
               copy string buffer into message data field
               send message to KCD
            else if (buffer[0] = '!'):
               send string buffer to uart debug decoder

         echo the in character by triggering a transmit interrupt

         enable interrupts
      else if (state = write):
         write character to serial port
         mask transmit ready interrupt
         if (character = carriage return AND requires a new line):
            print newline

      release processor
```

1.4.5 Timer I-Process

The `i_process_timer()` process runs immediately in response to timer interrupts. Upon receiving an interrupt, this process performs two tasks. First, it increments its internal time counter. Next, the delayed messages queue is checked to see if there are any messages to send out. All messages that were scheduled to be sent at the current time or sooner are sent. At this point, this process will relinquish control to the processor. See Listing 1.13 for the pseudo code for this process.

Listing 1.13: Pseudo code for the UART I-Process

```
void i_process_timer():
    timer = 0
    while(1){
        release processor
        increment timer

        if(delayed messages queue is not empty):
            while (queue head send time <= current time):
                remove message from delayed messages queue
                send message to original receiver
```

1.4.6 Wall Clock Process

The `process_wall_clock()` is used to display the current OS time on UART1. This clock is given an initial time by sending the `%WS` command and is updated every second. Clock time is printed until the `%WT`. Internally, the current time is stored in seconds.

The wall clock receives time updates by repeatedly sending delayed messages to itself. Every time it receives one of these messages, it updates its internal clock, prints out the time, and sends itself another delayed message. The delayed messages are sent with a one second delay. Once the internal clock reaches 86400 seconds (24 hours) it is reset to 0. These messages are only sent if the wall clock has been started with the `%WS` command.

When the wall clock receives a message from the KCD process it parses the command and performs one of two actions. If the command was `%WS`, it parses the time from the command, sets the internal clock, and sends itself a delayed message. The user input is validated, and if it does not pass the clock is not changed.

Pseudo code for the wall clock process can be seen at Listing 1.14.

Listing 1.14: Pseudo code for the wall clock process

```
void process_wall_clock():
    while (1):
        receive message
        if (sender_id = wall clock pid):
            increment clock

            if (clock = 86400):
                reset clock

            if (clock is currently on):
                send a delayed message to itself with a delay of one second
                convert clock to hours, minutes and seconds
                convert hours, minutes, seconds to time string
                send time string as message to CRT

        else if (sender_id = KCD pid):
            if (%WS):
                if (clock is currently off):
                    send a delayed message to itself with delay of one second

                parse string and validate input
                store hours, minutes and seconds
            else if (%WT):
                turn off clock
        release message
```

1.4.7 Set Priority Command Process

The `process_set_priority_command()` is a process that allows the user to change the priority level of any process through a command. This process accepts data in the form `%C process_id priority_level`. When the process is initialized it registers the `%C` command. It then waits to receive a message. Upon receiving a message it skips the `%C` characters and begins parsing the data. While parsing, the process id of the process whose priority is to be changed and the new priority level are extracted. Checks are made to ensure that proper process ids and priority values were provided and that nothing else was entered after the priority value. A kernel function call of `set_process_priority(...)` is then made using the parsed process id and priority value. This function call is responsible for the actual changing of the priority level. The memory block and the processor are then released. Pseudo code for this process can be seen at Listing 1.15.

Listing 1.15: Pseudo code for the set priority command process

```
void process_set_priority_command():
   register %C Command

   while (1):
      receive a message

      skip %C in message

      parse process id
      validate process id

      parse priority level
      validate priority level

      validate line endings

      call set_process_priority()
         update the priority level of the specified process

      release memory block;
      release processor;
```

1.4.8 Test Processes A, B, C

The system also has three test processes implemented. These processes are used to stress test our system by depleting memory blocks. `Process A` requests a memory block and on receiving one sends a message to `Process B`. `Process B` receives the message and then sends the message to `Process C`. `Process C` then gets a message and checks if the number of messages it has received is divisible by 20. If it is it prints out "Process C" to the UART1. Then the process hibernates for 10 seconds. All of these processes loop forever inorder to deplete memory. In order to have these processes work properly within our system `Process C` was assigned a priority of 1 and the other two processes were assigned a priority of 2.

1.5 Software Interrupt Handler

The system uses a single software interrupt handler for performing context switches into kernel mode for kernel functions. It is initialized in the 0th location on the trap table. The trap is only called from the function `do_system_call(int call_id, int* args, int num_args)` which is used to abstract common code used by all the primitives. The software handler itself that is called by the trap mnemonic is `system_call()` which retrieves the parameters from data registers 1 to 4. It calls the appropriate kernel level function based on the call id parameter. The call ids are defined in an enumerator.

The software interrupt handler, as described in Listing 1.16, is installed in vector 0 of the trap table. The operation of this interrupt handler retrieves the parameters from the data registers, saves the rest of the user process state into the stack, and calls the appropriate kernel level function. The parameters retrieved are passed in as arguments for the function. The function return value is stored in one of the data registers to be passed back to the function that called the trap. Finally, the function returns from exception to switch back from kernel mode into user mode.

Listing 1.16: Software Interrupt Handler description

```
void system_call() {
    disable interrupts

    retrieve call id and arguments from data registers

    save current user process state

    use call id to select the appropriate kernel level function
    pass in the arguments as parameters to the function

    restore user process state

    load return value into a data register

    return from exception
}
```

1.6 Hardware Interrupt Handlers

Two hardware interrupt handlers were implemented. The first is for the timer interrupts and the second is for the UART for keyboard input from the user. They both use very similar logic. All of the data and address registers are moved onto the stack after interrupts are disabled. Then the processor is preempted with the the respective interrupt process selected as the next running process. In the case of the timer interrupt, the interrupt event is acknowledge to prevent the interrupt from firing multiple times for the same event. After the interrupt process releases the processor, and the processor returns back to the interrupted process, the data and address registers are restored from the stack and the interrupt handler returns from the exception. See Listing 1.17 for the pseudo code description of the two interrupt handlers.

Listing 1.17: Hardware Interrupt Handler Pseudo Code

```
void interrupt_handler() {
    disable interrupts
    move data and address registers onto the stack

    acknowledge the interrupt
    preempt processor with the appropriate interrupt process

    restore data and address registers from the stack

    return from exception
}
```

Interrupt service routines are loaded into the interrupt vector table and initialized with the required settings in the initialization period of the system startup. This is described in Section 1.8.

1.7 Hot Keys

A total of five hot key commands were implemented to assist in debugging and testing of the system and user processes. A hot key decoder function was used to interpret the inputted command from the UART interrupt process. The interrupt process checks the first character for an exclamation mark ('!') and calls the hot key function directly with the input string as a parameter. See Table 1.2 for a summary of the hot key strings and descriptions. All of the code related to the debug hot keys are wrapped in `#ifdef _DEBUG_HOTKEYS` and `#endif` to ensure that in normal operation of the system that this code will not interfere.

Table 1.2: Hot Keys and Descriptions

Hot Key Command	Description
!RQ	Print ready processes and priorities
!BMQ	Print processes blocked on memory
!BRQ	Print processes blocked on receiving messages
!FM	Print the current number of free memory blocks
!M	Print the last *i* messages sent where *i* is the debug message log size

Each of the debugging functions are kernel level calls as they require access to data structures in the kernel. This is done using the same soft interrupt method as the the rest of the kernel routines. The hot key command decoder uses a similar parsing method as the `process_wall_clock()` process to interrupt which kernel routine to run. The debugging processes related to the ready queues and blocked queues use the kernel print queue function as described in Listing 1.18. The specific queue debugging function simply passes in the relavent queue into the queue print function.

Listing 1.18: Kernel Print Queue Psuedo Code

```
int queue_debug_print(process_queue queue[]) {
   for each process in the queue:
      print process ID
      print process priority
}
```

The hot key, `!FM`, simply outputs the number of free blocks and the allocation bit field. The allocation bit field is described in Section 1.2.3.

The last hot key, `!M`, is the most complicated of the five. In each of the kernel functions related to message sending and receiving, a copy of each message is stored on a circular buffer that contains a data structure that maintains a record of the sender ID, receiver ID, message type, the first 16 bytes of data, and the time stamp of the message. Seperate circular buffers are used for the sent messages and received messages to assist in debugging if a message has been sent but not received. When the hot key is called, it goes through every message in both queues and outputs first all the sent messages and then all of the received messages. The number of messages store in the two queues can be changed by modifying the `DEBUG_MESSAGE_LOG_SIZE` definition.

1.8 Initialization

The RTX initialization happens in four phases: process, priority queues, interrupts, and the memory management system. Pseudo code for everything except memory management can be seen in Listing 1.19 while the pseudo code for memory management is located in Listing 1.20.

The process initialization occurs in `init_processes()` and is done in two stages. The first is an initialization of the test processes and the second is the initialization of the null and user processes. For each process, a PCB is created on the process table. This PCB is then added to the kernel's own process list but not yet added to a priority queue. Then, for each process in the kernel's process list, a stack pointer is setup and an exception frame is added to the process' new stack. The format/vector word portion of the exception frame is set to 0x4000 to represent a 4 byte aligned stack and the SR portion is 0x0000. The program counter in this frame is set to the entry point of the process.

Next, priority queues are initialized. The data structures are already allocated but are not initialized. To ensure proper operation, the queues are put into a consistent empty state. After the queues are setup, the processes are added to appropriate queue.

Then, the interrupts are setup. First, the vector base register (VBR) for the interrupts is set to memory location 0x10000000. Next, the soft interrupt for `system_call()` is installed to the VBR at vector 0.

Listing 1.19: Pseudo code for RTX initialization

```
void init() {
    init_processes()
    init_priority_queues()
    init_interrupts()
}

void init_processes() {
    init_test_processes()
    init_user_processes()
    for each process in process list:
        initialize stack pointer
        push PC portion of exception frame onto the stack
        push F/V and SR portions of exception frame onto the stack
}

void init_test_processes() {
    for each test process:
        setup PCB
        add to process list
}

void init_user_processes() {
    user process list:
        null process
        Process A
        Process B
        Process C
        Uart
        Timer
        CRT
        KCD
        Wall Clock Display
        Set Priority Process

    for each process in user process list:
        setup PCB
        add to process list
}

void init_priority_queues() {
    for each process:
        add process to correct priority queue
}

void init_interrupts() {
    initialize VBR to 0x10000000
    install system call ISR at vector 0
}
```

The final phase of the RTX initialization is the memory management phase. Memory blocks are allocated beginning at the start of free memory. The function call `init_memory(void* memory_start)`, as seen in Listing 1.20. A free list is used to implement linking and management of the memory blocks. Over-allocation and double deallocation prevention is handled by a bit field containing a bit for each memory block. When the memory is allocation the associated bit is set to 1 and for when it is released it is set back to 0.

Listing 1.20: Pseudo code for RTX initialization of memory management

```
void init_memory(void* memory_start){

    head of memory = start of free memory + (number of memory blocks - 1)*(size of
        memory block)

    for each block in free list:
        store address of next free memory block in current block

    initialize the memory allocation bit field to 0
}
```

1.9 Implementation

Chapter 2

Major Design Changes

Chapter 3

Measurements

Chapter 4

Leasons Learned