Section 201 - Group #004

ECE 354 – University of Waterloo

RTX Project Software Design Document

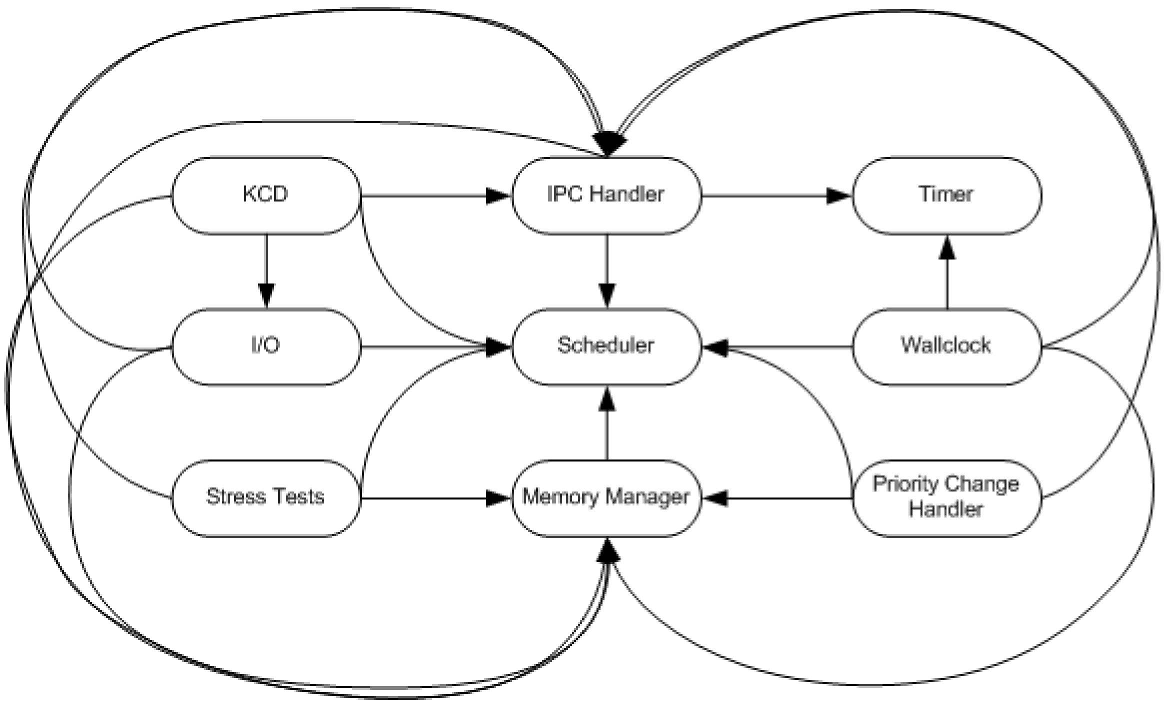
Group Members:

Zi Wei Zhang
Julie Laver
Adam A. Flynn
Michael A. Soares

RTX Project Software Design

Overall Design

The RTX design discussed in this document has been broken up into nine areas outlined in the dependency diagram below:



Scheduler: Handles process switching and preemption based on process priorities and ready/blocked statuses.

Memory Manager: Handles memory requests, as well as the allocation and de-allocation of memory throughout the RTX.

Timer: Keeps track of the internal system time in milliseconds.

Wallclock: Displays the current time based on the starting time specified by the user onto the display. It has the ability to reset/roll back if the time reaches 11:59:59.

IPC Handler: Handles interprocess communication for mutual exclusion through sending (regular and delayed messages) and receiving of messages.

Priority Change Handler: Changes the priority of processes and preempts them based on user input and/or calls made by other processes.

I/O: Handles predefined hotkeys, takes in user input from a keyboard, and displays it on the display. All input is later processed by the KCD.

KCD: Keyboard Command Decoder - Decodes user input and forwards all command calls to the appropriate processes.

Stress Tests: Used to test and verify the behaviour of the RTX when under heavy load situations.

Kernel Core

At a high level, the kernel core module of the RTX contains the functionality for initialization, process scheduling, and process switching. On boot, the loader calls the core module's initialization code which initializes the other system modules as well as initializes all processes to run. The core module then responds to calls to block/unblock processes as well as to switch the currently running process. This module makes all decisions surrounding what the next process to run should be. The module is also responsible for gracefully handling RTX shutdown when there are no processes to run (used for debugging).

Initialization

The process initialization code works by iterating over a series of hardcoded "process initialization blocks" (PIBs) which define basic information about the processes to be created such as PID, stack size, starting address, and priority. One PIB is created for each process to be run. These PIBs are iterated over and "process control blocks" (PCBs) are created for each one and put onto the appropriate process stack in the "new" state.

Scheduling

When the RTX tries to change processes, it looks for the first entry in the highest-priority non-empty ready queue. If no such process exists (typically, the null process will always exist in the lowest-priority ready queue), it terminates the RTX.

To ensure that the RTX is always able to be running even when there are no user processes running, there exists a process called the "null process". This process just loops indefinitely at a custom priority level which is lower than allowed by a normal user process. This guarantees that the null process will never be running when there are any ready user processes and that it will be preempted as soon as there is work that can be done.

Process Blocking

A process can be blocked for either waiting on memory or a message. When it is blocked, its PCB is removed from the ready queue and placed onto the respective blocked queue at the appropriate priority level. A process cannot be blocked for more than one reason at a time. If the blocked process is currently running, the RTX chooses a new process to run.

Unblocking a process can either be done in two ways. A process can be unblocked for a "reason" (i.e. memory or message), meaning that the highest priority process is taken off of the blocked queue for the given blocking reason. Alternatively, a process can be unblocked by PID, meaning it is taken off of whatever blocked queue it is on. In either case, if the currently running process has lower priority than the recently unblocked process, a context switch is performed. Typically, memory will unblock on reason because it does not care which process gets the memory and messaging will unblock on PID because the message is addressed to a specific process.

Preemption

Preemption is fairly simple in the RTX design. Most of the code to handle preemption lives inside the kernel core in the unblock_process() and unblock_process_by_pid() methods. If a process is being unblocked which is of a higher priority than the currently running process, the release_processor() primitive is called which will automatically handle context switching to the appropriate process.

The exception to this rule is the set_process_priority() primitive. It will preempt by calling release_processor() itself if the priority change should result in preemption (i.e. if the currently running process is going lower than the highest-priority ready process in the system or if another ready process is getting a priority higher than the currently running process).

Memory Allocation

The kernel core also contains the RTX's malloc() functionality which makes allows allocation of memory on the heap in arbitrary sizes for internal kernel data structures. This makes memory management inside the kernel easier as there is no dependency on the memory management module nor the risk of not having any available blocks. This functionality was put in the kernel core because it is required by all kernel modules and is required for RTX initialization (i.e. before memory management initialization has happened).

Implementation Details

Data Structures

process_state Enumeration

Describes the current state of a process. The distinction between new and ready is if the process has been started yet. A ready process can just return to the address in the stack pointer whereas a new process must be started by jumping to its starting address.

The possible values are:

- new
- ready
- blocked_mem
- blocked_msg

block_type Enumeration

Describes the reasons a process can be blocked for.

The possible values are:

- blocked_mem
- blocked_msg

Process Control Block

The PCB data structure is a struct which contains the following fields:

- UINT32 PID - process identifier
- process_state state - process state
- UINT32 priority - process priority
- UINT32 d[8], UINT32 a[8] - arrays to store data and address register values to be restored when context switching
- void * starting_addr() - function pointer to the starting address of the process
- pcb * next - pointer to the next element in the queue or null

These structs are chained together as a singly-linked list when appropriate.

Process Initialization Block

The PIB data structure is a struct which contains the following fields:

- int PID - process identifier
- int priority - process priority
- void * starting_addr() - function pointer to the starting address of the process
- int stack_size - number of bytes of stack space to allocate for the process
- pib * next - pointer to the next element in the list or null

These structs are chained together as a singly-linked list consisting of all processes to initialize on RTX startup. Every PIB corresponds to a running process in the system and a PCB is created with a process in the "new" state for each PIB.

Process Queues

There are 3 sets of process queues: ready, memory blocked, and message blocked. Each of these process queues are an array of 5 queues (one for each priority level). The queues themselves are are implemented as singly-linked lists of PCB structures. These queues are stored as global variables. When a new process is either created or is context switched out, its PCB is enqueued at the end of the appropriate process queue for its state and priority.

The PCB for the process which is currently running is taken out of its queue and put into a global variable called running_process.

Global Variables

PCB * running_process

Pointer to the PCB of the running process.

PCB * ready_queue[], PCB * msg_blocked_queue[], PCB * mem_blocked_queue[]

Arrays of PCB pointers for the process queues.

void * heap

Pointer to the current bottom of the heap. Incremented by malloc() every time new memory is allocated.

Primitives

int release_processor()

Relinquishes control of the processor from the current process, puts that process back onto the ready queue, and then searches the ready queue for the highest priority process to run. Saves the state (all register values) for the currently running process in memory and restores the state for the new process into the registers.

To safely save and restore state, it was necessary to ensure a minimal number of registers were clobbered. To do this, all register values are saved onto the stack (does not require any register-impacting instructions) and then are popped from the stack and put into memory. The new values for the registers are then pushed onto the stack. Immediately before jumping back into the user process, the values are popped from the stack and moved into registers. This series of awkward steps is necessary because a few address registers are used to calculate offsets when trying to read or write memory from the PCB which will clobber the contents of the registers themselves. Stack pushes/pops were chosen because they do not exhibit this property so the values can be safely stored and restored.

Like all primitives, this primitive has a handler which is accessed via a TRAP instruction. When TRAP is called, an exception stack frame is added to the stack which contains the return address the user came from. By saving/restoring all registers and executing an RTE instruction at the end, the hardware will automatically return the PC to the appropriate location by going back through the stack.

An exception to this is new processes which have not been started yet (since there's no address to return to). In this case, the process must be started and must be able to stop gracefully. To do this, an internal method called process_bootstrap() is called with the PCB of the new process. This jumps to the starting address of the process and lets it run. When the process finally finishes, it will return into this function which handles graceful termination of the process (i.e. removing the PCB from the priority queues, setting the running_process to null, and calling release_processor() to start up a new process).

Pseudocode

```
● Save registers to stack
● Restore registers from stack and into PCB
● Save the current process onto the correct queue
● Dequeue the highest priority ready process and assign it to the running_process global
● Restore stored register values to stack
● If the running_process is in the new state
    ○ Restore registers from the stack
    ○ Start the process
● Else
    ○ Restore registers from the stack
    ○ Use RTE to return to process execution
```

int get_process_priority(int pid)

Gets the priority of the process with the given PID. If the process does not exist in the system, returns -1. Returns the priority otherwise.

int set_process_priority(int pid, int priority)

Sets the priority of the process with the given PID to the given priority. This method can operate on any process in the system (running, ready, or blocked) and will cause preemption if required.

Pseudocode

```
● if PID <= 0, return error
● if PID is not in system, return error
● if process's current priority == priority, return success
● if process is not currently running
    ○ remove it from its current queue
    ○ change its priority
    ○ add it to the appropriate queue for its state and priority
    ○ if it ready and has higher priority than the running process, preempt
● if process is currently running
    ○ change its priority
    ○ if its priority is decreasing and lower than a ready process, preempt
```

Other Methods

This section describes other methods that are actively used by other kernel components. Effectively, these methods comprise the “public interface” of the kernel core module which is available to other kernel modules. Helper methods used exclusively within the kernel core module are considered “private” and are not documented.

void main()

The main() method is effectively the starting and ending point of the RTX. It is responsible for starting up the RTX on boot and gracefully shutting it down when there are no other processes to run. It calls the initialization code to initialize all of the PCBs and priority queues, calls initialization methods for all other kernel modules, and then uses release_processor() to start the RTX. During initialization and before any user process is started, a stub PCB with PID of -1 is created and set as the running_process so that release_processor() can handle startup gracefully.

void block_process(int pid, block_type reason)

Blocks the process given by pid for the reason of either “msg” or “mem”. This method will move the process from whatever queue it is in to the appropriate blocked queue. If the process is currently running, it will be preempted.

void unblock_process_by_pid(int pid)

Unblocks a process with the given PID. Puts the process into the ready queue for its priority. If the process is a higher priority than the currently running process, it will preempt. This is used when a particular process must be unblocked (typically used by IPC when a message arrives for a given PID).

void unblock_process_by_type(block_type type)

Unblocks the highest priority process which is blocked for the given block_type (i.e. memory or message), if such a process exists. If the unblocked process has a higher priority than the currently running process, it will preempt. This is used when a resource becomes available which should be allocated to the highest priority process waiting for it (typically used by memory when a memory block becomes available).

int get_current_pid()

Returns the PID of the currently running process.

int is_pid_valid(int pid)

Returns whether or not the given PID is a valid process in the system (returns 0 for false, 1 for true).

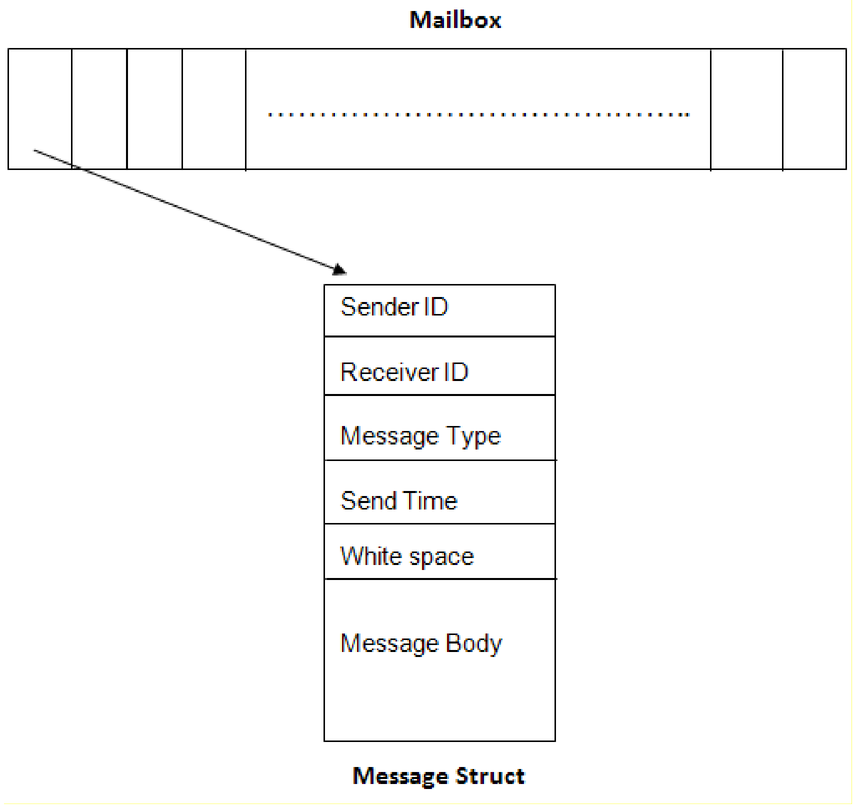
void * malloc(int size)

Returns a void pointer to the start of a memory block on the heap of the given size which can be used for kernel data structures.

Messaging (Interprocess Communication)

This section of primitives allow processes to achieve mutual exclusion by sending (normal, delayed) and receiving messages with each other or with the other components of the operating system when necessary. The messages are carried in envelopes with the structure and contents below:

- The sender's ID (int tx), which are process IDs
- The receiver's ID (int rx), which are process IDs
- Message type (int type)
- The send time (int send_time), which is used to store the time the message is to be sent (used in the delayed_send primitive)
- White spaces to ensure the actual message start at the right location (BYTE whitespace[48])
- The message (CHAR msg [64])



Sending Messages

A "mailbox" is created to store all the messages sent by the processes. The structure of the storage is an array with the size equal to the allowed memory blocks allocated. This way, the array would have enough spaces for messages requested. The "mailbox" stores the pointers to the messages (structs), which are stored somewhere in memory. The send message function takes in two parameters, which are the process ID of a process and a void pointer to the message. Thus, this function takes the message from the sender, fills in necessary information (header information, message, etc) in the message structure header and stores it in an empty entry in the mailbox. The process the message is for (i.e., the receiver) then gets unblocked by the scheduler to receive the message.

void init_messaging()

Pseudocode

- Create the message struct prior to this function (stored in the header file for access - rtx_msg.h).
- Initialize and give memory to the "mailbox" using malloc().
- Set the entries of the array (mailbox) to NULL before putting entires inside.
- Set the ISR to the correct memory address for trapping purposes.

int send_message(int process_id, VOID * envelope)

Pseudocode

- Atomicity and supervisor handling (section 1.7)
- Obtain values from data registers.
- Error check for null envelope - an error is outputted if the message envelope is empty.
- Cast the void pointer (passed in the send_message function) to the correct message type for usage
- Check for the validity of process ID (receiver) - if the PID does not exist, an error message is outputted.
- If the PID is valid, fill out the message header information (receiver & sender).
- Search through the mailbox and insert the pointer to the message into an empty spot in the "mailbox".
- Unblock the receiver.

Receiving Messages

This function will allow the process to receive a message from the "mailbox". The mailbox is scanned to determine if there is a message for a process. Furthermore, this is a blocking receive function. If a process attempts to receive a message that does not exist, the scheduler will block the process until a message comes in. This function takes If there is a message waiting, a pointer to the appropriate message is returned.

void * receive_message(int * tx)

Pseudocode

- Atomicity and supervisor handling (section 1.7)
- Check for whether the message or the sender information is empty - if it is not, return the pointer containing the message.
- Create a local pointer variable for return to the received function.
- Determine the process that is currently running.
- Traverse through the "mailbox" and find if there is a message whose receiver ID matches the process that is currently running since this is the process that is requesting retrieval. If there isn’t, break.
- If the message exists, get the message and set that entry in the mailbox to NULL.
- Shift the contents of the mailbox so there are no NULL spots in between messages.
- Trap out of supervisor mode.
- If there is nothing in the mailbox for that process, block the process from running and trap out of supervisor mode.

Delayed Message Sending

The delayed send message PIA is very similar to send_message with the exception that it allows the sender to specify a delayed time. The message will then be sent to the receiver when the delayed time is up. There are two additional functions that are created to ensure the delayed send functions properly. The delayed_send_handler function puts the delayed message into another mailbox called delayed_mailbox for storage. The reason that another mailbox is created is to ensure organization. Messages that are ready to send (i.e. reached the delayed time) is then moved to the actual mailbox. This is achieved by the check_existing_delayed_message function (described blow). This function is being called by the timer to see if the message is ready to be send.

int delayed_send (int process_ID, VOID * envelope, int delay)

Pseudocode

- Atomicity and supervisor handling (section 1.7)
- Obtain the values from the data register for usage
- Malloc memory to the delayed mailbox in init
- Cast the void pointer to the the appropriate message type pointer
- Check if the envelope given is empty and output an appropriate error message
- Check if the receiver (process ID) is valid and output an appropriate error message if its not valid
- Fill out the message struct header (receiver, sender, and send time (current time + delayed time)
- Find an empty spot in the delayed mailbox to store the message

void check_existing_delayed_message()

Pseudocode

- Check if any entry in the delayed mailbox needs to be sent
    - If such an entry exists, move the message from the delayed mailbox into the regular normal mailbox (by traversing through the regular mailbox and finding an empty location)
    - Shift the entries in the delayed mailbox after certain entry has been moved
    - Output an appropriate error message if there are no spaces in the regular mailbox
    - Check to see if the receiver is blocked. If it is, unblock the receiver by process ID
    - Update the highest priority process found if this one is higher than the previous best. If this does not apply, release the processor

Debugging

There are two functions which traverse through the regular and delayed mailboxes and print the sender, receiver, and process type. These are triggered by hotkeys or manual calls and help with debugging.

Memory Management

Memory management uses two queues of memory blocks to keep track of the available and used memory. Each memory block is contained in a memBlock structure with the following fields:

- void * startBlock (points to the start of the allocated memory)
- struct memBlock * next (points to the next memory block)
- int pid (contains the owning processes ID)

The queues were implemented using two linked list. In order to manage the queues, functions for enqueuing and dequeing processes have been implemented.

Request Memory

When memory is requested, a memory block is removed from the available queue and placed into the used queue. The pointer to startBlock, which is contained in the memory block, is returned to the requesting user. The number of blocks and the size of each memory block can be configured in the rtx_mem.h header file.

void init_memory()

- Get a block of memory of the configured size for each of the blocks (number of blocks x size of each block). This will be the memory used by other processes.
- Get a block of memory for the memBlock structs.
- Add all memory blocks (memBlock structs) to the available queue.
- Assign the startBlock in each memory block.
- Set the memory address of the release memory block and request memory block ISR.

void * request_memory_block()

- Switch into atomic mode.
- Loop while there are no available memory blocks.
    - Call the block_process function to add requesting process to the memory blocked queue.
- Dequeue a block from the available queue and enqueue the same block onto the used queue.
- Turn off atomic mode.
- Return the pointer to the block of memory contained within the struct into a data register.

Release Memory

When a block is released, it is removed from the used queue and placed back in the available queue. Only the process which owns the memory block can release it.

int release_memory_block()

- Switch into atomic mode.
- If the used queue is empty,
    - There is nothing to release so return.
- Remove the block from the used queue and zero the memory for future processes.
- Enqueue the block onto the available queue.
- Turn off atomic mode.

- Call the unblock_process_by_type() function to let it know there are available memory blocks.

void reassign_mem_block (void * memoryBlock, int newPid)

- Searches the queue for the block containing the specified memory block
- Changes the owning PID of the memory block to the new one

Input/Output Processes

The input/output processes were designed with simplicity in mind and were split up into three separate processes (excluding the uart_handler()/i-process): kcd(), crt(), and uart().

void uart_handler()

Process Description

uart_handler() (interrupt handling is initialized by uart_init()) is used to read in characters coming from the serial port (when other processes are interrupted by a user's keystroke) and to send said characters to the kcd() process for further interpreting. If a hotkey is entered, debug text is output in the debug console. From a high level perspective, the uart_handler()/i-process does the following:

- reads in user input (a character)
- outputs debug text when certain "hotkeys" are entered (when hotkeys are enabled)
- sends user input (if not a hotkey) to the kcd process via interprocess communication

Hotkeys

Hotkeys are handled by the uart_handler() and were were implemented are as follows:

- ! - prints a listing of all of the processes on the ready queue
- @ - prints a listing of all of the processes that are currently memory blocked
- # - prints a listing of all of the processes that are currently message blocked
- $ - prints a listing of all of the available memory blocks
- ^ - prints a listing of all of the currently used memory blocks and their owners
- & - prints the contents of the KCD buffer
- * - prints a listing of all of the available commands a user can enter

NOTE: hotkeys can be enabled by uncommenting #define _DEBUG_HOTKEYS in shared/dbug.h.

Initialization

uart_handler() is initialized by the function uart_init() in which the entire UART is reset, interrupts are installed, the baud rate is set, transmitting and receiving is set up, and the rest of the UART is set up and interrupts enabled.

void kcd() (Keyboard Command Decoder)

Process Description

kcd() is run when there is a request from another process (usually uart_handler()) to output characters for a user to see and to interpret commands. From a high level perspective, the KCD process does the following:

- process is unblocked when there is a new message for it
- character in the message is stored in a temporary variable and sent to the CRT process via interprocess communication (IPC) for output to the screen
- if the character is a line feed, a newline character is appended to the message and sent to the CRT process for output

- any input starting with the ‘%’ symbol is stored sequentially in a buffer and then interpreted/decoded as a command when a line feed character is sent to the KCD process; the buffer is cleared before storing the initial % character as well as after any line feeds

No additional initialization is needed for the KCD process.

Command Registration & Decoding

In the case of the RTX developed, command registration and decoding has been implemented in the easiest, yet most functional way possible. Processes that wish to register a command send a message to the KCD process via IPC with type ‘TYPE_REGISTER_CMD’ which stores this command and the command recipient’s PID in a linked list. The reg_cmd data structure used for the linked list is a struct which contains the following fields:

- UINT32 pid - process identifier
- BYTE cmd[KCD_BUFFER_SIZE] - character array for the command registered
- reg_cmd * next - pointer to the next element in the list or NULL

For example, the wallclock process uses the commands ‘%WS’ and ‘%WT’ to start and stop the wallclock, respectively, where ‘%WS’ takes some additional arguments. The only exception to command registration is in the case of a custom ‘%clr’ command which simply clears the UART1 screen, similar to what the ‘clear’ command on a Linux-based machine does. In this special case, the command is registered directly from within the KCD process upon its initialization.

When a user inputs the starting character of a command, ‘%’, into UART1, the KCD process begins to store the starting character as well as any subsequent characters into a temporary buffer (described above). The buffer is only cleared once the return key on the user’s keyboard is pressed and the command has been decoded as valid or invalid input, or if the user enters another starting character, in which case the buffer will be cleared and a new command can be entered.

Decoding is done in quite a simple manner. After a user enters a command and presses down on the return key, the KCD process does a linear search through the linked list storing all of the previously registered commands and compares each character in each command to what is stored in the KCD process’ buffer. The decoder ignores all arguments to a command (separated by a space) since it is assumed that the recipient process of the command will properly interpret said arguments (or discard them) as needed. Once a command has been properly decoded, a message containing the command and any arguments is sent to the respective process to be further interpreted. If an invalid command is entered, it is ignored; no indication is given to the end-user via UART1 of this (only in UART0 when debugging is enabled).

void crt()

Process Description

crt() is run when there is a request from another process (usually KCD) to output a string for a user to see. From a high level perspective, the CRT process does the following:

- process is unblocked when there is a new message for it

- sends each character in the string to the uart() process via interprocess communication to output to the screen

CRT simply acts as an intermediary process between the KCD and uart() processes so that full strings of text can be sent to it by any other process and then be output to UART1 by sending one character followed by the next to the uart() process.

No additional initialization is needed for the CRT process.

void uart()

Process Description

The uart() process is used to write individual characters to the screen/serial port. From a high level perspective, the uart() process does the following:

- process is unblocked when there is a new message for it
- waits for the serial port to become ready
- once ready, the character is written to the screen/serial port

While processes can send uart() messages directly, it is preferred that all text to be output to the user console is sent to CRT since uart() cannot handle strings of text directly. This was done by design as per the project specifications.

Initialization for this function is handled by uart_init(), described in more detail in 1.4.1.2.

Timer

The RTX timing service is based on a timer ISR which is triggered once every millisecond. That particular timer interval was chosen because it is the lowest level of granularity needed by any OS services (e.g., the wallclock operates per second and delayed messages operate per millisecond). Time is stored as a number of ticks since RTX startup in a global variable.

Every time the timer ISR is called, the interrupt is acknowledged and the global variable with the number of ticks is incremented by 1. On every tick, a function inside the messaging service, check_existing_delayed_messages(), is called to check the delayed message queue and to deliver any messages as and when needed.

The timer service provides a method called get_time() which returns the number of ticks since startup (as stored in the global variable).

User Processes

Wallclock

wallclock() is a user process that is called every time the clock ‘ticks’ or increments by one millisecond. When it first runs, it registers the %WS and %WT commands with the KCD process. When these commands are used, the clock is set to on (when %WS is used) or off (when %WT is used). When the wallclock is on, it sends itself a message delayed by one second to print out the current time. The original time is initialized when the %WS command is issued (in the format %WS HH:mm:ss).

- Register %WS by sending a message to KCD
- Register %WT by sending a message to KCD
- In a continuous loop:
    - Receive message:
        - Message will either contain a message string with one of the commands or will have a type of wallclock tick...
        - If message contains %WS command:
            - Check to make sure time parameter is valid
            - Convert hour, minute, and second values from string to decimal
            - Set the base time (the time at which the clock was started) and set the wallclock to on
            - If the wallclock was already on, set the new time and continue in the loop
        - If message contains %WT command:
            - Set the clock to off
        - If the message type is of type ‘TYPE_WALLCLOCK_TICK’ and the clock is on:
            - Calculate the current second, minute, and hour
                - Current_second = (set_second + current_time) mod 60
                - Current_minute = (set_min + (set_second + current_time) / 60) mod 60
                - Current_hour =(set_hour + (set_min + (set_second + current_time) / 60) / 60) mod 24
            - Convert decimal values of hour, minute and second to strings
            - Send a message to CRT with the time to output
    - If the wallclock is on, send itself a message of type ‘TYPE_WALLCLOCK_TICK’, delayed by one second and loop back.

Priority Change Command Handler

In order to facilitate processing of the %C command to change a process’s priority, a command handling process was created. This process starts up and sends a message to the KCD process to register itself as the command handler for “%C”. It then loops indefinitely waiting for messages from KCD to indicate that the user has entered that command. When a command is detected, it verifies that it has valid values and then processes it.

- Loop waiting for a message from the KCD
    - Search for a space in the 3rd character position

- Find the space after that, replace it with null to separate the 2 numbers
- Pass a character pointer for the 4th character position and the position after the null to a function to convert it to an integer
- If there were conversion errors, display an error message
- Else:
    - Pass the values to the set_process_priority primitive
    - If there was an error, display an error message
    - Else, display success

Null Process

The null process has the lowest priority of all of the processes in the RTX (priority 4) and is executed when no other processes are in the ‘ready’ queue. It consists of nothing but a single, empty while() loop that iterates until a process enters the ‘ready’ queue.

Atomicity and Supervisor Mode

All of the primitives in the OS run in atomic mode to avoid being preempted. The atomic up function causes interrupts to be masked, while the atomic down function re-enables them. In order to put the primitives into atomic mode, they first must be trapped up into supervisor mode.

To trap up into supervisor mode, all input parameters which will be needed in the function first need to be placed into registers. Using the asm function, the assembly instruction TRAP is executed. This executes a software interrupt and places the OS into supervisor mode and starts execution of the appropriate handler function (which is registered in various initialization functions). When coming out of supervisor mode, any return values must be saved to registers before returning from the handler.

Memory Map

The RTX's memory layout divides the OS up into several segments, each with a static starting address. It is assumed that no section will be long enough to overflow into another section's memory.

Starting Address	**Description**
0x10100000	RTX instruction code
After RTX code (dynamic; approximately 0x10104f00 in submitted code)	Start of heap; used for RTX data structures and memory blocks
0x10200000	RTX loader code
0x10300000	RTX test processes
0x10500000	RTX test process data area

Implementation, Testing, and Measurement Plan

Implementation Plan

Like every project, the implementation started out by understanding the main requirements. At first, we each briefly read the overall project description and then each part of the project in more detail to ensure each member of the team understood the purpose of the project even if the project were to be split in to parts. To begin, the first meeting was held in order to decide the responsibility of each member. We split the project into four basically equal modules. The four modules are:

- process scheduling (Adam)
- messaging IPC (Zi Wei)
- memory management and wall clock (Julie)
- I/O and hot keys (Michael)

The group would held several meetings from the beginning until the end of the implementation of each part of the project. The first meeting was used to discuss the requirements and clarify them with teaching assistants and the lab instructor, if necessary. We also ensured everyone had a fair share of work. Next, we brainstormed together, either on a white board or on a black board, to come up with effective designs for implementation. The brainstorming section took a great amount of time since we designed each person's section together so everyone could give input and share creative ideas; at the end, we all had a better understanding of what to do. After the design phase ended, all members worked in parallel for the actual implementation. When we implemented the sections in parallel, we were still in the same implementation phase so ideas could be shared and problems could be solved on the fly with input from every member.

After the implementation stage comes the testing and debugging. Each member would write appropriate tests to test his/her section. The tests usually start out simple to ensure general cases function and that the design indeed hold. Then, they are edited to include more complicated test cases to test each section in detail. During this time, the group would also be in the same area to help each other test and solve any problems. At the very end, each member contributed to the automated testing, which tests the OS in general with more details. Lastly, each member contributed to the documentation at the end of each part of the submission and also the final document.

Testing Plan

Apart from some basic testing of the various I/O processes, much of the RTX's functionality (e.g., memory, messaging, context switching, etc.) was verified by debug text output in the debug console during the execution of the RTX. The states of our process queues (i.e., ready, memory blocked, and message blocked) were verified by the use of the hotkeys described in 1.4.1.1, as well as available/used memory blocks.

All debug information returned by the RTX is wrapped in #ifdef...#endif precompiler blocks in the source. Every module has its own precompiler test macro defined for debug. By commenting out the various #define lines in a header file, it is easy to control the level of debug information printed to the console (and which modules are relevant to debugging) which makes debugging much easier.

Stress Test Processes

The stress tests were implemented as per the specifications and pseudocode supplied. They are used to test and verify the behaviour of the RTX when under heavy loadsituations (i.e., memory block depletion or near depletion) using different priority combinations for each one of the three stress test processes.

Other Test Processes

The test processes implemented during the course of the project test a variety of the different features implemented in the RTX. Most of the features tested include, but are not limited to, requesting/releasing/protecting memory blocks, preemption, delayed and non-delayed sending/receiving of messages (including the order of sent/received messages), and blocking/unblocking processes.

The tests were designed using a master/slave architecture. Tests are initially started by a master function which sends out messages to various slave test functions to continue the tests. Depending on the test at hand, the slave functions could potentially send messages between themselves or back to the master test function when done to continue on with additional tests.

Results of each test were output with PASS/FAIL text in the debug console with a description of what the test was able to or failed to verify. This was especially useful during various debugging stages of the project and led to the discovery of many bugs and/or features missing from the RTX.

Measurement Plan

In terms of timing the primitives used in the RTX, since a get_time() function was implemented into the timer service, it was decided that said function would be used as a kind of stop watch; the time right before each called primitive was recorded as well as that right after the primitive returned. This process was repeated 200 times for each primitive (request_memory_block(), send_message(), and receive_message()).

Results

As described in the measurement plan above, below are the time measurements for the three primitives. All times are in microseconds and have an error of ± 10 microseconds. By inspection, it can be seen that these numbers are what would be expected for each of these primitives. The range between the min and max for each is low, meaning the results were consistent across all tests.

	Min (µs)	**Max (µs)**	**Mean (µs)**
request_memory_block()	450	480	571
send_message()	1430	1500	1470
receive_message()	580	610	602

Team Member Contributions

Adam Flynn

- kernel core module
- timer module
- test processes
- miscellaneous debugging

Julie Laver

- memory management
- wallclock process
- test processes
- atomic mode for primitives
- miscellaneous debugging

Michael A. Soares

- all I/O processes and i-processes
- hotkey implementation
- command registration and decoding
- stress test implementation
- implementation of various other tests (mainly messaging and the automated test files)
- miscellaneous debugging (i.e., wallclock process, interrupt handling, automated test files, etc.)
- compiling of source files and compilation of README for submission

Zi Wei Zhang

- all messaging primitives - send, receive, and delayed_send
    - other private functions related to the primitives (i.e. check_existing_delayed_messages, atomic functions, printing functions for mailboxes)
- testing (message tests) and debugging
- automated test file (messaging)

In addition to the above, all team members contributed equally to the compilation of this report and all other intermediate design documents submitted.

Major Design Changes

As implemented, the RTX did not require many design changes. The implementation of the RTX was planned out from the very beginning of the project, taking into account all of the primitives and other aspects of the project that were required, whether or not they were to be implemented at a certain point in the project's progression. This was done to avoid having to make any drastic design changes to the RTX mid-way through the project which could have potentially affected its functionality at any given moment. As such, this section outlines two of the major design changes that were made to improve upon the RTX's initially planned out functionality.

Send and Delayed Send Message Primitives

When the original functions were implemented, the order the messages sent were not necessarily the same as the order they were received. In the messaging test, one of the major tests is to verify the ordering of messages as they are received. For example, one of the tests verifies the preemption aspect of sending a message. The sending primitive is preempted if the receiving process is blocked waiting for a message and has higher priority. If not, the, sender continues sending. Thus, the test verifies the orders the messages arrive in and are sent. In order to make this function properly, one major design change was applied to the mailbox. Originally, after the message was taken out of the mailbox or the delayed mailbox, the contents were left as they were with NULL spaces scattered. The new design shifted the contents of the mailboxes (regular and delayed) so all NULL spaces are at the end of the mailboxes.

Wallclock Process

In the wallclock's original design, the timer ISR would send the wallclock process a message of type 'TYPE_WALLCLOCK_TICK' every 1000 milliseconds/one second which would indicate to the process to output the current time. Architecturally, this didn't make much sense because it led to the constant use of a memory block for a 'tick' whether or not the wallclock process was supposed to be outputting the time. The ISR should not need to be aware of the existence of a user process depending upon it. Thus, 'ticks' were no longer sent by the timer ISR in the final design. Instead, at the end of each iteration of the wallclock process, the process sends itself a message of the same type defined above, but delayed by one second, so that the dependence of the timer ISR is passed on to delayed_send() primitive as it should be.

Lessons Learned

One of the major difficulties we all realized for this project is the difficulty of debugging. Since no good debugger exists for this infrastructure (we are working with the bare hardware and putting software on top), it is difficult to find out the cause of a problem when it arises. The only effective way to debug was to use the rtx_dbug_outs() function to print information that may inform us of the problem. For example, printing the address, values of a register, queues, etc. to display necessary information to the problem. However, this technique resulted in many instances of the rtx_dbug_outs() function being called, which lead to poor organization. For example, turning on debug text in our core module also enabled debug text in many other places which were not always useful, which is why it was difficult to pinpoint problems. Also, JanusROM running on the Coldfire boards could not keep up with the amount of debug text that our RTX was ouputting and sometimes led to the system freezing up. From this, we learned that when working with a system that is difficult to debug and if debug text is to be output, it needs to be organized properly. Having a good organization of debug text and code will assist in debugging more efficiently.

In order to achieve efficiency for testing and debugging, instead of just using the actual Coldfire board in the laboratory, we also tried to test and debug with cf-server and the emulator. There were definitely some problems using both the emulator and cf-server. Though the cf-server connects to the actual board through ecelinux, the speed differs depending on the number of users present. Connections were impossibly slow closer to the project's submission time, which made testing and debugging very inefficient. From this, we learned to have more organization in terms of testing to either try to implement testing earlier or test on the actual board in the lab more frequently. The emulator, on the other hand, does not have connection speed issues. However, the emulator only emulated the functions of the Coldfire board. Sometimes, testing done on the emulator would not provide the same results as with cf-server or the actual board. For example, the RTX would crash or give the wrong results when using the emulator and then crash under a different set of conditions when using cf-server. The lesson learned from this is that although the emulator is a good additional and alternative for testing, results always need to be checked to ensure they are correct.

Throughout the project, we used SVN as a file repository and for source control. The repository was accessible using a terminal client as well as by using a full-blown GUI-based piece of software called 'TortoiseSVN'. This repository helped the group manage different versions of files to allow us to work in parallel. Furthermore, older file revisions could be easily obtained if needed. Having a user-friendly and great file repository system definitely provided good organization and increased efficiency.