RTX DOCUMENTATION

GROUP 23
Nathan Woltman
Justin Gagne
Cody Chung
Evet Dinkha

Table of Contents

Chapter 1 – P1 IMPLEMENTION

1.1 Process Switching

This function handles switching from the old process to the newly selected one. If the new process is in the NEW state, the old process is configured if it is not the same as the current process and is not NEW. Old processes are configured by updating their stack pointer and placing them in the ready queue if their current state is RUNNING or INTERRUPTED. Then, the new process's state is set to RUNNING, and the stack pointer is updated to grab the new process's stack state. Finally, *__rte()* is called to pop the exception stack frame. The steps for process switching, when the new process's state is not NEW, is almost identical. The only changes are that nothing happens if the new process is the same as the old one, and the exception stack frame does not need to be popped to run the new process.

1.2 Memory Management

1.2.1 Requesting a Memory Block

If a process requests a memory block and there is a memory block available (i.e. the heap is not empty), the process can use the memory at the address returned by the result of popping the first block from the heap. Otherwise, the current process's state is changed to BLOCKED, the process is added to the blocked queue, and the processor is released.

1.2.2 Releasing a Memory Block

When a process releases a memory block, the specified block is pushed back onto the heap. If there are any processes in the blocked queue, the first blocked process is popped off the blocked queue and placed back into the ready queue (because now there is memory for it to use, so it may continue where it left off). The processor is then released, giving the recently unblocked process a chance to run if its priority is high enough.

1.3 Process Priority

1.3.1 Getting the Process Priority

Returns the current priority of the process with the specified pid.

int get_process_priority(int pid)

1.3.2 Setting the Process Priority

Allows user processes to change their own priority, or that of any other user process. The priority must be valid, and no unnecessary work is done if the new priority is the same as the current priority. If the process is currently in the ready queue, it is removed from the sub-queue associated with the old priority, then added to the sub-queue associated with the new priority. The process's priority attribute is then changed and the processor gets released.

int set_process_priority(int pid, int priority)

1.4 Problems Encountered

1.4.1 Memory Management

Our first task for coding the RTX was to build a memory management system that would make it easy to allocate and deallocate memory. It didn't take long for us to decide to use a forward linked list as a data structure for our heap, but it was challenging to make it generic so that any object could be added to the list. Due to the lack of generics in C, we had to come up with a way to generalize the code. We could have used macro definitions, but we decided instead to use type casting to cast input objects (i.e. memory blocks) to list nodes that could be added to the list. It also took some thought as to whether we should allocate 128 bytes or 132 bytes for each memory block. The manual stated that each block should be a minimum of 128 bytes, but we were unsure if the 128 bytes accounted for the 4 byte pointer indicating the beginning of each memory block or not. We decided to use 128 bytes for each memory block, with 4 of those bytes being used to locate the remaining 124 bytes of memory space.

1.4.2 Sharing Variables Across Different Files

Initially, we struggled to share variables across different files. This was required so that we could access our ready and blocked queues from both k_memory and k_process source files. Searching online and looking through given code, we realized we needed to use the “extern” property to declare the variables to solve our problem.

1.4.3 Preemption

One of the issues that we came across in Part 1 was determining how to implement preemption without interrupts. Eventually, it became clear that releasing the processor would cause preemption if implemented correctly. Thus, we needed to release the processor after changing a process’s priority, after requesting a memory block and blocking a process, and after freeing a memory block and unblocking a process. In each of these situations, preemption may occur to allow higher priority processes to execute.

1.4.4 Pointer vs. Non-Pointer Queue

We encountered two issues with our generic queue data structure, both related to using copies of data structures instead of copies of pointers to those structures. The problem was that in some functions we had created local variables and modified them, thinking that this changed the Queue. However, the local variables were only copies of Queues and QNodes, and after some debugging, we realized we needed to use pointers to update the actual data structures.

1.4.5 Getting Code onto the Cortex M3

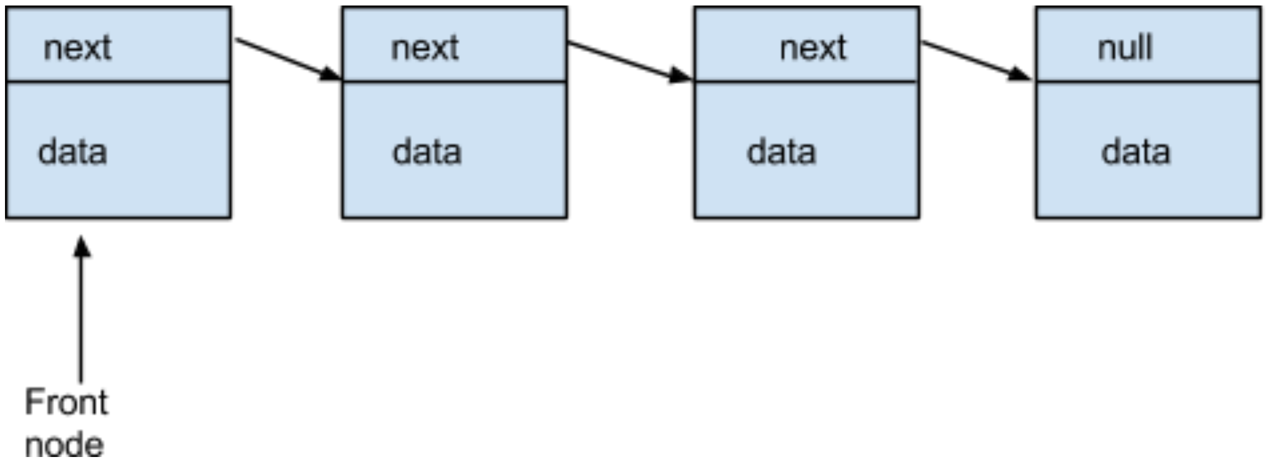
When we went to flash the code onto the Cortex M3, we were getting errors saying that the code could not be flashed successfully. Another student faced the same issue and recommended copying our source files into a new project and trying to flash the processor again. This approach fixed our problem.

1.4.6 Pointer Initialization

When we ran our program on the simulator, everything was fine; however, running our code on the hardware caused Hard Faults. The issue ended up being that we assumed pointers were initialized to NULL. It turns out that the simulator initializes pointers to NULL, but the hardware does not. Thus, we had to add initialization functions to our data structures to initialize all pointer values to NULL, else face the dire consequences of trying to access invalid pointers.

1.5 Generic Data Structures

1.5.1 Linked List



We used a *forward linked list* to keep track of the memory in the heap. The forward list structure itself contains only a "front" pointer that points to the first node in the list, which saves on memory from a normal list implementation which has pointers to the front and back of the linked list. In the case of the heap, we use this pointer to easily request a new memory block (i.e. pop_front), and to easily release a memory block (i.e. push_front). Only four methods were needed for the implementation of the linked list:

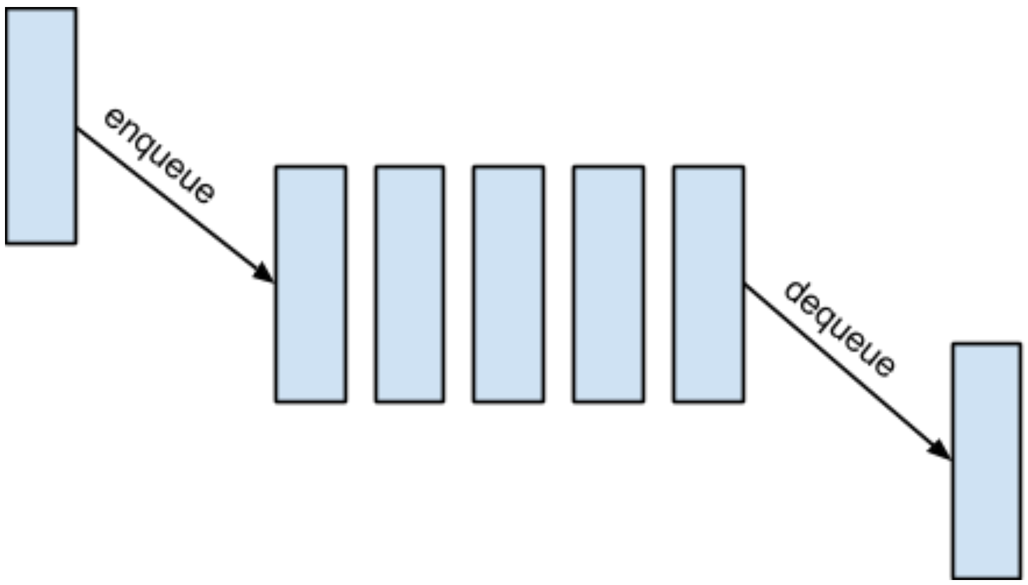
1. *void init (ForwardList* list);*
   Initializes a forward list by setting its pointer to the front of the list to *NULL*.

2. *int empty (ForwardList* list);*
   Returns 1 if the list is empty, 0 otherwise.

3. *ListNode* pop_front (ForwardList* list);*
   Removes and returns a pointer to the first node in the list.

4. *void push_front(ForwardList* list, ListNode* node);*
   Adds the input node to the front of the list.

1.5.2 Generic Queues



We use generic queues for our blocked queue and for each queue within our priority queue. As a typical queue, it is a FIFO structure. Our queue structure defines four methods:

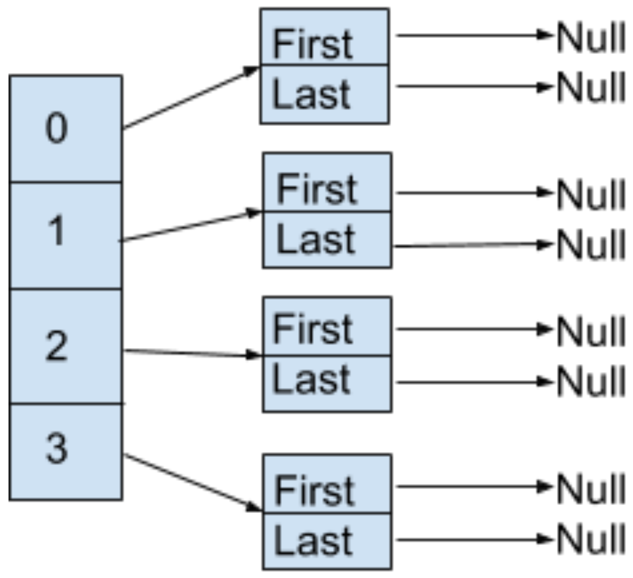
1. *void init_q (Queue* queue);*
   Initializes the queue by setting its pointers to the front and back of the queue to *NULL*.

2. *int q_empty (Queue* queue);*
   Returns 1 if the queue is empty, 0 otherwise.

3. *void enqueue (Queue* queue, QNode* node);*
   Adds the input node to the end of the queue.

4. *QNode* dequeue (Queue* queue);*
   Removes and returns a pointer to the node at the front of the queue.

1.5.3 Priority Queue



Initialized Priority Queue

A priority queue structure is used for the ready queue. Our implementation of the structure is specific to the RTX project, in that there are only 4 priorities to keep track of in the system. Because there are a fixed number of priorities, the priority queue simply holds an array of 4 generic queues. Each of the generic queues in the ready queue will only contain PCBs that have a priority matching that queue's position in the ready queue's array. For example, all PCBs with the lowest priority will be stored in the ready queue's 4th generic queue (i.e. in code: *queues[3]* because the lowest priority is 3). In the proc_init function, each process (with a state of NEW) is added to its respective queue in the ready queue. There are four methods defined for the priority queue:

1. *void init_pq (PriorityQueue* pqueue);*
   Initializes the priority queue by setting the *first* and *last* nodes of the queue at each priority to *NULL*.

2. *QNode* pop(PriorityQueue* pqueue);*
   Removes the highest-priority node (i.e. PCB) from the ready queue, and returns a pointer to it.

3. *void push(PriorityQueue* pqueue, QNode* node, int priority);*
   Adds the input node (i.e. PCB) to the end of the queue with the given priority.

4. *int remove_at_priority(PriorityQueue* pqueue, QNode* node, int priority);*

Removes a specific node (i.e. PCB) from the queue with the given priority. We use this function to remove a PCB from the ready queue before putting it back into the queue with a new priority when we call *set_process_priority()*.

Chapter 2 – P2 IMPLEMENTATION

2.1 Interprocess Communications

2.1.1 Message Structure

The RTX supports a message-based Interprocess Communication (IPC). Messages are stored in envelope blocks, storing information about the sending and receiving processes, the next messages, and the type and contents of the message. The implementation of a message envelope is as follows:

```
typedef struct msg_envelope
{
        struct msg_envelope *next;
        U32 sender_pid;
        U32 destination_pid;
        int mtype;
        char mtext[1];
} MSG_ENVELOPE;
```

Due to the restrictive privileges set to the user process view, a message buffer was created to send and receive messages between two processes, with only the message type and contents being accessible. The implementation used is as follows:

```
typedef struct msg_envelope
{
        int mtype;
        char mtext[1];
} MSG_BUF;
```

2.1.2 Send Message Process

In the send_message primitive, a process id and message was passed in.

```
send_message (int process_id, void *message)
```

Since the process is only passed in the message buffer, the rest of the contents (the header of the message envelope) of the of the message envelop need to be saved for the kernel process view. The message is then added to the message queue of the receiving process. If the current process was blocked while waiting on a message, its state is changed to ready and the receiving procedure is added to the ready queue.

2.1.3 Receiving Message Process

In the receive_message primitive, a sender id is passed in.

void *k_receive_message(int* sender_id)

The primitive checks while the message queue of the current process is empty for messages and blocks the incoming message. The process is then added to the blocked priority queue and the processor is released to deal with preemption. If the process is not blocked, the envelope is removed from the process's message queue and then returned.

2.2 Timing Services

With the possibility of timer interrupts, the messages that are sent through the interrupt are saved in a delayed queue. When the time has expired, the messages are sent using this primitive:

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

In this primitive, the process id, the message envelope and the delayed time value is passed in. After the expiration (delay), the message is sent to the process_id.

2.2 Interrupt I-Processes

2.2.1 The UART I-Process

To consider UART0 interrupts, there are two handler primitives used. The first primitive is done using assembly.

__asm void UART0_IRQHandler(void)

In the UART0_IRQHandler, the registers being used are saved and restored. Within this function, the c_UART0_IRQHandler is called, and that function deals with the rest of the irq handling.

void c_UART0_IRQHandler(void)

This UART I-Process forwards the characters passed to the KCD. Additionally, when the user inputs a character, an interrupt is called that calls the CRT function and the characters are echoed on the CRT display. Within the UART I-Process, there are hot keys for the user to click. The hotkeys implemented in our solution are:

1. "!" hotkey for printing the processes and priorities of those priorities from the ready queue
2. "@" hotkey for printing the processes and priorities of those processes from the blocked priority queue
3. "#" hotkey for printing the processes and priorities of those processes in the blocked on receiving messages priority queue

For each of these hotkeys, if they are ever pressed, the method leads to a print helper function.

void print(PriorityQueue* pqueue)

In this print function, a priority queue is passes in, and while looping through the priorities, the corresponding processes and priorities are printed.

2.2.1 Timer I-Process

The timer I-Process is responsible for any hardware timer interrupts. After the time has expired, the timer I-Process has to deal with delayed messages.

void c_TIMER0_IRQHandler(void)

In this process, any incoming messages are received and saved in a delay queue. The i-process then checks if the time has expired. When the timer expires, all of the messages are sent to their appropriate destination.

2.4 User Processes

2.4.1 24 Hour Wall Clock Display

//Something has to be written here

2.5 Problems Encountered

2.5.1 Message Structure

When implementing the IPC primitives, one of the problems faced was how to implement the structure of the message envelope. The issue was whether to split the data into a kernel and user view or to include all of the details in the message envelope structure but hide the certain parts from the kernel view. The latter was what was chosen. The message envelope structure includes all of the necessary details mentioned in Section 2.1.1. Essentially, it is split into two sections: the header, the content only accessible to the kernel, and the message content, accessible to the user and the kernel. To deal with the accessibility, the header part of the message envelope can only be accessed through the use of offsets of the addresses.

2.5.2 Issuing CRT Interrupts

When the user inputs commands, it was unclear whether or not to wait until they finish their input (carriage return) to output their message or if it was necessary to output each character. After asking on the class discussion board, it was clarified that the expected output is that the character should be immediately echoed back to the user. To implement this expectation, the char was sent to the CRT was sent to the console through a message to be echoed back to the console.