

UNIVERSITY OF WATERLOO

Faculty of Engineering

REAL TIME OPERATING SYSTEM DESIGN DOCUMENT

Prepared by

Yasser Al-Khder, ID 20356957

Kal Sobel, ID 20322299

Shari King, ID 20328892

Peter Robertson, ID 20332330

2B Mechatronics Engineering

October 20, 2011

TABLE OF CONTENTS

1.0	OVERVIEW AND INTRODUCTION	1
1.1	DESIGN REQUIREMENTS.....	1
1.2	FUNCTIONAL OVERVIEW	2
1.2.1	PROCESS MANAGEMENT.....	2
1.2.2	PROCESSOR SCHEDULING.....	2
1.2.3	INTERPROCESS COMMUNICATION AND SYNCRONIZATION.....	2
1.2.4	STORAGE MANAGEMENT.....	2
1.2.5	INTERRUPT HANDLING FRAMEWORK.....	2
1.2.6	TIMING SERVICES.....	2
1.2.7	DEVICE DRIVER INTERFACES	2
2.0	GLOBAL INFORMATION	3
2.1	DATA STRUCTURES	3
2.1.1	MESSAGE ENVELOPES	3
2.1.2	QUEUES.....	5
2.1.3	PROCESS CONTROL BLOCKS.....	6
2.1.4	CLOCK STRUCT.....	7
2.2	VARIABLES.....	7
2.3	CONSTANTS	7
2.4	MEMORY MAP	8
3.0	PRIMITIVES	9
3.1	SEND_MESSAGE.....	9
3.2	RECEIVE_MESSAGE.....	9
3.3	REQUEST_MSG_ENV	10
3.4	RELEASE_MSG_ENV.....	10
3.5	RELEASE_PROCESSOR.....	11
3.6	REQUEST_PROCESS_STATUS.....	11
3.7	TERMINATE.....	12
3.8	CHANGE_PRIORITY	12
3.10	GET_CONSOLE_CHARS.....	13
3.11	GET_TRACE_BUFFERS.....	14
4.0	PROCESSES.....	15
4.1	SYSTEM PROCESSES	15
4.2	CONSOLE COMMAND INTERPRETER.....	16
4.3	SIMULATED UART PROCESSES	18

4.3.1	KEYBOARD UART PROCESS.....	18
4.3.2	CRT UART PROCESS.....	18
4.4	NULL PROCESSES	18
4.5	USER PROCESSES.....	19
4.5.1	KNOCK-KNOCK JOKE	19
4.5.2	TIME WARP	20
4.5.3	PONG.....	20
4.6	CLOCK PRIMITIVES	21
4.7	TEST PROCESSES A, B, C	23
4.7.1	PROCESS A.....	23
4.7.2	PROCESS B.....	23
4.7.3	PROCESS C.....	23
4.8	MESSAGE TRACE DISPLAY	23
5.0	INTERRUPT HANDLING	24
5.1	I-PROCESSES (INTERRUPT HANDLER).....	24
5.1.1	TIMER I-PROCESS (SYSTEM CLOCK).....	26
5.1.2	KEYBOARD I-PROCESS.....	26
5.1.3	CRT I-PROCESS	26
5.2	TIMING SERVICES.....	27
5.2.1	REQUEST_DELAY	27
5.2.2	TIMEOUT_I_PROCESS	28
6.0	KERNEL ENTRY	30
6.1	USER PROCESS / KERNEL INTERFACE.....	30
6.2	PROCESS_SWITCH	30
6.3	ATOMIC FUNCTION	31
7.0	INITIALIZATION.....	32
7.1	INITIALIZATION STAGE 1: QUEUE INITIALIZATION	32
7.2	INITIALIZATION STAGE 2: USER PROCESSES	33
7.3	INITIALIZATION STAGE 3: INTERRUPT PROCESSES	34
7.4	INITIALIZATION STAGE 4: BEGIN RTX OPERATION.....	34
8.0	IMPLEMENTATION.....	35
9.0	GLOSSARY	37
	REFERENCES	38

1.0 OVERVIEW AND INTRODUCTION

The overall RTX system should function as per Figure 1. The RTX will be divided into user processes and system (kernel) processes. It will be able to handle processor management, scheduling, initialization, inter-process communication, interrupt handling, and timing processes. The RTX will be able to communicate with the keyboard and CRT via a simulated UART environment.

There will be three test processes run within the RTX, as well as three processes written by us, and a wall clock. Both the keyboard and the CRT will have i-processes within the

RTX, which they will use to communicate with the simulated UART. As well, a

Console Command Interpreter (CCI) will be run which will interpret a user's commands and perform the underlying functions.

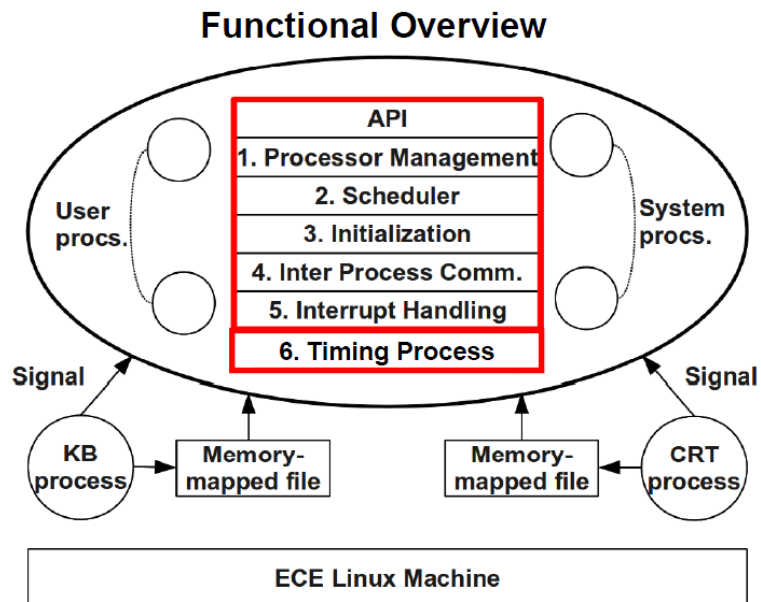


Figure 1: Functional Overview of RTX (Reidemeister, 2010)

1.1 DESIGN REQUIREMENTS

In order to simplify design, several assumptions must be made. It will be assumed that the RTOS will operate in a non-preemptive manner; several priority queues will be created to handle prioritization of user and system processes. As well, it is assumed that no process will be created or terminated after initialization. This will simplify the creation of processes, as mid-execution operations will not be considered. It is also expected that the processor loading will be a lean, mean, processing machine.

As a further simplification, all process priorities and memory allocation blocks are assumed to be static, and all processes are assumed to be non-malicious. This eliminates the need for complicated dynamic memory allocation, dynamic priority handing, and complex security protocols.

This RTX will also detect some errors. It must detect an attempt to send a message or set the priority of a non-existent process. It also must identify if an illegal priority level is specified. If the RTX detects either of these errors, it will return an error code. It is assumed that each of the processes can deal with whatever error code is returned.

1.2 FUNCTIONAL OVERVIEW

According to the requirements given, the RTX will deal with the processes described in the following sections.

1.2.1 PROCESS MANAGEMENT

As part of process management, the RTX deals with creation and management of user processes, as well as the ability to terminate processes that are not needed. Second, it deals with complete system initialization (hardware, creation of processes and stacks).

1.2.2 PROCESSOR SCHEDULING

The scheduler deals with the four priorities of the processes. It picks the highest priority process from the ready queue and starts the process of initialization.

1.2.3 INTERPROCESS COMMUNICATION AND SYNCHRONIZATION

Processes will need to send and receive information from other processes. They do this by using message envelopes.

1.2.4 STORAGE MANAGEMENT

A process will need to allocate and de-allocate memory as required by other processes. Most of memory is divided into arbitrary sized envelopes.

1.2.5 INTERRUPT HANDLING FRAMEWORK

Interrupts will be simulated by UNIX signals. A program will take an interrupt and halt current processes in order to execute the code specified by the interrupt.

1.2.6 TIMING SERVICES

The clock that provides timing will also be simulated by UNIX signals. As per instructions, it is known that the RTX requires at least 0.1s time resolution.

1.2.7 DEVICE DRIVER INTERFACES

The RTX will need to interface with the keyboard and screen to get input and output. This will be done using processes to simulate a UART.

Processor Management: Process States

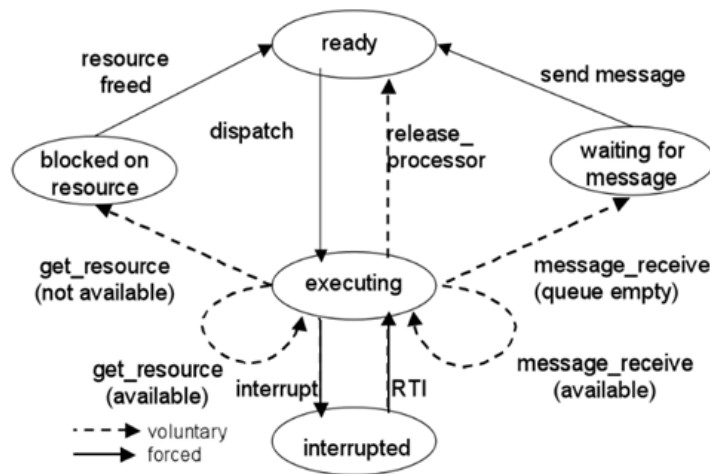


Figure 2: Process State Diagram (Reidemeister, 2010)

Whenever you introduce an abbreviation you should explain what it stands for!

2.0 GLOBAL INFORMATION

Global information about the RTX is defined in this section.

2.1 DATA STRUCTURES

There are a number of different data structures that need to be initialized at the system start up. They are as follows: envelopes, queues, **PCB**

2.1.1 MESSAGE ENVELOPES

In order for the RTX to operate, processes must be able to communicate with each other. Therefore, a message-based Inter-Process Communication (IPC) system must be constructed. The RTX will use the “message envelope” scheme for IPC. A sending process will write the message into a shared memory block (a message envelope), and send a pointer which points to the memory block to the receiving message. When the receiving process is ready, it will follow the pointer and retrieve the message.

To avoid run-time overheads, a fixed number of envelopes (120) will be created during system initialization, and the envelopes will have a fixed size. An envelope will be allocated to a process when a process requests an envelope. When a process releases the message, the envelope will be deallocated and can be used by another process.

Figure 3 shows the common structure and fields of a message envelope.

The `kernel_pointers` are used to by the kernel to place the envelope in the required linked list, such as the free envelope queue or receive message queue. The `sender_process_id` and `destination_process_id` contain the process IDs of the sender and the receiver respectively. They will be updated by the kernel when the envelope is used, and may be checked by the receiving process to validate the sender. The `message_type` field is an optional field that contains the type of message being sent. Message_text area contains the actual message that will be sent. The following pseudo-code provides a general idea of creating the envelope struct:

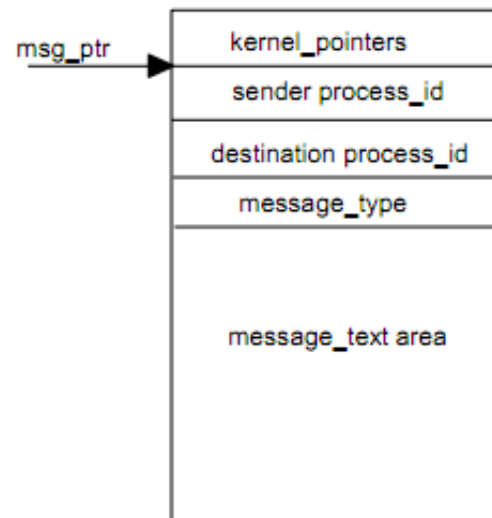


Figure 3: Message Envelope Structure
(Dasiewicz, 2009)

```
struct MSG_type {
    struct MSG_type *p1 // kernel pointers used for placing on linked list
    struct MSG_type *p2 // number of pointers may vary
    int sender_process_id
    int receiving_process_id
    char array msg_type // since C does not have strings
    char array msg_text
}
```

*You do not have to talk about any thing except
}; global information in this section!*

```
typedef struct MSG_type *MSG;    /*MSG is the pointer that point to env
```

Also, since all message envelopes are free and unused at initialization, they will all be placed in a free envelope queue which contains all envelopes not being used. It is created by joining the envelopes together using the kernel pointers. The head of the queue will be connected to a variable that will be used by the kernel to access the queue. The four primitives that will be used for IPC are request_msg_env, release_msg_env, message_send, and message_receive.

For a process to obtain a message envelope, it will call request_msg_env(). After checking the free envelope queue, if an envelope is available the primitive returns a pointer to that envelope. If the queue is empty, the process is blocked and placed in the blocked_on_envelope queue. The pseudocode in Section 3.3 shows the request_msg_env() primitive.

The purpose of the while loop in the pseudocode is to make sure that an envelope is available even if the process is in the ready queue. This is needed because the RTX will make ready the first process on the blocked_on_envelope queue, dequeue it from the queue, and place it in the ready queue whenever an envelope is deallocated. However, the envelope might be taken by a process of higher priority before the original process has the chance to execute, so there must be a check on the availability on envelope even if the process is on the ready_queue.

After a process is done with a message, it needs to free the envelope so it can be used by other processes. This is done using release_msg_env(). The primitive will enqueue the envelope to the free envelope queue, and then check the blocked_on_envelope queue. If the blocked_on_envelope queue is empty, then the primitive is complete. If it is not empty the first process on the queue will be dequeued, made ready, and placed in the ready queue. Pseudocode of the primitive can be seen in Section 3.4.

The RTX is required to have a synchronous message receive system. To avoid any complications that may arise due to multiple messages being sent to the same process simultaneously, each process that can potentially receive a message will have the receive_message queue connected to its PCB. This way, messages that have been sent but not yet received can be stored and handled. The queue will be FIFO based.

When the receive_message primitive is called, it will check the process's receive_message queue. If the queue is not empty, the function will return a pointer to the message. If the queue is empty then the process status will be changed to blocked_on_receive. In order for the process to be made ready again, the process must be sent a message. This will be further discussed in the send_message primitive.

When the send_message primitive is invoked, the sender_proc_id and the destination_proc_id will be written into the appropriate fields in the message envelope. The

envelope will then be delivered to the destination process by enqueueing the envelope in the receive_message queue of the destination process. The status of the destination process is then checked. If it is blocked, then the destination's process is made ready and enqueued in the ready queue.

2.1.2 QUEUES

The RTX will have four ready process queues prioritized from 0 to 3 (0 being the highest priority): a blocked on envelope queue, a free envelope queue, a receive message queue (for every PCB that will receive messages), and a timeout queue. A queue struct containing a head and a tail pointer will be used to point to queues. The queues are explained in more detail below.

Ready Queues

The RTX uses four separate Ready Queues, each containing the PCBs of the processes that are ready to execute. The processes are organized into their respective Ready Queues based on their priority level. Dequeueing of the Ready Queue is handled by the Scheduler, which selects the highest priority process in the Ready Queues when the processor is free or running the NULL process, which is the lowest possible priority and simply repeatedly frees the processor. Processes with equal priority will be handled using the FIFO structure. Figure 4 illustrates the ready queues.

Blocked-on-Envelope Queue

This queue contains the PCBs of processes that are blocked while waiting for a free envelope. The Blocked on Envelope Queue is checked every time an envelope is released, and free envelopes are assigned based on the FIFO structure.

Priority should be maintained as well in this queue

Free Envelope Queue

This queue contains empty envelopes that are ready to be used by processes. The queue will be checked whenever a process requests an envelope, and dequeue an envelope if available. When a process releases an envelope, it is enqueue in the free envelope queue.

Receive Message Queue

Every process that can potentially receive messages has a receive message queue within its PCB. The main reason for this queue is to avoid any potential problems that might arise

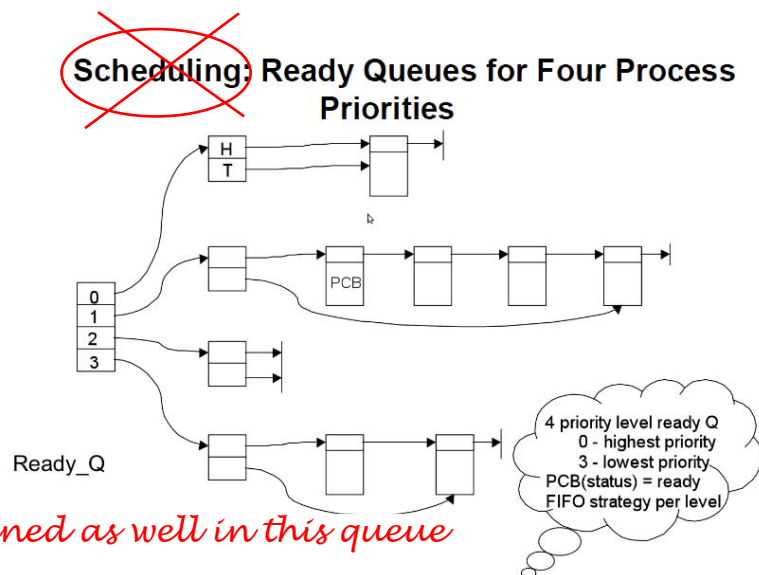


Figure 4: Ready Queue Priority (Reidemeister, 2010)

from sending multiple messages to a process. The Receive Message Queue is also used when a process calls the `receive_message()` function; if its queue is empty, the invoking process will be blocked.

envelops sent by processes

Timeout Queue

This queue contains only those ~~processes~~ that have invoked the timeout i-process and have not yet expired. The queue is organized in order from shortest to longest time remaining, and processes are removed as their wait times expire.

2.1.3 PROCESS CONTROL BLOCKS

A process control block (PCB) is a data structure that contains information of a process that the kernel can use to identify and manipulate the process, such as changing its status. It is the PCB, not the actual process, that is placed in lists and queues. The PCB cannot be accessed by user processes since it is a private data structure maintained by the kernel. Although the fields of a PCB differ based on their application, they generally have the same fields. Those fields are shown in Figure 5.

Every process requires a PCB which is constructed when the process is created, and destroyed when the process is terminated. The PCB must have several kernel pointer fields so that it can be placed in different linked lists. For example, every PCB needs to be placed in the process list, which contains all the processes, and may be added to other lists such as the process-ready list or process-blocked list.

The process state field is used to indicate the current state of a process. Kernel primitives should be able to examine and modify the field accordingly.

The process ID field contains a number that is unique to each process. It is used to by the kernel to identify the process.

The process priority field contains the priority level of the process. It is used by the scheduler to determine when the process should be executed.

The program counter, stack pointer, and CPU registers are used to maintain the process context when it is not executing. The process context is then restored when the process starts to run again.

The memory structure field contains information about memory allocated to the process. The file usage field records any files accessed by the process and their states.

An additional field that can be added to PCBs is the receive message queue pointer. It can potentially receive messages in order to avoid any problems that can be caused by sending multiple messages to a process.

Kernel pointers
Process state
Process id
Process priority
Program counter (PC)
CPU registers
Stack pointer (SP)
Memory structure
File usage
Any additional process related control information

Figure 5: PCBs (Dasiewicz, 2009)

```

struct PCBtype{
    struct PCBtype *p1          //kernel pointers
    struct PCBtype *p2
    char array process_state
    int process_id
    int process_priority
    int PC
    CPU registers
    *stack pointer
    memory structure
    file usage
    *queue receive_msg_Q
};
typedef struct PCBtype *PCB      //PCB is a pointer to a PCB

```

Are you going to use these fields?? If not, why do you include them in the SDD?

You might need another field called process_type: user_process or i_process!

2.1.4 CLOCK STRUCT

It was decided that in order to simplify the operation of all clock processes, a data structure will be written to handle the clock data in the appropriate format. As well, user functions will be written in order to handle the proper incrementing, output format, and reset functions (see Section 4.6 for clock functions).

```

typedef struct {
    int ss;
    int mm;
    int hh;
} clock;

```

2.2 VARIABLES

There are a number of variables that must be defined in order to be used throughout the code and pseudocode for the RTX.

CURRENT_PROCESS - Current_process is a pointer to a PCB that allows the RTX to know which process is currently executing. It always points to the currently executing process's PCB.

PROCESS_ID - This is the ID of a process, which is kept in PCBs.

MSG_TYPE - This shows the message type of a msg_env.

MSG_STATE - This shows the state of a process, contained in its PCB.

PROCESS_PRIORITY - This field in a PCB contains the priority of a process.

2.3 CONSTANTS

There are a number of constants in this RTX.

MESSAGE IDS - These will be hardcoded as numbers starting at zero and incrementing each time by one.

DISPLAY_ACK - This is a message type of acknowledgement used in `send_console_chars`.

CONSOLE_INPUT - This is a message type of an envelope used by `get_console_chars`.

READY - This status illustrates that a process has a status of ready.

BLOCKED_ON_ENV - This status illustrates that a process has a status of blocked while waiting for an envelope.

BLOCKED_ON_RECEIVE - This status illustrates that a process has a status of blocked while waiting for a message envelope.

EXECUTING - This status illustrates that a process has a status of executing.

INTERRUPTED!

2.4 MEMORY MAP

The system memory is used mainly for two tasks: context switching processes, and shared memory for the CRT and Keyboard forks.

Context switching is handled by blocking off a stack of memory for the Program Counter, CPU Registers, and Stack Pointer of the current process. When `context_switch()` is called, the current process copies the three data sets listed above into the stack, the new process is moved into the CPU, and the Program Counter, CPU Registers, and Stack Pointer are moved back into the PCB of the switched process. This allows blocked or interrupted processes to resume functioning at the same place where the block or interrupt was called.

The shared memory for the CRT and Keyboard forks are two separate blocks of memory used solely to transfer data between the RTX and the CRT or Keyboard. These blocks of memory act as envelopes for the CRT and Keyboard processes, allowing an easy transfer between the RTX and external processes.

Data structure for the tracing buffer is missing!!

3.0 PRIMITIVES

The RTX will handle primitives such as the following.

3.1 SEND_MESSAGE

This process sends a message to the receiving process by placing the PID of the receiving and sending processes into a message envelope, along with whatever data is required to be sent. This process is not a blocking process, as it is assumed that if `send_message()` is invoked, an envelope has already been successfully received (and thus, `request_msg_env()` has not blocked), and the message can be sent without issue. As well, `send_message()` may return a blocked `receive_message()` process to the ready state.

```
int send_message(target_pid, env) {  
    atomic(on);  
    set sender_procid, destination_procid fields in env  
    convert target_pid to process obj/PCB ref  
    enqueue env onto the msg_queue of target_proc  
  
    if(target_proc.state is blocked_on_receive) {  
        set target_proc state to ready  
        rpq_enqueue(target_proc);  
        atomic(off);  
    }  
}
```

Send transactions should be recorded in the trace-buffer!

(Dasiewicz, 2009)

3.2 RECEIVE_MESSAGE

This process checks to see if the invoking process has a message in its message queue. If the message queue is empty, the process becomes blocked. It is important to note, however, that the process does not enter the blocked queue as it contains its own code for becoming ready once it has received a message. If the message queue is not empty, a pointer to the message envelope is returned and the message is delivered.

```
MsgEnv *receive_message() {  
    atomic(on);  
    if(current_process->priority = -1)  
        return NULL;  
    else  
    {  
        while(current_process's msg_queue is empty) {  
            set state of current_process to blocked_on_receive,  
            process_switch();  
            return here when this process executes again  
        }  
    }
```

WHAT do you mean -1

according to the pseudo code the i-proc can be blocked !

*Receive transactions should be
recorded in the trace-buffer!*

```
}  
Dequeue envelope from the process' message queue  
atomic(off);  
return env;  
}  
(Dasiewicz, 2009)
```

3.3 REQUEST_MSG_ENV

This process returns a pointer to a standard sized envelope to the invoking process. If there is not sufficient memory for an envelope to be returned, the invoking process is blocked and placed in the blocked queue until adequate resources are available.

```
MsgEnv * request_msg_env() {  
    atomic(on);  
    if(current_process->priority = -1) WHAT do you mean -1  
        return NULL;  
    else  
    {  
        while(free_env_queue is empty) {  
            put process object/PCB on blocked_env_queue  
            set process state to blocked_on_env  
  
            process_switch(); according to the pseudo code the i-proc  
can be blocked !  
  
            restart here when process executes eventually  
        }  
        env -> reference to de-queued envelope  
        atomic(off)  
        return env;  
    }  
}  
(Reidemeister, 2010)
```

3.4 RELEASE_MSG_ENV

This process returns an envelope that is no longer needed. If there are blocked processes waiting for envelopes, this will allow the blocked process with the highest priority to become ready (although the process itself will not unblock or deliver the empty envelope, it will only free a resource to the CPU which will allow the scheduler to deliver the envelope to the highest priority blocked process).

```
int release_msg_env (MsgEnv * msg) {  
    atomic(on);  
    put env onto free_env_queue  
    if(blocked_env_queue not empty) {  
        dequeue one of the blocked processes  
        set its state to ready and enqueue it on ready process queue  
    }  
}
```

```

        atomic(off);
    }
    (Reidemeister, 2010)

```

3.5 RELEASE_PROCESSOR

This process may be invoked when a process has completed or voluntarily releases the CPU. The current process stack is copied from the CPU into the PCB stack of the current process, the invoking process status is set to ready, moved into the ready queue based on its priority, and the processor is set to ready to receive the next highest priority process from the ready queue. Since it is assumed that all processes are non-malicious, it is safe to assume that any process that calls `release_processor()` is doing so with the correct intent, and thus `release_processor()` is not a blocking process.

```

int release_processor() {
    atomic(on);
    put processor stack into current_process PCB
    set current_process state to ready
    rpq_enqueue(target_proc);
    if(rpq != NULL)
    {
        get highest priority pid from rpq
        execute process(high_priority_pid) How will you execute it??
        change process state to executing
    }
    else
        Execute NULL process
    atomic(off);
}

```

3.6 REQUEST_PROCESS_STATUS

This process returns the status and priority of all current processes in the format of a two-dimensional array, ordered by process ID. Each row of the n-row array contains three columns: process ID, status, and priority, with the exception of the first row, which contains an integer value of the number of rows in the array. This process requires an envelope in which to return this array, and for the same reasons as listed above in `release_processor()`, is not a blocking process.

```

int request_process_status(MsgEnv *msg) {
    atomic(on);
    Initialize array[total # of process types (see section 4.0) + 1][3]
    set Array[0][0, 1, 2] = total # of process types ???? not clear!
    For(i=1; i < number of processes) (indexing begins at 1, because 0
    is the integer) {
        Set Array[i][0] = target_proc(process_pid)
        Set Array[i][1] = target_proc(process_status)
        Set Array[i][2] = target_proc(process_priority)
    }
}

```

```

    }
    Place array in env
    atomic(off);
}

```

3.7 TERMINATE

This process requires that the RTX stop (or terminate) its execution. Rather than have a dead-stop of the executable, this function stops all processes in an orderly way, allowing data to be saved and then terminated. However, since this RTX is being simulated in a UNIX environment, the terminate() function will kill the RTX, its children and return control and all resources back to the UNIX system.

```

int terminate() {
for(all processes)
{
    if (process is one which may own an envelope)
    {
        if(process owns a message)
            release_msg_env();
    }
    set all PCB data to NULL
    Remove all pointers
}
release_processor();
return 0;
}

```

Atomicity !??

A lot of details are missing here! you should have highlighted the termination of the help processes, the shared memory in addition to releasing any dynamic memory has been reserved in the initialization.

3.8 CHANGE_PRIORITY

This process takes a target process and changes the priority to a new, currently specified priority. This change happens immediately. Since this process could be in a priority queue, it is possible that the change_priority() function may change the order of the queue.

```

int change_priority(new_priority, target_process_id) {
    if (new_priority is a valid priority) {
        target_proc_curr_priority = new_priority;
    }
    else
        print error message

    check if proc_curr_id is in correct place in priority
    if(not right place)
        move to correct place
}
(Dasiewicz, 2009)

```

Atomicity !??

What do you mean valid priority!

What are the places?

3.9 SEND_CONSOLE_CHARS

This function sends a message envelope containing a character string to the CRT i-process. This string is in typical C string format, terminated by a null character. The invoking process will not block. After this string has been output to the screen, the envelope is sent back to the invoking process as confirmation that the string has been displayed correctly. The returned message type will be “display_ack”.

```
int send_console_chars(MsgEnv * message_envelope) {  
    set sender_procid, destination( to CRT_iprocess) fields in env  
    write string to msg_env The string is already written!!  
    int CRT = CRT_iprocess(msg_env)  
  
    if(CRT = 1, ie. successful) {  
        set message type to 'display_ack'  
        return msg_env to sender_procid  
    }  
    Else return error  
}
```

This primitive is not responsible for returning the msg to the invoking process

There should not be any error returned here. The Ack should be sent in an envelope. it is not a return flag!!!

3.10 GET_CONSOLE_CHARS

When a process requests input from the keyboard, get_console_chars() sends a pointer to a message envelope to the Keyboard i-process, and continues executing. It is assumed that a prompt for the user is provided by the invoking process, and thus no user prompts are displayed by this primitive. When a full string ended by a return carriage (or null character) is received in the envelope, it is sent back to the invoking process. If the receiver buffer in the UART is full or the message envelope fails to send to the invoking process for any reason, an error message will be returned to the process inside of the envelope, and the process will continue executing if possible. The message type of this envelope will be “console_input”.

```
int get_console_chars(MsgEnv * message_envelope) {  
    set sender_procid, destination (goes to KB_iprocess) fields in env  
    give pointer to env to Keyboard i-process  
    KBD -> send string to env  
    If(sending to env = done) {  
        Set message type to 'console_input'  
        Return env to invoking process  
    }  
    Else return error  
}
```

Where is the atomicity !??

Does not make sense ! What you need here is just to set the msg type and send it to the i-process!

3.11 GET_TRACE_BUFFERS

This primitive will return the trace of the sixteen most recent successful invocations of the send and receive primitives. Each entry will have destination and sender process IDs, message types, and time stamps of the RTX clock (for both send and receive). This function takes a snapshot of the trace buffers at this time.

This process provides a pointer to a message envelope. It then copies the past sixteen send and receive transactions to the envelope, in that order. The process invoking this primitive will not block.

```
int get_trace_buffers(MsgEnv * message_envelope) {  
    atomic(on);  
    take pointer to msg_env  
    for (trace = 0; trace < 16; trace++) {  
        write send_pid, send_message_type, send_time_stamp to env  
        write rec_pid, rec_message_type, rec_time_stamp to env  
    }  
    Return envelope to invoking process  
    atomic(off);  
}
```

You just need to read the trace-buffer and write its content into the message!

4.0 PROCESSES

4.1 SYSTEM PROCESSES

The system processes are as follows.

RPQ_ENQ

This process will take a PCB pointer and place it at the end of the appropriate ready process queue.

This is not a process! Does it have a PCB??

```
int rpq_enq (PCB *x){
    if(x->proc_priority == 0){
        priority_Q_0 -> tail = x;
        x -> p1 = NULL;
        return 0;
    }
    if(x->proc_priority == 1)
        enqueue PCB to priority_Q_1
    if(x->proc_priority == 2)
        enqueue PCB to priority_Q_2
    enqueue PCB to priority_Q_3
}
```

RPQ_DEQ

This function dequeues a process from the ready process queue. It will try to dequeue from the ready process queue with the highest priority. If that queue is empty, then it will attempt to dequeue from the next highest ready process queue.

```
PCB *rpq_deq (){
    PCB *p;
    if(priority_Q_0 is not empty){
        p = priority_Q_0 -> head;
        priority_Q_0 -> = priority_Q_0 -> head -> p1; //head pointer points to next
    }
    return p;
}
if(priority_Q_1 is not empty)
    dequeue priority_Q_1
if(priority_Q_2 is not empty)
    dequeue priority_Q_2
dequeue priority_Q_3
}
```

BLOCKED_Q_ENQ

This process enqueues a PCB to the blocked process queue. The pseudocode for enqueueing is similar to enqueueing a PCB into a ready process queue.

Those are not processes!

BLOCKED_Q_DEQ

This process dequeues a PCB from the blocked process queue. The pseudocode for dequeuing is similar to dequeuing a PCB from a ready process queue.

ENV_ENQ

This process enqueues a message envelope into a queue. The pseudocode is similar to enqueueing a PCB with the difference of having an extra parameter that tells the RTX to which queue should the envelope be enqueued.

ENV_DEQ

This process dequeues a message envelope from a queue. The pseudocode is similar to dequeuing a PCB with the difference of having an extra parameter that tells the RTX from which queue should the envelope be dequeued.

4.2 CONSOLE COMMAND INTERPRETER

The CCI is a high priority user process. It receives the keyboard commands given to it by the `get_console_chars` function and converts them into commands usable by the RTX. The CCI then performs the requested action, and outputs the required output to the CRT using `send_console_chars`. Any invalid commands will prompt an output displaying an error message. On top of this, the CCI will display the 24 hour clock.

CCI: s <cr>

This command sends a message envelope to the first test process, Process A. This envelope will not be returned to the interpreter.

```
request_msg_env();  
request_pid of process A  
send_message();
```

CCI: ps <cr>

This command will display the process status of all processes in an array. It will show the process ID, priority, and status ordered according to process ID.

```
env = request_msg_env( );  
env2 = request_msg_env( )  
request_process_status(env);  
place headings of process status (e.g. pid, priority, status) into env2  
print env2 to CRT  
print env to CRT  
release_msg_env(env)  
release_msg_env(env2)
```

The pseudocode of the CCI as a process is missing? How CCI interact with the other processes? What will happen to the received messages! where do you receive the output acknowledgments and how do you deal with them? You mentioned that CCI will maintain the Wall clock. How is it implemented?...

CCI: c hh:mm:ss <cr>

This command will set the clock to any valid specified 24-hour time. It should update the clock immediately, but will not affect the internal clock time stamp. It will print the new time to console and will continue to increment the new time with each clock pulse.

Get_console_chars to read the hours, minutes, and seconds
clock_set(clock, hh, mm, ss)

But the clock pulse is much faster than a second!!

CCI: cd <cr>

This command allows for the clock function to be displayed on the console.

loop until receive off command
check for clock_increment(clock)
clock_out(clock)

CCI: ct <cr>

This command will halt the display of the clock on the console.

request_msg_env()
send clock off command

CCI: b <cr>

This command will display to screen the past sixteen instances of the send and receive trace buffers. It will be invoking the function get_trace_buffers.

Message_trace_display();

CCI: t <cr>

This command will terminate the execution of the overall RTX. It will call the terminate function.

Terminate();

CCI: n new_priority, process_id <cr>

This command will take in a process ID and a priority. It will change the priority of the target process.

Change_priority(new_priority, target_process_id);

CCI: knock <cr>

This command accesses one of the user processes “Knock-Knock Joke”.

knock_knock();

CCI: timewarp <cr>

This command will run the user process “Time Warp”.

```
Time_Warp();
```

CCI: pong <cr>

This command will run the user process “Pong”.

```
Pong();
```

4.3 SIMULATED UART PROCESSES

Because our operating system is not actually an embedded microsystem that uses a UART for its input and output, we will simulate the UART using input (Keyboard) and output (CRT) Linux processes to communicate with the RTX.

4.3.1 KEYBOARD UART PROCESS

The simulated Keyboard UART process is invoked by the Keyboard i-process when keyboard input is required from the user. The Keyboard UART process will take the keyboard input via Linux and put it in the shared buffer memory, then signal the Keyboard i-process with a UNIX signal that the input is ready to be taken.

```
void kb_uart() {  
    get input from console  
    put input in memory buffer  
    signal Keyboard i-process  
}
```

This is very abstract! where do you pass the parent id and file id. How you create the mapped memory? where do you setup the signals? More details are needed!

4.3.2 CRT UART PROCESS

The simulated CRT UART process is notified by the CRT i-process when something is ready to be output to the screen. The CRT UART process will then take the output out of the shared buffer memory and display it to the screen via Linux, then signal the CRT i-process with a UNIX signal if the output was displayed correctly or not.

```
void crt_uart() {  
    get signal from CRT i-process  
    retrieve output from memory buffer  
    print output to screen  
    signal CRT i-process success/not success  
}
```

The same like above!??

4.4 NULL PROCESSES

The purpose of null processes is to ensure that the CPU is executing a process at all times. Null processes have the lowest possible priority so that they will only execute when there are no other processes in the ready queue. They also tend to be redundant operations such as calculating pi or repeatedly releasing the processor. For this project, our null process will be the latter example.

Note: the function `change_priority()` will not be able to change the priority of the null process.

Example null process:

```
void null_process() {
    while(1) {
        release_processor();
    }
}
```

(Reidemeister, 2010)

4.5 USER PROCESSES

User processes serve many functions. Some processes deal with communications between programs, such as shared memory or message passing. Others concentrate on resource allocation between programs, and error detection within CPU, memory, storage devices, networks, and so on. Lastly, more still focus on protection, which serves to ensure users don't interfere with other users' data, and the prevention of unauthorized access to devices.

In this RTX, the user processes serve a more superficial function. The user processes implemented in our RTX exist only to demonstrate the functionality of our system. The three user processes implemented in our RTX are “Knock-Knock Joke”, “Time Warp”, and “Pong”.

4.5.1 KNOCK-KNOCK JOKE

The Knock-Knock Joke process will output the message “Knock-knock” to the CRT. It will then wait until the user inputs “Who's there?” and output a joke answer. If the user inputs anything other than “Who's there?”, the process will return a different message and end the process.

```
void Knock_knock ( )
{
    Env = Request_Msg_Env( )
    Put “Knock Knock” char array into env
    Send_Console_Chars( Env )
    Release_Msg_Env( )
    Rec_Env = Request_Msg_Env( )
    Get_Console_Chars( Rec_Env )
    if(Rec_Env contains “Who's there?”) {
        Release_Msg_Env( )
        Rec_Env = Request_Msg_Env( )
        Put Joke Answer char array into env
        Send_Console_Chars( Env )
    }
    Else {
```

```

        Release_Msg_Env( )
        Rec_Env = Request_Msg_Env( )
        Put Ruined Joke char array into env
        Send_Console_Chars( Env )
    }
    Release_Msg_Env( )
    Return 0
}

```

4.5.2 TIME WARP

The Time Warp process displays the system clock artificially sped up to 100 times its original speed. After an hour has passed at the increased time rate, the system will output the message: “Want to do the Time Warp again? Then run the process again!”. The system clock maintains its original functionality, however, and only the display is sped up.

```

void Time_Warp( clock* clock )
{
    for(60 iterations) {
        receive clock pulse
        for(10 iterations) {
            clock_increment(clock, SYSTEM)
            clock_out(clock)
        }
    }
    Env = Request_Msg_Env( )
    Put “Time Warp” char array into env
    Send_Console_Chars( Env )
    Release_Msg_Env( )
    Return 0;
}

```

4.5.3 PONG

The pong process is designed to push the CRT process, the envelope queue, and the send_console_chars() function to their limits. This process will display a series of messages to simulate a short game of pong.

```

void Pong( ){
    for(i = 20 iterations)
    {
        Env = Request_Msg_Env( )
        if(i = 20th iteration)
            Put “/” into env at char location 1
        else
            Put “ “ into env at char location 1
    }
}

```

```

        For(j = 20 iterations) {
            if(j = 20/i)
                Put "." into env at char location j
            else
                put " " into env at char location j
        }
        if(i = 1st iteration)
            Put "/" into env at char location 21
        else
            Put " " into env at char location 21
        Send_Console_Chars( Env )
        Release_Msg_Env( )
    }
}

```

4.6 CLOCK PRIMITIVES

The clock primitives are functions that are called whenever the wall clock or system clock (except in the case of `clock_set`) must be modified or output in any way. Since the clock requires a specified format and has specific rules for valid and invalid entries, it is simpler to write functions that handle the operations without user input. These functions will handle operations such as `clock_increment`, `clock_set`, and `clock_out`.

CLOCK_INCREMENT

This function will increment the clock by one second, and check to ensure that the clock is still in the proper format for its specific type. This function will require a pointer to a clock struct, as well as a boolean operator telling the function whether it should format the clock as a wall or system clock. If the function receives a 0, it is a system clock and is formatted with the hour field as an unbounded integer. If the function receives a 1, it is a wall clock and is formatted with the hour field as an integer between 0 and 23. This is not a user function, but rather a helper function that ensures the clock increments within its specified format.

```

void clock_increment(clock* clock, bool system_or_wall) {
    if (clock.ss < 60)
        system_clock.ss++
    else {
        clock.ss = 0

        if (clock.mm < 60)
            clock.mm++
        else {
            clock.mm = 0
            if (system_or_wall = 1) {

```



```

        if(clock.hh < 24 || clock.hh > 0)
            clock.hh++
        else
            clock.hh = 0
    }
    else
        clock.hh++
}
}
if(system_clock.ss < 0 || system_clock.ss > 60 || system_clock.mm < 0 ||
    system_clock.mm > 60 || system_clock.hh < 0)
    output error
}

```

CLOCK_SET

This function will allow the user to set the wall clock to a 24 hour time, and return an error if the input time is invalid. Although technically this function will allow the system clock to be set as well, the CCI will ensure that the user call to set the clock will only modify the wall clock.

```

void clock_set(clock* clock, int hours, int minutes, int seconds) {
    if(seconds < 60 && seconds >= 0) {
        if(minutes < 60 && minutes >= 0){
            if(hours < 24 && hours >= 0){
                clock.ss = seconds
                clock.mm = minutes
                clock.hh = hours
                return //ends function before hitting error message
            }
        }
    }
    output error message //hit if any if statements are invalid
}

```

CLOCK_OUT

The clock_out() function will receive a pointer to a clock and output the clock to the CRT. Similar to clock_set, this function could technically be used to output the system clock, but the CCI will ensure that in the final deliverable only the wall clock is printed. This function will alternatively be used to output the system clock, but this will only be for debugging purposes.

```

void clock_out(clock* clock){
    put clock.ss in env
    put ":" in env
}

```

```

    put clock.mm in env
    put ":" in env
    put clock.hh in env
    error check
    send env to Send_Console_Chars
}

```

4.7 TEST PROCESSES A, B, C

There will be three processes to test the functionality of the RTX. The functionality is as follows. Pseudocode for these processes can be found in the mte241Proj-c-v4.53.pdf file.

4.7.1 PROCESS A

Process A receives an envelope in response to an input from keyboard. It continually requests and sends message envelopes to Process B. It increments the count each time it requests a new envelope and passes it on. Essentially, this process is a counter. Process A begins the implementation as blocked, waiting for an envelope from the CCI.

4.7.2 PROCESS B

Process B receives messages from Process A and sends them on to Process C. This process is essentially a message relay.

4.7.3 PROCESS C

After initialization, Process C continually checks for messages in its message queue. If it receives one and the first element of the data is divisible by 20, it displays "Process C" on the screen, then delays.

4.8 MESSAGE TRACE DISPLAY

The message trace display takes the result of the primitive `get_trace_buffers()` and sends it to `send_console_chars`.

```

Get return -> get_trace_buffers();
send trace buffers to Send_Console_Chars via an env

```

5.0 INTERRUPT HANDLING

Because the RTX will be operating very quickly, interrupts must be handled quickly and efficiently. First, the RTX in general must be quick to recognise that a process called an interrupt. Then, it must be swift in determining which process invoked said interrupt. Figure 6 shows an average response interval.

In general, interrupts cause a change of state. Often they will move a blocked process to the ready state. This could occur when a process was waiting on input from the keyboard or screen. When the message envelope has the needed information, an interrupt could be sent to the process to send it back to the ready queue.

Since the RTX will be run in a simulated system, for the purpose of this project the interrupts will not come from external hardware. Instead, interrupts will be simulated through signals from the UNIX environment.

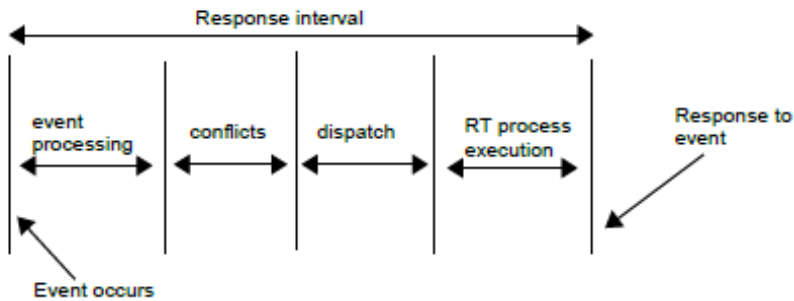


Figure 6: Interrupt Handling Response Time (Dasiewicz, 2009)

The process should execute then cause an interrupt. Three process should have an i-process that deals with interrupts - the timer (clock), keyboard (get_console_chars), and CRT (send_console_chars). Because this system is non-preemptive, the interrupt handler will always return to the same user process.

5.1 I-PROCESSES (INTERRUPT HANDLER)

I-processes handle and receive the UNIX signals (interrupts), and are not allowed to block. Because this system will operate using i-processes to deal with interrupts, all other kernel functions must be modified to ensure that they will not block if an i-process invokes them. This interrupt handling process itself is an i-process.

When a signal is received, the i-process should acquire the processor immediately through an initial exception handling sequence. It should save the current process's context in its own PCB then will select the appropriate i-process to deal with the interrupt, and assign it to run.

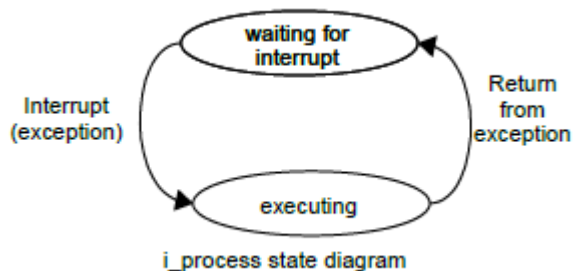


Figure 7: I-process State Diagram (Dasiewicz, 2009)

After the i-process has finished it should return to the handling sequence, which should allow the interrupted process to continue running. The i-process state diagram is shown in Figure 7. Note that because our system is non-preemptive, interrupt handling will follow the process shown in Figure 8, except the handler will always return to the same user process that was executing at the time of the interrupt.

When an i-process is not running, it is always ready to run. However, it is not placed on any ready queues. Also, each PCB will have a field for `atomic_count()`. If the i-process is called while atomic is on, the field will be set to 1 when the i-process is initialized.

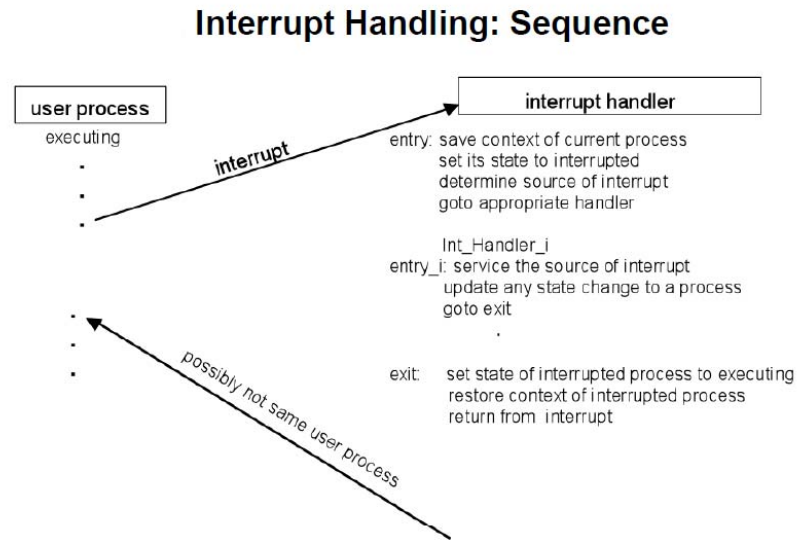


Figure 8: Interrupt Handling Sequence (Reidemeister, 2010)

exception_handler:
begin

// code here to save current process context into its context save area as defined
// by its PCB

save_PCB = current_process
select interrupt source

A: current_process <- i_proc_A_pcb
//restore i_proc_A context
//invoke i_proc_A handler
break

Z: current process i_proc_Z_pcb
//restore i_proc_Z context
//invoke i_proc_Z handler
break
end select

//code to save context of interrupt handler (i_process)

current_process = save_PCB

//code to restore current_process context including "atomic_count" field effect
// (if used). Perform a return from exception sequence. This restarts

// the original process before i_handler

(Reidemeister, 2010)

5.1.1 TIMER I-PROCESS (SYSTEM CLOCK)

The RTX will use a UNIX signal as a timer. With each pulse, the clock will keep a count at 0.1s intervals. Using these pulses, the clock will be able to assist timing services. Every tenth pulse will increment both the system and wall clock, using the `clock_increment()` function (see Section 4.6). The user will be able to request the time from either the wall or system clock, and the clock will be able to display it.

```
loop(forever) {  
    int dummy_pulse_counter  
    get clock pulse (where pulse is at 0.1s intervals)  
    increment dummy_pulse_counter  
    if dummy_pulse_counter = 10  
    {  
        clock_increment(system_clock, 0)  
        clock_increment(wall_clock, 1)  
        dummy_pulse_counter = 0  
    }  
}
```

5.1.2 KEYBOARD I-PROCESS

The Keyboard i-process receives a pointer to an envelope from `get_console_chars`. It then sends a UNIX signal to the Keyboard UART process to get the keyboard input and waits for the Keyboard UART to place the keyboard input in the memory buffer. When the Keyboard UART signals back that there is something in the buffer, the Keyboard i-process takes the input out of the shared buffer, puts it in an `console_input` envelope, and sends it back to `get_console_chars`.

```
void kb_iprocess(MsgEnv * message_envelope) {  
    signal Keyboard UART to get input  
    get signal that input is in buffer  
    retrieve input from buffer  
    put input in the envelope  
    return (env)  
}
```

5.1.3 CRT I-PROCESS

The CRT i-process receives a pointer to an envelope from `send_console_chars`. It then puts the contents of the envelope (the output) into the shared memory buffer and sends a UNIX signal to the CRT UART process notifying that there is output. The CRT UART will take the output from the buffer, display it, and send a signal back upon successful display. The CRT i-process returns a 1 if successful to `send_console_chars`.

```
void CRT_iprocess(MsgEnv * message_envelope) {
```

```

        put envelope contents in buffer
        send signal to CRT UART that output is in buffer
        receive confirmation signal from CRT UART
        if (confirmation = true)
            {
                return (1)
            }
        return (0)
    }
}

```

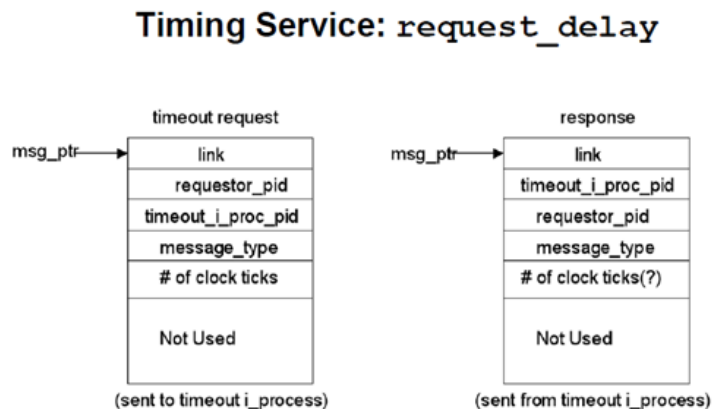
5.2 TIMING SERVICES

Timing services are among the most fundamental and critical processes in the RTX. These processes allow the functions in the RTX to perform actions based off of the system time, such as `request_delay()` and `timeout_i_process()`. As well, included under timing services are the functions utilized by the system clock in order to perform basic clock operations such as increment, reset, and output to screen.

All of these timing requests could either be counted using the relative time since the RTX has initialized (measured in clock ticks), or in the absolute time which would be external to the computer. We will use relative time as established in Initialization Stage 4 (see Section 7.4).

5.2.1 REQUEST_DELAY

For each process calling the `request_delay()` function, it is delayed for a certain number of 100 ms intervals. However, the process calling this function will not block; it will simply sleep for the length of the specified time delay. After the timer has finished, the invoking process will receive a message of type “wakeup_code” to execute the process again. This is one process that voluntarily gives up the CPU after a certain amount of time passes, which is sometimes referred to as “going to sleep”.



```

int request_delay(time_delay,
wakeup_code, MsgEnv *
message_envelope) {
    copy processor stack into process_PCB
    release_processor();
    set sender_procid, destination_procid (timeout_i_process_pid), and message
(wakeup_code) fields in message_envelope

```

Figure 9: Timing Services Delay (Reidemeister, 2010)

```

    send_message(timeout_i_process_pid, message_envelope)
    message_envelope = receive_message( )
    if(message_envelope->message = wakeup_code)
        end request_delay, place in ready queue
    else print error message (since all pids are known, the code should never get here.
This is for debug purposes only)
}

```

5.2.2 TIMEOUT_I_PROCESS

When the timeout_i_process is invoked, the invoking process requests to be notified after a specified amount of clock ticks have passed. This is somewhat like setting an alarm clock to retrieve something while still awake. The process will continue to execute while it is waiting for its notification. This is an i-process, and therefore it cannot block.

It is assumed that all processes know the process ID of the timeout_i_process() which is needed in order to send a message envelope to the process requesting a timing notification.

```

timeout_i_process:
{
    env = receive(); //to get pending requests
    while(env is not null)
    {
        //code to insert the envelope into the timeout_list
        env = receive(); //see if any more requests
    }

    if (timeout_list is not empty)
    {
        while(head_of_timeout_list.expiry_time <= cur_time)
        {
            env = timeout_list.dequeue();
            target_pid = env.source_pid;
            env.source_pid = timeout_i_process_pid;
            send(target_pid, env); //return envelope
        }
    }
}

```

(Reidemeister, 2010)

A variation of this timeout_i_process could be to ask for a repetitive timeout. This would be a request to the kernel to ask to be informed every n microseconds until the process requests this notification be terminated.

In order to access timing services, each process would send a request in a message envelope which would then be put on an ordered queue. This queue should be arranged such that the request with the shortest expiry time is first. For each clock tick, the timeout process invokes the `receive_message()` primitive to scan for any new requests. If a new envelope is received, it will be added to the sorted queue according to the length of time before notification, and the expiry time of all processes would be decremented. After each tick, the queue is scanned again to see if the count of any process is zero. When a count reaches zero, the process's message envelope will be returned back to the process while the other processes continue counting down.

Timing Service: Design

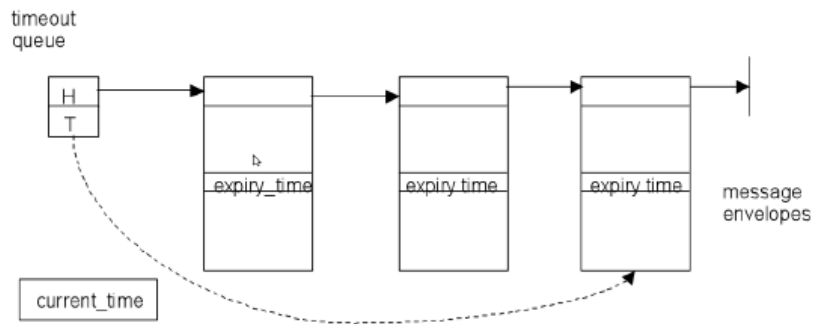


Figure 10: Timing Services Design (Reidemeister, 2010)

6.0 KERNEL ENTRY

Kernel functions can be found in the following sections. The kernel primitives are identical to the user primitives found in Section 3. The only difference between the kernel and user primitives is that the atomic function defined in Section 6.3 is used in order to ensure kernel functions are not interrupted.

6.1 USER PROCESS / KERNEL INTERFACE

User processes run with a limited scope. They cannot see kernel functions (and therefore cannot access them) and they have a shortened number of CPU instructions that they can access. It is also restricted in how much address space it is able to access. These processes run in what is called ‘user mode’. The kernel functions run in ‘supervisor mode’, and have complete access to all CPU instructions and address space. The kernel ensures that when a user process is scheduled, the process must run in user mode.

The user processes request services from the kernel by calling the user visible API. This process interface defines all services provided to the user processes by the kernel. The SWI handler allows the user API services to access some kernel functions. This was done to avoid issues if the API is in user mode yet needs to run a kernel function. This is illustrated in Figure 11.

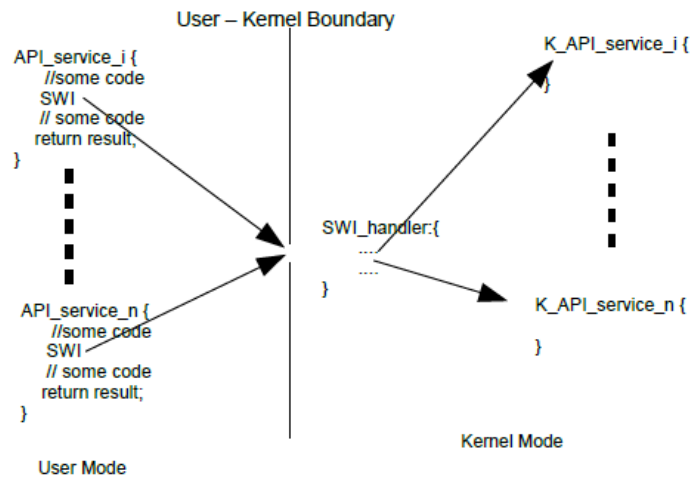


Figure 11: User Process / Kernel Interface (Dasiewicz, 2009)

6.2 PROCESS_SWITCH

The process_switch function takes pointers to the current and new process PCBs and switches between processes. Process switches can occur in two places; the first being in kernel functions, and the second in an interrupt handler after returning from a kernel primitive. It can be requested by a kernel function or by an interrupt.

Process switch gets a pointer to the PCB of the current process from the kernel, then a pointer to the PCB of the next highest priority process in the ready queue. It switches the status of the current and next process to ready (or blocked) and executing respectively, then calls the function context_switch().

```
process_switch( ) {
    PCB_ptr *next_pcb, *old_pcb; //local variable to hold ptr to next pcb
    next_pcb = rpq_dequeue( ); //get ptr to highest priority ready process
    next_pcb->state = running; //set state field of 'next' to running
    old_pcb = current_process;
```

```

        current_process = next_pcb; //make the next_pcb the current process
        context_switch( old_pcb->jmp_buffer, next_pcb->jmp_buffer );
    }

```

Context switch stops the currently running process and starts the next process. It saves the context of the current process into the PCB (using the call of `setjmp`), and restores the context of the next process in the ready queue using information saved in its PCB (using the call of `longjmp`). It restarts the next process and returns back to the invoking function. Execution of the invoking function will resume on the line after the line with the function `process_switch()`.

```

// context_switch() - performs the context switch between two user processes
// Once we call longjmp(), we resume executing the "next" process
// at the point where its context was last saved with a setjmp(). */

```

```

void context_switch(jmp_buf * previous, jmp_buf * next) {
    return_code = setjmp(*previous);
    if (return_code == 0) {
        longjmp(*next, 1);
    }
}

```

Process switching often takes a lot of CPU time, and therefore must be as efficient as possible to minimize this time.

6.3 ATOMIC FUNCTION

The `atomic(on/off)` function masks or unmasks UNIX signals to a process. If a process is running with atomic on, as most kernel functions do, this process will not receive interrupts. It will run until completion, then, if atomic was turned off at the end of completion, will deal with any interrupts accumulated during the time atomic was on. The function could be coded as follows.

```

atomic(bool on) {
    static sigset_t oldmask;
    sigset_t newmask;
    if (on) {
        sigemptyset(&newmask);
        sigaddset(&newmask, SIGALRM); //the alarm signal
        sigaddset(&newmask, SIGINT); // the CNTRL-C
        sigaddset(&newmask, SIGUSR1); // the CRT signal
        sigaddset(&newmask, SIGUSR2); // the KB signal
        sigprocmask(SIG_BLOCK, &newmask, &oldmask);
    }
    else {
        //unblock the signals
        sigprocmask(SIG_SETMASK, &oldmask, NULL);
    }
}

```

7.0 INITIALIZATION

Initialization of the RTX will be handled in four stages. The first stage will initialize the queues used by the RTX, the second stage will initialize the user processes as well as distributing all user processes into their appropriate queue, the third stage will initialize the i-processes as well as the CRT and Keyboard helper processes and their shared memory blocks, and the fourth stage will transfer CPU control to the highest priority ready queue and begin RTX operation. The following sections will further detail the design functionality as well as provide pseudocoded examples of initialization functions.

7.1 INITIALIZATION STAGE 1: QUEUE INITIALIZATION

The first stage of initialization will initialize the ready, blocked, and envelope queues. Since the maximum number of processes is known, the initialization will be simplified by creating the queues at the maximum size of each queue. The ready queues and blocked queues will all be initialized in this stage as empty, while the envelope queue will be initialized containing the maximum number of envelopes in the RTX (in the case of our design, 120 envelopes will be used in total).

In the queue initialization function, a create queue function will be called that returns a pointer to the newly created queue. Pseudocode showing how a queue is created is shown below.

```
PCBQ *create_Q () {
```

```
    PCBQ *p;  
    p->head=NULL;  
    p->tail=NULL;  
    return p;  
}
```

This pseudocode also works for envelopes simply by replacing PCB with envelope.

```
void init_queues() {  
    ready_q_priority0 = PCBQ *create_Q()  
    ready_q_priority1 = PCBQ *create_Q()  
    ready_q_priority2 = PCBQ *create_Q()  
    ready_q_priority3 = PCBQ *create_Q()  
  
    blocked_on_receive = PCB *create_Q()  
    blocked_on_envelope = envelopeQ *create_Q()  
    envelope_q = envelopeQ *create_Q()  
    //error checking code goes here  
}
```

7.2 INITIALIZATION STAGE 2: USER PROCESSES

The second stage will initialize the user processes using an Initialization Table-driven approach. The user processes will each be assigned their ID, priority, initial PCB memory blocks, and initial contents of their respective PCBs. This Initialization Table will be in the format of an array of size m , where m is the number of user processes, and will contain the necessary data for the process ID, process priority, and initial PCB for each user process. As well, the second stage will distribute all user processes into their appropriate ready or blocked queues, based on their initial status and priority as set by the Initialization Table.

```
void init_processes ( ) {
    for ( i < m number of processes) {
        allocate new_pcb = (struct PCB *) malloc (sizeof (struct PCB))
        initialize new_pcb->priority = itable[i]->priority
        initialize new_pcb->proc_pid = itable[i]->id
        initialize pointer stack
        new_pcb->PC = itable[i]->PC;
        initialize new_pcb->status = READY
        initialize new_pcb->p1 = NULL
        initialize new_pcb->p2l = NULL
        rpq_enqueue(new_pcb) //Enque will take care of prioritizing
        // now set up the process context and stack
        if (setjmp (kernel_buf) == 0) {
            jmpsp = apcb->proc_stack;
            #ifdef i386
            __asm__ ("movl %0,%%esp" : "=m" (jmpsp)); // if Linux i386 target
            #endif // line 2
            #ifdef __sparc
            _set_sp( jmpsp ); // if Sparc target (eceunix)
            #endif
            if (setjmp (apcb->context) == 0)
            {
                longjmp (kernel_buf, 1);
            }
            else
            {
                void (*tmp_fn) ();
                tmp_fn = (void *) current_process->start_PC;
                tmp_fn (); // process starts for the first time here
            }
        }
    }
}
```

7.3 INITIALIZATION STAGE 3: INTERRUPT PROCESSES

The third initialization stage operates in almost exactly the same manner as the second stage. The i-processes, using a second hardcoded Initialization Table, will each be assigned their ID, initial PCB memory blocks, and initial contents of their respective PCBs. All i-processes will be assigned a priority of -1, which will indicate to invoking and receiving processes that they are i-processes. The Initialization Table, as with the second initialization stage, will be in the format of an array of size n , where n is the number of i-processes, and will contain the necessary data for the creation of each i-process. The i-processes are then placed in a dummy queue in the order in which they are created.

```
void init_i_processes ( )
{
    for ( i < n number of i_processes)
    {
        allocate new_pcb = (struct PCB *) malloc (sizeof (struct PCB))
        initialize new_pcb->priority = -1
        initialize new_pcb->proc_pid = iproctable[i].id
        initialize new_pcb->status = READY
        iproc_enq(new_pcb)
    }
    Fork crt and keyboard processes
}
```

7.4 INITIALIZATION STAGE 4: BEGIN RTX OPERATION

The fourth and final initialization stage will initialize the clock, transfer control of the CPU to the highest priority process in the ready queue, and begin operation of the RTX.

```
void begin_RTX( )
{
    if(init_queues( ))
    else return error message
    if(init_processes( ))
    else return error message
    if (init_i_processes( ))
    else return error message
    allocate system_clock = (struct clock *) malloc (sizeof (struct clock))
    system_clock.ss = 0
    system_clock.mm = 0
    system_clock.hh = 0
    release_processor( )
    process_switch( )
}
```

8.0 IMPLEMENTATION

While working on this design document, all team members aided in the writing and research of topics. Kal worked on the initialization, structs, and description and pseudocode of primitives, as well as on queues with Yasser. Yasser concentrated primarily on all the data structures, researched into the pseudocode for primitives, and found images for the report. Shari was elected Project Manager, and concentrated on interrupts and timing services, as well as half of the pseudocode for the primitives. Peter was in charge of learning UNIX processes as well as responsible for the keyboard and CRT i-processes.

For the partial and full initialization, we will be splitting up writing the code. Kal will be in charge of writing the initialization, the clock struct, and the request_process_status, release_processor, and terminate primitives. Yasser will be writing the data structures and the envelope primitives (send, receive, allocate, and deallocate). Shari will code the interrupt handler and timing services, and will work with Peter in coding the CCI and the Kernel Entry. Peter will also write the change_priority, send_console_chars, get_trace_buffers, and get_console_chars primitives.

For the partial implementation, we will all attempt to split up and write the code on our own. For the sections that are semi-independent, we will each attempt to test and debug before bringing all code together. Once all sections are functional, the group will meet and integrate our code together. We shall ensure that all variables are the same and attempt to run the system. As shown in the image below, the partial implementation primarily needs the code for initialization. It will also need a few primitives, a process, and i-processes. Because of this, we shall assist Kal in writing the initialization section since that must be completed before we can begin work on the other sections.

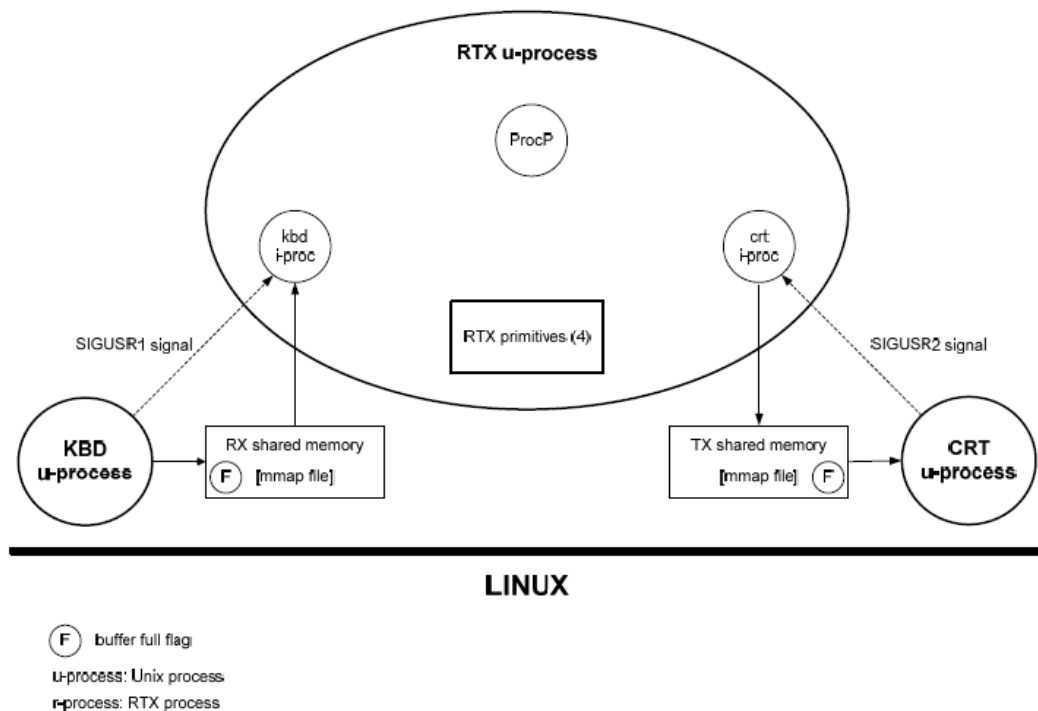


Figure 12: Partial Implementation (Seviora)

The full implementation will follow a similar format to the partial implementation.

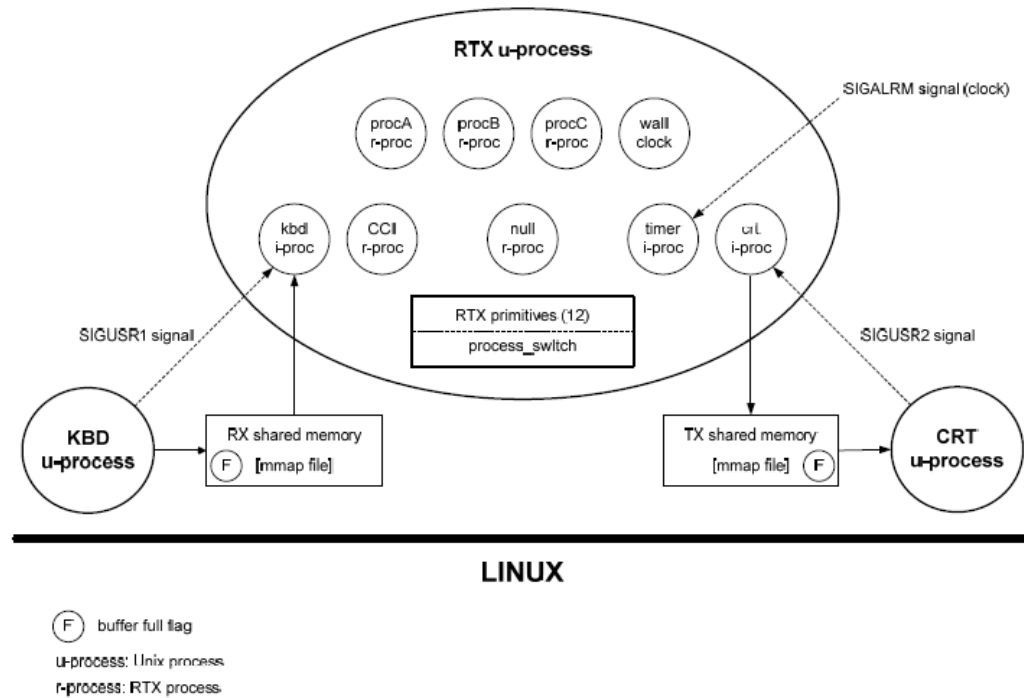


Figure 13: Full Implementation (Seviora)

The three test processes will be coded from the pseudocode given to us. The CCI, null process, clock, all primitives, and our three user processes will be implemented in the full implementation deliverable.

9.0 GLOSSARY

API – Application Process Interface

CCI – Console Command Interpreter

PCB – Process Control Block

PID – Process ID

RPQ – Ready Process Queue

RTX – Real Time Executor

UART – Universal Asynchronous Receiver / Transmitter

REFERENCES

Dasiewicz, P. (2009, September 10). *An Example of a Non-preemptive Real-Time Kernel*.

Retrieved October 20, 2011, from UWACE:

https://uwangel.uwaterloo.ca/AngelUploads/Content/UW-MCL-C-110819-134422/_assoc/D723C8F8E236439C8AE4937A14D1059E/rtxBook241V103.pdf

Reidemeister, T. (2010). *MTE 241 Project - Real Time Operating System*. Retrieved

October 20, 2011, from UWACE:

<https://uwangel.uwaterloo.ca/uwangel/section/default.asp?id=UW-MCL-C-110819-134422>

Seviora, R. (n.d.). *MTE 241 Project: Partial Implementation Deliverable*. Retrieved

October 20, 2011, from UWACE:

https://uwangel.uwaterloo.ca/AngelUploads/Content/UW-MCL-C-110819-134422/_assoc/6F54126C0AA649029FB10F1643D870FD/iRTXdescription-v1.0.pdf