TOC Baby

Introduction

This report details the design of a Real Time Executable (RTX) that emulates the functionality of an embedded operating system. The RTX and this design document are being completed for the MTE 241 Introduction to Computer Structures and Real-Time Operating Systems course.

The design of the RTX explained in this document is based on a highly modular coding philosophy, in which each portion of the RTX with a specific, broad function takes the form of its own class or process. These fundamental RTX classes are referred to as “Modules” in this report, and include the following: Messaging Services, Scheduling Services, Timing Services, Signal Handler and Console Input/Output. The report is organized so as to present each of these modules individually. It will be seen that the core logic of the majority of the RTX primitives resides in one, or many of these modules.

In the interest of organization, a standardized approach has been used to describe each subset of functionality required by the project description. Each of these sections is divided, where appropriate, into subsections Implementation and Class Structure/Provided Operations. The Implementation will describe the specific steps taken to achieve that goal, and the Class Structure/Provided Operations section will present pseudocode to illustrate the core logic of the class or function.

Finally, it is important to note that the constructor, destructor/termination and member variable get and set functions, as well as other similar standard class functions are not discussed in this report, as their existence is assumed where applicable.

RTX Class

The RTX class comprises the bulk of the RTX functionality. The class includes all RTX primitives, helper functions, global variables, and an instantiation of each module class.

Primitives

Due to the highly modular design of this implementation of the RTX, the RTX primitives have been kept as simple as possible. The majority of the RTX primitive functions simply call helper functions that belong to a more appropriate module. This helps preserve the functionality of the RTX class as well as the helper modules.

Send Message

This primitive is called by a process to send a message envelope containing data to another process.

Implementation

This function calls a Messaging Services class function. The implementation is included

here for convenience.

- Fill the origin and destination process id fields of the message envelope
- Record message information to message trace
- Determine if destination process is blocked on receive and if so, call unblocking function
- Place message on destination process' PCB mailbox linked list

Pseudocode

```
int RTX::send_message( int dest_process_id, MsgEnv * msg_envelope )
{
      call MsgServ.sendMsg with dest_process_id and msg_envelope as
      paramenters
}
```

Receive Message

This primitive is used to receive a message from another process for use within the invoking process.

Implementation

This function calls a Messaging Services class function. The implementation is included here for convenience.

- Check if the process' mailbox is empty and if so, the process becomes blocked on receive
- If there is a message waiting, the primitive dequeues the message, adds the information to the message trace and returns a pointer to the message

Pseudocode

```
MsgEnv * RTX::receive_message( )
{
      call MsgServ.recieveMsg()
}
```

Request Envelope

This primitive allows a process to request an envelope from the free_env_Q to send to another process.

Implementation

This function calls a Messaging Services class function. The implementation is included here for convenience.

- checks if free_env_Q is empty and if so, move invoking process to blocked on envelope queue
- if an envelope is available, dequeue it from the free_env_Q and give it to the process
- this primitive also erases anything in the envelopes struct members to ensure that there

is no random data carried over from initialization or previous use

Pseudocode

```
MsgEnv * RTX::request_msg_env( )
{
      call MsgServ.requestEnv()
}
```

Release Envelope

This primitive takes an envelope no longer required by the invoking process and returns it to the free_env_Q.

Implementation

This function calls a Messaging Services class function. The implementation is included here for convenience.

This primitive puts the envelope onto the free_env_Q and checks if there are any processes waiting for an envelope. If so, the waiting process is moved to the ready queue where it must wait for the processor before it can acquire the envelope. There is a possibility that the envelope will not be available when the process gets the processor again. In this case the process would have to request the envelope again. This implementation was selected due to the small volume of message traffic expected which will likely cause little delay if the process becomes blocked multiple times.

Pseudocode

```
int RTX::release_msg_env( MsgEnv * msg_env_ptr )
{
      call MsgServ.releaseEnv(msg_env_ptr)
}
```

Release Processor

This primitive allows a process to give up control of the processor, such that if another process is ready to execute, control of the processor would be given up to that waiting process.

Implementation

This functionality belongs to the scheduler, and so the Release Processor primitive merely calls the scheduler function by the same name. The implementation of that function is given here for convenience.

1. Save the context of the currently executing process
2. Put the current process back on the scheduler's ready queue (if no other process is ready, then this will be the "next ready process")
3. Allow the next ready process to start executing and restore its context

Pseudocode

```
int RTX::release_processor( )
{
```

```
	return Scheduler.release_processor;
}
```

Request Process Status

The Request Process Status primitive will return an array which contains the id, status and priority of every process. This is done by returning an envelope which contains the array.

The array will consist of a number indicating how many sets of process information are to follow, and then list that information in the order: pid, status, priority. See the following illustration for reference

Implementation

- Loop through PCB list, making note of each one's status and priority using the PCB class' get functions.
- Send message back to sender with a nicely formatted array of status' and priorities.

Pseudocode

```
int RTX::request_process_status( MsgEnv * msg_env_ptr )
{
	array returnArray

	for each PCB
	{
		status = PCB.get_status
		priority = PCB.get_priority
		Add status and priority to returnArray
	}
	Put returnArray into message contents
	send_message( invoking_process, msg_env_ptr )
}
```

Terminate

The terminate function will be invoked in order to cleanup and end the RTX. It will be invoked by the signal handler as a result of a SIGINT signal or any other terminating conditions. It will be responsible for signalling the RTX's child threads (KB and CRT) as well as freeing any resources used by the RTX. The function will immediately become atomic to prevent the termination sequence from being interrupted.

Both the KB and CRT child processes will be signaled to terminate by sending a SIGINT

signal to their respective signal handlers. Since all data structures are contained within classes, the parent class of each data structure will be responsible for cleaning up all resources retained by that class. This means the RTX termination function will not have to be modified as a result of a change in another class such as messaging. Note that the terminate function will be executing on the current_process's stack, and thus it is not de-allocated with the other stacks.

Pseudocode

```
int RTX::terminate( )
{
      atomic(true);
      smCleanup();
      terminate messaging class.
      terminate timing services class
      terminate scheduling services class
      free PCB stack if (PCB[i] != current_process)
      for (each PCB)
            terminate PCB class

      kill KB and CRT processes, wait before continuting
      return 0;
}
void smCleanup(){
{
      kill(keyboard_process, TERMINATE)
      wait();
      kill(crt_process, TERMINATE)
      wait();

      munmap (rx_memmap_pt)       //unmap the memory buffers
      munmap (tx_memmap_pt)

      close(rx_shared_mem_f_id)   //close temporary files
      close(tx_shared_mem_f_id)

      unlink(rx_shared_mem_f)     //delete temporary files
      unlink(tx_shared_mem_f)
}
```

Change Priority

This primitive allows a process to immediately change another process' priority.

Implementation

This functionality belongs to the scheduler, and therefore the Change Priority primitive

merely calls the scheduler function by the same name. The implementation of that function is given here for convenience. A priority change will be done by modifying the priority level variable inside the process' PCB to the desired priority.

Special cases that must be taken into account include determining whether the new priority is in the valid range of 0-3, whether the target process exists and whether the process is executing. If the process is executing, its execution will not stop, but its priority field will still be changed. If the requested value is the same as the current priority value, then no action will be taken.

Pseudocode

```
int RTX::change_priority( int new_priority, int target_process_id )
{
return Scheduler.change_priority(new_priority,target_process_id);
}
```

Request Delay

This primitive is used to put a process to sleep, effectively stopping its execution until the delay time has passed.

Implementation

- the function fills a message with the requested sleep duration and sends it to the timing i_process
- The invoking process moves to the blocked on env queue until the message is sent back with the wakeup_code message type from the Timing i_Process

Pseudocode

```
int RTX::request_delay( int time_delay, int wakeup_code, MsgEnv *
message_envelope )
{
      call MsgEnv.setMsgData(time_delay)
      write wake_code to message_envelope
      call RTX.send_msg_envelope to timing i_process
      put invoking process on blocked_on_recieve queue
      process_switch()
}
```

Console Send

The send_console_char acts as a bridge between processes and the console/screen, sending information from the system to the user.

Implementation

- Information to be sent to the display will be included as the contents of the message envelope input parameter (the invoking process must allocate this message envelope, but does not block while this primitive executes).

- The message is a character string in usual C++ string format, ending with a null character.
- This primitive moves contents of the envelope into TX shared memory.
- After the CRT process completes transmission, this primitive will return the input message envelope back to the invoking process with message type display_ack as acknowledgement that the transmission was successful.
- If the CRT process is busy and may not immediately print the message, this primitive loop MAX_LOOP times. If the transmission is not compelted in that time, -1 will be returned.

Pseudocode

```
int RTX::send_console_chars(MsgEnv * message_envelope )
{
      if(message_envelope.getMsgType().[last character] != NULL)
            return -1;

      int counter = 0;
      while(counter < MAX_LOOP)
      {
            if(CRT process not busy)
            {
                  string to_transmit = message_envelope.getMsgData();
                  copy to_transmit into shared memory buffer
                  //crt process will poll shared memory and will respond
                  message_envelope.setMsgType(display_ack)
                  send_message(message_envelope)

                  return 0;
            }
      counter++
      }
      return -1;
}
```

Console Get

The get_console_chars primitive will be used to gather information from the user and relay the user data to the invoking process.

Implementation

- The input parameter is a message envelope allocated by the invoking process.
- This primitive will extract data from RX shared memory, insert it in the provided envelope, and then return the envelope with message type console_input to the invoking process.
- The information may only be extracted if the keyboard process has already sent the RTX

a signal that information is ready to be transmitted.

Pseudocode

```
int RTX::get_console_chars( MsgEnv * message_envelope )
{
      if( keyboard_proc has already sent signal )
      {
            string user_input = extract information from shared memory
            message_envelope.setMsgData(user_input)
            message_envelope.setMsgType(console_input)
            send_message(invoking process, message_envelope)
            empty shared memory
            set flag to indicate that the buffer is cleared

            return 0;
      }
      return -1;
}
```

RTX Member Variables

const int PCB_COUNT

This constant retains the number of processes present in the RTX. It is used when iterating through processes.

PCB* pcbList[];

This array contains pointers to the process control block for each process executing on the system. Each PCB resides in the array index specified by its process_id variable (i.e. pcbList[i].process_id = i). Therefore the PCB for any process can be accessed directly without mapping.

input buf* rx_mem_buf

Pointer to the memory buffer that the RTX kernel will use to access the shared memory between itself and the keyboard process.

input buf* tx_mem_buf

Pointer to the memory buffer that the RTX kernel will use to access the shared memory between itself and the crt process.

Data Structures

Data structures used in this implementation include priority queues, contexts and PCBs. Additional simple data structures such as queues and linked lists that may be used throughout the project will not be discussed in this report.

Priority Queues

Priority queues will be used often in the project. Items with lower-number priorities are considered most urgent. For any two items at the same priority level, they are dealt with on a first-come-first-served basis.

Implementation

- One array of *n* FIFO queues will be used, where n is the number of priorities available.
- These queues are labeled 0-*n*, which signify the priority levels. 0 is the most urgent priority, and *n* is the least urgent.

Class Structure, Provided Operations

```
class PQ {
      public:
            void enqueue (PCB new_data, priority_level);
            PCB * dequeue ();
      private:
            Queue master[];
};

void PQ::enqueue ( PCB new_data )
{
      //Find priority level from PCB
      int priority_level = new_data.priority
      locate queue[priority_level].tail_ptr;

      Put new_data in the last position of the located queue.
}

PCB * PQ::dequeue ( )
{
      int priority;
      // Find most urgent queue entry
      for (i = 0 to n){ // Find highest non-empty priority level
            if (master_queue[i] is not empty)
            {
                  set priority = i;
                  exit for;
            }
      }
      PCB return_value = master[priority].head_ptr
      Increment master[priority].head_ptr
      return return_value;
}
```

Context

Saving and restoring the context of a process are often-used operations. The context class exists to facilitate these procedures.

Implementation

- Two functions are provided to save and restore context respectively.
- The setjmp() and longjmp() functions are used to achieve context saves.
- Save returns the value of the set_jmp call, so that the function may be used in a similar way to actually calling set_jmp.

Class Structure, Provided Operations

```
class Context
{
      public:
            save();
            restore();

      private:
            jmp_buf local_jump_buf;
};

int Context::save( )
{
      return set_jmp( (*this).local_jump_buf );
}

void Context::restore( )
{
      long_jmp( (*this).local_jmp_buf, 1 );
}
```

Process Control Block (PCB)

The PCB will consist of all relevant information pertaining to a process. Pointers to PCBs will be handled by other processes which concern themselves with PCBs, such as the scheduler.

Implementation

The PCB will be represented as a class in our project. Wherever possible, member variables have been kept public to allow a high level of control to the many modules that modify PCB information.

The PCB stack will be accessed using a character pointer in order to allow for byte level addressing precision. The stack pointer will be set when the stack memory is allocated, and will

have an offset of 4 bytes.

Class Structure, Provided Operations

```
class PCB
{
      public:
            int processType;
            char* stack;
            (void*) fPtr;          //Function pointer
            Context context;       //Includes jmp_buf
            int atomicCount;
            LL_List mailbox;       //Message mailbox

            int set_priority( int pri );

      private:
            int id; //Process id
            int state;
            int priority;
}

int PCB::set_priority( int pri )
{
      if(pri is valid 0 to 3)
      {
            this.priority = pri;
            return 0;
      }
      return -1;
}
```

Initialization

A proper initialization of the RTX is critical to reducing potential problems downstream. Since classes are used extensively in the RTX, each class is relied upon to perform the instantiation of all of its data structures and variables. This means that there is memory allocation occurring outside of the application's main function, however all memory allocation occurs before the first process is en-queued and as such it is believed that this strategy still conforms to the project requirements.

Implementation

This is the first section of code to be executed once the RTX application is executed by the Linux Shell. Once the initialization table is created, the RTX class is instantiated. Any initialization problems are detected through the use of the project specific "ensure" function described below. Any errors from which the RTX cannot recover (i.e. memory allocation error),

result in the application exiting and debugging information being displayed in the Linux shell.

Shared Memory

Shared memory is used in order to allow the RTX to read data stored by the keyboard process. This memory in initialized early in the initialization so that its location can be passed as an argument to the Keyboard process. The shared memory is allocated in the form of a memory map file. Upon failure, the application will return an error to the user and terminate. For more information regarding the use of shared memory, see the Console I/O section.

Pseudocode

```
int smInit(){
      for(2 iterations)     //set up the file
      {
            create file with permissions for owner rwx access only
            verify file created successfully
            truncate file to 128bytes to match the memory buffer
            //note: first file is rx_shared_mem_f, second file is
            tx_shared_mem_f
      }

      in_pid = fork keyboard process
      if(in_pid == 0)  //deal with child process
      {
            //rx_shared_mem_f descriptor is an arg
            execl(./keyboard, keyboard args);
            //should never reach here
            declare execl failed and terminate
      }
      sleep(1);  //give keyboard process time to start

      out_pid = fork crt process
      if(out_pid  == 0)  //deal with child process
      {
            execl(crt args); //tx_shared_mem_f descriptor is an arg
            //should never reach here
            declare execl failed and terminate execution
      }
      sleep(1);  //give crt process time to start

      //create shared memory buffers for two processes
      caddr_t rx_memmap_pt = mmap(keyboard args);
      if(memory mapping not successful)
            cleanup and terminate execution
```

```
      caddr_t tx_memmap_pt = mmap(crt args);
      if(memory mapping not successful)
            cleanup and terminate execution

      //create membuf pointers so RTX kernel may access the shared mem
      rx_mem_buf = (inputbuf*) rx_memmap_pt
      tx_mem_buf = (inputbuf*) tx_memmap_pt
}
```

Initialization Table

The initialization table is created to simplify the initialization of each process. Since each process's PCB must be filled with hard-coded information, it is simpler to manage the processes' information if it is all assigned sequentially rather than spread out throughout a series of initializations. Having a data type to create each process's PCB also allows for a simple for loop rather than repeated code (more in the Process Initialization section, below).

Implementation

- Each process's hard-coded information will be stored in a customized struct ('PcbInfo') in order to allow simple assignment and retrieval of the information.
- The initialization table will be an array of pointers to PcbInfo objects.
- The array size will be adjusted to the number of processes.

Class Structure, Provided Operations

```
struct PcbInfo
{
      unsigned int    processId;
      unsigned int    priority;
      unsigned int    stackSize;
      unsigned int    processType;
      (void*)                    address;
      bool                            i_process;
};

void createInitTable()
{
      PcbInfo* initTable[PROCESS_COUNT]

      for (0 to PROCESS_COUNT -  1)
      {
                  initTable[i] = allocate(PcbInfo);
                  ensure initTable[i] != NULL
      }
       initTable[0].processId =                                  1;
```

```
    initTable[0].priority =                    1;
    …
    initTable[1].processId =                                2;
    …continue initializing each process sequentially
}
```

RTX Instantiation

With the initialization table created, the RTX class will be instantiated. The RTX contains many other classes, which each have their own initialization functions. This cascading effect assures that all necessary initialization for each class.

Signal Handler

It is critical that the signal handler be initialized properly so that RTX process can communicate with its KB and CRT children processes as well as receive timing alarms. All of the signal handler initialization is performed by instantiating an instance of the signalHandler class. The signalHander constructor associates the expected signals to the static signalHandler::handler function. This will be done using a series of 'sigset' calls for each signal. For more information on handled signals, see the Signal Handler Class section.

Once the signal handler has been initialized, the signalHander.sigSetBlocked signal set must be created. To do so, a sigset_t is initialized using 'sigemptyset' to mark it as a set of signals to be masked by the system. sigaddset is then used to add each signal to be blocked.

Once the set is created, the global pointer setAtomicOn is set to point at the set so that it can be used during 'atomic' function calls. The sigSetHandled signal set must subsequently be initialized. This is done using sigprocmask to copy the existing set (created earlier) into the struct.

PCB Creation

Each PCB needs to be initialized before its process is ready to initialized and en-queued for the first time. To do this, the initialization table will be looped through with each PCB initialized individually. At this time, the stack of each PCB will be allocated. The stack size will be four kilobytes unless this value proves to be insufficient during testing.

First, the PCB each must be allocated and filled with the process's information stored in the initialization table. With the process's basic information stored, the PCB's stack will be allocated using the c++ 'new' method With the PCB created and initialized, the memory allocated for the PCB's associated initialization table entry will be freed since it has been copied into the PCB.

Messaging Class

As stated above, the initialization of the messaging envelopes will occur when the RTX class instantiates the messaging class. At this point 10 envelopes will be allocated for the RTX. If this value proves to be insufficient, it will be increased accordingly. Due to the relatively large amount of memory available, it has been determined that it is more important to maximize processor time (avoid blocking situations) than memory usage. Since envelopes are classes, they will be allocated using the c++ new command. An array of pointers to each envelope will be

maintained in order to allow for efficient deletion of all envelopes during termination.

Message trace buffers will also be allocated during Messaging class initialization. As a class, the trace buffers will also be allocated using the new command.

Scheduling Services

The next step in the initialization process will be to instantiate the scheduling class. Once the necessary queues are intantiated, each process will be initialized so that it is ready to run when scheduled. This will be performed using the logic provided on pages 13 and 14 of the MTE241 “rtxinitialization.pdf” document availible on UW-ACE.

With the queues and processes initialized, all of the processes in the READY state will be en-queued into the ready queue.

Keyboard and CRT Initialization

The keyboard and console interpreter will be the last things to be initialized before the RTX is “started”. The I/O processes will run independently of the RTX process and as such, two ‘fork’ commands will be used to start the processes. Since the code for each process will be stored in a separate file, the ‘execl’ command will be used to execute their respective code.

Both the keyboard and the CRT processes will receive the RTX process’ process id and shared memory locations as arguments in their initialization. The RTX will also store both child process ids so that they can be terminated when the RTX terminates.

Initialization of the CRT and keyboard processes involves the creation of the shared memory segments between the RTX and the I/O processes. With the shared memory initialized, the first process is moved onto the processor and the RTX begins operating.

Messaging Services

Messaging Class

This class will incorporate the messaging services functions required to communicate between processes.

Implementation

- Four public functions will be used to send and receive messages between functions
- The private member free_env_Q will hold the free envelopes that can be requested by processes

Psuedocode

```
class MsgServ
{
      public:
            int sendMsg( int destPid, MsgEnv* msg )
            MsgEnv * recieveMsg( )
            int releaseEnv( MsgEnv * msg )
            MsgEnv * requestEnv( )
```

```
      private:
            LL_List free_env_Q
};

int MsgServ::sendMsg( int destPid, MsgEnv * msg )
{
      if (destPid is not valid or msg is NULL)
            return -1 //error
      origin_PID = RTX.scheduler.getCurrentProcess.processId;
      destication_PID = destPid
      call MsgTrace.add_trace(MsgEnv * msg, SEND)
      if (get_proc_status(destPid) = BLOCKED_ON_RECEIVE)
            call unblock_process(destPid)
      else if(get_state(destPid) = SLEEPING)
      {
            //allows the process to remain in sleep mode until correct
            //msg is sent
            if(MsgEnv.getMsgType = wakeup_code)
                  call unblock process
      }
      call linked list enqueue function to enqueue msg on to PCB
      mailbox
      return 0 //success
}

MsgEnv * MsgServ::receiveMsg()
{
      if (PCB mailbox is empty)
      {
            call block_process(process_id, BLOCKED_ON_RECEIVE)
            process_switch()
      }
      dequeue msg env from PCB mailbox
      call add_trace(MsgEnv*ptr, RECEIVE)
      if (current_process = i_processA)
      {
            return NULL
      }
      return pointer to msg env
}

int MsgServ::releaseEnv(msgEnv* msg)
{
      if (msg = NULL)
            return -1 //error
```

```
      call free_env_Q.add_env(msg)
      tempPtr = is_blocked_on_envelope()
      if(tempPtr is not NULL)
            call unblock_process(tempPtr)
      return 0 //success
}

MsgEnv* MsgServ::requestEnv()
{
      if (free_env_Q is empty)
      {
            call block_process(process_id, BLOCKED ON ENV)
            process_switch()
      }
      call dequeue function on free_env_Q
      clear env fields
      return (MsgEnv* ptr)
}
```

Message Envelopes

Message envelopes are the fundamental components of interprocess communication. They are used to facilitate the sending of information between processes.

Implementation

- A class containing a private struct will hold the required information used to represent the message envelope.
- A number get and set function will be implemented to access the private struct members. These were not included in the psuedocode below to due to the self explanatory nature of the functions.
- The setMsgType function will be used to set the size as well as input data into the msg_data struct member

Psuedocode

```
class MsgEnv
{
      public:
            int setMsgData( string )
            //get and set functions for each struct member
      private:
            struct MsgFields
            {
                  int destitation_PID
                  int origin_PID
                  string msg_type
                  string msg_data
```

```
            }msgFields;
};

int MsgEnv::setMsgData( message information )
{
      sets the msg_data and determines the msg_data size most
      appropriate for the data
}
```

Message Trace

The message trace is used to store the most resent message send and message receive calls. It will be used primarily during debugging to determine if message transactions are executing correctly.

Implementation

- Class structure with two, 16 member circular arrays as private members
- Function to add a message to the trace
- Function to send all 32 successful transactions to the screen, used in termination

Psuedocode

```
class MsgTrace
{
      public:
            int add_trace( MsgEnv * msg, int calling_function )
            MsgEnv * get_traces( MsgEnv * msg )
      private:
            struct TraceElement
            {
                  int dest_PID
                  int origin_PID
                  int msg_type
                  int time_stamp
            };

            TraceElement sendArray[16]
            int sendArrayPosition
            TraceElement recieveArray[16]
            int receiveArrayPosition
}

int MsgTrace::add_trace( MsgEnv * msg, int sendOrReceive )
{
      if (msg = NULL or sendOrReceive is invalid)
            return -1
      if(sendOrReceive = SEND)
```

```
      {
            add info from msg to sendArray[sendArrayPosition]
                  using MsgEnv class functions
            sendArrayPosition increment for circular array
      }
      else //RECEIVE
      {
                  add info from msg to receiveArray[receiveArrayPosition]
                  using MsgEnv class functions
            receiveArrayPosition increment for circular array
      }
      return 0
}

MsgEnv* MsgTrace::get_traces( MagEnv * msg )
{
      if (msg is NULL)
            error
      format trace data into table
      place table in msg_data field of MsgEnv *msg
      send msg to console I/O
}
```

Scheduling Services

This module provides the process scheduling functionality for the RTX. It will control which order processes get to execute based on the priority of each process. A FIFO priority queue as described in the Data Structures section will be used to keep track of which process should next be allowed to execute on the CPU. Four levels of priority will be available (0-3) with priority 0 being the most urgent priority. If two processes of the same priority are ready to execute, then they will be treated in a FIFO order.

Module Class Header

```
class Scheduler
{
      public:
            int release_processor( );
            int change_priority( PCB * target, new_priority );
            int process_switch( );

            //Place new process on ready queue
            bool add_ready_process( PCB * target );
            bool block_process (PCB * target, string reason );
            bool unblock_process( PCB * target );
```

```
            //Returns if a process is currently blocked on envelope
            bool is_blocked_on_envelope( PCB * target );

      private:
            PCB * current_process; // Executing state
            PQ ready_procs; // Ready to execute state
            PQ blocked_env_procs; // Blocked on resource state
            LL_List blocked_msg_recieve;

            int context_switch( PCB * next_proc );
            int context_save( );
};
```

Process Management

Release Processor

This function allows a process to surrender control of the processor. This is achieved checking is there are any other processes that are ready to execute. If so, control of the processor would be given up to the highest priority of those processes in accordance with the usual scheduler algorithm.

```
int release_processor( ) {
      current_process.context.save

      Put the current process back on the scheduler's ready queue

      Allow next process to start executing and restore its context
}
```

Change Priority

This function allows a process to immediately change another process' priority.

Implementation

A priority change will be done by modifying the priority level variable inside the process' PCB to the desired priority.

Special cases that must be taken into account include determining whether the new priority is in the valid range of 0-3, whether the target process exists and whether the process is executing. If the process is executing, its execution will not stop, but its priority field will still be changed. If the requested value is the same as the current priority value, then no action will be taken. If the process is currently in a priority queue, then the process' position in that queue will be corrected.

Pseudocode

```
int Scheduler::change_priority(int new_priority,int target_process_id)
```

```
{
      Perform checks on parameters

      Locate process' PCB and set it to the specified value.

      If (the PCB is in a ready/blocked queue ) {
            //Reset the PCB's position in the ready queue
            Scheduler.ready_queue.dequeue(PCB)
            Scheduler.ready_queue.enqueue(PCB)
      }
}
```

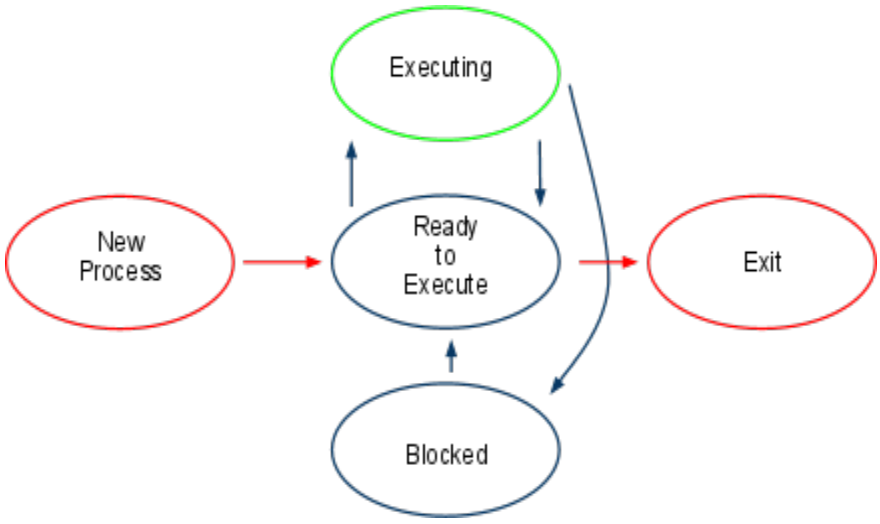
Process State Model

The process state model chosen for use is a five-state system. In this non-preemptive operating system, processes will exist in the new and exit states only at initialization and termination of the operating system. These one-time phases are marked with red arrows in the following figure.

Implementation

The processes ready to execute and the blocked processes (coloured green) will be stored on queues by the scheduler. The executing process is stored as a private variable that belongs to the scheduler. The scheduler moves processes back and forth between the executing, ready and blocked states, while movement to and from the new and exit states is performed by the initialization and termination procedures. Blocked states include waiting to receive a message from a process or blocked on envelope.

See the entry for priority queues for a detailed explanation of how the "ready to execute" priorities are handled.



Process State Diagram

Block/Unblock/Ready Process

This group of functions allows control of blocking/unblocking and making ready other processes.

Implementation

Each of these functions controls the position of a target PCB on the ready/blocked queues. Therefore, a target PCB will be passed to the function, and the function will change the PCB's state, and place it on the appropriate queue.

Processor Management

Null Process

The null process ensures that the processor is always in use. This is done by providing arbitrary functionality with the sole purpose of using CPU cycles.

Implementation

The null process shall loop, continually releasing the processor to any ready process. The null process is kept clean and simple to keep it as close as possible to its intended functionality. The null process has a priority of 3 (lowest priority) to ensure that it only executes when no other process is ready to execute.

Pseudocode

This pseudocode is based on the code given in the "Sample Kernel Design" document.
Extra "time wasting" functionality was not added to preserve the fundamental purpose of the null process.

```
void Scheduler::null_process( ){
      while(true)
            release_processor();
}
```

Process Switch/Context Switch

The process_switch() function and context_switch() function go hand in hand as process_switch() calls context_switch(). Therefore the implementation of the process switch procedure shall be divided into sections based on these two functions.

Implementation

The process_switch() function will figure out which process should be next set to executing based on the scheduler's priority queues. It will then analyze that process' PCB, and call context_switch(), passing it the necessary information, namely the PCB pointer. Once the process that has been switched out returns to an executing state, it will continue execution exactly from where it left off.

The context_switch() function will save the context of the currently executing process, set the process pointer that was passed to it to be the currently executing process, restore this process' context, and then set the new process' state to executing.

Pseudocode

```
int Scheduler::process_switch( )
{
      new_proc = highest priority proc from the scheduler's priority
queue

      context_switch(next_proc);
}

int Scheduler::context_switch ( PCB * next_proc ){
      //Switch the value of current_process.
      old_proc = current_process
      current_process = next_proc

      //Save context of current_process
      save_return = old_proc.PCB.context.save();

      //Restore context of next_proc iff setjmp is not returning from
      //a long_jmp
      if (save_return == 0)
            current_process.PCB.context.restore();

      Set new process' state to executing
}
```

Timing Services

Wall Clock Class

This class is used for storing the wall clock to be displayed by the displayClock process. upon command in the console command interpreter.

Implementation

The _isDisplayed boolean defines whether the wall clock should be displayed in the CCI. The increment function adds the tick length to the second count, and updates the wall clock if the next second is reached by sending an envelope to the CCI process. The increment function also updates the display if appropriate. If no envelopes are available then it aborts the attempt and will try again next tick.

Class Structure, Provided Operations

```
class wallClock
{
      public:
            int increment( );
```

```
            int setTime( int hours, int minutes, int seconds );
            string toString();

      private:
            int _hours = 0;
            int _minutes = 0;
            int _seconds = 0;
            bool _isDisplayed;
            bool _isNewTime;
}

int wallClock::increment()
{
      add tick value to _seconds
      if (_seconds increments full second){
            increment mins/hrs as appropriate
            _isNewTime = true;
            if (_isDisplayed AND _isNewTime){
                  MsgEnv myMsg = request envelope;
                  if (myMsg != NULL){
                        send_console_chars(toString());
                        _isNewTime = false;
                  }
            }
      }
}
string wallClock::toString()
{
      return formatted string of time variables;
}
```

Signal Handler

The signal handler will be implemented as a class which will exist within the RTX class and will be instantiated during the RTX class initialization. The SignalHander class will contain all of the methods relating to signal handling and atomicity. It should be noted that the actual signalHandler method will be static so that it is capable of handling signals.

Please see the section on initialization for information regarding the signalHandler class initialization.

Pseudocode

```
class signalHandler
{
      public:
            static void handler( int sigNum );
```

```
            int setSigMasked( bool masked );
      private:
            sigset_t setBlocked;
            sigset_t setHandled;
}
int setSigMasked( bool masked )
{
      if (masked)
            //set current signal mask to setBlocked (sigprocmask)
      else
            //set current signal mask to setHandled (sigprocmask)
}
static void signalHander::hander( int sigNum )
{
      PCB* prevProcess = RTX.scheduler.getCurrentProcess();

      switch (sigNum)
      {
            case SIGINT:
                  RTX.Terminate();
            case SIGALARM
                  set scheduler.currentProcess to timer i_process
                  execute timer i_process using PCB.address
            case SIGUSR1
                  set scheduler.currentProcess to keyboard i_process
                  execute keyboard i_process using PCB.address
            case SIGUSR2
                  set scheduler.currentProcess to crt i_process
                  execute crt i_process using PCB.address
      }
      set scheduler currentProcess to prevProcess
      //executing code automatically returns to previous process
}
```

Atomicity

When a process goes 'atomic', it cannot be interrupted by system signals until it returns to a normal state. This function allows for this capability.

Implementation

This is accomplished by masking all of the signals applicable to the RTX. Any signals that arrive while a process is atomic are automatically queued by Linux. This is critical in functions which access shared resources (i.e. envelopes) and where interruptions may cause data corruption. Signals are masked using a signal set, to which all of the handled signals are added. The 'sigprocmask' function is used to accomplish this with the help of 'sigaddset'. The

following signals are masked upon going atomic:

- SIGALRM
- SIGINT
- SIGUSR1
- SIGUSR2

Pseudo Code

```
void atomic( bool on )
{
      signalHandler.setSigMasked(on);
}
```

i_Processes

Timing i_Process

The timing i_process is given the processor when the Signal Handler (described below) receives a UNIX timing signal. This process keeps track of these clock pulses for use in the delay process primitive as well as the user display clock.

Implementation

- Increments internal counter
- Moves messages from PCB mailbox to internal ordered linked list. This list holds the pending messages in ascending order of time delay requested
- Determines if the time delay requested has been met and processes messages accordingly
- Calls the wall clock class to increment the user display clock

Psuedocode

```
void timing_iProcess( )
{
      increment clock tick count
      if(timing_iProcess PCB mailbox is not empty)
      {
            receive msg
            put msg onto ordered linked list based on comparison of
            clock pulses requested
      }
      if (first msg in ordered linked list has expired)
      {
            MsgEnv.setMsgType(wakeup_code)
            call MsgServ.sendMsg to send msg back to process
      }
      call WallClock.increment() //which will update clock and
                  //display it to screen if full second has passed
```

```
}
```

Keyboard i_Process

This i_process will be executed any time the keyboard reader process sends a signal to the RTX that either the RX shared memory is full, or the user has issued a command. The i_process will wait until the get_console_chars is called. When get_console_chars is called, the information is extracted from the shared memory and set a may_get_chars flags to true to indicate that the keyboard process may continue accepting user input. The keyboard processes must be “disabled” (wait until the get_console_chars is called) so the commands are performed sequentially and no information is lost or overwritten in the shared memory.

```
void i_keyboard_handler( )
{
      set flag to indicate signal has been received

      MsgEnv * wakeup = request_envelope()

      //wait for get_console_chars to extract data
      while(data not been extracted by get_console_chars from RX_shmem)
      {
            request_delay(100us, WAKEUP_code, wakeup)
      }

      release wakeup envelope
      set flag to indicate that shared memory has been cleared
}
```

CRT i_Process

Once the CRT has completed transmission it will send the RTX a signal as notification. The i_CRT_handler must accept this signal and correspondingly update a may_send flag.

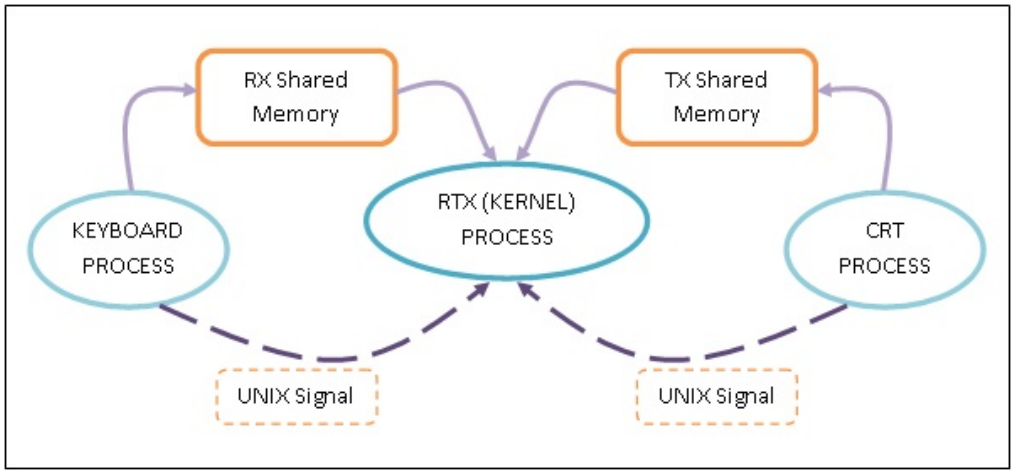
Pseudocode

```
void i_crt_handler()
{
      set may_send flag to indicate CRT may continue
}
```

Console I/O

Normally, embedded microcomputers communicate with the system console via an advanced UART (universal asynchronous receiver/transmitter). The UART, through the use of a Receiver and a Transmitter, transports input from the keyboard to the microprocessor’s pertinent buses, and also transports information from those buses out to the system console/

character display. This UART will be emulated in the UNIX environment through the use of two processes, called the KB (keyboard) and the CRT (cathode ray tube/character display) processes. The emulation is achieved through the use of these two processes in addition to shared memory and UNIX signals. It is also noteworthy that since these processes act as intermediaries between the UNIX system and the RTX kernel, they would be blocked on some UNIX I/O call instead of the kernel processes.



A Schematic View of the Interactions between KB, CRT and Kernel Processes

Keyboard Process

The KB Process emulates the Receiver component of the UART. It collects the input provided by the user through the use of the keyboard and then sends this data to the kernel process.

The KB process accepts the input characters and stores them in shared memory. The UART Receiver uses a 128-byte buffer for this purpose, and as such the size of the shared memory should be an equivalent 128-bytes (a detailed description of the realization of shared memory is located in the Shared Memory section).

Once the KB process detects that the input characters match a pre-specified stopping pattern (ex., the “Enter” key/end-of-line/carriage-return), or if the buffer becomes full, the KB process will transmit a UNIX signal to the RTX process to indicate that the input has been collected. The RTX then handles the signal by retrieving the collected information from the shared memory and dealing with it accordingly. After extracting, the RTX will clear the buffer. The keyboard process must wait until the buffer is cleared before accepting any more input to prevent any information being lost, overwritten or destroyed.

Implementation

- A user signal will be defined SIGUSR1, which will be handled by the RTX’s keyboard i_process.
- The keyboard reader process will copy any input that is provided by the user into the RX shared memory.
- If memory becomes full or the terminating character is provided, the KB process will send SIGUSR1 to the RTX.

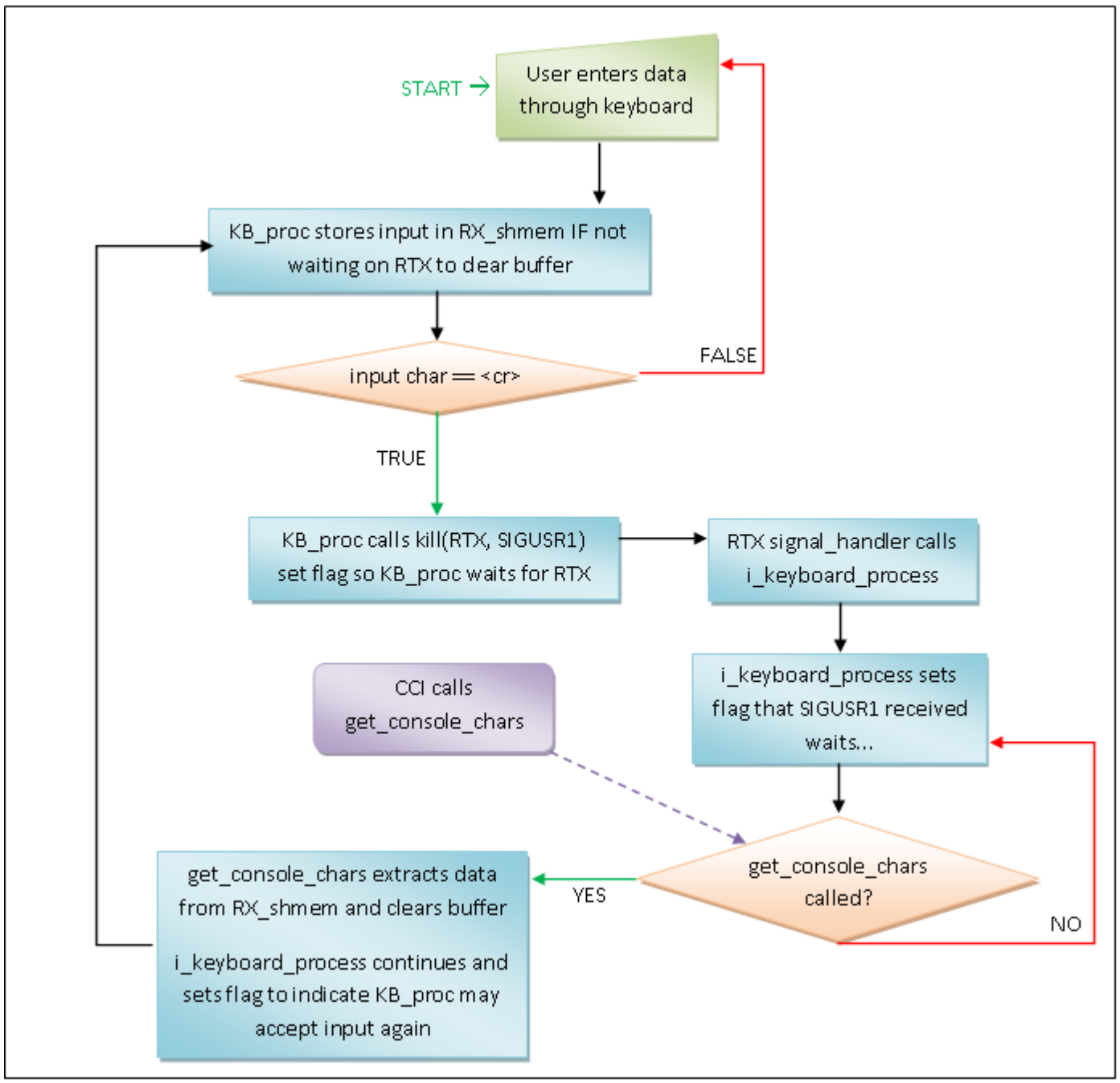
- The KB process must wait for the shared memory to be cleared by the RTX before accepting more input.
- The RX shared memory is allocated during initialization, before the KB process is forked

Pseudocode

```
void keyboard_die (int signal)
{
      exit(0) //for termination
}

void main( int argc, char * argv[ ] ) //keyboard_reader_main
{
      set signal handler for SIGINT signal to call keyboard_die

      //perform mem mapping to the temporary file using input arguments
      caddr_t rx_memmap_pt = mmap(input args);
      if(memory mapping not successful)
            cleanup and terminate execution of keyboard process

      //create membuf pointer so the kb_proc may access the shared mem
      inputbuf* rx_mem_buf = (inputbuf*) rx_memmap_pt

      while(true)
      {
            if( KB process not waiting for RTX to clear memory )
            {
                  user_input = getchar()
                  if(user_input != NULL)
                  {
                        if character is backspace
                              remove previous entry in shared memory
                        else if (character is carriage return)
                        {
                              set flag that KB_proc waits for RTX
                              send SIGNUSR1 to RTX
                        }
                        else
                        {
                              add to shared memory using rx_mem_buf
                              if (memory overflows)
                              {
                                    set flag that KB_proc waits for RTX
                                    send SIGNUSR1 to RTX
                              }
```

```
                        }
                    }
                }
                sleep(SLEEP_TIME); //don't use up all processor time!
        }
}
```

Below is a schematic representation of the input workflow. This schematic displays the entire process of data input from the user inputting information through the keyboard to the CCI invoking get_console_chars.



Sequence of Events involving User Input

CRT Process

The CRT Process is essentially a dual of the KB process, emulating the Transmitter component of the UART. The CRT process takes data from the kernel process and transmits this information to the user via the system console. It will also use its own block of 128-bytes of shared memory.

Here, the RTX will be populating the shared memory, and as soon as characters begin being populated in the memory, the CRT process will transmit them to the console. Once the

last removed character is a null character (signifying that the entire message provided by the RTX has been transmitted), the CRT process will send a UNIX signal to the RTX indicating that it has finished the transmission, at which point the RTX is free to send another message to be transmitted to the screen.

Implementation

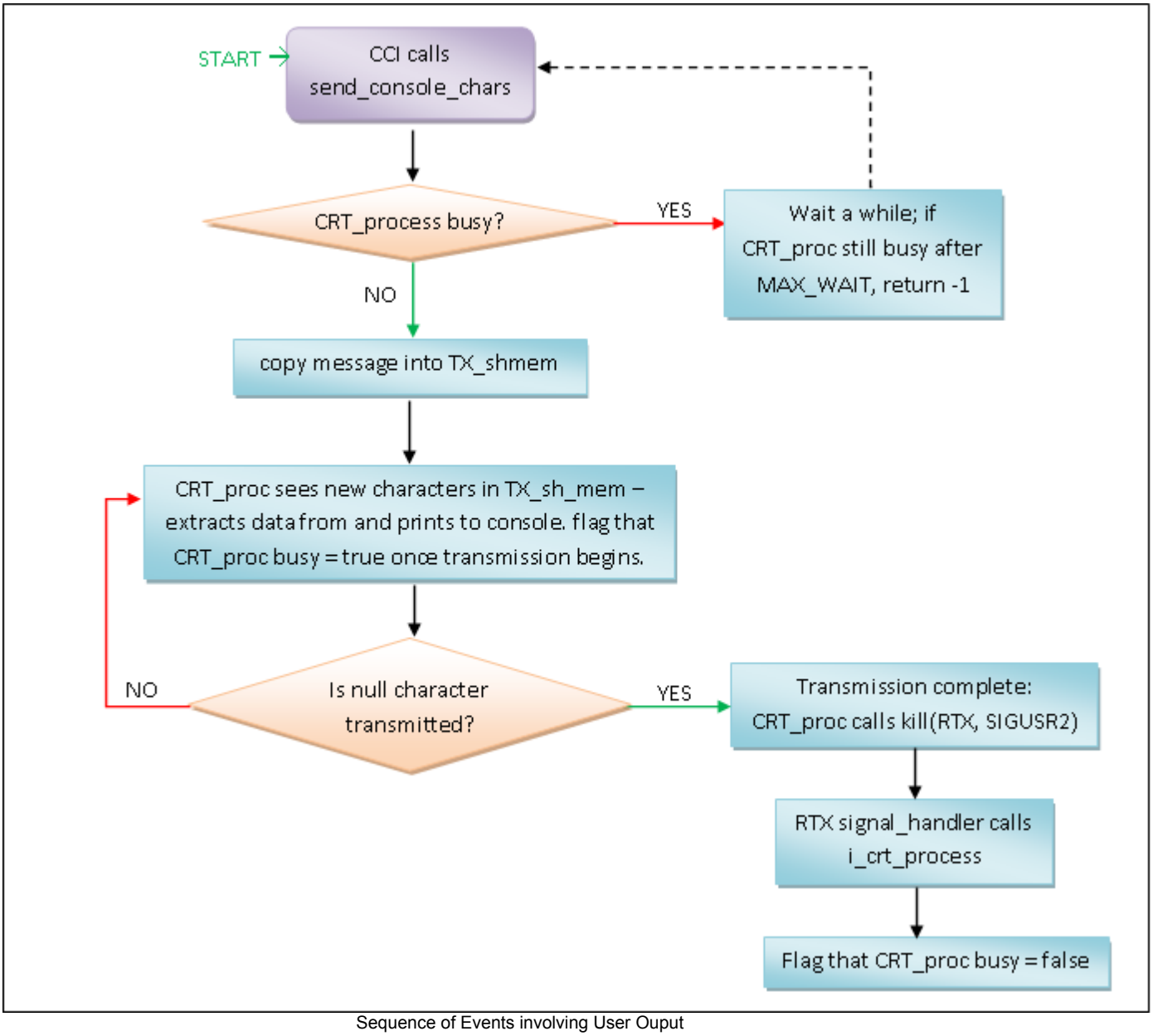
- The RTX process will append characters to the shared memory as desired.
- The CRT process is polling the shared memory, and if ever it finds that there is a character string to transmit, it will copy the provided characters to the screen and remove that character from shared memory.
- Once entire message is transmitted, the CRT_proc will send a signal to RTX as acknowledgement

Pseudocode

```
void crt_die ( int signal )
{
      exit(0) //terminate the process
}

void main( int argc, char * argv[ ] ) //crt_writer_main
{
      set signal handler for SIGINT signal to call crt_die

      //perform mem mapping to the temporary file using input arguments
      caddr_t tx_memmap_pt = mmap(input args);
      if(memory mapping not successful)
            cleanup and terminate execution of crt process

      //create membuf pointer so the crt_proc may access the shared mem
      inputbuf* tx_mem_buf = (inputbuf*) tx_memmap_pt

      while(true)
      {
            //check if RTX has sent complete message
            if( TX_sh_mem is not empty )
            {
                  string out = take char from TX_shmem via tx_memmap_pt
                  printf(out);
                  remove out from shared memory
                  //tx_shmem should be empty at this point
                  if( out was NULL indicating end of message )
                        send SIGUSR2 to RTX
            }
            sleep(SLEEP_TIME); //dont use up all processor!
      }
```

}

Below is a schematic representation of the output workflow. This schematic displays the entire process of data outpt from the CCI invoking send_console_chars printing data to the screen.



Sequence of Events involving User Ouput

Shared Memory

To realize the shared memory required to communicate between the RTX and the KB/ CRT processes, a subset of POSIX standard functions will be used. The ideology behind this implementation is to use a memory mapped file as the shared memory/buffer. A temporary

read/write file will be created with permissions restricted to owner “rwx” access only. When the I/O helper process is forked, it inherits the file along with its permissions. Next, both the parent (RTX) and child (keyboard and crt) processes map the temporary file to a memory buffer using mmap, and they use an inputbuf* to deal with the memory buffer. Thus when either parent or child process access the memory buffer through the pointer, they are accessing the shared, temporary file.

To create the temporary file in an accurate simulation of the transmitter or receiver of the UART, the file must be the same size as the required shared memory segment – 128 bytes.

Once the shared memory is no longer needed, the memory buffer must first be unmapped, then the temporary file must be closed and deleted. Note that two temporary files are needed, one for the KB process and another for the CRT process.

The POSIX functions of interest are

- *ftruncate* - forces the size of the file to the required 128 bytes
- *mmap* - maps the temporary file to a memory buffer
- *munmap* - unmaps the file from memory
- *unlink* - unlinks/delete the temporary file (must be closed first)

Should any errors occur during execution, the appropriate cleanup of the temporary files and memory buffer should be performed. The pseudocode for both shared memory creation and cleanup is located in the initialization and termination sections of this report, respectively.

Console Command Interpreter

The Console Command Interpreter (CCI) is an RTX user application process of high priority that interfaces between the user and the RTX. The CCI allows the user to monitor or affect the execution of the RTX and its processes. Interfacing between the user and the RTX, the CCI must accept keyboard input from the user and send it to the RTX through the `get_console_chars` primitive, and it must also transmit information from the RTX onto the screen for the user with the `send_console_chars` primitive.

Parsing Commands

The text entered by the user can be divided into three categories: a command, a parameter and a carriage return. As such, the input will be stored in an array of three strings named `arrInput`. `arrInput` consists of a command string as well as two parameter strings (since this is the maximum number of expected parameters).

Once an envelope is received, its contents are parsed by spaces. Therefore all of the characters up until the first space are moved (without the space) into the command string. If there are any characters left, the following group (separated by a space) is moved into the first parameter string. If there are any further characters remaining, they are moved into the second parameter string. At this point, if there are any characters remaining a syntax error will be displayed to the user.

Once the input string has been parsed, the command string will be passed through a

switch statement which defaults to an error message if the command does not match any of the predetermined commands. Should the command string match a command, the parameter(s) will be validated using case specific validation. If any parameter validations fail, or if too many parameters are specified, an error message will again be displayed to the user. If all validations pass, then the appropriate command will be issued to the RTX.

Pseudocode

```
int parseString( string input, string * output[], string token, int
maxCount ){
      while(strPos != end_of_string){
            //copy next token from 'input' to *output[count];
            if (++i > maxCount)
                  //break, return error
      }
      return i;
}

int processCCI( ){
      do{
            while(the KB process waiting for RTX to clear buffer)
                  wait for KB process to be ready

            send command prompt message (send_console_chars)
            request characters (get_console_chars)

            do{
                  myEnv = receive_message();
                  if (getCharsEnv)
                        copy contents to msgData
                  else if updateClockEnv
                        update clock using send_console_chars
                  else if crtAcknowledgeEnv
                        do nothing
            }
            while(msgData is empty);

            if(parseString(msgData, &arrInput, " ", 3) == 0)
                  empty string, ignore, display next prompt

            release messageEnvelope

            switch(toLower(arrInput[0])){
                  case "cmd1":
                        if (valid applicable params)
                              //execute fn
```

```
                        else
                              //display error message (invalid param)
                        break;
                  default:
                        //display error message (invalid command)
            }
      }while(true);
}
```

Commands

Send Message Envelopes: `s <cr>`

Implementation

This command will first request an envelope from the messaging class. Upon receiving a free envelope, a message will be sent to User_Process_A. The function will then return since no response is returned.

Pseudocode

```
myEnv = RTX.request_msg_env();

RTX.send_message(User_Process_A, myEnv);
```

Process Status: `ps <cr>`

Implementation

This command will result in a summary of the processes currently executing on the RTX and their current statuses. It will be formatted using standard C/C++ escape characters to form a table containing the following columns: { process_id, priority , process_status(verbose) }.

Pseudocode

```
for(i=0 to PCB_COUNT)
{
      strPcb = formatted string of PCB info
      RTX.send_console_chars(envPcnInfo);
}
```

Set Wall Clock: `c hh:mm:ss <cr>`

Implementation

The set wall clock command will update the RTX's time struct with the specified time. The param1 string will be parsed by ":" characters, and checked that they are both integers and within the acceptable value range.. If the parameter is valid the time struct will be updated.

Pseudocode

```
int val[3] = {0,0,0}
```

```
if parseString(arrInput[1], &val, ":", 3)
{
      //validate arrInput[0 to 2] are within valid range - return error
if not valid

      //set RTX.wallTime (hours/minutes/seconds)
}
```

Display Wall Clock: `cd <cr>`

Implementation

The display wall clock command sends an enable clock message to the displayClock process. This will bring the process out of its blocked_on_envelope state and begin polling and displaying the time.

Pseudocode

```
myMsg = request_msg_env();
set myMsg type to enable clock message
send_message(displayClock, myMsg);
```

Hide Wall Clock: `ct <cr>`

Implementation

The hide wall clock command sends a disable clock message to the displayClock process. The process will then go into the blocked_on_receive state until an enable clock message is sent.

Pseudocode

```
myMsg = request_msg_env();
set myMsg type to disable clock message
send_message(displayClock, myMsg);
```

Display Contents of Trace Buffers: `b <cr>`

Implementation

This command will result in a formatted table containing the message trace buffer information being displayed to the user. The formatting and sorting will be performed within the messaging class. The Tx buffer will be displayed after the Rx buffer, and messages will be displayed in descending chronological order.

Pseudocode

```
myEnv = request_msg_env();
get_console_chars(myEnv);
//validate correct returned envelope
send_console_chars(myEnv);
```

Terminate: `t <cr>`

Implementation

The terminate command will have the same functionality as receiving a terminate (interrupt) signal. As a result, a signal will be simulated but calling the signal handler with a SIGINT signal parameter.

Pseudocode

```
signalHandler(SIGINT);
```

Change Process Priority: `n new_priority process_id <cr>`

Implementation

The change priority command will change the priority of the designated process within the RTX. The specified process_id will be verified against those in the PCB list. The new_priority will be verified to assure that it is valid. If either of these parameters are invalid, an error will be returned to the user. Otherwise the CCI will invoke the PCB's setPriority function.

Pseudocode

```
isValid = (process_id < PCB_COUNT) and (new_priority is 0 to 3)
if (isValid)
      RTX.PCB[process_id].setPrioirty(new_priority);
```

Implementation Strategies

Code Standards

A coding style guide document has been compiled to be used during the implementation of this project. It outlines naming conventions, comment styles and other standards which contribute towards a more readable and consistent final product.

Development Environment

A standard development environment virtual machine image based on Linux Mint will be created. All code development will be performed in this environment. This ensures that less platform dependant-issues arise throughout the duration of the development cycle than if code development were being done on different platforms by each member of the group.

The development image will feature the g++ compiler. Since this is the compiler available on the ECE Linux machines, this will allow code to be compiled by the same compiler throughout each step of the development cycle.

Version Control

Git version control software will be used for distributed version control of the project code. Private git repository server space will be purchased from Github for the duration of the development cycle so that project source code can be easily and securely distributed.

The development strategy will be fairly simple, with one main branch which will always

be kept in a state that will compile, and many experimental/development branches will branch off from it. Only once code that resides on a development branch compiles and has been tested for compatibility with the rest of the source will the development branch be merged in with the main branch.

Debugging

Strategies

The main strategy is to simultaneously build up the modules, accomplishing various stages of functionalities as the development progresses. First each member must complete the header files for their modules and upload it to the git. Next the skeleton for the functions must be completed and uploaded. These skeletons will not have the correct functionality but will return the correct object. The next stage involves filling in the correct code and testing along the way, so that compilable and functional code may be located on the git repository.

Finally, a week before the due date, group testing will begin. Each person will perform the testing for their respective modules before the group testing begins. This means the entire last week may be used to verify the interactions of the modules. Also, this final testing will occur on the ecelinux servers to ascertain that the RTX will be full functional in the demo environment.

Ensure Function

An ensure function will be implemented to allow for error checking and handling. There will be two basic types of error checking, fatal and non-fatal. Fatal errors will be errors such as a memory allocation failure for a PCB, while a non-fatal error could be an invalid priority specified for the change_priority primitive. The function should always be called using the '__LINE__' and '__FILE__' pre-processor variables in their respective fields.

The ensure function will be passed a condition (i.e. ptr != NULL), which will be evaluated. If the condition is true then the function will immediately return true. Should the condition be false, then the function will print to the screen the provided error message as well as the file name and line number. If the error is fatal the terminate function will be called, and if not the function will return false to allow for the calling function to perform error handling.

This function uses the c++ print_f function rather than the CRT since this function may be called from the CRT in the event of an error and as such this could cause an infinite loop. The print_f function call will be disabled for demonstration purposes since all errors should be removed by this point (note: #if will be used so that the print_f call does not exist in the final presentation compiled code).

Implementation

```
bool ensure( bool condition, string message, string fileName, int
lineNum, bool isFatal )
{
      if(!condition)
      {
#if DEBUG == 1
            print error message using print_f
```

```
#endif
            if (isFatal)
                RTX.Terminate();
      }
      return condition;
}
```

Planned Distribution of Work

To distribute the work, the RTX project was divided into seven main modules which were then collaboratively ranked in terms of anticipated difficulty and complexity (from 1 to 7, 7 being the most intricate and challenging). Then the modules were assigned to the group members so each person's "workload value" averaged to approximately the same value, yet slightly larger work sums were allocated to the stronger developers on the team. The chosen modules, levels of difficulties, and assigned experts are outlined below.

Expert	**Module**	**Anticipated Difficulty Level**
Brock Kopp	- Initialization - Signal Handler - CCI	3 5 2
Karl Price	- Scheduling (process and processor management) - Basics: Data Structures	7 1
Angelica Ruszkowski	- Console I/O - KB and CRT i_processes - Basics: Data Structures	6 2 1
Eric Vandenberg	- Messaging Services - Timing Services	4 2

In terms of testing, each team member is responsible for designing test cases for their respective modules, while group members will work together to test interprocess collusion.

Timeline

To assist in the timely, successful completion of this project, the team has decided upon a timeline that will outline the estimated completion dates for various benchmark development stages. Deliverable due dates are included.

Saturday, October 16; 10:00am	Draft 1 of SDD due: Main information is included. Ensure that all requirements have been met. Identify large missing links between the various modules.

Monday, October 18; 4:30pm	Draft 2 of SDD due: Major gaps have been eliminated and documented. Remaining minor discrepancies are then analyzed.
Wednesday, October 20; 6:00pm	Final Draft of SDD due: Document will be printed and bound.
Thursday, October 21; 11:59pm/ Friday, October 22; 11:30am	Soft/Hard copies of the SDD are submitted
Monday, November 1	Version Control (git) purchased
Tuesday, November 2; 4:30pm	Header files for each module/class must be uploaded to the git repository
Saturday, November 6; 11:59pm	Skeleton functions are written; code is not yet operational, but all function outlines are uploaded into the repository and return the correct values (so that modules may begin calling other functions)
Saturday, November 13; 11:59pm	Code is completed and compilable; all code must be uploaded to the repository
Sunday, November 14	Testing of individual modules begins
Saturday, November 20	Individual module testing complete; group testing of the RTX as one unit begins
Saturday, November 27	All code is complete and fully functional. Compilable version is in git repository
Sunday, November 28; 11:59pm	Source Code Due
Tuesday, November 3	Demonstrations Begin