ECE 354: Part 3

Group: ECE.354.S11-G031
Members: Ben Ridder (brridder), Casey Banner (cccbanne), David Janssen (dajjanss)

This document is concerned with the high level implementation of the specified primitives and processes. As such, most of the low level back end is omitted for simplicity. Any kernel structures required to understand the primitive are described when the primitive or process is discussed.

Design of Delayed Send

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

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

Each time the timer i-process runs, it checks the head of the delayed message queue to see if a message should be sent. If so, the message is dequeued and forwarded to the correct process and the head is checked again. The pseudo code for the delayed send is in Listing 1. The pseudo code and implementation information for the timer forwarding is shown below in the timer i-process section.

Listing 1: Pseudo code for delayed send

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

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

        return RTX_SUCCESS
```

The KCD Process

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

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

When the message type is a command registration, the PID of the registering process is saved along with the command itself. The command character is passed in the message body. The pseudo code for the KCD can been seen in Listing 2.

Listing 2: Pseudo code for implementing the KCD Processes

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

The CRT Process

The `process_crt_display()` blocks until a message is received. If type is correct, then the message data is sent to the UART and the message is released. Pseudo code for the process can be found in Listing 3.

Listing 3: Pseudo code for the CRT process

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

The UART I-Process

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

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

If it is a write, a character is written to serial port and transmit ready interrupt is masked. If a carriage return was written, then a newline character is also queued to be written. A special case is considered for when the message type is specified to not require any newlines. Pseudo code for this process is below as Listing 4.

Listing 4: Pseudo code for the UART I-Process

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

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

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

        echo the in character by triggering a transmit interrupt

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

    release processor
```

The Timer I-Process

The `i_process_timer()` process runs immediately in response to timer interrupts. Upon receiving an interrupt, this process performs two tasks. First, it increments its internal time counter. Next, the delayed messages queue is checked to see if there are any messages to send out. All messages that were scheduled to be sent at the current time or sooner are sent. At this point, this process will relinquish control to the processor. See Listing 5 for the pseudo code for this process.

Listing 5: Pseudo code for the UART I-Process

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

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

The Wall Clock Process

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

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

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

Pseudo code for the wall clock process can be seen at Listing 6.

Listing 6: Pseudo code for the wall clock process

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

            if (clock = 86400):
                reset clock

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

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

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

Set Priority Command Process

The `process_set_priority_command()` is a process that allows the user to change the priority level of any process through a command. This process accepts data in the form `%C process_id priority_level`. When the process is initialized it registers the `%C` command. It then waits to receive a message. Upon receiving a message it skips the `%C` characters and begins parsing the data. While parsing, the process id of the process whose priority is to be changed and the new priority level are extracted. Checks are made to ensure that proper process ids and priority values were provided and that nothing else was entered after the priority value. A kernel function call of `set_process_priority(...)` is then made using the parsed process id and priority value. This function call is responsible for the actual changing of the priority level. The memory block and the processor are then released. Pseudo code for this process can be seen at Listing 7.

Listing 7: Pseudo code for the set priority command process

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

    while (1):
        receive a message

        skip %C in message

        parse process id
        validate process id

        parse priority level
        validate priority level
```

```
        validate line endings

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

        release memory block;
        release processor;
```