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 works using a separate queue that holds all delayed messages regardless of which process they belong to. Inside the queue they are ordered from smallest to largest, using the combined values of the time it was first put on the queue plus the delay. This function takes three parameters: the id of a process that is being sent the message, a message envelope containing the message, and the delay time in milliseconds. A system call is made causing a switch to kernel mode. After the switch, 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 messages queue where it stays until it is forwarded by the timer into the corresponding message queue. This implies that the message would be at the head of the queue and the timer has reached the correct value. 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 keyboard command decoder is implemented as a process located in the `system_processes.c`. The `process_kcd()` idles until a message is received, as it blocks on receiving messages and does not need to do anything until a message is passed to it. Once a message is received 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 type is key input and the message body matches to a previously registered command, then the message data is copied into a new message envelope and sent to that process. The original message is forwarded to the CRT. If the message body is not previously registered, it is only forwarded to the CRT. When the type is a command registration, it stores the command by storing the PID of the registering process and the character of command located 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(){
    initialize local variables;

    while(1){
        receive a message;
        if (message type = key input){
            if (message body = registered command){
                copy message and send to process;
            }

            send message to CRT;

        } else if (message type = command registration){
            register a new command;
        } else {
            release message;
        }
    }
}
```

The CRT Process

The CRT process is also implemented in `system_process.c`. Much like the KCD, the `process_crt_display()` blocks until a message is received. Once a message has been received, the type is checked. If the type is not an output or a key input then message is released and the process releases itself to the processor. If type is correct, then the message data is sent to the UART and the process and message are released. Pseudo code for the process can be found in Listing 3.

Listing 3: Pseudo code for the CRT process

```
void process_crt_display(){
    initialize local variables;

    while(1){
        receive a message;
        if (message type = output || message type = key input){
                send message data to uart;
        }

        release message;
        release process;
    }
}
```

The UART I-Process

The UART interrupt process can be found in `system_process.c`. This process runs via an interrupt. After the interrupt has fired and the process is switched to this process, 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 TX ready is masked on the serial status register. A check is made to ensure that if carriage return is being written then a newline character is also written. A special case is considered for when the message type is specified to not require any newlines. Then the process is released. Pseudo code is below as Listing 4

Listing 4: Pseudo code for the UART I-Process

```
void i_process_uart(){
    initialize local variables;

    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 tx interrupt

            enable interrupts;
        } else if (state = write) {
            write out character;
            mask tx ready;
            if (character = carriage return AND requires a new line){
                print newline;
            }
        }

        release processor;
    }
}
```

The Timer I-Process

The `i_process_timer()` primitive is a process that is run immediately in response to timer interrupts. Upon receiving an interrupt, this process performs several tasks. The first thing it does is increment a time counter. Next, the delayed messages queue is checked to see if there are any messages to send out. It will keep sending out messages, until there are no longer any more to send. At this point, this process will relinquish control to the processor. See Listing 5 for the pseudo code implementation.

Listing 5: Pseudo code for the UART I-Process

```
void i_process_timer(){
    initialize local variables;

    while(1){
        release processor;
        increment timer;

        if(delayed messages queue != NULL){
            while (there are messages to be sent){
                remove message from delay messages queue;
                append message to appropriate message queue;
            }
        }
    }
}
```

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 the user and is updated every second. Clock time is printed until a second command to terminate the display is received. The process checks to see if it has received a message from itself, if has not a message is sent with a delay of 1 second. Then the process blocked until it receives a message and verifies that the sender was itself. The clock is incremented and if it has reached 86400 seconds (24 hours) it is reset to 0.

A check of a display flag is made. If the flag is true then the integers are converted to a string and a message is sent to the CRT to be outputted. If the message was from the KCD the message body is checked to see if it was a %WS or %WT command. If it was a %WS command the message body is parsed and the initial timer conditions are converted from characters to integers and the display flag is turned on. When parsing the string error checking is done to make sure the initial value sent by the user is valid. If the %WT command is received the output string is overwritten with null characters and the display flag is turned off. Upon completion the message is released. 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(){
    initialize local variables;

    while (1){
        if (process has not sent a message to itself) {
            send a delayed message to itself with delay of 1s;
        }

        receive message;
        if (sender_id = wall clock pid) {
            increment clock;

            if (clock = 86400) {
                reset clock;
            }

            if (clock display = true) {
                convert clock to hours, minutes and seconds;
                convert int to string;
                send message to CRT;
            }
```

```
        } else if (sender_id = KCD pid) {
            if (%WS) {
                parse string and validate input;
                store hours, minutes and seconds as ints;
                set clock display = true;
            } else if (%WT) {
                set out string = null;
                set clock display = false;
            }

            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 directly from the UART. 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(){
    initalize local variables;

    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;
    }
```

```
}
```