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. 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 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 with the messages that are to be sent the soonest at the front of the queue. This function takes three parameters: a process id of the process that is being sent a message, a message envelope containing the message, and the delay value in milliseconds. A system is call made causing a switch to kernel mode. After the switch the message 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. 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 processs_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 in `system_processes.c`. The `process_kcd()` blocks on receiving messages. 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. If the type is key input and the message body matches to a previously registered command then the message data is sent to that process and the original message is sent to the CRT. If the message body is not previously registered it is not sent anywhere and again the original message is sent to the CRT. If the type is command registration it then registers a command for the sender pid where the command is is 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 output or key input then the process and message are released. 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 with interrupts. After the interrupt has fired the UART status register is checked to determine whether a read of a write is to occur. When reading in data all characters are appended to a buffer until a carriage return is read. At this point a newline and null value are appended and a message is created. A check is made to determine the first value in the string buffer. If it is a percent (%) the message is sent to the KCD, if it is an exclamation mark (!) the message is sent to the CRT and the string buffer is sent to the UART debug decoder, otherwise the message is sent to the CRT. After this the interrupts are re-enabled. If it is a write state, a character is written to serial port and tx ready is masked. A check is made to ensure that if carriage return is being written then a newline character is also writte. Then the process is released. We made a UART I-Process. Pseudo code here at 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 characater;
            append character to buffer;
            if (character = carriage return){
                append newline to buffer;
                append null to buffer;

                create a new message;
                set message data to buffer;

                if(buffer[0] = '%'){
                    send message to KCD;
                } else if (buffer[0] = '!'){
                    send message to CRT;
                    send buffer to uart debug decoder;
                } else {
                    send message to CRT;
                }
            }
                enable interrupts;
        } else if (state = write){
            write out character;
            mask tx ready;
            if (character = carriage return){
                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 the process will release itself and then increment the timer counter. The process also checks to see if the delayed send queue is empty. If the queue is not empty another check is made to determine if any messages in the queue need to be sent. If enough time has passed then the messages is appended to the appropriate message queue. Pseudo code is shown at Listing 5.

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 a time on UART1. This clock is given
an initial time and updates it every second. This 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 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 hasn't 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;
```

```
         }
      }
}
```