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 waits for an interrupt to occur. After the interrupt has fired the state of the UART 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. 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;
        }

        release processor;
    }
}
```

The Timer I-Process

We made a Timer I-Process. Pseudo code here at Listing 5.

Listing 5: Pseudo code for the UART I-Process

```
UART_Process();
```

The Wall Clock Process

There is a clock on the wall. Pseudo code here at Listing 6.

Listing 6: Pseudo code for the wall clock process

```
wall_clock_Process();
```