**ECE 354: Real Time Operating System**

**Project 3 and 4**

**Documentation**

**Delayed Send Primitive:**

This non-blocking primitive is responsible for sending the given message to the given process after the given amount of time has elapsed. The delay is specified in milliseconds. If the calling process is not pre-empted, it continues to execute after calling this primitive. The design of this primitive is described in the pseudocode that follows.

int delayed\_send (int process\_ID, void \*MessageEnvelope, int delay)

perform error checking

initialize TimeAdded\_delay as Counter2 + delay

if TimeAdded\_delay > Counter2 then

enqueue onto delayed send queue

end if

In the above code Counter2 is a global timer variable maintained by the OS and incremented by the Timer ISR. All message are stored in a priority queue where the TimerAdded\_delay is the priority of the node.

**CRT Display Process:**

The CRT display process is responsible for display text to the the screen using UART1. The CRT display process is able to receive a string of characters from any process. It then iterates through each character in the string and send each character to the UART I-Process for printing. The following pseudocode describes the design of the CRT display process.

CRT Display Process:

loop forever

msg\_envl <- receive message

initialize buffer as char array

loop char from message does not equal '\0'

set buffer at index equal char from message

increment buffer

end loop

release memory block msg\_envl

loop buffer at index does not equal '\0'

request memory block to use as message envelope

place character from buffer in message

send message to UART I-Process

enable transmit interrupts

increment buffer

end loop

end loop

**KCD Process:**

The KCD process serves 2 purposes. First, any process can send it a command identifier to register. The KCD then becomes responsible for forwarding any input that contains the command identifier to the process. Second, the KCD process receives keyboard input characers from the UART I-Process and send them to the CRT display process to be echoed back on screen. The KCD uses a linked list to maintain a list of command identifier and the IDs the command is registered to. The desing of the KCD process is described by the following pseudocode.

KCD Process:

initialize linked list to store command registrations

initialize buffer to store input strings until the enter key is pressed

loop forever

e <- receive message

if sender of message does not equals UART I-Process then

add string in message envelope to command registration list

else

initialize input\_char with character from message envelope

request memory block to use as message envelope

if input\_char equals 'CR' or 'LF' then

place CR + LF + '\0' in message envelope

else

place input\_char + '\0' in message envelope

end if

send message to CRT dispaly process

if input\_char equals 'CR' or 'LF' then

search command registration list for match

if match is found then

request memory block to use as message envelope

place buffer in message envelope

send message to registrant process

end if

else if input\_char equal 'BKSP' or 'DEL' then

decrement buffer index

else

set bufer at index to input\_char

end if

release memory block e

end loop

**UART I-Process:**

The UART I-Process is the process that gets loaded when an UART interrupt is generated. It is responsible for both transmitting and receiving characters. The following pseudocode describes the desgin of the UART I-Process.

UART I-Process:

loop forever

initialize temp with SERIAL1\_USR register's value

if temp & 1 equals true then

initialize in\_char with SERIAL1\_RD's value

check for hotkeys

request memory block to use as message envelope

place character in message envelope

send message to KCD process

end if

if temp & 4 equals true then

e <- receive message

if e does not equal NULL then

set SERIAL1\_WD equal to character in message

else

disable transmit interrupts

end if

end if

release processor

end loop

**Timer I-Process:**

In order to optimize our OS we decided to merge the timer I-Process and ISR. This was done because the overhead involved in performing a context switch to a timer I-Process was resulting in significant lag and leading to missed timer interrupts. The timer ISR was thus design to be very efficient in order to accomodate the 1ms interrupt interval. The follow pseudocode describes the design of the timer ISR.

Timer ISR:

increment global timer counter Counter2

acknowledge interrupt by setting TIMER0\_TER to 2

if delayed send queue is not empty then

if expiration value of head node is less than Counter2 then

initialize process\_ID to node's process\_ID

initialize sender\_ID to node's sender\_ID

dequeue from delayed send queue

set message envelope sender\_process\_ID to sender\_ID

set message envelope dest\_process\_ID to process\_ID

enqueue onto destination process's mailbox queue

unblock destination process if it blocked

end if

**Wall Clock Process:**

Once the wall clock process registers the %W command with the KCD process, it's job is to display the user supplied time and update it every second. The follow pseuodocode described the design of the wall clock process.

Wall Clock Process:

register this process as the hanlder for the '%W' command

loop forever

initialize sender\_ID to -1

envelope <- receive message, storing the sender ID in sender\_ID

if sender\_ID equals KCD process's ID then

extract string from message

if command equals '%WS' then

parse and error check time string

set clock active flag to true

else if command equals '%WT' then

set clock active flag to false

end if

end if

release memory block envelope

if clock active equals true then

initialize counter to secs + mins \* 60 + hours \* 3600

end if

request memory block to use as message envelope

1 sec delayed send to self

increment counter

convert counter to hours, minutes and seconds

populate string with hours, minutes and seconds

request memory block to use as message envelope

cope time string into message envelope

send message to CRT display process

end loop

**Test Process Documentation**

The 6 test process are used to test 3 test cases. Test processes 1 through 5 are used to test various primitive calls such as send/receive message and request memory block while test process 6 is used to print the results to the janusROM terminal.

Since processes are not allowed to terminate in the OS, a call to receive message is used to block each process after it has completed its necessary functions to simulate process termination.

**Test Case 1: Testing send message, receive message and request memory block**

Process 1

request memory block mem\_block to use as message envelope

place 'a' in message part of message envelope

send message to process 1

block process by calling receive message

loop forever

release processor

end loop

Process 2

receive message

if sender of message is 1 and message contains 'a' then

flag test1 as ok

end if

block process by calling receive message

loop forever

release processor

end loop

**Test Case 2: Testing delayed send, getting and setting process priority and preemption**

Process 3

set process 3's priority to 1

if process 3's priority is 1 then

request memory block to use as message envelope

place 'b' in message part of message envelope

5000 msec delayed send to process 3

block process for 5 seconds by calling receive message

flag process 3 as having come out of block

if message sender is 3 and message contains 'b' then

set process 3's priority back to 3

if process 3's priority is 3 then

flag test2 as ok

end if

end if

end if

block proces by calling receive message

loop forever

release processor

end loop

**Test Case 3: Testing proper stack management through modifications of variables and function calls**

Process 4

initialize i to 0

initialize n to 5

set process 4's priority to 2

if process 4's priority is 2 then

loop i < n

i equals add\_one(i)

end loop

end if

if i equals n

flag test3 as ok

end if

block process by calling receive message

loop forever

release processor

end loop

add\_one(num)

initialize result to num + 1

release processor

return num

Process 5

loop process 3 is not done

release processor

end loop

request memory block to use as message envelope

send message to process 6

loop forever

release processor

end loop

**Print Results:**

Process 6

receive message

if test1 is ok then

print test 1 ok

else

print test 1 fail

if test2 is ok then

print test 2 ok

else

print test 2 fail

if test3 is ok then

print test 3 ok

else

print test 3 fail

print number of tests ok / number of total tests

print number of tests fail / number of total tests

print end