Lab 5

Name: Anirudh Pal (pal5@purdue.edu)

Section: Wednesday @ 3:30 PM

Instructor: Nicolas Ogg

Part 3. Blocking Message Send

Testing

Blocking Sender

Description

In this test case I test if sendblk() is blocking when it is suppose to. I do this I use three processes. The first to be run is a sluggish receiver that sleeps and then does mutiple receive() function calls. After that I start two senders that use sendblk(). The first sends '1213' and the second sends '2324'. The sender prints the action, calls sendblk() and then prints when it is done.

Expectation

The receiver will start and go to sleep. Then the first sender will complete the sendblk() and then finish. The second sender will start sendblk() and get blocked and added to the queue. Then the receiver will wake up and get the first message and stage the second message. This will allow the second sender to dequeue and run till completion. Then the receiver gets the second message.

Code in main.c

```
resume(create(recvslug, 2042, 10, "recvslug", 0));
resume(create(senddone, 2042, 10, "senddone", 2, 3, 1213));
resume(create(senddone, 2042, 10, "senddone", 2, 3, 2324));
```

Output with Annotations

```
(Pal, Anirudh)
pal5                                           <-- Receiver is Sleeping
PID: 4 Sender (SENDBLK) -> 1213 to PID: 3      <-- First Sender Sends
PID: 4 Sender (SENDBLK) Done Sending           <-- First Sender Completes
PID: 5 Sender (SENDBLK) -> 2324 to PID: 3      <-- Second Sender Sends and Blocks

                                               <-- Stuck till sleep() Completes
```

```
PID: 5 Sender (SENDBLK) Done Sending           <-- It is allowed to Finish
PID: 3 Receiver (RECEIVE) <- 1213              <-- Receiver Prints First Message
and Executes Second receive()
PID: 3 Receiver (RECEIVE) <- 2324              <-- Receiver Prints Second Message
```

Conclusion

Results are consistent with our expectations.

Multiple Sender

Description

This is similar to the previous test case. The only difference is that there are multiple senders.

Expectation

The messages should arrive in the same order as they were sent after the receiver wakes up.

Code in main.c

```
resume(create(recvslug, 2042, 10, "recvslug", 0));
resume(create(senddone, 2042, 10, "senddone", 2, 3, 1213));
resume(create(senddone, 2042, 10, "senddone", 2, 3, 2324));
resume(create(senddone, 2042, 10, "senddone", 2, 3, 3435));
resume(create(senddone, 2042, 10, "senddone", 2, 3, 4546));
resume(create(senddone, 2042, 10, "senddone", 2, 3, 5657));
```

Output with Anotations

```
(Pal, Anirudh)
pal5
PID: 4 Sender (SENDBLK) -> 1213 to PID: 3
PID: 4 Sender (SENDBLK) Done Sending
PID: 5 Sender (SENDBLK) -> 2324 to PID: 3
PID: 6 Sender (SENDBLK) -> 3435 to PID: 3
PID: 7 Sender (SENDBLK) -> 4546 to PID: 3
PID: 8 Sender (SENDBLK) -> 5657 to PID: 3
PID: 5 Sender (SENDBLK) Done Sending
PID: 3 Receiver (RECEIVE) <- 1213           <-- Message 1
PID: 6 Sender (SENDBLK) Done Sending
PID: 3 Receiver (RECEIVE) <- 2324           <-- Message 2
PID: 7 Sender (SENDBLK) Done Sending
PID: 3 Receiver (RECEIVE) <- 3435           <-- Message 3
PID: 8 Sender (SENDBLK) Done Sending
PID: 3 Receiver (RECEIVE) <- 4546           <-- Message 4
PID: 3 Receiver (RECEIVE) <- 5657           <-- Message 5
```

Conclusion

Results are consistent with our expectations.

Receiver Termination

Problem

When the receiver terminates with senders still in its queue, these sender will be blocked potentially forever. Also if a second process gets the same PID, it will now have zombie senders that will give it false messages because they are still in the queue.

Solution

In the kill() function we should dequeue all senders and make them ready. This will allow them to finish execution and will also leave the queue empty for any process that might get the same PID. This is assuming kill() is run upon process termination.

Part 4. Asynchronous IPC with Callback Function

Testing

Interrupted Callback Receiver

Description

In this test case I test to see if the callback feature works. I do this by using three processes. The first to be run is a callback receiver that registers a callback function. The callback function calls receive() and populates a var called 'mbuf'. The receiver itself prints 'mbuf' in an infinite loop. After that I start two senders that use send(). The first sends '1213' and the second sends '2324'. The sender prints the action, calls send() and then prints when it is done.

Expectation

The receiver will start and register the callback and start printing 'mbuf'. 'mbuf' is set as 0 by default. Then the first sender will complete the send() and finish. Then the second sender will complete the send() and finish but this time it will fail to change the message as the receiver has not received the processor to process the first message. After the second sender has finished, the receiver will get the processor through an interrupt and execute the callback. Once the callback has finished, it will continue the loop with the message that it received. This message will be the first message '1213'.

Code in main.c

```
resume(create(recvcb, 2042, 10, "recvcb", 0));
resume(create(senddoneog, 2042, 10, "senddoneog", 2, 3, 1213));
resume(create(senddoneog, 2042, 10, "senddoneog", 2, 3, 2324));
```

Output with Annotations

```
PID: 3 Receiver (CALLBACK) <- 0                  <-- Loop in Receiver
PID: 3 Receiver (CALLBACK) <- 0
PID: 4 Sender (SEND) -> 1213 to PID: 3           <-- First Sender Starts
PID: 4 Sender (SEND) Done Sending                <-- First Sender Finishes
PID: 5 Sender (SEND) -> 2324 to PID: 3           <-- Second Sender Starts
PID: 5 Sender (SEND) Done Sending                <-- Second Sender Finishes but has
not Changed the Message
PID: 3 Receiver CallBack (CALLBACK) <- 1213      <-- Receiver gets CPU and executes
Callback
PID: 3 Receiver (CALLBACK) <- 1213               <-- Loop in Receiver
PID: 3 Receiver (CALLBACK) <- 1213
PID: 3 Receiver (CALLBACK) <- 1213
PID: 3 Receiver (CALLBACK) <- 1213
.
.
.
```

Conclusion

Results are consistent with our expectations.

Sleeping Callback Receiver

Description

Same as the previous test but instead of an interrupt we have a wake from sleep() in receiver.

Expectation

Same as the previous test but instead of an interrupt we have a wake from sleep() in receiver and then run the callback.

Code in main.c

```
resume(create(recvcbslp, 2042, 10, "recvcbslp", 0));
resume(create(senddoneog, 2042, 10, "senddoneog", 2, 3, 1213));
resume(create(senddoneog, 2042, 10, "senddoneog", 2, 3, 2324));
```

Output with Annotations

```
(Pal, Anirudh)
pal5                                             <-- Receiver Registers Callback and
Sleeps
PID: 4 Sender (SEND) -> 1213 to PID: 3           <-- Same as Previous
```

```
PID: 4 Sender (SEND) Done Sending
PID: 5 Sender (SEND) -> 2324 to PID: 3
PID: 5 Sender (SEND) Done Sending

                                                <-- Receiver Wakes Up

PID: 3 Receiver CallBack (CALLBACK) <- 1213     <-- Same as Previous
PID: 3 Receiver (CALLBACK) <- 1213
PID: 3 Receiver (CALLBACK) <- 1213
PID: 3 Receiver (CALLBACK) <- 1213
PID: 3 Receiver (CALLBACK) <- 1213
.
.
.
```

Conclusion

Results are consistent with our expectations.

Isolation / Protection

XINU Isolation and Protection

In XINU isolation and protection requires that the callback runs in the context of the registering process (receiver) and is in user mode (interrupts enabled).

Design

A callback handler has to be run to execute the callback if there is a message with interrupts enabled. This has to be run after 'sti' in 'clkdisp.S' and 'restore(mask)' in 'sleep.c'.

Implementation

cbhandler.c

```
// Import Libs
#include <xinu.h>

// Define Handler
void cbhandler() {
        // Get Process
        struct procent* prptr = &proctab[currpid];

        // Check if Message & Callback
        if(prptr->prhasmsg && prptr->prhascb) {
                // Call Callback
                prptr->fptr();
        }
}
```

clkdisp.S:18

```
sti                     # Restore interrupt status

call    cbhandler       # Anirudh Pal - Lab 5 for Callback
```

sleep.c:48

```
restore(mask);

// Anirudh Pal - Lab 5
// Call Callback Handler
cbhandler();
```

Bonus

The files have been placed in *system/*.