

3. Blocking send() IPC

3.1 Kernel modifications: bsend()

See *bsend.c* for additions, and *process.h* for modifications.

Increase NQENT and explain:

The NQENT is increased by NPROC - 1 send processes. This is assuming all processes can queue to a receiver process, which is NPROC - 1.

3.2 Kernel mods: receive()

3.3 Testing

Test scenarios: I added debug statements and made sure the messages have been sent, queued, received, and processed correctly.

1. Zero sender processes, one receiver process

Result: receiver process is blocking.

```
pid32 receiverpid = create(receive, 8192, 20, "receive", 0);
resume(receiverpid);
```

2. One sender process, one receiver process

Result: sender process is not queued, and receiver processes the only message.

```
pid32 receiverpid = create(receive, 8192, 20, "receive", 0); // Receiver
process
pid32 senderpid1 = create(bsend, 8192, 20, "send", 2, receiverpid, "hi");

resume(senderpid1); // Run sender process first iteration
resume(receiverpid); // Run receiver process
```

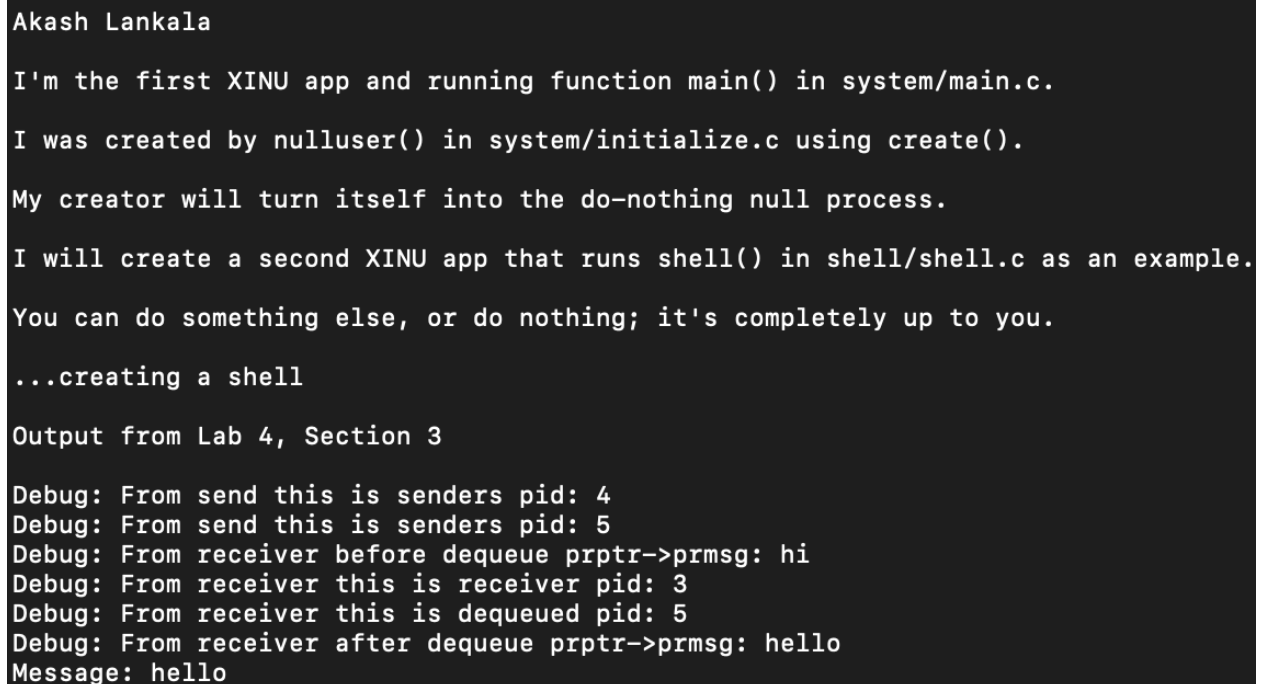
3. Two sender processes, one receiver process

Result: subsequent sender is queued and dequeued by receiver process, and previous message is returned by receiver.

```
pid32 receiverpid = create(receive, 8192, 20, "receive", 0); // Receiver
process
pid32 senderpid1 = create(bsend, 8192, 20, "send", 2, receiverpid, "hi"); //
Sender process first iteration
```

```
pid32 senderpid2 = create(bsend, 8192, 20, "send", 2, receiverpid, "hello");  
// Sender process second iteration  
  
resume(senderpid1); // Run sender process first iteration  
resume(senderpid2); // Run sender process second iteration  
resume(receiverpid); // Run receiver process
```

The output of the last test case is shown in the screen capture is shown below.



```
Akash Lankala  
I'm the first XINU app and running function main() in system/main.c.  
I was created by nulluser() in system/initialize.c using create().  
My creator will turn itself into the do-nothing null process.  
I will create a second XINU app that runs shell() in shell/shell.c as an example.  
You can do something else, or do nothing; it's completely up to you.  
...creating a shell  
Output from Lab 4, Section 3  
Debug: From send this is senders pid: 4  
Debug: From send this is senders pid: 5  
Debug: From receiver before dequeue prptr->prmsg: hi  
Debug: From receiver this is receiver pid: 3  
Debug: From receiver this is dequeued pid: 5  
Debug: From receiver after dequeue prptr->prmsg: hello  
Message: hello
```

4. Hijacking a process by modifying its run-time stack

4.1 Basic idea

4.2 Overt attack

See *attackerA.c*, *victimA.c*, and *hellomalware.c* for additions.

Return address is base pointer address + 1. The way I have calculated is using the base pointer address as a reference, and looping through 3 nested stack frames.

Example below for taking over the first process. Tested with others too.

```
Spawn victim's processes for Lab 4, 4.2 & 4.3

before funcA: 6 5
before funcA: 7 5
before funcA: 8 5
Debug: output for Section 4.2
Successful takeover of victim: 6
after funcA: 7 5
after funcA: 8 5
```

4.3 Stealth attack

The value from 5 to 9 has been changed at variable address location 0x14 by looking at the stack trace and finding the variable address location. Third, the return address was overwritten by *quietmalware()*'s function address. And once *quietmalware()* is finished, it goes back and completes the rest of the instructions normally.

The following screen capture is the output for the third process (this is actually the combined output for section 4.2 as well, taking over the first process).

```
before funcA: 6 5
before funcA: 7 5
before funcA: 8 5
Successful takeover of victim: 6
after funcA: 7 5
Output for Section 4.3:
Debug: currrpid from qmalware: 8
Debug: Print var value from mem addr before change: 5
Debug: Print var value from mem addr after change: 9
It's supposed to be quiet takeover, but added for debugging. Successful takeover of victim: 8
after funcA: 8 9
```

Bonus problem

If the receiver is terminating without processing all the queued messages from FIFO, use the following steps

1. Set a new proc table field *bool18 prreceiverterminated*.
2. After *receiver()* processed the first message from the FIFO queue, the receiver will check if there are more queued entries, then receiver will set the above flag using the following syntax. This can go in *receive.c*

```
if (nonempty(prptr->prblockedsenders)) {  
    prptr->prreceiverterminated = TRUE;  
}
```

3. *brend()* can check the flag *prptr->prreceiverterminated = TRUE*, then dequeue the queue using the following syntax and send the SYSERR message.

Sample solution that would go in *brend.c*.

```
while (nonempty(prptr->prblockedsenders)) {  
    ready(dequeue(prptr->prblockedsenders));  
}  
restore(mask);  
return SYSERR;
```

The data structure is the FIFO queue: *prptr->prblockedsenders*, and *prptr->prreceiverterminated*.

The *restore(mask)* is enabling interrupts.