Aaron Althoff

CS 354

Lab 5 Answers

3. To implement this function I first verified the validity of the pid that is passed as a parameter to the function, then I checked if the process already had a message in its buffer. If it didn't then I called the normal send function. To test that the function worked correctly I wrote 2 files, testsendblk.c which defined a function that called sendblk on a pid passed in as a parameter and sent it's pid, and testreceive.c which defined a function that called receive a hardcoded number of times (I called receive more as the tests got more complex). I initially tested after completing sendblk but before I updated receive to be able to handle the buffer. I then created a testreceive (with priority 10) process and two testsendblk (one with priority 16 and one with 15) processes that sent to the testreceive process. The first message was sent normally, and then I verified that the queue was working correctly. I then added a few more processes to further verify. Then I updated the receive() system call and made sure that messages were received as expected and in the right order with one testreceive process and 4 testsendblk processes, where the sendblk processes executed first and then testreceive called receive multiple times. I think that if a process terminates while other processes are blocking to send, they should be unblocked and allowed to complete execution. Because they are sending data to the other process, but not relying on data coming back from it, whether or not the message is received should not affect their ability to complete execution. This may not be the case if synchronous inter-process communication is being used to simulate semaphores, but because XINU supports semaphores already, there is no reason to use IPC to accomplish this.

4. In order to implement asynchronous inter-process communication, I wrote a function cbcheck (in cbcheck.c) which checked if a process had a message and if it had a registered callback function, and

if both of those were true it would call the callback function. I called cbcheck from within ctxsw, after interrupts are re-enabled, right before context switch returns to the new process. By running the function right before the new process is executed, the function is executed in the context of the new process (not the previous process or in kernel mode with interrupts disabled). To test this I wrote testasipcrec in testasipcrec.c, which registered a function similar to the example callback in the handout (in callback.c, callback stores the message in the global mbuf variable in testasipcrec.c, and then prints the value of the message received and the value of currpid to make sure that it was being executed in the right context). Then back in testasipcrec, I print the value of mbuf I sent and slept between prints to force context switches (and more calls to callback), and then print the value of mbuf to make sure a new message is received when it should be.

Bonus. I wrote man pages for sendblk and receive