Lab 4

3.3

To test that both *send()* and *sendb()* works as expected with the modified *receive()*, I wrote helper functions, *sender()*, *senderb()*, and *receiver()*, to call those functions and include print statements.

- In *sender()*, I printed *clkcountermsec* and the message it's sending in *send()*. Then, I printed its status based on the output of *send()*.
- Similarly for *senderb()*, I printed *clkcountermsec*, the message it's sending in *sendb()*, and the send status based on the output of *sendb()*. However, I added a *sleepms()* call for 40 milliseconds to keep the process running, since *receive()* still works with the sender's PID after receiving its message.
- Lastly, in *receiver()*, I implemented an infinite while loop that has a *sleepms()* call for 5 milliseconds, so it won't always be in a ready state. Then, I check if its *prhasmsg* flag is set. If it is, I printed *clkcountermsec* and the message it received from *receive()*.

```
1 ms: Send 'hi' from pid 5.... Sent!
---------------------------
7 ms: Send 'hello' from pid 6.... Failed to send :(
---------------------------
14 ms: Receive 'hi' at pid 4
---------------------------
15 ms: Send 'hey' from pid 7.... Sent!
---------------------------
21 ms: Receive 'hey' at pid 4
---------------------------
```

In *main()*, I created and resumed 1 *receiver()* process with INITPRIO priority so it would still allow *main()* to execute. Then, I created and resumed 3 *sender()* processes with a higher priority than *main()*, all sending different messages to the same receiver process. The results are as above, which works as expected because the receiver is only able to hold one message at a time when using *send()*.

```
30 ms: Sendb 'pls' from pid 8.... Sent!
---------------------------
35 ms: Sendb 'bye' from pid 9.... Sent!
---------------------------
42 ms: Sendb 'ok' from pid 10.... Sent!
---------------------------
48 ms: Receive 'pls' at pid 4
---------------------------
54 ms: Receive 'bye' at pid 4
---------------------------
60 ms: Receive 'ok' at pid 4
```

I then called *sleepms()* for 10 milliseconds so *receiver()* can receive any messages sent by *sender()* and it won't affect the *senderb()* processes created later. After that, I created and resumed 3 *senderb()* processes with a higher priority than *main()*, all sending different messages to the same receiver process. The results are as above, which works as expected because the receiver managed to receive all of the sent messages through the queue implemented in *sendb()*.

4

```
pid32 vic = create(abc, 1024, 25, "abc_vic", 0, NULL);
resume(vic);
resume(create(wrongwayhome, 1024, 25, "wrongwayhome", 1, vic));
```

As expected, executing the above code in *main()* invokes *curveball()* and outputs “This is a curveball”, even though we did not directly call *curveball()* in any of the executing processes.

5

I added a new process table field *char *detourptr*, which stores the most recent detour pointer for a process. It is initialized to NULL in *create.c* and is updated in *dreamvac()* with the given function pointer every time it is called, ensuring that the process’ detour function pointer is updated with the most recent *dreamvac()* call.

The stack layout of the process that called *dreamvac()* is modified in *wakeup()*, since the process will be context-switched out there. I used a global variable to store the EBP of *sleepms*’s stack so I would know how many stack elements to move. In *wakeup()*, I access the ESP of the process that called *dreamvac()* through *prstkptr*. Then, the number of elements to move is calculated by subtracting the ESP from *sleepms*’s EBP and stored in a variable *offset*. I then accessed the EBP of the process that called *dreamvac()* by adding 2 to the ESP, as that’s how *ctxsw()* left it. Then, I used a while loop to subtract each modified function’s EBP value by 4 to account for the stack modification. Each function’s EBP address is accessed indirectly using the previous function’s EBP value. Afterward, I copied the stack elements one address down for *offset* times. This allowed space for me to add the address of the function pointer we are detouring to after *sleepms()* returns. Finally, I subtracted the ESP value by 4 to account for the stack modification.

To test and verify that my implementation works, I created a function in *main()* called *test5()* that calls *dreamvac(vacation)*. It then calls *sleepms()* for 500 milliseconds and prints out “back in test5”. As expected, when creating and resuming a process to execute *test5()*, the code prints out “On vacation” before printing out “back in test5”.

Bonus

Since we overwrite RET1 to jump to *detourwrapper()*, *sleepms()* now returns to *detourwrapper()* instead of *vacation()*. To perform the second part of the detour in *detourwrapper()*, we need to first store RET1 in a global variable before it is overwritten. Then, in *detourwrapper()*, after we have called *vacation()*, we access the EBP pointer and overwrite the address above the EBP pointer with the saved RET1. Hence, *detourwrapper()* now returns to the original caller of *sleepms()*.