Lab 3

3.1

Method for testing: My method to test cpuusage() was simply to create a function with a for-loop that incremented a variable. After the for-loop ended, I called cpuusage() to see the amount of cpu usage. I increased and decreased the number of loops to see if cpuusage() would also increase and decrease accordingly.

Why currcpu may be inaccurate: currcpu is incremented in clkhandler. Whenever an interrupt is triggered, it will prevent clkhandler from running at the regular intervals and thus throw off the accuracy currcpu. Therefore when using currcpu to determine cpu usage, it may not return the “true” usage.

3.2

To test responsetime(), I created two processes, one IO bound process and one CPU bound process. I then called resume() on both of them, and then printed out their prctxswcount, prresptime, and the result of calling responsetime() on them. This test does not fully account for edge cases, but it confirmed that that responsetime() worked for the majority of cases because the values appeared to be what I expected.

4.4A

I ran the benchmark as described in the handout. Because STOPPINGTIME is set to 8000 ms, and all CPU bound processes start with equal priority and have the same behavior (i.e., are CPU bound), they retain the same priority as each other. Therefore, I expected them to share the CPU equally and have similar response times (which the handout also mentioned).

My results were as follows:

```
**********BENCHMARK A*********************
'CPU' PID: 5 MSCTR: 8001 CU: 1381 RT: 237
'CPU' PID: 6 MSCTR: 8005 CU: 1385 RT: 238
'CPU' PID: 7 MSCTR: 8009 CU: 1415 RT: 238
'CPU' PID: 8 MSCTR: 8014 CU: 1445 RT: 238
'CPU' PID: 9 MSCTR: 8018 CU: 1475 RT: 239
'CPU' PID: 4 MSCTR: 8023 CU: 1375 RT: 229
```

As you can see, my code seems to be working as expected, since the CPU usage is roughly equal and there are similar response times. The reported CPU usage is equivalent to roughly 8000 ms (STOPPING TIME) divided by 6 (for the 6 processes).

For reference, in this print statement, ‘IO’ indicates that it’s an IO bound process, ‘PID’ indicates the PID, ‘MSCTR’ represents the value of msclkcounter2, ‘CU’ represents CPU usage, and ‘RT’ indicates response time.

4.4B

I ran the benchmark as described in the handout. I expected the IO bound processes to have a lower cpu usage than the CPU bound processes, as well as a much faster response time. Additionally, I expected the IO bound processes to share the CPU equally with each other since they would all retain the same priority since they would all increase priority each time they were run (which the handout also mentioned.

My results were as follows:

```
**********BENCHMARK B**********************
 Hello! My name is Rayyan Khan, and my Purdue username is khan274.
'IO' PID: 7 MSCTR: 8036 CU: 286 RT: 1
'IO' PID: 4 MSCTR: 8040 CU: 292 RT: 1
'IO' PID: 9 MSCTR: 8055 CU: 286 RT: 1
'IO' PID: 5 MSCTR: 8058 CU: 289 RT: 1
'IO' PID: 8 MSCTR: 8077 CU: 303 RT: 1
'IO' PID: 6 MSCTR: 8080 CU: 304 RT: 1
```

One thing I had to do while debugging was make the print statements shorter. This is because when it had the same length as the print statement I used in cpubnd(), it wasn't able to completely print within the IO-bound time slice, so the print statement was getting cut off by the next process. In this print statement, 'IO' indicates that it's an IO bound process, 'PID' indicates the PID, 'MSCTR' represents the value of msclkcounter2, 'CU' represents CPU usage, and 'RT' indicates response time. As expected, the response time is extremely fast (1 ms, which is the lowest possible), and the cpu usage is a fraction of that of the cpu-bound process's cpu usage.

4.4C

I ran the benchmark as described in the handout. When executing both CPU bound and IO bound processes, I expected to see the IO bound processes to have significantly faster response times and lower CPU usage than the CPU bound processes. I also expected the IO bound processes to all have roughly equivalent numbers and the CPU bound processes to have roughly equivalent numbers. Initially I expected the IO bound processes to print their outputs first, but that isn't what happened for me.

My output is as follows:

```
**********BENCHMARK C**********************
'CPU' PID: 9 MSCTR: 8001 CU: 2170 RT: 76
'IO' PID: 4 MSCTR: 8004 CU: 559 RT: 1
'CPU' PID: 5 MSCTR: 8007 CU: 2090 RT: 55
'CPU' PID: 7 MSCTR: 8011 CU: 2061 RT: 55

 Hello! My name is Rayyan Khan, and my Purdue username is khan274.
'IO' PID: 6 MSCTR: 8040 CU: 558 RT: 1
'IO' PID: 8 MSCTR: 8076 CU: 558 RT: 1
```

The reason that I believe that the IO bound processes printed after the CPU bound processes is actually because of the call to sleep(). The msclkcounter2 variable hit the value of STOPPINGTIME() while the IO bound processes were sleeping, leading to the CPU being turned over to the CPU bound processes. While the IO-bound processes were sleeping, the CPU bound processes were able to print their outputs. Overall, the IO bound processes did have faster response times and CPU usage, but they didn't print in the expected order due to my particular for-loop numbers.

4.4D

I ran the benchmark as described in the handout. I expected to see similar output as 4(c) for the first four processes, which were IO bound processes and CPU bound processes. For chameleon(), I expected to see a fast response time as well as high CPU usage. This is because I coded it with the intention of exploiting the priority scheduler we made, as asked in the handout. What I did was copy most of iobnd(), except I reduced sleep(80) to sleep(0). I also moved sleep(0) before my for-loop because it seemed to work more effectively there. I did this because the handout says, “If a process knows that there are no ready processes of equal priority then repeatedly calling sleepms(0) could be a way for a process to amplify its priority, a potential system vulnerability.”

My output was as follows:

```
**********BENCHMARK D**********************
'CMLN' PID: 4 MSCTR: 8001 CU: 7996 RT: 1
'CPU' PID: 5 MSCTR: 8004 CU: 0 RT: 7999
'CPU' PID: 6 MSCTR: 8008 CU: 0 RT: 8003
'IO' PID: 7 MSCTR: 8011 CU: 0 RT: 8006
'IO' PID: 8 MSCTR: 8015 CU: 0 RT: 8010
```

It worked as expected.

Bonus

For the bonus, I decided to modify iobnd() in a new function called iobnd9() to have an extremely long while loop. This way, sleep(80) is not called until iobnd9() has incurred a very high CPU usage. Additionally, I put sleep(0) in the for-loop because this wouldn't give up the cpu, but it would increase the priority to ensure that it always gets picked before a CPU bound function. The CPU bound function will be starved if the 5 iobnd() functions together take a very long time to run, therefore preventing the CPU bound function from running for a long time.

My output was as follows:

```
*******************BONUS**********************
'IO9' PID: 4 MSCTR: 8660 CU: 8654 RT: 1
'IO9' PID: 5 MSCTR: 8664 CU: 0 RT: 8658
'IO9' PID: 6 MSCTR: 8668 CU: 0 RT: 8662
'IO9' PID: 7 MSCTR: 8672 CU: 0 RT: 8666
'IO9' PID: 8 MSCTR: 8677 CU: 0 RT: 8671
'CPU' PID: 9 MSCTR: 8681 CU: 0 RT: 8675
```

What you can see is happening is that the first IO bound process is using all of the allotted time just to run. All the other processes are starved, as you can see because CPU usage is 0. We can improve our dynamic priority scheduler by implementing a threshold, as Unix Solaris does. We would need to add a field to the process table that tracks how much time a process has spent in the readylist. If it reaches the predetermined schedule, then we should increase the priority of that process such that it is next picked.