3.1:

To test my CPU usage, I create two processes that perform an empty for loop to take up CPU time. The first process has an upper bound for the loop set to 10 million and the other one has an upper bound of 20 million. We expect a non-zero CPU usage for both processes with the first process taking half the CPU time compared to the second process. Even though the two processes will run in parallel since they have the same priority (round-robin fashion), the second process is bound to take twice as long compared to the first since the for loop has an upper bound that is also twice as much. When running the tests, we see that the first process has a CPU usage of 276 (msec) and the second process has a CPU usage of 552 (msec).

There are many reasons as to why our currcpu and cpuusage is not very accurate. The first reason being that the clock is incremented via clock interrupts. The problem is that whenever a process tries to perform a system call in XINU, we disable interrupts until after the system call is complete. Disabling interrupts stops the clock from being incremented. Since our currcpu is incremented the same time that as the clock, when a clock interrupt occurs, disabling interrupts would also result in us not incrementing currcpu leading to an unreliable count of how long the current process has been running for. Another reason for it not being accurate is that we only interrupt every 1 (msec). There can be processes that run faster than 1 (msec) and for those we would be overestimating by rounding to 1 (msec).

3.2:

To test the responsetime of our processes, I create a process (P1). P1 will create another process (P2) with a lower priority so that it must wait until P1 is finished to start running. We will resume it as soon as it is created to insert it into the ready list. We will then do a cpu extensive for loop that will waste the CPUs time. After the for loop finished, we will print the response time of P1 and then P2 will start running since P1 is done. We won't do anything in P2 expect print the response time of that process. P2 has been in the ready list since before the cpu extensive for loop in P1 so we should expect its response time to be relatively higher than P1. When we run our test case, we see this exact behavior thus we can confirm that our response time is working correctly.

4.4:

Benchmark A:

The cpu usage of the 6 processes is evenly split during the 8 seconds that we are waiting for msclkcounter2 to be larger than STOPPINGTIME. This is to be expected because all the cpu processes have the same priority, so they go in round robin, evenly splitting the cpu time between the processes.

Benchmark B:

When running benchmark B, we see that the cpu usage and response time for all the processes are very similar. The response time for IO processes is significantly lower compared to CPU bound processes because IO bound processes spend most of their time sleeping. When they wake up, they will have a high priority meaning the CPU will pick it during resched first and run it thus explaining the low response time. The CPU usage is also very low. Since CPU usages represents the amount of time it has spent in the running state. Since sleep puts the process to sleep and no longer running, this results in the CPU usage not increasing. The response time is very low as expected because the priority of the process is higher than any other running processes. This means that as soon as the iobnd process wakes up from sleepms(80), it will start running resulting in a super low response time.

Benchmark C:

For benchmark C, I create 3 cpu processes and 3 io processes. Resuming them all, we see that cpu bound processes have a high cpu usage and response time and io bound processes have the opposite, as expected. The cpu bound processes all finish and print their outputs first before the IO bound processes. While IO bound processes should run as soon as the cpu is available, it doesn't because of the sleepms(80) inside of iobnd. The sleepms(80) constantly keeps taking the process out of the ready list and when it wakes up to run, it gets put back to sleep. The odds of any of the iobnd processes being awake when msclkcounter gets above SLEEPINGTIME is very low. This explains why all the cpu bound processes finish before any of the io bound processes.

Running the test multiple times, we see various outputs based on if an io bound processes is awake when msclkcounter gets above SLEEPINGTIME. Sometimes the output of the cpu bound process gets interrupted and the output of the iobnd prints before letting the rest of the cpu process's output. We can further confirm this theory by lowering the amount of time an io bound process sleeps for by changing sleepms(80) to something like sleepms(10). This increases the chances of an io bound process being awake msclkcounter gets above SLEEPINGTIME thus resulting in iobnd processes all running and finishing after one cpu processes finished (it was hogging the cpu until now). Once all the iobnd processes finish, then the rest of the cpu bound processes will finish and output the result.

Benchmark D:

For benchmark D, I create 2 cpu bound processes, 2 io bound processes, and 1 chameleon process. For the chameleon process, I call sleepms(0) on it while the msclkcounter2 is less than STOPPINGTIME. This exploits a weakness in how our dynamic priority XINU scheduler classifies processes to promote/demote priorities. Doing this causes the priority of the chameleon process to keep increasing until it reaches the ceiling of 5. Since it keeps increasing its priority, it will always keep running on the cpu until the msclkcounter2 < STOPPINGTIME condition becomes false which explains the high cpu usage. The response time is low as expected because the chameleon process

For the cpu bound processes, we get a cpu usage of 30 (msec) which is the exact time slice that the cpu bound process should receive since its initial priority is 3. The response time for the cpu processes is around 4,000 (msec). This makes sense because the process stay in the ready list for STOPPINGTIME (8,000 msec). It also only context switches twice, once when it first becomes ready and once when it runs again after finishing chameleon. Since the response time function returns prresptime / prcxswcount we would expect an output of 8,000 / 2 = 4,000 (msec)

For the io bound processes, we get a cpu usage that is super low (5 msec in my case) which is because of the for loop inside of iobnd. Response time for iobnd processes on average is 40 msec less than cpu bound processes. This makes sense because the io bound processes call sleepms(80) inside which causes them to be removed from the ready list for 80 msec. When we call responsetime() for the iobnd process, it would divide prresptime by 2 (the number of times it context switched in). Since prresptime for io bound processes is 80 msec less, we would expect the output of responsetime() to be 40 msec less when compared to cpu bound processes.

Bonus:

To mitigate starvation in the xinu dynamic scheduler we can use the new process field, prbeginready, that we created in this lab to keep track of when a process becomes ready. Then at every clock interrupt, we can loop through the ready list and see if the difference of prbeginready and msclkcounter is more than a specified threshold (ex: 1 sec). If it is, then we increase the priority so that it has a higher chance of running soon.

I created a function, iobnd9, that calls sleepms(0) over and over for 8 seconds. This happens in each process that calls iobnd9. This causes the iobnd9 process to have a really high priority and takes up a huge amount of cpu time since it will keep running until the while loop equates to false. When running this workload, we see that the first iobnd9 process has cpu usage of 8000 (8 sec) and response time of 1 since it's the first one picked and to finish. The second iobnd9 process has a cpu usage of 8000 (8 sec) as well which is expected because the while loop forces the process to call sleepms(0) for 8 seconds. The response time for this process is 8000 (8 sec) which is also expected because it must wait for the first process to finish before it can start. This continues for all iobnd9 processes, cpu usage is 8000 (8 sec) and

response time is 8000 more than the process that finished before it, expect for the first process which will always have a super low response time.