Brow1325
Connor Brown

3.1

a. Discuss in Lab3Answers.pdf how long (in days and hours) clktimemilli can keep time after bootloading until the counter overflows. Truncate fractional hours. For how long can a 64-bit millisecond counter keep time?
    i. Clktimemilli can count for (2^32) - 1ms. Which gives us a total of 4294967295 ms. Which is 49 days and 17 hours that clktimemilli can keep time.
    ii. A 64 bit millisecond counter would be (2^64)-1ms. Which gives us a total of 18446744073709551615 ms. Which gives us about 584942417 years, 4 months, 7 days, and 22 hours that a 64-bit millisecond counter can keep time.

3.4

a. Explain in Lab3Answers.pdf why adding instrumentation code before and after calling ctxsw() in resched() is a viable method for estimating a process's gross CPU usage. Consider three scenarios.
    i. One, the current process makes a sleepms() system call which context-switches out the current process and context-switches in a new (i.e., different) process.
    ii. Two, while the current process is executing, a clock interrupt is raised which causes the current process's time slice to be decremented by clkhandler(). If the remaining time slice becomes 0, resched() is called which may, or may not, trigger a context-switch depending on the priority of the process at the front of XINU's ready list.
    iii. Three, while the current process is executing, a clock interrupt is raised which causes a previously sleeping process to be woken up and placed into the ready list. If the newly awoken process has priority greater or equal than the current process, a context-switch ensues.
b. Describe the sequence of XINU kernel function calls that are evoked in the three scenarios. Explain why the 3.3 method will perform correct gross CPU usage tracking in the three scenarios.
    i. In the first scenario when sleepms() is called it calls the clkdisp(), which calls clkhandler() which calls resched(). Within resched() before we context switch out the sleeping process for one that is not sleeping we update the cpu usage for the sleeping process and then switch. Then after we update the currentgrosscpu time for the new process so that when resched is called again we can track the cpu usage of the next process.
    ii. In the second scenario when the clkhandler() is called from a clock interrupt and the time slice is 0 then resched() is called. Within resched() before we context switch out the sleeping process for one that is not sleeping we update the cpu usage for the sleeping process and then switch. Then after we update the currentgrosscpu time for the new process so that when resched is called again we can track the cpu usage of the next process.

iii. In the third scenario when a clock interrupt is executed to wake a sleeping process we call wakeup(), which then calls ready(), which calls resched(). Within resched() before we context switch out the sleeping process for one that is not sleeping we update the cpu usage for the sleeping process and then switch. Then after we update the currentgrosscpu time for the new process so that when resched is called again we can track the cpu usage of the next process.

4.2

a. Specify in Lab3Answers.pdf what these changes are and implement them in XINU. This includes how you handle XINU's null process. We will continue to use XINU's priority queue which has linear insert overhead over heaps that have logarithmic overhead for three reasons. One, since you will know from data structures how to code heaps and balanced search trees, we will omit its implementation for simplicity; two, we will not be testing with hundreds and thousands of processes where difference in overhead will surface in a pronounced manner; and three, kernel support for dynamic memory allocation which is needed for managing heaps will be discussed under memory management later in the course..
    i. The only change was in the while loop where instead of checking if the current process in the queue has a priority greater than the priority of the process we want to insert we check if it has a lower priority than the process we want to insert and then loop through until we reach a process that has a higher priority. That is then the one that is saved in curr when we exit the loop.
    ii. The rest is the same as it insert the process between the current node and the previous node.
    iii. In order to use rinsert instead of insert I had to change ready.c and resched.c by changing the call to insert to a call for rinsert.
    iv. We also have to change the queuehead to MINKEY and queuetail to MAXKEY in newqueue.c

4.3

a. The null process must be treated as a special case so that its CPU usage is strictly greater than all other processes. This implies that the null process, when not current, is always at the end of the ready list. Explain in Lab3Answers.pdf how you ensure that this is the case.
    i. To ensure that the null process is always at the end of the list we set it to the MAXKEY - 1. This ensures that it always has the largest value. We also make sure to check for the null process in resched when updating the oldptr->prvgrosscpu so that the null process will always be the largest.

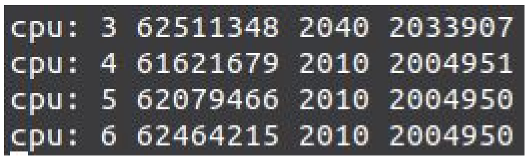
4.4

a. When a new process is spawned using create(), ignore the third argument specifying the process priority and follow mechanism (i) to set the process's initial priority. XINU's null process (PID 0) must be treated separately so that its priority is always strictly less than the priority of all other processes in the system. Explain in Lab3Answers.pdf how you go about assuring that.

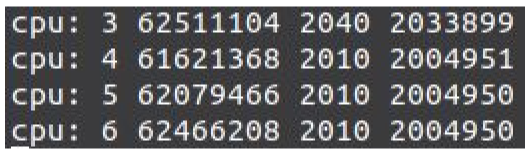
   i. To ensure that the null process has the lowest priority we initialize it with its prvgrosscpu value to MAXKEY-1. This keeps it at the end of the ready list. When creating a new process we check if the current process running is the null process and if it is then we set the new process prvgrosscpu to 0. We also check if the null process is the only process in the ready list and if it is then we set the new process prvgrosscpu to 0. If the current process running is not null and the ready list has more than the null process in it then we get a pointer to the end of the list and then get the id of the process before it in the list. This way we can set the priority of the new process prvgrosscpu to the highest process prvgrosscpu in the ready list that is not the null process.

5.2

a. Create 4 CPU-bound processes from main() back-to-back. If your scheduler is implemented correctly, we would expect to see the 4 processes printing similar CPU usage and x values. Repeat the benchmark test one more time and inspect your results. Discuss your findings in Lab3Answers.pdf.



```
cpu: 3 62511348 2040 2033907
cpu: 4 61621679 2010 2004951
cpu: 5 62079466 2010 2004950
cpu: 6 62464215 2010 2004950
```

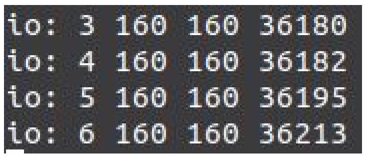


```
cpu: 3 62511104 2040 2033899
cpu: 4 61621368 2010 2004951
cpu: 5 62079466 2010 2004950
cpu: 6 62466208 2010 2004950
```

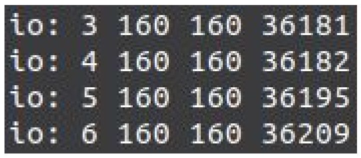
   i. In both benchmarks tests it shows that the 4 processes are printing similar x values and usage both times. There is a small variance in the x values and micro but it is so small that it is insignificant for comparison. This shows that when all processes are CPU-bound that the scheduler acts correctly.

5.3

a. Create 4 I/O-bound processes from main() and perform the same benchmark tests as 5.2. Discuss your findings in Lab3Answers.pdf.



```
io: 3 160 160 36180
io: 4 160 160 36182
io: 5 160 160 36195
io: 6 160 160 36213
```

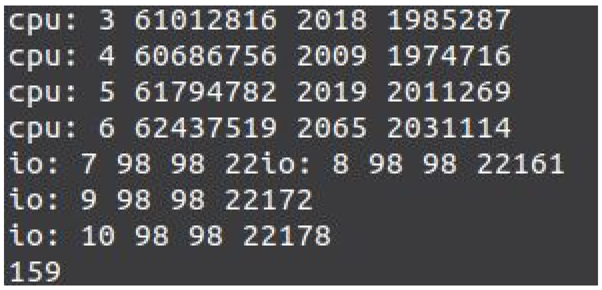


```
io: 3 160 160 36181
io: 4 160 160 36182
io: 5 160 160 36195
io: 6 160 160 36209
```

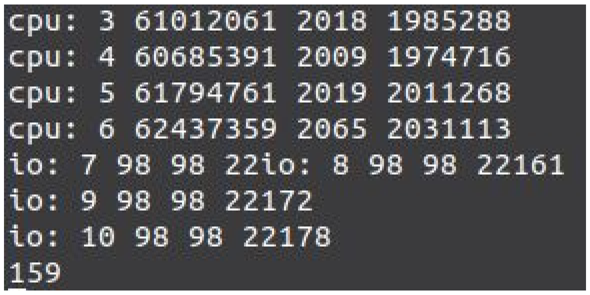
   i. Both benchmark tests show almost identical values. With slight differences among the microtime it is safe to assume that the I/O processes operate correctly in the scheduler.

5.4

a. Create 4 CPU-bound processes and 4 I/O-bound processes. We would expect the 4 CPU-bound processes to output similar x values and CPU usage with respect to each other, and the same goes for the 4 I/O-bound processes within the group of I/O-bound processes. Across the two groups, we would expect CPU-bound processes to receive significantly more CPU time than I/O-bound processes. Discuss your findings in Lab3Answers.pdf.



```
cpu: 3 61012816 2018 1985287
cpu: 4 60686756 2009 1974716
cpu: 5 61794782 2019 2011269
cpu: 6 62437519 2065 2031114
io: 7 98 98 22io: 8 98 98 22161
io: 9 98 98 22172
io: 10 98 98 22178
159
```

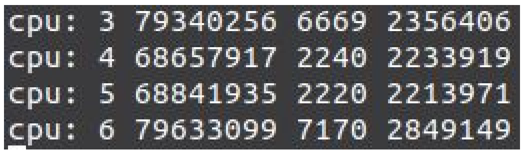


```
cpu: 3 61012061 2018 1985288
cpu: 4 60685391 2009 1974716
cpu: 5 61794761 2019 2011268
cpu: 6 62437359 2065 2031113
io: 7 98 98 22io: 8 98 98 22161
io: 9 98 98 22172
io: 10 98 98 22178
159
```

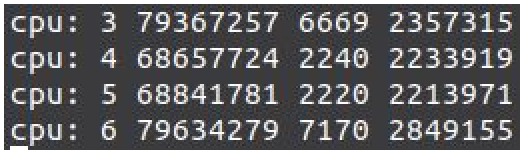
   i. In both cases it prints slightly jumbled output but this could be due to many reasons. In both cases it does still keep similar values and so it is safe to assume that when both CPU-bound and I/O-bound processes are implemented in the scheduler that they are being handled correctly.

5.5

a. A variant of test scenario A, create 4 CPU-bound processes in sequence with 500 msec delays (by calling sleepms()) added between successive create() (nested with resume()) system calls. Estimate how much CPU time the first process should receive. The same goes for the other 3 processes. Compare your calculations with the actual performance results from testing. Discuss your findings in Lab3Answers.pdf.



```
cpu: 3 79340256 6669 2356406
cpu: 4 68657917 2240 2233919
cpu: 5 68841935 2220 2213971
cpu: 6 79633099 7170 2849149
```



```
cpu: 3 79367257 6669 2357315
cpu: 4 68657724 2240 2233919
cpu: 5 68841781 2220 2213971
cpu: 6 79634279 7170 2849155
```

Time	CPU 3	CPU 4	CPU 5	CPU 6
First sleep end	500ms	0ms	0ms	0ms
CPU 4 initialized	500ms	500ms	0ms	0ms
Second sleep end	740ms	730ms	0ms	0ms
CPU 5 initialized	740ms	730ms	740ms	0ms
Third sleep	910ms	910ms	890ms	0ms

end				
CPU 6 initialized	910ms	910ms	890ms	910ms
Order finished	Third done	Second done	First done	Last done

i. Based on my calculations and the results observed I can see that the times for each process reflect the order in which they finish. And the order for determining which process finishes first is based on where it is inserted in the queue.

Bonus

a. Methods (i) and (ii) are heuristics aimed at addressing the concerns pointed out in 4.3. What is a potential weakness of method (i)? How would you improve on (i)? Describe your solution in Lab3Answers.pdf. Do the same for method (ii). What other issue does Linux CFS have compared to scheduling methods used in UNIX and Windows? Is this a big concern? Explain your reasoning.
    i. A potential weakness of method (i) as described in 4.3 is that it does not explain what to do in the case that the only process in the ready list is the null process. Because it does mention that we must make the new process the lowest priority, but we want to keep this priority still higher than the null process and if the null process is the only priority in the list then we would set it to the same value as the null process. By setting the value to the null process this way it would put it after it in the ready list based on how rinsert works in 4.1.
    ii. To fix this weakness we should have the newly created process initialized to 0 as this would insert it at the front of the list. This would also make sense for if another process is initialized after because then it could be set to the same value and then there would not be a huge gap in the amount of time until either process runs.
    iii. Method (ii) has the same weakness as we do not want to initialize the unblocked process to the null value if it is the only process in the ready list as this will insert it after it in the ready list and this should not be the case.
    iv. The fix for method (ii) is the same as we do a check for if the null process is the only process in the ready list and if it is the case then we set the unblocked process to 0.
    v. CFS on linux has the issue of having processes take longer than needed because it gives potential I/O processes higher priority when it could have a blocking nature at the start of the process and then after the initial block it is a cpu-bound process. This would then cause the scheduler to give that process a high priority and it would cause it to use the cpu over other processes that might need it.