3.1

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?

As a uint32 variable can have a value of 2^{32}-1. So clktimemilli has a maximum value of 2^{32}-1. Converting this to be representative of days and hours is as follows:
2^{32}-1 / 1000 / 60 / 60 / 24= 49 Days
(2^{32}-1 % (1000 * 60 * 60 * 24)) / (1000 * 60 * 60) = 17 hours

So, clktimemilli can record up to 49 days 17 hours.

A 64-bit millisecond counter would be able to keep time for 213503982334 days 14 hours.

OR 584942417 years 14 hours.

3.4

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. 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.

This is a viable way of estimating a process's gross CPU usage because the amount of time that the new process is switched in is calculated by the difference between clktimemilli before and after ctxsw() is called. This essentially counts the time in milliseconds that the new process was being run.

Scenario 1: 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.

sleepms() disables interrupts, changes process state to sleep, calls insertd()

insertd() inserts the process into the delta list, returns back to sleepms()

sleepms() calls resched()

resched() selects the next process in ready queue

The clock timers start.

resched() calls ctxsw()

ctxsw() switches the old processes stack with the new process stack, restores interrupts, returns to new process

Eventually, resched() is called in the new process, the old process is resumed in the call to ctxsw() which then returns back to resched().

Then the gross times are updated.

Scenario 2: 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.

Old process is interrupted by a clock interrupt, clkhandler() is called

clkhandler() decrements preempt, so resched() is called.

resched() selects a new process to run from the ready list and calls ctxsw(); however, if the process selected has a lower priority than the old process, then the old process will resume with a fresh preempt.

The clock timers start.

If ctxsw() is called, then it switches the old processes stack with the new process stack and resumes the new process

Eventually, resched() is called in the new process, the old process is resumed in the call to ctxsw() which then returns back to resched().

Then the gross times are updated.

Scenario 3: 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.

Old process is interrupted by a clock interrupt, clkhandler() is called

clkhandler() checks to see if the delta list (sleep list) is empty. If it isn't and the process is done waiting, then call wakeup()

wakeup() calls resched_cntl(DEFER_START)

resched_cntl() increments ndefers and returns back to wakeup()

wakeup() then turns all of the sleeping processes with no more time waiting to ready. Then it calls resched_cntl(DEFER_STOP)

resched_cntl() decrements ndefers then calls resched()

resched() now selects a new process to run from the ready list. Now assuming that the newly awoken process has the highest priority, then this process is chosen as the new process. ctxsw() is called.

The clock timers start.

ctxsw() it switches the old processes stack with the new process stack and resumes the new process

Eventually, resched() is called in the new process, the old process is resumed in the call to ctxsw() which then returns back to resched().

Then the gross times are updated.

3.5

Perform benchmark runs and compare the values returned by procgrosscpumicro() against those returned by procgrosscpu().

I created a process with an infinite loop in it, so that the gross cpu time will increase over time. The value returned by procgrosscpu() * 1000 will always be higher than the value returned by procgrosscpumicro() because resched() rounds up the gross cpu time whereas the gross cpu time in microseconds is dependent on the number of cpu ticks, which is never rounded. With this being said, the longer a process is active, the more inaccurate procgrosscpu() will become.

4.2

Describe changes made to support non-decreasing priority system:

Changes made to support non-decreasing priority system:

- **Added MAXPRIO to process.h**
- **Changed NULLPROC to have priority MAXPRIO**
- **Changed empty proc table entries to have prio MAXPRIO**
- **Changed the ready list so that the head has MINKEY and the tail has MAXKEY to account for the non-decreasing order.**
- **Changed the checks in rcreate to accomodate for non-decreasing**
- **Replaced calls to insert with rinsert.**
- **Changed comparator in resched() to accomodate for non-decreasing**

4.3

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 am ensuring that the NULLPROC always has the highest value of prvgrosscpu so that it is guaranteed to be the last item in the readylist. Wherever the values of prvgrosscpu are updated, a check is made to ensure that the NULLPROC's prvgrosscpu value will always be at least one value higher. Also, I added checks to physically check to see if NULLPROC is the last item in ready list, if it is not, then rearrange the ready list to make it happen.

4.4

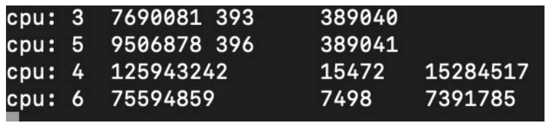
XINU's null process (PID 0) must be treated separately so that its priority is always strictly less than the priority of all other processes in the system. Explain in Lab3Answers.pdf how you go about assuring that.

Since we are using the prvgrosscpu value as the priority, a higher priority means a lower prvgrosscpu value and vise versa. Therefore, in order to ensure that the NULL process has the lowest priority, we have to ensure that the value of prvgrosscpu time is the highest at all times. This is taken care of by 4.3.

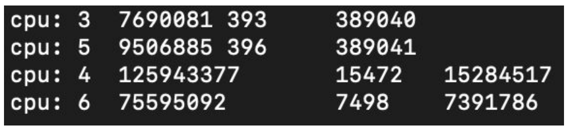
5.2

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.

With my implementation, this is the output for trials 1 and 2:



```
cpu: 3  7690081 393     389040
cpu: 5  9506878 396     389041
cpu: 4  125943242       15472   15284517
cpu: 6  75594859        7498    7391785
```



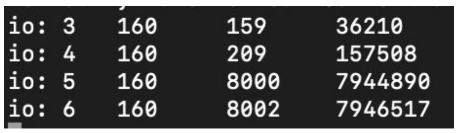
```
cpu: 3  7690081 393     389040
cpu: 5  9506885 396     389041
cpu: 4  125943377       15472   15284517
cpu: 6  75595092        7498    7391786
```

With my implementation, I can see that the first two processes are very close and the last two processes are very close. However, compared to each other, they are not very close.

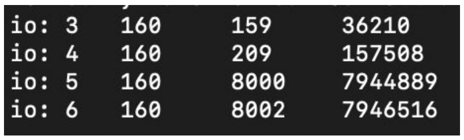
5.3

Create 4 I/O-bound processes from main() and perform the same benchmark tests as 5.2.

With my implementation, this is the output for trials 1 and 2:



```
io: 3   160     159     36210
io: 4   160     209     157508
io: 5   160     8000    7944890
io: 6   160     8002    7946517
```



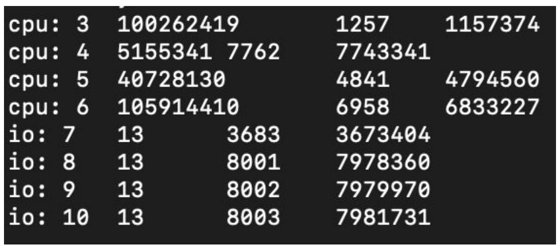
```
io: 3   160     159     36210
io: 4   160     209     157508
io: 5   160     8000    7944889
io: 6   160     8002    7946516
```

I see that the value of x is the exact same throughout the four tests, and, once again, the first two processes have a very similar cputime and the last two processes have a very similar cputime.

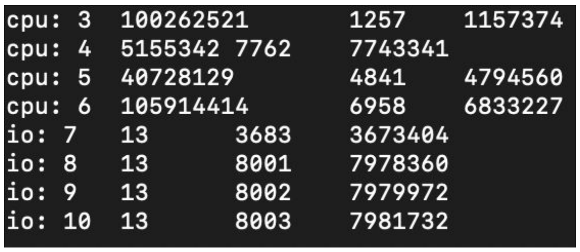
5.4

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

Trials 1 and 2:



```
cpu: 3  100262419       1257    1157374
cpu: 4  5155341 7762    7743341
cpu: 5  40728130        4841    4794560
cpu: 6  105914410       6958    6833227
io: 7   13      3683    3673404
io: 8   13      8001    7978360
io: 9   13      8002    7979970
io: 10  13      8003    7981731
```



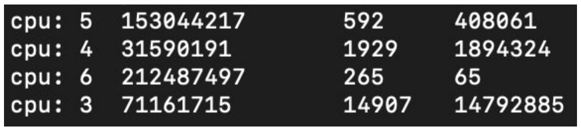
```
cpu: 3  100262521       1257    1157374
cpu: 4  5155342 7762    7743341
cpu: 5  40728129        4841    4794560
cpu: 6  105914414       6958    6833227
io: 7   13      3683    3673404
io: 8   13      8001    7978360
io: 9   13      8002    7979972
io: 10  13      8003    7981732
```

In this test, I see that the IO test produce the same count of x, and have relatively similar grosscputime throughout the tests. The cpu tests are slightly more scattered but are still relatively similar.

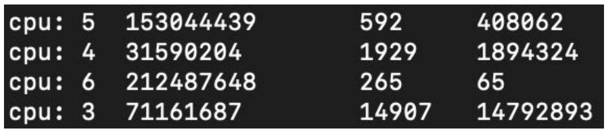
5.5

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.

Trials 1 and 2:



```
cpu: 5  153044217       592     408061
cpu: 4  31590191        1929    1894324
cpu: 6  212487497       265     65
cpu: 3  71161715        14907   14792885
```



```
cpu: 5  153044439       592     408062
cpu: 4  31590204        1929    1894324
cpu: 6  212487648       265     65
cpu: 3  71161687        14907   14792893
```

Adding sleep calls in between creation of processes yields interesting results. With my implementation, adding a sleep call messes with the scheduling quite a bit; however the results are consistent.