3.1

According to kernel.h, uint32 is type unsigned int.

And according to limits.h, maximum value of unsigned int32 is (2UL*LONG_MAX+1).

(2UL*LONG_MAX+1) = 2 * 2147483647 + 1 = 4294967295.

clktimemilli:

Days: 4294967295 % 8.64e+7 = 49 days;

Hours: (4294967295 % 3.6e+6) - (49 *24) = 17 hours;

64-bit unsigned integer:

2^64-1 = 18446744073709551615;

Days: 18446744073709551615 % 8.64e+7 = 213503982334 days;

So clktimemilli can keep time upto 49 days and 17 hours.

64-bit unsigned integer millisecond counter can keep time upto 213503982334 days.

3.4

Scenario One:

When sleepms() is called, we want to update old process' gross CPU usage and context switch to new process.

Sleepms() sets old process state to PR_SLEEP, which is eligible for context-switching.

Then resched is called, it will trigger context switch and execute our instrumentation code.

This is as expected.

Scenario Two:

clkhandler will call resched() when preempt <=0.

Depending on the priority of front process in Xinu ready list:

if current process has the higher priority, resched will simply return and doesn't touch our instrumentation code.

if first ready process has the higher priortiy and context switch is expected to happen, we execute our instrumentation code.

This is as expected.

Scenario Three:

clkhandler will call wakeup() when processes in sleep list run out of sleep time.

wakeup() calls resched_cntl() and calls ready() to put processes back to ready list.

Then what happens is just like in Scenario Two:

if current process has the higher priority, resched will simply return and doesn't touch our instrumentation code.

if first ready process has the higher priortiy and context switch is expected to happen, we execute our instrumentation code.

This is as expected.

To accurately represent CPU usage measurement for newly created process. I update current cpu time stamps not when it's resumed (not created).

3.5

I have found that the outputs of procgrosscpumicro and procgrosscpu are nearly at 1000:1 ratio.

However, procgrosscpumicro has higher precision, when procgrosscpu loses this precision and is rounded to miliseconds.

For example, procgrosscpumicro has the number between 59000 and 61000 microseconds when procgrosscpu is 60 miliseconds.

4.2

Changes made to implement "new world order":

process.h: I added prvgrosscpu in process.h and initialized it in create (), I updated it in resume before calling ready().

initialize.c: I changed insert to rinsert, initialize null's prvgrosscpu to be 10 times of quantum in initialize.c

rinsert.c: I made rinsert from insert, instead queue traverses to find a spot where all smaller numbers are in front of an entry to be inserted. To ensure null process is always the first process to be inserted (because we want to update null's prvgrosscpu if necessary if a new process is inserted and have null process always the last in ready list), I inserted null process right after readylist = newqueue() in initialize.c. Thus, I added a check in rinsert to check if null process is already in readylist.

resched.c: I changed insert to rinsert and increment prvgrosscpu along with prgrosscpu in resched.c

newqueue.c: I modified queuehead to have minkey, queuetail to have maxkey because we now use ascending order.

Ready.c: I changed insert to rinsert.

4.3

How I made sure null process, when not current, is always at the end of the ready list?

When a process with higher prvgrosscpu than null process is to be inserted, update null's prvgrosscpu to be that process's prvgrosscpu + QUANTUM. This way, null always has the highest prvgrosscpu.

4.4

How I assure io-bound gets lowest prvgrosscpu and cpu-bound gets highest prvgrosscpu?

When a newly created process resume, check the highest prvgrosscpu (of the process previous to null) in readylist and assign to it.

When a process calls wakeup(), check the lowest prvgrosscpu (of the first process in readylist) in readylist and assign to it.

5.2

This is the output I got from two tests.

Try 1:

```
cpu: 3 43819535 1650 1645206

cpu: 5 42245529 1650 1645923

cpu: 4 43042715 1650 1645925

cpu: 6 42245579 1680 1675854
```

Try 2:

```
cpu: 3 43819543 1650 1645207
```

```
cpu: 5 42245545 1650 1645923

cpu: 4 43042728 1650 1645925

cpu: 6 42245595 1680 1675854
```

The output (x, prgrosscpu and procgrosscpumicro) between 4 cpu-bound processes and between two tests are really close to each other.

5.3

This is the output I got from two tests.

Try 1:

```
io: 3 159 159 36083

io: 4 159 159 36061

io: 5 159 159 36071

io: 6 159 159 36084
```

Try 2:

```
io: 3 159 159 36083

io: 4 159 159 36061

io: 5 159 159 36071

io: 6 159 159 36084
```

The output between 4 io-bound processes and between two tests are the same for my tests.

The value of x, prgrosscpu and procgrosscpumicro are the same.

5.4

My output is as following:

```
cpu: 3 43513689 1657 1634020

cpu: 4 42783211 1629 1609545

cpu: 5 41710183 1647 1626112

cpu: 6 41391805 1670 1644051

io: 7 101 220 142625

io: 8 101 220 142840
```

```
io: 9 101 220 143086
io: 10 101 220 143319
```

The output between 4 io-bound processes are close to each other.

And the output between 4 cpu-bound processes are close to each other.

However, the output (x, prgrosscpu and procgrosscpumicro) for io-bound groups are significant smaller than the cpu-bound group.

5.5

Given that sleepms(500) is added in main process between each resume(create(...)) and sleepms is I/O bound and has the highest priority to execute. When first process is resumed and main is put into sleep by calling sleepms(), no other process gets created and resumed to compete for cpu, first process gets the most cpu time during his 8000 ms while loop.

The same logic applies to other processes. Thus, I would expect first process to have higher output (x, prgrosscpu and procgrosscpumicro) than second process, and the second process should have higher output than third process, and third process should have higher output than forth process.

My estimate is that:

first process: 2500 ms

second process: 2000 ms

third process: 1800 ms

forth process: 1700 ms

And the actual output is:

```
cpu: 3 61353249 2810 2303549
cpu: 4 49897003 2019 1874356
cpu: 5 47020039 1800 1795262
cpu: 6 49805938 1890 1884887
```

I would say that my estimate is pretty similar to the actual output, besides a little difference for process 3 and 4.

My interpretation is that as time goes by, more processes are here to compete for CPU resources, so it kind of averages out the difference.

Extra Credit

I would say current methods i and ii generalize every process and only differentiate between io-bound and cpu-bound. Also, the maximum time slices given to each process are the same. This way, each cpu-bound process among all cpu-bound processes are almost identical, this also applies to each io-bound process among all io-bound processes. I would improve method i and ii by using a weighted version with varied time slices and priority (in this case prvgrosscpu). The idea will be similar to Unix Time Sharing scheduling. Although the ultimate goal is similar: to give io-bound process the least cpu usage and give cpu-bound process the most cpu usage as possible. Depending on how important the process is, we can give a more important process lower cpu usage and smaller time budget relative to others in the same bound group(cpu-bound/io-bound), and we can give a less important process a higher cpu usage and longer time budget. Comparing to scheduling methods used in UNIX and Windows, Linux CFS’s algorithm has the complexity of O(logN). Whether this is a big concern depends on how big the machine is. For home computers with minimum usages, this might not make a big difference. However, it can be a performance issue if a commercial computer has tons of processes to run.