3.3

I updated prcputime in resched.c because all process that are rescheduled/context switched out go through the resched() function, whether it's the quantum interruption, sleep, etc. I have a global variable that updates in resched (variable named "initialtime" set equal to clktimefine every time resched is called). When updating the prcputime, I set the "old process" prcputime value to clktimefine-initialtime if preempt != QUANTUM.

4.4

Essentially, any time I dealt with updating a process's priority, I used if-statements to ensure that the updated priority would not change if the original priority was 0, or if the PID was 0, depending on the situation.

5.2

Trial 1:

```
cpu-bound: 4 165809140 5825
cpu-bound: 5 165785387 5800
cpu-bound: 6 165784799 5800
cpu-bound: 7 165784475 5800
cpu-bound: 8 165783809 5800
```

Trial 2:

```
cpu-bound: 4 165817653 5825
cpu-bound: 5 165784953 5800
cpu-bound: 6 165785516 5800
cpu-bound: 7 165785501 5800
cpu-bound: 8 165785258 5800
```

Trial 3:

```
cpu-bound: 4 165808924 5825
cpu-bound: 5 165785231 5800
cpu-bound: 6 165785540 5800
cpu-bound: 7 165785076 5800
cpu-bound: 8 165784976 5800
```

It seems that the processes are getting roughly the same CPU time, except for the first created process which seems to get one more quantum, thus incrementing X more. Also, 5800ms+5800ms+5800ms+5800ms+5825ms = 29.025 seconds henceforth leading me to the conclusion my implementation works (since the test is 29 seconds long).

5.3

Trial 1:

```
io-bound: 4 1450 1450
io-bound: 7 1450 1450
io-bound: 8 1450 1450
io-bound: 5 1450 1450
io-bound: 6 1450 1450
```

Trial 2:

```
io-bound: 4 1450 1450
io-bound: 7 1450 1450
io-bound: 8 1450 1450
io-bound: 5 1450 1450
io-bound: 6 1450 1450
```

Trial 3:

```
io-bound: 4 1450 1450
io-bound: 7 1450 1450
io-bound: 8 1450 1450
io-bound: 5 1450 1450
io-bound: 6 1450 1450
```

Since x is incremented every time the io-bound processes are context switched in, and when preempt==QUANTUM, we add 1 ms to the cpuusage, it makes sense that x is equal to prcpu time

5.4

Trial 1:

```
cpu-bound: 4 165251263 5810
5810
cpu-bound: 5 165565605 5819
5819
cpu-bound: 6 165822136 5825
5825
cpu-bound: 7 165821998 5825
5825
cpu-bound: 8 166292969 5842
5842
io-bound: 9 292 292
io-bound: 10 292 292
io-bound: 11 292 292
io-bound: 12 292 292
io-bound: 13 292 292
```

Trial 2:

```
cpu-bound: 4 165250115

cpu-bound: 5 165567249

cpu-bound: 6 165821828

cpu-bound: 7 165821965

cpu-bound: 8 166292973

io-bound: 9 292 292
io-bound: 10 292 292
io-bound: 11 292 292
io-bound: 12 292 292
io-bound: 13 292 292
```

Trial 3:

```
cpu-bound: 4 165250730 5810
cpu-bound: 5 165565948 5819
cpu-bound: 6 165821563 5825
cpu-bound: 7 165821967 5825
cpu-bound: 8 166293126 5842
io-bound: 9 292 292
io-bound: 10 292 292
io-bound: 11 292 292
io-bound: 12 292 292
io-bound: 13 292 292
```

The CPU-bound processes receive tremendous more time, thus incrementing x more. This makes sense for a fair scheduler. Also, 2 cpu-bound processes (4 and 5) seem to not have a cpu usage divisible by 5 therefore suggesting other interrupts (clkhandler waking up processes) put the io-bound processes ahead.

5.5

Hypothesis: The first process will have incremented X the most, the second process will have incremented X 2nd most, so on and so forth.

Trial 1:

```
cpu-bound: 4 349370565 12274
cpu-bound: 5 241920283 8499
cpu-bound: 6 218452622 7674
cpu-bound: 7 241960760 8500
cpu-bound: 8 350024215 12274
```

Trial 2:

```
cpu-bound: 4 349370939 12274
cpu-bound: 5 241920326 8499
cpu-bound: 6 218452447 7674
cpu-bound: 7 241960042 8500
cpu-bound: 8 350021718 12274
```

Trial 3:

```
cpu-bound: 4 349371645 12274
cpu-bound: 5 241920212 8499
cpu-bound: 6 218452614 7674
cpu-bound: 7 241960359 8500
cpu-bound: 8 350023919 12274
```

These results shocked me immensely. I didn't realize that by the time the process 8 was created, process 4 would have passed the amount of time it was given before termination. The reason process 6 has the least increments of x and the lowest cpu usage time, is because it has to constantly fight against 2 other processes first 4 and 5, then 5 and 7, then 7 and 8.