Chris Cohen
CS354 Spring 2020
Lab 3

3 - Kernel instrumentation: monitoring process aging and CPU usage

3.1

a) Since clktimemilli is of type uint32, the maximum value is the maximum value that an unsigned 32-bit integer can have, which is **(2^{32} - 1) = 4,294,967,295 milliseconds.** If we convert that many milliseconds to days and hours, we get **roughly 49 days and 17 hours** until the counter overflows.
   i) For a 64-bit millisecond counter, the maximum value is **(2^{64} - 1) = 18,446,744,073,709,551,615 milliseconds.** If we convert that many milliseconds to days and hours, we get **roughly 213,503,982,334 days and 14 hours** until the counter overflows.

3.2 → procbirth.c, procbirth.h, proclifetime.c, proclifetime.h

3.3 → currentgrosscpu in resched.c/clkinit.c, procgrosscpu.c

3.4

a) Adding instrumentation code before and after calling ctxsw() in resched() works because currentgrosscpu is initialized to clktimemilli when the new process starts after ctxsw() is called in resched() (aka the new process's stack is switched to being the runtime stack), and clktimemilli keeps running. When the process is context switched out using ctxsw(), its prgrosscpu time is recorded as being clktimemilli - currentgrosscpu. Since clktimemilli has been keeping track of time, and currentgrosscpu was the timestamp that the process was started, subtracting the two gives you the amount of CPU time that the process used. Redefining currentgrosscpu after ctxsw() doesn't mess up the previous process because the previous running process is no longer is running, and no longer is dependent on that value of currentgrosscpu - the new process now uses it.

b) **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.
   i) currentgrosscpu for the current process is already initialized in this situation, either from the prologue function or right as it is ctxsw()'d in in resched(). When sleepms() is called, it puts the current process to sleep and reschedules, causing it to be ctxsw()'d out, stopping and updating the current process's prgrosscpu. Then, the new process is ctxsw()'d in, starting the timer, and is run until later stopped.

c) **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.
   i) currentgrosscpu for the current process is already initialized in this situation, either from the prologue function or right as it is ctxsw()'d in in resched(). When a clock interrupt is raised, the time slice is decremented. If the time slice has run out, resched() is called. If the process at the front of XINU's ready list has a higher priority than the current one, then the current process is ctxsw()'d out for the new process. Right before that, the current process's currentgrosscpu timer stopped and its prgrosscpu field is updated, then currentgrosscpu is redefined to clktimemilli to keep track of the CPU time of the new process. If not, the time slice is renewed, and the process and timer continue from where they left off.

d) **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.
   i) currentgrosscpu for the current process is already initialized in this situation, either from the prologue function or right as it is ctxsw()'d in in resched(). After the clock interrupt is raised and the newly ready process is placed into the ready list. If the newly awoken process has priority greater or equal than the current process, the timer for the current process is stopped and its prgrosscpu field is updated, then currentgrosscpu is redefined to clktimemilli to keep track of the CPU time of the new process. If not, the process and timer continue from where they left off.

e) For this lab, my prologue function is start_timestamp.c, and my epilogue function is end_timestamp.c. start_timestamp() does the same thing as after ctxsw() - it initializes currentgrosscpu to clktimemilli. end_timestamp() does the same thing as before ctxsw() - it adds the additional currentgrosscpu time for that process to its 'prgrosscpu' entry in the process table. To make sure that these functions always run, I manipulate the runtime stack so that they are called. Initially, in create(), the (important part of the) stack is laid out like this:

INITRET (userret, aka kill() → kills the process)
funcaddr (address of the function to run as a process)

This makes the return address of create() funcaddr since it's on the bottom of the stack. After funcaddr() is called, it returns to INITRET (since it's on the bottom of the stack), which kills the process. I want start_timestamp and end_timestamp to be called before/after funcaddr, so I set up the stack like this:

INITRET (userret, aka kill() → kills the process)
end_timestamp
funcaddr (address of the function to run as a process)
start_timestamp

By the same logic as the previous paragraph, where the address on the bottom of the stack is the return address, this means that the flow is:
start_timestamp → funcaddr → end_timestamp → INITRET (kill),
which is what we want in order to accurately measure currentgrosscpu for a newly created function.

3.5

a) procgrosscpumicro() tends to be in the ballpark of prgrosscpu(), but not completely accurate. First, prgrosscpu rounds the time if it is over 1 millisecond, so if 5.5 milliseconds have passed, 5 milliseconds are recorded for prgrosscpu, and 5500 microseconds are recorded for prgrosscpumicro. This adds up over time. However, prgrosscpu also rounds up to 1 millisecond if it's used for less than that time. Therefore, if a 300 clock ticks occur for the process, it will round up to 1 millisecond, but only 0.3 milliseconds have passed. This will also add up over time as it occurs more.

4 - Dynamic priority scheduling

4.2 → rinsert.h, rinsert.c

a) To make smaller integer values in the priority queue mean higher priority,
   i) Change insert() in resched() and ready() to rinsert()
   ii) In resched(), when possibly ctxswitching or not (around line 27), it checks to see if the first process's priority in the ready list is greater than the current process's priority. If that's the case, it keeps running the current process. This was used for the original scheduling. We must change it to < from >, because if the currently running process has a lower priority value than the first process in the ready list, it should keep running.
   iii) In newqueue(), switch MAXKEY and MINKEY since the MINKEY should be the first value (first run)

4.3

a) Each process's *prvgrosscpu* field is incremented for every millisecond that the process runs, growing in parallel (but not necessarily equal) to its *prgrosscpu* field. To ensure that the null process is always at the end of the ready list, it must always have the highest value for *prvgrosscpu* out of all processes in the ready list. If the initial value of the null process's *prvgrosscpu* is set to the maximum value for a uint32 type, which is 4,294,967,295, it will overflow when the null process runs again and cause the null process to be (likely) at the front of the ready list with a very small *prvgrosscpu* value. Therefore, to fix this problem, the null process's *prvgrosscpu* field should be initialized to 0 as normal and incremented for every millisecond. This will make it so that no process can ever eclipse the null process's *prvgrosscpu* field, and make it so that the field doesn't overflow (as quickly). If a process happens to have their *prvgrosscpu* set from the null process's *prvgrosscpu* field, I make sure to initialize it to the value minus one so that the null process has a strictly higher value (lower priority). Also, when inserting a process into the ready list, I make sure that it can't be put after the null process, even if a coding mistake occurs and the null process happens to have a lower *prvgrosscpu* value.

4.4

a) To go about ensuring that the null process's priority is always strictly less than the priority of all other processes in the system, I follow the method listed in 4.3. However, if 4.3 is followed exactly, the null process will have the same priority as a new process created when there are no other processes in the XINU readylist, as specified in (i) on the lab handout. To prevent this in this situation, and make the null process's priority <u>strictly</u> less than all other processes, I set the newly created process's *prvgrosscpu* field to the null process's *prvgrosscpu* field minus one.

b) Changes Made:

i) For every call of rinsert, insert with the prvgrosscpu field as the key instead of prprio.

ii) create max_ready_val() and min_ready_val() functions to compute max/min *prgrosscpu* values in the readylist, according to specifications in the handout. This is written in queue.c.

iii) *prvgrosscpu* for a given process is updated everywhere that the process's *prgrosscpu* is updated.

iv) In clkhandler.c, increment the null process's *prvgrosscpu* value every time the clock interrupt is called.

v) In create.c, set new process's *prvgrosscpu* field to the maximum prvgrosscpu value (max_ready_val()) in the readylist, as specified in the handout.

vi) In ready.c, if the process is being readied from a sleeping state (after relinquishing control of the CPU via sleepms()), make its new *prvgrosscpu* field equal to the minimum prvgrosscpu value in the ready list (min_ready_val()).

vii) In rinsert.c, I make the readylist be sorted in increasing priority (lowest *prvgrosscpu* first). I also have a check to make sure that even if the null process for some reason has a lower *prvgrosscpu* value, the new process can't be put after the null process in the list.

5 - Evaluation of XINU Fair Scheduler

5.2

a) As you can see below, the 4 processes all print similar CPU usage and x values. All 4 of these processes are created with priority equal to the prvgrosscpu value of the null process. Also, all of them are CPU-bound, so there is no special treatment. Resched() switches processes if the first value of the readylist has a lower or equal value for *prvgrosscpu*. Therefore, one process runs for QUANTUM, then its *prvgrosscpu* field is greater than the first process in the readylist, and it switches to that new process. This repeats until all of the CPU-bound processes are done. Essentially, it does round-robin scheduling for all 4, meaning the CPU usage will be similar for all of them. They have similar values for x because they execute for the same amount of time, so roughly the same amount of increments for x.

   TEST 1:
```
cpu: 3 53399041 2010 2004730
cpu: 4 53404974 2010 2004931
cpu: 5 53801379 2010 2004924
cpu: 6 54136904 2010 2004930
```
   TEST 2:
```
cpu: 3 53399122 2010 2004724
cpu: 4 53404947 2010 2004930
cpu: 5 53801380 2010 2004924
cpu: 6 54136900 2010 2004930
```

5.3

a) As you can see below, the 4 processes all print similar CPU usage and x values. All 4 of the processes are created with priority of the *prvgrosscpu* value of the null process. It starts off pretty similar to 5.2, but uses sleepms() to make sure each process is I/O bound. The first of these processes starts running, increments x, and then sleepms() is called. This means that this process is no longer running (asleep), and the first process in the readylist is run. That process does the same thing - sleeps, and lets the next process run. Every time that one of these processes is done sleeping, it wakes up, is assigned the highest priority (lowest *prvgrosscpu* value), and is run instantly since it has the highest priority in the readylist. The gist is that the processes sleep, let other processes run, and then wake up and are instantly given the CPU, as I/O-bound processes should be. The x value makes perfect sense, since 8000/50 = 160, where 8000 is the number of ms that the process runs, and 50 is the number of ms that each iteration of incrementing x sleeps for, so the total number of x increments is 160.

TEST 1:
```
io: 3 160 160 36285
 io: 4 160 160 36218
io: 5 160 160 36310
io: 6 160 160 36308
```
TEST 2:
```
io: 3 160 160 36286
io: 4 160 160 36218
io: 5 160 160 36312
io: 6 160 160 36309
```
5.4

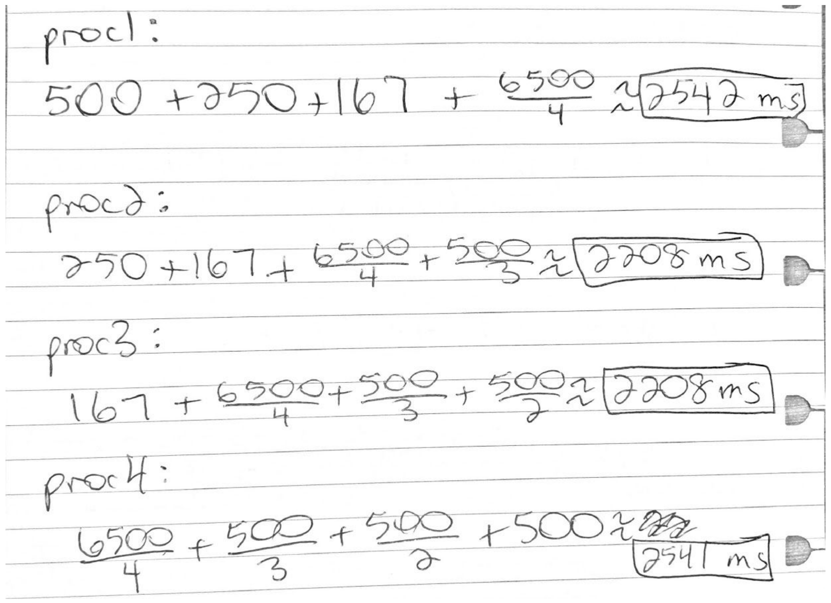
a) As you can see below, the CPU-bound processes have similar x and CPU usage values relative to each other, as do the I/O-bound processes relative to each other. The reasons for that are discussed in 5.2 and 5.3. The I/O-bound processes have smaller x values than the CPU-bound processes because of the sleepms() call delaying x being incremented. We can also see that the CPU-bound processes received significantly more CPU time than the I/O-bound processes. This is because when the I/O bound processes sleep, they relinquish the CPU because they don't need it to function for the time being. CPU-bound processes, on the other hand, need as much CPU time as possible to finish as fast as possible, which is shown clearly here. The testcpu() process never sleeps, and always needs the CPU to keep going.

TEST 1:
```
cpu: 3 52717820 2004 1979497
cpu: 4 52971794 2004 1982649
cpu: 5 53431548 2033 2006285
cpu: 6 53192097 2050 2012586
io: 7 99 99 22439
io: 8 99 99 22449
io: 9 99 99 22452
io: 10 99 99 22466
```
TEST 2:
```
cpu: 3 52716257 2004 1979413
cpu: 4 52971916 2004 1982650
cpu: 5 53431545 2033 2006285
```

```
cpu: 6 53192079 2050 2012585
io: 7 99 99 22439
io: 8 99 99 22448
io: 9 99 99 22452
io: 10 99 99 22466
```

5.5

a) My equations to estimate CPU usage for each process are below:



proc1:

$$500 + 250 + 167 + \frac{6500}{4} \approx \boxed{2542 \text{ ms}}$$

proc2:

$$250 + 167 + \frac{6500}{4} + \frac{500}{3} \approx \boxed{2208 \text{ ms}}$$

proc3:

$$167 + \frac{6500}{4} + \frac{500}{3} + \frac{500}{2} \approx \boxed{2208 \text{ ms}}$$

proc4:

$$\frac{6500}{4} + \frac{500}{3} + \frac{500}{2} + 500 \approx \cancel{[illegible]} \boxed{2541 \text{ ms}}$$

The tests come out to be reasonably close to my estimations (looking at *procgrosscpumicro* field), within margin of error/overhead and because the scheduling happens in 30ms intervals, rather than completely even 1ms intervals. What happens is that the first process gets 500ms to itself when main sleeps. The second time main sleeps, there are 2 processes to share the 500ms (250ms each). The third time, there are 3 processes to share the 500ms (167ms each). After that, the first process has 6500ms left to run, so all 4 share that 6500ms (6500/4 ms each). After that, the second process has 500ms left to run, so the remaining 3 share that (500/3 ms each). After that, the third process has 500ms left to run, so the remaining 2 share that (500/2 ms each). After that, the fourth process has 500ms left to run, which it doesn't have to share.

TEST 1:

```
cpu: 3 73858263 6839 2525383
cpu: 4 65015024 2430 2423882
cpu: 5 64776860 2430 2423886
cpu: 6 93969915 44850 1765011
```

TEST 2:

```
cpu: 3 73858106 6839 2525295
```

```
cpu: 4 65015026 2430 2423882
cpu: 5 64776867 2430 2423886
cpu: 6 93969899 44850 1764981
```

BONUS

a) What is a potential weakness of method (i), and how would you improve on it?
   i) This process may be very important, but it will not be run right away because it isn't necessarily at the front of the readylist. To improve upon this, I would make another process table field to specify if the process should be run right away. If this is the case, I would set the *prvgrosscpu* field to the minimum priority in the readylist. This would make it get context-switched out immediately, as well as get a bit of extra time to run, since it doesn't have to wait for the other processes to catch up in CPU usage before starting.

b) What is a potential weakness of method (ii), and how would you improve on it?
   i) You may have a very important process running that you don't want interrupted, even by an IO bound process that becomes ready. To improve upon this, I would make another process state constant to specify if the process shouldn't be interrupted. I would also have another process table field to specify whether or not the process is important or not. If it is important, I would set the state (while running) to this new process state constant. Therefore, resched() would know not to context-switch it out until it's done.

c) What other issue does Linux CFS have compared to scheduling methods used in UNIX and Windows? Is this a big concern?
   i) You might want to give a certain process more CPU time over other processes, (in a non-fair way). This is where normal priority numbers come in handy. UNIX and Windows still follow Linux's old way of scheduling, where it picks a highest priority process to run next and you could change priority yourself. This is nice when you want to manually give a process more CPU time by increasing the priority. In XINU's case (now), every process gets fair scheduling with the others. This isn't a huge concern in normal, everyday environments. However, in an enterprise environment, you may have an urgent process to run that needs to be done as quickly as possible, and XINU would not be ideal for this.