Akash Lankala
CS 354
Project 3
March 5th, 2020

Using 2 Late days

Note: My implementation of rinsert.c has a while loop that is stuck in an infinite loop. This is causing the overall system to never reach main(), thus it would automatically not pass any type of test cases. We would not see any kind of output either.
However, the code is very documented, especially the implementation of rinsert.c, with the other functions and variables implemented. Files like newqueue.c were changed as well to accommodate the priority changes implemented in rinsert.c (but still for some reason did not fix the infinite loop issue). Please see functions implementation for partial credit.

3. Kernel Instrumentation: monitoring process aging and CPU usage

3.1 Millisecond resolution system uptime

Check modifications in `clkinit.c` *and* `clock.h`

How long (in days and hours) `clktimemilli` can keep time after bootloading until the counter overflows? Truncate fractional hours.

It can keep time for about 49 or 50 days. $2^{32}/(1000 * 3600 * 24)$.

For how long can a 64-bit millisecond counter keep time?

About 213 *billion* days.

3.2 Process birthday and lifetime

Check new process table field `uint32 prbirth`.
Check new system call `uint32 procbirth(pid32 pid).` Check `procbirth.c` for additions.
Check `uint32 proclifetime(pid32 pid)` system call. Check `proclifetime.c` for additions.

3.3 Process gross CPU usage

Akash Lankala
CS 354
Project 3
March 5th, 2020

Check new process table field `uint32 prgrosscpu`.
See changes to `resched()` in `resched.c`.
Check millisecond counter `uint32 currentgrosscpu`
Check new system call `uint32 procgrosscpu(pid32 pid)` in `procgrosscpu.c.`

3.4 Rationale behind method for estimating gross CPU usage

Explain why adding instrumentation code before and after calling `ctxsw()` in `resched()` is a viable method for estimating a process's gross CPU usage.

To estimate a process's gross CPU, we need to monitor how much CPU time a process has consumed *including* the time that interrupts spend taking up CPU time. The rationale behind adding the instrumentation code before and after calling `ctxsw()` is intended to keep track of this amount of time a process spends taking up the CPU including interrupts.

To recall, a process becomes current when it is context switched in. This is what the code inserted after `ctxsw()` is keeping track of. To do this, we add code after `ctxsw()` to start a millisecond counter, uint32 currentgrosscpu, that is initialized to clktimemilli. This tracks gross CPU time consumed by the new process until it is context switched out. Code inserted before calling `ctxsw()` updates the current process prgrosscpu field in the process table and ultimately returns the gross CPU usage of a process. It does this by subtracting clktimemilli by currentgrosscpu. To recall, currentgrosscpu is initialized to clktimemilli when it is initially declared. clktimemilli keeps track of the system uptime in milliseconds. currentgrosscpu has a counter that increments every 1 millisecond. When the code executes clktimemilli – currentgrosscpu, it is essentially finding the difference which in other terms is the gross CPU time of the current process. This expression does not account for interrupts that borrow the current process's context, thus this represents the process's *gross* CPU usage, not *net* 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 -

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.

Akash Lankala
CS 354
Project 3
March 5th, 2020

`sleepms()` is the system call through which apps voluntarily relinquish CPU usage. Thus, that means they need to be context-switched out.
When `sleepms()` is called, `resched()` is called within `sleepms()` which then calls `ctxsw().`

In our instrumentation code, before we call `ctxsw()` in `resched(),` we update the current process's prgrosscpu field in the process table by adding clktimemilli – currentgrosscpu.

Then, `ctxsw()` is called which context switches in the new process. The previous process's gross CPU time was recorded before the process was context switched out with the update to prgrosscpu right before `ctxsw()` is called. Thus, when `sleepms()` is used to voluntarily switch out a process from the context, the prgrosscpu for that process is still updated before the new process gets context switched in.

Code is added after ctxsw() to start a millisecond counter, uint32 currentgrosscpu, that is initialized to clktimemilli. This tracks gross CPU time consumed by the new process until it is context switched out. Because we are tracking gross cpu usage, which includes the time interrupts spend in the process context, we initialize…
Code is inserted before calling ctxsw() to update the current process prgrosscpu field in the process table by adding clktimemilli – currentgrosscpu. In the case that clktimemilli = currentgrosscpu (this means that the current process has consumed less than 1 millisecond of CPU time before getting context switched out), round up and add 1 millisecond to prgrosscpu. Add a new system call, uint32 procgrosscpu(pid32 pid) in procgrosscpu.c that returns the process's gross CPU usage.

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.

If the current process that is executing has a higher priority than the next process in the front of XINU's ready list, then the current process will remain executing resched() will not trigger a context switch.

If the process that is executing's time slice decrements to 0, and resched() gets called and determines that the next process at the front of the ready list has a higher

priority, then the code before ctxsw() is run. The code before ctxsw() updates the current process's prgrosscpu field in the process table by adding clktimemilli – currentgrosscpu. Then ctxsw() get called, and when it returns, the new process becomes current as it is context switched in. At that point, a millisecond counter, uint32 currentgrosscpu that is initialized to clktimemilli tracks the gross CPU time consumed by the new process until it is context switched out.

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.

In this scenario, unsleep() gets invoked, and a similar sequence of function calls as in Scenario 2 get called. resched() gets called and executes a series commands to verify whether sleeping process has a priority greater to or equal to the current process. If so, the code before ctxsw() updates the current process's prgrosscpu field in the process table, ctxsw() changes the process context, and the code after ctxsw() (when it returns) initialized currentgrosscpu to clktimemilli which tracks the gross CPU time consumed by the new process until it is context switched out. Through these steps, XINU ensures that the interrupting process calling the sleeping process has a priority greater than or equal to the current process, saves the current process's CPU usage, and initializes the new (recently sleeping) process's CPU usage time to begin.

3.5 Improving accuracy of gross CPU usage estimation

Check new process table field uint64 prgrosscputick in `process.h`.
Check CPU tick counter uint64 currentgrosscputick in `resched.c`.
Check system call uint32 `procgrosscpumicro(pid32 pid)` in `procgrosscpumicro.c`.

procgrosscpucmicro() counts microseconds which is expectedly a higher granularity than prgrosscpu().

4. Dynamic priority scheduling

4.1 Objective of fair scheduling

Akash Lankala
CS 354
Project 3
March 5th, 2020

4.2 XINU compatibility

Specify what changes are needed to impose a new world order in XINU with respect to priority where a smaller integer value means a higher priority. Specify what these changes are and how to implement them in XINU. This includes how you handle XINU's null process.

See `rinsert.c` for the modified kernel function. `rinsert()` inserts processes in ascending order of priority into XINU's ready list.

Changes were made in `ready.c`, `resched.c`, and the header file `prototypes.h` to include `extern status rinsert(pid32, qid16, int32);`.

We handle the NULL's process by initially checking the following conditions:

1) If process being passed in is the NULL Process
2) If the process being passed in is a NEW process
3) If the process being passed in is an EXISTING process

If process is a NEW process (with a CPU usage value prvgrosscpu == 0)
If it is NULL Process, we simply insert it as the first process in the ready list.

If process is NOT the null process, but it is a new process, we iterate through the ready list queue until we reach the element right before the null process (which should be the last proc in ready list) and then insert the new process right before the null process in the ready list.

4.3 Linux CFS scheduler

Explain how to ensure that 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.

Question: to set the initial priority when a process is created, how to set to maximum cpu usage across all ready processes?

How to find (ii), how to find blocked process logic. ready.c.

Akash Lankala
CS 354
Project 3
March 5th, 2020

4.4 XINU fair scheduler implementation

We can ensure that the null process, when not current, is always at the end of the ready list by ensuring that the gross cpu usage of the null process always exceeds every other process. This is because since we are now sorting the ready list in ascending prgrosscpuusage values, we can have the null process be in the end by setting a value that will make the null process definitely have the greatest gross cpu usage value. We can set it to clkmillitime to make sure there is no other process with a greater cpu usage time.

Check prvgrosscpu process table field.

5. Evaluation of XINU fair scheduler

Check files `testcpu.c` and `testipu.c.`