Dan Cassidy
CSCI-C 435
Homework 2

Problem

Modify the MINIX 3 scheduler to keep track of how much CPU time each user process has had recently. When no task or server wants to run, pick the user process that has had the smallest share of the CPU.

What I Did

In summary, I expanded the `proc` structure, incremented a counter on every clock tick for the running process, modified the `pick_proc` function to run system processes first then the user process with the least amount of recent CPU time, and allowed the user to display a small matrix of the recent time and some other data by hitting the F12 key.

Here is what I did in simple list form in chronological order, basically akin to a rolling changelog. Explanations and the process will come later in the "Approach" section.

- Expanded `proc` struct with `clock_t p_recent_time` to keep track of the recent CPU time a process has consumed. [kernel/proc.h]
- <Test point 1>
- Added initialization for `p_recent_time` to `main()`. [kernel/main.c]
- Added reinitialization of `p_recent_time` to `do_fork()`. [kernel/system/do_fork.c]
- <Test point 2>
- Added prototype for `recent_dmp()`. [servers/is/proto.h]
- Increase the number of hooks, and add `recent_dmp()` to F-key function list. [servers/is/dmp.c]
- Copied `privileges_dmp()` to serve as the base of `recent_dmp()`. [servers/is/dmp_kernel.c]
- Modified `recent_dmp()` to display required information. [servers/is/dmp_kernel.c]
- <Test point 3>
- Added increment for `p_recent_time` to `clock_handler()`. [kernel/clock.c]
- <Test point 4>
- Added decay code for `p_recent_time` to `balance_queues()`. [kernel/proc.c]
- <Test point 5>
- Rewrote `pick_proc()` to follow guidelines of assignment. [kernel/proc.c]
- <Test point 6>

How To Run It

- Install virtual appliance.
- Start virtual appliance.
- Allow Minix to load on its own by NOT using any of the 3 options that appear during boot.
- Log in.
- Use command "cd src".
- Run commands "./testa &" and "./testb &".
- Hit 'F12' and observe the recent CPU ticks (among other data).

- Continue to hit 'F12' and observe that the test processes will usually be in sync, with testa beating out testb whenever testb goes to sleep until shortly after testb wakes up.

Approach

The first goal was to get some way to track the recent CPU time of a process. To make that happen, I expanded the `proc` struct by adding a variable named `p_recent_time` of type `clock_t`. Given that this number will be incremented and modified mathematically, it needs to be initialized, so I added a line of code in `main()` to do so. Recent CPU time also falls under the general time tracking of a process, so I also zeroed it out every time a process forks to keep in line with the other time tracking variables, `p_user_time` and `p_sys_time`.

In order to be able to see and keep track of the recent CPU time, I built a small dump function (`recent_dmp`) in the information server using the privilege dump function as a base, and attached it to the F12 key. This function is used to display the process ID, privilege ID, name, privilege flags, `priv` struct memory address, and the recent CPU time of all the currently running processes.

Once this was added, I decided it was time to actually start modifying the kernel to keep track of things. The `clock_handler` function in [kernel/clock.c] was where I chose to increment the recent CPU time counter, since this is also where the user and system time are incremented. In order to keep recent CPU time recent, I took inspiration from the class discussion about paging on demand and modified the `balance_queues` function to add a bitshift-based decay to the recent time variable. In short, every time `balance_queues()` is called, the recent CPU time for all processes gets bitshifted to the right by 1. I chose this method over doing a hard reset to 0 as it would provide an actual rolling account of recent CPU time, and would also work in tandem with the modified `pick_proc` function to mimic the bump in priority traditionally given by `balance_queues()`.

Deciding on how to modify the scheduling apparatus in the kernel probably took me the longest to settle on a method. I thought about a couple different methods which I shall briefly overview. One method was reducing the number of queues to one for kernel tasks, one for server processes, one for user processes, and one for the idle process, but that would have involved fairly extensive modifications to the kernel and would have negated the use of priority queues for the system/server processes. The second method was to modify the `sched` and `enqueue` functions to ensure that system processes were always at the head of their priority queue, then placing the user process with the least amount of recent CPU time directly next, with the user process with the next least amount of recent CPU time next, and so on. I scrapped this idea because again, it would have required extensive modification to the kernel to pull off effectively. I'm a big fan of the "work smarter, not harder" mentality.

I ultimately arrived at an idea to essentially utilize a hybrid solution. Kernel tasks and system/server processes would still be governed purely by the priority queue method, and user processes partially so. This was achieved by pretty much rewriting the `pick_proc` function. The new function scans through the entirety of the scheduling queues with the exception of the idle queue in the same manner as before, that is highest priority (0) to lowest (14), head to tail. But while doing so, it checks every process for the SYS_PROC flag, which if found will immediately set that process to run next. If the SYS_PROC flag is not found, this means the process is a user process, so it checks to see if the process's recent CPU time is less than the current low, whatever it may be. If the recent CPU time of the process is lower than the current low, the function stores the pointer to the process and the new low, then continues

scanning. This synergizes nicely with the priority queue system in that if two user processes have each had the same amount of recent CPU time, the higher priority process will be chosen first, and can't be supplanted by the lower priority process because the test is "less than", instead of "less than or equal to". If the function finishes scanning the scheduling queues (meaning that no kernel task or system/server process wanted to run), the function then checks to see if there are any user processes that want to run, that is, if `low_rp` is not `NIL_PROC`. If there is a user process waiting to run, then it gets run, but if not, the function attempts to run the idle process. If the idle process isn't there for whatever reason, a kernel panic is induced, just like in the original `pick_proc` function.

Testing

Testing in this project was a bit rough, as determining whether the processes were in sync was done by hand via using the F12 debug function.

One of the bit helps on this project were two programs I made called "testa" and "testb". The testa program utilizes a `while(1)` loop that simply adds 1 and 1, while the testb program counts to 3 million, then sleeps for 1 second and then repeats.

- Test point 1: Recompiled the Minix kernel and rebooted into the new image to ensure that the kernel was still functional after the `proc` struct was expanded. After the kernel was verified functional, the user commands were also recompiled in order to get programs like ps and top working again.
- Test point 2: Recompiled the Minix kernel and rebooted into the new image to again ensure things didn't hang and that the OS actually booted. Nothing to test beyond that.
- Test point 3: I simply recompiled the servers and hit F12 to see that the added dump function was working fine.
- Test point 4: For this, I recompiled the Minix kernel and rebooted into the new image as per usual, and utilized the F12 dump to ensure that the recent time is increasing as there is not yet a decay or reset added.
- Test point 5: Recompiled and booted into a new Minix image, and observed that the recent time was clamped to a max of around 200. At this point I ran the "testa" and "testb" programs mentioned in the testing summary and observed that testb consistently trailed testa in recent CPU time, usually by a significant amount (10 or more).
- Test point 6: Recompiled and booted into a new Minix image, and ran testa and testb again, this time noting that they were approximately equal with testa jumping ahead when testb went to sleep and shortly after it woke up.

What Was Difficult

The most difficult part of this assignment was determining a way to modify the scheduling, enqueueing, and/or process picking mechanism(s) in such a way that wouldn't require a near total rewrite of all of those functions. As explained in the Approach section, I finally decided on the hybrid method, but that took a lot of thought and running through the various scenarios in my head to ensure it wouldn't break things.

What Was Easy

Simply keeping track of the recent CPU time was pretty simple utilizing the knowledge gained from the last homework.