Assignment 3: Scheduling and Copy-on-Write in xv6

Out: October 29, 2024
Final phase Due: 9:00 PM, November 19, 2024

This assignment should be done *in teams of 2 (two) people*

TOTAL MARKS: 100

Introduction

For this assignment, you are going to use xv6 and implement a fair share scheduler and copy-on-write semantics for parent/child processes.

Part A: Implement fair share scheduling algorithm (40 marks)

The existing scheduler in xv6 is very simple. On each timer interrupt, the interrupt handler switches to the kernel scheduler, which then selects the first available process to run. This scheduler, while simple, is too primitive to do anything interesting.

In this assignment, you'll implement an advanced scheduler that schedules processes based on fair shares of the CPU. Each process has a share, and the scheduler must ensure that the number of quanta a process has been allocated stays *near* its predefined share. Note: to simplify the assignment, you may assume all processes are single-threaded. If you wish to use multi-threaded processes, that is fine, just know that it may be a bit more complicated.

Your queues should only store processes in the RUNNABLE state. For example, if a process calls `sleep()`, it should not stay in the ready queue, until it is woken up. The scheduler should put processes just created or woken up to the back of the corresponding ready queue.

You will likely run into trouble if you use your list library from A1 unmodified, so a simpler structure is advisable, as long as it satisfies the requirements of picking the next process to run, based on the priority calculated by fair share requirements. You are welcome to use your list implementation, but only if you are very confident in its operation.

For the purposes of guiding the grader, mark any changes you make to the code with the following comments:

```
/* CMPT 332 GROUP XX Change, Fall 2024 */
```

How to test the scheduler? Write a user program that calls `fork()` to generate children and have the children sleep for random amounts of time and then compute `square()` for a random number of calls. There is a random number generator in `usertests.c` that can be used. There is no concept of "user" in xv6, so each process will have shares associatied with it. It is up to your design how to allocate shares, so as long as you can verify that some "group" of processes as you have defined it gets a "share" of the CPU as you have defined it.

BONUS (10 marks): As an extension/bonus, implement process groups and differential sharing, so that not every group gets the same share. This will require a new system call `setshare()` that you are free to design in any way you see fit. This allows you some amount of creativity in the design of your solution. If you can justify the design and show that your implementation achieves this design, you are free to experiment.

We've given guidelines and will be able to give hints on approaching the implementation. It will not be a lot of code, but it could be tricky to get correct.

Part B: Implement Copy-on-Write Fork with xv6 (60 marks)

The goal of this part of the assignment is to understand memory management in operating systems. We will achieve this goal by implementing the copy-on-write fork feature in xv6. This implementation will require you to thoroughly understand how memory management and paging work in xv6. It is borrowed substantially from MIT's labs for Fall 2023. This will be done in several steps.

1. Add a system call `getNumFreePages()` to retrieve the total number of free pages in the system. This system call will help you see when pages are consumed, and can help you debug your CoW implementation.
2. Add a kernel data structure that keeps track of the reference count of pages, and functions to increment and decrement these counts. The reference count of a page is set to one when it is allocated, and is incremented every time a new child points to the same page. The reference count is decremented when a process no longer points to it, say, after acquiring its own copy of the page. A page can be freed and returned to the free pool only when no other process is pointing to it. Carefully think about where this count is incremented and decremented, and make sure you do these changes to the count with proper locks held.
3. You must change the `uvmcopy` function called from fork to implement `CoW`. When you fork a child, the page tables of the parent and the child should point to the same physical pages, and these pages must be marked read-only. Given that the parent's page table has changed (with respect to page permissions), you must reinstall the page table and flush TLB entries, as described above. This function is one place where you may have to increment reference counts of kernel pages.
4. When the parent or child processes attempt to write to a page marked read-only, a page fault occurs. The trap handling code in xv6 does not currently handle the page fault exception. Add a trap handler to handle page faults. You can simply print an error message initially, but eventually this trap handling code will call the function that makes a copy of user memory.
5. The bulk of your changes will be in this new function you will write to handle page faults. When a page fault occurs, the stval register holds the faulting virtual address, which you can get using the xv6 function call r_stval() You must now look at this virtual address and decide what must be done about it. If this address is in an illegal range of virtual addresses that are not mapped in the page table of the process, you must print an error message and kill the process. Otherwise, if this trap was generated due to the CoW pages that were marked as read-only, you must proceed to make copies of the pages as needed.

   Note that between the parent and the child, the first one that tries to write to a page should get a new memory page allocated to it. This new page's content must be copied from the contents of the original page pointed to by the virtual address. Even after this copy is made, note that the page is still marked as read only in the page table of the second process, and it will soon trap as well when it attempts to write to the read-only page. When the second process traps, no new pages need to be allocated; it suffices to remove the read-only restriction on the trapping page, since the first process already has its copy. Your page fault handling code should distinguish between these two cases using the reference count variable, and handle them suitably. Make sure you modify the reference counts correctly, and remember to flush the TLB whenever you change page table entries.

6. Finally, you must integrate the test program `cowtest` that tests the copy-on-write implementation.

Deliverables - What to hand in:

- tarball of your kernel source, named xv6-A3.tar, without the git details.
- `xv6-A3.diff`: a diff file listing all the changes to the kernel you made for this assignment.

Part A: Scheduler in xv6

- Design document in PartA.Design.txt. Description of the implementation of the scheduler and where you had to make changes to the kernel and for your scheduler tests.
- Test execution of your modified kernel in `xv6-scheduler-output.txt.` This will include runs of your test programs.

Part B: Copy on Write Fork in xv6

- Design document in PartB.Design.txt. Description of the implementation of copy-on-write and where you had to make changes to the kernel
- Test execution of your modified kernel in `xv6-copy-on-write-output.txt.` This is where the results of the sanity test program will be submitted.

Note that Part B will depend on Part A in the fact that you will use the FS scheduler when you implement the copy-on-write. Document clearly how to verify that your test programs show that your kernel modifications are correct.

Finally, proper Git logs are required as always.

All of this should be bundled in a .tar file and submitted via Canvas with no subdirectories for all of the design and testing files, right?? The tar file of the main source tree for xv6 source code, is again to be the `xv6-riscv` directory. So, yes, that's a .tar file that contains a .tar file.

No .o files; source code and documentation only.

Notes and Warnings:

Documentation: All documentation files are to be in UNIX text file format with lines shorter than 80 characters. Nothing else is acceptable; nothing else is necessary. There will not be diagrams, etc. in your work. This allows the marker to look at both your code and your documentation in the same tool: vi or emacs. It speeds the marking and a happy marker is a generous marker.