CS 4375 Fall 2025
Homework 2: Time Command
75 points

MD Armanuzzaman

Dude Date: October 6, 2025 (11.59 PM))

For this assignment, you will implement a time command for xv6 that takes a command and the command's arguments, executes the command, and, when the command has finished, outputs the CPU time, elapsed (also called wallclock) time, and the percentage of the CPU the process had while it was in the system. Your time command output will be similar to the `/bin/time` command in Linux, except that you do not need to keep track of the user and system portions of the CPU time separately, nor do you need to report memory usage information. Type `man time` on a Linux system for more information about the Linux time command.

Implementing the time command will help you to collect data to use for evaluating the scheduler you will implement for HW3. It's also useful to the user to be able to find out the CPU time and elapsed time for their program. In doing this assignment, you will also learn how to implement a new system call, the wait2() call that will be needed to get CPU time.

Important note: For this homework and the next one, we want to use just one CPU so that all processes are scheduled on the same CPU. To make this happen, in the Makefile, change `CPUS := 3 to CPUS := 1`.

You should create a hw2 branch in your xv6 repo for this assignment.

Task 0. Add the given user programs to xv6. (0 points)

You are given `matmul.c` and `sleep.c` programs that you can add to the xv6 user programs and use for tests.

You can use these two programs throughout the homework for testing your implementations.

Task 1. Implement a time1 command that reports just elapsed time. (25 points)

Your `time1` program should output a usage message and exit if the user does not enter a command to be timed. For this task, you will need to do the following:

- Handle the command-line arguments input by the user
- Assemble the arguments to pass to `exec()`
- Use `exec()` in the child to execute the program passed as a command to `time1`
- Call `wait()` in the parent to wait until the child finishes
- Get the elapsed time by calling the `uptime()` system call in the parent before forking the child and after the wait returns

Sample output for Task1 follows below:

```
$ time1 matmul
Time: 28 ticks
elapsed time: 28 ticks
$ time1 matmul &; time1 matmul &
$ Time: 29 ticks
elapsed time: 30 ticks
Time: 54 ticks
elapsed time: 56 ticks
$ time1 sleep 10
elapsed time: 10 ticks
```

Task 2. Keep track of how much *cputime* a process has used. (20 points)

Modifying process structure and calculating `cputime`

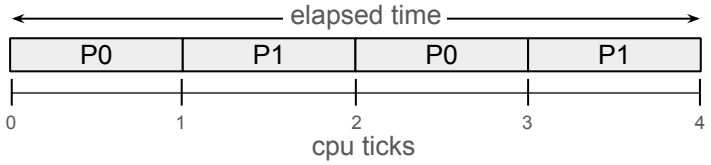

Figure 1: Elpased time and cputime of two processes

Defining *cputime*
By *cputime* we mean the total time a process uses the CPU, which may consist of multiple CPU time slots. In the task, your goal is to count the total time the OS scheduler may

schedule a process in the CPU.

For example, let's consider you have two processes running in the CPU, `P0` and `P1`. You need to calculate elapsed time and *cputime* for each process. As shown in Figure 1, the OS scheduler schedules `P0` for the first CPU tick, `P1` for the second CPU tick, `P0` for the third, and `P1` for the fourth CPU tick. The elapsed time for the two processes here would be 4 CPU ticks. While the *cputime* for each process will be 2 CPU ticks.

Relevant files and functions
In xv6, the `proc` structure in `proc.c` store all relevant information regarding each process. The `allocproc` function is executed when a process is created and is responsible for allocating pids, pagetables (i.e, memory), context, etc. The `trap.c` file contains the functions to trap all kernel and user traps, such as syscalls, faults, and interrupts. For example `usertrap` function invovked from `trampoline.S` to handle an interrupt, exception, or system call from user space (i.e., user processes). In case of a timer interrupt, the current user process gives up the CPU, and the scheduler can decide which process to schedule next.

Note that, in the given implementation, xv6 does not keep track of the *cputime*. In this task, you need to do the following:

- Modify `proc` structure to keep track of *cputime* (Hint: variable)
- When a process is created, initialize the variable
- When there is a timer interrupt, increment the *cputime* count (Hint: your goal is to count **user** process *cputime*, look for code when the timer interrupt is trapped for the user process)

Implementing wait2 system call and other user utility programs

- Understand the whole workflow of implementing system calls in xv6, for example `wait()`. Also refer to the important notes below these instructions reagading system call parameters.
- Create a file `kernel/pstat.h` with the following contents:

```
struct rusage
{
    uint cputime;
};
```

- Add the following line to `user/user.h` so that you can use `struct rusage` in user code:

```
struct rusage;
```

- Create a system call with the following prototype (put this in `user/user.h`):

```
int wait2(int*, struct rusage*);
```

- Add an entry for wait2() in `user/usys.pl`
- Add wait2 system call information to `kernel/syscall.h` and `kernel/syscall.c`
- Implement `sys_wait2()` in `kernel/sysproc.c` and wait2() in `kernel/proc.c` so that `sys_wait2()` returns child process status and rusage, with rusage currently being only cputime. You can copy `sys_wait()` to `sys_wait2()` and wait() to wait2() and add the necessary code to return the second argument (Hint: In the previous subtask you already have the *cputime* in `proc` structure). Add the kernel wait2() prototype to `defs.h` so that is visible outside `proc.c`. Add `#include "pstat.h"` to `proc.c` so that your kernel wait2() can use `struct rusage`.

Important notes on implementing system calls and their parameters
For this discussion let's condsider `wait` system call implementation. The prototype of `wait` system call in `user/user.h` are as follows:

```
int wait(int*);
```

The corresponding system call wrapper in `kernel/sysproc.c` are as follows:

```
uint64
sys_wait(void)
{
        uint64 p;
        if(argaddr(0, &p) < 0)
        return -1;
        return wait(p);
}
```

Note that the argument of the system call is changed from `int *` to `uint64` through the `argaddr` function call, where the `0` indicates the first argument. The argument of wait is basically a user-space pointer that **should never be dereferenced** in kernel space. You should always use the `copyout` kernel function to copy from kernel space to user space and `copyin` kernel function to copy from user space to kernel space. The wrapper function `sys_wait` uses a `uint64` type variable to store the virtual address of the user space pointer. As this function eventually calls the system call implementation of `wait`, after this point, the prototype is changed to `int wait(uint64 addr)` see `proc.c`.

When you implement a system call, you need to follow the same procedure; otherwise, it might throw a **kernel panic** error due to an invalid memory reference.

Task 4. Modify time1 using wait2() system call to output cputime, elapsed time, and CPU usage. (10 points)

Implement the time1 command so that it calls wait2() to get cputime and outputs elapsed time, CPU time, and %CPU. Test your program with some different inputs and show the results.

Sample output for Task 4 follows below:

```
$ matmul
Time: 35 ticks
$ time matmul
Time: 36 ticks
elapsed time: 36 ticks, cpu time: 36 ticks, 100% CPU
$ time matmul &; time matmul &
$ Time: 69 ticks
elapsed time: 70 ticks, cpu time: 35 ticks, 50% CPU
Time: 68 ticks
elapsed time: 69 ticks, cpu time: 35 ticks, 50% CPU
$ time sleep 10
elapsed time: 10 ticks, cpu time: 0 ticks, 0% CPU
```

Bonus Credit Task. (5 points) Discuss limitations of our time command.

Turn-in procedure and Grading

Please use the accompanying template for your report. Convert your report to PDF format and push your hw1 branch with your code and report to your xv6 GitHub repository by the due date. Also, turn in the assignment on Teams with the URL of your GitHub repo by the due date. Give access to your repo to the instructor and TA.

This assignment is worth 75 points. The breakdown of points is given in the task descriptions above. The points for each task will be evaluated based on the correctness of your code, proper coding style and adequate comments, and the section of your report for that task.

You may discuss the assignment with other students, but do not share your code. Your code and lab report must be your own original work. Any resources you use should be credited in your report. If we suspect that code has been copied from an online website or GitHub repo, from a book, or from another student, or generated using AI, we will turn the matter over to the Office of Student Conduct for adjudication.