CS4375-13948 Fall 2023 Homework 2 report

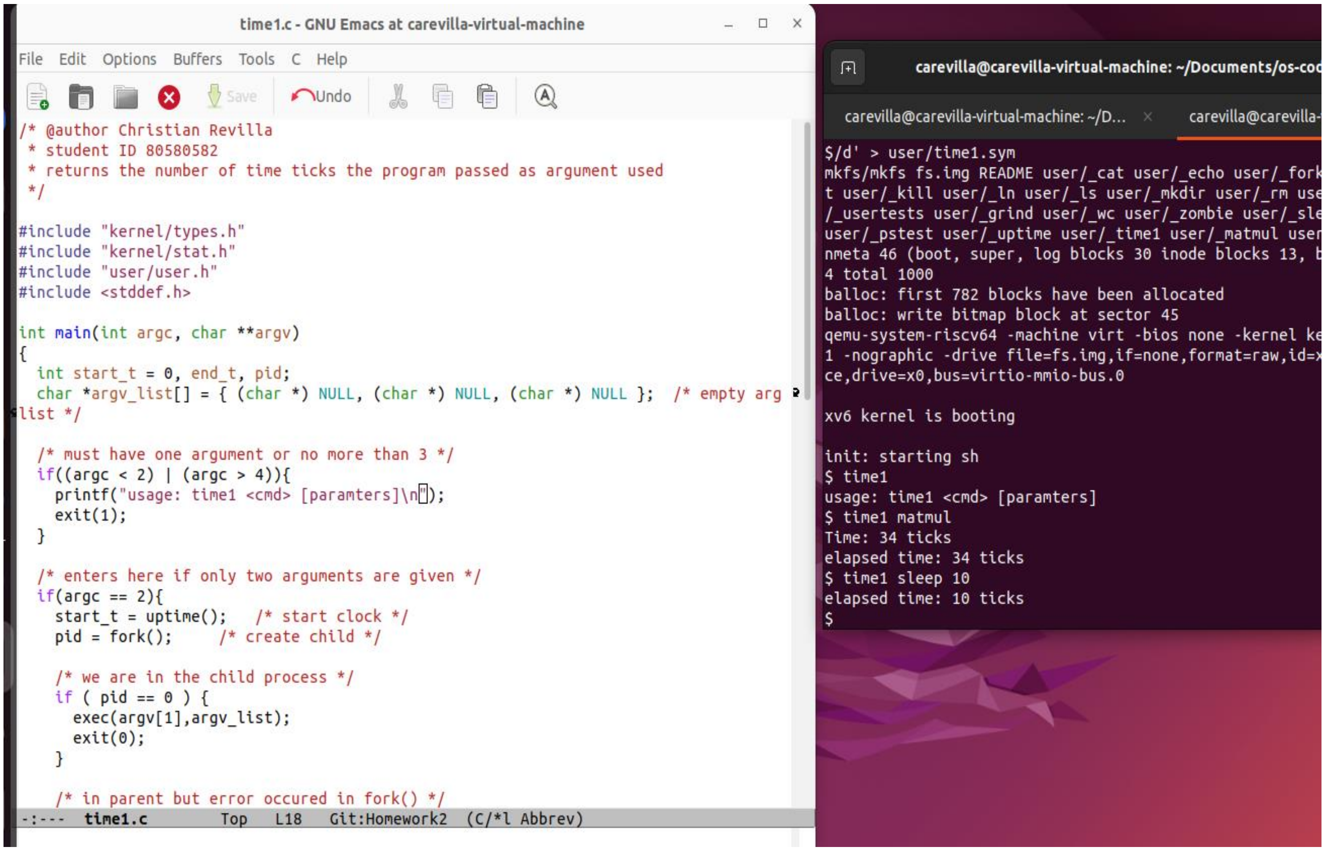
Christian Revilla Submitted on 10/02/2023

Carevilla@miners.utep.edu

Homework 2: Implementing the Time Command

Task 1: Implement the time1 command

For task 1 we were asked to implement a *time1* command that will report the elapsed time to the user after a command execution. To accomplish this task, I first downloaded the *matmul.c* file that was given to us for testing the *time1* command and created a brand-new *C* file called *time1.c* into the *user* directory. From here I added both of those files to the *Makefile* to be able to incorporate them in my code. My process to solve task 1 was to first understand how the *exec* file works and what parameters need to be passed to it. This was my main problem; I did not fully understand what the **path* and ***argv* were supposed to be set at. I first assumed that the **path* is the command that will be executed and the ***argv* is an array that holds the command, parameters, and a NULL terminator. The NULL terminator is important because this tells the *exec* when the parameters to the command end. With that understanding my code starts by checking the length of the arguments passed and when the proper argument length is verified, I populate the a*rgv_list* that will be passed to exec in the child process. I start the timer by calling *uptime* and once that is complete, I call *fork* to get the child process. The child process will execute the command while the parent process calls the *wait* system call, to wait for the child process to finish. Once the child process is finished, *uptime* is called once again, and the time difference is calculated and printed to *stdout* for the user. Below is a screenshot of my *time1.c* when all 3 different number of arguments are passed.



Task 2: Keep Track of CPU time a process has used

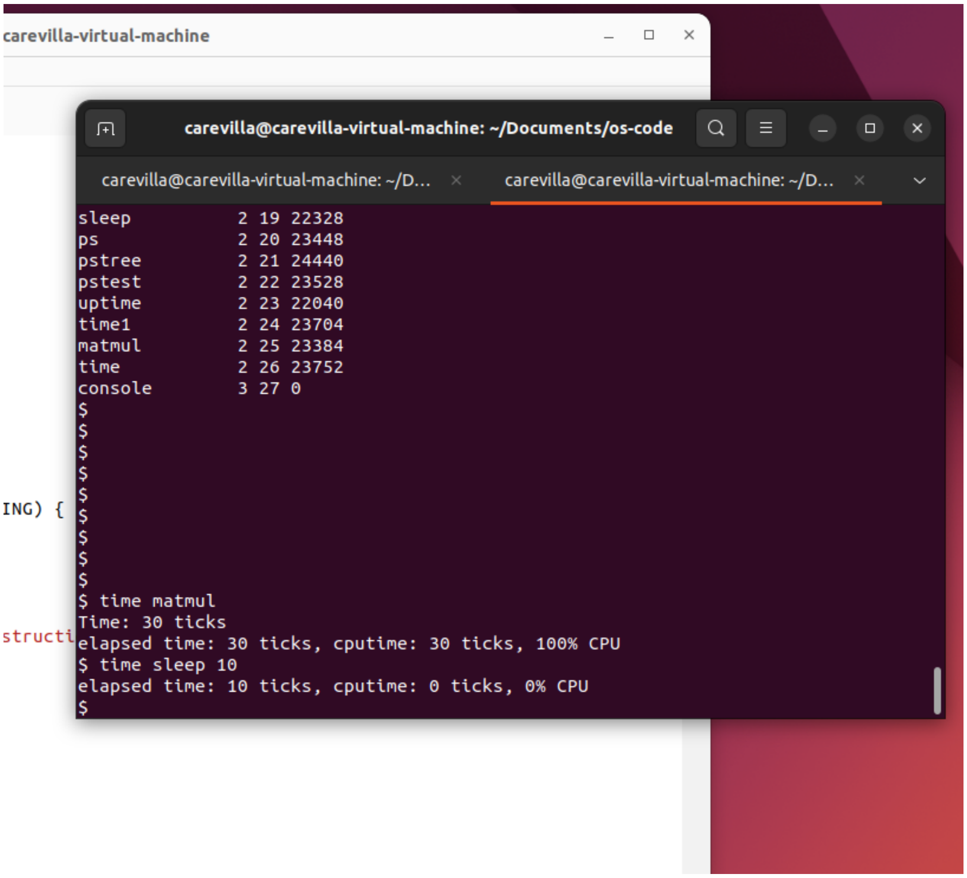
For task 2 we were asked to implement a *cputime* field that will keep track of much CPU time a process has used. This was a straightforward task to accomplish. I started off with first looking into *kernel/proc.h* file and added an integer *cputime* field to the struct *proc*. Once the field was added I reviewed the *kernel/proc.c* file and initialized the field to zero once the process is created during the *allproc()* call. The final part of the task was to go to *kernal/trap.c* file and in *usertrap()* and *kernaltrap()* functions increment the cputime field each time the processor is used. I figured out exactly where to place the increment of the cputime by understanding that if the program is in a RUNNING state, then the time slice is still in used. So, we must increment once the program has used its time slice which is often marked by an interrupt by the operating system. Overall, there were no difficulties in this specific task. I will provide pictures of how the cputime output works during task 4. If curious on code implementation they will be on my GitHub in the respective files that were discussed in task 2 of the report.

Task 3: Implement a wait2() system call

For Task 3 we were asked to implement a *wait2()* system call which will work similarly to the *wait()* system call; however, *wait2()* will return the child status and a structure *rusage* that will be created that includes the CPU usage counts. I first started by creating a structure named *rusage* with one field being unsigned integer *cputime* in the *kernel/pstat.h* file and added the struct definition to the header file in *user/user.h* so the code can use the structure. This structure holds the *cputime* to be used in Task 4. In *user.h* I also defined the system call *wait2(*int, struct rusage*)* so that code knows the appropriate syntax for the system call. Next, I added an entry for *wait2()* in *user/usys.pl* file to be able to generate an assembly language file for the call and added the *wait2* system call information to both *kernal/syscall.h* and *kernel/syscall.c* to ensure the kernal knows the system call. Source code is included in the GitHub repository.

Task 4: Implement time command to call wait2() system call

For the final Task we were asked to implement a new time command that will work similarly to our time1 command except this time command will call the wait2() system call from the parent as the child program executes the command that it was given. As we implemented in task 2 and task 3, the wait2 system call will return a structure rusage that was created in task 2 and the kernel will properly increment the cputime attribute to be returned to the parent process. Once the wait2 system call is returned I simply formatted the output to the user and marked the percentage of the usage of the CPU with the attribute of the rusage structure. Below is a picture of my sample output for different cases that my time command returns.



To conclude this homework assignment, I was able to successfully implement both time1 and time1 commands to my RISC-V emulator and I was able to successfully implement a wait2 system call. Overall, this homework assignment didn't pose too much trouble. The only difficult task I encountered was where exactly to the incrementation of the cputime should happen, but once I understood that I needed to wait for the process to complete its time slice and knew exactly how to access the interrupt from the operating system it was not too difficult.