Advanced Operating System Project xv6

INTRODUCTION:
The xv6 kernel source code is used to build a new kernel to include a syscall that returns the number of processes. The Id and Name of processes are written to a struct procinfo that has 2 variables- int id and string name.

CODING:
The coding for the xv6 kernel syscall getprocinfo() is done by editing files in the xv6 source code. These are the following:
syscall.h: This number 22 is assigned to *getprocinfo()*
syscall.c: The function pointers are defined here- and we add *[SYS_getprocsinfo] sys_getprocsinfo*.
sysproc.c: The *sys_getprocsinfo(void)* is added here and calls proc.c function *getprocsinfo(struct procinfo*).*
usys.S: The system call interface for the call to be user accessible. *SYSCALL(getprocsinfo)*
user.h: This would be the function which user program calls.*getprocsinfo()*

PROGRAM:
The ptable (process table) is locked. Then it is iterated and of the 64 available process slots- the ones with a state that is not 0 (unused)- increases the process counter and writes to the input struct pointer. The process counter is returned to the calling user program. All process slots need to be checked during each call because some processes might be killed and slots might be freed from the middle of the process table. Each process has its id and name fed into the struct pointer.

EXECUTION:
The syscall can be run after calling make qemu which launches the qemu emulator. The qemu emulator is used to execute myprogram and testgetprocsinfo. The myprogram has a simple single call while the testgetprocsinfo has a structured test.

TEST:
The testgetprocsinfo file has the tests needed for evaluating the execution of the getprocsinfo() system call.

The file has 4 print statements for process number with some code between each to manipulate the number of processes running as on the ptable. The method to manipulate the process is by using the fork process. When called from the parent, new children are spawned, while it returns a zero if called from child processes. The process exits after a 2 second sleep (if called by child process).

TEST1: Test1 is done prior to launching any new function and displays the 3 running functions, namely: init, sh and testgetprocs (this is the parent process).

TEST2: Test2 is done after launching 5 forked commands and has a result of 8 processes- previous 3 and the 5 forked testgetprocs.

TEST3: Test3 is done after launching 3 wait commands that makes the parent wait for 3 seconds- and outputs a value of 5. The expected number of processes is 6 (discussed in evaluation).

TEST4: Test4 executes after an additional 2 seconds and by this juncture, all the child processes have exited. The initial 3 processes are displayed again.

Evaluation: In an ideal situation the only time delays would be the wait() and sleep() commands, while the rest of the code runs instantly. However as we can see in Test3, the expected processes is not output, and we get 5 instead of 6 (a loss of 1 process). This can be attributed to the time delay in code processing. While the waits happen, the minute time delay prevents the capture of the 6th process- which exits.

CONCLUSION:

The system call is efficient and works as desired. The addition does not break the other system calls and is efficient. The process that are in the ptable are printed and the total number of processes are output.