Steps to implementing system call getppid()

I will need to edit some files inside xv6 kernel

Step 1: Add an entry for getppid at the end of the list of system calls in syscall.h file. I did that by adding this line to the list of system calls "#define SYS_getppid 22"

Step 2: I am supposed to register getppid() in the system call table in syscall.c. I did this by adding extern int sys_getppid(void); //the function
And [SYS_getppid] sys_getppid,// the new system call

Step 3: I will implement getppid() in the kernel in the sysproc.c, in which I will add the function that will retrieve the parent pid and I will use the already existing function myproc() which returns the current process's structure. So in the sysproc.c I added the implementation of getppid() as follows:

```
int
sys_getppid(void) //no paramaters are needed since the syscall work on the curr process
{
 struct proc *p = myproc();
 //i will use myproc() kernel func that will return a pointer to the currently running processes
 //and give me access to the data of the running process
 //p is a pointer the curr process calling the getppid() syscall

 if(p -> parent){
   //since the myproc() has a pointer to the parent process we will use this pointer to check if
   //p (the curr process calling getppid()) has a parent process, if yes return that pid
   //if not and that is rare since each running process has a parent we return -1.
   //what are the cases when a process doesnt have a parent? if that process is init which is the
   //first user process to run when we boot up the system, in that case if init called getppid()
   //we will return -1
   return p -> parent -> pid;
 }
 return -1;
```

```
}
```

Step 4: To allow user programs to call getppid() we must declare it in user.h file as follows: `int getppid(void);`
That way we exposed the function to user programs

Step 5: Declaring the system call in the user.h is not enough for a user program to call getppid(). Why? Because user programs cant directly call kernel functions like we studied, so to bridge that gap we need to create an assembly stub which is basically a small piece of assembly code that connects the user space programs to the kernel when making system calls. So what happens when a user calls getppid()? The assembly stub issue an interrupt to the kernel with the number of the system call and the kernel finds the corresponding function to that system call number and executes the function, retrieves the parent pid and then finally send the result to the user program in the user space providing the user with the ppid value.

To implement the assembly stub we do as follows in the usys.S file:
SYSCALL(getppid)

Step 6: After finishing all these steps its time to compile and test the added system call.
Rebuild by:
Make clean
Make qemu

If no errors test with a user program

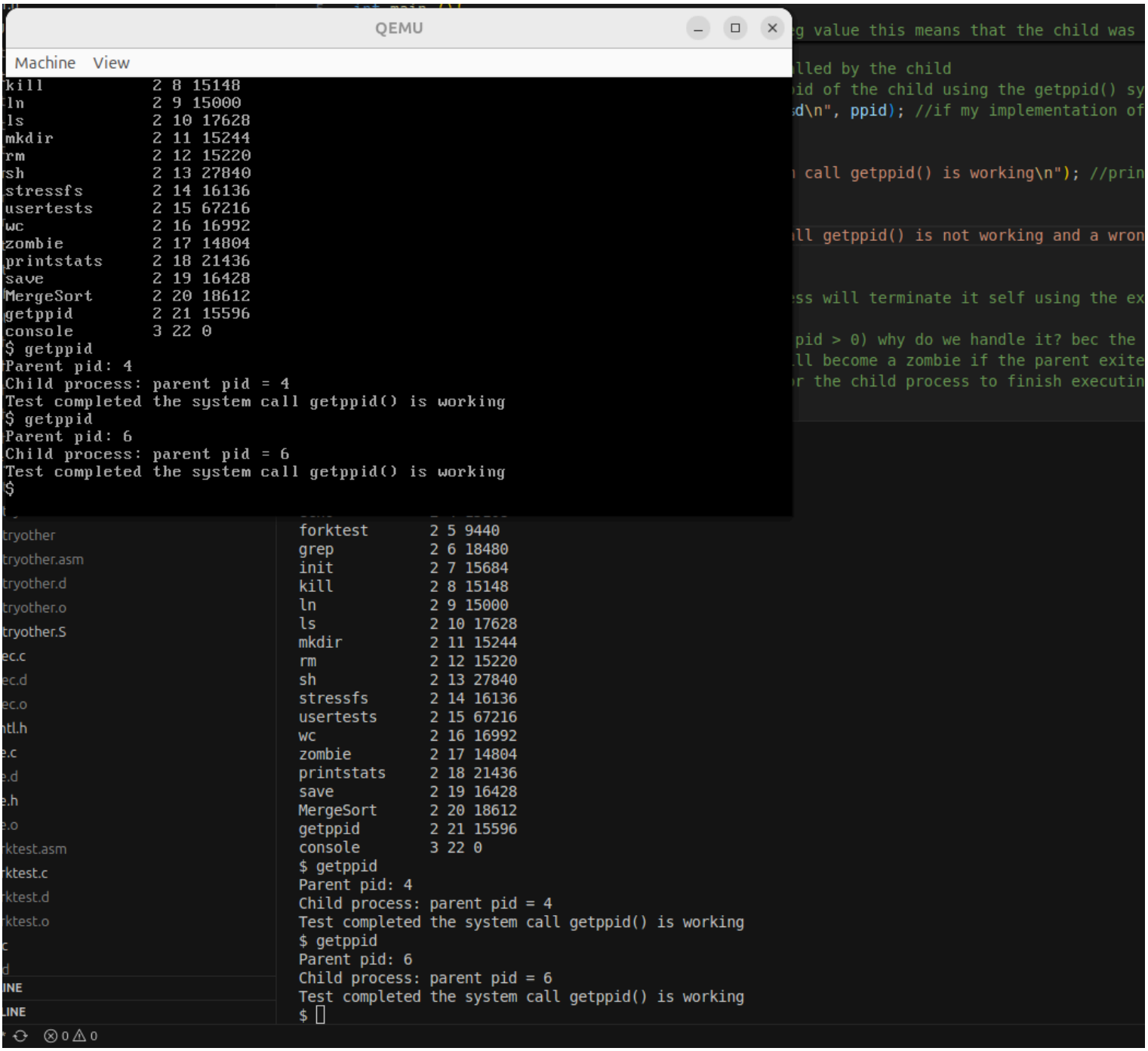
Step 7: create a user program to test your sys call, in my case I created a child using fork() and it returned the correct ppid. My user program:

```
#include "types.h" //integer types
#include "user.h" //for system calls like fork() and getpid() and exit() which i will use here

//we are running this in userspace
int main (){

   int parent_pid = getpid(); //get the parent process id using the already existing system call getpid() which i will use to compare with the child process id
   // to make sure that my getppid() system call is working correctly
   printf(1, "Parent pid: %d\n", parent_pid); //print the parent process id on the terminal
```

```
   int pid = fork(); //create a child process using fork() sys call and as we learned
the child will start exectuing from the same point as the parent
   //bec the child is an exact copy of the parent and fork() returns the child pid to
the parent and 0 to the child

   if (pid < 0) { //if the fork() returned a neg value this means that the child was
not created
       printf(1, "Fork failed\n"); //print that the fork failed

   } else if (pid == 0) { //the getppid() is called by the child
       int ppid = getppid(); //get the parent pid of the child using the getppid()
system call i implemented
       printf(1, "Child process: parent pid = %d\n", ppid); //if my implementation of
the getppid() sys call is correct then the ppid = parent_pid

       if (ppid == parent_pid){
           printf(1, "Test completed the system call getppid() is working\n"); //print
that the test is completed and the getppid() system call is working
       }
       else{
           printf(1, "Test failed the system call getppid() is not working and a wrong
number was returned\n", ppid, parent_pid);
       }

       exit(); //after the test the child process will terminate it self using the
exit() sys call

   } else { //this else handles the case when (pid > 0) why do we handle it? bec the
parent process has to wait for the child to finish executing
       //and if we didnt handle it the child will become a zombie if the parent exited
before the child is done executing
       wait(); //let the parent process wait for the child process to finish executing
   }

   exit(); //when the child exits the parent exits and we are done

}
```

Ps don't forget to add the userprogram.c in the MakeFile

Output:


QEMU
Machine View
kill 2 8 15148
ln 2 9 15000
ls 2 10 17628
mkdir 2 11 15244
rm 2 12 15220
sh 2 13 27840
stressfs 2 14 16136
usertests 2 15 67216
wc 2 16 16992
zombie 2 17 14804
printstats 2 18 21436
save 2 19 16428
MergeSort 2 20 18612
getppid 2 21 15596
console 3 22 0
$ getppid
Parent pid: 4
Child process: parent pid = 4
Test completed the system call getppid() is working
$ getppid
Parent pid: 6
Child process: parent pid = 6
Test completed the system call getppid() is working
$
forktest 2 5 9440
grep 2 6 18480
init 2 7 15684
kill 2 8 15148
ln 2 9 15000
ls 2 10 17628
mkdir 2 11 15244
rm 2 12 15220
sh 2 13 27840
stressfs 2 14 16136
usertests 2 15 67216
wc 2 16 16992
zombie 2 17 14804
printstats 2 18 21436
save 2 19 16428
MergeSort 2 20 18612
getppid 2 21 15596
console 3 22 0
$ getppid
Parent pid: 4
Child process: parent pid = 4
Test completed the system call getppid() is working
$ getppid
Parent pid: 6
Child process: parent pid = 6
Test completed the system call getppid() is working
$