CS330 Assignment 1

Shubhan R 200971
Manas Gupta 200554
Shivam Malhotra 190808
Dishay Mehta 200341

September 29, 2022

README

Part A

Ques 1: uptime

For this question, we made a system call to the already built **uptime()**, which returns the uptime of the xv6 virtual machine as number of ticks passed. To run this user program, run the xv6 virtual machine and run command *uptime* in the terminal.

Ques 2: forksleep

For this question, in the user program we first check whether the number of arguments are correct or not. If they are then we check the validity of those arguments i.e. if the first argument is positive and if the second the second argument is either 0 or 1. Next, we call **fork()** to create a child process. We store the return value of fork in a variable and if it equals 0 then it is a child process and if it is positive then it is the parent process. However if it is negative then there was an error in fork. Now, if second argument was 0, then we make child sleep for **m** ticks where m is the first argument or else if second argument was 1 then we make the parent sleep for m ticks. We print "**pid: Parent.**" in the parent and "**pid: Child.**" in the child where pid can be obtained by calling the system call **getpid()**.

Ques 3: pipeline

For this question, we first check the validity and number of arguments and proceed further. We store the first argument in n (program number) and the second one in x (sum). We then call the function *pipeline* recursively.
This function receives n and x, and immediately returns if n is 0 i.e. we are at the last process and do not need to proceed further creating children. Otherwise it adds the current process id to x, create a child, prints x and current process id, passes the new x to child which reads it as j and calls the function pipeline recursively with $n-1$ and j. Note that we immediately close the file descriptors after reading/writing to prevent running out of file descriptors, and wait for the child to exit before exiting parent.

Ques 4: primefactors

For this question, we have declared two global variables : one array storing all the prime numbers till 100 (*primes*), and a variable called *current_index* which stores the current prime number that we are dividng by. In the program, we first check the validity (argument is between 2 and 100) and number of arguments. We then store the argument in a variable n.
We use a recursive approach to calculate the result. However, we must first check whether we even need to go down that path and create further children and pipes. Thus, we call *reduce* first which does the job of dividing by the current prime factor ($primes[current\_index]$) which is 2, and prints the pid as well as number of times it can be divided.
If after the first *reduce*, n is not 1, then it must have other factors, and we need to create pipes and children. This is done by $primefactors$ function, which creates a pipe and a child, closes the relevant unused ends of pipe in the parent and child, then passes n from the parent to the child via the pipe, then closes other end of the pipe, increments the *current_index*, so we now check for the next prime number

and calls *reduce* on the updated n. If after reduction, n is still not 1, we recursively call $primefactors$ with the new n, that does the same job. If n is 1, then we exit the current process. Note, that each parent waits for its child before exiting.

Part B

For all the below mentioned question, we create a entry for the corresponding syscall in *usys.pl* and add a function prototype in *user.h*. We then assign a unique index to each syscall in *syscall.c* and a unique id name in the syscall table in *syscall.h*. Then we define this syscall function and wrap it around the corresponding content in *sysproc.c*.

Ques 1: getppid

In this question, we need to find the process id (pid) of the parent of a process. We know that in the process struct (proc struct) we have a pointer to the parent process struct. So we access the calling process and check if it's parent exist or not. We point to the calling process struct using $myproc()$ syscall already defined in xv6 and access its parent pointer using $myproc() \rightarrow parent$. If it exists then we return the parent's pid using $myproc() \rightarrow parent \rightarrow pid$ else we return -1.

Ques 2: yield

In this question, we make use of the already defined $yield()$ in xv6. We create a wrapper function which calls yield() function and always returns 0.

Ques 3: getpa

In this question, we use the $walkaddr()$ function to get the physical address using the virtual address. We pass a virtual address i.e. a pointer in the $getpa()$ syscall and then we get the physical address by calling walkaddr() where $walkaddr(myproc() \rightarrow pagetable, p) + (p \ \& \ (PGSIZE - 1))$ is the physical address and p is the passed virtual address.

Ques 4: forkf

Let us see the order of execution of fork() call first. We know that fork() returns 0 to the child and pid of child to the parent. Let's assume ***a0*** register is used to store return value of functions. Let us now trace the function and the return value when a fork() call is done by the user program. The syscall number for fork is defined in kernel/syscall.h. The wrapper function fork() exposed to the users is defined in user/usys.S. The wrapper function prototype is declared in user/user.h. Once the wrapper function is called by a user program, the ecall instruction will guide the execution through the trapping mechanism ultimately landing in the usertrap() function of kernel/trap.c. This function figures out the cause of the trap and if it is due to a system call, the syscall() function is called. This function is defined in kernel/syscall.c. This function uses the syscall number (from trapframe->a7) to index into the syscall function table which is also defined in kernel/syscall.c. Notice that the SYS_fork index of this table is populated with the sys_fork function. Therefore, sys_fork() function gets called next. This function is declared in kernel/syscall.c and defined in kernel/sysproc.c. sys_fork() calls fork() from kernel/proc.c which forks it, sets the return value as 0 in ***a0*** of the child. For the caling process, the *return pid;* statement saves pid of child in ***a0*** at the end of the fork(). Then, it traces back to sys_fork() which returns the same value in ***a0***. Then it traces back to syscall(), which sets [p->trapframe->a0] of the parent process to pid of child process. Then control goes to usertrap(), then it calls usertrapret().

Here, it changes exception program counter to [p->trapframe->epc]. Then, it does *sret* to switch mode that brings you back to the instruction right after the ecall instruction **(the instruction stored in exception program counter)** in the fork() wrapper function, which is the instruction that returns back to the user program. This means the address in ***ra*** register directly leads to the instruction of user program which reads the value of ***a0*** register as return value of fork(). Following this path, when ***a0*** is read by child user program as the return value of fork() system call, it reads 0 and when ***a0*** is read by parent user program, it reads pid of child.

Now let us pay attention to the **bolded text** in the above paragraph. If we can change [**p->trapframe->epc**] to point to some function in user program during forking, it will run the function first on returning to user mode. The user function, if it is non void type, can alter the value stored in register ***a0*** to the another return value. Then it returns to the address stored in ***ra***, which was not altered and still leads to the instruction of user program which reads the value of ***a0*** register as return value of fork().

This is the inspiration behind forkf(f), and this is how it is implemented. For the child, we change [p->trapframe->epc] to point to the passed function. Note that a non void return type function being called from forkf() can alter ***a0*** to another return value. This will be the value read by user program as return value of forkf() when it returns to instruction pointed by ***ra***. If f() is void return type and during execution of f(), some other functions are called which alter ***a0***, then also it is a similar case. If f() is void return type and during execution of f(), no other functions are called which alter ***a0***, then, the original value of ***a0*** itself is returned.

Based on this explanation, we can explain all the outputs as asked in the question. If f() returns any negative value in the question, the parent continues executing, but the child process treats it as an error. If f returns a positive value, the child too executes the code meant for parent and the output is again messed up.

If f() is void type and only the return statement is commented, the last called function, fprintf() alters ***a0*** register to a positive value (to the number of bytes printed). This return value is read and the child too executes code meant for parent and output is messed up.

Output for return value as 0:
Hello world! 100
6: Child.
5: Parent.

Output for return value as 1 (or any positive number):
Hello world! 100
6: 5Pa:r ePnarte.
nt.

Output for return value as -1 (or any negative number):
Hello world! 100
Error: cannot fork
Aborting...
3: Parent.

Output for void return and commenting only the return statement:
Hello world! 100
6: 5Pa:re nt.
Parent.

part 5: waitpid

We have a already defined system call for **wait()**. What *wait*() does is that it waits for any of the child process of the parent process and returns after any one of them has completed execution. However, for **waitpid()** we need our parent process to wait for a specific child process.
The *waitpid*() syscall takes two arguments: one integer and one address. If the integer is -1, then execute a normal wait() from kernel/proc.h else we execute the waitpid() function from kernel/proc.h.
For this we first acquire wait_lock and then iterate through all the currently running processes and check if they have the calling process as their parent. If they don't then we continue else we acquire the process_lock and check if the process pid equals the passed pid. This check is what differs between *wait*() and *waitpid*(). If it doesn't match then we release the process_lock and continue. If it pid matches, continue past the check and from here, the code of *wait()* and *waitpid()* is the same.

part 6: ps

We have added 3 new variables to the process table entry of a process. These are create_time, start_time, end_time

The value of create_time is updated to current ticks count in the allocproc() function of kernel/proc.c as this is when a process is created. The exec_time is set to 0 here. The start_time is set to ticks count in the forkret() function of kernel/proc.c as this is when a process is first scheduled. end_time is set to ticks count when the process exits. Now, we iterate over all processes of the process table entry. If it is unused we skip, otherwise, we first check if it is zombie. If it is zombie, we print relevant values from process table entry, with etime as (end_time - start_time). If process is not zombie, we print all relevant values from process table entry with etime as ((current ticks count) - start_time). Ticks count is obtained for all above function in a simiar way sys_uptime() from kernel/sysproc.c does.

part 7: pinfo

This syscall is largely similar to ps. We proceed by writing the *procstat* structure into the file as described in the question. Next, we implement the system call which reads an integer from the first argument and an address from the second argument. We check the validity of the arguments and then calls the actual function $pinfo$ defined in *proc.c*. If the first argument is -1, we call $pinfo$ with the current process's pid. The function $pinfo$ iterates over the list of process to find a match to the pid except the case when the process table entry is unused. If a match is found, we declare a new struct in the kernel space and copy over relevant the process table entry values into this new struct. We make sure that if the parent of the process is NULL, then we set the ppid to -1. Also the elapsed time for a ZOMBIE process is calculates as $EndTime - StartTime$, while for others it would be $CurrentTicks - StartTime$.
Lastly, we copy over this kernal space struct into the user space's struct using *copyout* to the given user space struct address.