CS330A: Assignment 1

Group: 30

Akshan Agrawal	Harshit Raj	Ananya Agrawal	Akansh Agrawal
180061	200433	200117	180050

September 29, 2022

Note: This pdf both act as a Readme file as well as it contains the explanations wherever asked.

Part A

A. uptime.c: This file is added under user/ , where the c program checks for the valid arguments and then prints the uptime as required. Suitable piece of code is also written in Makefile, user/user.h , user/usys.pl to make the code compile and check the implementation.
B. forksleep.c: This program first checks for the cases of valid arguments and then uses fork and sleep to get the desired answer. If n is 0 then inside the child the sleep call is made, so that parent gets wake up and prints its pid. Else if n is 1 , then parent waits for the child.
C. pipeline.c: This program first checks for the valid arguments and then utilises an fd[2] for read and write via pipeline mechanism. The fork is used for this purpose where the child reads with the help of fd[0] and writes with fd[1]. The pipeline for the n processes is created using for loop to get the desired functionality.
D. primefactors.c: This programs first checks for the valid arguments and then utilises the pipeline mechanism to pass the end result to the next process. We used fork inside for loop which will run 25 times (since 25 prime factors given) and performed read and write via pipeline mechanism. For each iteration (process) in for loop we check if the prime factor divides the number we got from reading via pipeline and write the remaining n for the next process (iteration).

Part B

A. getppid: The crux of the implementation of this syscall lies in utilizing the structure of the process (say p). Since the structure of p stores the access to the parent process, we utilised the value p->parent->pid to get the required parent pid. So, in the sysproc.c we implemented the syscall such that it returned the parent id if the parent process exists else we return -1 as asked.
B. yield: Refer to the function in kernel/proc.c. Note that a process that is giving up the CPU acquires the process table lock and updates its state as RUNNABLE. The yield then calls the sched(), which in turn calls the swtch() that saves the current context in &p->context and switches to the context of the scheduler. Thus allowing other processes to run. Thereafter, the acquired/locked process is released once it gets scheduled again. Note that, we defined yield() system call number as 23 within syscall.h.
C. getpa: This syscall is implemented (please refer to kernel/sysproc.c) such that it takes a virtual address as the only argument and utilises the walkaddr() defined in kernel/vm.c to get the corresponding physical address. As a safety parameter we also checked if the unsigned virtual address is properly assigned, else we return -1.
D. forkf: In this we create a new process by copying the parent and executing the function f (copy user memory from parent to child). The change with respect to normal fork call is that the forkf system call takes the function address as an argument and the parent process is resumed when the child process returns. In addition to fork system call, the following line is added in kernel/proc.c to the fork() call: np->trapframe->epc = (uint64)f, to take care of the

requirement that the child first executes a function right after returning to user mode and then returns to the code after the forkf() call.
Case 1: If return value is 0, then the output is following:
Hello world! 100
4: Child.
3: Parent.
First, the process goes to function f and prints and value 0 is returned. The forkf gets 0 as the argument, which returns 0 value to x and the valid child process is created successfully. Thus the child process first completes and then the parent process is completed.
Case 2: If return value is 1, then the output is following:
Hello world! 100
3: Parent.
4: Parent.
First, the process goes to function f and prints and value 1 is returned. The forkf gets 1 as the argument, which returns positive value and the valid parent process is entered, followed with another parent process. The first parent process would be completed, since there is no child and later the other parent process (with a pid value greater) is completed.
Case 3: If return value is -1, then the output is following:
Hello world! 100
Error: cannot fork
Aborting...
3: Parent.
First, the process goes to function f and prints and value -1 is returned. The forkf gets -1 as the argument, which returns negative value and no valid child process is created, but the parent process is completed then.
Case 4: If return value is changed to some other value: suppose to positive value we get the same output as when the return value was 1. This is because forkf gets positive value as argument, thus returning positive value to x. Suppose to negative value we get the same output as when the return value was -1. This is because forkf gets negative value as argument, thus returning negative value to x, since no legitimate process is created.
Case 5: If f is changed to void and the return statement in f is commented: then our program throws the error, because of the manner we implemented forkf() system call, because it expects int to be returned, instead of void (as we took forkf(int (*f)(void)) as our syntax to the system call implementation). If the forkf was implemented such that it expects an argument void* then the output would be the case similar to the case of having a return type from function f as a postive integer.
E. waitpid: In kernel/proc.c , we utilised the idea of implementation of wait syscall to implement the waitpid. The difference lies in the if condition where we ensure that wait is modified such that it waits for the child with the given pid. Since in the syscall regardless of any child pid the wait does it jobs and then parent continues , but due to one more condition set in the if part of the code , the syscall waits for the given pid to complete. The rest implementation is almost same just that we check for valid pid and status to be passed, otherwise it returns -1 as an error.
F. ps: In kernel/proc.c , for each process in process table we check whether the process is UNUSED or not. If it is UNUSED then we go to the next process in the process table entry, else we print the required answer. For the process (say p) whose state is not UNUSED is locked via acquire(&p->lock) and released at the end of the for loop. When we have locked the process p, then to measure etime we use acquire(&tickslock), since we wish to the end time at this instance , which can be computed by locking the current time and subtracting the starting time from it. To get the ppid of the process (if parent exists) , the parent process is locked in the similar fashion to get the pid of the parent and later released. In this way, we calculate the desired output as required for each process.
G. pinfo: It returns the information about a specific process to the calling program. Refer the pinfo function of return tpe int: pinfo(int pid, uint64 psaddr) written inside kernel/proc.c, wherein the two arguments refer to an integer and a pointer to a procstat structure. For a valid pid, the corresponding process is lock acquired and required information is obtained via copyout (copies data from the kernel to the user-space memory of the process) taking pagetable as argument and finally locked process is released.

References

[1] CS330A Lecture Notes, Fall 2022, IIT Kanpur
[2] xv6 RISC-V book