CS330

Assignment - 1

amogOS

Soham Samaddar
200990

Aditya Tanwar
200057

Samarth Arora
200849

Sarthak Kohli
200886

September 2022

Part A

There were no `syscalls` implemented in this part, rather only user programs. Thus, the implementation of these were fairly simple in comparison to those in *part-b*, and only involved one `*.c` file, and an entry in the `Makefile`.
The `*.c` files mostly use pre-implemented `syscalls` like `fork()`, `pipe()`, and `getpid()`.

Part B

Most of the `syscalls` were not trivial in nature and as a result, have been implemented in `proc.c`, with the sole exception of `getpa`, which is implemented completely in `sysproc.c`.
The exact details of implementation have been elaborated for each `syscall` below:

`getppid`: As advised in the comments of `struct proc` in `proc.h`, the `wait_lock` must be acquired before accessing any process' parent. Hence, we create a pointer to a `struct proc`. We acquire the `wait_lock`, access the parent and set the pointer to the parent. We then release the `wait_lock` before accessing any other locks to prevent a deadlock situation. Then finally, we acquire the parent's lock to access the pid, release the lock and then return the `pid` while taking appropriate measures in case the calling process has no parent.

`yield`: This was the most trivial implementation since the code for `yield` was already written in `proc.c`. All we had to do was call it from `sysproc.c`.

`getpa`: This function was entirely implemented in `sysproc.c`. First, the virtual address whose physical address is to be calculated was obtained by calling `argaddr`. Since the function in user mode passes only one argument (the virtual address), we need to access register `a0`. Hence 0 is passed in the parameters of `argaddr`. Then, the formula given in the assignment was used to calculate the physical address and returned.

`forkf`: The key to implementing this syscall was understanding the entire pipeline of execution when a syscall is called. When any syscall is called, the syscode is first stored in the `a7` register. Then the `ecall` instruction is executed which switches to the kernel mode. After switching to the kernel mode, assembly code in `trampoline.S` labelled `uservec` is executed which saves all the values of the general purpose registers in the `trapframe`. It also makes some other modifications and jumps to `usertrap()` in `trap.c`. The user program counter (which stores the address of the `ecall` instruction at this point) is obtained by calling `r_sepc()` and is stored in the `trapframe`. Now, the cause of the trap is ascertained by calling `r_scause()`. If the trap was indeed due to a syscall, the program counter in the trapframe is moved forward by 4 bytes so that it points to the `ret` instruction right after the `ecall` instruction in the user mode. **This is the crucial information**. After this, the syscall is finally executed. Then, after execution of the syscall, the program state is restored by calling reloading all the registers from the trapframe (done in `trampoline.S` and then the `sret` instruction is executed which puts the control back to user mode. The user program counter points to `ret` which returns to the original user program.

With this pipeline, it is clear that we need to shift the program counter so that it executes the function from the function pointer passed, instead of the `ret` instruction. So, `forkf` is identical to `fork` in almost all aspects except when `fork` changes the `a0` register (return value) in the trapframe of the child process, we additionally change the `epc` register to point to the function address. So, after `sret` is called, the child process starts executing from the function address.

Note that changing the value of `a0` to 0 of the child process in the `forkf` syscall is irrelevant since this value is never received. On returning to user mode in the child process, execution starts from the function (whose address is passed in `forkf`) and the value returned by this function is stored in `a0` which is finally the value returned by `forkf`. This return value is returned at the same address that `fork` would have returned to, this is because we did not change the value of the register which stores the return address (`$ra`). It also provides an explanation for why we did not need to change the value of `$ra`, rather, only the value of `$epc`, since we still need to return at the same line as `fork`. In all situations, the parent has the `child pid` in x. We describe the outputs of the given test code below when the return value of the function is:

0: The child process has $x = 0$. The parent process sleeps for 1 second which is a judicious amount of time for the child to finish executing both the function and the fprintf statements. The outputs are distinct and clean.

1: The child process has $x = 1$. Hence both the parent and child sleep for 1 second. So, the output to the console from the function `f` is distinct and clean, while the outputs of the other `fprintf` statements are all jumbled up. This is because both the processes (xv6 being multicore) simultaneously write to the console leading to overlap in the output.

-1: The child process has $x = -1$ which goes into the error clause of the if statement. So, `f` writes the output to the console, followed by the error message. Then, the parent wakes up after 1 second and prints to the console. The output is distinct and clean.

`void`: When `f` is made void, a shortcoming in our implementation comes to light. The value in x can be anything depending upon the function `f`, so any branch may be taken. When the program counter is switched to the function address, the compiler has no idea that a function call has been made. As such, the calling convention of registers are not respected and the `caller saved registers` are polluted during the function execution. Since `a0` is a `caller saved register` it does not store the expected value of 0 (the value which was stored during the `forkf` syscall). Instead, it returned with the unexpected value of 1. Hence the output was similar to the situation when the return value of `f` was 1. In another test, we made `f` execute nothing (i.e it had no instructions). The value of x was found to be 0. It might be possible to fix this by saving the `caller saved registers` before executing `f` and then restoring them after executing `f`, or by somehow letting the compiler know that a function call is being made after executing `sret` so that it may respect the register calling conventions.

`waitpid`: This function was implemented entirely in `proc.c` with the code being by and large similar to the pre-implemented `wait` syscall. The only change made here was to not pick any arbitrary child of the calling process (unless the first argument is -1, in which case the behaviour of the program is identical to that of `wait`), but instead, pick and wait exactly for the child whose `pid` equals the given first argument. It returns when the child exits, or if the

child was never alive to begin with.
The acquisition and release of locks is the same as that in `wait`

`ps`: This function was implemented entirely in `proc.c` with the code being similar to that of `procdump` with some changes, the first of them being the acquisition of `wait_lock` before entering the `for` loop and its release after the loop ends, because a process' lock is frequently acquired inside the loop.
Now, inside the loop, firstly, the process' lock is acquired. In the event the process' state is currently "UNUSED", the lock is released and the loop continues. Otherwise, the `ppid` of the process is stored in a local variable to take care of edge cases (which themselves can involve acquiring and releasing the lock of a process' parent). Similarly, the `etime` of a process is also stored in a local variable, to handle the two cases of a process being "ZOMBIE" and otherwise separately. If it is a "ZOMBIE", the values (of `start_time` and `end_time`) stored in the process' `proc struct` are used directly to calculate the value of execution time (`etime`), otherwise, the value of current ticks is used in place of the value of `end_time` in `proc struct`. After all the required values are ready, they are printed and the process' lock is released, and the loop continues on.

`pinfo`: This function was also implemented completely in `proc.c`. Its execution can be divided into some major steps, the first of them being finding the process whose info we need to return. If the first argument is -1, the process of importance is simply allocated the process returned by `myproc()`, otherwise, a loop is used to iterate over all the processes and find the appropriate process. If the process is not found, or a `NULL` pointer is sent in as the second argument, -1 is returned.
After all the necessary checks, the required information is copied into a local variable of type `struct procstat`, and then `copyout()` is used to copy all the information from this local variable in kernel address space into the address (in user address space) sent as the second argument. A local variable is used so that `copyout()` is called only once, thus improving efficiency.
Lastly, the locks are released, and appropriate measures are taken to ensure that the same lock is not released twice (important when the first argument is -1).