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. If the process does not have a child, we release the `wait_lock` and immediately return -1. Otherwise, we acquire the parent's lock, store its `pid`, and then release its lock. Finally, we release the `wait_lock` as well and return the parent's `pid`.

`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 fact is crucial to our implementation**. 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, since the values of (`$ra`) are identical in the parent and child at this point of execution. It also provides an explanation for why we did not need to change the value of `$ra`, instead of the value of `$epc` (which was our attempt initially), since we still need to return at the same line as in `fork`. If `$ra` was changed to the function pointer address, then we get stuck in an infinite loop since `f` returns to the start of `f`.

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 (since xv6 supports 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. But all attempts to address this issue without resorting to coding in assembly failed.

`waitpid`: This function was implemented almost entirely in `proc.c` with the corresponding function in `sysproc.c` being used only to obtain the required arguments from the `trapframe` (by calling `argint` and `argaddr`). The code is by and large similar to the pre-implemented `wait` syscall. It is assumed that a `pid` less than -1 or equal to 0 is invalid, and returned -1. 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. To find the child, we iterate over all the processes, call the process being iterated over `np` and the calling process `p`.
If the parent of `np` is indeed `p`, then `np`'s lock is acquired. If `np` is not the required child (i.e., `pid` of `np` is not equal to x , where x is the first argument to the waitpid system call) when $x \neq -1$, its lock is released, and we move on to the next process (We can access `np`'s `pid` because we acquired its lock earlier). In the case when `np` is indeed the required child, we return from the function only when it becomes a zombie, in a fashion similar to that of `wait`. Summarising, the function is similar to `wait` except in the way it searches for the child, which is facilitated by the addition of an `if` conditional in the function body.

`ps`: This function was implemented entirely in `proc.c` with the corresponding function in `sysproc.c` being used only to call the function in `proc.c`. The code is similar to that of `procdump` with some changes, mainly the proper usage of locks. The lock acquisition order respects the order used throughout the codebase (`wait_lock` is always to be acquired **before** any process' `lock` is acquired), ensuring that a deadlock does not occur.
We iterate over every process in the process table. 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, all information except `ppid` is stored locally. This is because we would like to release the process' lock as soon as possible in light of improving performance. The `etime` of a process is calculated using an `if` statement to tackle the two cases of a process being "ZOMBIE" and otherwise. 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 releasing the process' lock, we finally store the `ppid` locally by acquiring the `wait_lock` first. Then we check if the parent is null or not and set the `ppid` accordingly. If the parent is not null, we also acquire the parents' lock since we are accessing the `pid` of the parent. Finally, all locks are released. The required values are printed and the loop continues.

`pinfo`: This function was also implemented completely in `proc.c` with the corrseponding function in `sysproc.c` just used to parse arguments and call the function in `proc.c`. Its execution can be divided into some major steps, the first of them being the "search" phase where the pointer of the process, whose info we need to return, is found. 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. We call the process of interest, if found, `p` and acquire its lock before moving on to the next phase.
After all the necessary checks, the required [personal] information of `p` is copied into a local variable of type `struct procstat`, followed by the release of `p`'s lock (recall that it was acquired during the "search" phase). The only piece of information left to obtain from `p` is

its ppid; wait_lock is acquired for the same. If p's parent is NULL, then ppid is assigned -1, otherwise, the lock of p's parent is acquired in order to access its (the parent's) pid followed by the release of the lock. Finally, copyout() is used to write all the information from kernel address space into the given address (which points to memory in user address space). In the case of any error, -1 is returned, and 0 otherwise.

The tickslock is acquired and released for storing the values of creat_time, start_time, and end_time and while the lock is acquired, the value of ticks variable is used, inspired from the sys_uptime(). The exact instances when there values are assigned are described below:

creat_time of a process is stored when a process is being allocated in the function allocproc(). It is assigned right before the function's return statement.
start_time of a process is stored when it is first scheduled and starts its execution in forkret. It is assigned just before the current process' lock is released.
end_time of a process is stored when the function calls exit(). It is assigned just after the process' lock is acquired.
etime (which stands for execution time) is used in syscalls like ps() and pinfo(). If the process is a "ZOMBIE", then we are assured that both start_time and end_time hold meaningful values, and it is assigned their difference. Otherwise, if the process is alive, then end_time does not hold a meaningful value for the process and thus, in this case, etime is assigned the difference between current time (found using ticks) and start_time.