Operating System: Assignment-1

Team Members (Group-27)

- Jaya Gupta(200471)
- Harshit Bansal(200428)
- Avi Kumar(200229)
- Arpit Kumar Rai(200190)

SysCall Implementation Details

Syscall : getppid

(*Implemented in kernel/proc.c*)

`getppid` does not take any argument and returns the pid of the parent of the current process if parent exists otherwise it returns `-1`. This syscall is implemented by accessing the parent field of the current process' `proc` struct. wait_lock must be acquired before accessing parent field to avoid losing wakeups of wait()ing parent.

Syscall : yield

(*Implemented in kernel/sysproc.c*)

`yield` does not take any argument and the calling process voluntarily yields the CPU to other processes. This syscall is implemented using the `yield()` function defined in kernel/proc.c. This syscall always returns 0.

Syscall : getpa

(*Implemented in kernel/proc.c*)

`getpa` syscall takes a virtual address (i.e., a pointer) as the only argument and returns the corresponding physical address. This syscall uses the `walkaddr` function (looks up a virtual address, returns the physical address) defined in vm.c . The syscall reads the virtual address passed as an argument using the `argaddr()` funtion defined in syscall.c .

Syscall : forkf

(*Implemented in kernel/proc.c*)

`forkf` syscall takes a function address as an argument. This syscall is implemented similar to fork with the differnce that the program counter for the child process (executed when child returns to user mode) is set to the start of the function passed as argument. Things done to implement `forkf` are

- The syscall reads the function address passed as an argument using `argaddr()` function defined in `syscall.c`.
- The entire implementation of `forkf` is similar to `fork` except the fact that the created child's(`np`) `epc` is set to the start of the function. This will do the required job because in case of `fork`, the child's `epc` will be similar to parent's `epc`, which is, in case of xv-6, the address of the `ret` instruction for the forkf entry in `usys.S`. And then `ret` jumps to the address stored in the `ra` register. But since we have changed the epc to the start of the function, the return value will remain same, hence child after executing the function will return to the same point in the user program where `forkf` was called.

We need to observe that the child will never return to the `ret` instruction in `usys.S` in the forkf implementation.

On executing forkf, for the child process, the return value is set to 0 in register `a0` in kernel. However, before returning to the code after forkf(), function f() gets executed in the user mode. Hence the return value of forkf() is overwritten in the register `a0` by f() and set to the value returned by f() itself. For the parent process, return value of forkf() is always pid of child process which is greater than 0. Hence parent always executes `else if` block and goes to sleep.

Case1: When return value of f is 0

Child executes `else` block. Since parent is sleeping in `else if`, child prints its output followed by parent printing its output and then both exits.

Case2: When return value of f is 1

Child and Parent both executes `else if` block and goes to sleep. Both parent and child are sleeping. If parent wakes up and is scheduled first, it calls printf() and goes to sleep again. In the meantime, child wakes up and calls its printf() and exits. Child does not do anything on wait call since it has no children. Thus both the processes print their outputs together in a jumbled and haphazard way. The same thing happens if child is scheduled first.

Case3: When return value of f is -1

Child executes `if` block and exits. Eventually, the sleeping parent wakes up and prints its output and exits if the child has exited.

Case4: If the return value of f is changed to some integer value other than 0, 1, and -1

If the return value > 0, the behaviour is similar to `case 2` above. If the return value is less than 0, the behaviour is similar to `case 3` above.

Case5: If return type of f is changed to void and the return statement in f is commented

In this case the return value of forkf() (i.e the contents of the return value register) in child process depends on whether fprintf() in function `f` gets executed successfully or not. If fprintf executes successfully, the return value is 1 and hence the behaviour is similar to case 2 above. If fprintf fails, the return value is -1, hence the behaviour is similar to case 3 above.

Syscall : waitpid

(*Implemented in kernel/proc.c*)

`waitpid` system call takes two arguments: an integer and a pointer to the status code variable.

The passed arguments are read using the `argint()` and `argaddr()` implementation in `syscall.c`.

The integer argument can be a pid or -1. The system call waits for the process with the passed pid to complete provided the pid is of a child of the calling process. If the pid is a valid pid, `waitpid()` iterates over the process table and looks for children of currently running process having pid same as the passed argument. If such process is found, it sleeps until the process is completed. And if the found process is already zombie, it returns.

If the first argument is -1, the system call behaves similarly to the wait system call.

The normal return value of this system call is the pid of the process that it has waited on. The system call returns -1 in the case of any error.

Syscall: ps

(*Implemented in kernel/proc.c*)

`ps` syscall does not take any arguments and returns 0 on successful execution. To implement this syscall, iterate over the process table, acquire the process lock for the current chosen process to access critical fields in its Process Control Block struct. Skip the process if the process is in `UNUSED` state else get the rest of information and print it. *Start time* is set in `forkret` function (new process starts execution from this function when it is scheduled for the first time). *End time* is set in `exit` syscall function. *Creation time* for the process is set in `allocproc` function when the process is allocated. It is required to acquire wait_lock before accessing the parent of process and process lock needs to released before acquiring wait_lock to maintain the order locks in which to acquire them.

Syscall: pinfo

(*Implemented in kernel/proc.c*)

`pinfo` returns the information for process in the struct for the given pid, both of which are passed as arguments. If pid is -1, the information for calling process in returned. The implementation of this syscall is similar to `ps`, except for the information storing part, where the process info is first stored in a kernel data structure (declared in *kernel/procstat.h*) and the copied into the user data structure(using the `copyout` function) whose address is passed as argument.