README

Pratyush Gupta (200717), Parinay Chauhan (200667),
Kunwar Preet Singh (200536), Aryan Sharma (200203)

September 2022

1 Introduction

The general methodology to add a new syscall to xv6 is as follows:

1. Add a wrapper function with name f to *user/user.h* with appropriate arguments and return type. For example: *int forkf(int (*)(void))*.
2. Add an entry point to f in *user/usys.pl*. For example: *entry("forkf")*. This ensures that appropriate assembly code for this syscall is added to the *usys.S* file when we make qemu.
3. Add function definition to *kernel/defs.h* with appropriate return type and arguments. For example : *int forkf(int (*)(void))*.
4. Assign appropriate syscall number like *#define SYS_forkf 25* in *kernel/syscall.h*.
5. Link the syscall number with appropriate function call in *kernel/syscall.c* by creating an entry of the form *[SYS_f] sys_f,* in the *syscalls* array.
6. We now, define the $sys\_f(void)$ function in file *kernel/sysproc.c*. Note that the argument is always void since the arguments have already been written to registers a_0 to a_7 in the process's *trapframe*.
7. *sys_f* generally invokes another function $return\_typef(args)$ which is defined in *kernel/proc.c*.

In order to test an implemented syscall we have made a file *user/syscall_test.c* which contains functions *test_f* to test system call f. Corresponding changes have been made to the Makefile.

2 getppid

In *sys_getppid(void)*, we call *myproc()* and check if its parent is *NULL*. If so, we return -1, else we return the *ppid* i.e. the value stored in the parent field.

3 yield

To implement this call, the process's state is changed to *RUNNABLE* and then the scheduler is invoked, thus yielding the CPU to some other process.

4 getpa

To implement this call, we return the physical address corresponding to virtual address *A* i.e. *walkaddr(p − > pagetable, A) + (A & (PGSIZE - 1))* from the *sys_getpa()* function in the *kernel/sysproc.c* file.

5 forkf

We implement this call in a way similar to $fork$, the only difference being that the program counter of the new process is set to the function pointer i.e. the address of the instruction pointing to the beginning of the function.

This works because there is a function call preceding $forkf$ i.e. sys_forkf, the ra register is set to instruction corresponding to return from sys_forkf when $forkf$ is called. In the child *epc* is set to the function address and therefore when child is scheduled, after it has executed the function f the return instruction from that function f causes the child process to come back to the instruction corresponding to the return of sys_forkf, hence it looks like the child just returned from $fork$ call and executed a function before returning.

If return value of f is 0: parent process gets child's pid as the return value from $forkf$, child gets 0 as the value of x. Parent goes into *else if (x > 0)* branch and child goes into *else* branch. Parent executes $sleep(1)$ and therefore, child prints earlier. Output comes out as:
Hello world! 100
4: Child.
3: Parent.

If return value of f is 1: parent process gets child's pid as the return value from $forkf$, child gets 1 as the value of x. Parent and child both go into *else if (x > 0)* branch. Both parent and child execute $sleep(1)$ and then simultaneously print. One of the possible output comes out as:
Hello world! 100

4: 3P:a rPeanrte.n
t.

If return value of f is -1: parent process gets child's pid as the return value from $forkf$, child gets -1 as the value of x. Parent and goes into *else if (x* > 0*)* branch and child goes into *if (x* < 0*)* branch. Parent executes *sleep*(1) and child executes *fprintf(2, "Error: cannot fork...");* in that time. Output comes out as:
Hello world! 100
Error: cannot fork
Aborting...
3: Parent.

If the return value was something else (say x), then $forkf$ behaves as if the return value of f was 1 if $x > 0$, -1 if $x < 0$.

If return type is changed to void: function f itself doesn't change the register $a0$ which is otherwise supposed to contain the return value of the function, but f calls $fprintf$("...", $g(x)$), therefore the return register $a0$ is set to the last called function that had some return value when f returns. Tracing the execution of the $fprintf$ function : $fprintf >$ ... $>$ $vprintf >$ ... $>$ *putc* $>$ ... $>$ *write*. *write* is an *int* return type function and probably returns the number of bytes printed, *write* function is called by *putc* for printing a character at a time and therefore *write* returns 1. When f returns it has register $a0$ set to the last called function that saved something in $a0$. Therefore void f has the same effect as f with return value 1. In general for other variants of f, if f is void, during the return of f whatever was the last value set in $a0$, behaviour of $forkf$ is as if the function f had returned the value stored in $a0$, in general the behaviour would be unpredictable.

6 waitpid

This is implemented in a way similar to *wait* call, the only difference being an additional check for equality between passed *pid* and *pid* of the process being checked.
When value of passed pid is -1 this call performs the exact same function as the wait call.

7 ps

In order to implement this call, we add 3 entries to the *proc* structure namely, *ctime, stime and etime..*
ctime denotes creation time, it is assigned its value by the reading the *ticks* variable when the process is allocated i.e. in the *allocproc* call.

stime denotes the start time, it is set when the *scheduler* first schedules this process .i.e. inside the *scheduler* function.
etime denotes the end time of the process, it is set when the process is destroyed i.e. inside the *exit* call.
The initial values of *stime* and *etime* are -1, denoting that the process has not started or ended respectively.
ps returns the creation time, start time and execution i.e. (end time - start time) or (current time - start time) if the process has not yet ended.
Before saving the information like pid, state, command and size of the process we acquire the *p lock* of the process and release it after the information has been saved and printed.
Similarly, before saving the ppid, we acquire the *wait_lock* of the process and the *p lock* of the parent of the process and release them after saving that information.

8 pinfo

We make a procstat structure to gather all the relevant data into one struct. The *pinfo* function takes *pid* and *addr* as arguments, gathers the corresponding entry from the the process table and copies the data from the kernel space to the *addr* address in the user space using the *copyout* function.
If passed *pid* is -1 we return the data corresponding to the calling process.

Moreover, as we did in *ps*, here also before saving the information like pid, state, command and size of the process we acquire the *p lock* of the process and release it after the information has been saved and printed.
Similarly, before saving the ppid, we acquire the *wait_lock* of the process and the *p lock* of the parent of the process and release them after saving that information.