January 2023 CSE 314
Offline Assignment 2: xv6 - System Call

In this offline, you will add new system calls to xv6, which will help you understand how they work and will expose you to some of the internals of the xv6 kernel. You have to implement two tasks.

Suppose our favourite OS xv6 is under virus attack. The virus is calling various system call on its own. That's why we need to track the system calls with the relevant history of each system call.

Task 1: Trace

Implement a new system call *trace* that will control your program tracing. It should take one argument, an integer *syscall_number* which denotes the system call number to trace for a user program. For example, to trace the fork system call, a user program calls *trace(SYS_fork)*, where SYS_fork is the syscall number of fork from *kernel/syscall.h*.

You have to modify the xv6 kernel to print out a line when each system call is about to return for a process, if the current system call's number is the same as the argument pass in the trace system call. The line should contain the following:

- process id: a number representing the id of running process
- the name of the system call: a string
- the arguments of the system call: a tuple that shows output corresponding to each argument's datatype
- the return value of the system call: a number

The trace system call should enable tracing for the process that calls it but should not affect other processes.

You will be given a *trace.c* user program that runs another program with tracing enabled. For example, when you run `trace 7 echo hello`, the trace user program will first enable tracing for syscall_num 7 (i.e *exec*) and run the `echo hello` program (look at the *trace.c* file). Make necessary changes so that you can use the given *trace.c* user program in the shell. You might expect output like this:

```
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ trace 15 grep hello README
pid: 3, syscall: open, args: (README, 0), return: 3
$ grep hello README
$ trace 3 grep hello README
$ trace 5 grep hello README
pid: 4, syscall: read, args: (3, 0x0000000000001010, 1023), return: 1023
pid: 4, syscall: read, args: (3, 0x000000000000103a, 981), return: 981
pid: 4, syscall: read, args: (3, 0x0000000000001023, 1004), return: 350
pid: 4, syscall: read, args: (3, 0x0000000000001010, 1023), return: 0
$ trace 21 grep hello README
pid: 5, syscall: close, args: (3), return: 0
$ trace 7 echo hello
pid: 6, syscall: exec, args: (echo, 0x0000000000003e60), return: 2
hello
$
```

Explanation

- `trace 15 grep hello README`: It means you want to trace system call number 15 (i.e: *open*) for the program `grep hello README`. You can see 1 line of output, so it called the *open* system call 1 time. The arguments to *open* are: a string representing the path to the file (here, *README*) and an integer representing some flags (here, 0). The arguments are printed in their expected format.
- `grep hello README`: It has no tracing output because it did not set the tracing through the *trace* user program.
- `trace 3 grep hello README`: It sets trace for system call number 3 (i.e: *wait*) which does not print anything as *wait* is never called for *grep hello README*.
- `trace 5 grep hello README`: It sets trace for system call number 5 (i.e: *read*) which occurs 4 times for `grep hello README`. *read* takes a file descriptor (integer), a destination for reading data (pointer), and number of bytes to read (integer) as arguments. The arguments are printed in their expected format.
- `trace 21 grep hello README`: It traces *close* system call which is called once here (as the file README will be closed once after reading it). *close* takes a file descriptor (integer) as argument.
- `trace 7 echo hello`: It traces *exec* system call which is called once here (as the echo command will be executed once). *exec* takes the name of the program (string) and command line arguments to that program (pointer). The arguments are printed in their expected format.

Hints

- You might need to add an extra field in the *proc* structure (see *kernel/proc.h*) to remember which system call the process wants to trace.
- The *syscall()* function in *kernel/syscall.c* is the point where the kernel decides which system call handler to invoke for a specific system call number, calls that handler and returns the return value. So it is a perfect place to modify to print the trace output. To print the syscall name, you might need to add an **array of syscall name**s in that file to index into.
- All the arguments to system calls go through one of: `argint, argaddr, argstr`. So, printing the arguments in their corresponding type should be easy.

Task 2: History

In this task, implement a system call *history* that will return the aggregated history of a system call (how many times it was called and the system time it consumed). Like **task - 1** it should take one argument, an integer *syscall_number* which denotes the system call number to trace for a user program. This time to get history about system call fork, a user program calls *history(SYS_fork)*, where *SYS_fork* is the syscall number of fork from *kernel/syscall.h*.

The return from your system call would be a pointer to a struct object. This struct object would contain:

1. the **name** of the system call
2. the number of times this system call was made
3. the total time consumed from boot up by this system call.

A sample structure can be:

```
struct syscall_stat{
  char syscall_name[16];
  int count;
  int accum_time;
};
```

You have to design a *history.c* similar to *trace.c* from **task-1** . For example, when you run *history 5*, the history user program will fetch the necessary info from the kernel and then print in the console from **user mode**. A sample output might be like this:

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ history 12
12:     syscall: sbrk, #: 1, time: 0
$ history 5
5:      syscall: read, #: 21, time: 86
$ history 5
5:      syscall: read, #: 31, time: 156
$ history 22
22:     syscall: history, #: 3, time: 0
$ history
1:      syscall: fork, #: 6, time: 0
2:      syscall: exit, #: 0, time: 0
3:      syscall: wait, #: 4, time: 1
4:      syscall: pipe, #: 0, time: 0
5:      syscall: read, #: 50, time: 296
6:      syscall: kill, #: 0, time: 0
7:      syscall: exec, #: 7, time: 2
8:      syscall: fstat, #: 0, time: 0
9:      syscall: chdir, #: 0, time: 0
10:     syscall: dup, #: 2, time: 0
11:     syscall: getpid, #: 0, time: 0
12:     syscall: sbrk, #: 5, time: 0
13:     syscall: sleep, #: 0, time: 0
14:     syscall: uptime, #: 0, time: 0
15:     syscall: open, #: 3, time: 0
16:     syscall: write, #: 656, time: 1
17:     syscall: mknod, #: 1, time: 0
18:     syscall: unlink, #: 0, time: 0
19:     syscall: link, #: 0, time: 0
20:     syscall: mkdir, #: 0, time: 0
21:     syscall: close, #: 1, time: 0
22:     syscall: history, #: 25, time: 0
```

Explaination

- *history 12* means return history for system call 12. sbrk is the system call for index 12. (Please refer to $syscall.h$). It has been called only once. Moreover, the time (xv6 ticks) consumed by it is 0.
- Notice the difference between the outputs of two *history 5*.
- *history 22* confirms our implementation is correct. Here we have used number 22 for history system call.
- Finally, history without any arguments should print history for all system calls.

You have to print this info from **user mode**. Printing anything in the kernel mode is not allowed.
Please refer to the system call **fstat** to get an idea of how to return a structure object pointer.

Locking

Xv6 is a multiprocessor system. There is a variable $CPUS$ in *Makefile* . What will happen if two processes running the same system call on different cpus. Suppose they incremented the counter for this system call at the exact same time. This may lead to one update not being recorded. This will be covered in detail in your theory classes. One solution to this situation is to use locks. Please follow *tickslock* to get an idea how to use locks in xv6.

Bonus Task

Linux terminal has a nice little command *exit*. But alas! Our favourite xv6 doesn't have one. Can you implement it? Do your own research and implement it.

Please don't copy. Viva will be there for this bonus task.

General Guideline

Don't forget to acquire and release locks when needed, look out for the **proc** struct in *kernel/proc.h* and **kmem** struct in *kernel/kalloc.c* . You should look at how other existing functions use the fields of those structs to get an idea. Remember xv6 is multi-core.

Submission Guideline

Start with a fresh copy of xv6 from the original repository. Make necessary changes for this offline. In this offline, you will submit just the changes done (i.e.: **a patch file**), not the entire repository.

Don't commit. Modify and create files that you need to. Then create a patch using the following command:

```
git add --all
git diff HEAD > {studentID}.patch
```

Where studentID = your own six-digit student ID (e.g., 1905001). Just submit the patch file, do not zip it. In the lab, during evaluation, we will start with a fresh copy of xv6 and apply your patch using the command:

```
git apply {studentID}.patch
```

Make sure to test your patch file after submission the same way we will run during the evaluation.

Please DO NOT COPY solutions from anywhere (your friends, seniors, internet, etc.). Any form of plagiarism (irrespective of source or destination), will result in getting -100% marks in this assignment. You have to protect your code.

Submission Deadline: Monday, July 03, 2023, 11:45 PM

Mark Distribution

Task No	Sub-task	Marks
1	Properly tracing system calls	15
	Tracing only for the calling process	10
	Printing system call name	5
	Printing with system call arguments	10
2	Designing history.c	10
	Counting system call	15
	Calculating system call time	15
	Using appropiate locking	15
	Proper submission	5
	Total	**100**
	Bonus task	25