January 2024 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 three tasks.

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 5 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. Don't forget to properly **clean the added field** upon process completion.
- 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: Reporting process related information

Add a system call named $info$ that returns the aggregated information of currently running processes as struct of $procInfo$. Structure $procInfo$ is defined as follow:

```
struct procInfo {
 int activeProcess;  // # of processes in RUNNABLE and RUNNING state
 int totalProcess;  // # of total possible processes
 int memsize;            // in bytes; summation of all active process
 int totalMemSize;   // in bytes; all available physical Memory
};
```

You need to also a add a user program $load$. The user program will have 2 input (childCount, allocationAmount): the number of children, it will create and the amount of malloc each child process do. In the user program, call fork $childCount$ times. After forking, in the child processes, do malloc of $allocationAmount$ size and sleep for a large chunk of time. After creating all the childs, the parent will sleep until all the childs are done allocating their memory (a sleep of a big time will suffice) and finally call the system call and output the info.

```
$ load 2 2048
Parent going to sleep.
Child is created.
Child allocated 2048 byte.
Child going to sleep.
Child is created.
```

```
Child allocated 2048 byte.
Child going to sleep.
Parent wake up.
Current system information:
Processes: 7/64
RAM   : 8.8/128 (in MB)
```

Note, the numbers are madeup here.

Hints

- The argument to your system call will be a pointer. You need to allocate the memory for the struct object in user space and pass the pointer to that struct object to kernel.
- The kernel and user space uses different page table. So, kernel can't directly access the struct object. You need to create another struct object in kernel space. So while returning back from system call you need to $copyout$ the kernel space object to user space object. Look for the usage of $copyin$ and $copyout$.
- To extract the argument of $info$ system call, as it is a pointer, you need to user $argaddr$.
- Be cautious while printing the allocated memory, because xv6 doesn't support float numbers.
- Study $KERNBASE$ and $PHYSTOP$ to get $totalMemSize$.

Task 3: !!

This task carries only 10% of the total marks. So if you are struggling with the previous tasks, please ignore this one.

In your bash class you've used !! command. This command at the execution time returns the most recently used command in the terminal (i.e. previous command). You need to add this command to xv6 kernel.

```
$ wc README
49 325 2354 README
$ !!
wc README
49 325 2354 README
```

Hints

- Study the $sh.c$ in detail: how fork and exec is working together to create a new process.
- You should keep track of commands which are executed within the terminal. This can be done in two places:
  - **user space:** The shell($sh.c$) saves the last used command (may be in a file or variable) and while running any new command it will check if it is !!. If true, it will replace itself with the previous command.
  - **kernel space:** Kernel has a variable to store last user command and provides two system calls to set and get last user command. Now, the shell just sets the variable and !! can simply get the last user command and do its own exec .

  You need to use the **second approach**.
- Don't forget the case where two processes simultaneously tries to update the last command.

Bonus Task

The bonus is an extension on Task 3. Add support for executing recently used commands. In this case, a sample command could be !! 3 for executing the last 3rd command that was executed. So Task 3 should equivalent to !! 1. See the example for better understading.

```
$ wc README
49 325 2354 README
$echo hello
hello
$ echo world
world
$!! 3
wc README
49 325 2354 README
$!! 3
echo hello
hello
```

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 try to increment a shared counter for any 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.

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., 2005001). 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: Saturday, September 28, 2024, 11:45 PM

Mark Distribution

Task No	Sub-task	Marks
1	Properly tracing system calls	25
	Printing system call name	5
	Printing with system call arguments	10
2	Designing load.c	5
	Formatted printing	5
	Info collection	10
	Correctly passing pointer	20
3	!! command	10
	Using appropriate locking	10
	Total	**100**
	Bonus task	5

Contact

If you have any question or confusion, please post it in discussion thread opened in moodle. If you don't get any reply by 24 hours, please mail me at kowshic@teacher.cse.buet.ac.bd.