ICT1012

OPERATING SYSTEMS

LABORATORY MANUAL

Lab1-w3: Lab Assignments

AY 2025/26

BACHELOR OF ENGINEERING

IN

INFORMATION AND COMMUNICATIONS TECHNOLOGY

Contents

Lab Assignment

Prerequisite

- Download "**lab.zip**" from "xSiTe" folder "**Labs/xv6labs-w3: Syscall**" to "**c:/ICT1012/xv6labs-w3**".
- Unzip and keep all folders and files directly under "**c:/ICT1012/xv6labs-w3**" without the "labs" sub-folder.

Assignment Task 1: handshake

- Your solution should be in the file "*user/handshake.c*".
- Write a user-level program that uses xv6 system calls to "*handshake*" a byte between two processes over a pair of pipes, one for each direction.
- The parent should send a byte to the child; the child should print *"<pid>: received <x> from parent*", where *"<pid>"* is its process ID and "*<x>"* is the byte, write the byte on the pipe to the parent, and exit; the parent should read the byte from the child, print *"<pid>: received <x> from child*", and exit.
- Hints:
  - Add the program to "*UPROGS"* in "*Makefile"*.
  - Use the pipe, fork, write, read, and "*getpid*" system calls.
  - User programs on xv6 have a limited set of library functions available to them. Refer to the list in "*user/user.h*"; the source (other than for system calls) is in "*user/ulib.c*", "*user/printf.c*", and "*user/umalloc.c*".
  - Changes to the file system persist across runs of "*qemu*"; to get a clean file system run "**make clean**" and then "**make qemu**".
- The following example illustrates "*handshake's*" behavior:

```
$ make qemu
...
init: starting sh
$ handshake 1
4: received 1 from parent
3: received 1 from child
$
```

- Exit xv6 shell with keys "**Ctrl + a**" followed by "**x**".
- Run all test cases for "*handshake*".

```
:/mnt/c/ICT1012/xv6labs-w3 $ ./grade-lab-syscall handshake
make: 'kernel/kernel' is up to date.
== Test handshake == handshake: OK (1.5s)
:/mnt/c/ICT1012/xv6labs-w3 $
```

Assignment Task 2: sniffer

The xv6 kernel isolates user programs from each other and isolates the kernel from user programs. An application cannot directly call a function in the kernel or in another user program. Instead, interactions occur only through system calls.

However, if there is a bug in the kernel's implementation of a system call, an attacker may be able to exploit that bug to break the isolation boundaries.

- To get a sense for how bugs can be exploited, a bug is intentionally introduced into xv6 and you're the goal of this task is to exploit that bug to steal a secret from another process.
- The bug is that the call to "*memset(mem, 0, sz)" in uvmalloc()* in *kernel/vm.c* to clear a newly-allocated page is omitted when compiling this lab.
- Similarly, when compiling "*kernel/kalloc.c"* for this lab the two lines that use "*memset*" to put garbage into free pages are omitted. The net effect of omitting these 3 lines (all marked by "*ifndef LAB_SYSCALL*") is that newly allocated memory retains the contents from its previous use.
- Thus, an application that calls "*sbrk()*" to allocate memory may receive pages that have data in them from previous uses.
- Despite the 3 deleted lines, xv6 mostly works correctly; it even passes most of "*usertests*".
- "*user/secret.c*" writes a secret string in its memory and then exits (which frees its memory).
- The goal is to add a few lines of code to "*user/sniffer.c*" to find the secret that a previous execution of "*secret.c*" wrote to memory, and to print the secret on a line by itself.
- "*sniffer.c*" must work with unmodified xv6 and unmodified "*secret.c*".
- You can change anything to help you experiment and debug, but must revert those changes before final testing and submitting.
- The "*secret*" program takes the secret as an argument.
- Test the "*sniffer*" program by running "*secret*" with some argument, then running "*sniffer*", and seeing whether "*sniffer*" prints exactly the argument passed to "*secret*".
- Hints:
    - Add the programs "*secret*" and "*sniffer*" to "*UPROGS*" in "*Makefile*".
    - Changes to the file system persist across runs of "*qemu*"; to get a clean file system run "**make clean**" and then "**make qemu**".
- The following example illustrates "*sniffer's*" behaviour:

```
…
$ secret ict1012
$ sniffer
ict1012
$
```

- Exit xv6 shell with keys "**Ctrl + a**" followed by "**x**".
- Run all test cases for "sniffer".

```
:/mnt/c/ICT1012/xv6labs-w3_instructor$ ./grade-lab-syscall sniffer
make: 'kernel/kernel' is up to date.
```

```
== Test sniffer == sniffer: OK (1.1s)
:/mnt/c/ICT1012/xv6labs-w3_instructor$
```

Assignment Task 3: monitor

- Your solution should be in the file "*user/monitor.c*".
- In this assignment add a new system call monitoring feature that may help you when debugging later labs.
- Create a new monitor system call that will control monitoring.
- It should take one argument, an integer "*mask*", whose bits specify which system calls to monitor.
- For example, to monitor the fork system call, a program calls "*monitor(1 << SYS_fork)*", where" *SYS_fork*" is a" *syscall*" number from "*kernel/syscall.h*".
- Modify the xv6 kernel to print a line when each system call is about to return, if the system call's number is set in the mask. The line should contain the process id, the name of the system call and the return value; No need to print the system call arguments.
- The monitor system call should enable tracing for the process that calls it and any children that it subsequently forks, but should not affect other processes.
- Hints:
    - Add *"$U/_monitor*" to "UPROGS" in "*Makefile*".
    - Run "*make qemu*". The compiler cannot compile "*user/monitor.c*", because the user-space stubs for the monitor system call don't exist yet.
    - Add a prototype for monitor to "*user/user.h*", a stub to "*user/usys.pl*", and a "*syscall*" number to "*kernel/syscall.h*".
    - The "*Makefile*" invokes the "*perl*" script "*user/usys.pl*", which produces "*user/usys.S*", the actual system call stubs, which use the RISC-V "*ecall*" instruction to transition to the kernel.
    - Once the compilation issues are settled, run "*monitor 32 grep hello README*"; it will fail because the system call in the kernel is not implemented yet.
    - Add a new variable "*uint32 monitor_mask;*" in the "*proc*" structure "*kernel/proc.h*".
    - Add a "*sys_monitor()*" function in "*kernel/sysproc.c*" that implements the new system call by remembering its argument in the variable "*monitor_mask*" (see "kernel/proc.h").
        - The functions to retrieve system call arguments from user space are in "*kernel/syscall.c*", and see examples of their use in "*kernel/sysproc.c*".

```
uint64
sys_monitor(void)
{
 int mask;
 argint(0, &mask);

 struct proc *p = myproc();
 p->monitor_mask = (uint32)mask;
```

```
  return 0;
}
```

- Add the new "*sys_monitor*" to the "*syscalls*" array in "*kernel/syscall.c*".

```
static uint64 (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
...
[SYS_close]   sys_close,
[SYS_monitor] sys_monitor,
};
```

- Add an array of "*syscall*" names to index into in "*kernel/syscall.c*".

```
static const char *syscall_names[] = {
[SYS_fork]    "fork",
[SYS_exit]    "exit",
[SYS_wait]    "wait",
[SYS_pipe]    "pipe",
[SYS_read]    "read",
[SYS_kill]    "kill",
[SYS_exec]    "exec",
[SYS_fstat]   "fstat",
[SYS_chdir]   "chdir",
[SYS_dup]     "dup",
[SYS_getpid]  "getpid",
[SYS_sbrk]    "sbrk",
[SYS_pause]   "pause",
[SYS_uptime]  "uptime",
[SYS_open]    "open",
[SYS_write]   "write",
[SYS_mknod]   "mknod",
[SYS_unlink]  "unlink",
[SYS_link]    "link",
[SYS_mkdir]   "mkdir",
[SYS_close]   "close",
[SYS_monitor] "monitor",
};
```

- Modify "*kfork()*" (see "*kernel/proc.c*") to copy the monitor mask from the parent to the child process. Add:

```
np->monitor_mask = p->monitor_mask;
```

    - Modify the "*syscall()*" function in "*kernel/syscall.c*" to print the monitor output.
    - Changes to the file system persist across runs of "*qemu*"; to get a clean file system run "**make clean**" and then "**make qemu**".
- The following example illustrates "*monitor's*" behavior:

```
init: starting sh
$ monitor 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 965
3: syscall read -> 437
3: syscall read -> 0
$ monitor 2147483647 grep hello README
4: syscall monitor -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 965
4: syscall read -> 437
4: syscall read -> 0
4: syscall close -> 0
$ grep hello README
$ monitor 2 usertests forkforkfork
usertests starting
test forkforkfork: 6: syscall fork -> 7
7: syscall fork -> 8
8: syscall fork -> 9
9: syscall fork -> 10
9: syscall fork -> 11
9: syscall fork -> 12
...
9: syscall fork -> -1
8: syscall fork -> -1
10: syscall fork -> -1
OK
ALL TESTS PASSED
$
```

- In the first example above, trace invokes grep tracing just the read system call. The 32 is "*1<<SYS_read*".
- In the second example, trace runs grep while tracing all system calls; the "*2147483647*" (0x7FFFFFFF) has all 31 low bits set.
- In the third example, the program isn't traced, so no trace output is printed.
- In the fourth example, the fork system calls of all the descendants of the "*forkforkfork*" test in "*usertests*" are being traced.
- Solution is correct if the program behaves as shown above (though the process IDs may be different).
- Exit xv6 shell with keys "**Ctrl + a**" followed by "**x**".

- Run all test cases for "*monitor*".

```
:/mnt/c/ICT1012/xv6labs-w3_instructor$ ./grade-lab-syscall monitor
make: 'kernel/kernel' is up to date.
== Test monitor 32 grep == monitor 32 grep: OK (2.0s)
== Test monitor close grep == monitor close grep: OK (0.7s)
== Test monitor exec + open grep == monitor exec + open grep: OK (0.9s)
== Test monitor all grep == monitor all grep: OK (1.0s)
== Test monitor nothing == monitor nothing: OK (1.0s)
== Test monitor children == monitor children: OK (5.3s)
:/mnt/c/ICT1012/xv6labs-w3_instructor$
```

Assignment submission

1. Execute "**make grade**" to test "handshake", "sniffer" and "monitor" tasks.

```
:/mnt/c/ICT1012/xv6labs-w3_instructor$ make grade
…
make[1]: Leaving directory '/mnt/c/ICT1012/xv6labs-w3_instructor'
== Test handshake ==
$ make qemu-gdb
handshake: OK (8.1s)
== Test sniffer ==
$ make qemu-gdb
sniffer: OK (0.7s)
== Test monitor 32 grep ==
$ make qemu-gdb
monitor 32 grep: OK (1.0s)
== Test monitor close grep ==
$ make qemu-gdb
monitor close grep: OK (1.1s)
== Test monitor exec + open grep ==
$ make qemu-gdb
monitor exec + open grep: OK (0.9s)
== Test monitor all grep ==
$ make qemu-gdb
monitor all grep: OK (1.0s)
== Test monitor nothing ==
$ make qemu-gdb
monitor nothing: OK (1.1s)
== Test monitor children ==
$ make qemu-gdb
monitor children: OK (4.9s)
Score: 65/65
:/mnt/c/ICT1012/xv6labs-w3_instructor$
```

2. Execute "**make zipball**" to create "**lab.zip**".

3. Upload the "**lab.zip**" file to Gradescope assignment "**xv6labs-w3: Syscall**" to get auto graded.