CSC10007 – OPERATING SYSTEM

PROJECT 2 – SYSTEM CALL

No.	Name	Student ID	Contribute
1.	Le Quang Khai	22120148	50%
2.	Nguyen Quang Thang	22120333	50%

No.	Exercise	Person in charge
1.	Using gdb	Le Quang Khai, Nguyen Quang Thang
2.	System call tracing	Nguyen Quang Thang
3.	Sysinfo	Le Quang Khai
4.	Load average (challenge)	Le Quang Khai
5.	Report	Le Quang Khai, Nguyen Quang Thang

Table of Contents

1. Setup 2
2. Using gdb (easy) 2
   2.1. Documents 2
   2.2. Step by step 2
   2.3. Answer questions 9
3. Add a new system call 10
4. System call tracing (moderate) 11
   4.1. Declare system call `tracing` 11
   4.2. Implement system call `tracing` 12
   4.3. Implement user level program `tracing` 14
   4.4. Result 15
   4.5. Git diff 16
5. Sysinfo (moderate) 16
   5.1. Declare system call `sysinfo` 16
   5.2. The idea of `copyout()` 17
   5.3. Implement system call `sysinfo` 18
   5.4. Result 19
   5.5. Git diff 19
6. Load average (challenge) (moderate) 20

1. Setup

In this lab, we will use either `gdb-multiarch` or `riscv64-unknown-elf-gdb`. I will be using `gdb-multiarch` because I am unable to install `riscv64-unknown-elf-gdb`.

```
$ sudo apt install gdb-multiarch
```

For separate terminals, `tmux` will be used. Check here for more details about `tmux`.

```
$ sudo apt install tmux
```

To start the lab, switch to the syscall branch:

```
$ git fetch
$ git checkout -b syscall
$ make clean
```

Note: In this report, the figures will be placed below the text that describes them. While this is unconventional, we have chosen this format so that even if the figures are unclear, the report can still be understood by reading the accompanying text.

2. Using gdb (easy)

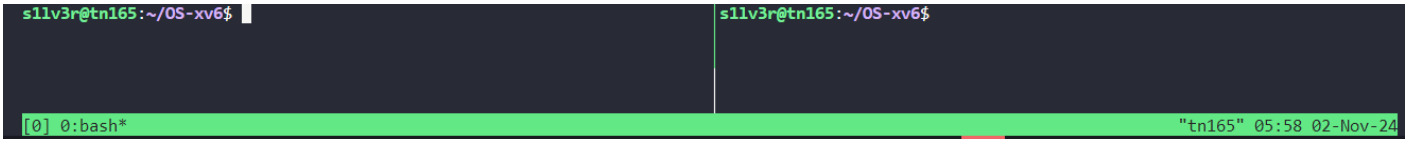
2.1. Documents

- To learn more about how to run GDB and the common issues that can arise when using GDB, check out GDB Guidance.
- For debugging tips, check out labs guidance.

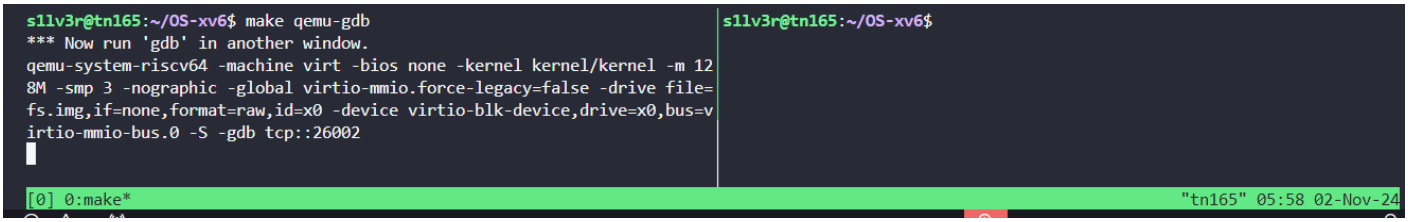
2.2. Step by step

Note: Make sure to stay in the project's root directory; otherwise, it won't work.

Start `tmux`:

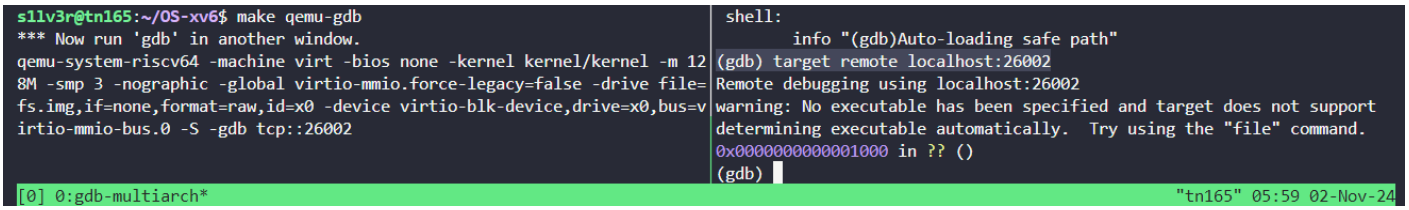


Start `gdb-mode` in the first window (check out GDB Guidance):



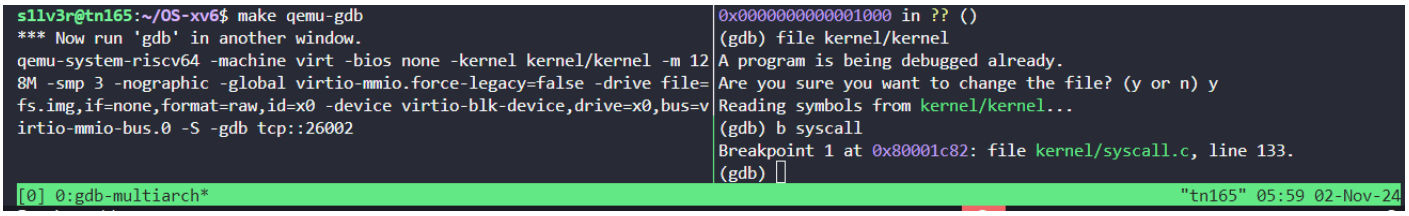
Start `gdb` in the second window:

```
$ gdb-multiarch
(gdb) target remote localhost:26002
```



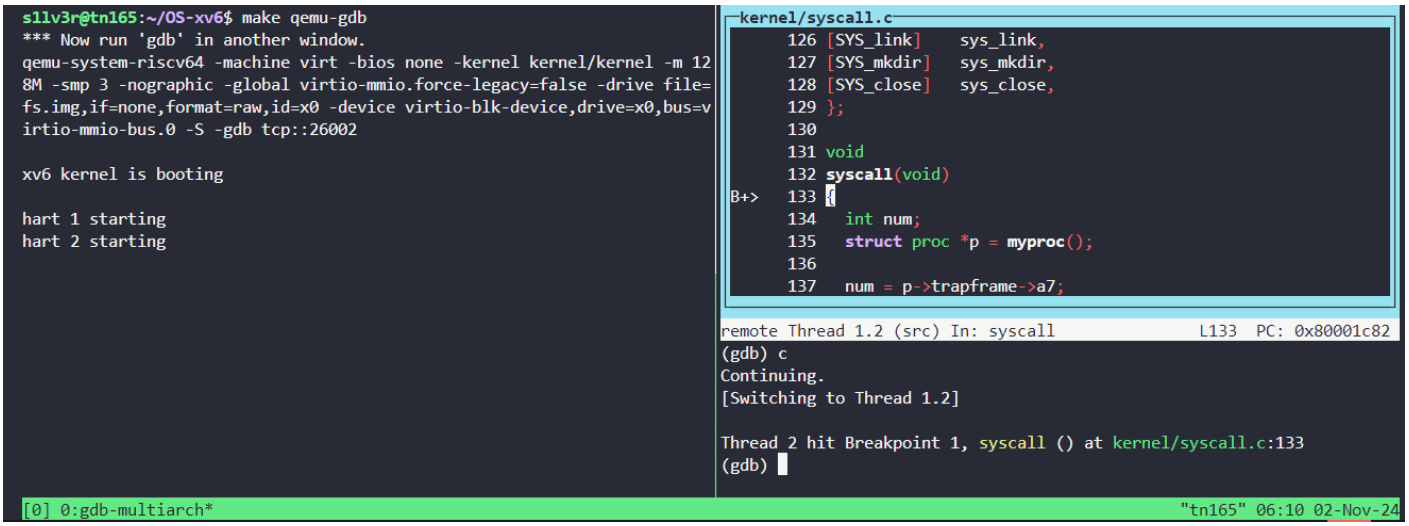
Load `kernel/kernel` file to debug (this is a binary that has all kernel code):

```
(gdb) file kernel/kernel
(gdb) b syscall
```



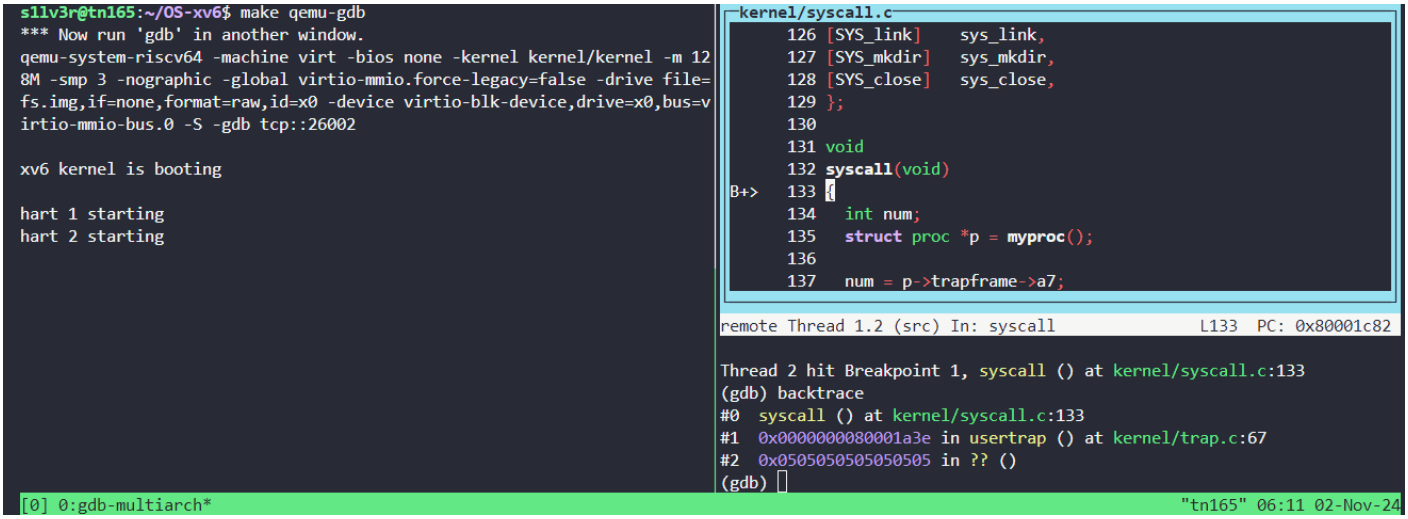
Use the following command to see the source code:

```
(gdb) layout src
(gdb) c
```



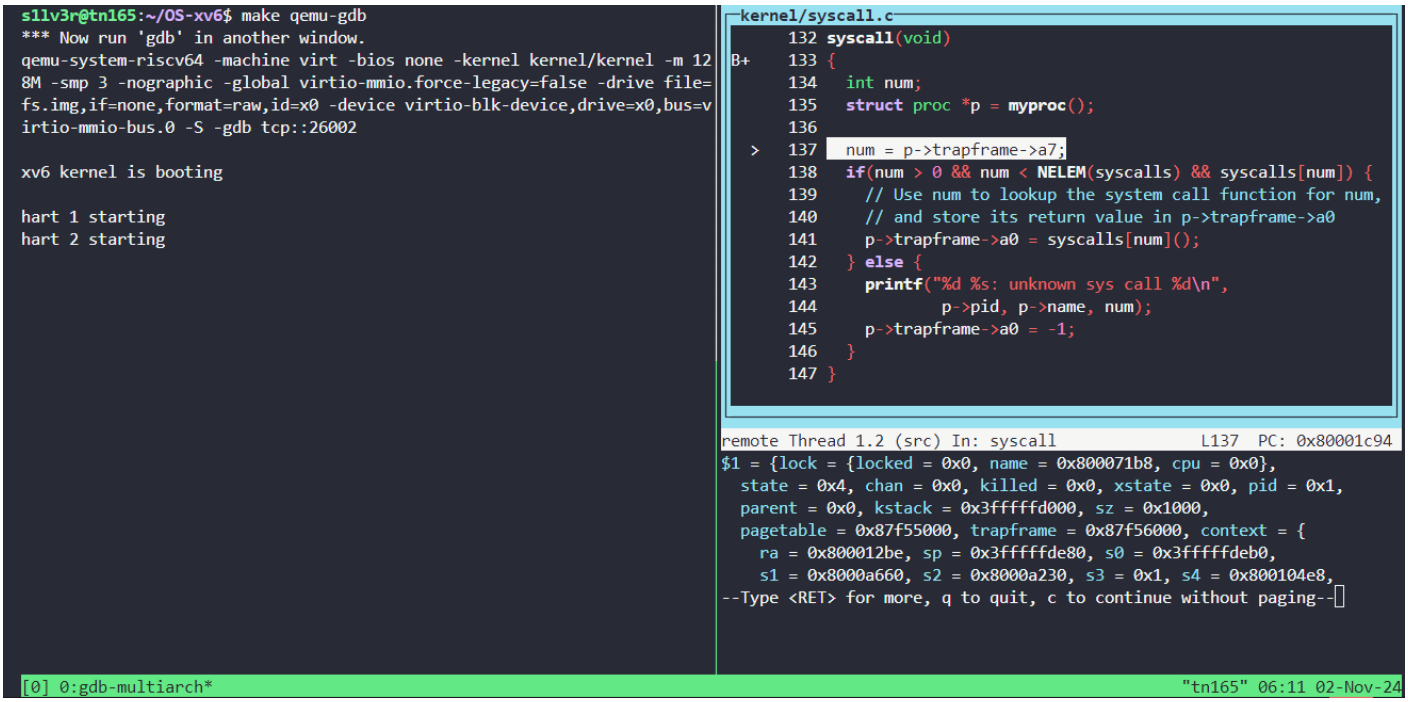
Keep hitting `c` to continue the program, until it meet the breakpoint again, or program is done.

Use the `backtrace` command to view the function that invoked the syscall:

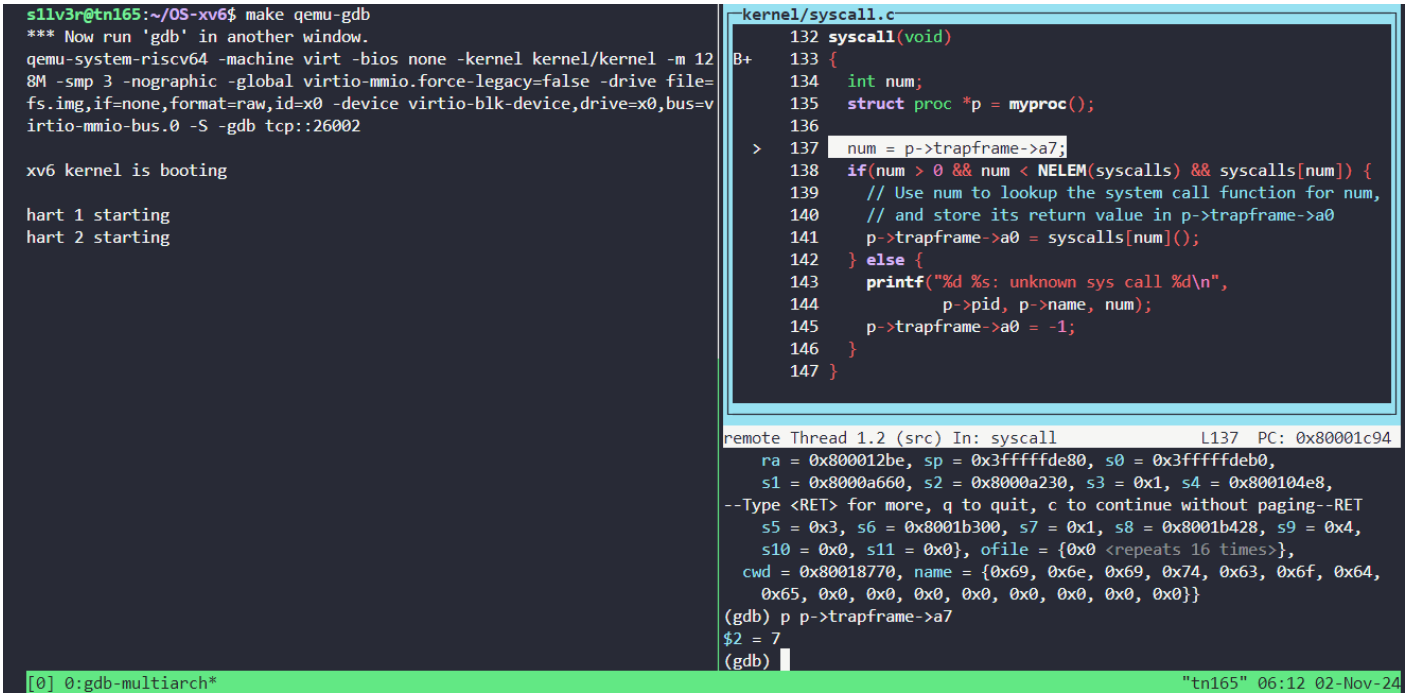


Type n a few times to step past `struct proc *p = myproc();`. Once past this statement, type `p /x *p`, which prints the current process's `proc struct` (see `kernel/proc.h`) in hex.

```
(gdb) p /x *p
```



Type `p p->trapframe->a7` and we have the value of `p->trapframe->a7` is 7.



Look at `user/initcode.S` file, the system call number will be stored in register `a7`.

```
#include "syscall.h"

# exec(init, argv)
.globl start
start:
        la a0, init
        la a1, argv
        li a7, SYS_exec
        ecall
```

Look at `kernel/syscall.h` file, system call number 7 is `SYS_exec`.

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
```

Note: All above commands have the following syntax:

```
(gdb) p /x <<register>>
```

Here, `p` is used to display information, `/x` specifies hexadecimal output (`/o` for octal, `/d` for decimal), and `<<register>>` is the register whose contents you want to view.

See the mode that the CPU is in as follows:

```
(gdb) p /x $sstatus
$3 = 0x200000022
```

Now we move and test the case that cause the xv6 kernel to panic.

In `kernel/syscal.c` file, replace `num = p->trapframe->a7;` with `num = * (int *) 0;`.

```
131 void
132 syscall(void)
133 {
134   int num;
135   struct proc *p = myproc();
136
137   // num = p->trapframe->a7;
138   num = * (int *) 0;
139   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
140     // Use num to lookup the system call function for num, call it,
141     // and store its return value in p->trapframe->a0
142     p->trapframe->a0 = syscalls[num]();
143   } else {
144     printf("%d %s: unknown sys call %d\n",
145             p->pid, p->name, num);
146     p->trapframe->a0 = -1;
147   }
148 }
149
```

Then run `make qemu`. This error will show up:

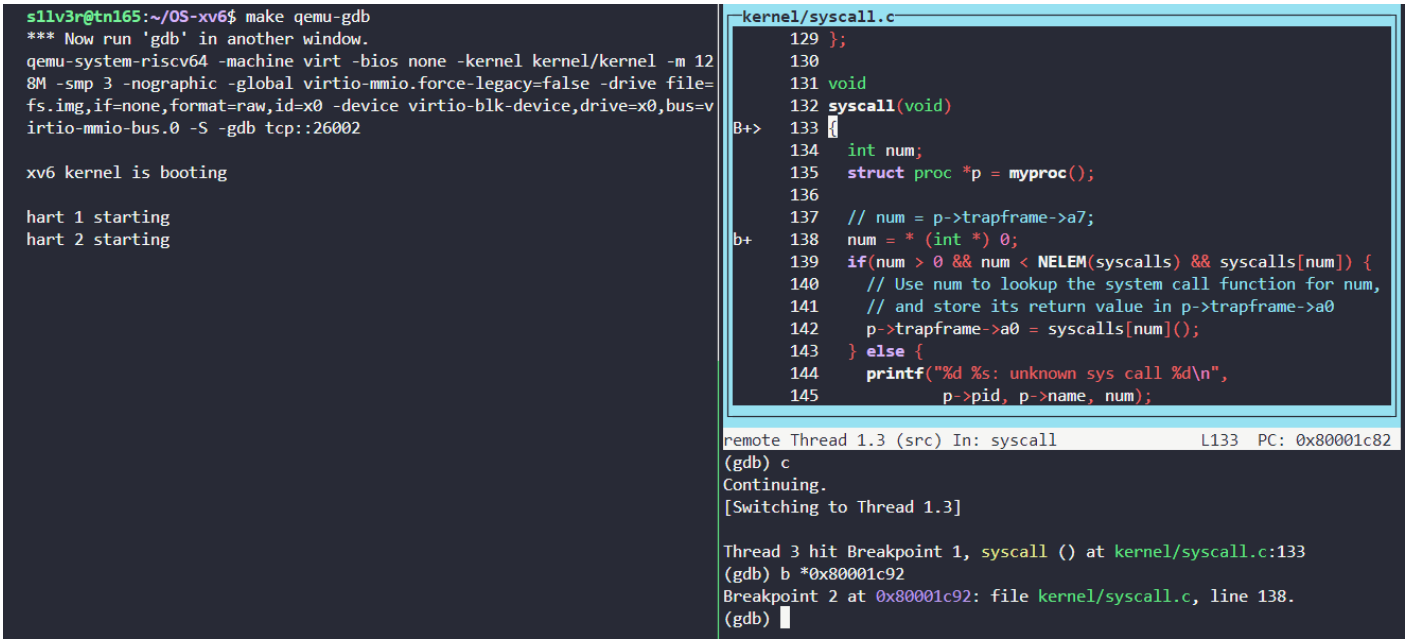
```
xv6 kernel is booting

hart 1 starting
hart 2 starting
scause=0xd sepc=0x80001c92 stval=0x0
panic: kerneltrap
```

To track down the source of a kernel page-fault panic, search for the `sepc` value printed for the panic you just saw in the file `kernel/kernel.asm`, which contains the assembly for the compiled kernel.

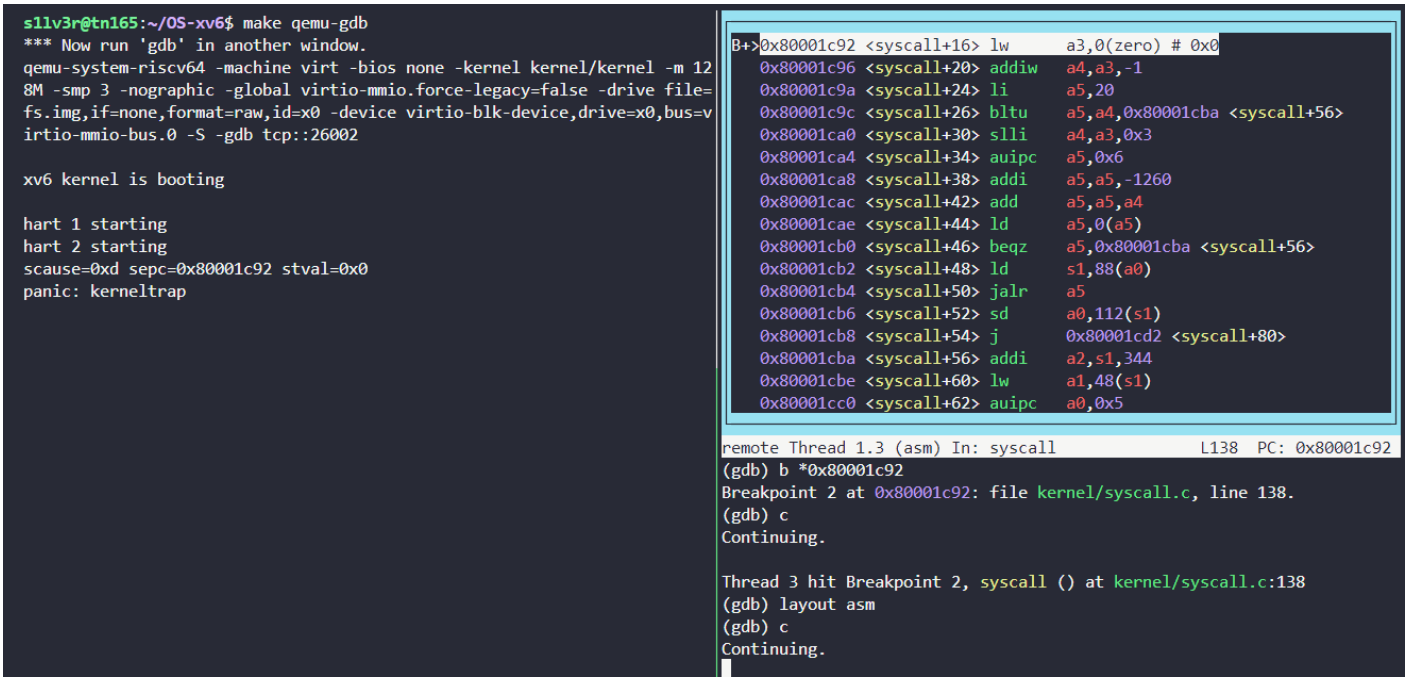
```
kernel.asm
kernel > kernel.asm
4293  {
4303
4304    // num = p->trapframe->a7;
4305    num = * (int *) 0;
4306    80001c92:   00002683          lw  a3,0(zero) # 0 <_entry-0x80000000>
```

We can see the assembly instruction the kernel is panicing at is `lw a3,0(zero)` and the register corresponds to the variable `num` is `a3`.

To inspect the state of the processor and the kernel at the faulting instruction, fire up `gdb`, and set a breakpoint at the faulting `epc`:



Switch to assembly code and continue running the program:

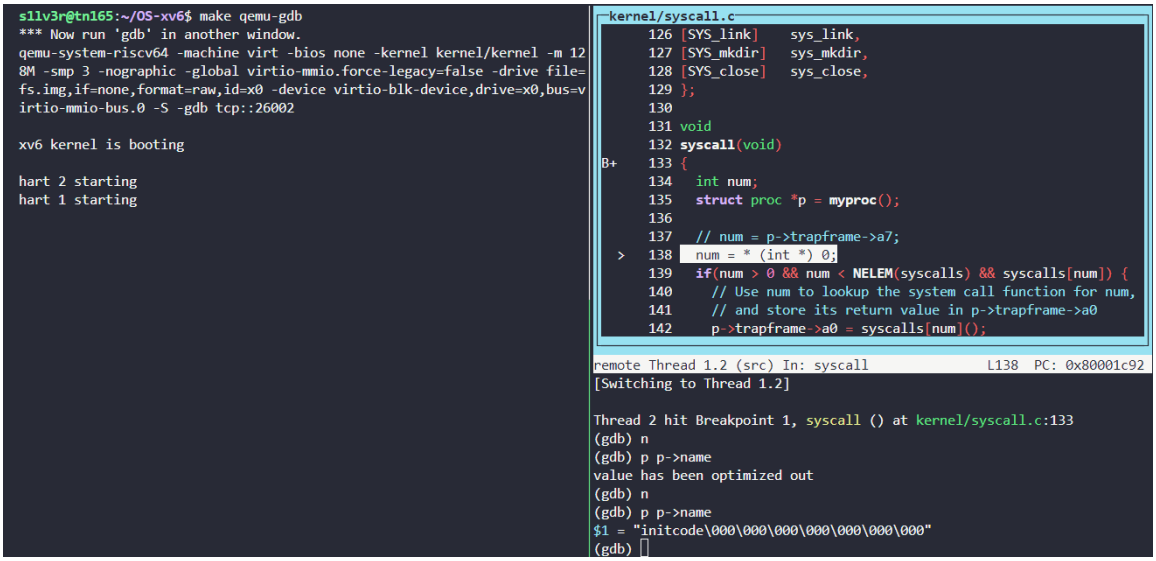


So, the faulting assembly instruction is the same as the one we found above.

The Kernel crashed because it tried to access address `0`, which is **not mapped** in the kernel's address space, leading to a **page fault**. This is confirmed by the `scause` value of `0xd`, corresponding to **Load Page Fault**, indicating the fault occurred when loading data from an invalid address.

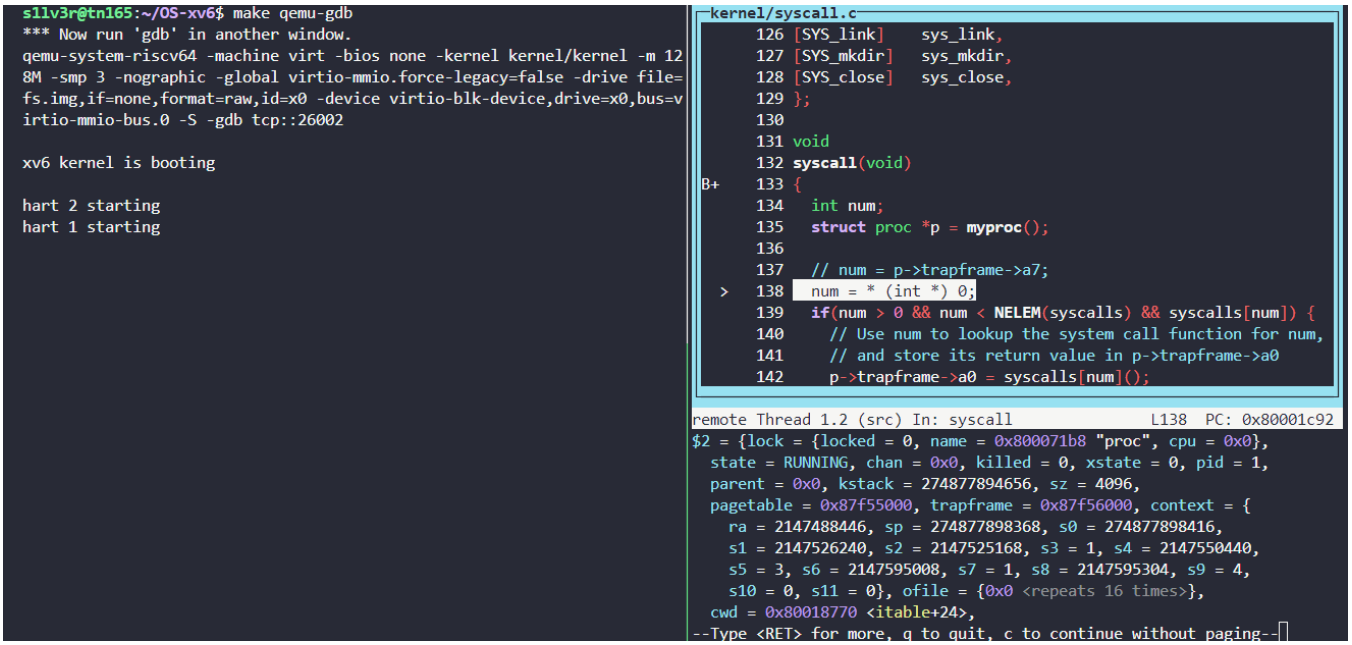
To find out which user process was running when the kernel paniced, print out the process's name by:

```
(gdb) p p->name
```



So the process's name is **initcode**. Print the process information by

```
(gdb) p *p
```



Look carefully, we have `pid = 1`.

2.3. Answer questions

Looking at the backtrace output, which function called syscall?

The `usertrap()` function called `syscall()` function.

What is the value of `p->trapframe->a7` and what does that value represent? (Hint: look `user/initcode.S`, the first user program xv6 starts.)

The value of `p->trapframe->a7` is 7, representing the system call number `SYS_exec`. This is the `exec system call` executed in `initcode.S` to start the `/init` process.

What was the previous mode that the CPU was in?

The previous mode that the CPU was in is **0x200000022**.

Write down the assembly instruction the kernel is panicing at.
Which register corresponds to the variable `num`?

The assembly instruction the kernel is panicing at is `lw a3,0(zero)` and the register corresponds to the variable `num` is `a3`.

Why does the kernel crash?
Hint: look at figure 3-3 in the text; is address 0 mapped in the kernel address space? Is that confirmed by the value in scause above? (See description of scause in RISC-V privileged instructions)

The Kernel crashed because it tried to access address `0`, which is **not mapped** in the kernel's address space, leading to a **page fault**. This is confirmed by the `scause` value of `0xd`, corresponding to **Load Page Fault**, indicating the fault occurred when loading data from an invalid address.

What is the name of the binary that was running when the kernel paniced? What is its process id (pid)?

The binary running when the kernel paniced is **initcode**. `pid = 1`.

3. Add a new system call

Assume that a new system call is about to be added, named `hello`.

1. Add a prototype for the system call to `user/user.h`

```
int hello(int);
```

2. Add a stub to `user/usys.pl`

```
entry("hello");
```

3. Add a syscall number to `kernel/syscall.h`

```
#define SYS_hello  300
```

4. Add the new syscall into `kernel/syscall.c`

```
// Prototypes for the functions that handle system calls.
extern uint64 sys_hello(void);

// An array mapping syscall numbers from syscall.h
// to the function that handles the system call.
static uint64 (*syscalls[])(void) = {
…
[SYS_hello]   sys_hello,
};
```

5. Add `sys_hello` (a function takes void as argument and returns `uint64`) in `kernel/sysproc.c`. This function do fetch arguments about the system call and return values.

```
uint64
sys_hello(void)
{
  // code body
  // return value of uint64
}
```

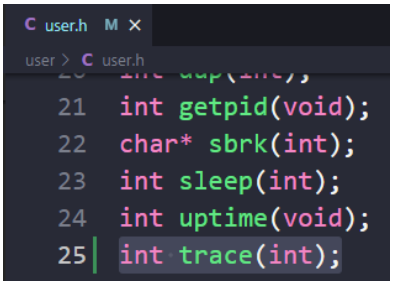
6. Create a file and install the syscall in the kernel.

4. System call tracing (moderate)

In this assignment you will add a system call tracing feature that may help you when debugging later labs. You'll create a new `trace` *system call that will control tracing. It should take one argument, an integer "mask", whose bits specify which system calls to trace. For example, to trace the fork system call, a program calls* `trace(1 << SYS_fork)`*, where* `SYS_fork` *is a syscall number from* `kernel/syscall.h`*. You have to 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; you don't need to print the system call arguments. The* `trace` *system call should enable tracing for the process that calls it and any children that it subsequently forks, but should not affect other processes.*

4.1. Declare system call `tracing`

1. Add a prototype for the system call to `user/user.h`



2. Add a stub to `user/usys.pl`

3. Add a syscall number to `kernel/syscall.h`

```
kernel > C syscall.h
19  #define SYS_unlink 18
20  #define SYS_link   19
21  #define SYS_mkdir  20
22  #define SYS_close  21
23  #define SYS_trace  22
24
```

4. Add the new syscall into `kernel/syscall.c`

```
129  [SYS_close]   sys_close,
130  [SYS_trace]   sys_trace,
131  };
132  extern uint64 sys_trace(void);
133
134  void
135  syscall(void)
136  {
```

5. Add `sys_trace` (a function takes void as argument and returns `uint64`) in `kernel/sysproc.c`

```
kernel > C sysproc.c
95  // return *
96  uint64
97  sys_trace(void)
98  {
99    ...
100 }
101
```

4.2. Implement system call `tracing`

As the hint said, create a new variable for tracing in the `proc` structure (in `kernel/proc.h`).

```
kernel > C proc.h
85  struct proc {
104   struct file *ofile[NOFILE];  // Open files
105   struct inode *cwd;           // Current directory
106   char name[16];               // Process name (debugging)
107   int tracemask;               // trace() to trace system call (debugging)
108 };
109
```

The function `uint64 sys_trace(void)` in `kernel/sysproc.c` should take argument and assign the mask value into current process's tracemask value. Use `void argint(<pos>, <ptr>)` defined in `kernel/syscall.c` to take the argument at `<pos>` position and assign to `<ptr>`. Use `myproc()` to get current process's pointer.

```c
// save tracemask to process info
uint64
sys_trace(void)
{
  int mask;
  argint(0, &mask);
  myproc()->tracemask = mask;
  return 0;
}
```

Modify `fork()` in `kernel/proc.c` to copy the trace mask from the parent to the child process. **Note:** `kernel/proc.c` file, not `kernel/sysproc.c` file.

```c
{
  safestrcpy(np->name, p->name, sizeof(p->name));

  pid = np->pid;

  // copy trace mask from parent process
  // to child process
  np->tracemask = p->tracemask;

```

Add an array of syscall names to help printing, as the hint said.

```c
// An array mapping syscall numbers from syscall.h
// to the function that handles the system call.
static uint64 (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_trace]   sys_trace,
};

// An array mapping syscall numbers from syscall.h
// to the name of the function
// that handles the system call.
char* syscallnames[] = {
[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_sleep]   "sleep",
[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_trace]   "trace",
};
```

Get into `kernel/syscall.c`, add the code block to print out a line when each system call is about to return, **if** the system call's number is set in the mask.

```c
void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // Use num to lookup the system call function for num, call it,
    // and store its return value in p->trapframe->a0
    p->trapframe->a0 = syscalls[num]();

    // if the tracemask is required
    if(p->tracemask & (1 << num)){
      printf("%d: system call %s -> %ld\n", p->pid, syscallnames[num], p->trapframe->a0);
    }

  } else {
    printf("%d %s: unknown sys call %d\n",
           p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

Note: cannot use `p->name` in the `printf` since it prints the current process's name, not the one it is tracing.

Explain the syscall function: The kernel uses the number in register `a7` to call the desired system call. This value is stored in the variable `num`. The function then checks if the value is valid—ensuring it falls within the valid range and that there is an existing function pointer to execute the system call. If the conditions are met, execute the system call and store the output in `p->trapframe->a0`.

4.3. Implement user level program `tracing`

Firstly, we couldnot find the `user/trace.c` as mentioned in the lab manual. Therefore, we implemented it by ourself. Then we find out that we should **change the branch in the original repo from MIT**, not ours!

We think the file can be written better like below to **prevent copying** the array.

```c
#include "kernel/param.h"
#include "kernel/types.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
  if(argc < 3 || (argv[1][0] < '0' || argv[1][0] > '9')){
    fprintf(2, "Usage: %s mask command\n", argv[0]);
    exit(1);
  }

  if (trace(atoi(argv[1])) < 0) {
    fprintf(2, "%s: trace failed\n", argv[0]);
    exit(1);
  }

  exec(argv[2], argv + 2);
  exit(0);
}
```

Pass `argv + 2` into `exec()` since `argv[0]` is trace program and `argv[1]` is tracemask.

4.4. Result

Add `$U/_trace\` to `UPROGS` and run the OS via `make qemu`.

```
s1lv3r@tn165:~/OS-xv6$ make qemu
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_UTIL -DLAB_UTIL -MD -mcmodel=medany -fno-common -nostdlib -fno-builtin-strncpy -fno-builtin-strncmp -fno-builtin-strlen -fno-builti
n-memset -fno-builtin-memmove -fno-builtin-memcmp -fno-builtin-log -fno-builtin-bzero -fno-builtin-strchr -fno-builtin-exit -fno-builtin-malloc -fno-builtin-putc -fno-builtin-free -fno-builtin-memcpy -Wno-main -fno-b
uiltin-printf -fno-builtin-fprintf -fno-builtin-vprintf -I. -fno-stack-protector -fno-pie -no-pie   -c -o kernel/syscall.o kernel/syscall.c
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_UTIL -DLAB_UTIL -MD -mcmodel=medany -fno-common -nostdlib -fno-builtin-strncpy -fno-builtin-strncmp -fno-builtin-strlen -fno-builti
n-memset -fno-builtin-memmove -fno-builtin-memcmp -fno-builtin-log -fno-builtin-bzero -fno-builtin-strchr -fno-builtin-exit -fno-builtin-malloc -fno-builtin-putc -fno-builtin-free -fno-builtin-memcpy -Wno-main -fno-b
uiltin-printf -fno-builtin-fprintf -fno-builtin-vprintf -I. -fno-stack-protector -fno-pie -no-pie   -c -o kernel/sysproc.o kernel/sysproc.c
riscv64-linux-gnu-ld -z max-page-size=4096 -T kernel/kernel.ld -o kernel/kernel kernel/entry.o kernel/kalloc.o kernel/string.o kernel/main.o kernel/vm.o kernel/proc.o kernel/swtch.o kernel/trampoline.o kernel/trap.o
kernel/syscall.o kernel/sysproc.o kernel/bio.o kernel/fs.o kernel/log.o kernel/sleeplock.o kernel/file.o kernel/pipe.o kernel/exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o kernel/virtio_disk.o kernel/start
.o kernel/console.o kernel/printf.o kernel/uart.o kernel/spinlock.o
riscv64-linux-gnu-ld: warning: kernel/kernel has a LOAD segment with RWX permissions
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel/kernel.sym
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=v
irtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$  trace 2147483647 grep hello README
3: system call trace -> 0
3: system call exec -> 3
3: system call open -> 3
3: system call read -> 1023
3: system call read -> 971
3: system call read -> 409
3: system call read -> 0
3: system call close -> 0
```

Althrough our output is as same as the lab manual and the regex in `grade-lab-syscall` checker, we have no idea why the checker give us result FAIL.

```
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel/kernel.sym
== Test trace 32 grep == trace 32 grep: FAIL (2.5s)
    ...
        3: system call read -> 1023
        3: system call read -> 971
        3: system call read -> 409
        3: system call read -> 0
        $ qemu-system-riscv64: terminating on signal 15 from pid 193898 (make)
    MISSING '^\d+: syscall read -> \d+'
    QEMU output saved to xv6.out.trace_32_grep
== Test trace close grep == trace close grep: FAIL (0.4s)
    ...
        hart 1 starting
        init: starting sh
        $ trace 2097152 grep hello README
        3: system call close -> 0
        $ qemu-system-riscv64: terminating on signal 15 from pid 194403 (make)
    MISSING '^\d+: syscall close -> 0'
    QEMU output saved to xv6.out.trace_close_grep
== Test trace exec + open grep == trace exec + open grep: FAIL (0.8s)
    ...
        init: starting sh
        $ trace 32896 grep hello README
        3: system call exec -> 3
        3: system call open -> 3
        $ qemu-system-riscv64: terminating on signal 15 from pid 194442 (make)
    MISSING '^\d+: syscall exec -> 3'
    QEMU output saved to xv6.out.trace_exec_open_grep
== Test trace all grep == trace all grep: FAIL (1.0s)
    ...
        3: system call read -> 971
        3: system call read -> 409
        3: system call read -> 0
        3: system call close -> 0
        $ qemu-system-riscv64: terminating on signal 15 from pid 194472 (make)
    MISSING '^\d+: syscall trace -> 0'
    QEMU output saved to xv6.out.trace_all_grep
== Test trace nothing == trace nothing: OK (1.1s)
== Test trace children == trace children: FAIL (18.6s)
    ...
        9: system call fork -> -1
        OK
        3: system call fork -> 66
        ALL TESTS PASSED
        $ qemu-system-riscv64: terminating on signal 15 from pid 194540 (make)
    MISSING '3: syscall fork -> 4'
    QEMU output saved to xv6.out.trace_children
s1lv3r@tn165:~/OS-xv6-lab$ make clean
```

4.5. Git diff

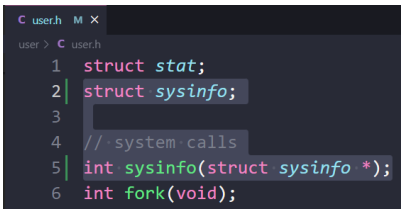
It is put in `docs/lab-2-syscall/trace.patch`.

5. Sysinfo (moderate)

In this assignment you will add a system call, `sysinfo`*, that collects information about the running system. The system call takes one argument: a pointer to a* `struct sysinfo` *(see* `kernel/sysinfo.h`*). The kernel should fill out the fields of this struct: the* `freemem` *field should be set to the number of bytes of free memory, and the* `nproc` *field should be set to the number of processes whose* `state` *is not* `UNUSED`*. We provide a test program* `sysinfotest`*; you pass this assignment if it prints* `"sysinfotest: OK"`*.*

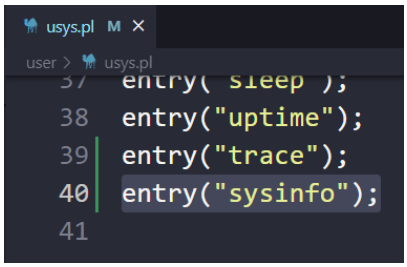
5.1. Declare system call `sysinfo`

1. Add a prototype for the system call to `user/user.h`

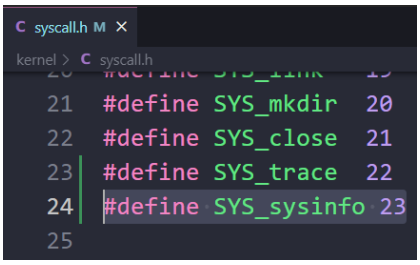


   Struct sysinfo is defined in `kernel/sysinfo.h`.

2. Add a stub to `user/usys.pl`



3. Add a syscall number to `kernel/syscall.h`



4. Add the new syscall into kernel/syscall.c

```
syscall.c M ×
kernel > C syscall.c
130  [SYS_close]   sys_close,
131  [SYS_trace]   sys_trace,
132  [SYS_sysinfo] sys_sysinfo,
133  };
134
135  extern uint64 sys_sysinfo(void);
```

5. Add `sys_sysinfo` into `kernel/sysproc.c`

```
sysproc.c M ×
kernel > C sysproc.c
105  #include "sysinfo.h"
106
107  // return ...
108  uint64
109  sys_sysinfo(void)
110  {
111    ...
112  }
113
```

Note: must have `#include "sysinfo.h"` since the struct sysinfo is defined in that header.

5.2. The idea of `copyout()`

Imagine Alice has two friends, **File Keeper** and **File Checker**. Each has a job to help Alice find out what's inside a toy box without you looking herself.

Here's how the job works:

1. **File Keeper's Job** (`sys_fstat()`):

* **File Keeper** is the first one Alice asks, "What toys do I have in box #1?"

* **File Keeper** goes to that box, but they don't actually know what's inside—so they call their friend, **File Checker**, to take a look.

2. **File Checker's Job** (`filestat()`):

* **File Checker** is the one who knows how to look inside the box. So, they open the box and write down what's inside.

* Once they know, **File Checker** has to give Alice the list of toys, but they can't just hand it to her because she is outside (like the "user space").

* So, they use a little paper slide called `copyout()` to safely pass you the list through a window.

3. **Using the Paper Slide** (`copyout()`):

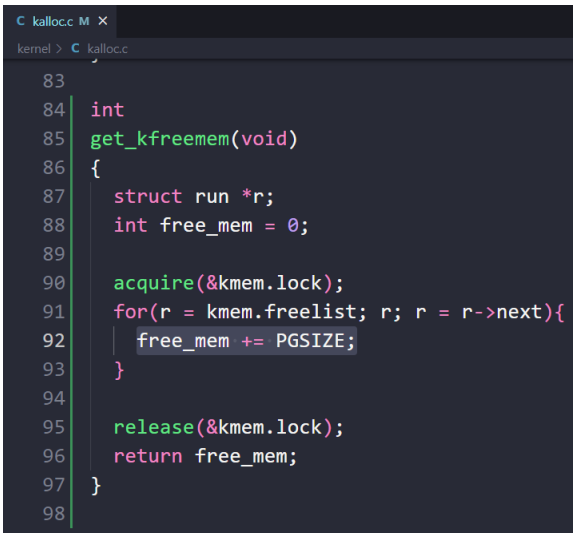
* `copyout()` is like the special paper slide that passes notes through the window from **File Checker** to Alice without letting her stepping inside.

* **File Checker** writes down, “You have 3 blocks and a teddy bear,” puts it in **copyout()**, and sends it to Alice.

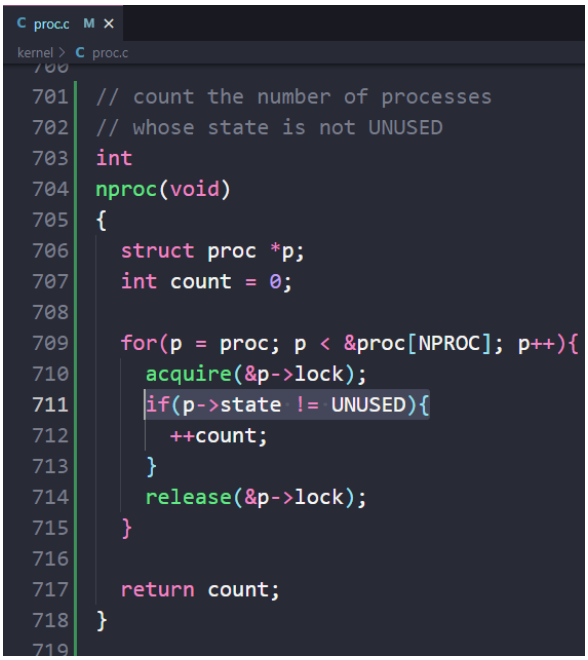
That’s how `copyout()` helps safely pass information from the inside (kernel space) to the outside (user space).

5.3. Implement system call `sysinfo`

To collect the amount of free memory, a function is added to `kernel/kalloc.c`



To collect the number of processes, a function is added to `kernel/proc.c`



Now they will be called in `sys_sysinfo()` function in `kernel/sysproc.c`. The book said “The inter-module interfaces are defined in `defs.h` (`kernel/defs.h`)”, therefore declare them into `kernel/defs.h` header.

```
C defs.h M ×
kernel > C defs.h
192  #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
193
194  struct sysinfo;
195
196  // kalloc.c
197  int             get_kfreemem(void); //sysinfo
198
199  // proc.c
200  int             nproc(void); //sysinfo
201
```

Last, define function `sys_sysinfo` in `kernel/sysproc.c` file.

```
C sysproc.c M ×
kernel > C sysproc.c
107  // get system info and return by using page table
108  uint64
109  sys_sysinfo(void)
110  {
111    struct sysinfo info;
112    struct proc *p = myproc();
113    uint64 addr;
114
115    // Get address of sysinfo pointer from user space
116    argaddr(0, &addr);
117
118    // Call update informations into struct info
119    info.freemem = get_kfreemem();
120    info.nproc = get_nproc();
121
122    // Copy struct info into page table
123    if(copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0){
124      return -1;
125    }
126
127    return 0;
128  }
```

5.4. Result

6. Add `$U/_sysinfotest\` to `UPROGS` and run the OS via `make qemu`.

```
exec syscalltest failed
$ sysinfotest
sysinfotest: start
sysinfotest: OK
$
```

5.5. Git diff

It is put in `docs/lab-2-syscall/sysinfo.patch`.

6. Load average (challenge) (moderate)

This is optional challenge exercise.

Compute the load average and export it through sysinfo.

In `kernel/defs.h`, define a variable to save data of loadavg

```
extern uint64 current_loadavg;
```

In `kernel/proc.c`, define a function to count the number of processes that are either runnable or running:

```c
// count the number of processes
// whose state is RUNNABLE or RUNNING
int
nrun(void)
{
  struct proc *p;
  int count = 0;

  for(p = proc; p < &proc[NPROC]; p++){
    acquire(&p->lock);
    if(p->state == RUNNABLE || p->state == RUNNING){
      count++;
    }
    release(&p->lock);
  }
  return count;
}
```

Next, define a function to update the system load average using an exponential moving average with a smoothing factor (alpha) of 0.8. The load average is scaled by 1000 for precision.

Note: do not forget to declare `current_loadavg` again in `kernel/proc.c` file, or the huge bug will come.

```c
uint64 current_loadavg = 0;
/**
 * @brief Updates the system load average.
 *
 * This function calculates the system load average using an exponential moving average
 * with a smoothing factor (alpha) of 0.8.
 * The load average is scaled by 1000 for precision.
 */
void
update_loadavg(void)
{
  uint64 n = nrun();
  current_loadavg = (current_loadavg * 8 + n * 1000) / 10;
}
```

Define a function to get the current load average

```
C proc.c M ×
kernel > C proc.c
752  // Return the current load average
753  uint64
754  loadavg(void)
755  {
756    return current_loadavg;
757  }
758
```

Next, define them in `kernel/defs.h` for using inter-module:

```
C defs.h M ×
kernel > C defs.h
195  // proc.c
196  void            update_loadavg(void);
197  uint64          loadavg(void);
198
```

Modify `clockintr()` in `kernel/trap.c` to update load average after several ticks:

```
C trap.c M ×
kernel > C trap.c
162
163  void
164  clockintr()
165  {
166    if(cpuid() == 0){
167      acquire(&tickslock);
168      ticks++;
169      // Update load average every 10 ticks
170      if(ticks % 10 == 0){
171        update_loadavg(); // in kernel/proc.c
172      }
```

Modify `struct sysinfo` in `kernel/sysinfo.h` to save loadavg, too:

```
C sysinfo.h M ×
kernel > C sysinfo.h
1  struct sysinfo {
2    uint64 freemem;   // amount of free memory (bytes)
3    uint64 nproc;     // number of process
4    uint64 loadavg;   // load average
5  };
6
```

Modify function `uint64 sys_sysinfo(void)` in `kernel/sysproc.c`:

```
C sysproc.c M ×
kernel > C sysproc.c
110  {
118    // Call update informations into struct info
119    info.freemem = get_kfreemem();
120    info.nproc = nproc();
121    info.loadavg = loadavg();
```

Done!