OS-HOMEWORK 2
Ikhlas Ahmed Khan
27096

Q1) -Ecall is a csr instruction more related to hardware in RISC V and is called when we want to shift control from user mode to supervisor mode.
An example can be taken from the code of **initcode.S** assembly file, as shown below

xv6-riscv / user / **initcode.S**

kaashoek Thanks snoire@github

Code Blame 28 lines (23 loc) · 413 Bytes

```
1   # Initial process that execs /init.
2   # This code runs in user space.
3
4   #include "syscall.h"
5
6   # exec(init, argv)
7   .globl start
8   start:
9           la a0, init
10          la a1, argv
11          li a7, SYS_exec
12          ecall
13
```

NOTICE how the **ecall** is used to transfer control to supervisor mode (line 12) after setting up the register a7.

-System call is the same concept but its more software related to xv6 OS and is called when we want to shift control from user space to kernel space.
Notice the various types of system calls below in syscall.c which helps to achieve this.

xv6-riscv / kernel / syscall.c

Code Blame 147 lines (137 loc) · 3.39 KB

```c
extern uint64 sys_link(void);
extern uint64 sys_mkdir(void);
extern uint64 sys_close(void);

// 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,
};

```

AN example of the implementation of the pipe system call in sysfile.c

xv6-riscv / kernel / sysfile.c

Code Blame 505 lines (430 loc) · 8.29 KB

```c

uint64
sys_pipe(void)
{
  uint64 fdarray; // user pointer to array of two integers
  struct file *rf, *wf;
  int fd0, fd1;
  struct proc *p = myproc();

  argaddr(0, &fdarray);
  if(pipealloc(&rf, &wf) < 0)
    return -1;
  fd0 = -1;
  if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
    if(fd0 >= 0)
      p->ofile[fd0] = 0;
    fileclose(rf);
    fileclose(wf);
    return -1;
  }
  if(copyout(p->pagetable, fdarray, (char*)&fd0, sizeof(fd0)) < 0 ||
     copyout(p->pagetable, fdarray+sizeof(fd0), (char *)&fd1, sizeof(fd1)) < 0){
    p->ofile[fd0] = 0;
    p->ofile[fd1] = 0;
    fileclose(rf);
    fileclose(wf);
    return -1;
  }
  return 0;
}
```

The reason why i said system calls are software related is because each system calls implementation is done in c and coded (like the one above), while ecalls are hardware related and their implementations and usage is in assembly files (written in risc-V)

HOW THEY WORK TOGETHER TO SHIFT CONTROL FROM USER SPACE TO KERNEL SPACE….?

So take for example a user program in c like initcode and it includes a system call of SYS_ESEC. Now we know that every c file when compiled forms an assembly file like the one in the first pic which is an equivalent representation of the c file. <u>The part where the system call is called in c is the part in assembly where a7 register is loaded with the index of the system call from **syscall.h** header file.</u>

← Files | riscv | xv6-riscv / kernel / syscall.h

Robert Morris separate source into kernel/ user/ mkfs/

22 lines (22 loc) · 485 Bytes

Code | Blame

```c
// 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
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
```

After this, ecall is called which shifts to software based handling os system call into the syscall function in **syscall.c.**

E.G

xv6-riscv / user / **initcode.S**

kaashoek Thanks snoire@github

Code | Blame | 28 lines (23 loc) · 413 Bytes

```
# Initial process that execs /init.
# This code runs in user space.

#include "syscall.h"

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

ecall is called which invokes the function syscall(void) in syscall.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]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

Notice in the above pic num extracts the value from a7 register stored in the trapframe
This index is interpreted here and the relevant system call function called,
E.g for exec num=7
syscalls[num]--> syscall[7]-->**sys_exec(void); called whose function implementation is in sysfile.c**

Q2) the **ls** file is already in the user programs of xv6 and it generally does the same thing as **/bin/ls -al /** in linux where it lists down all the files in the root directory.
Its sample output in xv6 is as follows:

```
$ ls
.              1 1 1024
..             1 1 1024
README         2 2 2292
cat            2 3 35168
echo           2 4 34024
forktest       2 5 16160
grep           2 6 38544
init           2 7 34520
kill           2 8 33984
ln             2 9 33808
ls             2 10 37112
mkdir          2 11 34048
rm             2 12 34024
sh             2 13 56472
stressfs       2 14 34912
usertests      2 15 179408
grind          2 16 49904
wc             2 17 36120
zombie         2 18 33392
test           2 19 33936
Simpleprog     2 20 33312
console        3 21 0
```

Unfortunately for translating all of this into uppercase, xv6 doesn't have a tr on its own implemented.

I have implemented one approach as listed below: (also attached tr file in submission)

tr.c

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
#include "kernel/fs.h"
#include "kernel/fcntl.h"



int main(int argc, char *argv[]) {

        char buffer[512];
        int n;


        while ( (n=read(0,buffer,sizeof(buffer))) >0){
        for (int i=0;i<n;i++){
                if(buffer[i]>='a' &&buffer[i]<='z'){
                buffer[i]=buffer[i]+('A'-'a');
                }
        }

        write(1,buffer,n);
}
        exit(0);

    return 0;
}
```

I have readed the console and stored each line in buffer to read and then translated that line to upper case and then rewritten to the console from the buffer.

And its output is listed below:

```
$ ls | tr
.              1 1 1024
..             1 1 1024
README         2 2 2292
CAT            2 3 35168
ECHO           2 4 34024
FORKTEST       2 5 16160
GREP           2 6 38544
INIT           2 7 34520
KILL           2 8 33984
LN             2 9 33808
LS             2 10 37216
MKDIR          2 11 34048
RM             2 12 34024
SH             2 13 56472
STRESSFS       2 14 34912
USERTESTS      2 15 179408
GRIND          2 16 49904
WC             2 17 36120
ZOMBIE         2 18 33392
TR             2 19 34048
SIMPLEPROG     2 20 33312
CONSOLE        3 21 0
$
```

Why this after all cannot be done ?

- This translation to uppercase specifically cannott be done directly as xv6 doesn't support text processing utilities and has a very limited number of string supported functions.

(in **string.c** file in kernel folder).

- For simple translations our custom tr might work but for different and complex character sets this would also fail. As xv6 does not support as many c libraries for string manipulation and handling.

Q3) To make the system call table inaccessible from the user space we do various things;

- Firstly before we begin you must know that stvec is a csr register that holds the address of trap here.
- Traps can occur from kernel mode so stvec will have address of kernelvec
- If traps occur from usermode, so stvec has the address of uservec which is stored in trampoline.S.
- The a7 register in both cases acts as a very important register in storing the system call index number from syscall.h file.

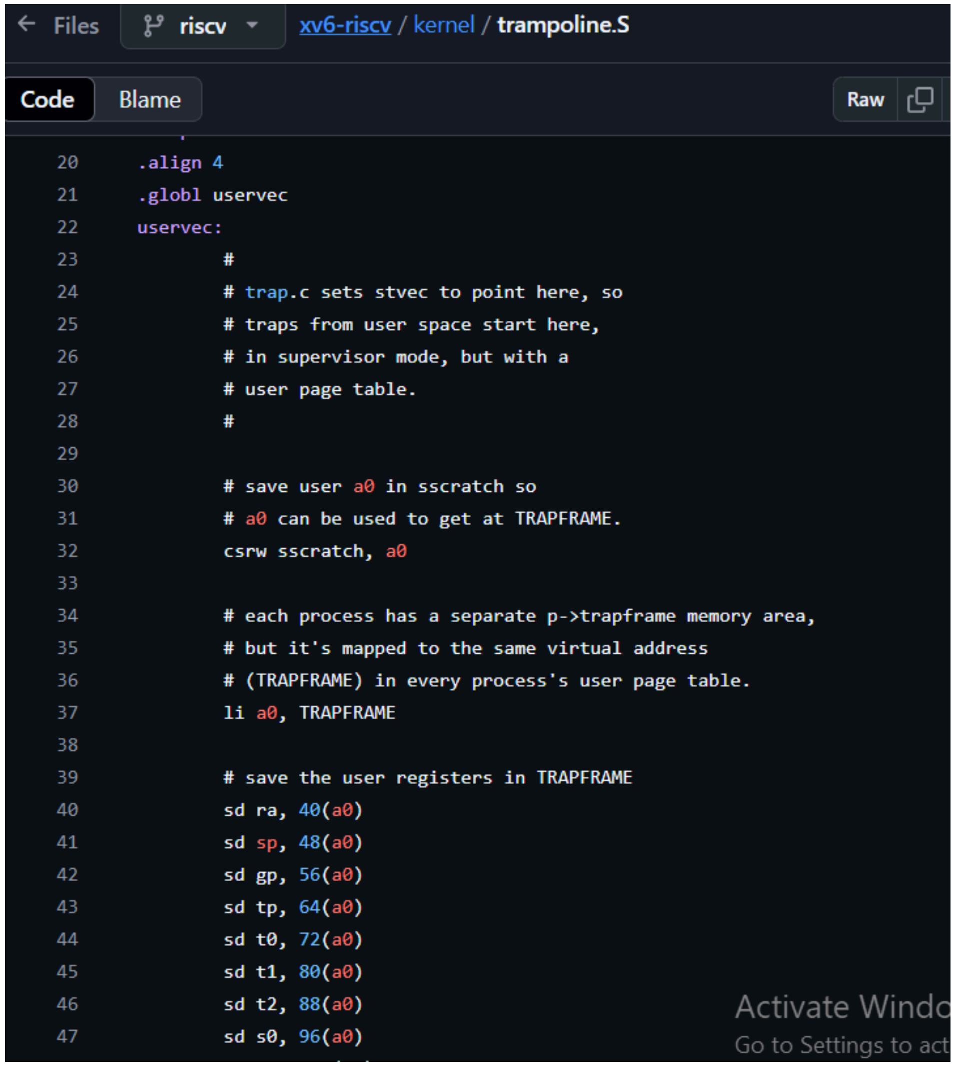
xv6-riscv / kernel / **syscall.h**

Robert Morris separate source into kernel/ user/ n

Code Blame 22 lines (22 loc) · 485 Bytes

```
// 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
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
```

1. First way to make system calls table inaccessible is if we store a random value or an incorrect value in a7
   e.g
   ld a7,-1 (BUT ONLY IN USERVEC not in kernelvec as we still want system calls to function in kernel mode)



   so whenever a system call would occur in user mode register a7 would have -1 stored which infers nothing regarding the system calls table and syscall.c would always print ‘unknown sys call’ hence system calls from user space inaccessible.
2. 2nd way is by returning simply from usertrap function whenever the condition for scause==8 gets true because scause=8 indicates that a trap (e.g system call) is made from user space..

Files riscv xv6-riscv / kernel / trap.c

Code Blame

```c
  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();

  // save user program counter.
  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call

    if(killed(p))
      exit(-1);

    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;

    // an interrupt will change sepc, scause, and sstatus,
    // so enable only now that we're done with those registers.
    intr_on();

    syscall();
  } else if((which_dev = devintr()) != 0){
```

3. In syscall.c in the function below i can make some changes..

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


I know that satp register stores the page table address of current mode so if initially i store the satp address and identify if its from uservec or kernelvec i can determine that from where the trap has initiated and store that in some variable**(e.g initial_satp)** which i would be using here.

Now i just simply put up a condition that if **initial_satp= uservec** then i don't run this function of syscall and simply return from here else if **initial_satp= kernelvec** then i will only run this function to call system call.

So like i add this line at the start of syscall function

syscall(){

if(initial_satp= uservec){

return;

}

—------- (else run the code normally like above if satp=kernelvec)

—---------

}

POSSIBLE FLAW in 3rd one: THE page tables take time to shift due to RISC V hardware constraint which takes some time to switch the page tables in satp, that's the reason trampoline pages are given in both in user space and kernel space to easily control this constraint..
This could potentially cause the satp register in our case to update untimely which may cause disturbance above. **(since satp is a volatile variable so it can change in runtime in my opinion)**

Q4) The different data structures in proc.h are as follows;
-**kstack** is a data-structure which is accessed by threads(e.g kstack 0,kstack 1) in the virtual address space in the kernel. It basically holds the temporary register values so that after the trap is handled in the kernel, those registers(data) can be restored to the same state on which the kernel was working before the trap.

-**pagetable** is another data structure which basically has addresses of the pages in the virtual address space for each of the processes.
-**trapframe** is another data structure which is found in trampoline.S which basically holds the data in temporary register values in the user mode before the OS goes to the kernel mode for trap handling so that after trap is handled these registers are restored. When the system returns to the user mode after handling the trap.
-**context** data structure is used to save registers, stack pointers. Eg when there is context-switching (switching between processes e.g scheduling) .
-**file** is another data structure in a process where each process can have a multiple array of values to interact with that file known as file descriptor table.

-**Kstack** thread pointer can be found in kernel virtual address space
-**Context** can be found in proc.h same file
-**Trapframe** data can be found in trampoline.S assembly file
Note: that this trampoline consists in both user and kernel space to control and resolve the risc v hardware issue of late page switching in satp.

Q5)

- Whenever in sysfile.c the syscall function is called, So before just starting that function we could ask the user for a secret code that only that person knows, who can implement and practice/use system calls. If that code is entered incorrectly the user cannot perform a system call.
- We can also do virtualization of kernel such that when xv6 runs we show it an illusion that it has full control of the kernel while in reality the kernel is isolated

and runs on a separate virtualized environment just like we did with processes when we gaved them virtual memory etc.

- The 3 files **syscall.c, syscall.h, and sysfile.c** are important for system calls and any modification in these could disrupt the system call table so we can encrypt these files inside kernel and the encryption key would only be with the authorized person and no one else.

-In my opinion in a rare case if the hacker implants his custom file (e.g anti-cheat softwares that operate majorly in kernel) inside the kernel with the intention of harming the system call table, So there is nothing we can do to prevent that file from executing. Because whenever xv6 starts up it executes all the user and kernel files automatically whenever we type in the command **make qemu.** So in this case, that custom file would also get executed which could potentially harm and change the system call table.

Q6) There are 3 types of registers in risc-v which are each used for different purposes. The s registers are typically saved registers used in function calls where it stores values across functions that need to be used by both caller(callee) and called function both. (Note: if function uses these first it saves the original data in these registers in a stack and then after doing its thing it restores them)
The t registers are temporary registers and can be used for any purpose throughout the code.(mostly used when you are out of registers or to simplify the register usage in your code).
Among the **a** registers a0 and a1 stores return values/arguments while a2-a7 are only used for function arguments as shown below….

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Registers a2-a7 stores arguments but in major cases there are either no arguments passed into a function or the arguments are shorter such that they wont reserve all these registers but to be on the safe side the last register in this sequence (a7) is used to store system call so that it doesn't disturb the sequence of execution in assembly coding. The t and s registers are used entirely for different purposes as stated above hence they might get used at any point in the code and more frequently, hence they aren't used to store system calls.

Q7)

- **CLINT** is the core-level interrupt more related to the <u>internal</u> software and timer interrupts of the current operating core. They are only found in physical address space as user processes in virtual addresses have no relation to them. As the name suggests, they are core level interrupts so occur only in the main system or main memory which is linked to the physical address space.
- **PLIC** is the platform level interrupt more related to the <u>external</u> interrupt signals from I/O devices e.g ROM,DISK,UART, and other HARTS(other cores). They are found in both virtual address space and physical address space as these interrupts can occur in both main memory (physical address space) or in processes (virtual address space).

Q8) One thing to note here is that whenever u make a system call in a program e.g fork()
And then you compile it…
Its equivalent representation in your assembly file translation is something like this
Li a7,SYS_fork
Ecall
Now according to the system call table due to forking a7 has 1;

xv6-riscv / kernel / **syscall.h**

Robert Morris separate source into kernel/ user/ m

Code Blame 22 lines (22 loc) · 485 Bytes

```c
// 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
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
```

And the ecall statement in assembly leads to syscall(void) function in syscall.c;

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

Now notice here that if fork system call is called, the number in a7=1 which is eventually stored in the variable **int num** in this function.
Now due to hacking if the system call number gets stored as x+50 **this is the part where the system could fail**. As now num=50+1=51.
MORE SPECIFICALLY the line which states: **p->trapframe->a0 = syscalls[num]();**
This tried to call out the system call function form the system call table as shown below:

```
// 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,
};
```

But now since the num is incremented by 50 each time a system call is called and this array(or table) only consist from index 0 to 20;
Hence, this line **p->trapframe->a0 = syscalls[num]();** **FAILS IN SYSCALL.C**
e.g in fork→ syscall[51] will be called out due to hacking which would definitely fail.