System call

Youjip Won

KAIST EE
Passing the parameters in system call

- When a user process calls a system call, it pushes an argument to the user stack.
- Trap handler stores the memory address of the user stack in the esp field of the trapframe (trapframe is stored in the kernel stack).
- Access to the address of the inserted parameter in the user mode.
int main()
{
    char *uargv[3];
    uargv[0] = "echo";
    uargv[1] = "hi";
    uargv[2] = 0;
    printf(1, "Call exec\n");
    exec("echo", uargv);
    exit();
}
System Call

```c
int main()
{
    char *argv[3];

    argv[0] = "echo";
    18: c7 44 24 14 26 07 00 movl $0x726,0x14(%esp)
    1f: 00
    argv[1] = "hi";
    20: c7 44 24 18 2b 07 00 movl $0x72b,0x18(%esp)
    27: 00
    argv[2] = 0;
    28: c7 44 24 1c 00 00 00 movl $0x0,0x1c(%esp)
    2f: 00

    printf(1, "Call exec\n");
    30: e8 8b 03 00 00 call 3c0 <printf>
    35: 8d 44 24 14 lea 0x14(%esp),%eax
    39: 89 44 24 04 mov %eax,0x4(%esp)
    3d: c7 04 24 26 07 00 00 movl $0x726,(%esp)
    44: e8 61 02 00 00 call 2aa <exec>

    exit();
    49: e8 24 02 00 00 call 272 <exit>
    4e: 66 90 xchg %ax,%ax
}
```
user stack

- Compiler determines the unit of stack allocation and the call convention.
  - It is controlled by ‘-mpreferred-stack-boundary’ option in gcc (Default is 16-byte unit).

- The size of the user stack can be checked in [program source code name].asm

- The compiler determines the size of the stack:
  - the space for the local variables
  - the space for function arguments: based upon the function with the largest number of arguments to be called
Example of using the stack in `sysprog.c`.

```c
int main()
{
    char *uargv[3];
    uargv[0] = "echo";
    uargv[1] = "hi";
    uargv[2] = 0;
    printf(1, "Call exec\n");
    exec("echo", uargv);
    exit();
}
```

Local variables

- Three pointer variables.
- 12 bytes (4 bytes * 3) required.
stack allocation: Example 1

• Example of sizing the stack in sysprog.c

```c
int main()
{
    char *uargv[3];
    uargv[0] = "echo";
    uargv[1] = "hi";
    uargv[2] = 0;

    printf(1, "Call exec\n");
    exec("echo", uargv);

    exit();
}
```

Function arguments
• For each function, two parameters, 8 bytes (4 bytes * 2) required.

Local variables + function arguments: 32 Byte (16 Byte aligned)

```
int main()
{
    push %ebp
    mov %esp,%ebp
    and $0xfffffffff0,%esp
    sub $0x20,%esp

    0x20 = 32byte
```
int main()
{
    char *uargv[5];

    uargv[0] = "echo";
    uargv[1] = "hi";
    uargv[2] = "hello";
    uargv[3] = "world";
    uargv[4] = 0;

    printf(1, "Call exec\n");
    exec("echo", uargv);

    exit();
}

Local variables:
• The total of 20 bytes (4 bytes * 5) required.
• 32 bytes allocated (16 byte aligned)
int main()
{
    char *uargv[5];
    uargv[0] = "echo";
    uargv[1] = "hi";
    uargv[2] = "hello";
    uargv[3] = "world";
    uargv[4] = 0;
    printf(1, "Call exec\n");
    exec("echo", uargv);
    exit();
}
int main()
{
    char *uargv[5];

    uargv[0] = "echo";
    uargv[1] = "hi";
    uargv[2] = "hello";
    uargv[3] = "world";
    uargv[4] = 0;

    printf(1, "%s%s%s\n", "param1", "param2", "param3");
    exec("echo", uargv);

    exit();
}

Local Variables:
• Five pointer variables
• 20 byte (4byte*5) required
• 32 byte allocated (16 byte aligned)
int main()
{
    char *uargv[5];

    uargv[0] = "echo";
    uargv[1] = "hi";
    uargv[2] = "hello";
    uargv[3] = "world";
    uargv[4] = 0;

    printf(1, "%s%s%s\n", "param1", "param2", "param3");
    exec("echo", uargv);

    exit();
}

Function parameters
Printf: 5 arguments, 20 byte required 20byte(4byte*5), 32byte allocated
Exec: two arguments, 8 byte required, 16 byte required

Local variables + function parameters: 64byte
Stack Setup (from Example 1)

```c
int main(){
    push %ebp
    mov %esp,%ebp
    and $0xfffffffff0,%esp
    sub $0x20,%esp

    char *uargv[3];
    uargv[0] = "echo";
    movl $0x736,0x14(%esp)
    uargv[1] = "hi";
    movl $0x73b,0x18(%esp)
    uargv[2] = 0;
    movl $0x0,0x1c(%esp)

    printf(1, "Call exec\n");
    movl $0x73e,0x4(%esp)
    printf(1, "Call exec\n");
    call 3c0 <printf>
    exec("echo", uargv);
    lea 0x14(%esp),%eax
    mov %eax,0x4(%esp)
    movl $0x736,(%esp)
    call 2ad <exec>

    exit();
    call 275 <exit>
    xchg %ax,%ax
}
```

Total stack size = 32 byte
new esp = esp + 32
Function call

```c
int main(){
    0:  55       push %ebp
    1:  89 e5     mov %esp,%ebp
    3:  83 e4 f0  and $0xfffffffff0,%esp
    6:  83 ec 20  sub $0x20,%esp
    char *uargv[3];

    uargv[0] = "echo";
    9:  c7 44 24 14 36 07 00 movl $0x736,0x14(%esp)

    uargv[1] = "hi";
    11: c7 44 24 18 3b 07 00 movl $0x73b,0x18(%esp)
    18: 00
    uargv[2] = 0;
    19: c7 44 24 1c 00 00 00 00 movl $0x0,0x1c(%esp)
    20: 00

    printf(1, "Call exec\n");
    21: c7 44 24 04 3e 07 00 movl $0x73e,0x4(%esp)
    28: 00
    29: c7 04 24 01 00 00 00 00 movl $0x1,%(esp)
    30: e8 8b 03 00 00 call 3c0 <printf>
    exec(\"echo\", uargv);
    35: 8d 44 24 14 lea 0x14(%esp),%eax
    39: 89 44 24 04 mov %eax,0x4(%esp)
    3d: c7 04 24 36 07 00 00 movl $0x736,(%esp)
    44: e8 64 02 00 00 call 2ad <exec>

    exit();
    49: e8 27 02 00 00 call 275 <exit>
    4e: 66 90     xchg %ax,%ax
}
```
```
int main()
{
    push ebp
    mov %ebp,esp
    and $0xfffffffff0,esp
    sub $0x20,esp
    char *uargv[3];

    uargv[0] = "echo";
    movl $0x736,0x14(%esp)
    movl $0x73b,0x18(%esp)

    uargv[1] = "hi";
    movl $0x73b,0x18(%esp)
    movl $0x0,0x1c(%esp)

    printf(1,"Call exec\n");

    movl $0x73e,0x4(%esp)
    movl $0x1,(%esp)
    call 3c0 <printf>

    exec("echo", uargv);

    lea 0x14(%esp),%eax
    mov %eax,0x4(%esp)
    mov $0x736,(%esp)
    call 2ad <exec>

    exit();
    call 275 <exit>
}
```
Function call

```c
int
main(){
  0:  55           push %ebp
  1:  89 e5        mov %esp,%ebp
  3:  83 e4 f0     and $0xffffffff0,%esp
  6:  83 ec 20     sub $0x20,%esp
      char *uargv[3];
    uargv[0] = "echo";
  9:  c7 44 24 14 36 07 00 movl $0x736,0x14(%esp)
 10:  00          uargv[1] = "hi";
 11:  c7 44 24 18 3b 07 00 movl $0x73b,0x18(%esp)
 18:  00          uargv[2] = 0;
 19:  c7 44 24 1c 00 00 00 movl $0x0,0x1c(%esp)
 20:  00          printf(1, "Call exec\n");
 21:  c7 44 24 04 3e 07 00 movl $0x73e,0x4(%esp)
 28:  00          exec("echo", uargv);
 29:  c7 04 24 01 00 00 00 movl $0x1,(%esp)
 30:  e8 8b 03 00 00 call 3c0 <printf>
      lea 0x14(%esp),%eax
 35:  8d 44 24 14     mov %eax,0x4(%esp)
 39:  89 44 24 04     movl $0x736,(%esp)
 3d:  c7 04 24 36 07 00 00 movl %eax,0x4(%esp)
 44:  e8 64 02 00 00 call 2ad <exec>
    exit();
 49:  e8 27 02 00 00 call 275 <exit>
 4e:  66 90        xchg %ax,%ax
}
```
Function call

```c
int main(){
    push %ebp
    mov %esp,%ebp
    and $0xfffffffff0,%esp
    sub $0x20,%esp

    char *uargv[3];
    uargv[0] = "echo";
    movl $0x736,0x14(%esp)
    uargv[1] = "hi";
    movl $0x73b,0x18(%esp)
    uargv[2] = 0;
    movl $0x0,0x1c(%esp)

    printf(1, "Call exec\n");
    movl $0x73e,0x4(%esp)

    printf(1, "Call exec\n");
}
```
Function call

```c
int main(){
  0: 55           push %ebp
  1: 89 e5        mov %esp,%ebp
  3: 83 e4 f0     and $0xffffffff0,%esp
  6: 83 ec 20     sub $0x20,%esp
  char *uargv[3];

  uargv[0] = "echo";
  9: c7 44 24 14 36 07 00 movl $0x736,0x14(%esp)
 10: 00
  uargv[1] = "hi";
 11: c7 44 24 18 3b 07 00 movl $0x73b,0x18(%esp)
 18: 00
  uargv[2] = 0;
 19: c7 44 24 1c 00 00 00 movl $0x0,0x1c(%esp)
 20: 00

  printf(1, "Call exec\n");  
21: c7 44 24 04 3e 07 00 movl $0x73e,0x4(%esp)
 28: 00

  29: c7 04 24 01 00 00 00 movl $0x1,0(%esp)
 30: e8 8b 03 00 00 call 3c0 <printf>
  exec("echo", uargv[3]);
 35: 8d 44 24 14 lea 0x14(%esp),%eax
 39: 89 44 24 04 mov %eax,0x4(%esp)
 43: c7 04 24 36 07 00 00 movl $0x736,0x14(%esp)
 44: e8 64 02 00 00 call 2ad <exec>

  exit();
  49: e8 27 02 00 00 call 275 <exit>
  4e: 66 90 xchg %ax,%ax
}
```
Function call

```
int main()
{
    push %ebp
    mov %esp, %ebp
    and $0xfffffffff0, %esp
    sub $0x20, %esp
    char *uargv[3];
    uargv[0] = "echo";
    c7 44 24 14 36 07 00 movl $0x736,0x14(%esp)
    uargv[1] = "hi";
    c7 44 24 18 3b 07 00 movl $0x73b,0x18(%esp)
    uargv[2] = 0;
    c7 44 24 1c 00 00 00 movl $0x0,0x1c(%esp)
    00
    printf(1, "Call exec\n");
    c7 44 24 04 3e 07 00 movl $0x73e,0x4(%esp)
    00
    c7 04 24 01 00 00 00 movl $0x1,0x1(%esp)
    e8 8b 03 00 00 call 3c0 <printf>
    exec("echo", uargv[2]);
    8d 44 24 14 lea 0x14(%esp), %eax
    89 44 24 04 mov %eax,0x4(%esp)
    c7 04 24 36 07 00 00 movl $0x736,0x14(%esp)
    e8 64 02 00 00 call 2ad <exec>
    exit();
    e8 27 02 00 00 call 275 <exit>
    66 90 xchg %ax,%ax
}
```

---

**Call printf() and return**

Stack Grow

```
esp+4 0x73e
esp 0x1
esp+0x14 0x736
esp+0x18 0x73b
esp+0x1c 0
```

Return address (0x35) is pushed and popped here
Youjip Won

int main()
{
    push %ebp
    mov %ebp,%esp
    and $0xffffff0,%esp
    sub $0x20,%esp

    char *uargv[3];
    uargv[0] = "echo";
    c7 44 24 14 36 07 00 movl $0x736,0x14(%esp)
    uargv[1] = "hi";
    c7 44 24 18 3b 07 00 movl $0x73b,0x18(%esp)
    uargv[2] = 0;
    c7 44 24 1c 00 00 00 movl $0x0,0x1c(%esp)

    printf(1, "Call exec\n");
    c7 44 24 04 3e 07 00 movl $0x73e,0x4(%esp)
    c7 04 24 01 00 00 00 movl $0x1,(%esp)
    e8 8b 03 00 00 call 3c0 <printf>

    exec("echo", uargv);
    8d 44 24 14 lea 0x14(%esp),%eax
    89 44 24 04 mov %eax,0x4(%esp)

    exit();
    e8 27 02 00 00 call 275 <exit>
    66 90 xchg %ax,%ax

    lea mem regs
    : load effective address to register

    esp+0x14 
    esp+0x18 
    esp+0x1c 
    esp+0x04 
    esp+0x08 
    esp+0x0c

    0 
    0x72b 
    0x726

    Stack Grow
Function call

```c
int main(){
    0: 55              push %ebp
    1: 89 e5           mov %esp,%ebp
    3: 83 e4 f0        and $0xffffffff0,%esp
    6: 83 ec 20        sub $0x20,%esp
    char *uargv[3];

    uargv[0] = "echo";
    9: c7 44 24 14 36 07 00 movl $0x736,0x14(%esp)
    10: 00              uargv[1] = "hi";
    11: c7 44 24 18 3b 07 00 movl $0x73b,0x18(%esp)
    18: 00              uargv[2] = 0;
    19: c7 44 24 1c 00 00 00 movl $0x0,0x1c(%esp)
    20: 00              printf(1, "Call exec\n");
    21: c7 44 24 04 3e 07 00 movl $0x73e,0x4(%esp)
    28: 00              29: c7 04 24 01 00 00 00 movl $0x1,(%esp)
    30: e8 8b 03 00 00 00 call 3c0 <printf>
    39: 89 44 24 04     mov %eax,0x4(%esp)
    39: c7 04 24 36 07 00 00 movl $0x736,(%esp)
    3d: e8 04 02 00 00 00 call 2ad <exec>

    exit();
    49: e8 27 02 00 00 00 call 275 <exit>
    4e: 66 90          xchg %ax,%ax
}
```

Data Section

```
esp+0x04
 esp+0x14
 esp+0x18
 esp+0x1c
```

```
0x736

"echo"
```

```
uargv[0]
uargv[1]
uargv[2]
```

```
4 Byte
```

```
address of "echo"
```

```
0x726
0x72b
```

```
youjip won
```

```
kaist oslab operating systems laboratory
```

```
youjip won
```
Function call

- Push the return address before jump.

```c
int main(){
    0: 55 push %ebp
    1: 89 e5 mov %esp,%ebp
    3: 83 e4 f0 and $0xffffffff,%esp
    6: 83 ec 20 sub $0x20,%esp
    char *uargv[3];
    uargv[0] = "echo";
    9: c7 44 24 14 36 07 00 movl $0x736,0x14(%esp)
    10: 00
    uargv[1] = "hi";
    11: c7 44 24 18 3b 07 00 movl $0x73b,0x18(%esp)
    18: 00
    uargv[2] = 0;
    19: c7 44 24 1c 00 00 00 movl $0x0,0x1c(%esp)
    20: 00
    printf(1, "Call exec\n");
    21: c7 44 24 04 3e 07 00 movl $0x73e,0x4(%esp)
    28: 00
    29: c7 04 24 01 00 00 00 movl $0x1,0x4(%esp)
    30: e8 8b 03 00 00 call 3c0 <printf>
    exec("echo", uargv);
    35: 8d 44 24 14 lea 0x14(%esp),%eax
    39: 89 44 24 04 mov %eax,0x4(%esp)
    3d: c7 04 24 36 07 00 00 movl $0x736,0x4(%esp)
    44: e8 64 02 00 00 call 2ad <exec>
    exit();
    49: e8 27 02 00 00 call 275 <exit>
    4e: 66 90 xchg %ax,%ax
}
```
System Call

- System call increases the privilege level.
- System call enters the kernel switching the stack from user stack to kernel stack.
- System call copies parameters from the user stack to the kernel address space.
  - Access the arguments in the user stack using `esp` in the trapframe.
  - Should be careful: **check if the address belongs to valid address range.**
Copy parameters

0xFFFFF000

Stack top

Kernel space

User space

Stack top

0x0

ss
esp
eflags
cs
eip
...

... 3rd argument 2nd argument 1st argument return address ...

Copied by argint(), argstr(), argptr()
int argint(int n, int *ip)

- read n-th integer in the user stack and write it to *ip.
- check the dereferenced address if it is within the user address range.

```c
int argint(int n, int *ip)
{
    return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
}
```

check if the address is less than p->sz

```
myproc()->tf->esp + 4 + 4 * 0
myproc()->tf->esp + 4 + 4 * 1
myproc()->tf->esp + 4 + 4 * 2
myproc()->tf->esp
```

Position of the argument

User Stack
int argptr(int n, char *pp, int size)

- Copy the address in the n-th parameter, check if the size is valid and copy it to *pp.
- n: Position of parameter to fetch which contains the address of the object.
- *pp: Start address of the object.
- size: Size of object

```c
int argptr(int n, char *pp, int size)
{
    int i;
    struct proc *curproc = myproc();

    if (argout(n, &i) < 0) // check if n is valid.
        return -1;
    if (size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)
        return -1;
    *pp = (char*)i;
    return 0;
}
```
int argstr(int n, char *pp)

- Copy the start address of string to *pp.
- Returns the length of the string.

- n: Position of the parameter which contains the start address of the string.
- *pp: start address of the string.

```c
int argstr(int n, char **pp)
{
    int addr;
    if(argint(n, &addr) < 0)
        return -1;
    return fetchstr(addr, pp);
}
```
Copy file descriptor

```c
int argfd(int n, int *pfd, struct file **pf)
{
    int fd;
    struct file *f;
    if(argint(n, &fd) < 0)
        return -1;
    if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd] == 0)
        return -1;
    if(pfd)
        *pfd = fd;
    if(pf)
        *pf = f;
    return 0;
}
```
Add a system call: testsys (syscall number 22)

- Add a new system call

  ```c
  int testsys(int i, int* ptr, char* str, int fd)
  ```
  
  - Do nothing but copy parameters from user into kernel and return.

User Program (testsys.c)
```
#include "types.h"
#include "user.h"
#include "fcntl.h"

int main()
{
  int i = 1;
  char *str "hello";
  int fd = open("test", O_CREATE);
  testsys(i, &i, str, fd);
  exit();
}
```

System Call Body
(Append at the end of sysfile.c)
```
int sys_testsys(void)
{
  int i;
  char *ptr;
  char *str;
  int fd;
  argint(0, &i);
  argptr(1, &ptr, 4);
  argstr(2, &str);
  argfd(3, &fd, 0);
  return 0;
}
```
Edit Makefile to compile and make executable testsys

1. Add testsys to variable UPROGS.

```makefile
UPROGS=\ 
  ... \n  _zombie\ 
  _sysprog\ 
  _printpid\ 
  _testsys\ 
  ... 
```

2. Add system call number to syscall.h.

```c
#define SYS_mkdir 20
#define SYS_close 21
#define SYS_tests 22
```

3. Add system call to system call table in syscall.c.

```c
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_tests(void);

static int (*syscalls[])(void) = {
  ...
  [SYS_mkdir]   sys_mkdir,
  [SYS_close]   sys_close,
  [SYS_tests]   sys_tests,
};
```
Edit user library: Add prototype and body

4. Add prototype of new system call to `user.h`.

```c
... char* sbrk(int);
int sleep(int);
int uptime(void);
int testsys(int, void*, char*, int);
```

5. Add new system call to `usys.S`.

```assembly
... SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(testsys)
```
execute `testsys` program.

1. Allocate the user stack.

```c
int main()
{
    push %ebp
    mov %esp,%ebp
    and $0xffffffff0,%esp
    sub $0x20,%esp
}
```
execute testsys program.

2. Initialize local variables: \( i, \text{str} \).

- \( fd \) is initialized with return value of `open()` which is passed over `eax`.

```c
int i = 1;
char *str = "hello";
int fd = open("test", O_CREATE);
```

- `movl $0x1, 0x1c(%esp)` sets `i` to 1.
- `movl $0x200, 0x4(%esp)` initializes `str`.
- `movl $0x736, (%esp)` sets `fd`.
- `call 2b5 <open>` calls the `open` function.
execute testsys program.

2. Initialize local variables: i, str.
   - fd is initialized with return value of open() which is passed over eax.

```
int i = 1;
  \(\text{movl } $0x1,0x1c(%esp)\)
char *str = "hello";
  \(\text{movl } $0x200,0x4(%esp)\)
int fd = open("test", O_CREATE);
  \(\text{movl } $0x736,(%esp)\)
  \(\text{call 2b5 <open>}\)
```

Kernel space
---
User space
---

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1c</td>
<td>c7 44 24 01 00 00</td>
<td>movl $0x1,0x1c(%esp)</td>
</tr>
<tr>
<td>0x18</td>
<td>c7 44 24 04 02 00</td>
<td>movl $0x200,0x4(%esp)</td>
</tr>
<tr>
<td>0x24</td>
<td>c7 04 24 36 07 00 00</td>
<td>movl $0x736,(%esp)</td>
</tr>
<tr>
<td>0x20</td>
<td>e8 90 02 00 00</td>
<td>call 2b5 &lt;open&gt;</td>
</tr>
</tbody>
</table>

0x1 = esp + 28
0x200 = esp + 4
32Byte
2. Initialize local variables: \( i, \text{str} \).

- \( fd \) is initialized with return value of \texttt{open()} \) which is passed over eax.

```c
int i = 1;
char *str = "hello";
int fd = open("test", O_CREATE);
```

![Kernel space vs User space diagram](image)

32Byte

Data section

"test"

esp

esp + 4

esp + 28

Kernel space

User space

execute \texttt{tests} program.
execute testsys program.

2. Initialize local variables: \( i, \text{str} \).
   - \( fd \) is initialized with return value of `open()` which is passed over `eax`.

```c
int i = 1;
char *str = "hello";
int fd = open("test", O_CREATE);
```

The return value of `open`: 3

3 is saved in `eax` after this instruction.
3. Set up arguments of the `testsys()` and call the `testsys()`.

- Copy the address of the string “hello”.

```assembly
extern testsys(i, &i, str, fd);
testsys(i, &i, str, fd);
25: c7 44 24 08 3b 07 00 movl $0x73b,0x8(%esp)
2c: 00
2d: 89 44 24 0c mov %eax,0xc(%esp)
31: 8d 44 24 1c lea 0x1c(%esp),%eax
35: 89 44 24 04 mov %eax,0x4(%esp)
39: 8b 44 24 1c mov 0x1c(%esp),%eax
3d: 89 04 24 mov %eax,(%esp)
40: e8 d0 02 00 00 call 315 <testsys>
exit();
45: e8 2b 02 00 00 call 275 <exit>
```
3. Set up arguments of the `testsyst()` and call the `testsyst()`.
   - Copy the file descriptor.

```
  testsyst(i, &i, str, fd);
  c7 44 24 08 3b 07 00    movl $0x73b,0x8(%esp)
  00
  89 44 24 0c             mov    %eax,0xc(%esp)
  8d 44 24 1c             lea    0x1c(%esp),%eax
  89 44 24 04             mov    %eax,0x4(%esp)
  8b 44 24 1c             mov    0x1c(%esp),%eax
  89 04 24                mov    %eax,(%esp)
  e8 d0 02 00 00          call   315 <testsyst>
  e8 2b 02 00 00          call   275 <exit>
```
Prepare for the system call

3. Set up arguments of the \texttt{testsys()} and call the \texttt{testsys()}.
   • Copy the address of the local variable ‘i’.

\begin{verbatim}
  testsys(i, &i, str, fd);
  25:  c7 44 24 08 3b 07 00  movl  $0x73b,0x8(%esp)
  2c:   00
  2d:   89 44 24 0c         mov    %eax,0xc(%esp)
  31:   8d 44 24 1c         lea    0x1c(%esp),%eax
  35:   89 44 24 04         mov    %eax,0x4(%esp)
  39:   8b 44 24 1c         mov    0x1c(%esp),%eax
  3d:   89 04 24         mov    %eax,(%esp)
  40:   e8 d0 02 00 00     call   315 <testsys>
   exit();
  45:   e8 2b 02 00 00     call   275 <exit>
\end{verbatim}
Before practice: when we execute `testsys` program...

3. Set up arguments of the `testsys()` and call the `testsys()`.
   • Copy the value of the local variable `i`.

```assembly
"hello"
0x73b
0x2fda
esp
esp + 12
esp + 8
esp + 28
0x1
Kernel space
User space
Arguments of `testsys()`
3
0x73b
0x2fda
0x1
```
Before practice: when we execute `testsys` program...

3. Set up arguments of the `testsys()` and call the `testsys()`.
   - Push the return address to the user stack and jump.

```assembly
testsys(i, &i, str, fd);
25:  c7 44 24 08 3b 07 00    movl $0x73b,0x8(%esp)
2c:   00
2d:   89 44 24 0c             mov    %eax,0xc(%esp)
31:   8d 44 24 1c             lea    0x1c(%esp),%eax
35:   89 44 24 04             mov    %eax,0x4(%esp)
39:   8b 44 24 1c             mov    0x1c(%esp),%eax
3d:   89 04 24                mov    %eax,(%esp)
40:   e8 d0 02 00 00          call   315 <testsys>
exit();
45:   e8 2b 02 00 00          call   275 <exit>
```

Return address (0x45) is pushed on stack
Execute a system call.

4. Save the system call number in eax register and generate interrupt.

```assembly
00000315 <testsys>:
SYSCALL(testsys)

315:   b8 16 00 00 00          mov    $0x16,%eax
31a:   cd 40                   int    $0x40
31c:   c3                      ret
```

```c
#define SYSCALL(name) \
   .globl name; \
name: \ 
   movl $SYS_ ## name, %eax; \ 
   int $T_SYSCALL; \ 
   ret
```

- Interrupt number for system call: T_SYSCALL = 64
- SYS_testsys = 22 (0x16)
Memory layout before entering the kernel

Kernel stack →

Kernel space

User space

32Byte

0x1

3

0x73b

0x2fdc

0x1

0x45

esp

eax

0x16
4. Save the system call number in eax register and generate interrupt.

```
00000315 <testsys>:
SYSCALL(testsys)
315:   b8 16 00 00 00          mov    $0x16,%eax
31a:   cd 40                   int    $0x40
31c:   c3                      ret
```

User mode

Kernel mode

```
vector64:
  pushl $0
  pushl $64
  jmp alltraps
```
Immediately after `int` instruction

- `ss`
- `esp`
- `eflags`
- `cs`
- `eip`
- ...

`User space`

- `0x1`
- `3`
- `0x73b`
- `0x2f`
- `0x1`
- `0x45`

`eax`: `0x16`
Before practice: when we execute `testsys` program...

5. Set up the rest of trap frame and call the system call

```c
void trap(struct trapframe *tf)
{
    ...
    syscall();
    ...
}

void syscall(void)
{
    ...
    curproc->tf->eax = syscalls[num]();
    ...
}
```
After trapframe is setup...

![Diagram showing trap frame setup with registers and addresses]
6. Copy the first integer argument from user stack to kernel stack.

```c
int sys_testsys(void) {
    int i, fd;
    char *ptr, *str;

    argint(0, &i);
    argptr(1, &ptr, 4);
    argstr(2, &str);
    argfd(3, &fd, 0);

    return 0;
}
```
7. Copy second pointer argument from user stack to kernel stack

```c
int sys_testsys(void) {
    int i, fd;
    char *ptr, *str;

    argint(0, &i);
    argptr(1, &ptr, 4);
    argstr(2, &str);
    argfd(3, &fd, 0);

    return 0;
}
```
Copy parameters

8. Copy the third string argument from user stack to kernel stack.

```c
int sys_testsys(void) {
    int i, fd;
    char *ptr, *str;
    argint(0, &i);
    argptr(1, &ptr, 4);
    argstr(2, &str);
    argfd(3, &fd, 0);
    return 0;
}
```
Copy parameters

9. Copy fourth file descriptor argument from user stack to kernel stack

```c
int sys_testsys(void) {
    int i, fd;
    char *ptr, *str;
    argint(0, &i);
    argptr(1, &ptr, 4);
    argstr(2, &str);
    argfd(3, &fd, 0);
    return 0;
}
```

- `argint(0, &i);` copies the first argument to the kernel stack.
- `argptr(1, &ptr, 4);` copies the second argument to the kernel stack.
- `argstr(2, &str);` converts the third argument to a kernel address.
- `argfd(3, &fd, 0);` copies the fourth argument to the kernel stack.

The assembly code for the trap frame is:

```
ss
esp...
char *str = 0x73b
char *ptr = 0x2fdc
int fd = 3
int i = 0x1
```

The computation `myproc()->tf->esp + 4 + 4*3` results in copying the fourth file descriptor argument from user space to kernel space.
Toyota's killer firmware: Bad design and its consequences

Michael Dunn - October 28, 2013

On Thursday October 24, 2013, an Oklahoma court ruled against Toyota in a case of unintended acceleration that lead to the death of one the occupants. Central to the trial was the Engine Control Module’s (ECM) firmware.

Embedded software used to be low-level code we’d bang together using C or assembler. These days, even a relatively straightforward, albeit critical, task like throttle control is likely to use a sophisticated RTOS and tens of thousands of lines of code.

With all this sophistication, standards and practices for design, coding, and testing become paramount – especially when the function involved is safety-critical. Failure is not an option. It is something to be contained and benign.

So what happens when an automaker decides to wing it and play by their own rules? To disregard the rigorous standards, best practices, and checks and balances required of such software (and hardware) design? People are killed, reputations ruined, and billions of dollars are paid out. That’s what happens. Here's the story of some software that arguably never should have been.
Summary

- System call is more expensive than function call.
- Address check is a must.
- System call should not change the state of the user process.
- All modifications should be made within the kernel and should not be visible to the user.
Exercise
Compile and Run: Type “make qemu-gdb”.

```
joontaek@ojuntaeg-ui-MacBook-Pro:~/xv6-public$ make qemu-gdb
*** Now run 'gdb'.
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=l
```

Open another shell and Type gdb in xv6 directory

```
joontaek@ojuntaeg-ui-MacBook-Pro:~/xv6-public$ gdb
+ target remote localhost:25501
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
The target architecture is assumed to be i8086
[f000:fff0]  0xffff0: ljmp  $0x3630,$0xf000e05b
0x0000fff0 in ?? ()
+ symbol-file kernel
(gdb)
```

Set breakpoint at the new system call: testsys and type continue.

```
(gdb) br sys_tests
Breakpoint 1 at 0x80104d74: file sysfile.c, line 448.
(gdb) continue
Continuing.
```
Do it yourself.

- Run testsys in xv6 - it will stop at the execution of `sys_testsys` that is breakpoint
  
  ```
  $ ./testsys
  ```

- Dump the user stack - the user stack just before the `sys_testsys` function
  
  ```
  (gdb) x/24x myproc()-tf->esp
  => 0x8dfbef2f: int3
  0x2fac: 0x00000045 0x00000001 0x000002fcc 0x0000073b
  0x2fbc: 0x00000003 0x00000000 0x00000000 0x00000000
  0x2fcc: 0x00000001 0x00000000 0x00000000 0x00000000
  0x2fdc: 0x000003fc8 0xffffffff 0x00000001 0x00002fec
  0x2fec: 0x000002ff4 0x00000000 0x65742f2e 0x79737473
  0x2ffc: 0x00000073 Cannot access memory at address 0x3000
  ```

Arguments of `sys_testsys` are already pushed to user stack

```
int sys_testsys(int i, char *ptr, char *str, int fd);
```
Run testsys and examine user stack

- Dump the kernel stack - there is space for local variables: i, ptr, str, and fd
  - There is garbage value

```c
int sys_testsys(void)
{
    int i;
    char *ptr;
    char *str;
    int fd;

    argint(0, &i);
    argptr(1, &ptr, 4);
    argstr(2, &str);
    argfd(3, &fd, 0);

    return 0;
}
```

(gdb) x/24x $esp
0x8dee1f10: 0x00000000 0x00000000 0x8dee1f28 0x8dee1fb4
0x8dee1f20: 0x00000000 0x00000000 0x8dee1f38 0x80103a8c
0x8dee1f30: int fd 0x00 0x00 0x00 0x00 0x00
0x8dee1f40: 0x00 char *str 0x00 char *ptr 0x00
0x8dee1f50: 0x8dee1fb4 0x00 0x8dee1fa8 0x801057e9
0x8dee1f60: 0x80114c80 0x00 0x00 0x8dee1f8c

Youjip Won
Run testsys and examine user stack

- The dump of the kernel stack before calling argint() - there is no argument of

```c
int sys_testsys(void){
    int i;
    char *ptr;
    char *str;
    int fd;

    argint(0, &i);
    argptr(1, &ptr, 4);
    argstr(2, &str);
    argfd(3, &fd, 0);

    return 0;
}
```

(gdb) x/24x $esp
0x8dee1f10: 0x000002fb0 0x8dee1f2c 0x8dee1f28 0x8dee1fb4
0x8dee1f20: 0x00000000 0x00000000 0x8dee1f38 0x000000001
0x8dee1f30: 0x00000000 0x8dee1fb4 0x8dee1f58 0x8010478e
0x8dee1f40: 0x00000000 0x00000000 0x8dee1f58 0x80103a8c
0x8dee1f50: 0x8dee1fb4 0x00000000 0x8dee1fa8 0x801057e9
0x8dee1f60: 0x80114c80 0x00000000 0x00000000 0x8dee1f8c
Run testsys and examine user stack

- The dump of the kernel stack before calling argint() - there is no argument of

```c
int sys_testsys(void){
    int i;
    char *ptr;
    char *str;
    int fd;

    argint(0, &i);
    argptr(1, &ptr, 4);
    argcstr(2, &str);
    argfd(3, &fd, 0);

    return 0;
}
```

Because first arg was i and second arg was &i, there is value of first arg in second arg's address

```
(gdb) x/x 0x2fcc
0x2fcc: 0x00000001
(gdb)
```

```
0x8dee1f10: 0x00000001 0x8dee1f28 0x00000004 0x8dee1fb4
0x8dee1f20: 0x00000000 0x00000000 0x000002fcc 0x00000001
0x8dee1f30: 0x00000000 0x8dee1fb4 0x8dee1f58 0x8010478e
0x8dee1f40: 0x00000000 0x00000000 0x8dee1f58 0x80103a8c
0x8dee1f50: 0x8dee1fb4 0x00000000 0x8dee1fa8 0x801057e9
0x8dee1f60: 0x80114c80 0x00000000 0x00000000 0x8dee1f8c
```
Run testsys and examine user stack

- The dump of the kernel stack before calling argint() - there is no argument of

```c
int sys_testsys(void)
{
    int i;
    char *ptr;
    char *str;
    int fd;

    argint(0, &i);
    argptr(1, &ptr, 4);
    argstr(2, &str);
    argfd(3, &fd, 0);

    return 0;
}
```

There is string “hello” in third arg’s address

```
(gdb) x/s 0x73b
0x73b: "hello"
```

※“x/s (address) is command that prints string that is saved at (address)

```
(gdb) x/24x $esp
0x8dee1f10: 0x00000002 0x8dee1f24 0x00000004 0x8dee1fb4
0x8dee1f20: 0x00000000 0x00000073b 0x00000000 0x8dee1fb4
0x8dee1f30: 0x00000000 0x8dee1fb4 0x8dee1f58 0x8010478e
0x8dee1f40: 0x00000000 0x00000000 0x8dee1f58 0x80103a8c
0x8dee1f50: 0x8dee1fb4 0x00000000 0x8dee1fa8 0x801057e9
0x8dee1f60: 0x80114c80 0x00000000 0x00000000 0x8dee1f8c
```
Run testsys and examine user stack

- The dump of the kernel stack before calling argint() - there is no argument of

```c
int sys_testsys(void)
{
    int i;
    char *ptr;
    char *str;
    int fd;

    argint(0, &i);
    argvptr(1, &ptr, 4);
    argstr(2, &str);
    argvfd(3, &fd, 0);

    return 0;
}
```

(gdb) x/24x $esp
0x8dee1f10: 0x00000002 0x8dee1f24 0x00000004 0x8dee1fb4
0x8dee1f20: 0x00000003 0x0000073b 0x00002fcc 0x00000001
0x8dee1f30: 0x00000000 0x8dee1fb4 0x8dee1f58 0x8010478e
0x8dee1f40: 0x00000000 0x00000000 0x8dee1f58 0x80103a8c
0x8dee1f50: 0x8dee1fb4 0x00000000 0x8dee1fa8 0x801057e9
0x8dee1f60: 0x80114c80 0x00000000 0x00000000 0x8dee1f8c