System Call

Youjip Won

KAIST

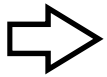
Passing the parameters in system call

- 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.

System Call

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

  uargv[0] = “echo”;
  uargv[1] = “hi”;
  uargv[2] = 0;

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

  exit();
}
```

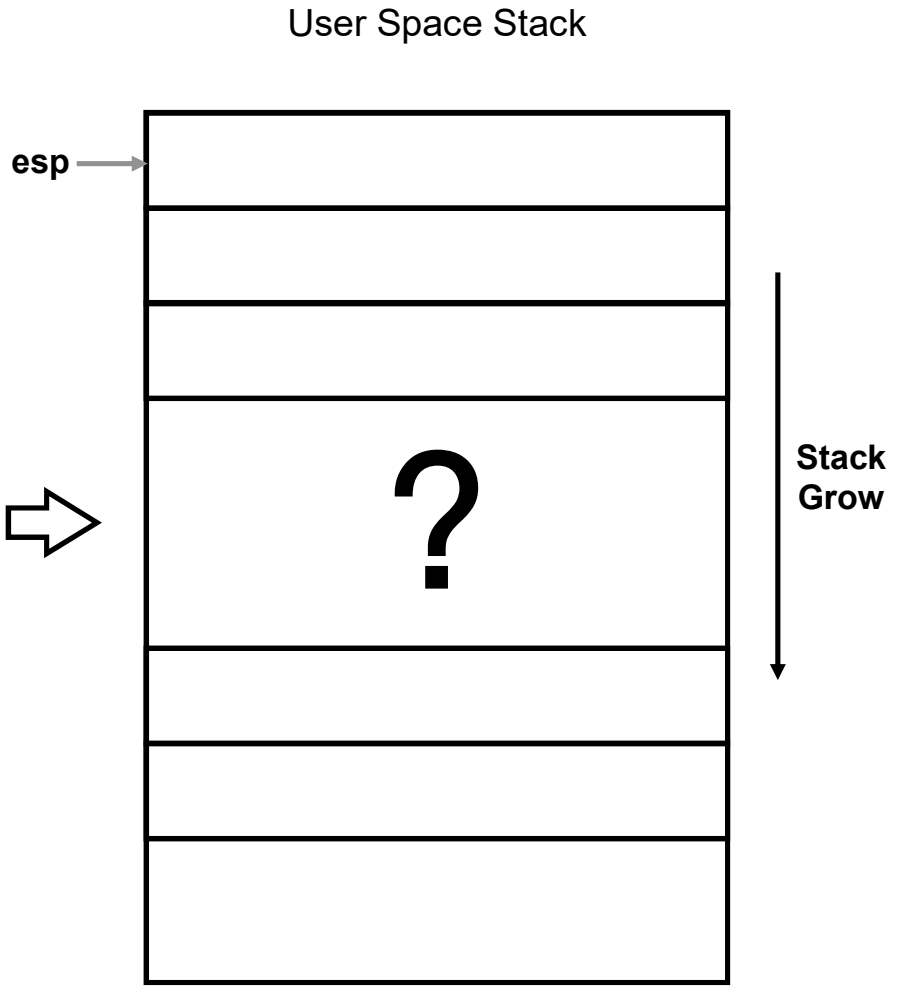
By Compiler



```
/* At the beginning of the main(); */
push   %ebp
mov    %esp,%ebp
and    $0xfffffff0,%esp
sub    $0x20,%esp

/* uargv[0] = "echo"; */
movl   $0x6be,0x14(%esp)
/* uargv[1] = "hi";   */
movl   $0x6c3,0x18(%esp)
/* uargv[2] = 0;      */
movl   $0x0,0x1c(%esp)

/* printf(1, "Call exec\n"); */
call   3e0 <printf>

/* exec("echo", uargv);      */
lea    0x14(%esp),%eax
mov    %eax,0x4(%esp)
movl   $0x6be,(%esp)
call   2ba <exec>

/* exit(); */
call   282 <exit>
xchg   %ax,%ax
```

System Call

```
/* At the beginning of the main(); */
push   %ebp
mov    %esp,%ebp
and    $0xfffffff0,%esp
sub    $0x20,%esp

/* uargv[0] = "echo"; */
movl   $0x6be,0x14(%esp)
/* uargv[1] = "hi";   */
movl   $0x6c3,0x18(%esp)
/* uargv[2] = 0;      */
movl   $0x0,0x1c(%esp)

/* printf(1, "Call exec\n"); */
call   3e0 <printf>

/* exec("echo", uargv);      */
lea    0x14(%esp),%eax
mov    %eax,0x4(%esp)
movl   $0x6be,(%esp)
call   2ba <exec>

/* exit(); */
call   282 <exit>
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

stack allocation: Example 1

- Example of using the stack in `sysprog.c`.

```
/* char *uargv[3];     */
/* uargv[0] = "echo"; */
movl    $0x6be,0x14(%esp)
/* uargv[1] = "hi";    */
movl    $0x6c3,0x18(%esp)
/* uargv[2] = 0;       */
movl    $0x0,0x1c(%esp)

/* printf(1, "Call exec\n"); */
call    3e0 <printf>

/* exec("echo", uargv);      */
lea     0x14(%esp),%eax
mov     %eax,0x4(%esp)
movl    $0x6be,(%esp)
call    2ba <exec>
```

Local variables

- Three pointer variables (**`char * uargv[3]`**).
- 12 bytes (4 bytes * 3) required.
- 16 bytes are allocated (due to the alignment)

stack allocation: Example 1

- Example of using the stack in `sysprog.c`.

```
/* char *uargv[3];     */
/* uargv[0] = "echo"; */
movl    $0x6be,0x14(%esp)
/* uargv[1] = "hi";    */
movl    $0x6c3,0x18(%esp)
/* uargv[2] = 0;       */
movl    $0x0,0x1c(%esp)

/* printf(1, "Call exec\n"); */
call    3e0 <printf>

/* exec("echo", uargv);      */
lea     0x14(%esp),%eax
mov     %eax,0x4(%esp)
movl    $0x6be,(%esp)
call    2ba <exec>
```

Note that String literal is stored

at the data section of ELF Binary

stack allocation: Example 1

- Example of sizing the stack in `sysprog.c`

```
/* printf(1, "Call exec\n"); */
movl    $0x6c6,0x4(%esp)
movl    $0x1,(%esp)
call    3e0 <printf>

/* exec("echo", uargv);      */
lea     0x14(%esp),%eax
mov     %eax,0x4(%esp)
movl    $0x6be,(%esp)
call    2ba <exec>
```

Function arguments

- `printf()` → two arguments (8 bytes required)
- `exec()` → two arguments (8 bytes required)

→ The stack is allocated according to the function which require largest stack.

→ Stack Size for Function Argument = Max (# of arguments of `printf()`, # of arguments of `exec()`)

→16 bytes are allocated (due to the alignment)

stack allocation: Example 1

- Example of sizing the stack in `sysprog.c`

```
/* printf(1, "Call exec\n"); */
movl    $0x6c6,0x4(%esp)
movl    $0x1,(%esp)
call    3e0 <printf>

/* exec("echo", uargv);      */
lea     0x14(%esp),%eax
mov     %eax,0x4(%esp)
movl    $0x6be,(%esp)
call    2ba <exec>
```

Function arguments

- `printf()` → two arguments (8 bytes required)
- `exec()` → two arguments (8 bytes required)

→ The stack is allocated according to the function which require largest stack.

→ Note that each function call reuse the stack for argument passing

stack allocation: Example 1

- Example of sizing the stack in `sysprog.c`

```
/* At the beginning of the main(); */
push   %ebp
mov    %esp,%ebp
and    $0xfffffff0,%esp
sub    $0x20,%esp
```

0x20 = 32byte

Local variables (16) + function parameters (16): 32 Bytes

Stack Allocation: Example 2

```
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)

Function arguments

- `printf()` → two arguments (8 bytes required)
- `exec()` → two arguments (8 bytes required)
- 16 bytes allocated (16 byte aligned)

Local variables (32) + function parameters(16): 48 Bytes

Stack Allocation: Example 3

```
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",
        "uargv[1]",
        "uargv[2]",
        "uargv[3]");
  exec("echo", uargv);

  exit();
}
```

Local variables:

- The total of 20 bytes (4 bytes * 5) required.
- 32 bytes allocated (16 byte aligned)

Function arguments

- `printf()` → five arguments (20 bytes required)
- `exec()` → two arguments (8 bytes required)
- 32 bytes allocated (16 byte aligned)

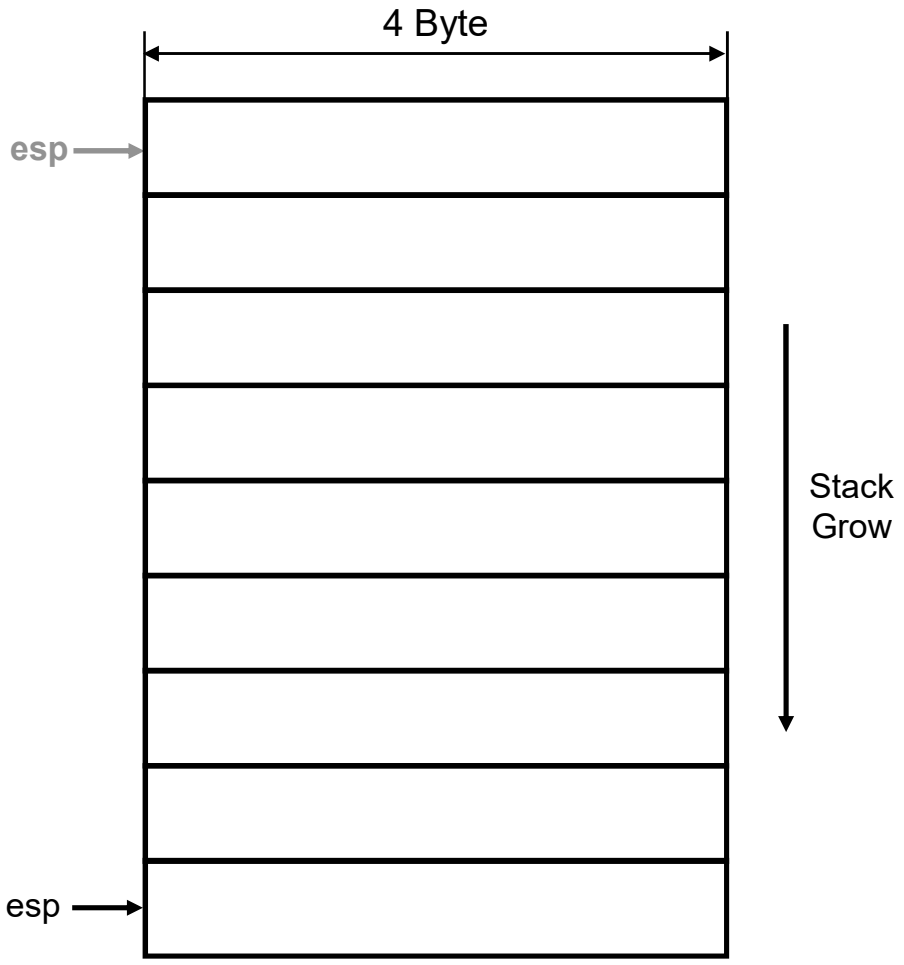
Local variables(32) + function parameters(32) : 64 Bytes

Stack Setup (from Example 1)

```
int main() {
  /* Stack Allocation   */
  push   %ebp
  mov    %esp,%ebp
  and    $0xfffffff0,%esp
  sub    $0x20,%esp

  /* char *uargv[3];    */
  /* uargv[0] = "echo"; */
  movl   $0x6be,0x14(%esp)
  /* uargv[1] = "hi";   */
  movl   $0x6c3,0x18(%esp)
  /* uargv[2] = 0;      */
  movl   $0x0,0x1c(%esp)

  /* printf(1, "Call exec\n"); */
  movl   $0x6c6,0x4(%esp)
  movl   $0x1,(%esp)
  call   3e0 <printf>

  /* exec("echo", uargv);      */
  lea    0x14(%esp),%eax
  mov    %eax,0x4(%esp)
  movl   $0x6be,(%esp)
  call   2ba <exec>

  /* exit(); */
  call   282 <exit>
  xchg   %ax,%ax
}
```

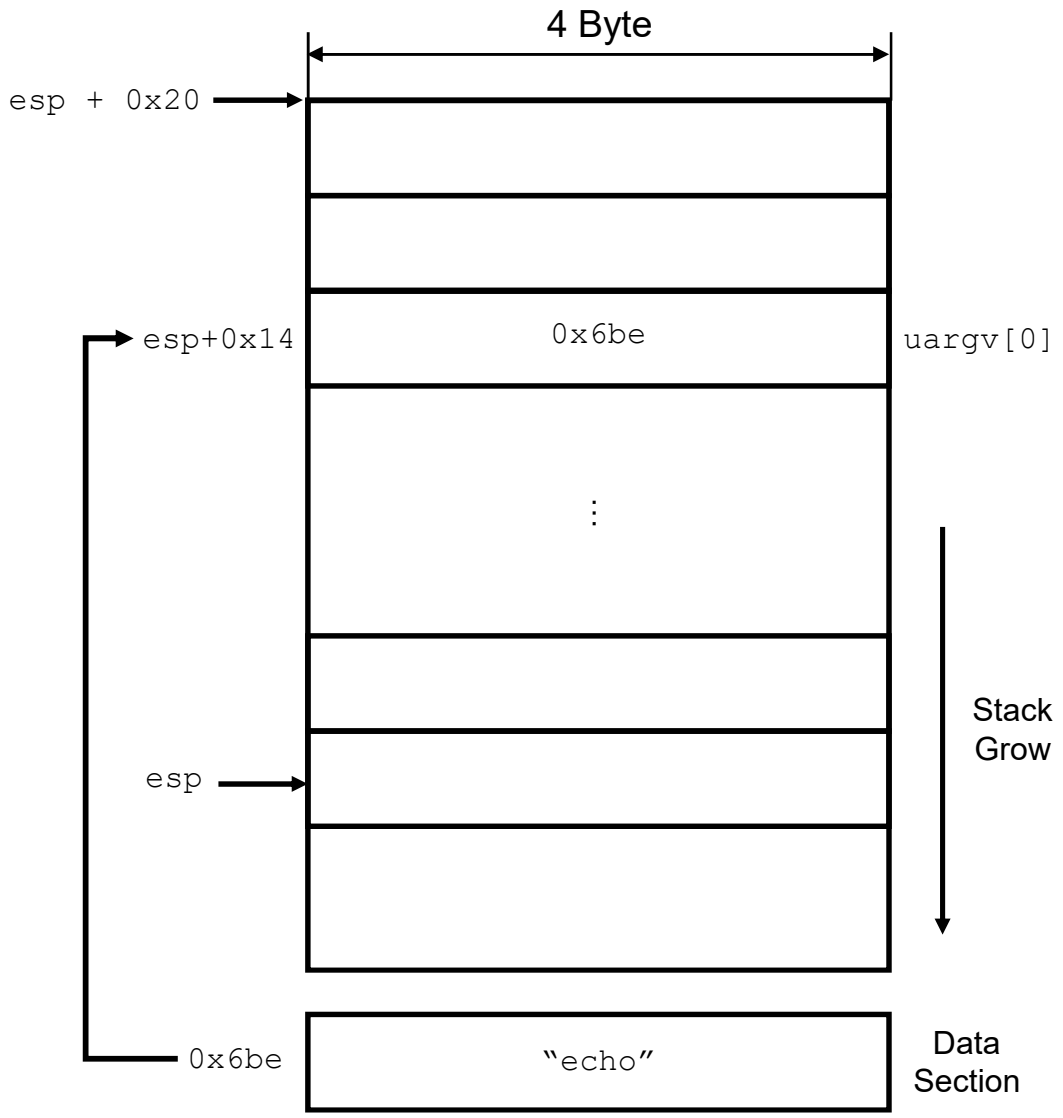
Total stack size = 32 byte

new **esp = esp + 32**



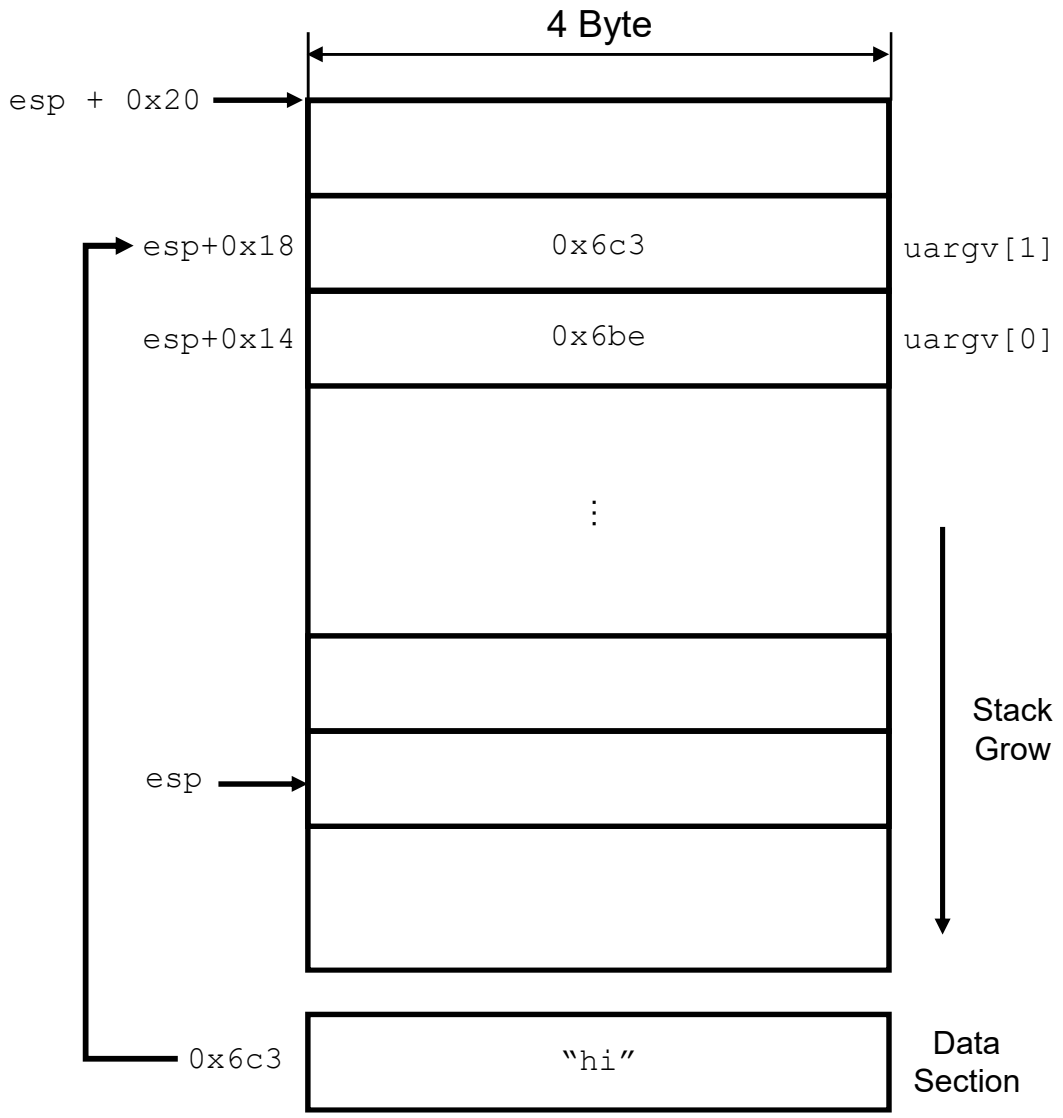
Function call

```
int main() {
  /* Stack Allocation   */
  push   %ebp
  mov    %esp,%ebp
  and    $0xfffffff0,%esp
  sub    $0x20,%esp

  /* char *uargv[3];    */
  /* uargv[0] = "echo"; */
  movl   $0x6be,0x14(%esp)
  /* uargv[1] = "hi";   */
  movl   $0x6c3,0x18(%esp)
  /* uargv[2] = 0;      */
  movl   $0x0,0x1c(%esp)

  /* printf(1, "Call exec\n"); */
  movl   $0x6c6,0x4(%esp)
  movl   $0x1,(%esp)
  call   3e0 <printf>

  /* exec("echo", uargv);      */
  lea    0x14(%esp),%eax
  mov    %eax,0x4(%esp)
  movl   $0x6be,(%esp)
  call   2ba <exec>

  /* exit(); */
  call   282 <exit>
  xchg   %ax,%ax
}
```



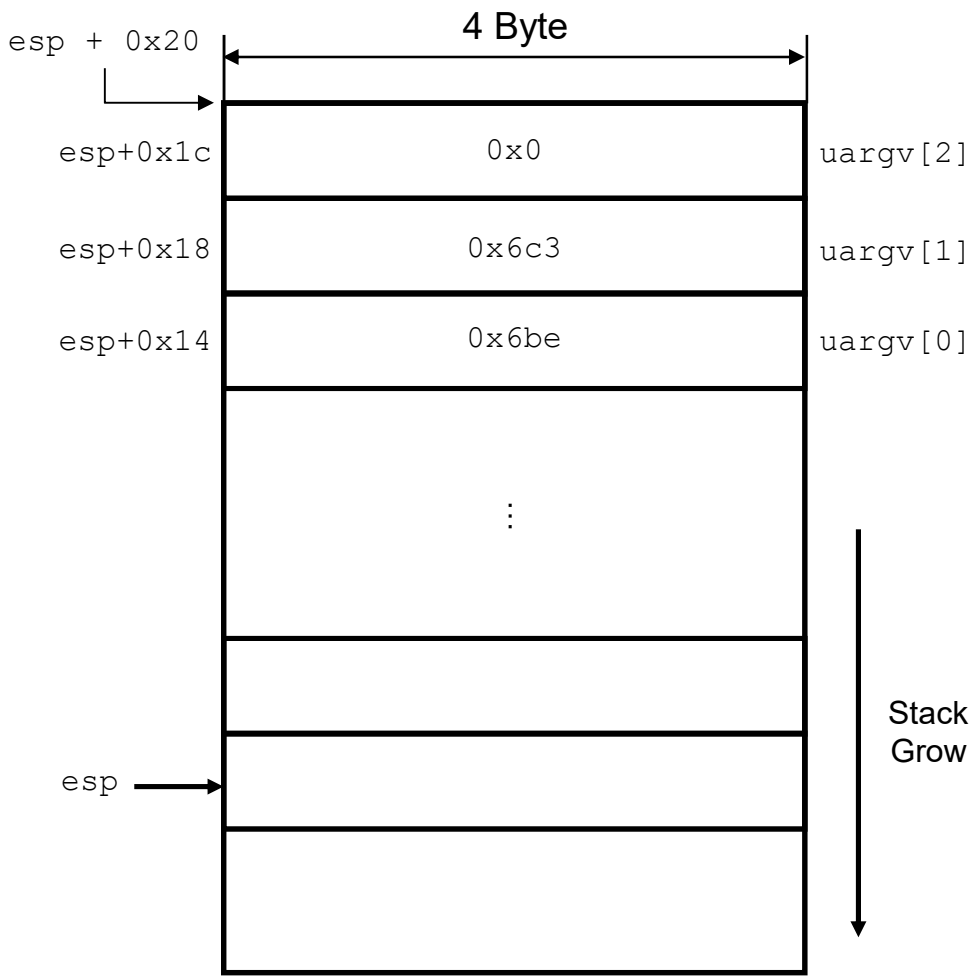
Function call

```
int main() {
  /* Stack Allocation   */
  push   %ebp
  mov    %esp,%ebp
  and    $0xfffffff0,%esp
  sub    $0x20,%esp

  /* char *uargv[3];    */
  /* uargv[0] = "echo"; */
  movl   $0x6be,0x14(%esp)
  /* uargv[1] = "hi";   */
  movl   $0x6c3,0x18(%esp)
  /* uargv[2] = 0;      */
  movl   $0x0,0x1c(%esp)

  /* printf(1, "Call exec\n"); */
  movl   $0x6c6,0x4(%esp)
  movl   $0x1,(%esp)
  call   3e0 <printf>

  /* exec("echo", uargv);      */
  lea    0x14(%esp),%eax
  mov    %eax,0x4(%esp)
  movl   $0x6be,(%esp)
  call   2ba <exec>

  /* exit(); */
  call   282 <exit>
  xchg   %ax,%ax
}
```



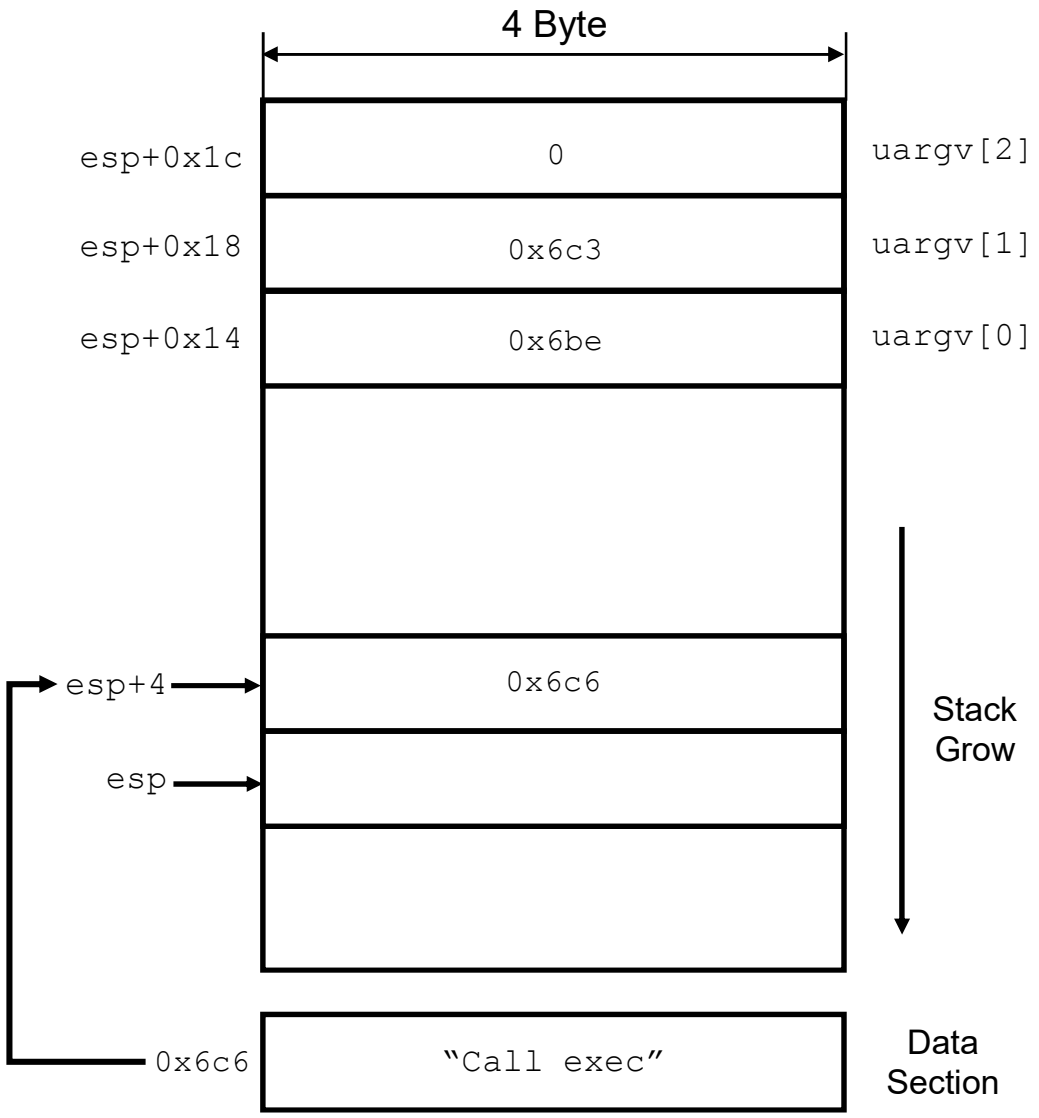
Function call

```
int main() {
  /* Stack Allocation   */
  push   %ebp
  mov    %esp,%ebp
  and    $0xfffffff0,%esp
  sub    $0x20,%esp

  /* char *uargv[3];    */
  /* uargv[0] = "echo"; */
  movl   $0x6be,0x14(%esp)
  /* uargv[1] = "hi";   */
  movl   $0x6c3,0x18(%esp)
  /* uargv[2] = 0;      */
  movl   $0x0,0x1c(%esp)

  /* printf(1, "Call exec\n"); */
  movl   $0x6c6,0x4(%esp)
  movl   $0x1,(%esp)
  call   3e0 <printf>

  /* exec("echo", uargv);      */
  lea    0x14(%esp),%eax
  mov    %eax,0x4(%esp)
  movl   $0x6be,(%esp)
  call   2ba <exec>

  /* exit(); */
  call   282 <exit>
  xchg   %ax,%ax
}
```



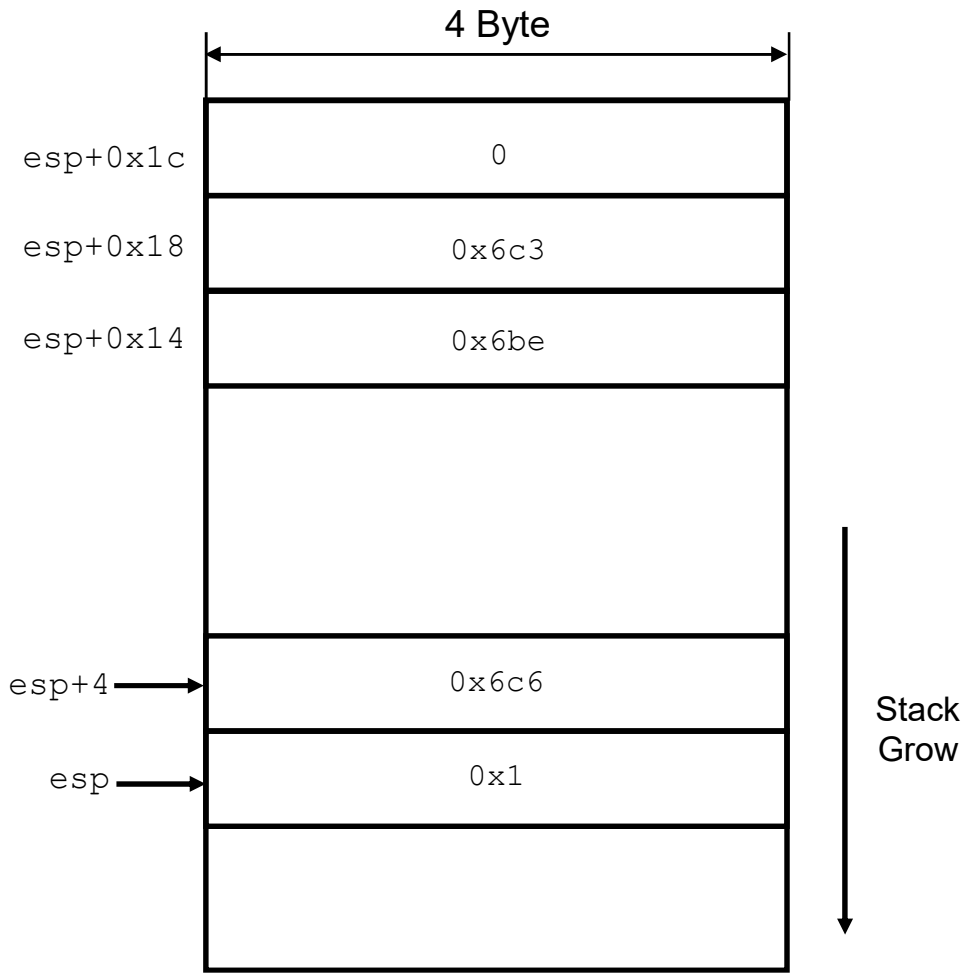
Function call

```
int main() {
  /* Stack Allocation   */
  push   %ebp
  mov    %esp,%ebp
  and    $0xfffffff0,%esp
  sub    $0x20,%esp

  /* char *uargv[3];    */
  /* uargv[0] = "echo"; */
  movl   $0x6be,0x14(%esp)
  /* uargv[1] = "hi";   */
  movl   $0x6c3,0x18(%esp)
  /* uargv[2] = 0;      */
  movl   $0x0,0x1c(%esp)

  /* printf(1, "Call exec\n"); */
  movl   $0x6c6,0x4(%esp)
  movl   $0x1,(%esp)
  call   3e0 <printf>

  /* exec("echo", uargv);      */
  lea    0x14(%esp),%eax
  mov    %eax,0x4(%esp)
  movl   $0x6be,(%esp)
  call   2ba <exec>

  /* exit(); */
  call   282 <exit>
  xchg   %ax,%ax
}
```



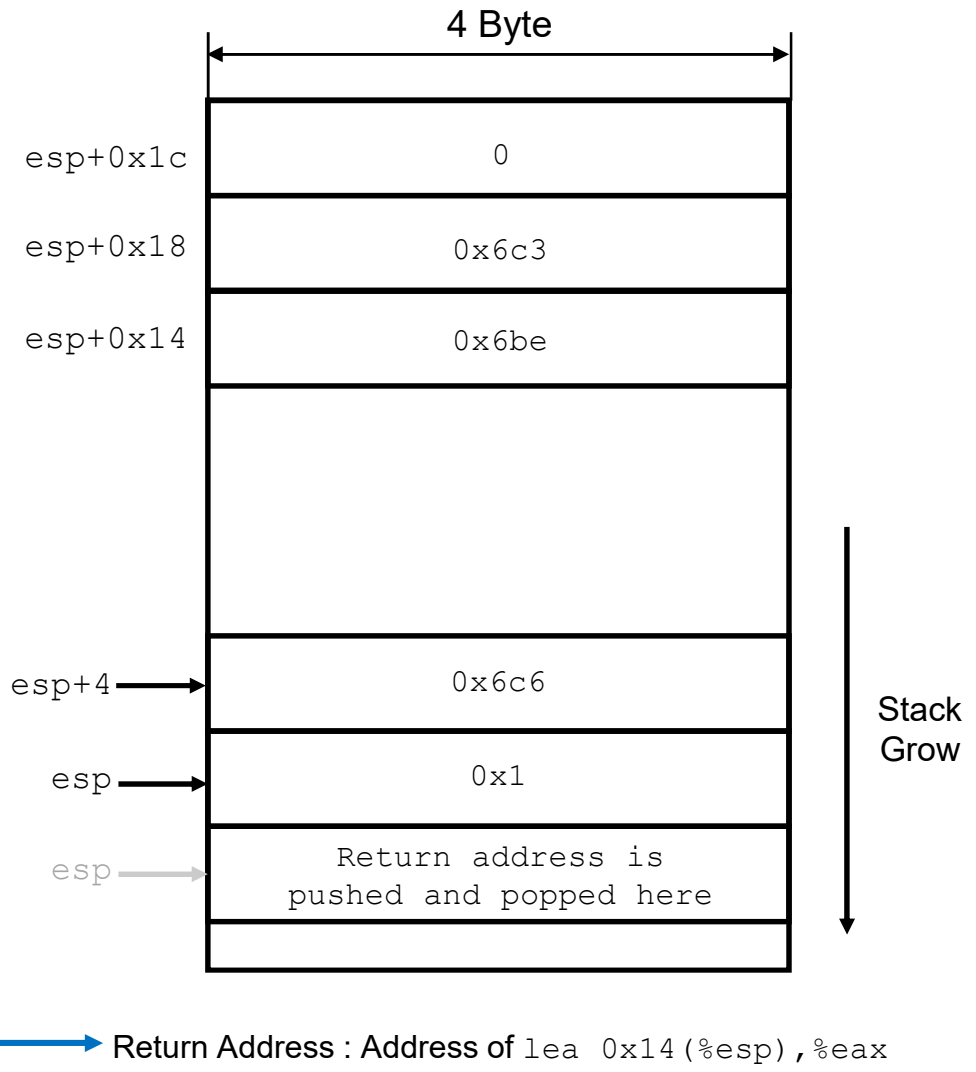
Function call

```
int main() {
  /* Stack Allocation   */
  push   %ebp
  mov    %esp,%ebp
  and    $0xfffffff0,%esp
  sub    $0x20,%esp

  /* char *uargv[3];    */
  /* uargv[0] = "echo"; */
  movl   $0x6be,0x14(%esp)
  /* uargv[1] = "hi";   */
  movl   $0x6c3,0x18(%esp)
  /* uargv[2] = 0;      */
  movl   $0x0,0x1c(%esp)

  /* printf(1, "Call exec\n"); */
  movl   $0x6c6,0x4(%esp)
  movl   $0x1,(%esp)
  call   3e0 <printf>

  /* exec("echo", uargv);      */
  lea    0x14(%esp),%eax
  mov    %eax,0x4(%esp)
  movl   $0x6be,(%esp)
  call   2ba <exec>

  /* exit(); */
  call   282 <exit>
  xchg   %ax,%ax
}
```



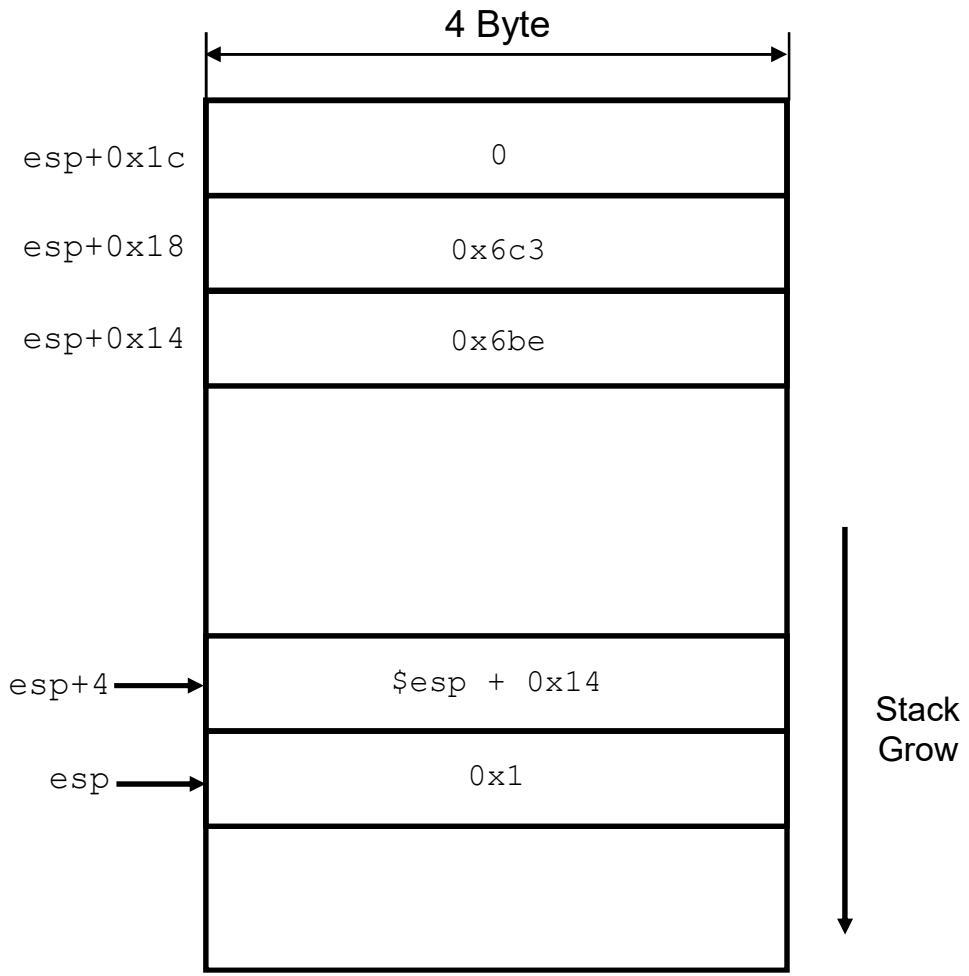
Function call

```
int main() {
  /* Stack Allocation   */
  push   %ebp
  mov    %esp,%ebp
  and    $0xfffffff0,%esp
  sub    $0x20,%esp

  /* char *uargv[3];    */
  /* uargv[0] = "echo"; */
  movl   $0x6be,0x14(%esp)
  /* uargv[1] = "hi";   */
  movl   $0x6c3,0x18(%esp)
  /* uargv[2] = 0;      */
  movl   $0x0,0x1c(%esp)

  /* printf(1, "Call exec\n"); */
  movl   $0x6c6,0x4(%esp)
  movl   $0x1,(%esp)
  call   3e0 <printf>

  /* exec("echo", uargv);      */
  lea    0x14(%esp),%eax
  mov    %eax,0x4(%esp)
  movl   $0x6be,(%esp)
  call   2ba <exec>

  /* exit(); */
  call   282 <exit>
  xchg   %ax,%ax
}
```



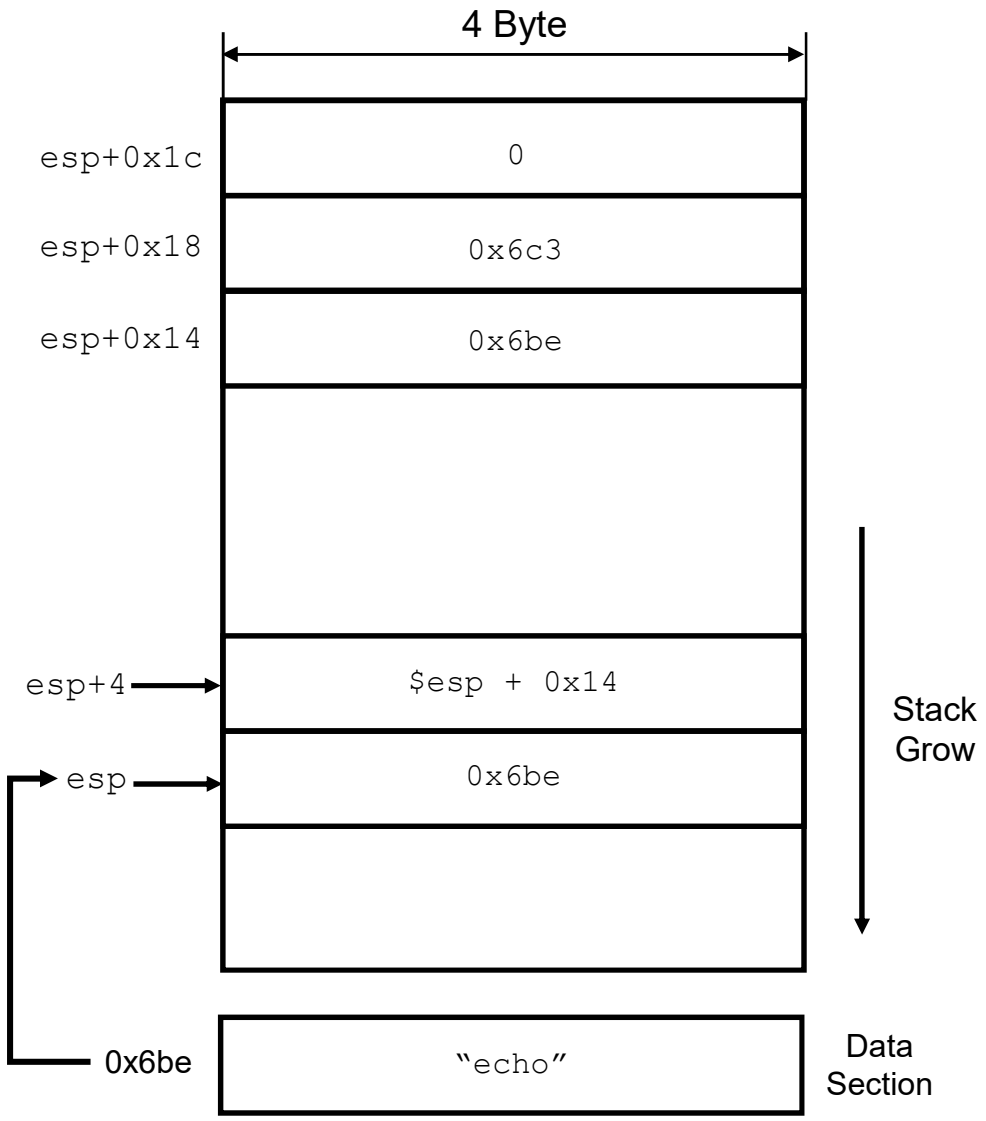
Function call

```
int main() {
  /* Stack Allocation   */
  push   %ebp
  mov    %esp,%ebp
  and    $0xfffffff0,%esp
  sub    $0x20,%esp

  /* char *uargv[3];    */
  /* uargv[0] = "echo"; */
  movl   $0x6be,0x14(%esp)
  /* uargv[1] = "hi";   */
  movl   $0x6c3,0x18(%esp)
  /* uargv[2] = 0;      */
  movl   $0x0,0x1c(%esp)

  /* printf(1, "Call exec\n"); */
  movl   $0x6c6,0x4(%esp)
  movl   $0x1,(%esp)
  call   3e0 <printf>

  /* exec("echo", uargv);      */
  lea    0x14(%esp),%eax
  mov    %eax,0x4(%esp)
  movl   $0x6be,(%esp)
  call   2ba <exec>

  /* exit(); */
  call   282 <exit>
  xchg   %ax,%ax
}
```



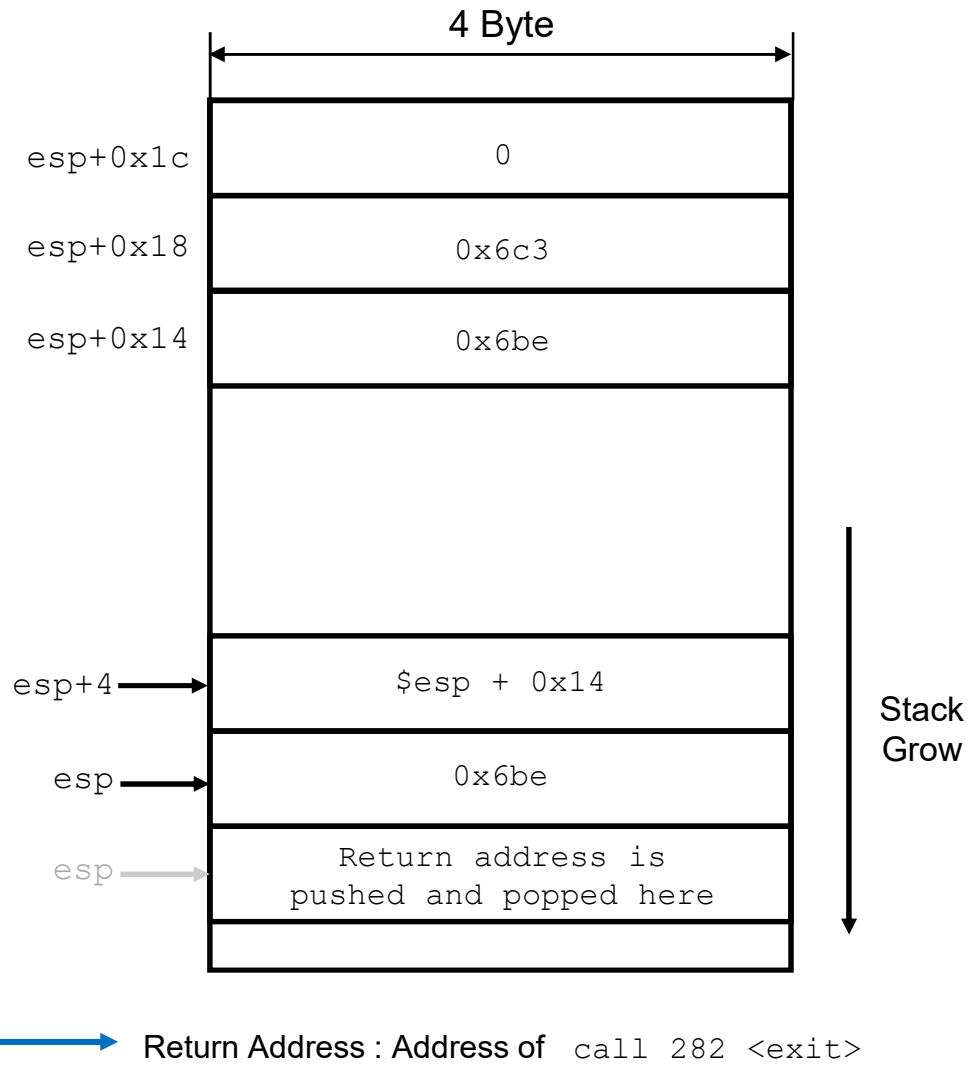
Function call

```
int main() {
  /* Stack Allocation   */
  push   %ebp
  mov    %esp,%ebp
  and    $0xfffffff0,%esp
  sub    $0x20,%esp

  /* char *uargv[3];    */
  /* uargv[0] = "echo"; */
  movl   $0x6be,0x14(%esp)
  /* uargv[1] = "hi";   */
  movl   $0x6c3,0x18(%esp)
  /* uargv[2] = 0;      */
  movl   $0x0,0x1c(%esp)

  /* printf(1, "Call exec\n"); */
  movl   $0x6c6,0x4(%esp)
  movl   $0x1,(%esp)
  call   3e0 <printf>

  /* exec("echo", uargv);      */
  lea    0x14(%esp),%eax
  mov    %eax,0x4(%esp)
  movl   $0x6be,(%esp)
  call   2ba <exec>

  /* exit(); */
  call   282 <exit>
  xchg   %ax,%ax
}
```



Function call

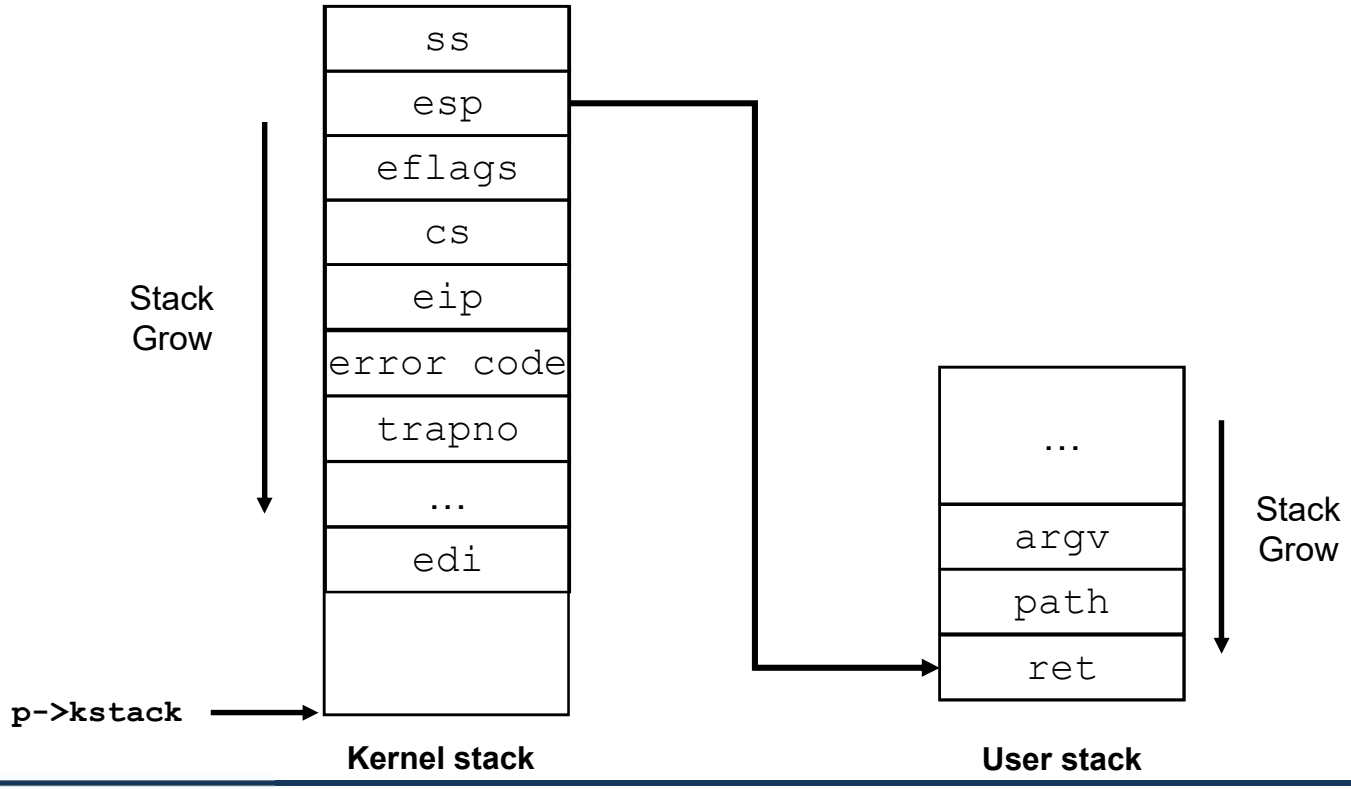
```
int main() {
  /* Stack Allocation   */
  push   %ebp
  mov    %esp,%ebp
  and    $0xfffffff0,%esp
  sub    $0x20,%esp

  /* char *uargv[3];    */
  /* uargv[0] = "echo"; */
  movl   $0x6be,0x14(%esp)
  /* uargv[1] = "hi";   */
  movl   $0x6c3,0x18(%esp)
  /* uargv[2] = 0;      */
  movl   $0x0,0x1c(%esp)

  /* printf(1, "Call exec\n"); */
  movl   $0x6c6,0x4(%esp)
  movl   $0x1,(%esp)
  call   3e0 <printf>

  /* exec("echo", uargv);      */
  lea    0x14(%esp),%eax
  mov    %eax,0x4(%esp)
  movl   $0x6be,(%esp)
  call   2ba <exec>

  /* exit(); */
  call   282 <exit>
  xchg   %ax,%ax
}
```



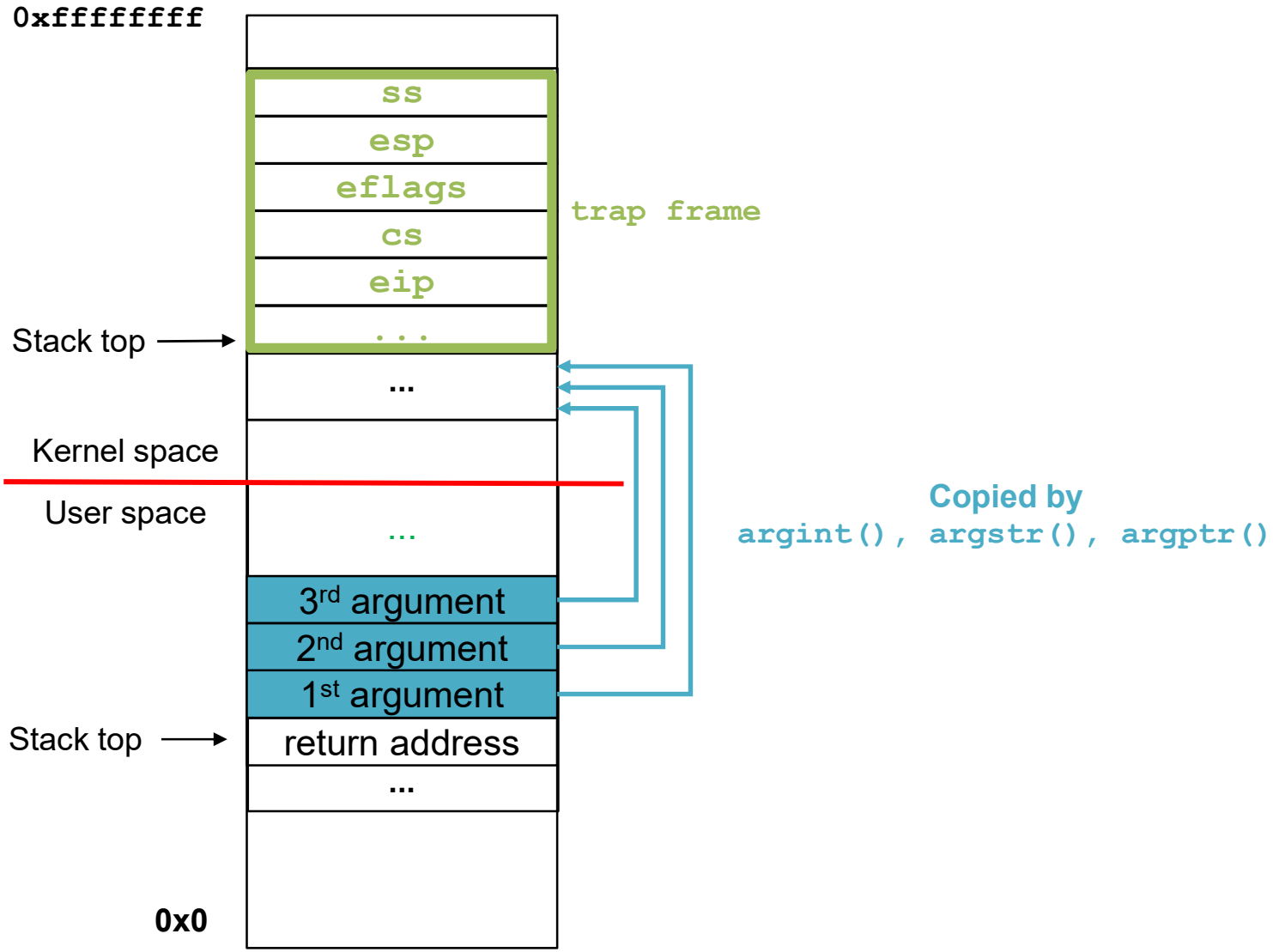
Return Address : Address of `call 282 <exit>`

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



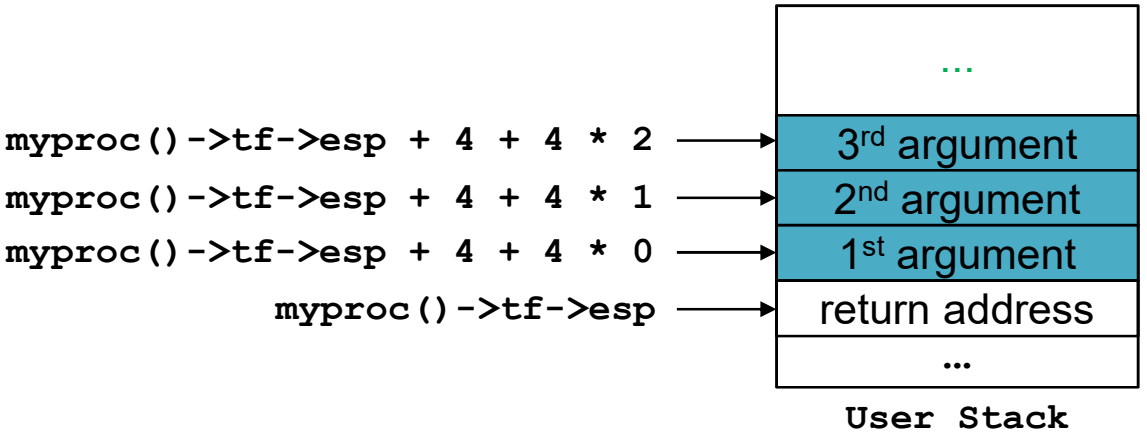
Copy integer

- `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.

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

check if the address is less than `p->sz`

Position of the argument



Copy pointer

- `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

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

  if(argint(n, &i) < 0)
    return -1;
  if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)
    return -1;
  *pp = (char*)i;
  return 0;
}
```

Address Sanity Check && Fetch the Address

Copy pointer

- `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

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

  if(argint(n, &i) < 0)
    return -1;
  if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)
    return -1;
  *pp = (char*)i;
  return 0;
}
```

Receive the `struct proc` for current process

`proc` – supplementary material

- `struct proc`
  - Data Structure for representing process, maintained for each process
  - `sz` : Size of the user memory for the process

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

proc – supplementary material

- struct proc
  - Data Structure for representing process, maintained for each process
  - sz : Size of the user memory for the process
  - User Space Address starts from 0 and go up to KERNBASE
  - sz is the limit of the user address space

```
 42   sz = 0;
 43   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
 44     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
 45       goto bad;
 52     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
 53       goto bad;
 58   }
```

exec.c, sz increases when ELF binary is loaded

```
 65   sz = PGROUNDUP(sz);
 66   if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
 67     goto bad;
 68   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
 69   sp = sz;
103   curproc->sz = sz;
```

exec.c, sz increases when the user stack is allocated

Copy pointer

- `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

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

  if(argint(n, &i) < 0)
    return -1;
  if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)
    return -1;
  *pp = (char*)i;
  return 0;
}
```

`curproc->sz` : User Address Limit, check if the address is within valid range

Copy file descriptor

- `int argfd(int n, int *pfd, struct file **pf)`
  - Copy the file descriptor to `*pfd` and copy the address of the associated file structure to `*pf`.
  - `n`: position of the parameter which contains the file descriptor.
  - `*pfd`: file descriptor
  - `*pf`: address of the file object (Kernel address space)

```
static 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;
}
```

Copy string

- `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.

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

Add a system call : testsys (syscall number 22)

- Add a new system call

```
int testsys(int i, int* ptr, char* str, int fd)
```

  - Do nothing but copy parameters from user into kernel and return.

```
#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();
}
```

User Program (testsys.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;
}
```

System Call Body
(Append at the end of sysfile.c)

Edit `Makefile` to compile and make executable testsys

Add `testsys` to variable `UPROGS`.

```
UPROGS=\
        …
        _zombie\
        _sysprog\
        _printpid\
        _testsys\
…
```

2. Add system call number to `syscall.h`.

```
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_testsys 22
```

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

```
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_testsys(void);

static int (*syscalls[])(void) = {
…
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_testsys] sys_testsys,
};
```

Edit user library: Add prototype and body

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

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

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

```
…
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(testsys)
```

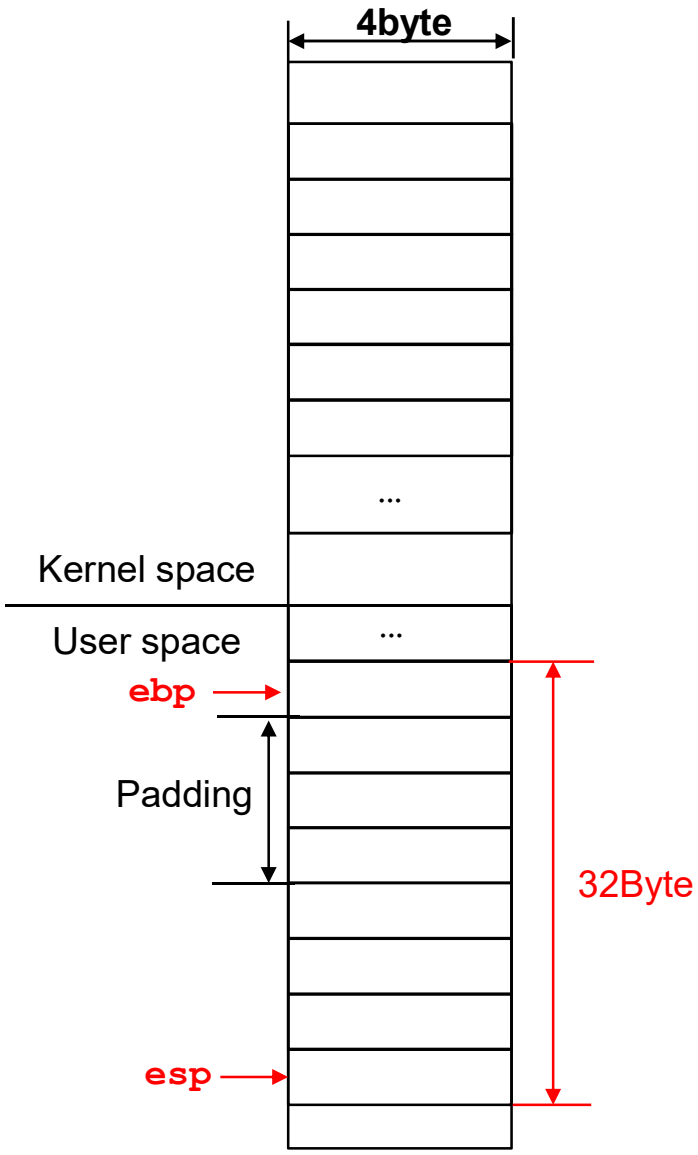
execute `testsys` program.

1. Allocate the user stack.

```
/* Start of the main() */
   push   %ebp
   mov    %esp,%ebp
   and    $0xfffffff0,%esp
   sub    $0x20,%esp
```

Allocate 32 bytes to User Stack

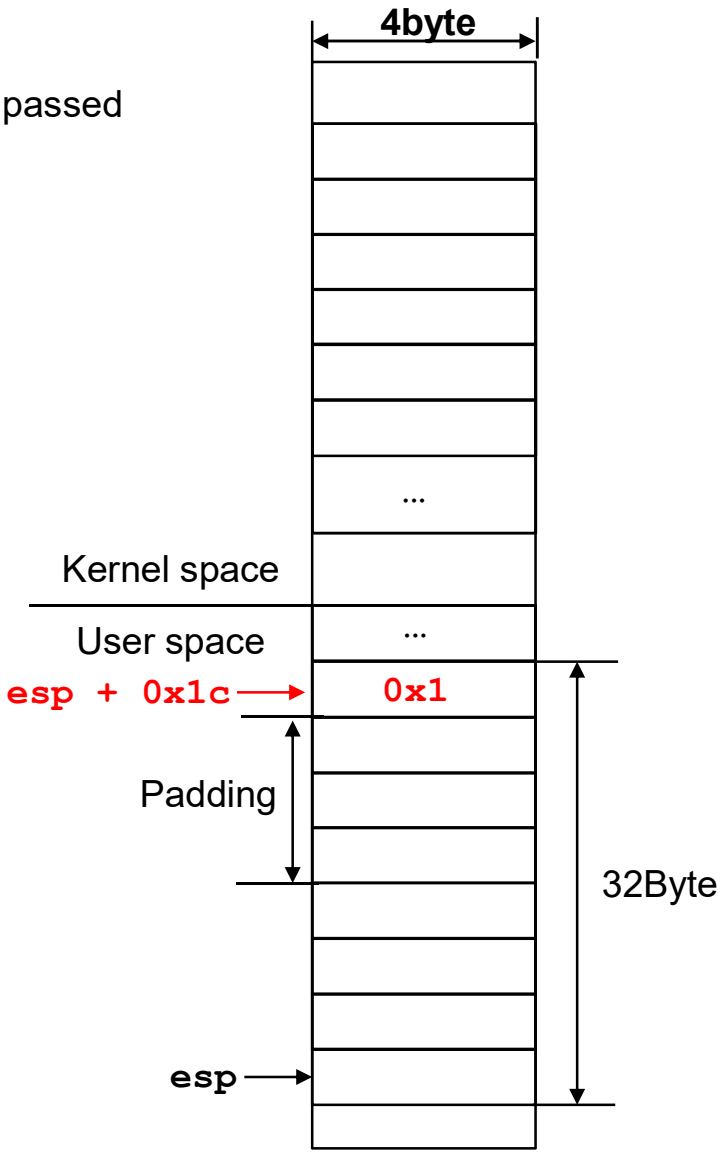
Local Variable (16) + Function Arguments (16) = 32bytes



execute `testsys` program.

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

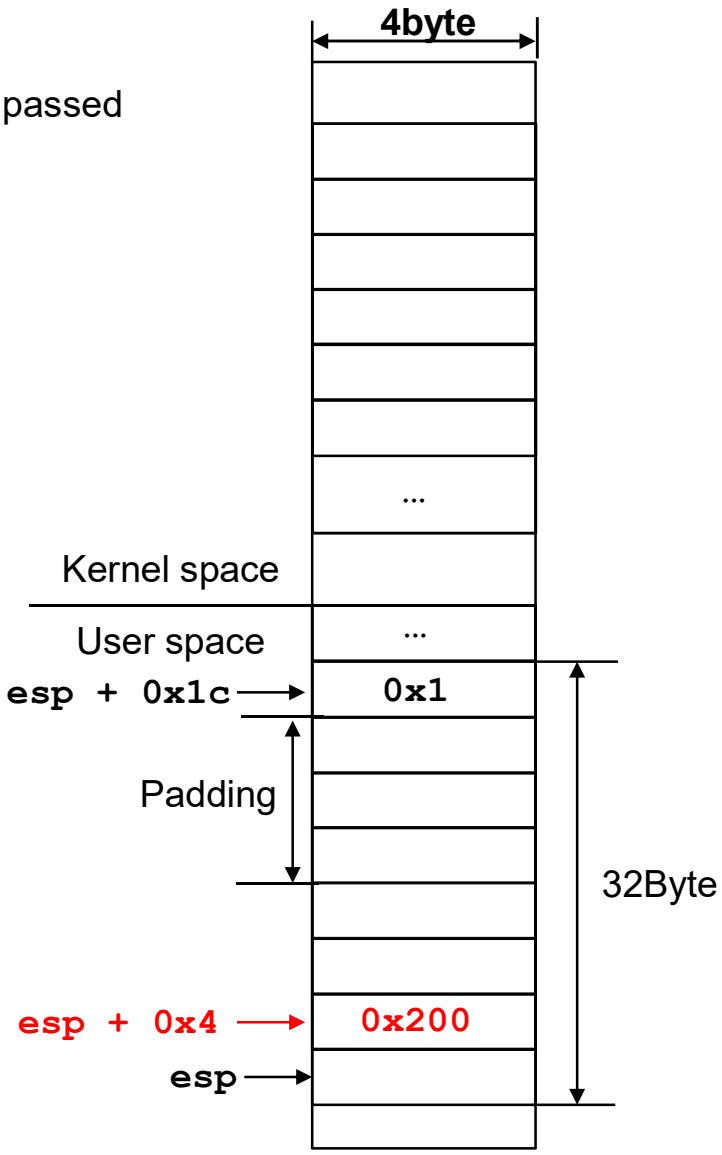
```
/* int i = 1; */
  movl   $0x1,0x1c(%esp)   0x1c = 28
/* char *str = "hello"; */
/* int fd = open("test", O_CREATE); */
  movl   $0x200,0x4(%esp)
  movl   $0x6be,(%esp)
  call   2b5 <open>
```



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; */
  movl   $0x1,0x1c(%esp)
/* char *str = "hello"; */
/* int fd = open("test", O_CREATE); */
  movl   $0x200,0x4(%esp)   0x200 == O_CREATE
  movl   $0x6be,(%esp)
  call   2b5 <open>
```



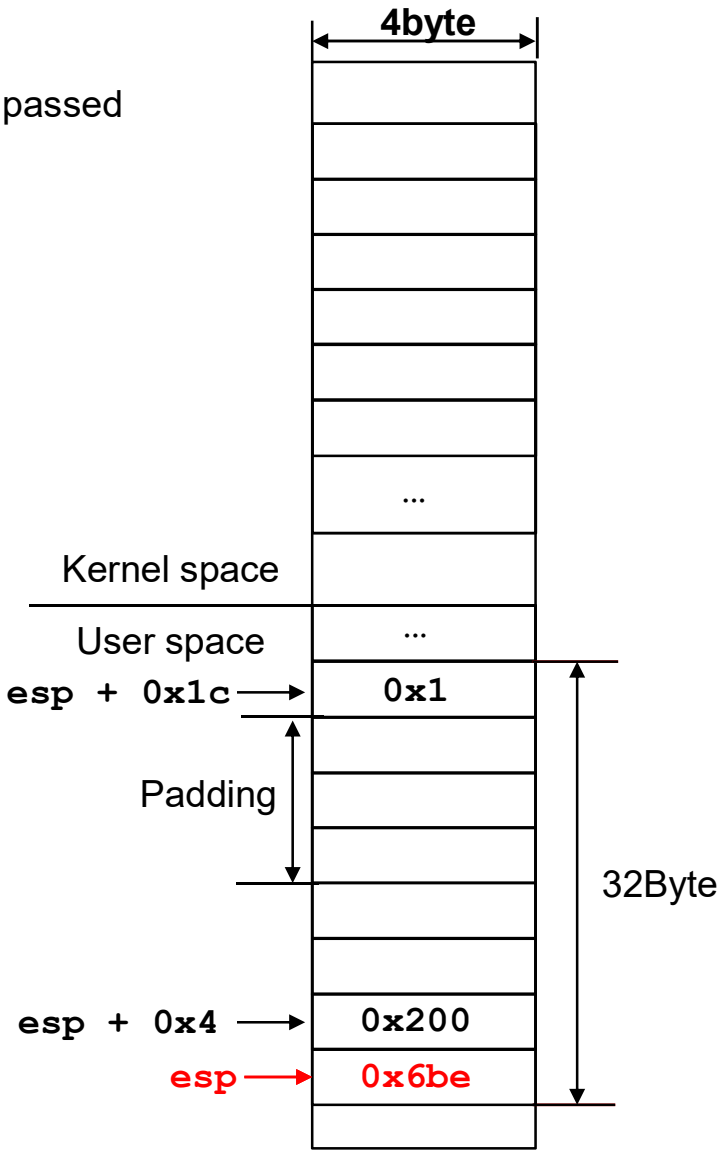
execute `testsys` program.

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

```
/* int i = 1; */
  movl   $0x1,0x1c(%esp)
/* char *str = "hello"; */
/* int fd = open("test", O_CREATE); */
  movl   $0x200,0x4(%esp)
  movl   $0x6be,(%esp)
  call   2b5 <open>
```

Data Section

0x6be "test"



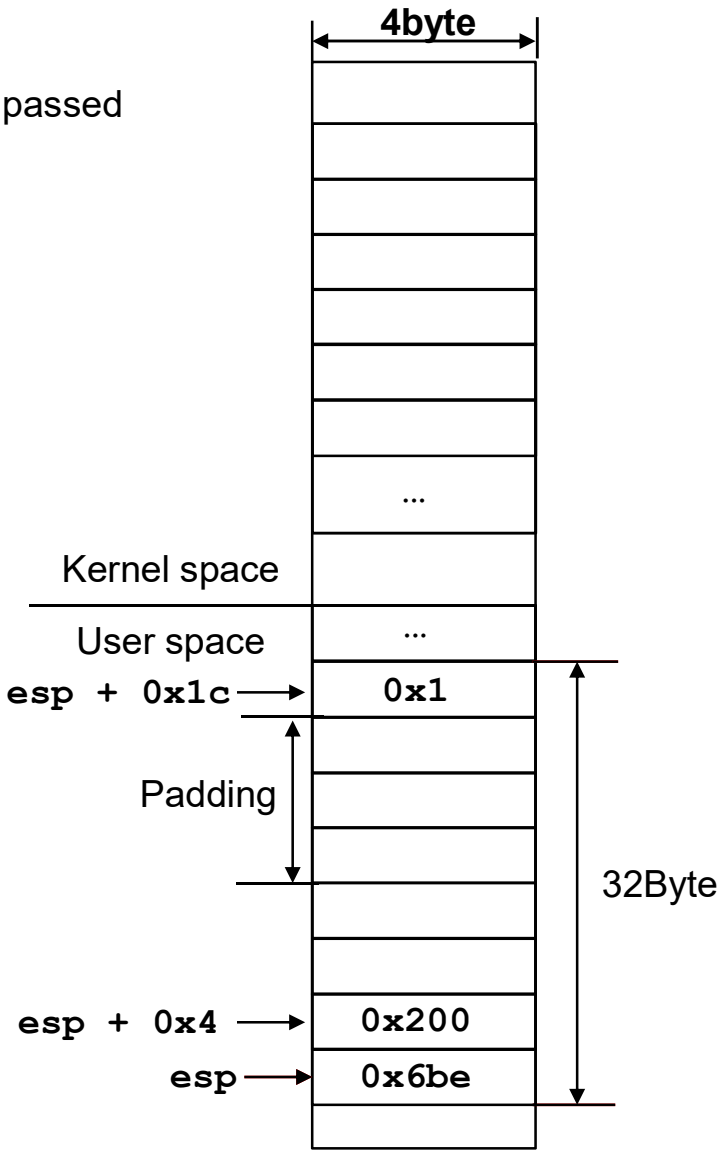
execute `testsys` program.

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

```
/* int i = 1; */
  movl   $0x1,0x1c(%esp)
/* char *str = "hello"; */
/* int fd = open("test", O_CREATE); */
  movl   $0x200,0x4(%esp)
  movl   $0x6be,(%esp)
  call   2b5 <open>
```

Register

eax	3



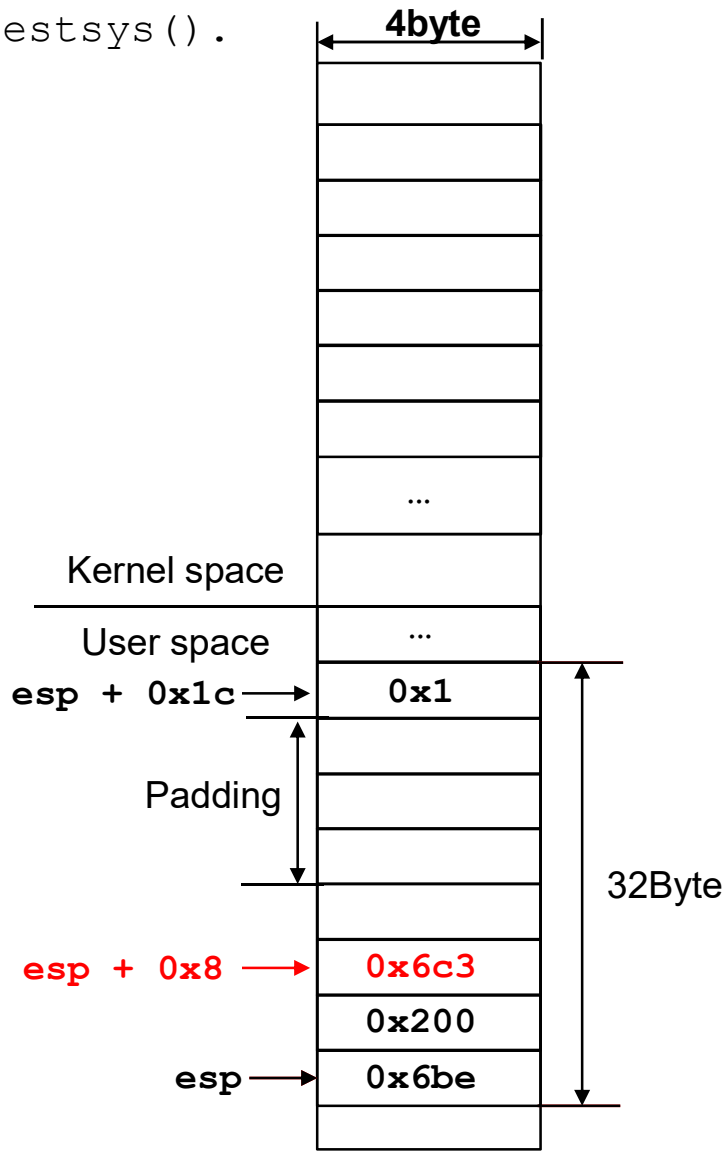
execute `testsys` program.

3. Set up arguments of the `testsys()` and call the `testsys()`.
   - Copy the address of the string "`hello`".

```
/* testsys(i, &i, str, fd); */
  movl    $0x6c3,0x8(%esp)
  mov     %eax,0xc(%esp)
  lea     0x1c(%esp),%eax
  mov     %eax,0x4(%esp)
  mov     0x1c(%esp),%eax
  mov     %eax,(%esp)
  call    322 <testsys>
/* exit(); */
  call    282 <exit>
```

Data Section

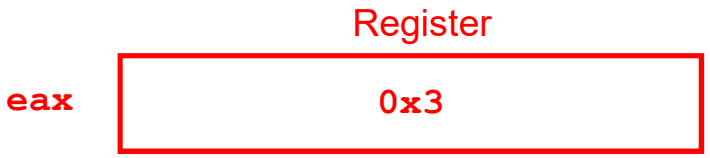
0x6c3 "hello"

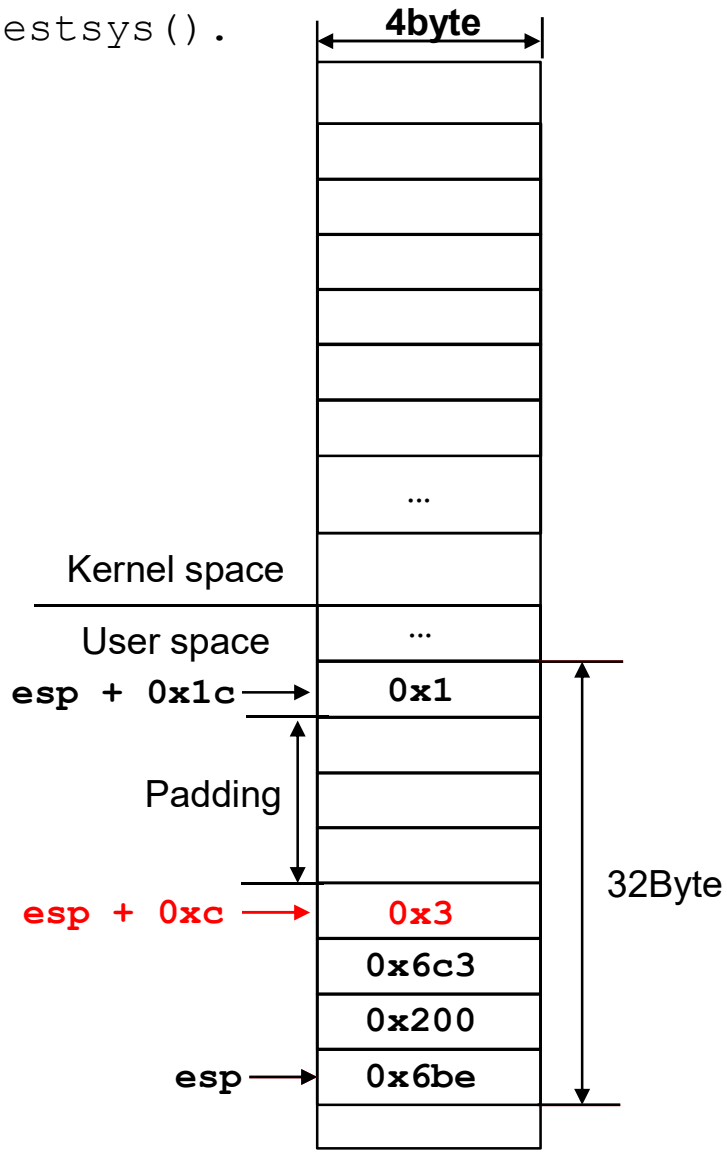


execute `testsys` program.

3. Set up arguments of the `testsys()` and call the `testsys()`.
   - Copy the file descriptor `0x3`.

```
/* testsys(i, &i, str, fd); */
  movl    $0x6c3,0x8(%esp)
  mov     %eax,0xc(%esp)
  lea     0x1c(%esp),%eax
  mov     %eax,0x4(%esp)
  mov     0x1c(%esp),%eax
  mov     %eax,(%esp)
  call    322 <testsys>
/* exit(); */
  call    282 <exit>
```





execute `testsys` program.

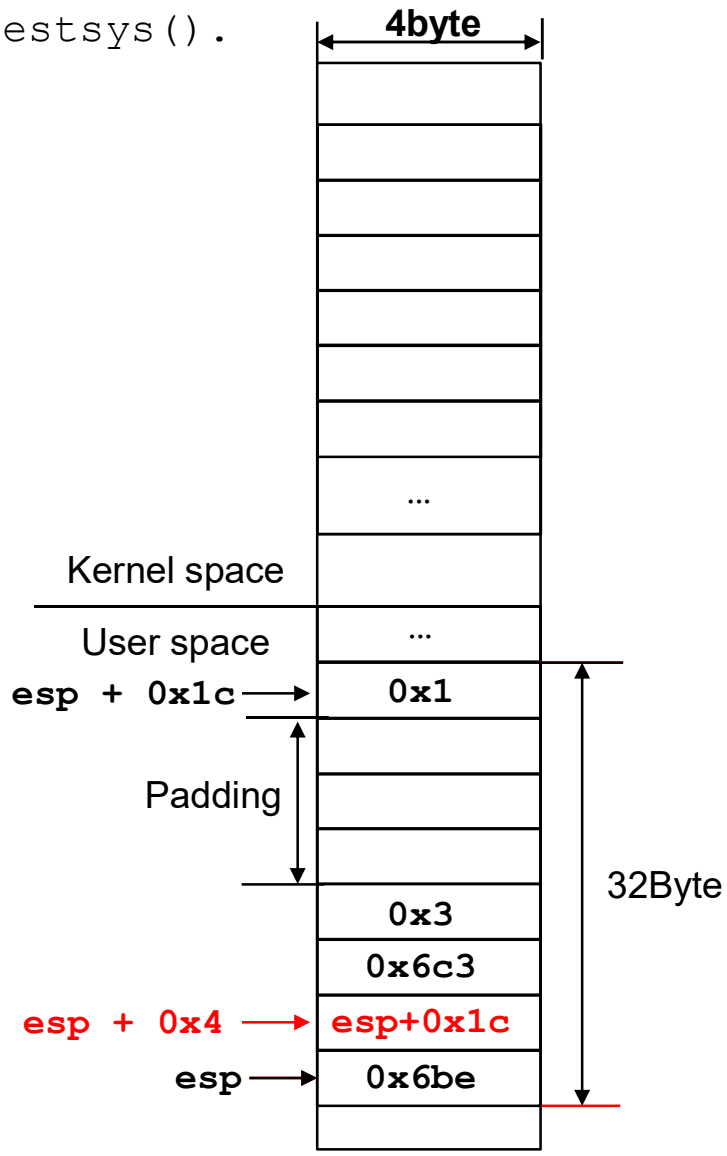
3. Set up arguments of the `testsys()` and call the `testsys()`.
   - Copy the address of local variable `i`.

```
/* testsys(i, &i, str, fd); */
  movl    $0x6c3,0x8(%esp)
  mov     %eax,0xc(%esp)
  lea     0x1c(%esp),%eax
  mov     %eax,0x4(%esp)
  mov     0x1c(%esp),%eax
  mov     %eax,(%esp)
  call    322 <testsys>
/* exit(); */
  call    282 <exit>
```

`i`'s Address : `esp + 0x1c`

Register

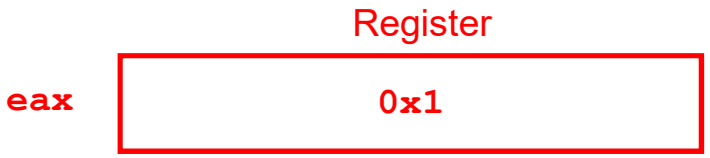
eax | esp + 0x1c

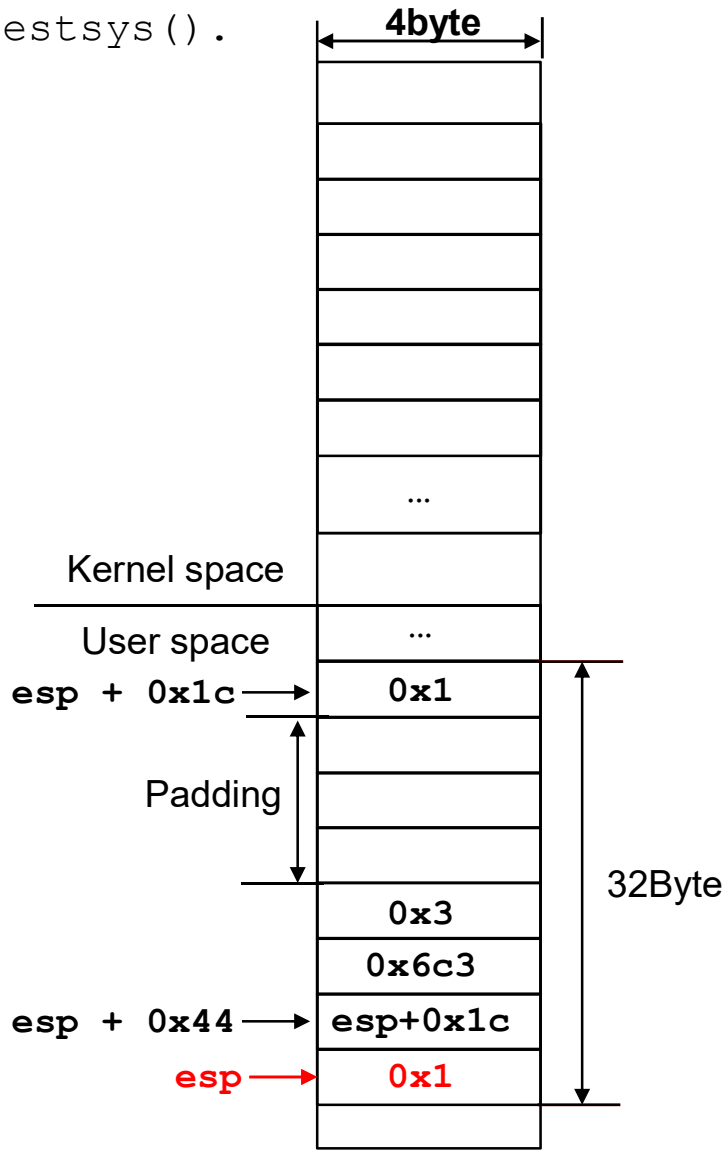


execute `testsys` program.

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

```
/* testsys(i, &i, str, fd); */
  movl    $0x6c3,0x8(%esp)
  mov     %eax,0xc(%esp)
  lea     0x1c(%esp),%eax
  mov     %eax,0x4(%esp)
  mov     0x1c(%esp),%eax
  mov     %eax,(%esp)
  call    322 <testsys>
/* exit(); */
  call    282 <exit>
```

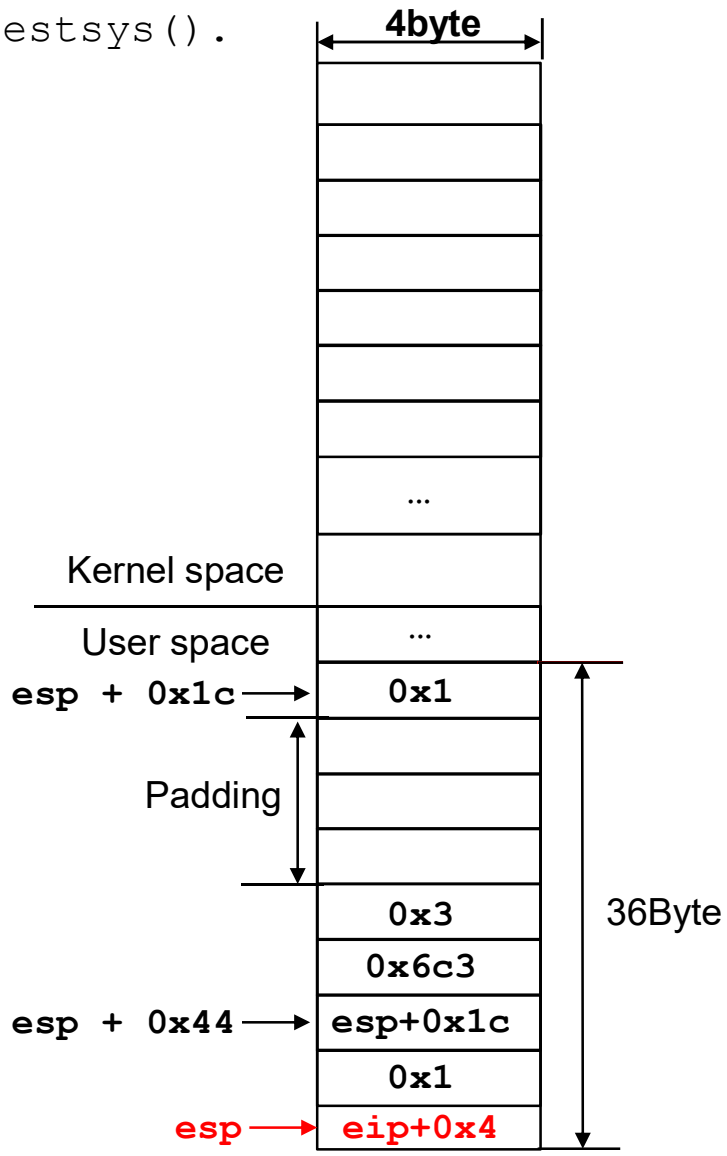




execute `testsys` program.

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

```
/* testsys(i, &i, str, fd); */
  movl    $0x6c3,0x8(%esp)
  mov     %eax,0xc(%esp)
  lea     0x1c(%esp),%eax
  mov     %eax,0x4(%esp)
  mov     0x1c(%esp),%eax
  mov     %eax,(%esp)
  call    322 <testsys>
/* exit(); */
  call    282 <exit>
```



Execute a system call.

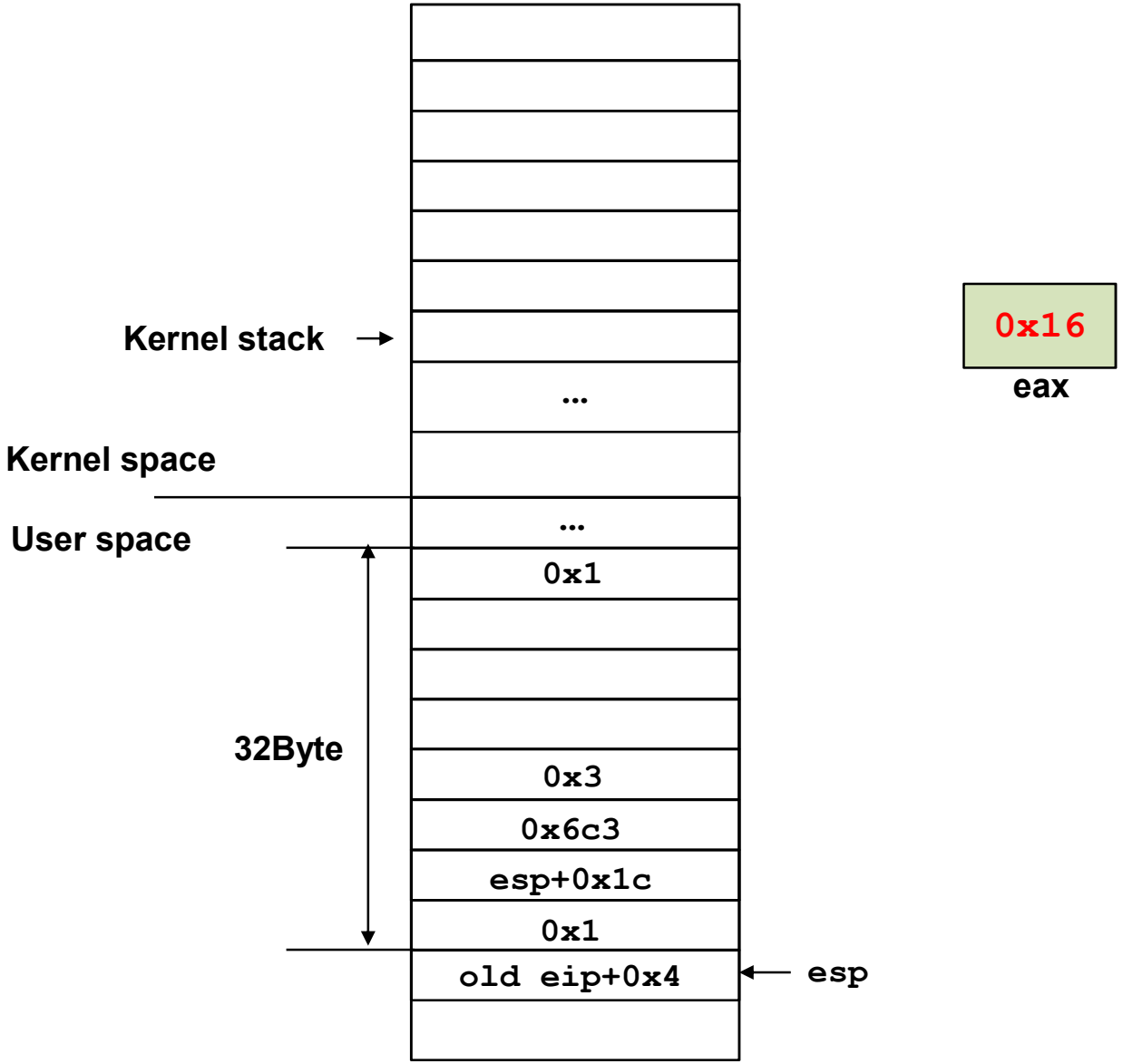
4. Save the system call number in eax register and generate interrupt.

```
00000315 <testsys>:
SYSCALL(testsys)
  mov    $0x16,%eax
  int    $0x40
  ret
```

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

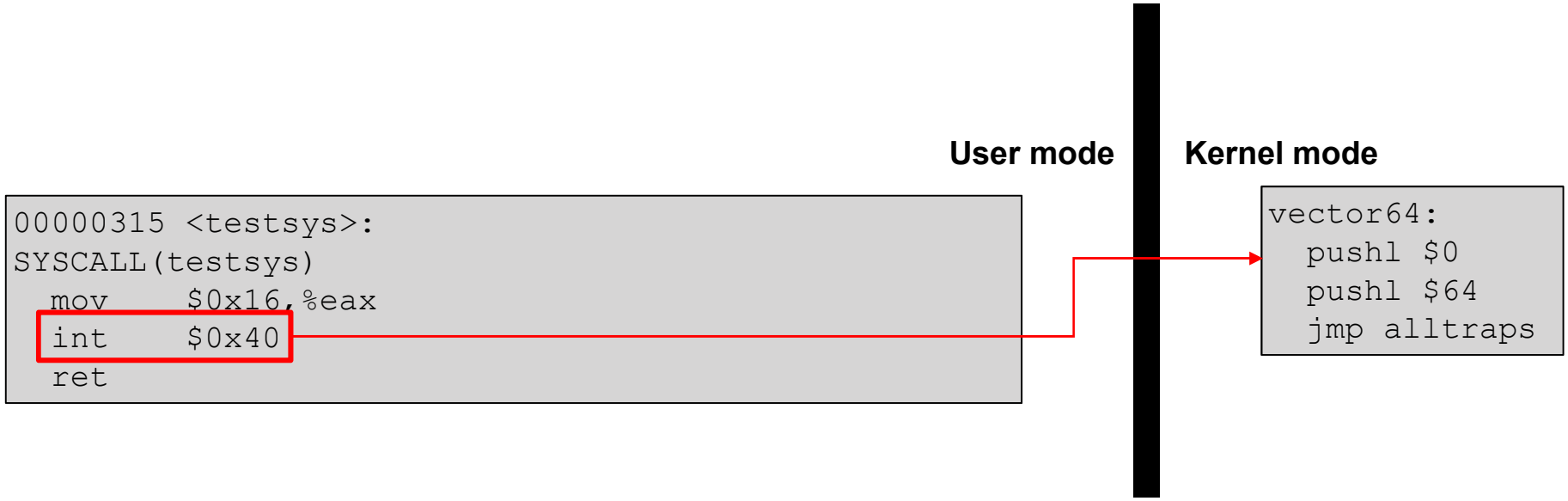
- Interrupt number for system call: **`T_SYSCALL = 0x40`**
- **`SYS_testsys = 0x16`**

Memory layout before entering the kernel

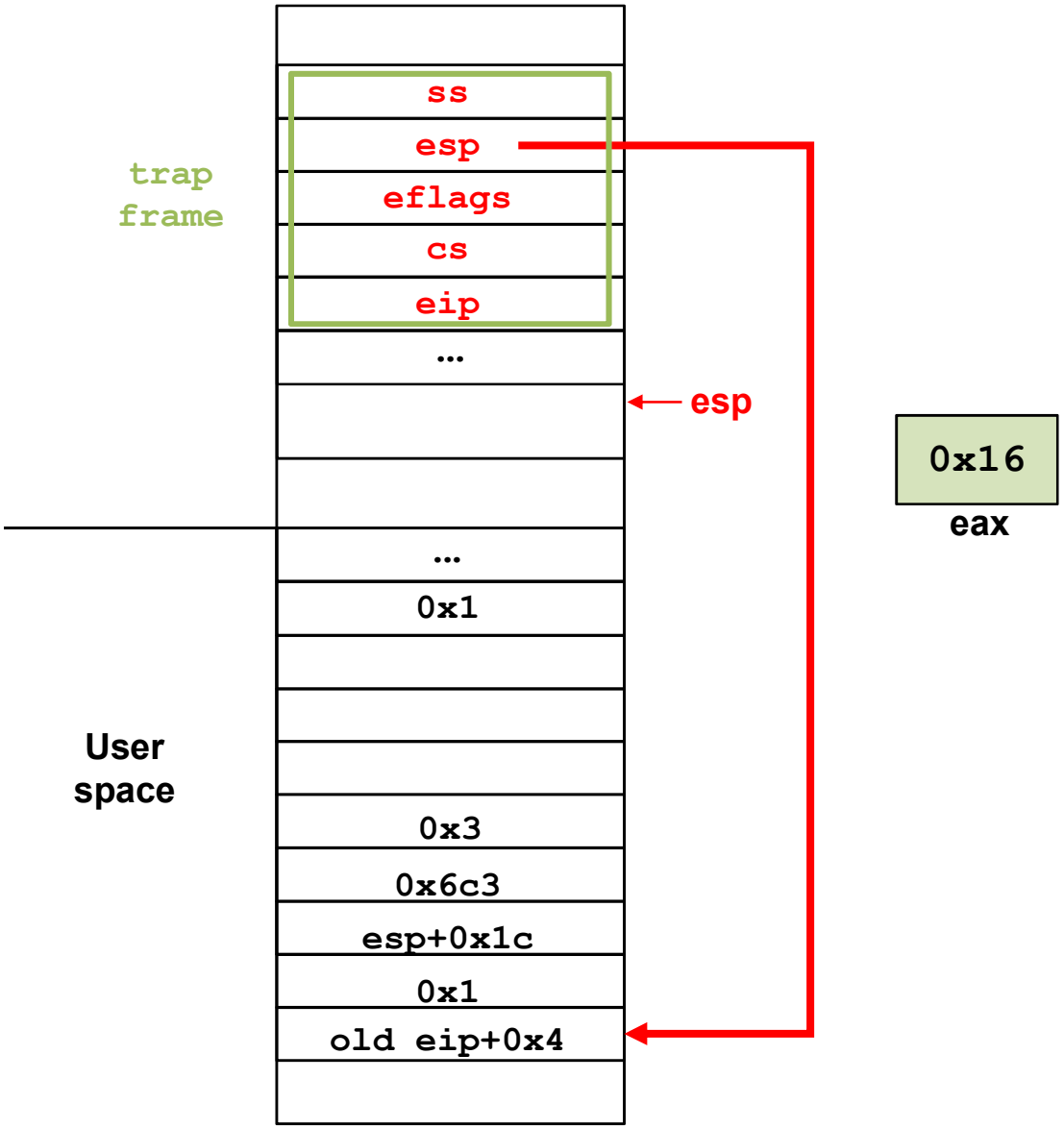


Immediately after `int` instruction

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

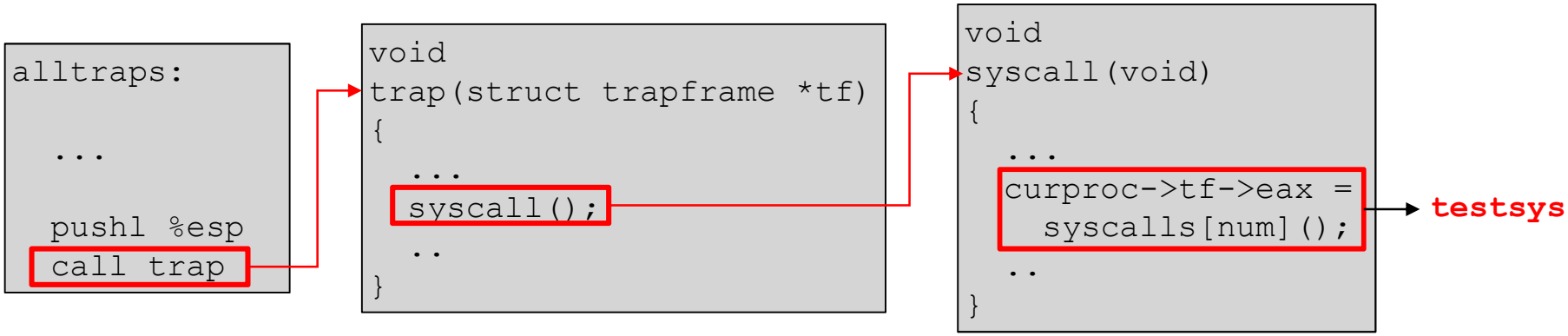


Immediately after `int` instruction



Before practice: when we execute `testsys` program...

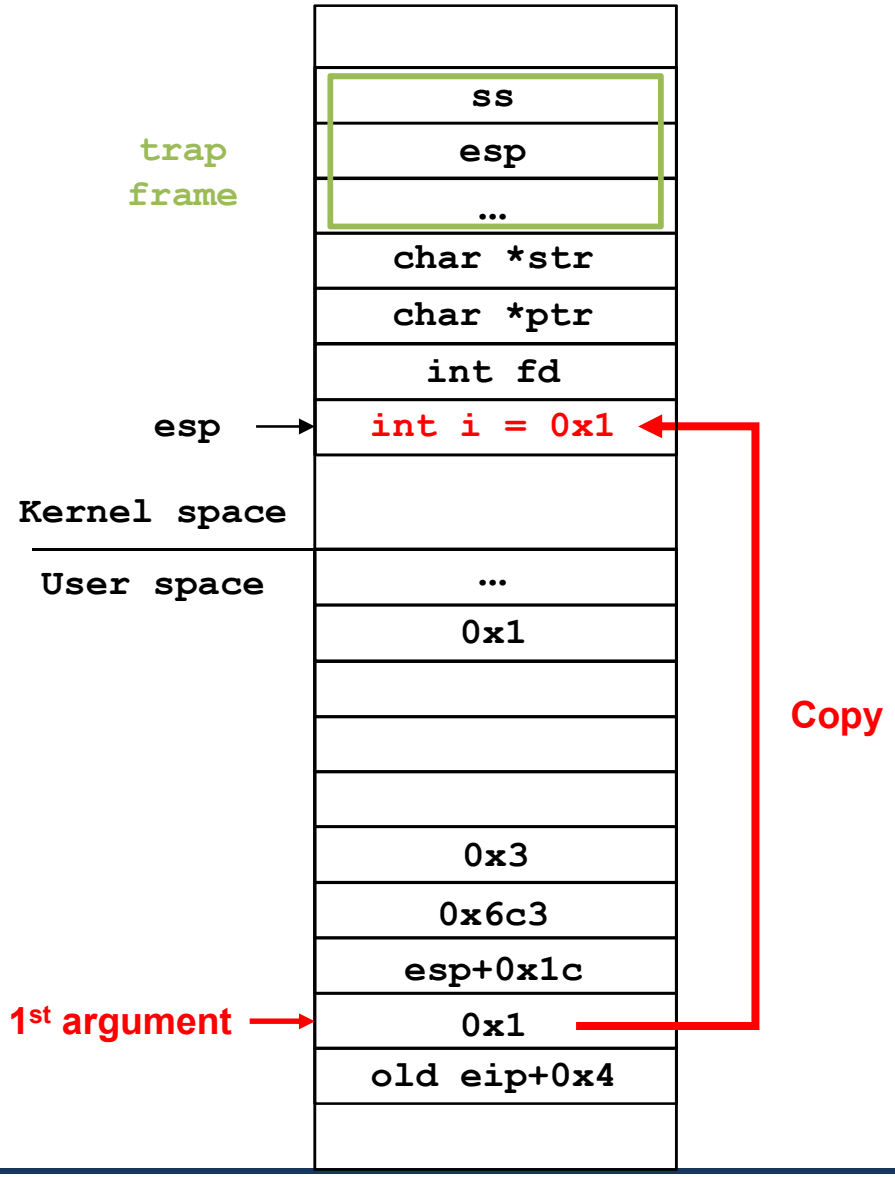
5. Set up the rest of trap frame and call the system call



Copy parameters

6. Copy the first integer argument from user stack to kernel stack.

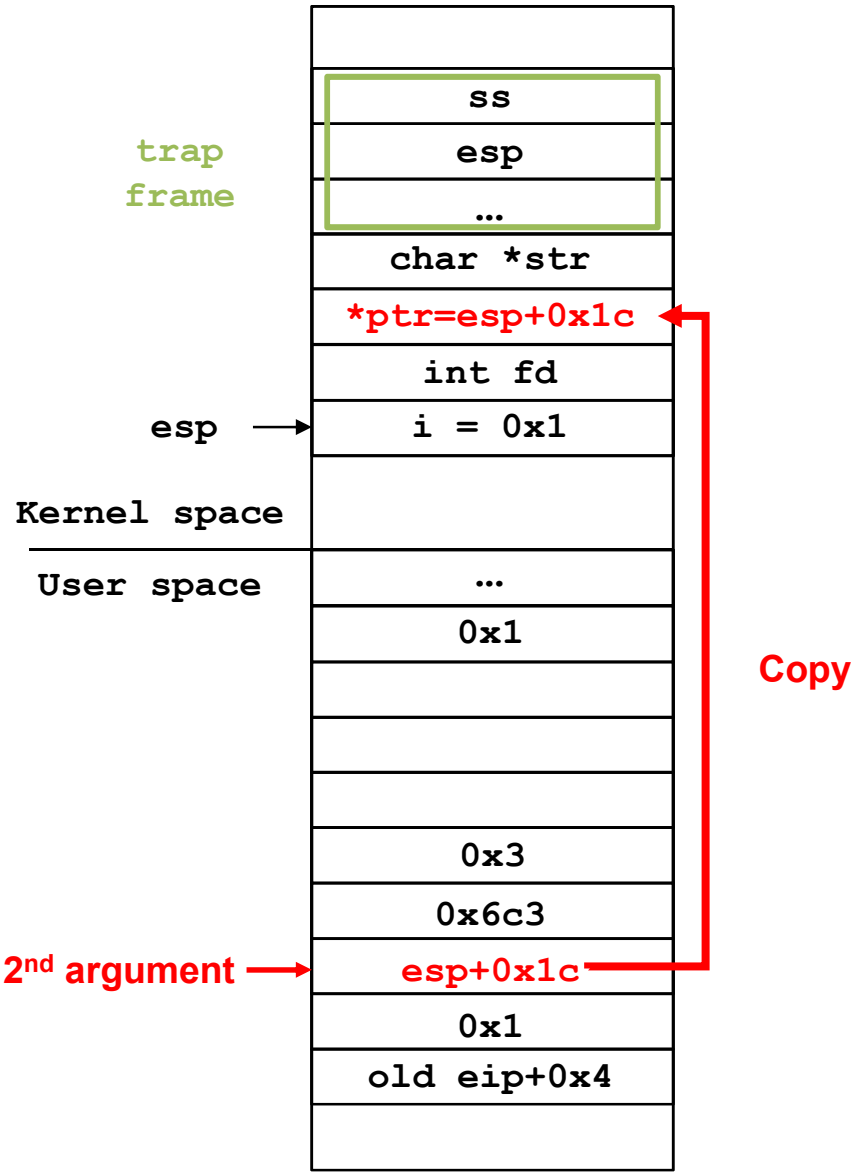
```
int sys_testsys(void) {
  int i, fd;
  char *ptr, *str;

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

  return 0;
}
```



Copy parameters

7. Copy second pointer argument from user stack to kernel stack

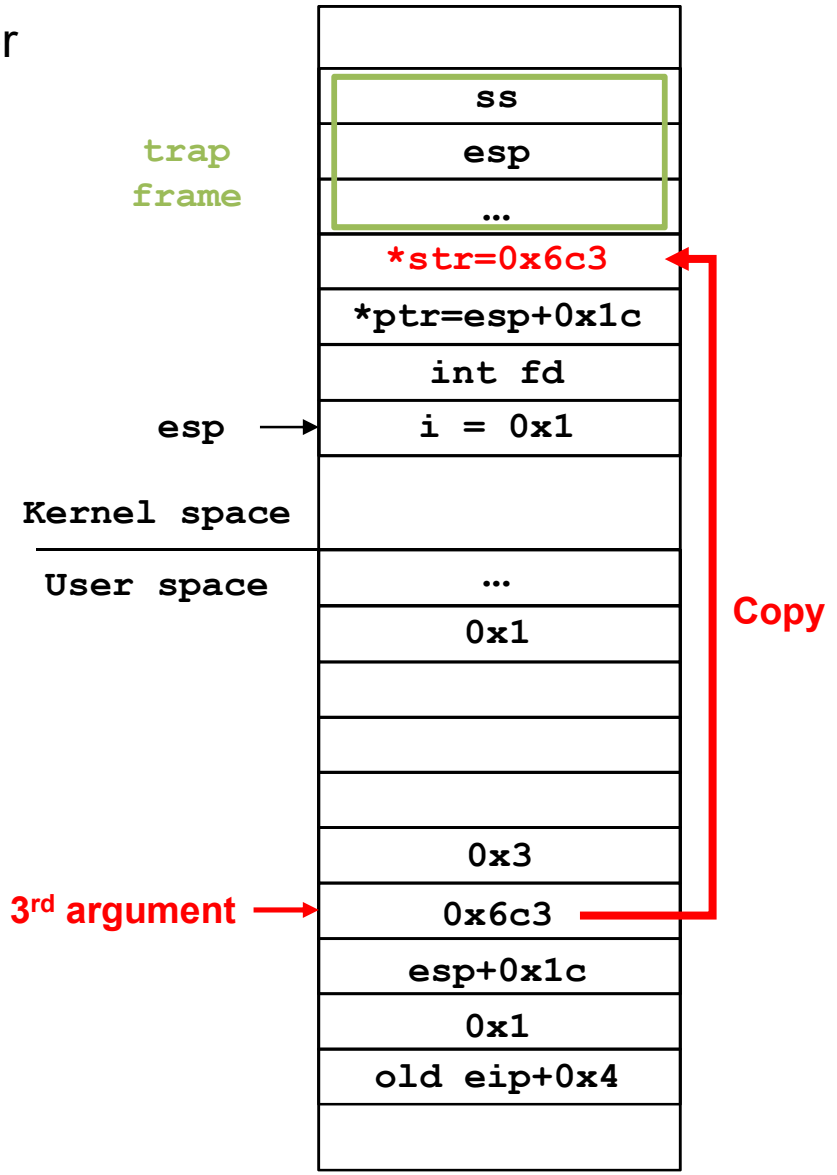
```
int sys_testsys(void) {
  int i, fd;
  char *ptr, *str;

  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

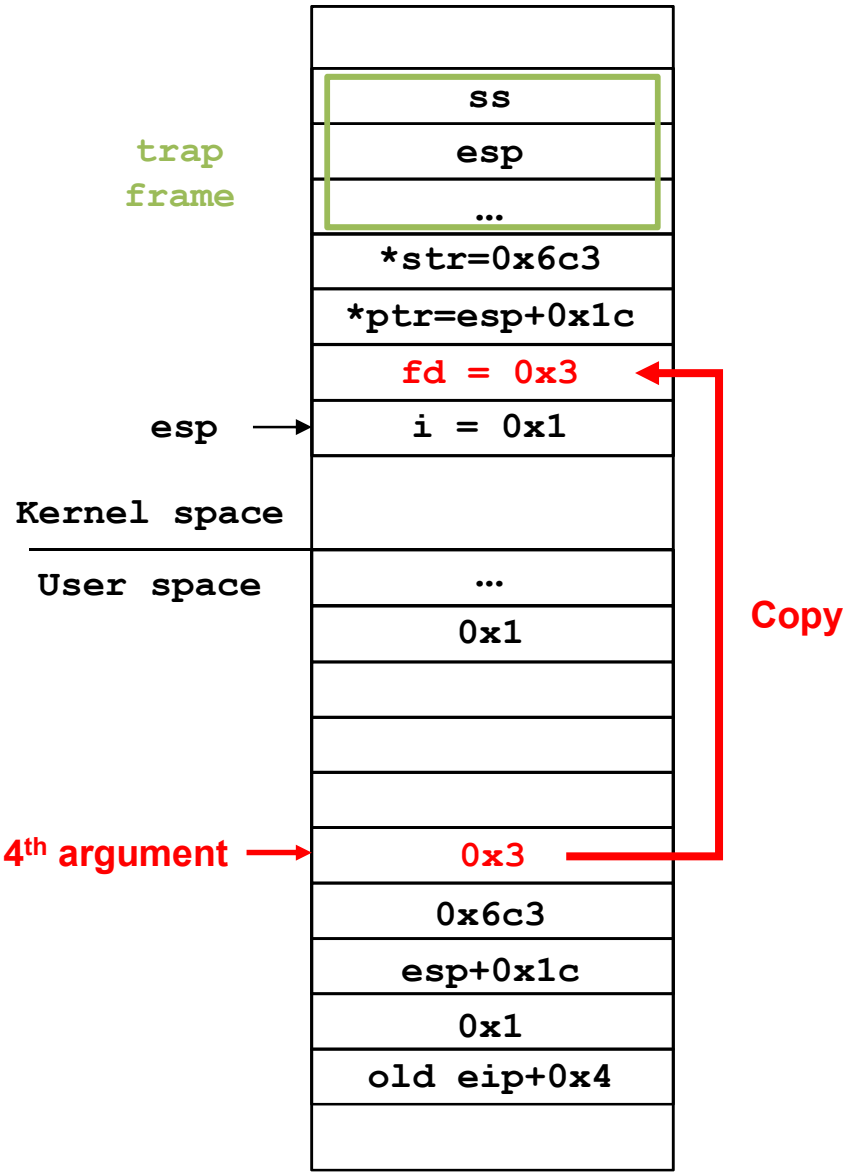
```
int sys_testsys(void) {
  int i, fd;
  char *ptr, *str;

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

  return 0;
}
```



Copy parameters

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

```
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;
}
```



EDN NETWORK

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

Excercise

- Compile and Run: Type "`make qemu-nox-gdb`".

```
laurent01@laurent01-B460MAORUSPRO:~/workspace/xv6-public$ make qemu-nox-gdb

*** Now run 'gdb'.
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=
0,media=disk,format=raw -smp 0
```

- Open another shell and Type `gdb` in xv6 directory

```
laurent01@laurent01-B460MAORUSPRO:~/workspace/xv6-public$ gdb -q ./kernel
Reading symbols from ./kernel...
+ target remote localhost:26000
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_testsys
Breakpoint 1 at 0x801056c0: file sysfile.c, line 447.
(gdb) c
Continuing.
```

Excercise

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

```
init: starting sh
$ ./testsys
```

- Dump the user stack - the user stack just before the `sys_testsys` function

```
(gdb) x/24x myproc()->tf->esp
0x2f9c: 0x0000003d      0x00000001      0x00002fcc      0x000006c3
0x2fac: 0x00000003      0x000006be      0x00000200      0x00000000
0x2fbc: 0x00000000      0x00000000      0x00000000      0x00000000
0x2fcc: 0x00000001      0x00000000      0x00002fe4      0x00003fc8
0x2fdc: 0xffffffff      0xffffffff      0x00000001      0x00002fec
0x2fec: 0x00002ff4      0x00000000      0x65742f2e      0x79737473
```

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

```
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
0x8dfbef30:     0x00000016      0x00000000      0x00000000      0x00000010
0x8dfbef40:     0x00000010      0x00000020      0x00000000      0x801056c1
0x8dfbef50:     0x[illegible]008      0[illegible]6      0[illegible]78      0x[illegible]ace
0x8dfbef60:     0x00000000      0x00000000      0x8dfbef78      0x801059d0
0x8dfbef70:     0x00000000      0x00000000      0x8dfbefa8      0x80105bb9
0x8dfbef80:     0x80104535      0x01010101      0x00000000      0x8dfbefac
```

int i; char *ptr char *str int fd

Run testsys and examine user stack

```
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
0x8dfbef30:     0x00002fa0      0x8dfbef40      0x8dfbef3c      0x8dfbef4f
0x8dfbef40:     0x00000001      0x00000000      0x8dfbef58      0xcc1039d0
0x8dfbef50:     0x80112e4c      0x00000000      0x8dfbef78      0x80104ace
0x8dfbef60:     0x00000000      0x00000000      0x8dfbef78      0x801039d0
0x8dfbef70:     0x00000000      0x00000000      0x8dfbefa8      0x80105bb9
0x8dfbef80:     0x80104535      0x01010101      0x00000000      0x8dfbefac
```

Run testsys and examine user stack

```
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;
}
```

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) x/24x $esp
0x8dfbef30:     0x00000001      0x8dfbef44      0x00000004      0x8dfbef4f
0x8dfbef40:     0x00000001      0x00002fcc      0x8dfbef58      0xcc1039d0
0x8dfbef50:     0x80112e4c      0x00000000      0x8dfbef78      0x80104ace
0x8dfbef60:     0x00000000      0x00000000      0x8dfbef78      0x801039d0
0x8dfbef70:     0x00000000      0x00000000      0x8dfbefa8      0x80105bb9
0x8dfbef80:     0x80104535      0x01010101      0x00000000      0x8dfbefac
```

Run testsys and examine user stack

```
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 0x6c3
0x6c3:  "hello"
```

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

```
(gdb) x/24x $esp
0x8dfbef30:     0x00000002      0x8dfbef48      0x00000004      0x8dfbef4f
0x8dfbef40:     0x00000001      0x00002fcc      0x000006c3      0xcc1039d0
0x8dfbef50:     0x80112e4c      0x00000000      0x8dfbef78      0x80104ace
0x8dfbef60:     0x00000000      0x00000000      0x8dfbef78      0x801039d0
0x8dfbef70:     0x00000000      0x00000000      0x8dfbefa8      0x80105bb9
0x8dfbef80:     0x80104535      0x01010101      0x00000000      0x8dfbefac
```

Run testsys and examine user stack

```
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
0x8dfbef30:     0x00000002      0x8dfbef48      0x00000004      0x8dfbef4f
0x8dfbef40:     0x00000001      0x00002fcc      0x000006c3      0x00000003
0x8dfbef50:     0x80112e4c      0x00000000      0x8dfbef78      0x80104ace
0x8dfbef60:     0x00000000      0x00000000      0x8dfbef78      0x801039d0
0x8dfbef70:     0x00000000      0x00000000      0x8dfbefa8      0x80105bb9
0x8dfbef80:     0x80104535      0x01010101      0x00000000      0x8dfbefac
```

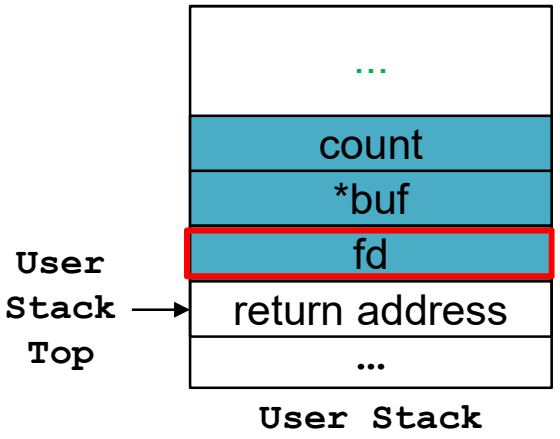
Appendix

Copy Parameters to Kernel Stack – Example

- `int write(int fd, const void *buf, size_t count)`
  - `write` System Call requires 3 arguments including integer and pointer.
  - `write` is implemented with `sys_write` in the `xv6`

```
int sys_write(void) {
  struct file *f; int n; char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return filewrite(f, p, n);
}
```

Fetch the address of the **`struct file`** associated with the file descriptor located at the user stack

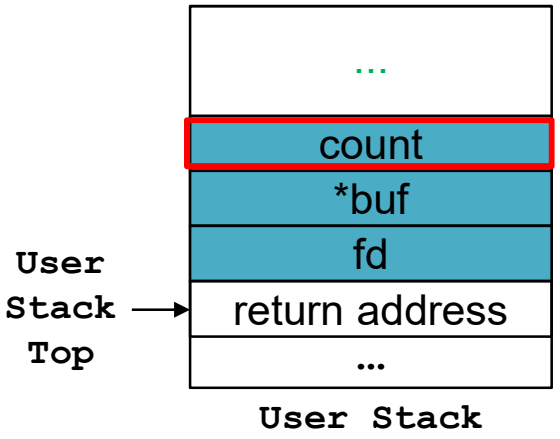


Copy Parameters to Kernel Stack – Example

- `int write(int fd, const void *buf, size_t count)`
  - `write` System Call requires 3 arguments including integer and pointer.
  - `write` is implemented with `sys_write` in the `xv6`

```
int sys_write(void) {
  struct file *f; int n; char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return filewrite(f, p, n);
}
```

Fetch the size value
to the kernel stack variable

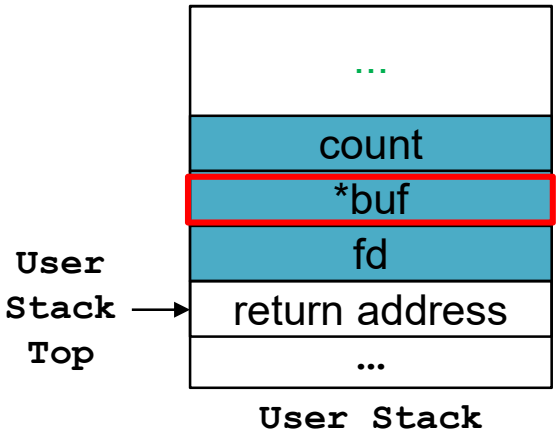


Copy Parameters to Kernel Stack – Example

- `int write(int fd, const void *buf, size_t count)`
  - `write` System Call requires 3 arguments including integer and pointer.
  - `write` is implemented with `sys_write` in the `xv6`

```
int sys_write(void) {
  struct file *f; int n; char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return filewrite(f, p, n);
}
```

Fetch the buffer address
to the kernel stack variable



After `trapframe` is setup…