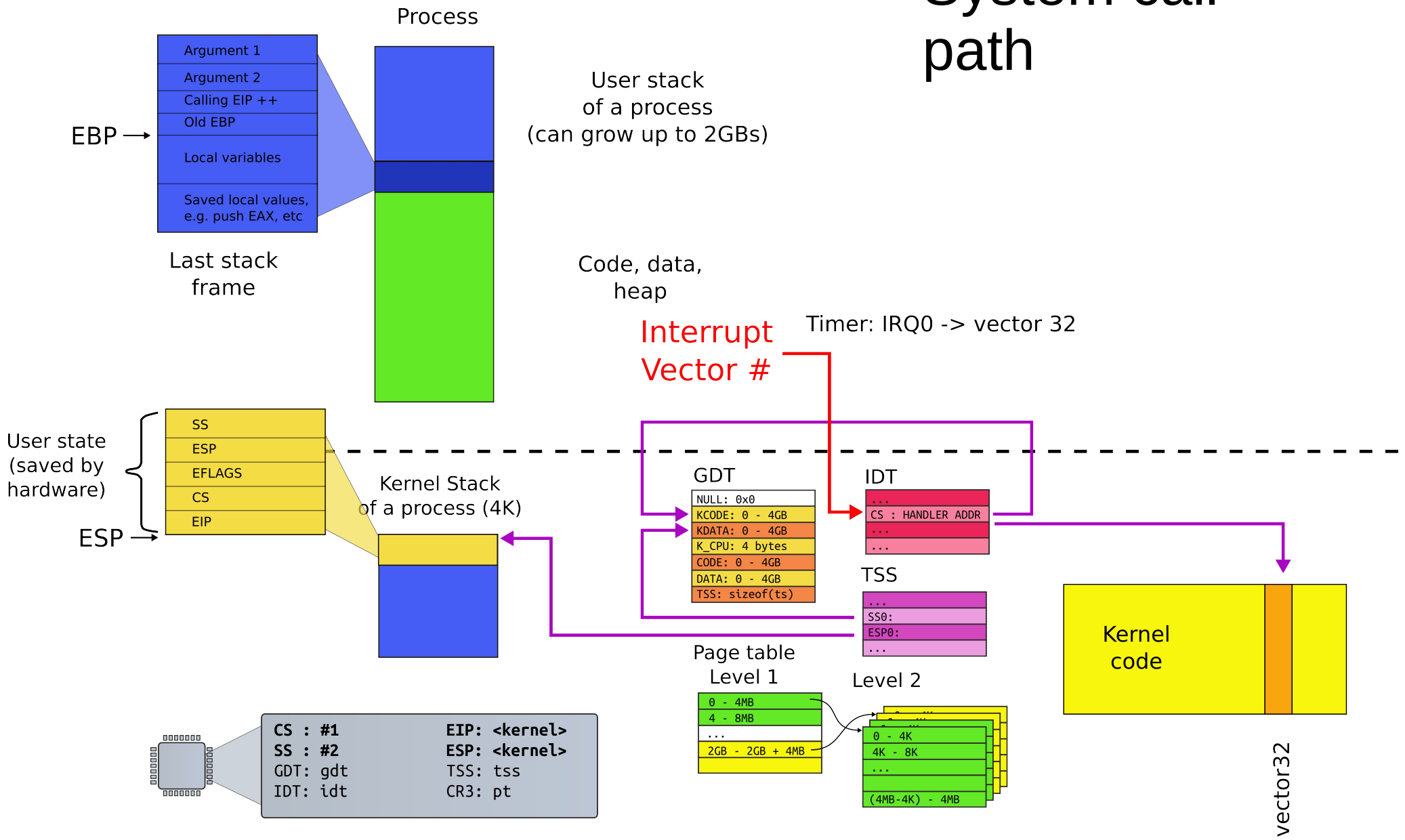
ICS143A: Principles of Operating Systems

Lecture 14: System calls (part 2)

Anton Burtsev
November, 2017

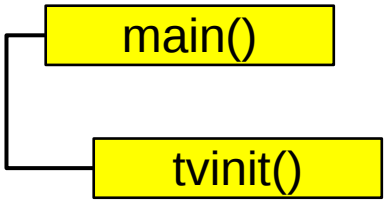
System call path



Initialize IDT

- tvinit() is called from main()

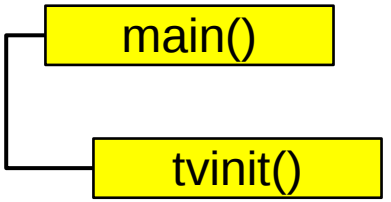
```
3316 void
3317 tvinit(void)
3318 {
3319   int i;
3320
3321   for(i = 0; i < 256; i++)
3322     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3323   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
                vectors[T_SYSCALL], DPL_USER);
3324
3325   initlock(&tickslock, "time");
3326 }
```



Initialize IDT

- A couple of important details

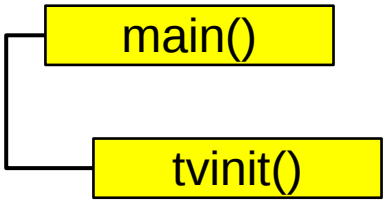
```
3316 void
3317 tvinit(void)
3318 {
3319   int i;
3320
3321   for(i = 0; i < 256; i++)
3322     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3323   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
                vectors[T_SYSCALL], DPL_USER);
3324
3325   initlock(&tickslock, "time");
3326 }
```



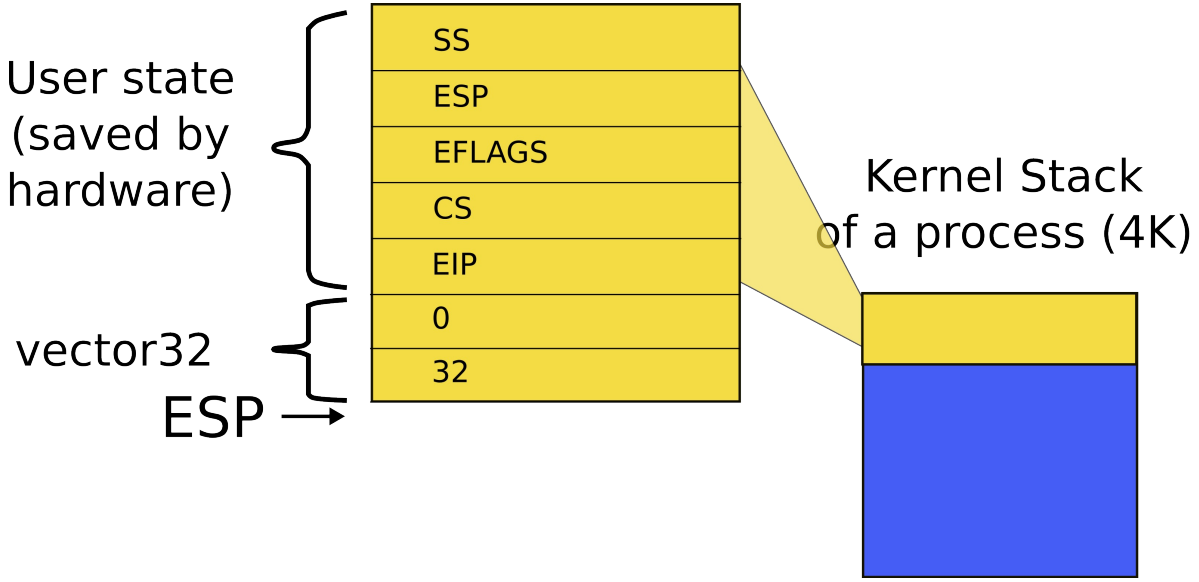
Initialize IDT

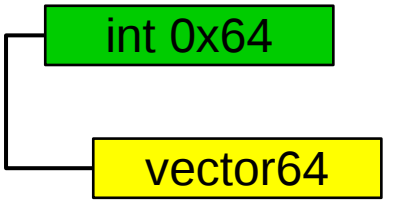
- Only int T_SYSCALL can be called from user-level

```
3316 void
3317 tvinit(void)
3318 {
3319   int i;
3320
3321   for(i = 0; i < 256; i++)
3322     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3323   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
                vectors[T_SYSCALL], DPL_USER);
3324
3325   initlock(&tickslock, "time");
3326 }
```



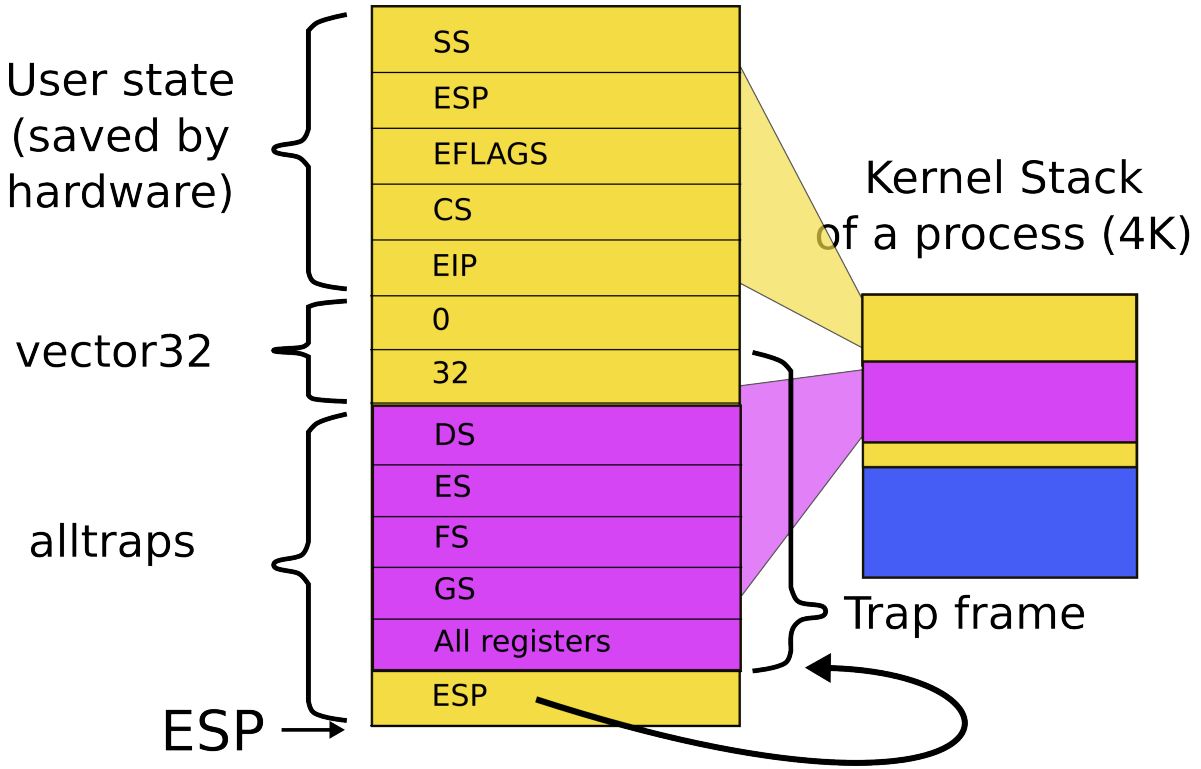
Kernel stack after interrupt



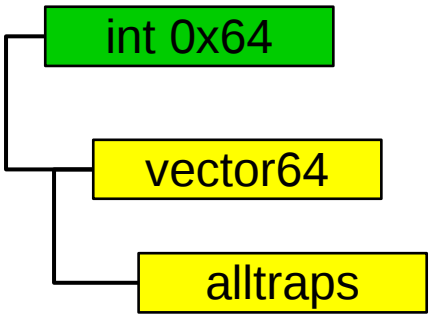


Call stack: vector32()

Kernel stack after interrupt



Call stack: vector32()
alltraps()

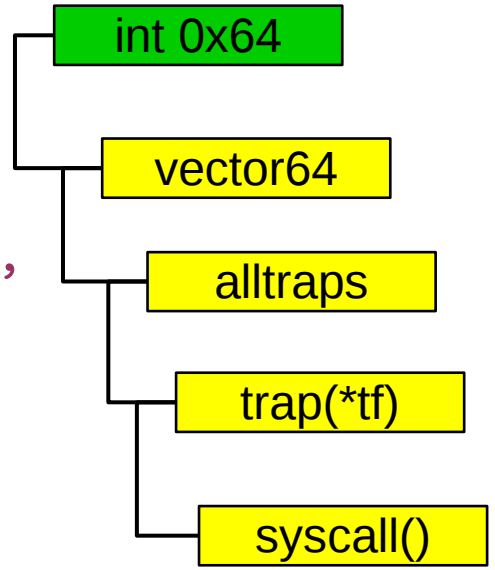


Syscall number

- System call number is passed in the %eax register
  - To distinguish which syscall to invoke,
    - e.g., sys_read, sys_exec, etc.
- alltrap() saves it along with all other registers

syscall(): get the number from the trap frame

```
3625 syscall(void)
3626 {
3627   int num;
3628
3629   num = proc->tf->eax;
3630   if(num > 0 && num < NELEM(syscalls) && syscalls[num])
       {
3631     proc->tf->eax = syscalls[num]();
3632   } else {
3633     cprintf("%d %s: unknown sys call %d\n",
3634     proc->pid, proc->name, num);
3635     proc->tf->eax = -1;
3636   }
3637 }
```



syscall(): process a syscall from the table

```
3625 syscall(void)
3626 {
3627   int num;
3628
3629   num = proc->tf->eax;
3630   if(num > 0 && num < NELEM(syscalls) && syscalls[num])
       {
3631     proc->tf->eax = syscalls[num]();
3632   } else {
3633     cprintf("%d %s: unknown sys call %d\n",
3634     proc->pid, proc->name, num);
3635     proc->tf->eax = -1;
3636   }
3637 }
```

System call table

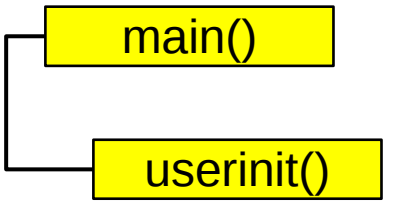
```
3600 static int (*syscalls[])(void) = {
3601   [SYS_fork] sys_fork,
3602   [SYS_exit] sys_exit,
3603   [SYS_wait] sys_wait,
3604   [SYS_pipe] sys_pipe,
3605   [SYS_read] sys_read,
3606   [SYS_kill] sys_kill,
3607   [SYS_exec] sys_exec,
3608   [SYS_fstat] sys_fstat,
3609   [SYS_chdir] sys_chdir,
3610   [SYS_dup] sys_dup,
3611   [SYS_getpid] sys_getpid,
3612   [SYS_sbrk] sys_sbrk,
3613   [SYS_sleep] sys_sleep,
3614   [SYS_uptime] sys_uptime,
3615   [SYS_open] sys_open,
3616   [SYS_write] sys_write,
3617   [SYS_mknod] sys_mknod,
3618   [SYS_unlink] sys_unlink,
3619   [SYS_link] sys_link,
3620   [SYS_mkdir] sys_mkdir,
3621   [SYS_close] sys_close,
3622 };
```

What do you think is the first system call xv6 executes?

```
1317 main(void)
1318 {
1319   kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320   kvmalloc(); // kernel page table
1321   mpinit(); // detect other processors
...
1323   seginit(); // segment descriptors
...
1330   tvinit(); // trap vectors
...
1338   userinit(); // first user process
1339   mpmain(); // finish this processor's setup
1340 }
```

main()

```
2502 userinit(void)
2503 {
2504   struct proc *p;
2505   extern char _binary_initcode_start[],
                  _binary_initcode_size[];
...
2509   p = allocproc();
2510   initproc = p;
2511   if((p->pgdir = setupkvm()) == 0)
2512     panic("userinit: out of memory?");
2513   inituvm(p->pgdir, _binary_initcode_start,
                (int)_binary_initcode_size);
2514   p->sz = PGSIZE;
2515   memset(p->tf, 0, sizeof(*p->tf));
...
2530 }
```



```
8409 start:
8410   pushl $argv
8411   pushl $init
8412   pushl $0 // where caller pc would be
8413   movl $SYS_exec, %eax
8414   int $T_SYSCALL
8415
...
8422 # char init[] = "/init\0";
8423 init:
8424   .string "/init\0"
8425
8426 # char *argv[] = { init, 0 };
8427 .p2align 2
8428 argv:
8429   .long init
8430   .long 0
```

initcode.S: call exec("/init", argv);

- exec("/init", argv) has two arguemens
- Push arguments on the stack
- Invoke system call with
  - `int $T_SYSCALL`

How do user programs access system calls?

- It would be weird to write

```
8410   pushl $argv
8411   pushl $init
8412   pushl $0 // where caller pc would be
8413    movl $SYS_exec, %eax
8414    int $T_SYSCALL
```

  - … every time we want to invoke a system call

user.h

```
// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(char*, int);
int mknod(char*, short, short);
int unlink(char*);
int fstat(int fd, struct stat*);
int link(char*, char*);
...
```

- user.h defines system call prototypes
- Compiler can generate correct system call stacks
  - Remember calling conventions?
  - Arguments on the stack

Example

- From cat.asm
- `if (write(1, buf, n) != n)`

```
A3:    53                    push   ebx
a4:    68 00 0b 00 00        push   0xb00
a9:    6a 01                 push   0x1
ab:    e8 c2 02 00 00        call   372 <write>
```

- Note, different versions of gcc
  - and different optimization levels
- Will generate slightly different code

Example

- From cat.asm
- `if (write(1, buf, n) != n)`

```
a0:   89 5c 24 08              mov    %ebx,0x8(%esp)
a4:   c7 44 24 04 00 0b 00     movl   $0xb00,0x4(%esp)
ab:   00
ac:   c7 04 24 01 00 00 00     movl   $0x1,(%esp)
b3:   e8 aa 02 00 00           call   362 <write>
```

Example

- From cat.asm
- `if (write(1, buf, n) != n)`

```
a0:   89 5c 24 08              mov    %ebx,0x8(%esp)
a4:   c7 44 24 04 00 0b 00     movl   $0xb00,0x4(%esp)
ab:   00
ac:   c7 04 24 01 00 00 00     movl   $0x1,(%esp)
b3:   e8 aa 02 00 00           call   362 <write>
```

Example

- From cat.asm
- `if (write(1, buf, n) != n)`

```
a0:   89 5c 24 08              mov    %ebx,0x8(%esp)
a4:   c7 44 24 04 00 0b 00     movl   $0xb00,0x4(%esp)
ab:   00
ac:   c7 04 24 01 00 00 00     movl   $0x1,(%esp)
b3:   e8 aa 02 00 00           call   362 <write>
```

- Still not clear...
  - The header file allows compiler to generate a call side invocation,
    - e.g., push arguments on the stack
  - But where is the system call invocation itself
    - e.g., `int $T_SYSCALL`

usys.S

```
8450 #include "syscall.h"
8451 #include "traps.h"
8452
8453 #define SYSCALL(name) \
8454   .globl name; \
8455   name: \
8456     movl $SYS_ ## name, %eax; \
8457     int $T_SYSCALL; \
8458     ret
8459
8460 SYSCALL(fork)
8461 SYSCALL(exit)
8462 SYSCALL(wait)
8463 SYSCALL(pipe)
8464 SYSCALL(read)
```

- Xv6 uses a SYSCALL macro to define a function for each system call invocation
  - E.g., fork() to invoke the “fork” system call

Example

- Write system call from cat.asm

```
00000362 <write>:

SYSCALL(write)

 362:   b8 10 00 00 00          mov    $0x10,%eax

 367:   cd 40                   int    $0x40

 369:   c3                      ret
```

System call arguments

- Where are the system call arguments?
- How does kernel access them?
  - And returns results?

Example

- Write system call
- `if (write(1, buf, n) != n)`

```
5876 int
5877 sys_write(void)
5878 {
5879   struct file *f;
5880   int n;
5881   char *p;
5882
5883   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5884     return -1;
5885   return filewrite(f, p, n);
5886 }
```

Example

- Write system call
- `if (write(1, buf, n) != n)`

```
5876 int
5877 sys_write(void)
5878 {
5879   struct file *f;
5880   int n;
5881   char *p;
5882
5883   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5884     return -1;
5885   return filewrite(f, p, n);
5886 }
```

```
3543 // Fetch the nth 32-bit system call argument.
3544 int
3545 argint(int n, int *ip)
3546 {
3547     return fetchint(proc->tf->esp + 4 + 4*n, ip);
3548 }
```

```
3515 // Fetch the int at addr from the current process.
3516 int
3517 fetchint(uint addr, int *ip)
3518 {
3519     if(addr >= proc->sz || addr+4 > proc->sz)
3520         return -1;
3521     *ip = *(int*)(addr);
3522     return 0;
3523 }
```

argint(int n, int *ip)

```
3543 // Fetch the nth 32-bit system call argument.
3544 int
3545 argint(int n, int *ip)
3546 {
3547     return fetchint(proc->tf->esp + 4 + 4*n, ip);
3548 }
```

```
3515 // Fetch the int at addr from the current process.
3516 int
3517 fetchint(uint addr, int *ip)
3518 {
3519     if(addr >= proc->sz || addr+4 > proc->sz)
3520         return -1;
3521     *ip = *(int*)(addr);
3522     return 0;
3523 }
```

argint(int n, int *ip)

```
3543 // Fetch the nth 32-bit system call argument.
3544 int
3545 argint(int n, int *ip)
3546 {
3547     return fetchint(proc->tf->esp + 4 + 4*n, ip);
3548 }
```

```
3515 // Fetch the int at addr from the current process.
3516 int
3517 fetchint(uint addr, int *ip)
3518 {
3519     if(addr >= proc->sz || addr+4 > proc->sz)
3520         return -1;
3521     *ip = *(int*)(addr);
3522     return 0;
3523 }
```

- Start with the address where current user stack is (esp)

argint(int n, int *ip)

```
3543 // Fetch the nth 32-bit system call argument.
3544 int
3545 argint(int n, int *ip)
3546 {
3547     return fetchint(proc->tf->esp + 4 + 4*n, ip);
3548 }
```

- Skip return eip

```
3515 // Fetch the int at addr from the current process.
3516 int
3517 fetchint(uint addr, int *ip)
3518 {
3519     if(addr >= proc->sz || addr+4 > proc->sz)
3520         return -1;
3521     *ip = *(int*)(addr);
3522     return 0;
3523 }
```

argint(int n, int *ip)

```
3543 // Fetch the nth 32-bit system call argument.
3544 int
3545 argint(int n, int *ip)
3546 {
3547     return fetchint(proc->tf->esp + 4 + 4*n, ip);
3548 }

3515 // Fetch the int at addr from the current process.
3516 int
3517 fetchint(uint addr, int *ip)
3518 {
3519     if(addr >= proc->sz || addr+4 > proc->sz)
3520         return -1;
3521     *ip = *(int*)(addr);
3522     return 0;
3523 }
```

- Fetch n'th argument

argint(int n, int *ip)

```
3543 // Fetch the nth 32-bit system call argument.
3544 int
3545 argint(int n, int *ip)
3546 {
3547     return fetchint(proc->tf->esp + 4 + 4*n, ip);
3548 }
```

```
3515 // Fetch the int at addr from the current process.
3516 int
3517 fetchint(uint addr, int *ip)
3518 {
3519     if(addr >= proc->sz || addr+4 > proc->sz)
3520         return -1;
3521     *ip = *(int*)(addr);
3522     return 0;
3523 }
```

fetchint(uint addr, int *ip)

```
3543 // Fetch the nth 32-bit system call argument.
3544 int
3545 argint(int n, int *ip)
3546 {
3547     return fetchint(proc->tf->esp + 4 + 4*n, ip);
3548 }
```

```
3515 // Fetch the int at addr from the current process.
3516 int
3517 fetchint(uint addr, int *ip)
3518 {
3519     if(addr >= proc->sz || addr+4 > proc->sz)
3520         return -1;
3521     *ip = *(int*)(addr);
3522     return 0;
3523 }
```

fetchint(uint addr, int *ip)

Any idea for what argptr() shall do?

- Write system call
- `if (write(1, buf, n) != n)`

```
5876 int
5877 sys_write(void)
5878 {
5879   struct file *f;
5880   int n;
5881   char *p;
5882
5883   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
5884     return -1;
5885   return filewrite(f, p, n);
5886 }
```

- Remember, buf is a pointer to a region of memory
  - i.e., a buffer
- of size n

```
3550 // Fetch the nth word-sized system call argument as a pointer
3551 // to a block of memory of size n bytes. Check that the pointer
3552 // lies within the process address space.
3553 int
3554 argptr(int n, char **pp, int size)
3555 {
3556   int i;
3557
3558   if(argint(n, &i) < 0)
3559     return -1;
3560   if((uint)i >= proc->sz || (uint)i+size > proc->sz)
3561     return -1;
3562   *pp = (char*)i;
3563   return 0;
3564 }
```

- Check that the pointer to the buffer is sound

argptr(uint addr, int *ip)

```
3550 // Fetch the nth word-sized system call argument as a pointer
3551 // to a block of memory of size n bytes. Check that the pointer
3552 // lies within the process address space.
3553 int
3554 argptr(int n, char **pp, int size)
3555 {
3556   int i;
3557
3558   if(argint(n, &i) < 0)
3559     return -1;
3560   if((uint)i >= proc->sz || (uint)i+size > proc->sz)
3561     return -1;
3562   *pp = (char*)i;
3563   return 0;
3564 }
```

- Check that the buffer is in user memory

argptr(uint addr, int *ip)

Summary

- We've learned how system calls work

Thank you

sys_exec()

```
6225 sys_exec(void)
6226 {
6227   char *path, *argv[MAXARG];
6228   int i;
6229   uint uargv, uarg;
6230
6231   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
6232     return -1;
6233   }
6234   memset(argv, 0, sizeof(argv));
6235   for(i=0;; i++){
6236     if(i >= NELEM(argv))
6237       return -1;
6238     if(fetchint(uargv+4*i, (int*)&uarg) < 0)
6239       return -1;
6240     if(uarg == 0){
6241       argv[i] = 0;
6242       break;
6243     }
6244     if(fetchstr(uarg, &argv[i]) < 0)
6245       return -1;
6246   }
6247   return exec(path, argv);
6248 }
```

```
6225 sys_exec(void)
6226 {
6227   char *path, *argv[MAXARG];
6228   int i;
6229   uint uargv, uarg;
6230
6231   if(argstr(0, &path) < 0
         || argint(1, (int*)&uargv) < 0){
6232     return -1;
6233   }
...
6247   return exec(path, argv);
6248 }
```

sys_exec()