Lecture 6: System Calls

Professor G. Sandoval

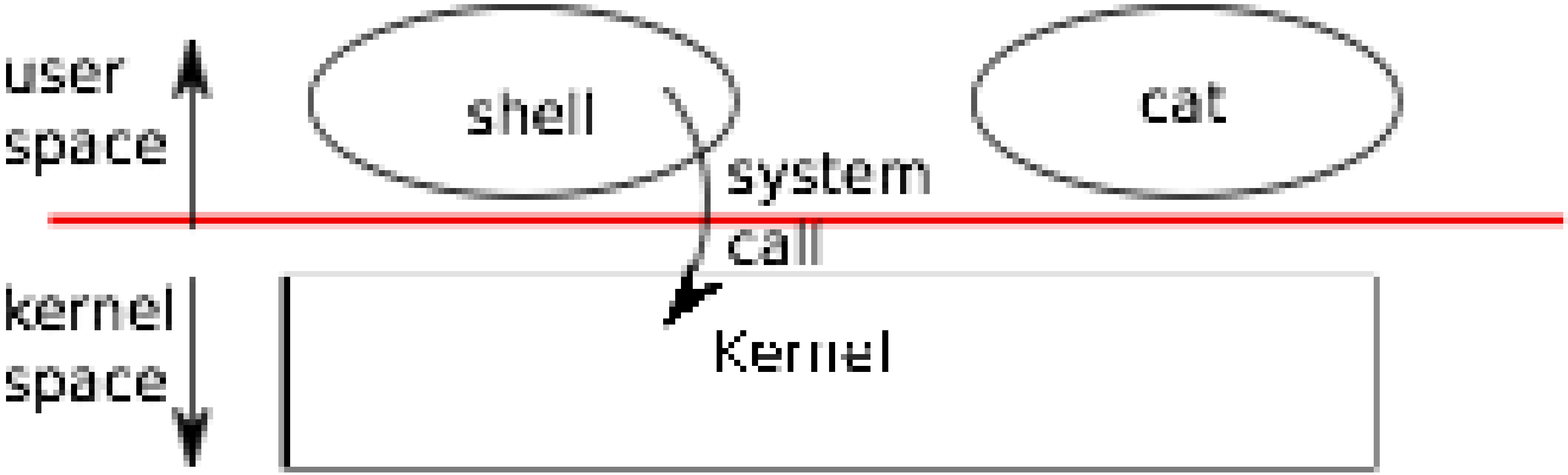
Today

- System Calls
- Int Instruction
- XV6 System Calls Implementation
- Adding our own System Call

System Call Interface

- Once we are running a process in user mode that process must call into the kernel to
- We do this using a *system call*
- A system call is like a normal function call, but goes from low-privilege to high-privilege (user->kernel)

Making a System Call



System call	Description
fork()	Create process
exit()	Terminate current process
wait()	Wait for a child process to exit
kill(pid)	Terminate process pid
getpid()	Return current process's id
sleep(n)	Sleep for n seconds
exec(filename, *argv)	Load a file and execute it
sbrk(n)	Grow process's memory by n bytes
open(filename, flags)	Open a file; flags indicate read/write
read(fd, buf, n)	Read n byes from an open file into buf
write(fd, buf, n)	Write n bytes to an open file
close(fd)	Release open file fd
dup(fd)	Duplicate fd
pipe(p)	Create a pipe and return fd's in p
chdir(dirname)	Change the current directory
mkdir(dirname)	Create a new directory
mknod(name, major, minor)	Create a device file
fstat(fd)	Return info about an open file
link(f1, f2)	Create another name (f2) for the file f1
unlink(filename)	Remove a file

mknod(name,major,minor)

- We already discussed the "everything's a file" concept
- So, we know that some files can refer to things other than bytes on a filesystem
- mknod allows you to create such pseudo-files
- The *major* and *minor* numbers tell the kernel how to interpret reads/writes to the file

Example device

Major Minor

crw-rw-rw- 1 root wheel   3,   3 Sep 7 12:12 /dev/zero

mknod("/dev/zero", 3, 3)

link(file1,file2)

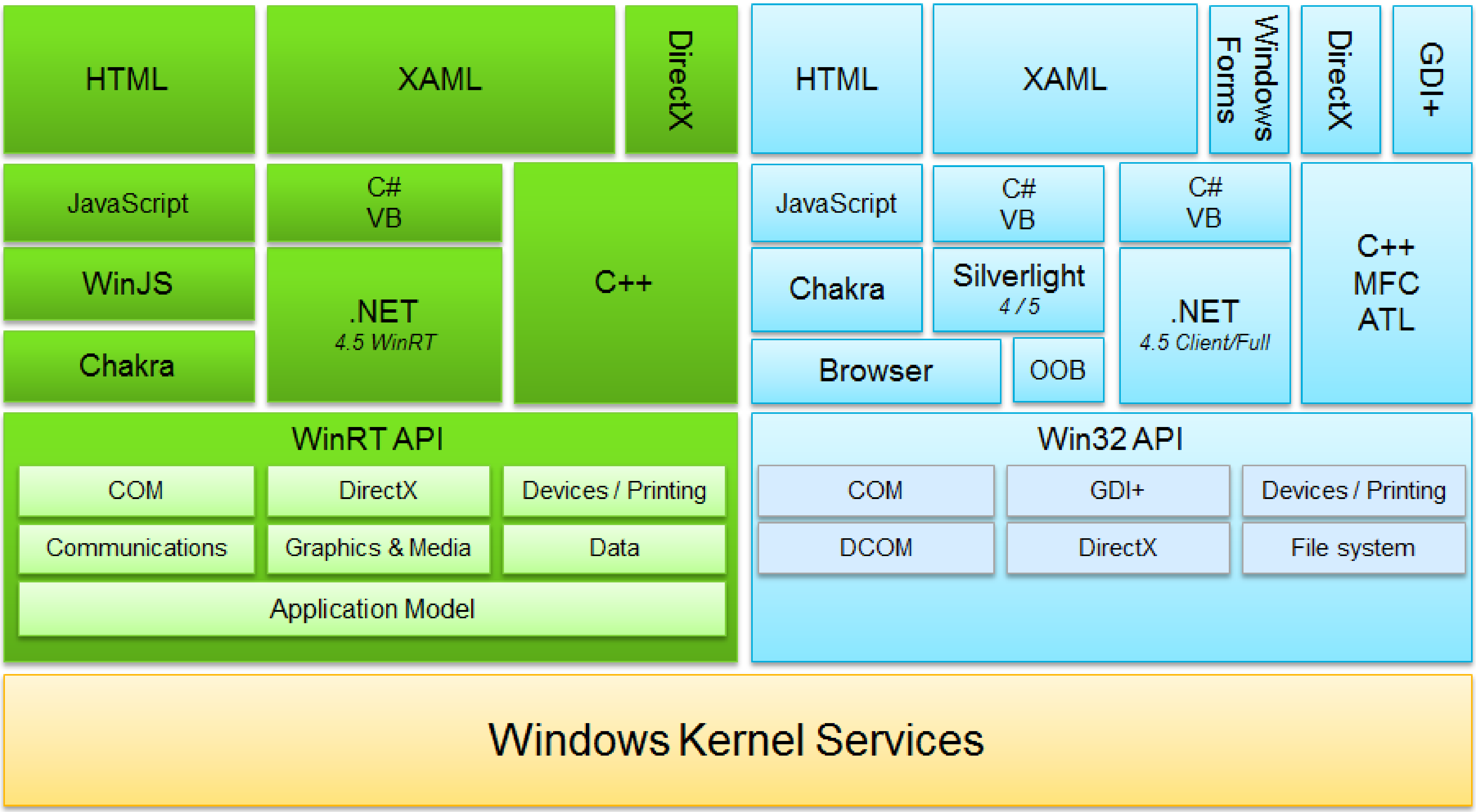
- Creates another name for file1
- Similar to how two *file descriptors* can refer to a single *file*
- Multiple names for a single file are *persistent* and implemented at the *filesystem* level

Real-World System Calls

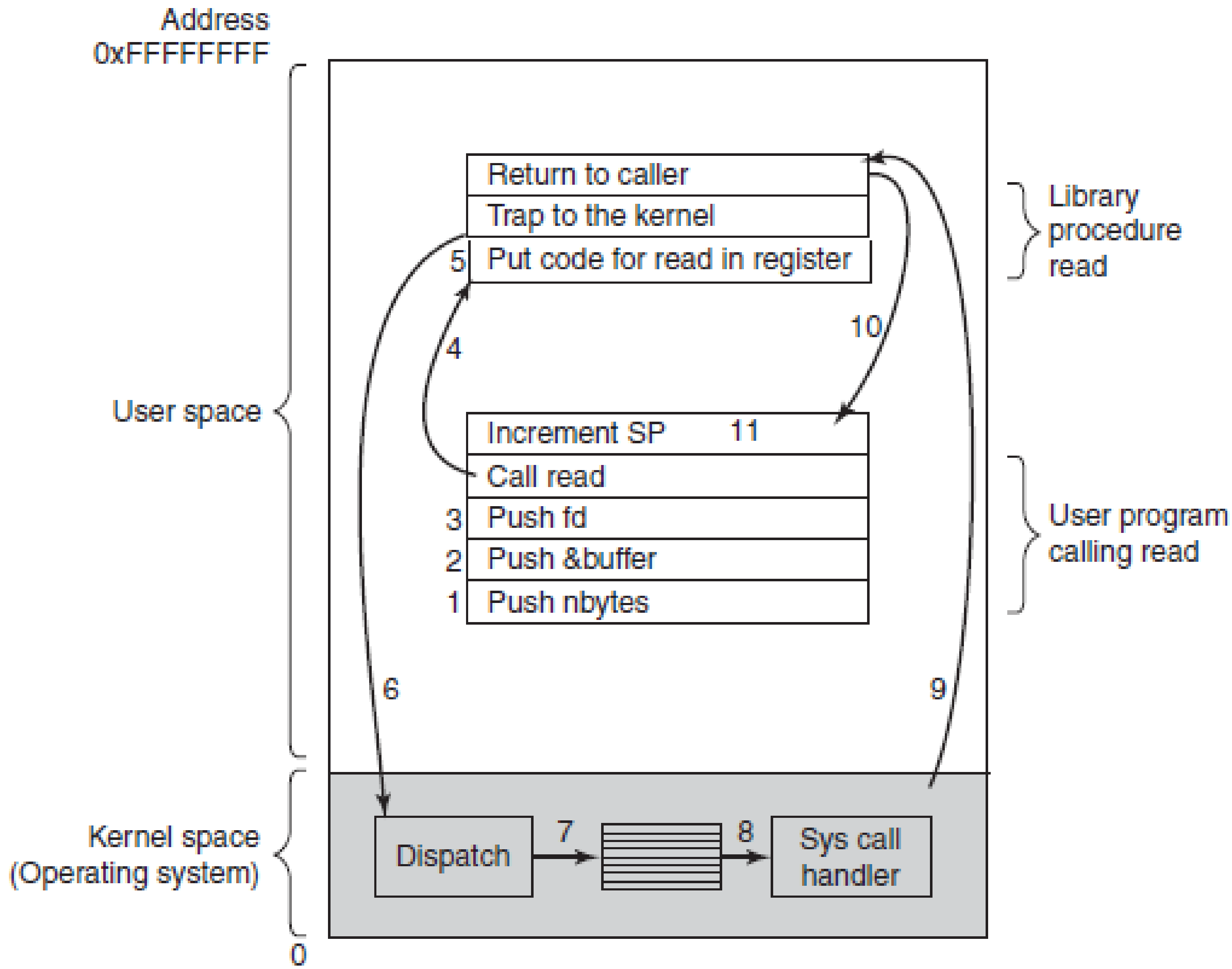
- Linux is a UNIX-like operating system, so the concepts are similar
- But it has many more calls – 389 as of kernel version 3.19
- Interface is *public* – user programs can use it directly
- Many more than xv6!

Real-World System Calls

- Windows has even more – **1,468** as of Windows 8
- The majority of these (1,036) are GUI-related
- The system call interface is *private*
- All calls go through a (much larger) set of user-space libraries that are stable (the Win32 API)


HTML
XAML
DirectX
JavaScript
C#
VB
WinJS
.NET
4.5 WinRT
C++
Chakra
WinRT API
COM
DirectX
Devices / Printing
Communications
Graphics & Media
Data
Application Model
HTML
XAML
Windows Forms
DirectX
GDI+
JavaScript
C#
VB
C#
VB
C++
MFC
ATL
Chakra
Silverlight
4 / 5
.NET
4.5 Client/Full
Browser
OOB
Win32 API
COM
GDI+
Devices / Printing
DCOM
DirectX
File system
Windows Kernel Services

Recall: System Calls



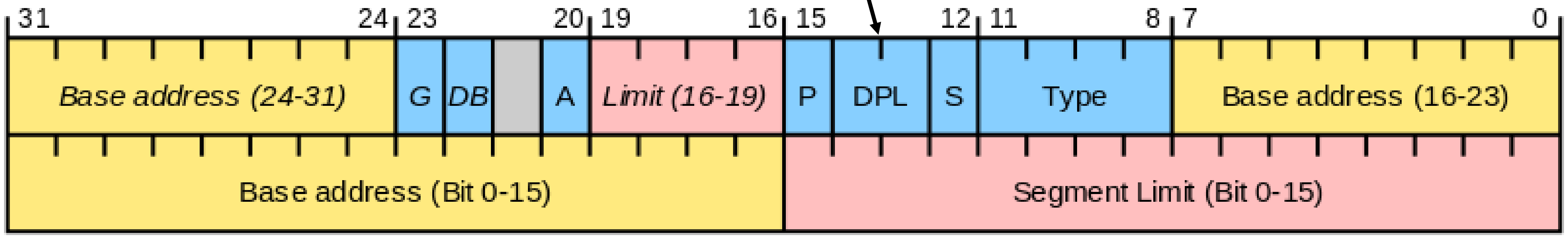
The 11 steps in making the system call
read(fd, buffer, nbytes).

System Call Mechanism

- To implement a system call, we need to do several things:
  - Have an instruction that initiates the call
  - Specify *which* call we want
  - Have the kernel retrieve the system call arguments
  - Save the process's current state and restore it when we return
  - Do all this securely, without breaking isolation between user and kernel space

Reminder: Segment Descriptors

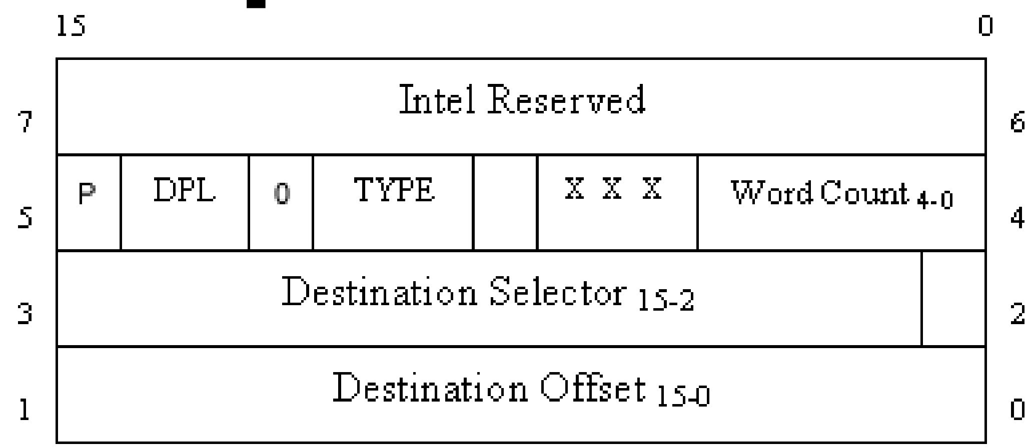
DPL – *privilege level* of the segment



The Interrupt Descriptor Table

- System calls, interrupts and exceptions are handled using the same mechanism
- Each one is handled by the processor by consulting the *interrupt descriptor table*
- The interrupt descriptor table contains 8-byte entries (similar to the GDT) that describe what to do for each interrupt number

Interrupt Descriptor



- SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, syscall_handler, DPL_USER);

```
(gate).off_15_0 = syscall_handler & 0xffff;
(gate).cs = SEG_KCODE << 3;
(gate).args = 0;
(gate).rsv1 = 0;
(gate).type = STS_TG32;
(gate).s = 0;
(gate).dpl = DPL_USER;
(gate).p = 1;
gate).off_31_16 = syscall_handler >> 16;
```

Today

- System Calls
- Int Instruction
- XV6 System Calls Implementation
- Adding our own System Call

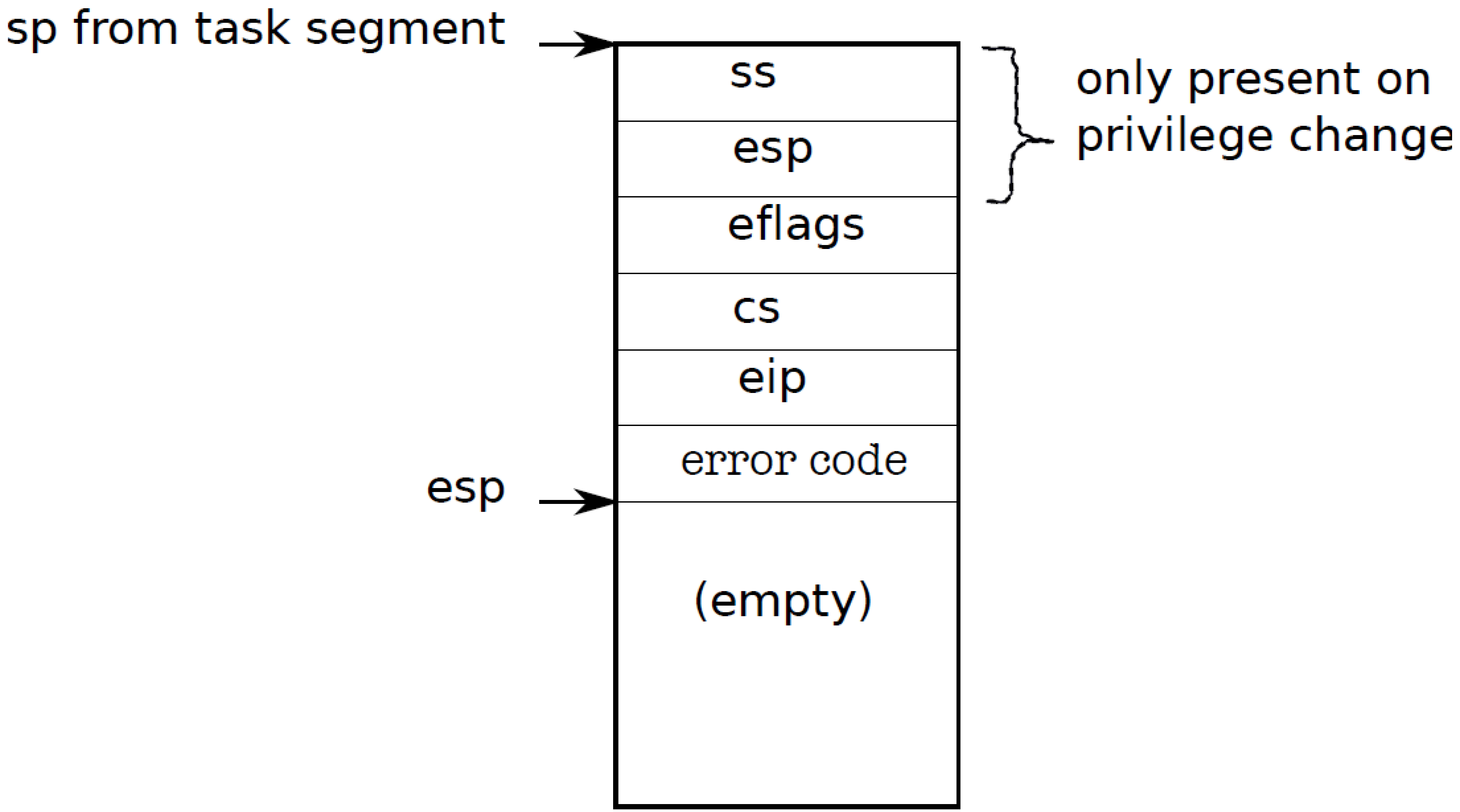
The int Instruction

- "int n" does a lot of work
  - Fetch the nth descriptor from the IDT, where n is the argument of int.
  - Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
  - Save %esp and %ss in a CPU-internal registers, but only if the target segment selector’s PL < CPL.
  - Load %ss and %esp from a task segment descriptor.

The int Instruction

- Push %ss.
- Push %esp.
- Push %eflags.
- Push %cs.
- Push %eip.
- Clear some bits of %eflags.
- Set %cs and %eip to the values in the descriptor.

Stack after int Instruction



Kernel stack after an int instruction.

Why So Complicated?

- Mostly to ensure *protection*
  - CPL check prohibits anything but a user process from making any other interrupts
  - Kernel reads some information off the stack; arbitrary user stacks might be incorrect (point to nonexistent memory) or malicious

Today

- System Calls
- Int Instruction
- XV6 System Calls Implementation
- Adding our own System Call

initcode.S

- Last time we saw the system's first system call

```
# exec(init, argv)
.globl start
start:
  pushl $argv
  pushl $init
  pushl $0  // where caller pc would be
  movl $SYS_exec, %eax
  int $T_SYSCALL
[...]
# char init[] = "/init\0";
init:
  .string "/init\0"
```

Trap Frames

- Once we are in the kernel, we save more registers (alltraps, trapasm.S)

```
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal
```

Pushes all general-purpose registers onto the stack

- Together with the registers the CPU saved for us, this makes up a *trap frame*

Trap Frame

```
// Layout of the trap frame built on the stack by the
// hardware and by trapasm.S, and passed to trap().
struct trapframe {
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort gs;
  ushort padding1;
  ushort fs;
  ushort padding2;
  ushort es;
  ushort padding3;
  ushort ds;
  ushort padding4;
  uint trapno;
...
```

Trap Frame

- By capturing all of this information in the trap frame structure, we can restore the CPU state exactly when we return from the system call

Trap Handler

- After we've set up our trap frame, we call the trap() function:

```
# Call trap(tf), where tf=%esp
pushl %esp
call trap
addl $4, %esp
```

Trap Handler – trap.c

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(proc->killed)
      exit();
    proc->tf = tf;
    syscall();
    if(proc->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpu->id == 0){
      acquire(&tickslock);
      ticks++;
             [...]
```

Inside the System Call

- After setting up the trap frame, we call the syscall() function
- syscall() examines %eax in the trap frame to find out what system call to execute
- Inside the system call, things look exactly the same as a normal function call
  - But we have extra privileges since we are in kernel mode; e.g. we can talk to hardware directly

kill(pid)

```
void
syscall(void)
{
  int num;

  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    proc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            proc->pid, proc->name, num);
    proc->tf->eax = -1;
  }
}

int
sys_kill(void)
{
  int pid;

  if(argint(0, &pid) < 0)
    return -1;
  return kill(pid);
}
```

Retrieving Arguments

- We changed stacks after leaving user mode
- But arguments are still stored on the user stack!
- Solution: use the %esp value saved in the trap frame
- Arguments are at %esp+4+(4*arg_no)

Retrieving Arguments

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

int
fetchint(uint addr, int *ip)
{
  if(addr >= proc->sz || addr+4 > proc->sz)
    return -1;
  *ip = *(int*)(addr);
  return 0;
}
```

Protection

- Note the check in fetchint to make sure the pointer is not outside proc->sz
- In user mode, we can rely on the paging hardware to disallow access to anything outside of process's memory
- But in kernel-mode we must do explicit checks, because the kernel has access to all memory
- Similar checks for getting a pointer (argptr) and getting a string (argstr)

Returning to Userspace

- syscall() put the return value in proc->tf->eax
- So when we get back to userspace, we will have our return value in %eax

```
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

Today

- System Calls
- Int Instruction
- XV6 System Calls Implementation
- Adding our own System Call

Adding a System Call

- From this, it should be easy now to add a system call
- We don't have to alter the mechanism, just extend the list and implement our function

Hello World

- We will add a system call that just prints hello world an a user-provided number

- First we need to add our call to the list in syscall.c

```
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_hello(void);
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
...
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_hello]   sys_hello,
};
```

Hello World

- Next, assign it a number in syscall.h:

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

- And give it a prototype in user.h:

```
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int hello(int);
```

usys.S

- Add it to usys.S, which generates the user-space assembly code for it

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

SYSCALL(fork)
SYSCALL(exit)
...
SYSCALL(uptime)
SYSCALL(hello)
```

Implementation

- Finally we add the implementation somewhere (e.g. sysproc.c)

```c
int
sys_hello(void) {
    int n;
    if(argint(0, &n) < 0)
        return -1;
    cprintf("Hello world %d\n", n);
    return 0;
}
```

Testing Our New Call

```c
#include "types.h"
#include "stat.h"
#include "user.h"

int main(void) {
    hello(5);
    exit();
}
```

Quick Demo

Real-World: A Newer Mechanism

- Older versions of Linux and Windows did use interrupts for system calls
- Nowadays most systems don't use them
  - Interrupts are very heavyweight, slow
- Instead we have sysenter and sysexit
  - These do much less work – mostly just switches to kernel mode, sets CS, ESP, and EIP
  - Up to the kernel to decide how much it wants to save from user mode