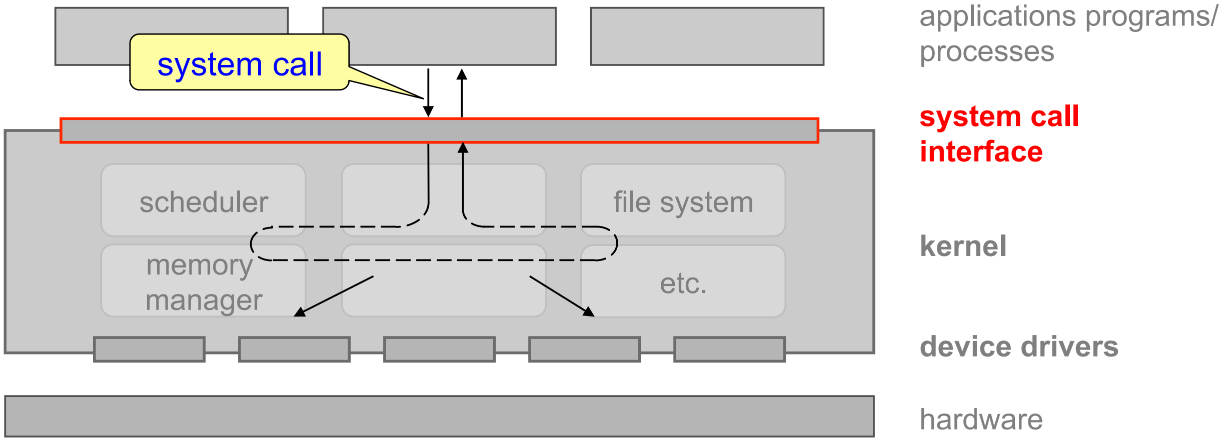
System Calls

1. System calls are the user API to the OS
2. System calls are not function calls!
3. How are system calls invoked?
4. Why are system calls implemented this way?

Operating System Interfaces: System Calls



Types of System Calls

Process Control

- load
- execute
- end, abort
- create process
- terminate process
- get/set process attributes
- wait for time, wait event, signal event
- allocate, free memory

File Management

- create file, delete file
- open, close
- read, write, reposition
- get/set file attributes

Device Management

- request device, release device
- read, write, reposition
- get/set device attributes
- logically attach or detach devices

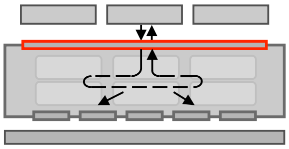
Communication

- create, delete communication connection
- send, receive messages
- transfer status information
- attach or detach remote devices

Information Maintenance

- get/set time or date
- get/set system data
- get/set process, file, or device attributes

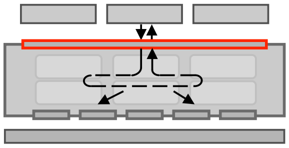
System Calls – They look like Functions ...



```
#include<unistd.h>

int main(void) {
  write(1, "Hello, world\n", 13);
  return 0;
}
```

... but they are not!



Making System Calls in 32-bit Linux

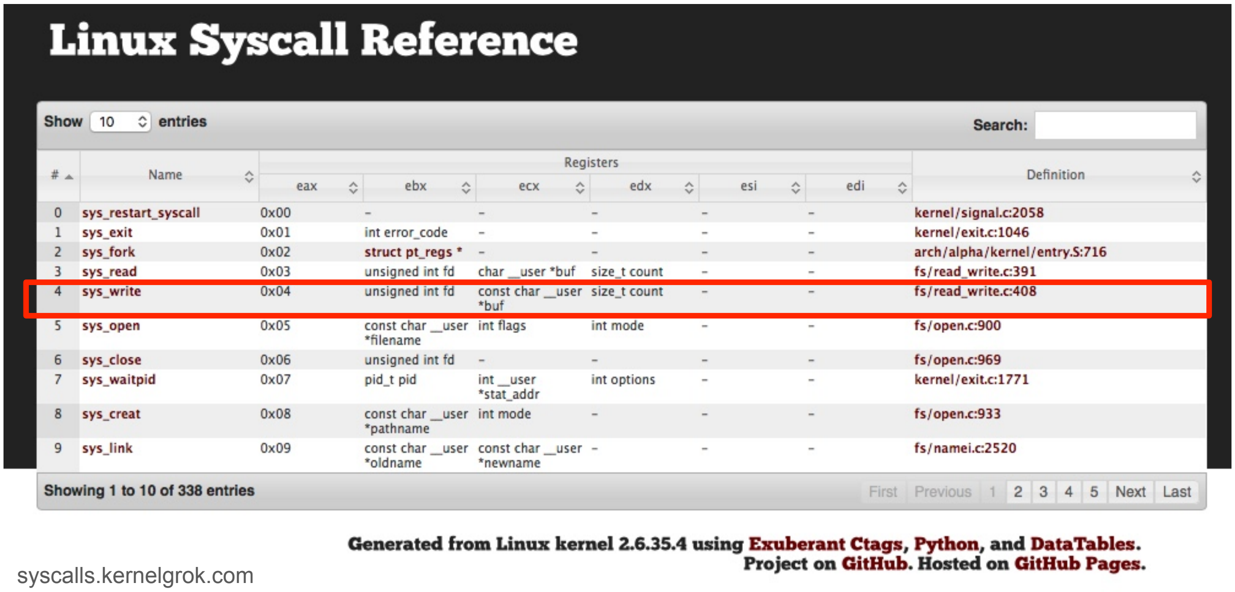
1. load system call number in register `eax`.
2. load arguments to system call in registers `ebx`, `exc`, `edx`, `esi`, `edi`, `ebp`
3. invoke software interrupt: `int 0x80`

Returned values are stored in `eax`.

64-bit Linux?

- same but different
- different registers
- different numbers
- `syscall` instruction

System Call Numbers



Linux Syscall Reference

Show 10 entries

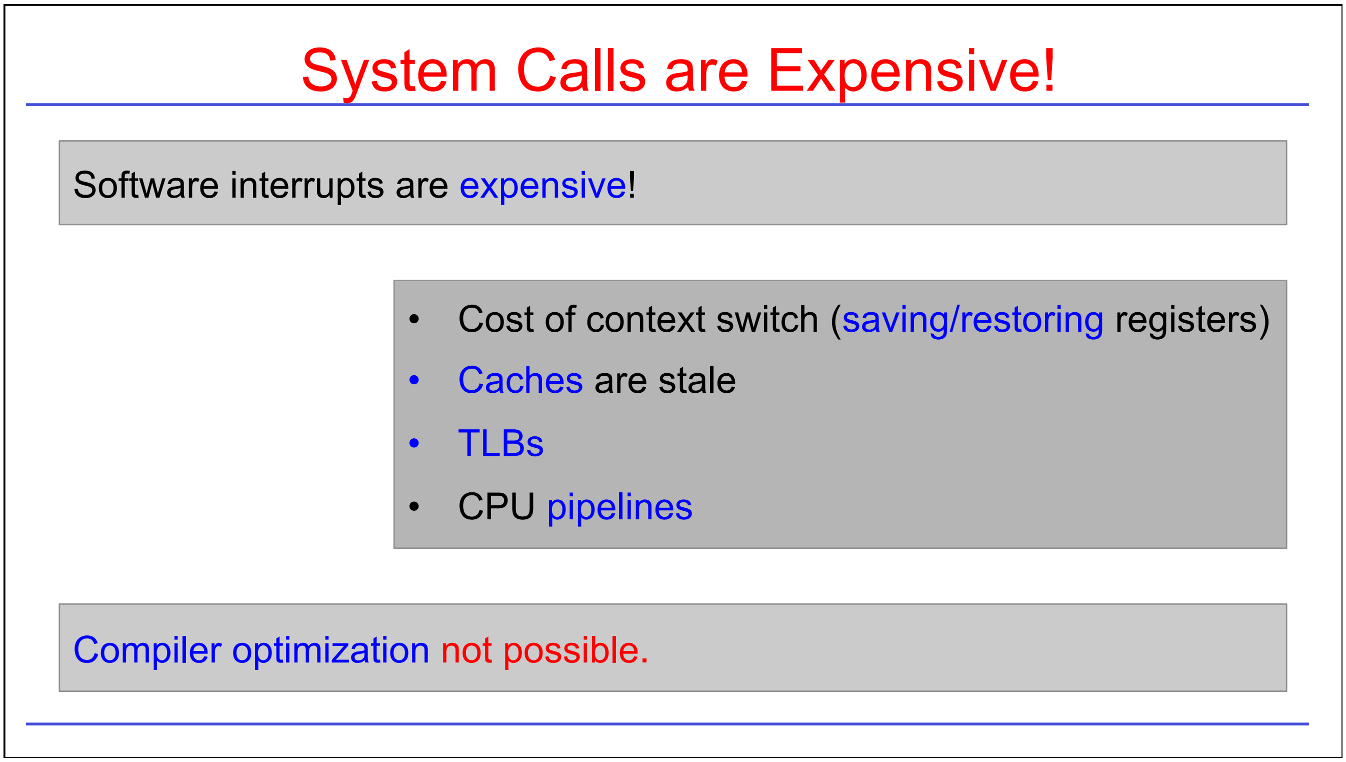
Search:

#	Name	Registers						Definition
		eax	ebx	ecx	edx	esi	edi	
0	sys_restart_syscall	0x00	-	-	-	-	-	kernel/signal.c:2058
1	sys_exit	0x01	int error_code	-	-	-	-	kernel/exit.c:1046
2	sys_fork	0x02	struct pt_regs *	-	-	-	-	arch/alpha/kernel/entry.S:716
3	sys_read	0x03	unsigned int fd	char __user *buf	size_t count	-	-	fs/read_write.c:391
4	sys_write	0x04	unsigned int fd	const char __user *buf	size_t count	-	-	fs/read_write.c:408
5	sys_open	0x05	const char __user *filename	int flags	int mode	-	-	fs/open.c:900
6	sys_close	0x06	unsigned int fd	-	-	-	-	fs/open.c:969
7	sys_waitpid	0x07	pid_t pid	int __user *stat_addr	int options	-	-	kernel/exit.c:1771
8	sys_creat	0x08	const char __user *pathname	int mode	-	-	-	fs/open.c:933
9	sys_link	0x09	const char __user *oldname	const char __user *newname	-	-	-	fs/namei.c:2520

Showing 1 to 10 of 338 entries

Generated from Linux kernel 2.6.35.4 using Exuberant Ctags, Python, and DataTables.
Project on GitHub. Hosted on GitHub Pages.

syscalls.kernelgrok.com

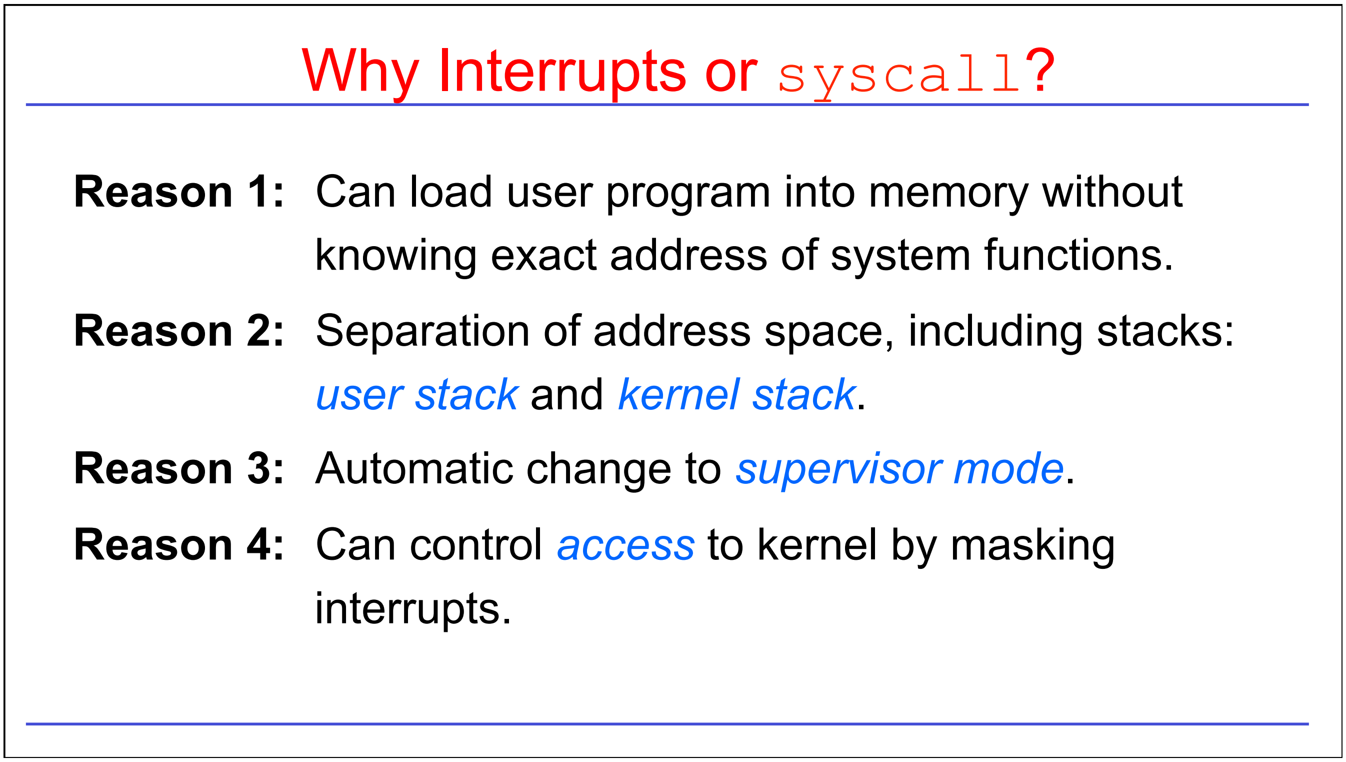


System Calls are Expensive!

Software interrupts are expensive!

- Cost of context switch (saving/restoring registers)
- Caches are stale
- TLBs
- CPU pipelines

Compiler optimization not possible.



Why Interrupts or `syscall`?

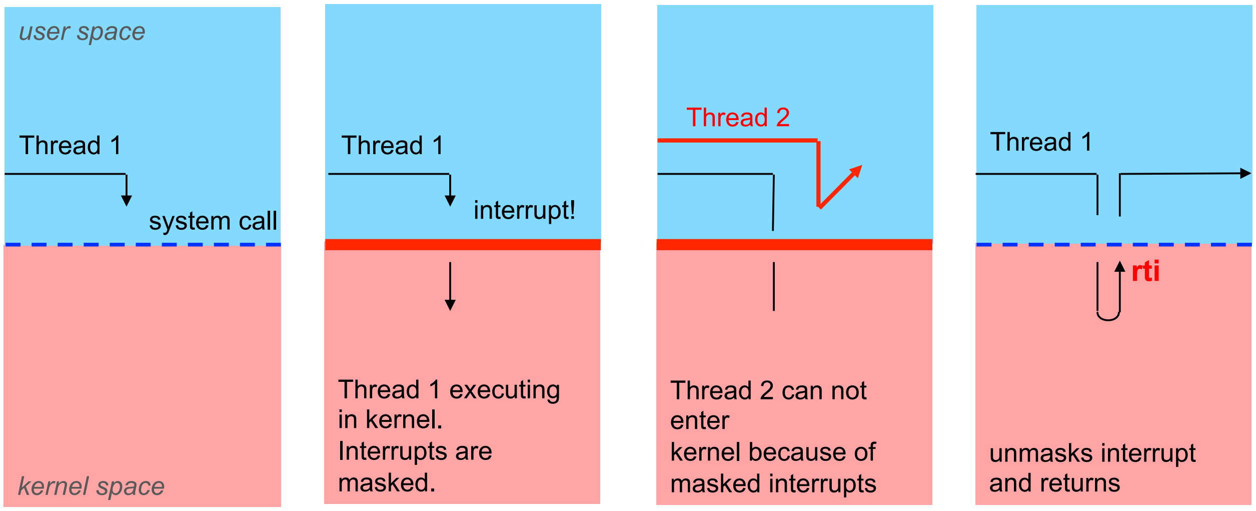
Reason 1: Can load user program into memory without knowing exact address of system functions.

Reason 2: Separation of address space, including stacks: *user stack* and *kernel stack*.

Reason 3: Automatic change to *supervisor mode*.

Reason 4: Can control *access* to kernel by masking interrupts.

Reason 4: Mutual Exclusion in Kernel



user space

Thread 1

system call

kernel space

Thread 1

interrupt!

Thread 1 executing in kernel. Interrupts are masked.

Thread 2

Thread 2 can not enter kernel because of masked interrupts

Thread 1

rti

unmasks interrupt and returns

Summary: System Calls

1. System calls are the OS API
2. They look like function calls...

   ... but they are not!
3. They are implemented using software interrupts (or variations thereof)
4. Why are they implemented this way?