Introduction to xv6

By Ghazal Sadeghian

What is xv6 ?

- simplified and lightweight OS based on Sixth Edition Unix
- It was designed in MIT for educational purposes
- theoretically actually boots on real 32-bit x86 hardware(and supports multicore)

  (but we'll run it only single-core, in qemu emulator)

- It has an easy-to-understand structure

How to run xv6?

XV6 can be installed as a standalone OS (not recommended)

XV6 can run in an emulated environment (recommended)

QEMU is the emulator we use to run XV6

(QEMU is a user program/system emulator)

xv6 technical requirements

- Linux environment
- Qemu simulator

Getting xv6 running

Step 1 – Install qemu: $ sudo apt install qemu

Step 2 – Install xv6: Create a directory, and clone xv6 to that directory:

$ git clone git://github.com/mit-pdos/xv6-public.git

Step 3 – Compile xv6 : In the newly created xv6 directory: $ make

Step 4 – Compile and run the emulator qemu: $ make qemu

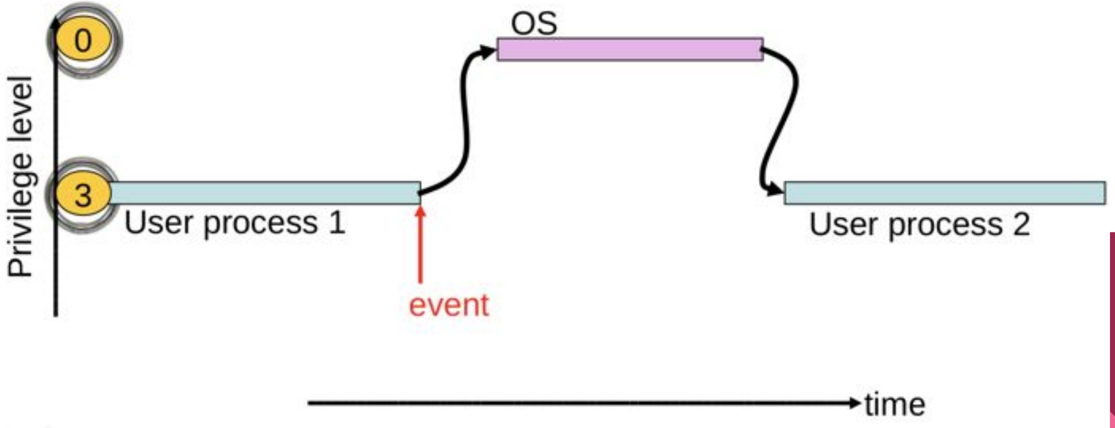
$ make qemu-nox

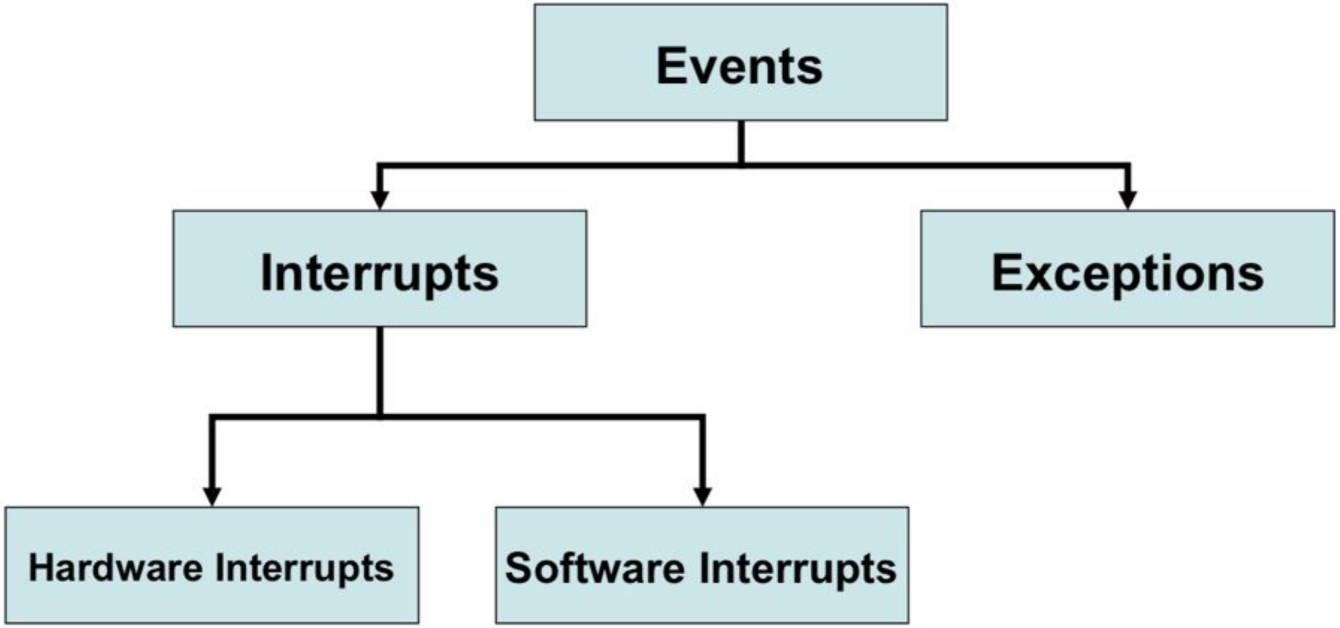
make qemu : runs the graphical version of qemu

make qemu-nox: runs the console version

Why event driven design?

- OS cannot trust user processes
  - User processes may be buggy or malicious
  - User process crash should not affect OS
- OS needs to guarantee fairness to all user processes
  - One process cannot 'hog' CPU time
  - Timer interrupts

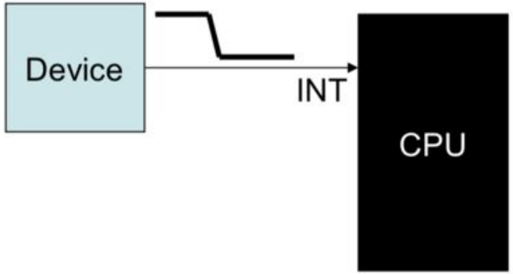




Events
Interrupts
Exceptions
Hardware Interrupts
Software Interrupts

Hardware vs Software Interrupt

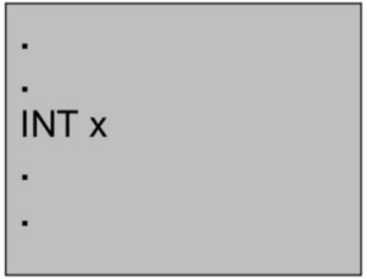
Hardware Interrupt



A device (like PIC)

asserts a pin in the CPU

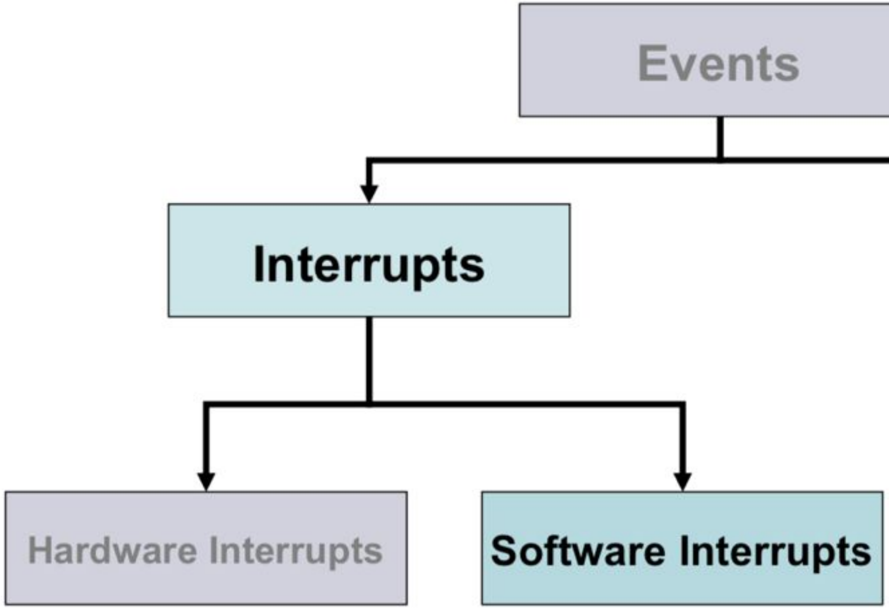
Software Interrupt



An instruction which when executed

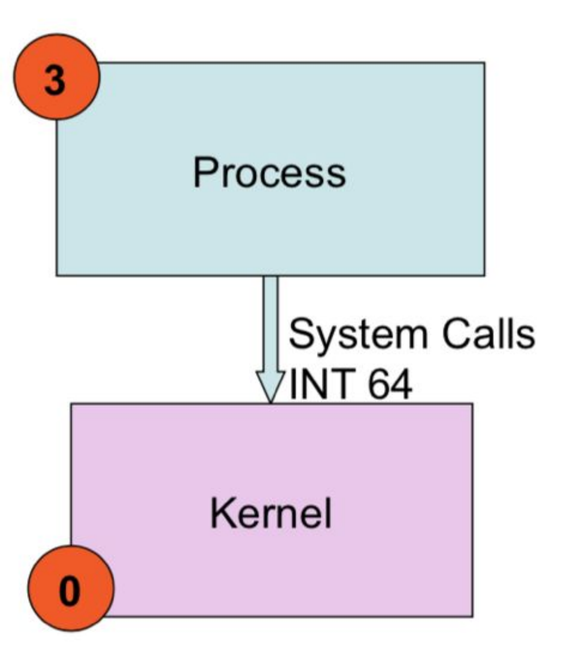
causes an interrupt

System Calls



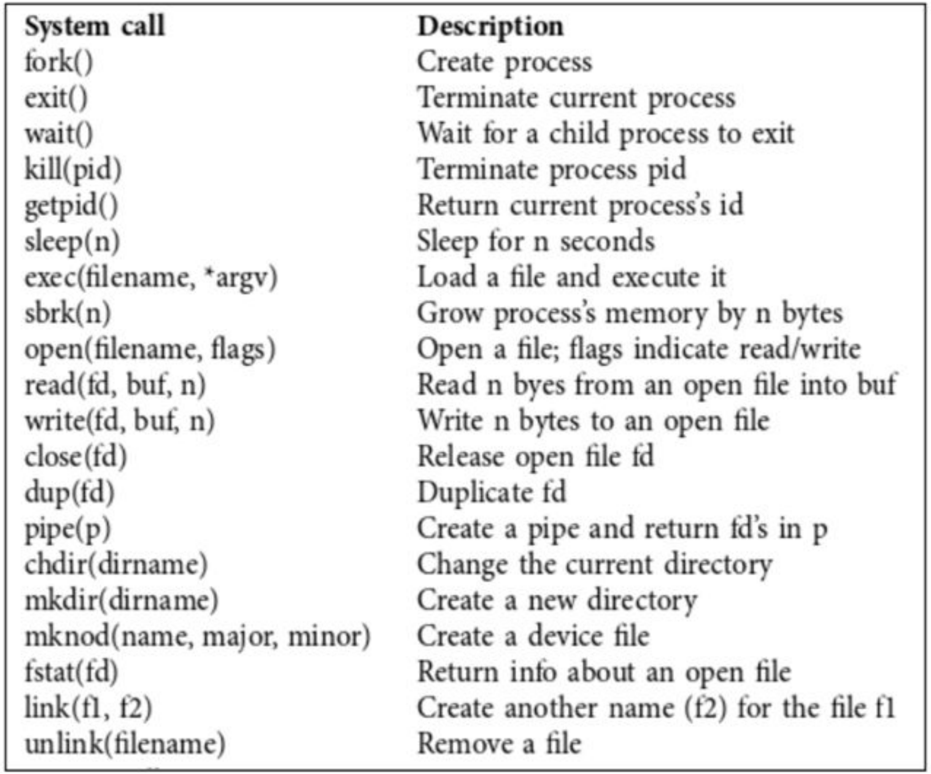
Software interrupt used for implementing system calls

- In Linux INT 128, is used for system calls
- In xv6, INT 64 is used for system calls



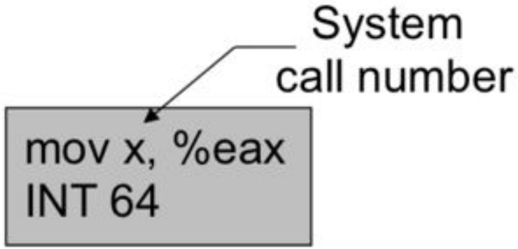
System calls in xv6

- fork, exec, exit, wait, kill, getpid — process control
- open, read, write, close, fstat, dup — file operations
- mknod, unlink, link, chdir — directory operations



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

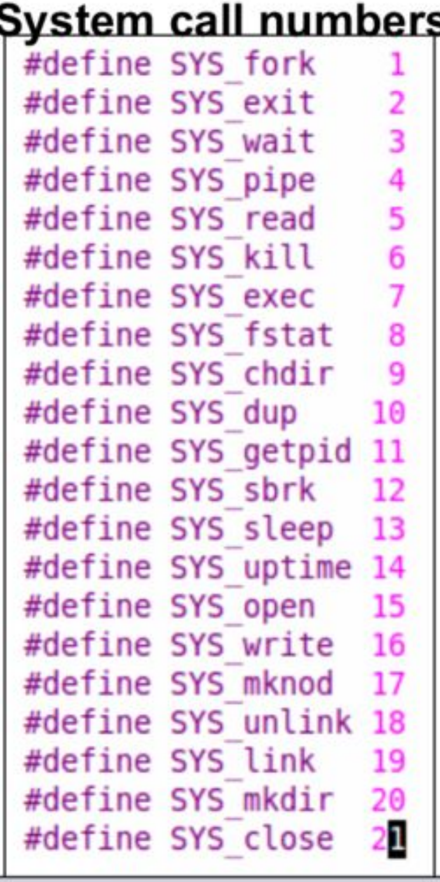
How does the OS distinguish between the system calls?



System call number

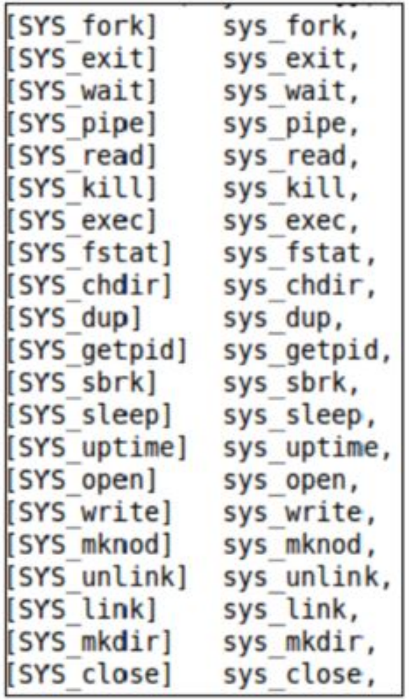
```
mov x, %eax
INT 64
```

System call numbers



```
#define SYS_fork     1
#define SYS_exit     2
#define SYS_wait     3
#define SYS_pipe     4
#define SYS_read     5
#define SYS_kill     6
#define SYS_exec     7
#define SYS_fstat    8
#define SYS_chdir    9
#define SYS_dup     10
#define SYS_getpid  11
#define SYS_sbrk    12
#define SYS_sleep   13
#define SYS_uptime  14
#define SYS_open    15
#define SYS_write   16
#define SYS_mknod   17
#define SYS_unlink  18
#define SYS_link    19
#define SYS_mkdir   20
#define SYS_close   21
```

System call handlers



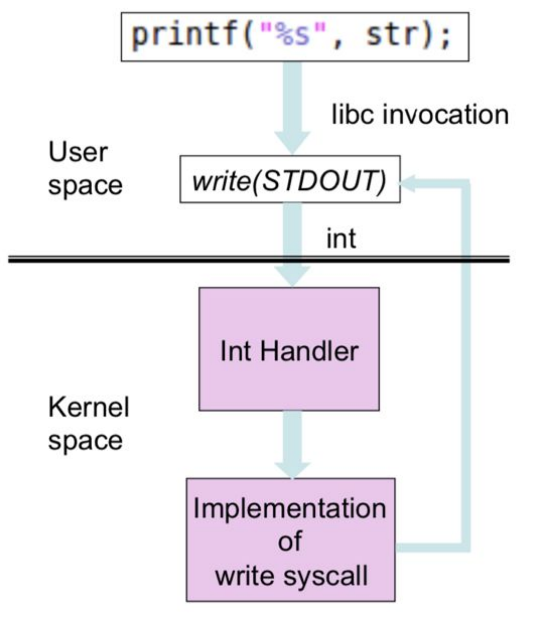
```
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
```

Based on the system call number function syscall invokes the corresponding syscall handler

ref : syscall.h, syscall() in syscall.c

SYSCALL([NAME]) in usys.S:

This line translates to the following assembly:

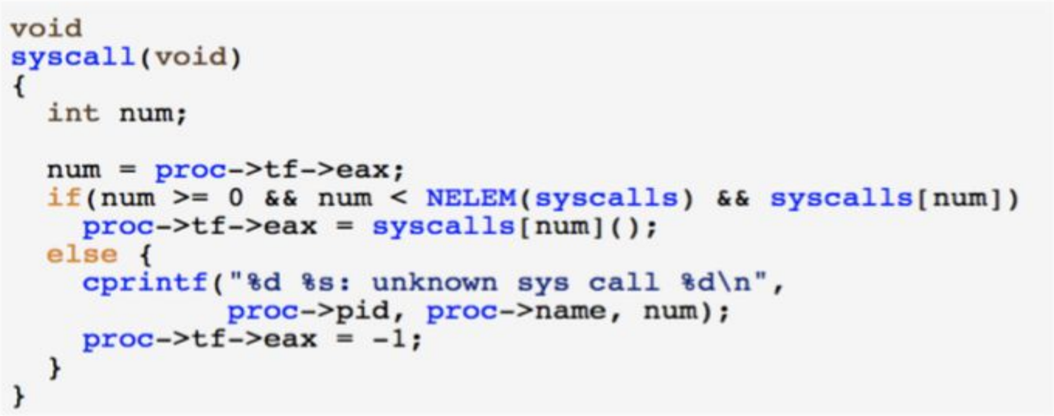


```
 .globl name;
name:
mov $SYS_name, %eax; //putting the system call number in eax
int $0x40; //calling the system call handler in interrupt mode(64)
```

Syscall(void)

All system calls are handled in this function.

The sys num which is saved in eax register is retrieved and system call is read from table.

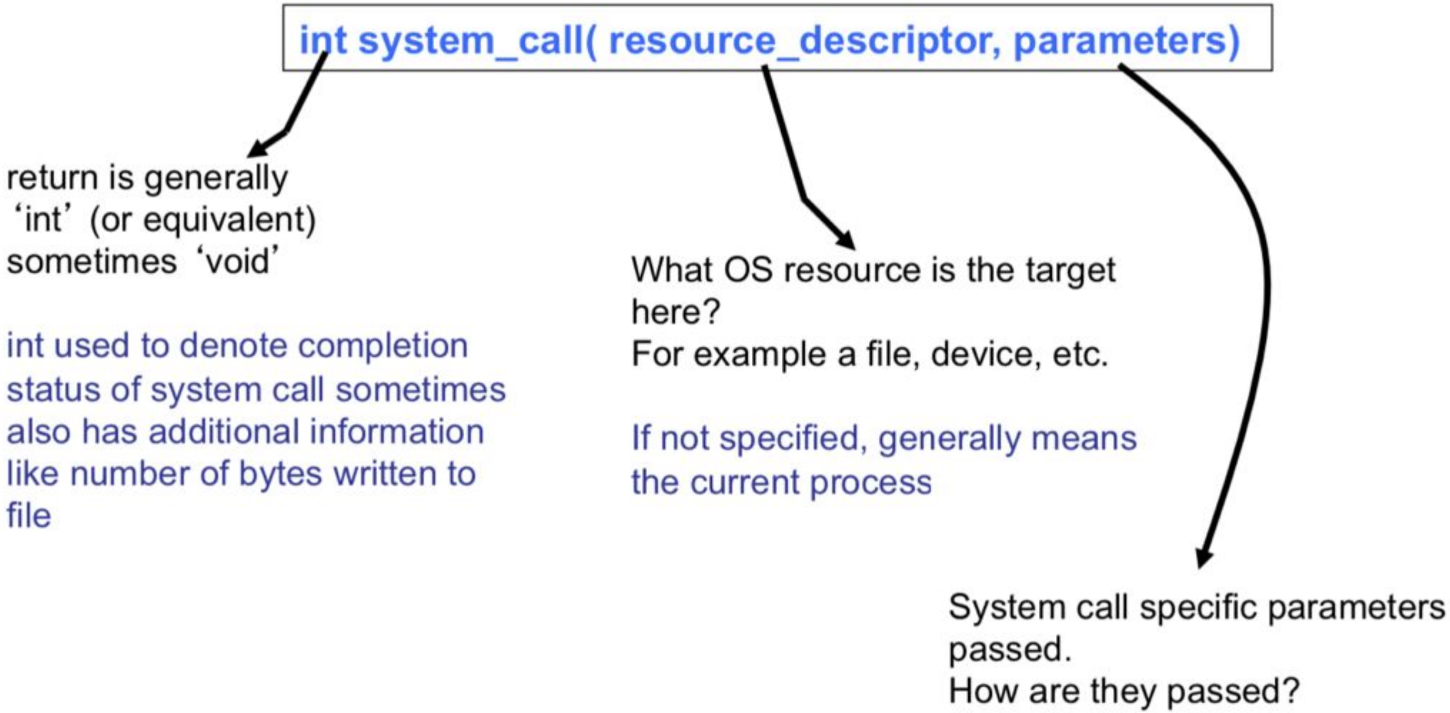


```
void
syscall(void)
{
  int num;

  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;
  }
}
```

ref : syscall.h, syscall() in syscall.c

Prototype of a Typical System Call



```
int system_call( resource_descriptor, parameters)
```

return is generally
'int' (or equivalent)
sometimes 'void'

int used to denote completion status of system call sometimes also has additional information like number of bytes written to file

What OS resource is the target here?
For example a file, device, etc.

If not specified, generally means the current process

System call specific parameters passed.
How are they passed?

Question 1

Fork()

- To create child process we use fork(). fork() returns :
    - **<0** fail to create child (new) process
    - **=0** Returned to the newly created child process
    - **>0** Returned to parent or caller. The value contains process ID of newly created child process..

pipe()

- pipe() is used for passing information from one process to another. pipe() is unidirectional

getChildren

- getppid
- getchildren(pid)

getCount

- number of times the referenced system call was invoked by the calling process