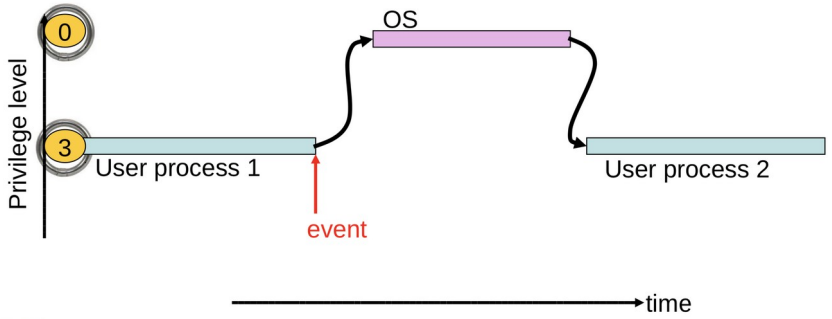
Lab Project 2: System calls and Processes

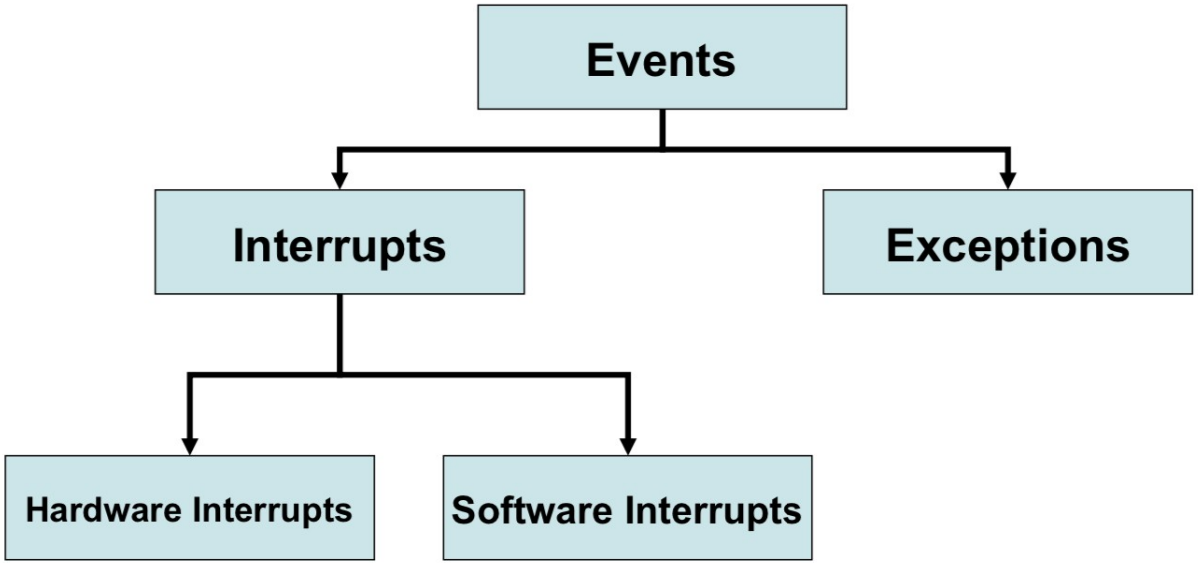
Sarina Hamedani
Ali Hajizadeh

Why Event Driven Design?

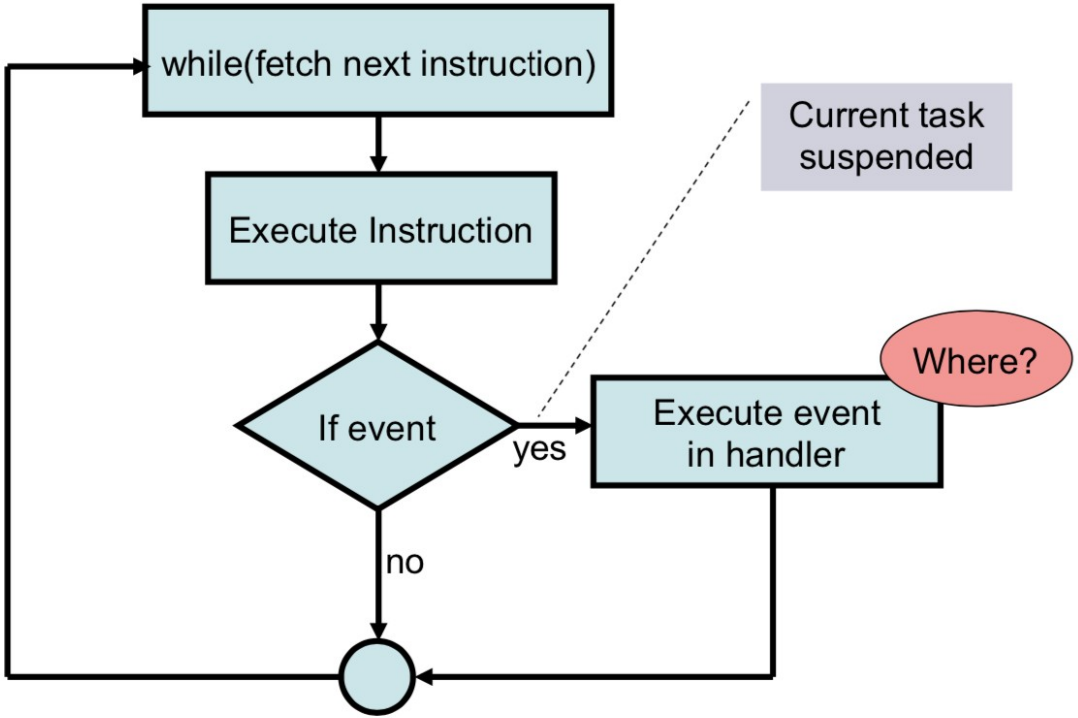
- OS cannot **trust** user processes
  - User processes may be buggy or malicious
  - User process crash should not affect OS
- OS needs to guarantee **fairness** to all user processes
  - One process cannot 'hog' CPU time
  - Timer interrupts



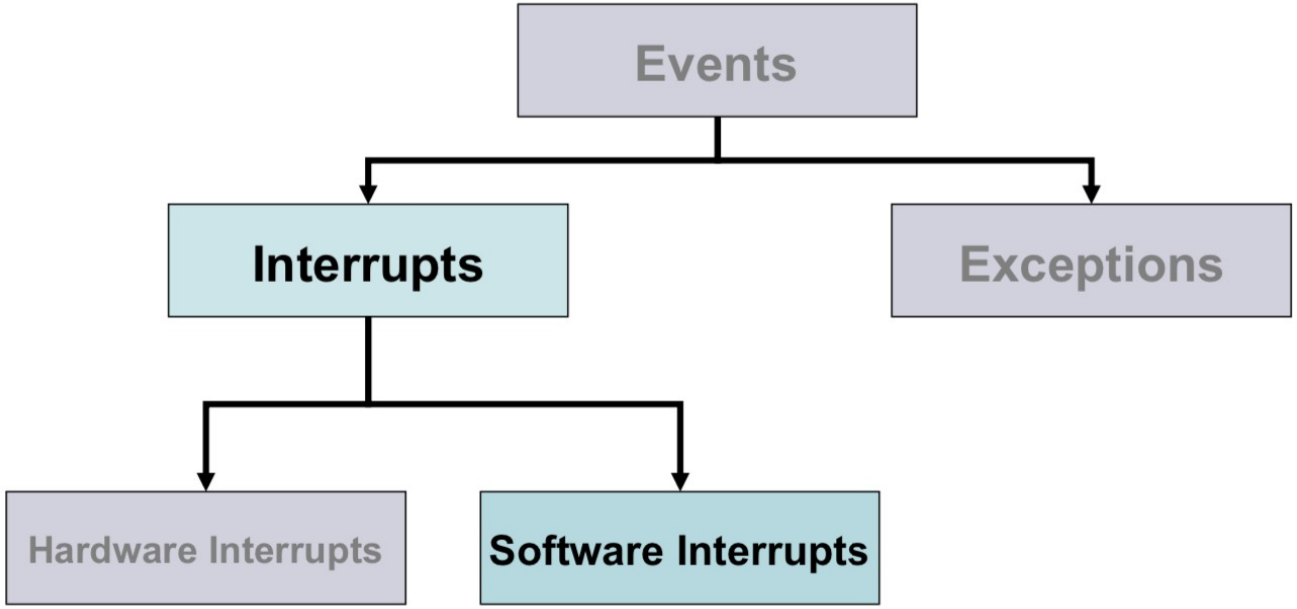
Event Types


Events
Interrupts
Exceptions
Hardware Interrupts
Software Interrupts

Event View of CPU

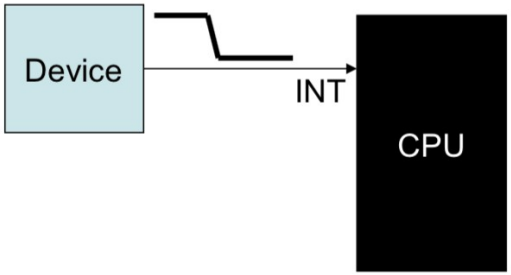


System Calls



Hardware vs Software Interrupt

Hardware Interrupt



A device (like PIC) asserts a pin in the CPU

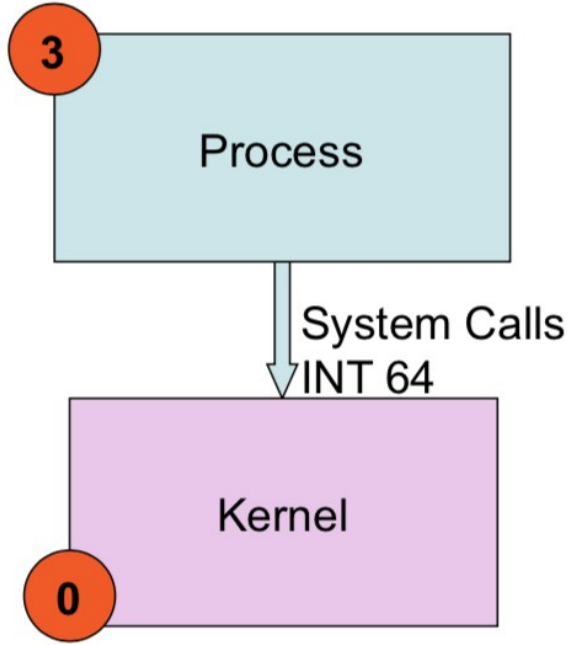
Software Interrupt

An instruction which when executed causes an interrupt

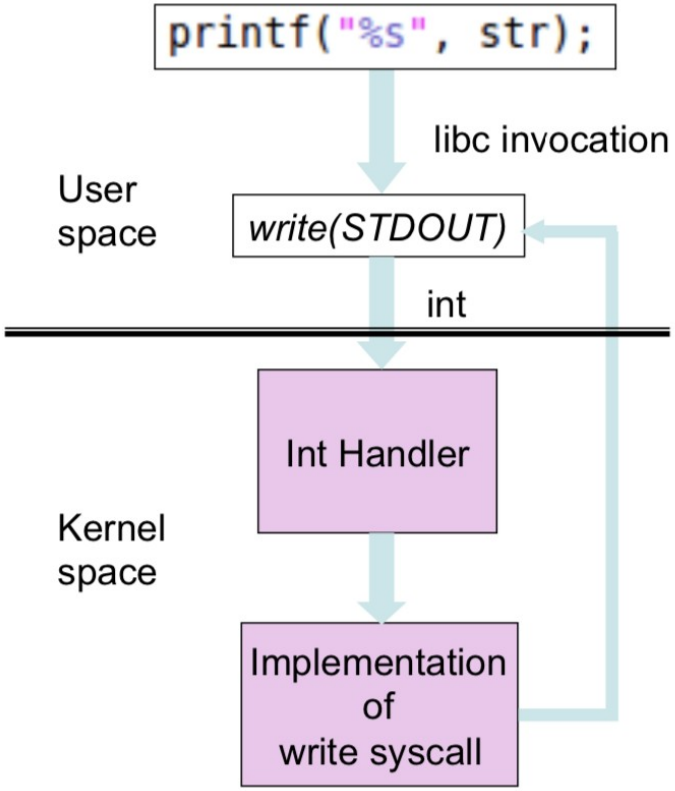
Software Interrupt

Software interrupt used for implementing system call

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

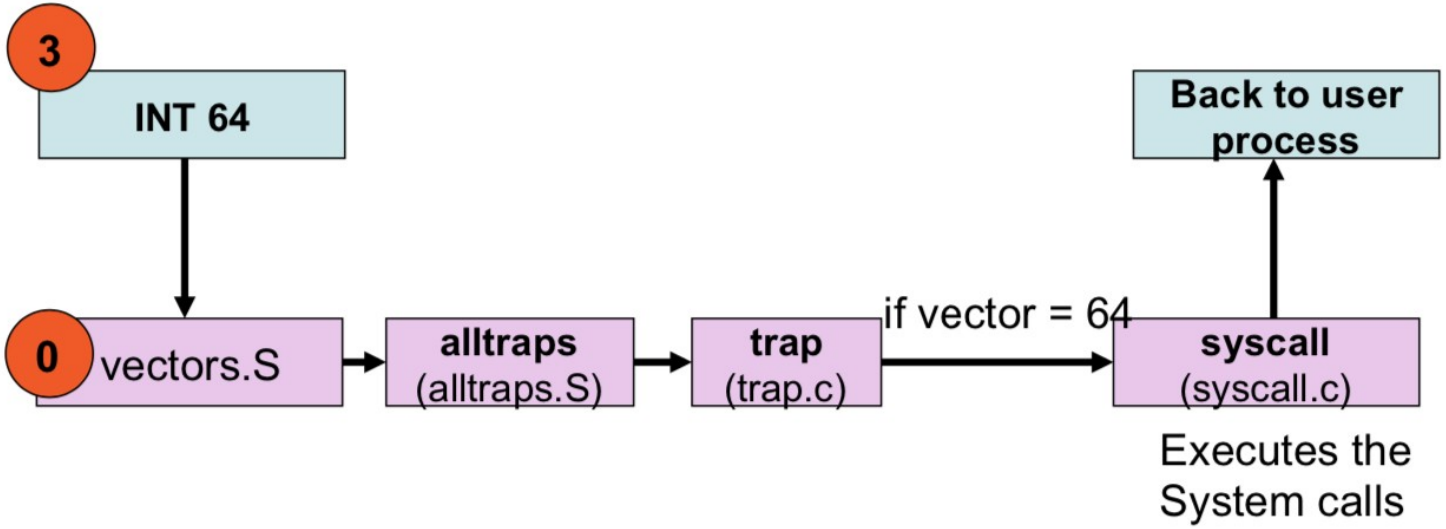


Example: Write System Call

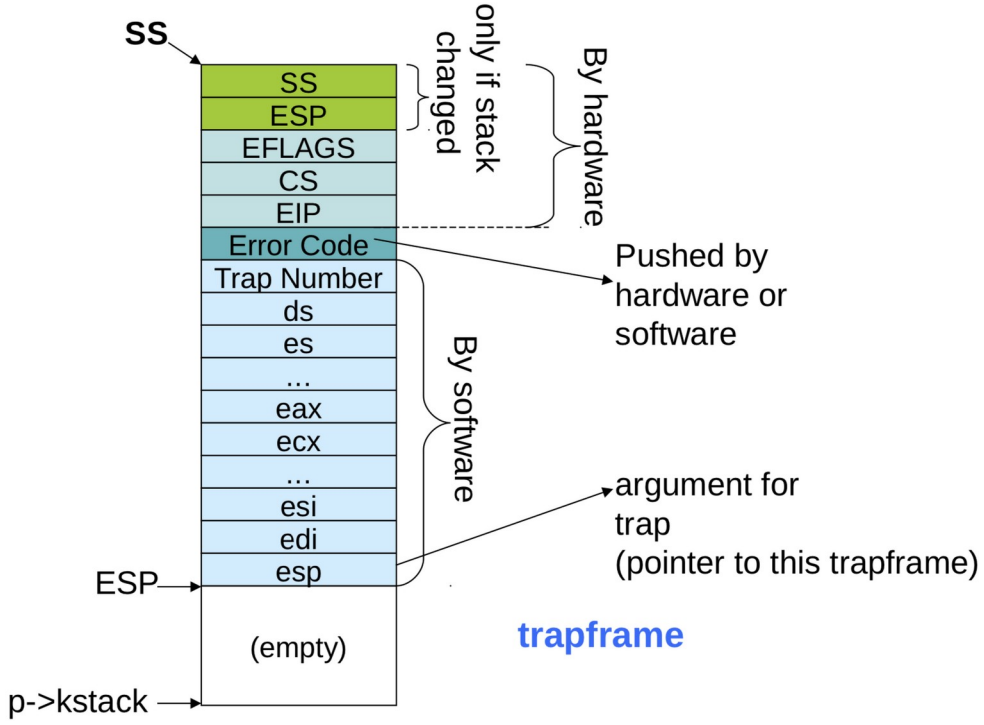


System Call Processing in kernel

Almost similar to hardware interrupts



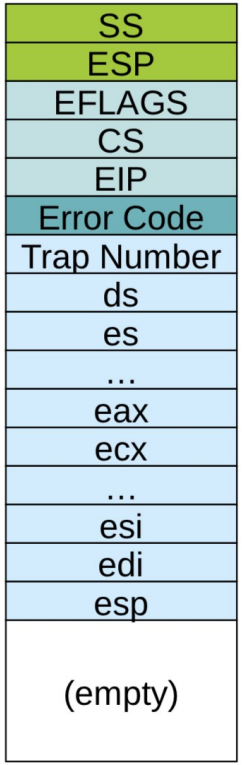
Trapframe



ref : struct trapframe in x86.h (0602 [06])

Trapframe Struct

```
0602 struct trapframe {
0603   // registers as pushed by pusha
0604   uint edi;
0605   uint esi;
0606   uint ebp;
0607   uint oesp;      // useless & ignored
0608   uint ebx;
0609   uint edx;
0610   uint ecx;
0611   uint eax;
0612
0613   // rest of trap frame
0614   ushort gs;
0615   ushort padding1;
0616   ushort fs;
0617   ushort padding2;
0618   ushort es;
0619   ushort padding3;
0620   ushort ds;
0621   ushort padding4;
0622   uint trapno;
0623
0624   // below here defined by x86 hardware
0625   uint err;
0626   uint eip;
0627   ushort cs;
0628   ushort padding5;
0629   uint eflags;
0630
0631   // below here only when crossing rings, such as from user to kernel
0632   uint esp;
0633   ushort ss;
0634   ushort padding6;
0635 };
```



System Calls in xv6

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

System call number

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

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

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,
```

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

xv6 System Call Naming Convention

- Usually a library function foo() will do some work and then call a system call sys_foo()
  - sys_foo() implemented in sys*.c (sysfile.c, sysproc.c)
- System call number for foo() is SYS_foo
  - syscalls.h
- All system calls begin with sys_

Syscall(void)

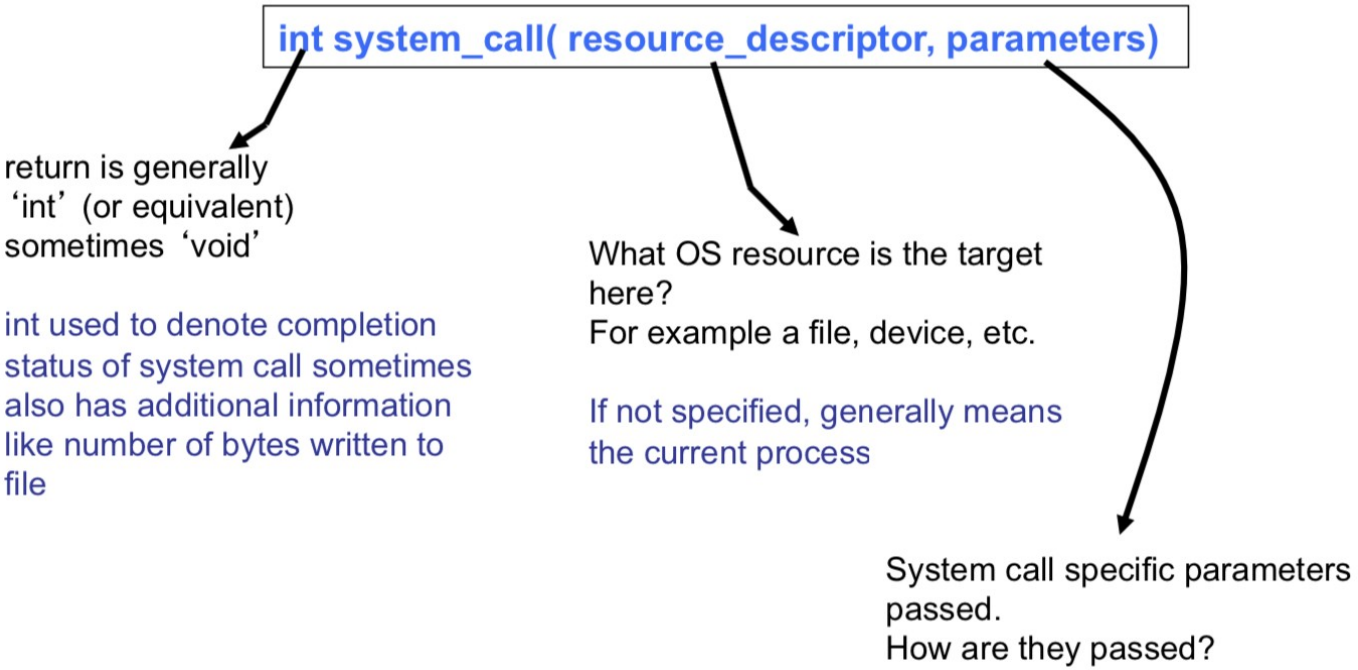
All system calls are handled in this function.
The sys num which is saved in eax register is retrieved and system call is read from table.

```
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



Adding New System Call

- A system call body is defined in sysproc.c or sysfile.c
- Multiple files are needed to be altered to add a new syscall.
- Desired new system calls in this project:
  - void find_largest_prime_factor (int num)
  - void change_file_size(const char* path, int length)
  - void get_callers(int syscall_number)
  - void get_parent_id()

Processes

A process is the running of a program, including the program's state and data. The state includes such things as:

- Memory the program occupied
- Memory contents
- Register values
- Files
- Kernel structures

Managing Processes

The kernel has a simple data structure for each process, organized in some list. The kernel juggles between the processes, using a context switch, which:

- Saves state of old process to memory.
- Loads state of new process from memory

The processes are held in a struct named ptable, who has a vector of processes named proc.

First Process

Processes are created by the kernel, after another process asks it to. Therefore, the kernel needs to run the first process itself, in order to create someone who will ask for new processes to be created

```
// 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
  volatile 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 structure Ref: proc.h

Passing Parameters in System Calls

- Passing parameters to system calls not similar to passing parameters in function calls
  - Recall stack changes from user mode stack to kernel stack.
- Typical Methods
  - Pass by Registers (eg. Linux)
  - Pass via user mode stack (eg. xv6)
    - Complex
  - Pass via a designated memory region
    - Address passed through registers

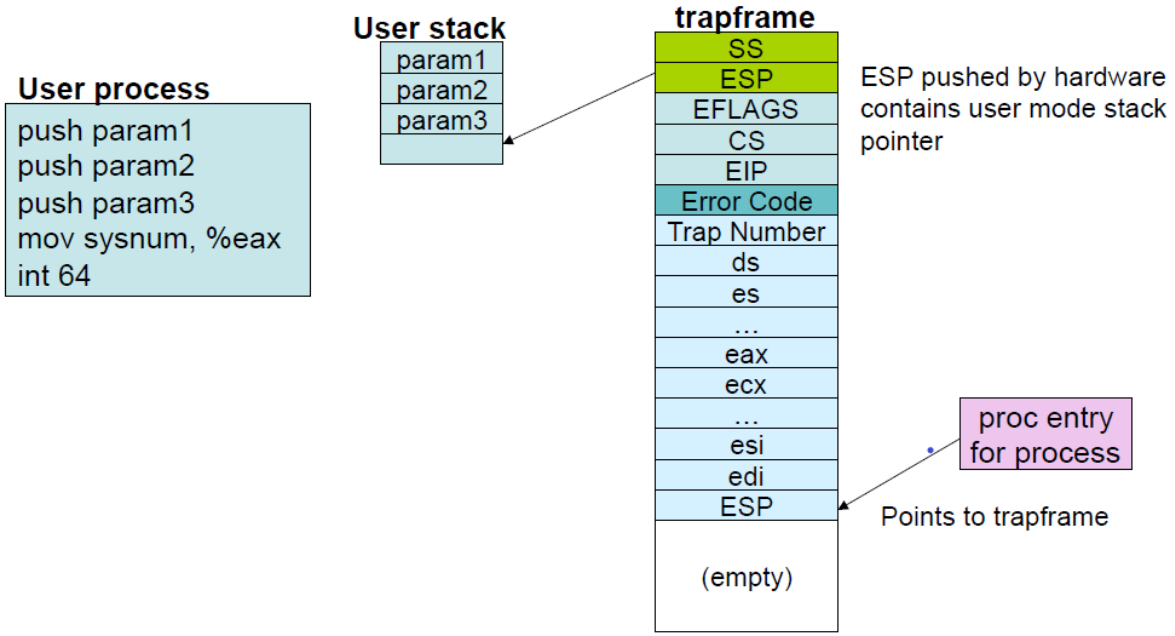
Pass By Registers (Linux)

- System calls with fewer than 6 parameters passed in registers
  %eax (sys call number), %ebx, %ecx,, %esi, %edi, %ebp
- If 6 or more arguments

  Pass pointer to block structure containing argument list
- Max size of argument is the register size (eg. 32 bit)

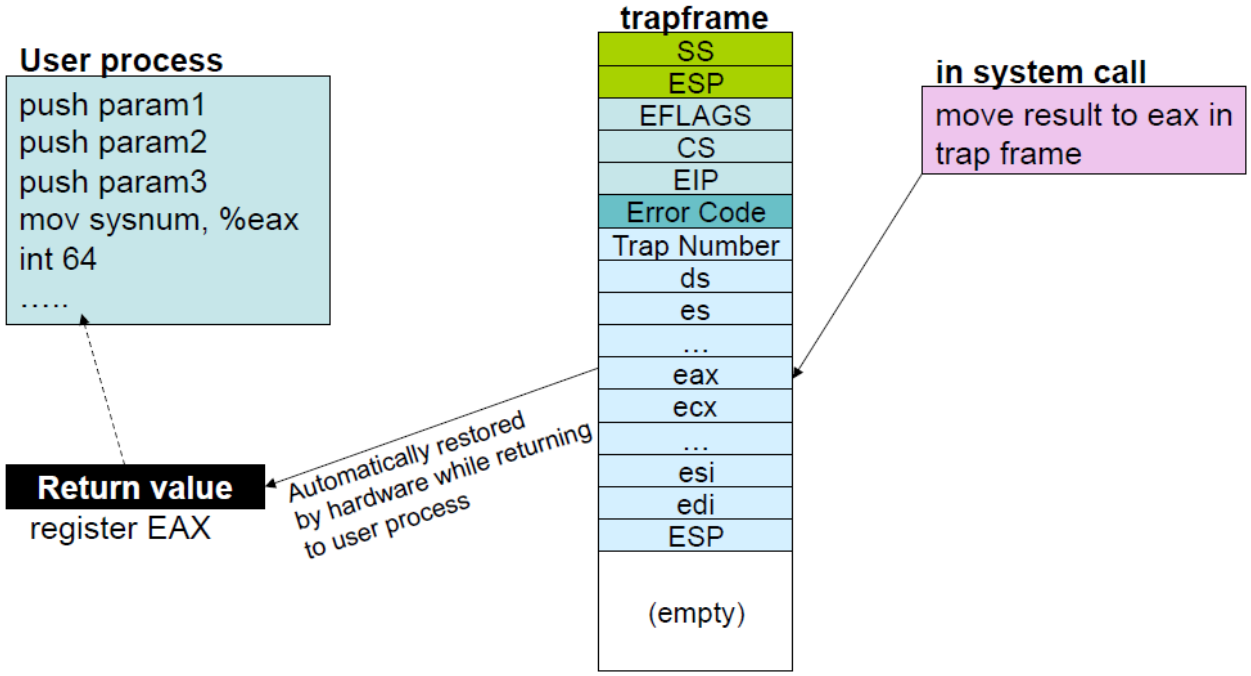
  Larger pointers passed through pointers

Pass via User Mode Stack (xv6)



ref : sys_open (sysfile.c), argint, fetchint (syscall.c)

Return from System Calls



Find Largest Prime Factor

int find_largest_prime_factor(int *num*)

- Get the number as argument
- Use a register to store the argument
- Return the largest prime factor of number

Change File Size

void chnage_file_size(const char* *path*, int *length*)

- Get the path of a file and the final length of file as argument
- Truncate the file to desired *length*
- If *length* > current length of file: extend the file with NULL characters

Get Callers

void get_callers(int *syscall_number*)

- ▷ Get the number of a system call as argument
- ▷ Print all the process PIDs which have used that system call

Get Parent PID

int get_parent_pid(void)

- Return the parent PID of currently running process