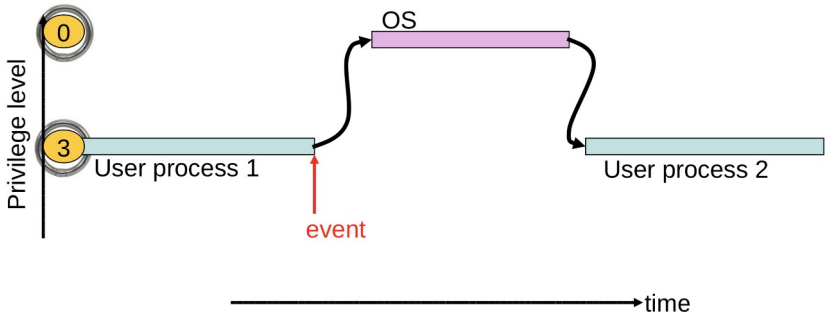
Lab Project 2:
System Calls and Processes

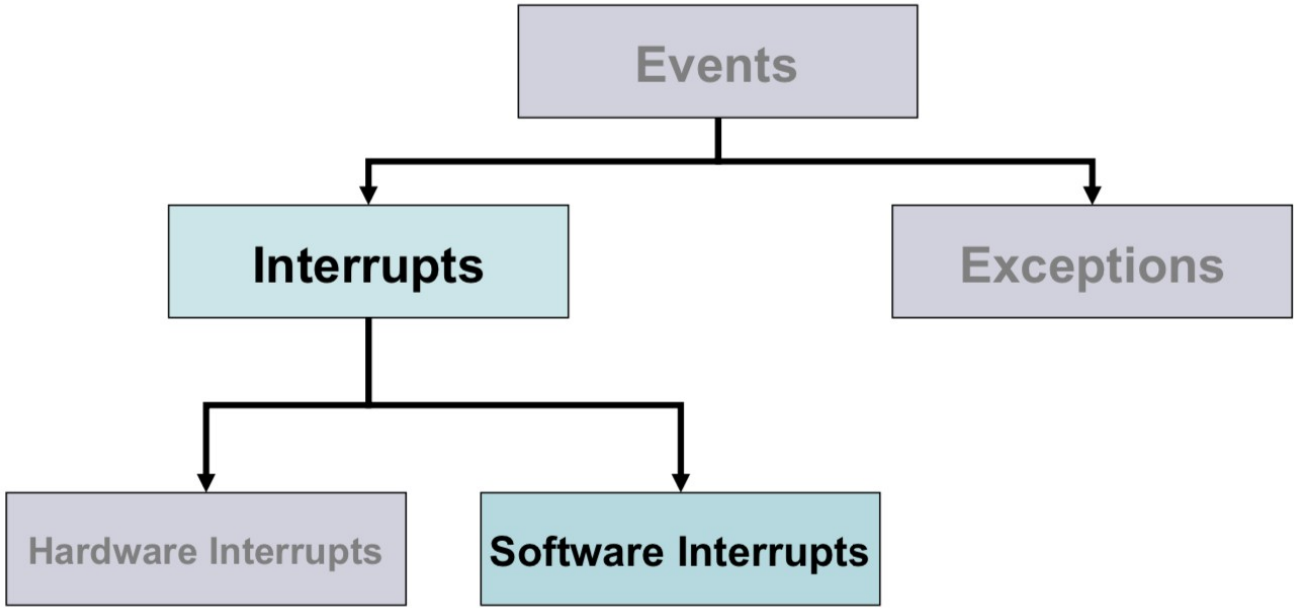
Misagh Mohaghegh
Mohammad GharehHasanloo

Why Event Driven Design?

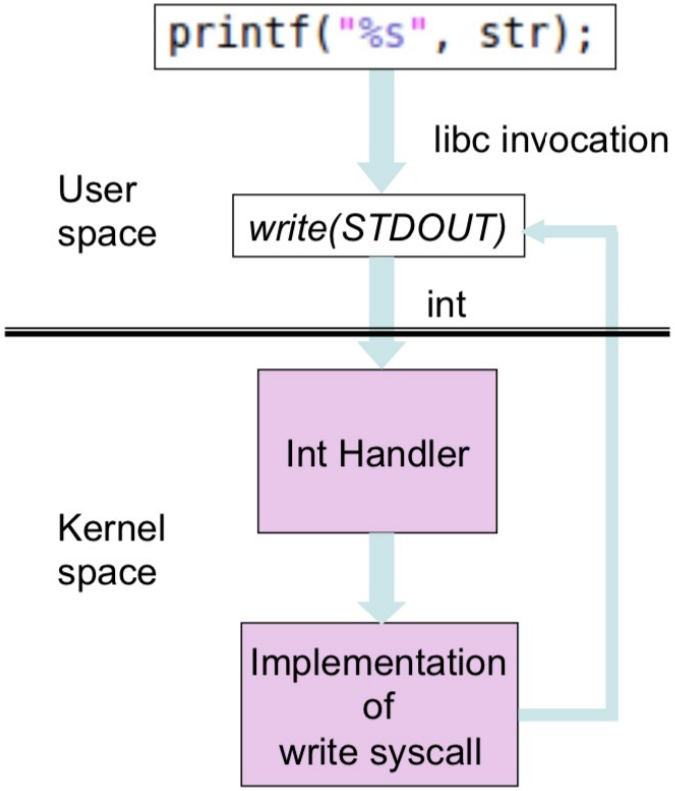
- OS cannot trust user processes
  - User processes may be buggy or malicious
  - User process crash should not affect OS
- OS needs to guarantee fairness to all user processes
  - One process cannot 'hog' CPU time
  - Timer interrupts



System Calls

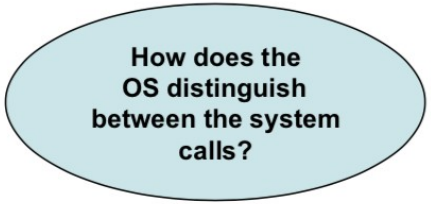


Write System Call

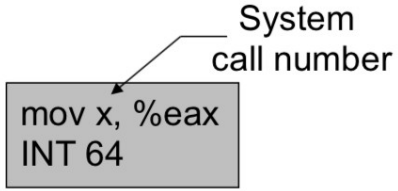


System Calls in xv6

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



System Call Number



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

Processes

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

find_digital_root

int find_digital_root(int n)

- Calculate the digital root of the input number. For instance, if the input number is 284, you should print the number 5 in the output.
- It's important to use registers to store the argument value, not for the address location. Additionally, after executing the system call, the unchanged value of the register should be preserved.
- Test:
  - Call this system call with different parameters and display results

copy_file

int copy_file(const char* src, const char* dest)

- This system call takes two file names as input and copies the file with the first name to the location specified by the second name. It returns 0 on success, otherwise -1.
- Test:
  - Write a user-level program and call the system call as described. Display the result of the copy operation.

get_uncle_count

int get_uncle_count(int)

- Return the number of uncles of a process. An uncle process refers to the siblings of the parent process
- Test:
  - Create three child processes. Then, create an additional child for one of them
  - Finally, use the system call for the specific process and display the output

get_process_lifetime

int get_process_lifetime(int)

- Calculate the lifespan of a process from its creation to the time this system call is called.
- Test:
  - Create a child process and terminate it after 10 seconds
  - Display the lifespans of both the child and the parent processes