National Yang Ming Chiao Tung University
Computer Architecture & System Lab

System Calls

IOC5226 Operating System Capstone

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao Tung University

Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - MIT 6.828 Operating system engineering class, 2018
  - MIT 6.004 Operating system, 2018
  - Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC

Outline

- System calls
- System Call Anatomy
- Passing Parameters
- Traps
- vDSO & Virtual System Call
- Create a System Call

System Call

- **What is a system call?**
  - A user program can interact with the operating system using a system call
  - A number of services are requested by the program
  - A system call is initiated by executing a specific instruction
    - Triggers a switch to kernel mode

https://www.geeksforgeeks.org/introduction-of-system-call/

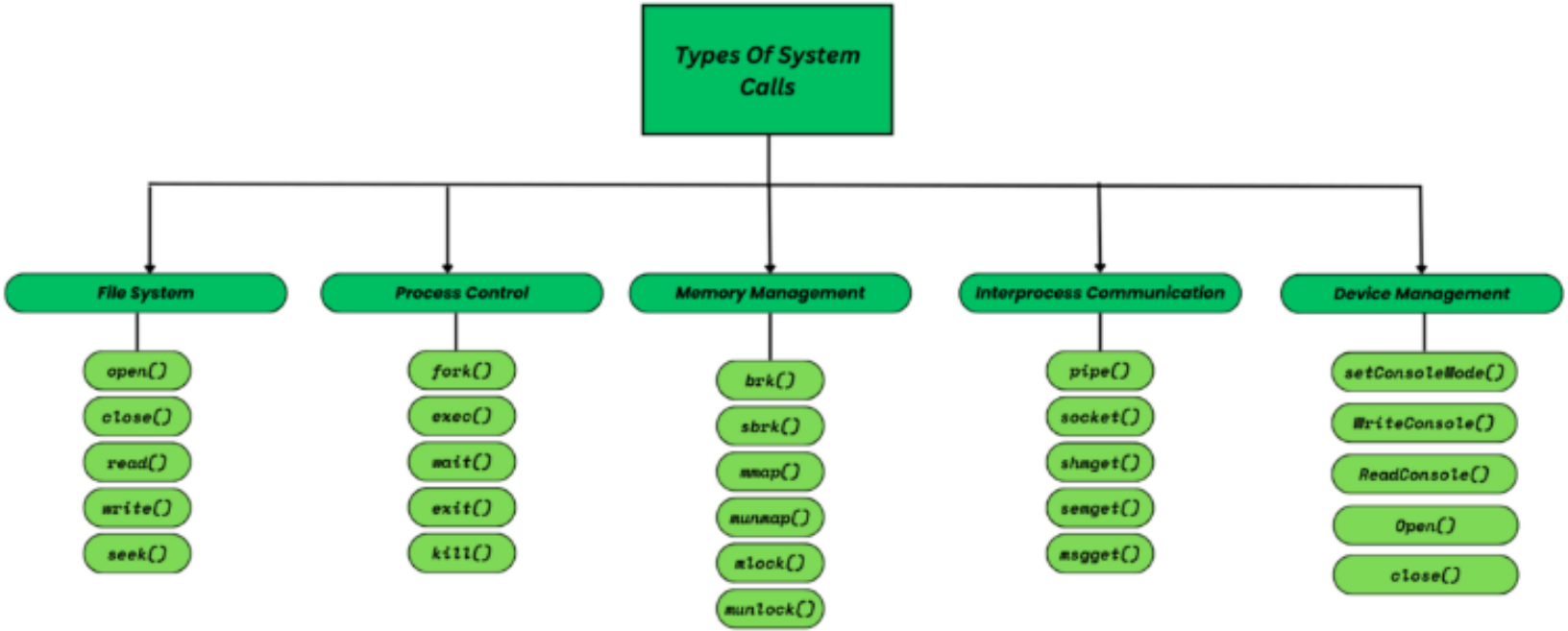
System Call

- **What is a system call?**
  - A user space request of a kernel service
  - A system call is just a C kernel space function
  - User space call to handle some request

```
#include <unistd.h>
int main (int argc, char **argv) {
…
  write (fd1, buf, strlen(buf));
…
}
```

System Call

- **Types of system calls**



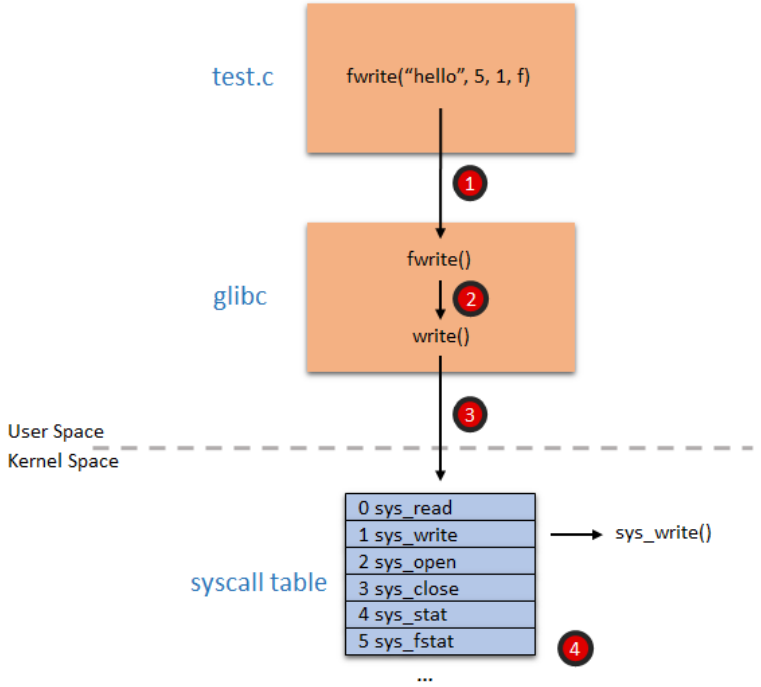
https://www.geeksforgeeks.org/introduction-of-system-call/

System Call

- **How many system call in Linux kernel?**
  - 322 different system calls in x86_64
  - 358 different system calls in x86
- **How to use system call from the user space?**
  - Using the wrapper functions defined in the C standard library
  - E.g. fopen, fgets, printf, and fclose …
  - Why do we use these wrapper functions without using the system call directly?
    - A system call must be quick and must be small

System Call

- **System calls**
  - Allow the kernel to expose certain key pieces of functionality to user programs
  - To execute a system call, a program must execute a special **trap** instruction



Source: http://randibox.blogspot.tw/2016/02/the-fascinating-world-of-linux-system.html

System Call

- **System calls**
  - Perform trap instruction-> vector to system call handler
    - Low level code carefully saves CPU state
    - Processor switches to kernel mode
    - Syscall handler checks param and jumps to desired handler
  - Return from system call
    - Result placed in register and low level code restores state
    - Perform “rte” instruction: switches to user mode and returns to location where “trap” was called

System Call

- In RISC-V
  - Execute `ecall` instruction to perform a system call
  - The `ecall` instruction
    - generates a software interrupt
    - raises a software exception
    - invokes an exception handling
  - Linux checks if the correct access rights to perform the requested operation
    - Give back an error code like EACCES (13) if denied

System Call

- **System calling convention is**
  - How should we do if a system call has many parameters?

1. **a0–a6**: Input parameters, up to seven parameters for the system call
2. **a7**: The Linux system call number
3. Invoke the operating system with "**ecall**"
4. **a0**: The return code from the call

System Call

- **What is a system call?**
  - A system call can be written in high-level languages or in assembly language

```
#include <asm/unistd.h>

.equ   O_RDONLY, 0
.equ   O_WRONLY, 1
.equ   O_CREAT,  0100
.equ   O_EXCL,   0200
.equ   S_RDWR,   0666
.equ   AT_FDCWD, -100

.macro  openFile    fileName, flags
        li    a0, AT_FDCWD
        la    a1, \fileName
        li    a2, \flags
        li    a3, S_RDWR    # RW access rights
        li    a7, __NR_openat
        ecall
.endm
```

System Call

- **Finding Linux System Call Numbers**
  - In Linux, the system call number
    - 93 is the number for exit
    - 64 is the number to write to a file
    - The Linux system call numbers are defined
    - /usr/include/asm-generic/unistd.h
    - #define __NR_write 64

System Call

- **Return codes**
  - The return code for system calls is usually zero (0) or a positive number for success
  - The negative number is for failure
    - /usr/include/errno.h

System Call

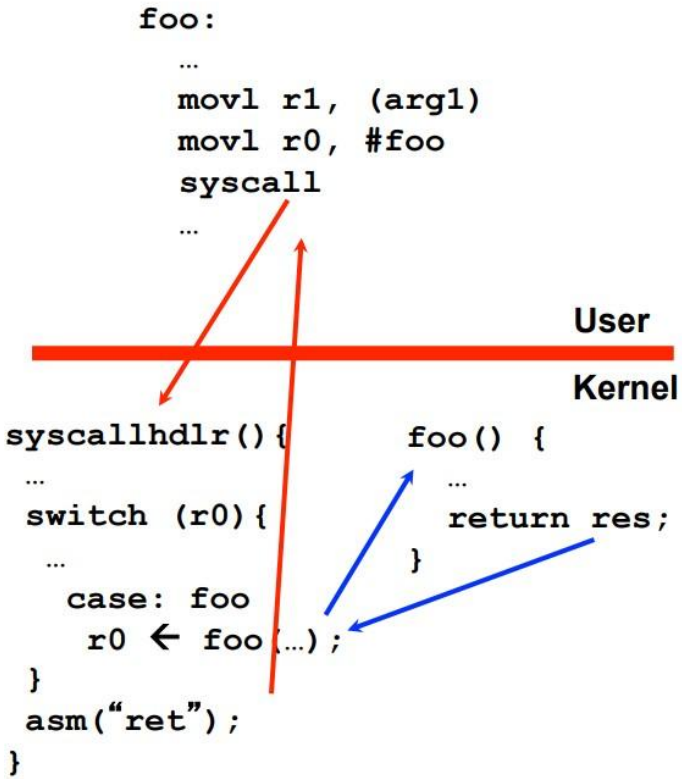
- **System call Examples**
  - The kill() system call can be used to send any signal to any process group or process.
    - Dont think that kill() is to terminate a process only. It can send all kinds of signals.
    - kill(getpid(),SIGSEGV);
  - The exit() terminates the calling process "immediately"
  - The exec() is the only way to get a program executed in Linux
    - execl(), execle(), execlp(), execv(), execvp()

Outline

- System calls
- System Call Anatomy
- Passing Parameters
- Traps
- vDSO & Virtual System Call
- Create a System Call

System Call Anatomy (1/5)

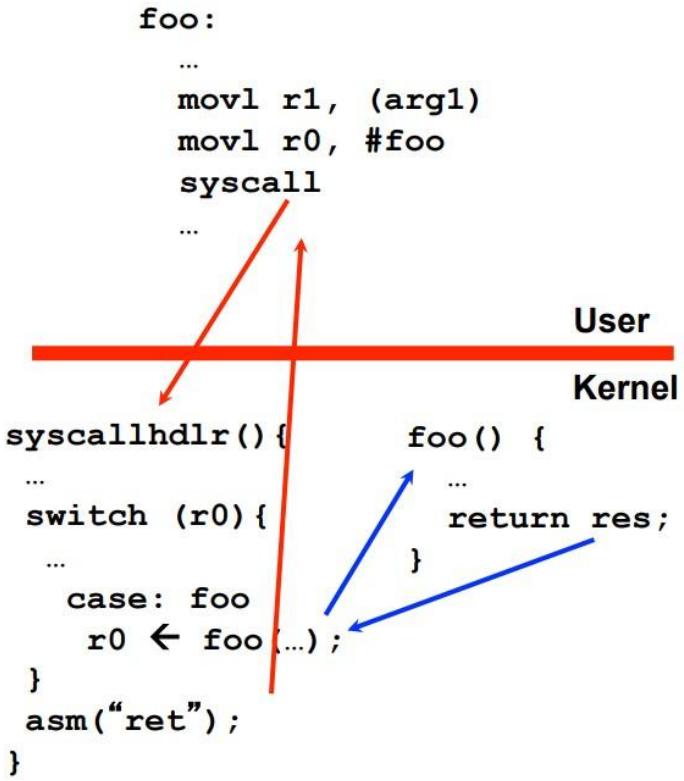
- **Anatomy of a system call**
  - Program puts syscall params in registers
  - Program executes a **trap**
    - Process state (PC, PSW) pushed on stack
    - CPU switches mode to KERNEL
    - CPU vectors to registered trap handler in the OS kernel



https://my.eng.utah.edu/~cs5460/slides/Lecture02.pdf

System Call Anatomy (2/5)

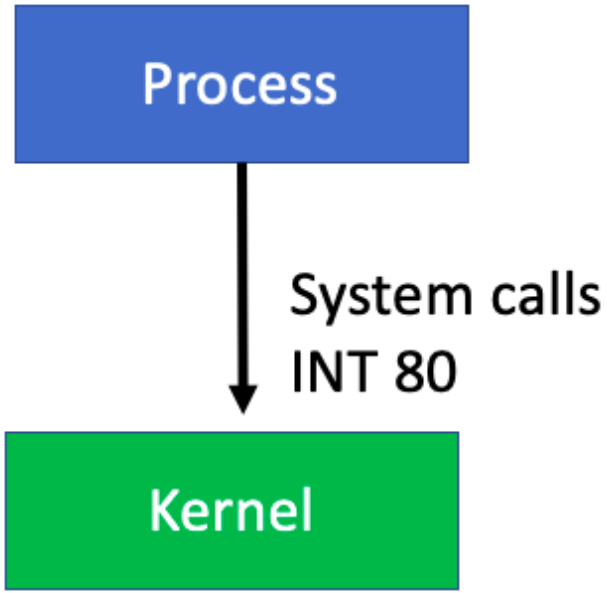
- **Anatomy of a system call**
  - Trap handler uses param to jump to desired handler (e.g. fork, exec, open…)
  - When complete, reserve operation
    - Place return code in register
    - Return from exception



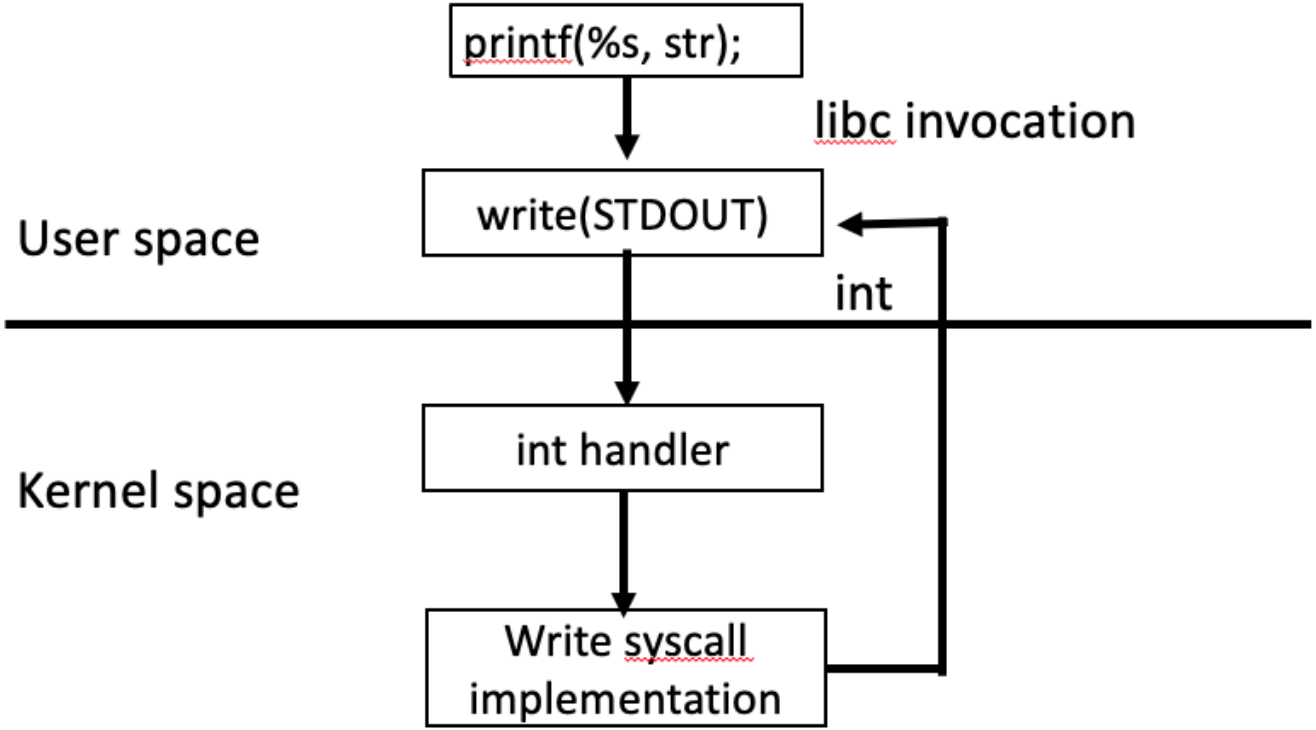
https://my.eng.utah.edu/~cs5460/slides/Lecture02.pdf

System Call Anatomy (3/5)

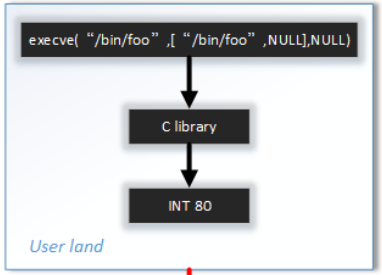
- **Software interrupt** used for implementing system calls
  - **INT** is an assembly language instruction for x86 processors that generates a software interrupt
  - In Linux INT 128 (0x80) (128 is interrupt number) used for system calls



System Call Anatomy (4/5)

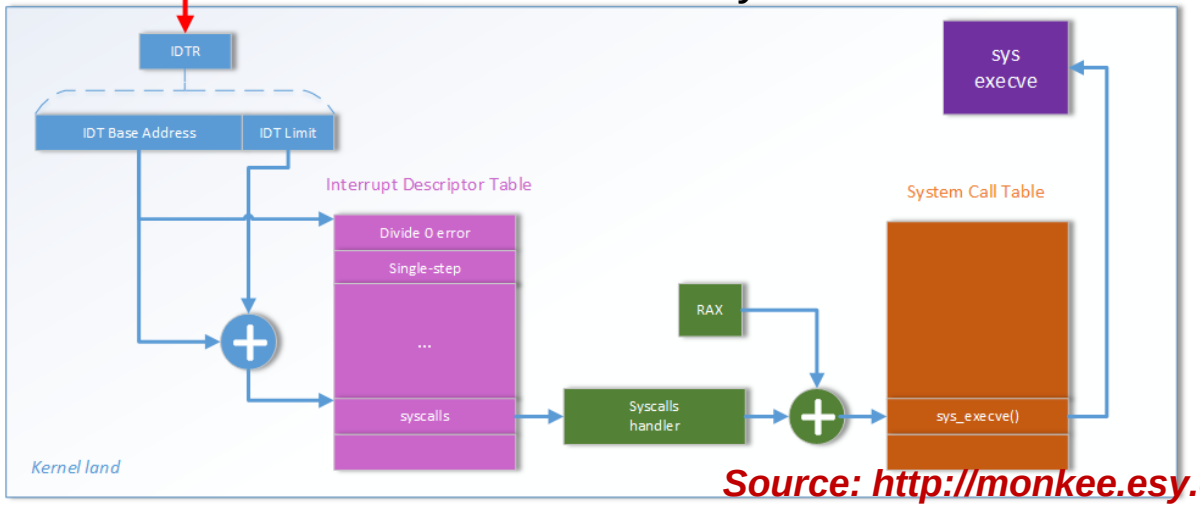


System Call Anatomy (5/5)



NR	syscall name	references	%rax
0	read	man/ cs/	0x00
1	write	man/ cs/	0x01
2	open	man/ cs/	0x02
59	execve	man/ cs/	0x3b

System Call Table for x86_64 in Linux



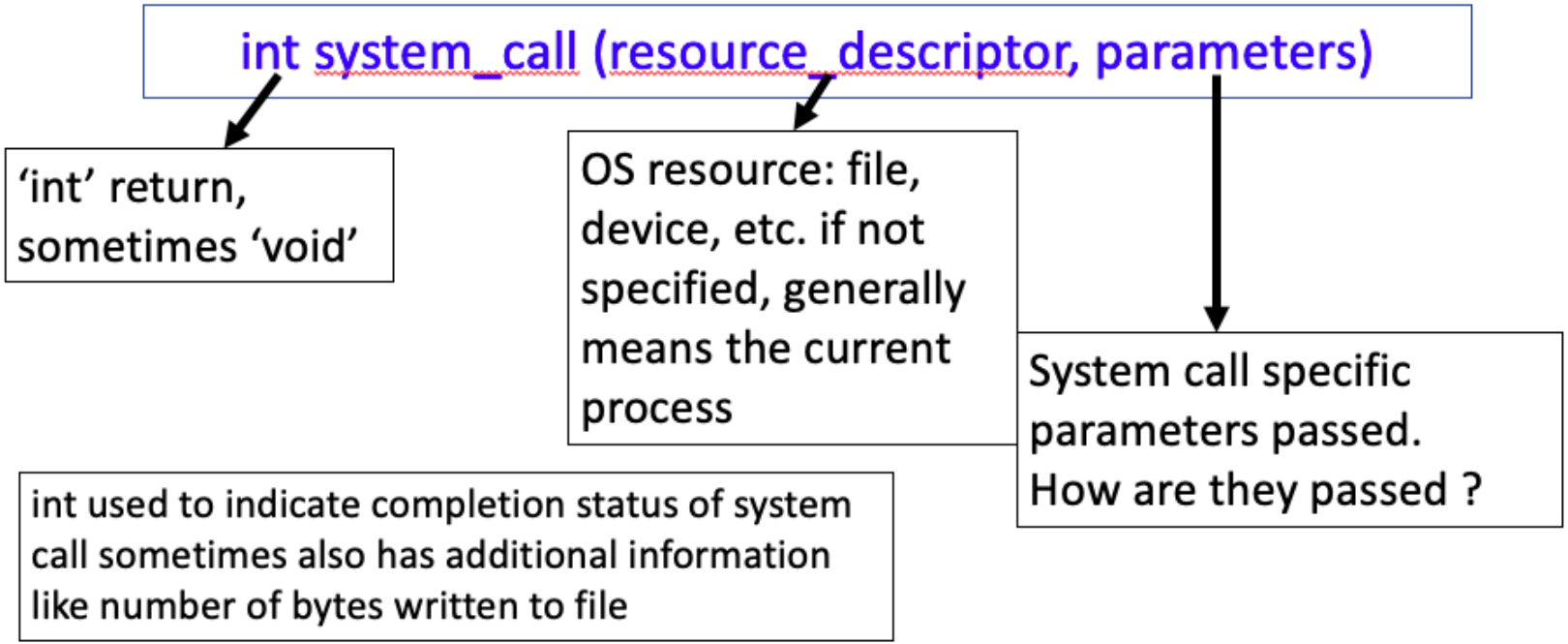
Source: http://monkee.esy.es/?p=1349

Outline

- System calls
- System Call Anatomy
- Passing Parameters
- Traps
- vDSO & Virtual System Call
- Create a System Call

Passing Parameters

- **Prototype of a system call**



Passing Parameters

Source

```
void foo (void) {
  write(1, "hello\n", 6);
}
```

Assembly code

```
<main>:
  pushq   %rax
  mov     $0x6,%edx
  mov     $0x694010,%esi
  mov     $0x1,%edi
  callq   libc_write
  xorl    %eax,%eax
  popq    %rdx
  ret
<libc_write>:
  mov     $0x1,%eax
  syscall
  cmp     $0xfffffffffffff001,%rax
  jae     <__syscall_error>
  retq
```

Passing Parameters

- Typical methods
  - Pass by **registers** (e.g. Linux)
    - Pros: fast
    - Cons: limited registers, cannot pass too many params

```
const char* pathname = "example.txt";
int flags = O_RDONLY;
mode_t mode = 0644;

int fd = open(pathname, flags, mode);
// in function call open(), we passed the parameters pathanme,flags,mode to the kernal directly
```

Passing Parameters

- Typical methods
  - Pass via a **designated memory region**
    - When the number of parameters are greater than the number of registers
    - Parameters are stored in blocks or table

```
int params[3];
      // Block of data(parameters) in array
params[0] = (int)pathname;
params[1] = flags;
params[2] = mode;

int fd = syscall(SYS_open, params);
      // system call
```

Passing Parameters

- Typical methods
  - Using **system stack** to store parameters
    - “Push”: store params; “Pop”: load params
    - Pros: can store more parameters; Cons: slow
    - Retrieve information from the top of the stack

```
int fd;
asm volatile(
    "mov %1, %%rdi\n"
    "mov %2, %%rsi\n"
    "mov %3, %%rdx\n"
    "mov $2, %%rax\n"
    "syscall"
    : "=a" (fd)
    : "r" (pathname), "r" (flags), "r" (mode)
    : "%rdi", "%rsi", "%rdx"
);
```

Advantage of System Calls

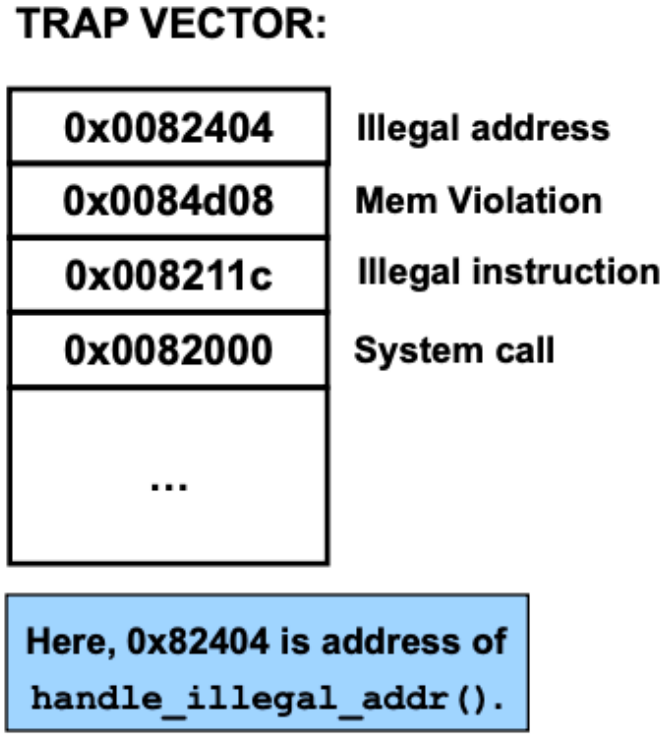
- **Access to Hardware Resources**
  - Allow programs to access hardware resources
- **Memory Management**
  - Allow programs to allocate and deallocate memory, as well as access memory-mapped hardware devices
- **Process Management**
  - Allow programs to create and terminate processes
- **Security**
  - Allow programs to access privileged resources

Outline

- System calls
- System Call Anatomy
- Passing Parameters
- Traps
- vDSO & Virtual System Call
- Create a System Call

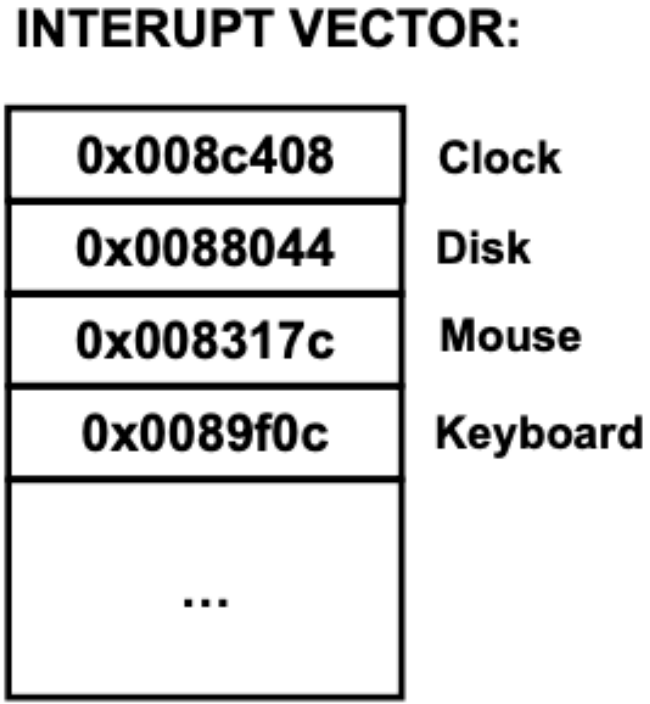
Traps (1/3)

- Used to detects special events
  - Invalid memory access…
- When processor detects condition
  - Save minimal CPU state (PC, sp, …)
  - Switch to KERNEL mode
  - Transfer control to trap handler
    - Indexes trap table w/ trap number
    - Jump to address in trap table
  - RTE/IRTE instruction reverses operation



Traps (2/3)

- Interrupt raises signal on CPU pin
  - Each device uses a particular interrupt number
  - CPU “traps” to the appropriate interrupt handler next cycle
- Interrupts can cost performance
  - Flush CPU pipeline + cache/TLB misses
  - Handlers often need to disable interrupt



Traps (3/3)

- Traps are synchronous
  - Generated inside the processor due to instruction being executed
  - Cannot be masked
  - System calls are one kind of trap
- Interrupts are asynchronous
  - Generated outside the processor
  - Can be masked

Booting

- **What happens at boot time?**

1. CPU jumps to fixed piece of ROM
2. Boot ROM uses registers as scratch space until it sets up VM and stack
3. Copy code/data from PROM to mem
4. Set up trap/interrupt vectors
5. Turn on virtual memory
6. Initialize display and other devices
7. Map and initialize "kernel stack" (*) for `init` process
8. Create `init`'s process cntl block
9. Create `init`'s address space, including space for kernel stack (*)
10. Create a system call frame on that kernel stack for `execl("/init",...)`
11. Switch to that stack
12. Switch to faked up syscall stack
13. Turn on interrupts
14. Do any initialization that requires interrupts to be enabled
15. "Return" from fake system call
16. Init runs – sets up rest of OS

- What is "kernel stack"?
- Where is "kernel stack"?
  - During boot process
  - During normal system call

- Whenever process "wakes up", it is in scheduler (including `init`)!

https://my.eng.utah.edu/~cs5460/slides/Lecture04.pdf

Outline

- System calls
- System Call Anatomy
- Passing Parameters
- Traps
- vDSO & Virtual System Call
- Create a System Call

vDSO & vsyscall (1/5)

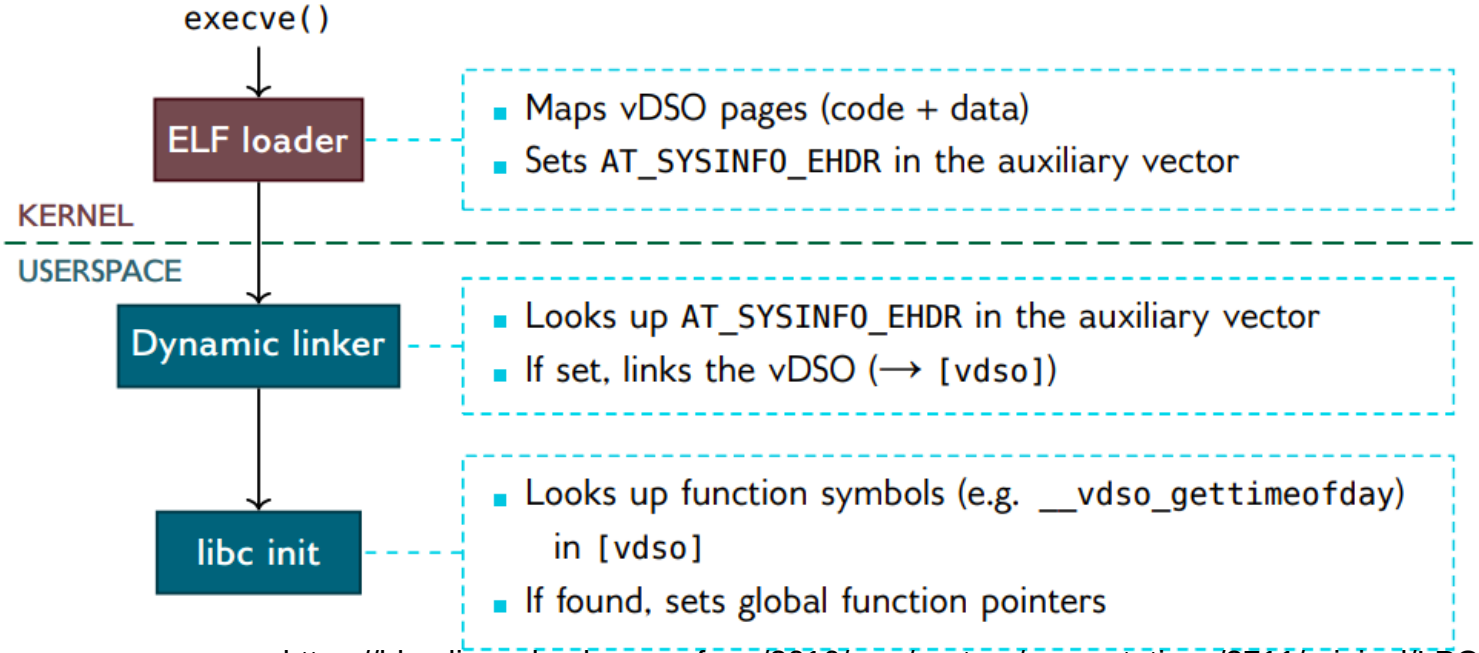
- Virtual system calls (vsyscall)
  - Certain system calls are fast to process
  - The system call itself (kernel enter/exit) causes a significant overhead
  - Certain system calls don’t require much privilege to process
- Solution: vsyscall
  - Map vsyscall data to two virtual memory addresses; write-only for kernel mode; read-only for user mode
  - The vsyscall won’t pass the user/kernel model transition
  - The vitural gettimeofday() can be up to 10 times faster

vDSO & vsyscall (2/5)

- vDSO: virtual DSO (Dynamic Shared Object)
  - Mapped by the kernel into all user processes
    - Linux kernel creates multiple DSO files and inserts them into the kernel during the compilation
    - The kernel will duplicates DSO to vsyscall memory pages
    - The kernel passes DSO address to the user space through “AT_SYSINTO_EHDR” in the auxiliary vector
  - Mainly meant for providing syscalls in user space

vDSO & vsyscall (3/5)

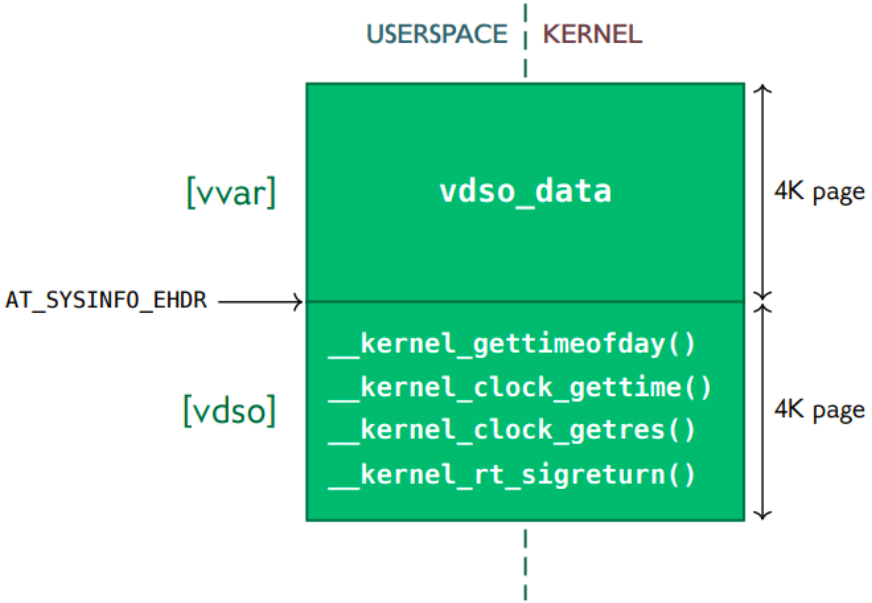
- Kernel and user space setup



https://blog.linuxplumbersconf.org/2016/ocw/system/presentations/3711/original/LPC_vDSO.pdf

vDSO & vsyscall (4/5)

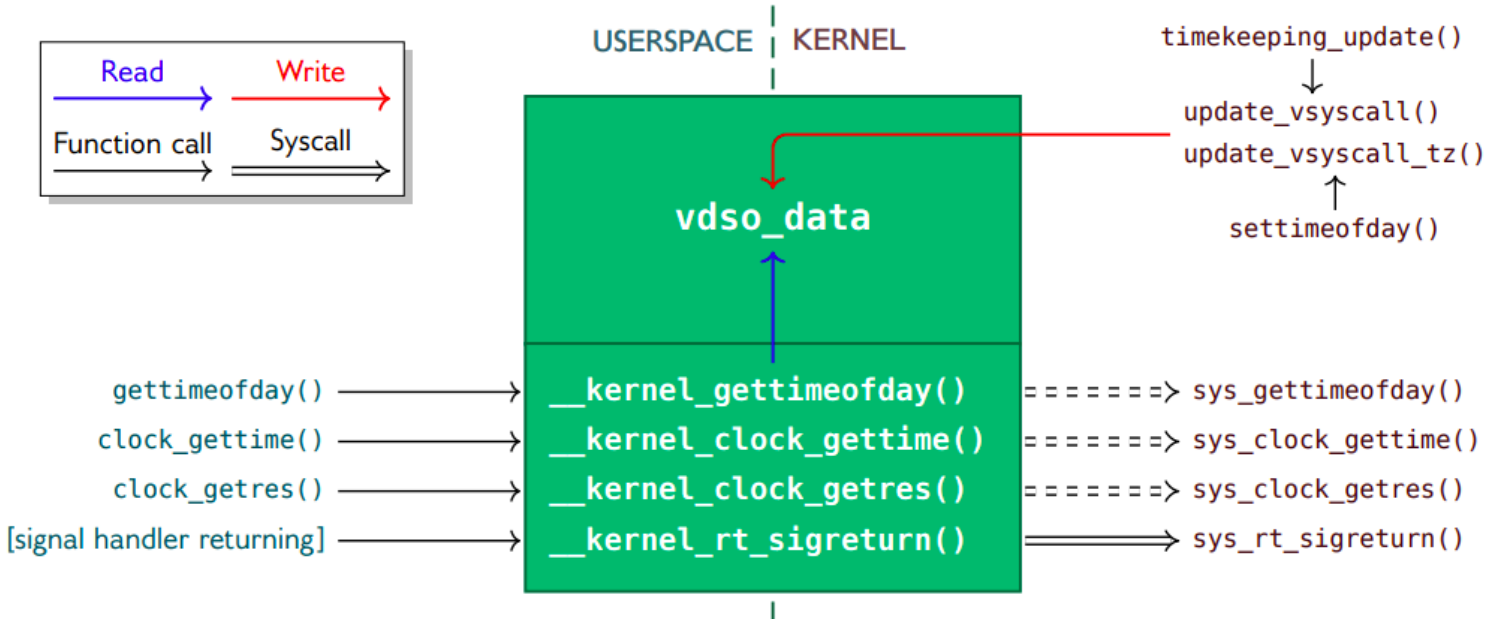
- Anatomy of the vDSO on arm64



https://blog.linuxplumbersconf.org/2016/ocw/system/presentations/3711/original/LPC_vDSO.pdf

vDSO & vsyscall (5/5)

- Anatomy of the vDSO on arm64



https://blog.linuxplumbersconf.org/2016/ocw/system/presentations/3711/original/LPC_vDSO.pdf

Outline

- System calls
- System Call Anatomy
- Passing Parameters
- Traps
- vDSO & Virtual System Call
- Create a System Call

Create a System Call (1/3)

- In Linux kernel > v4.10
- Create a new syscall folder
  - $ cd linux && mkdir workspace
- Write a new syscall
  - $ vim workspace/hello_world.c
- Create a Makefile
  - $ vim workspace/Makefile

```
obj-y := hello_world.o
```

```c
# include <linux/kernel.h>

asmlinkage long sys_hello_world(void)
{
    printk("Hello World!\n");
    return 0;
}
```

Create a System Call (2/3)

- Add our new syscall foler in the kernel Makefile

```
957 ...
958 ifeq ($(KBUILD_EXTMOD),)
959 core-y += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ workspace/
960 ...
```

- Update the system call table
  - $ vim arch/arm/tools/syscall.tbl

```
413 ...
414 397 common  statx         sys_statx
415 398 common  hello_world   sys_hello_world
```

Create a System Call (3/5)

- Update system call header file
  - $ vim include/linux/syscalls.h

```
941  ...
942                  unsigned mask, struct statx __user *buffer);
943  asmlinkage long sys_hello_world(void);
```

- Rebuild the kernel

tells your compiler to look on the CPU stack for the function parameters, instead of registers

Conclusion

- System calls
  - Arguments are placed in well-known registers
  - Perform trap instruction to activate the system call through the system call handler
  - IRTE/RTE returns from the system call
- OS manages trap/interrupt tables
  - Controls the “entry points” in the kernel -> secure

Takeaway Questions

- What is the purpose of a system call?
  - (A) A type of hardware interrupt
  - (B) A way for programs to request services from the OS
  - (C) A method of executing shell scripts
- Which system call is only used to terminate a process in Linux?
  - (A) exit ()
  - (B) kill ()
  - (C) terminate ()

Takeaway Questions

- What is the purpose of the exec () system call in Linux?
  - (A) To execute a program within the current process
  - (B) To control user permissions
  - (C) To create a new file