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

What is a System Call? (1/4)

- **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

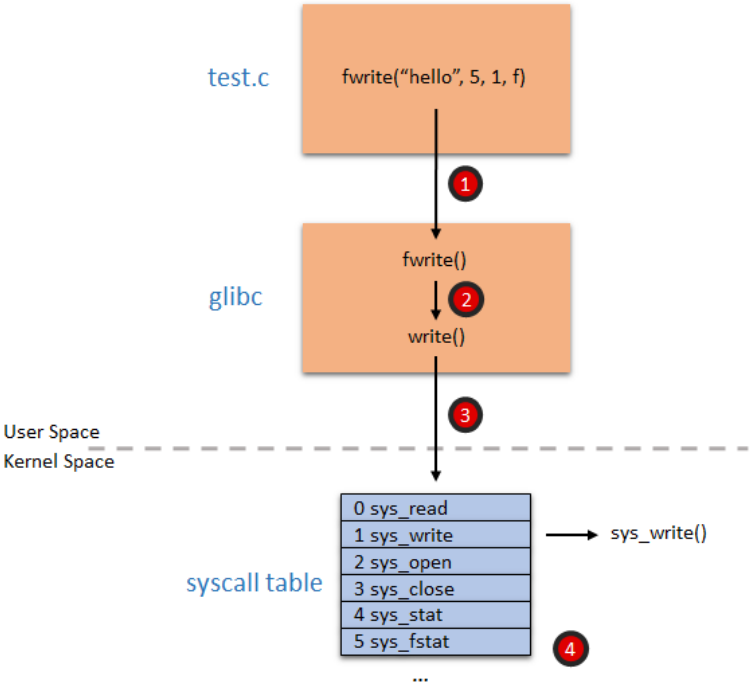
```
#include <unistd.h>
int main (int argc, char **argv) {
…
    write (fd1, buf, strlen(buf));
…
}
```

What is a System Call? (2/4)

- **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

What is a System Call? (3/4)

- **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

What is a System Call? (4/4)

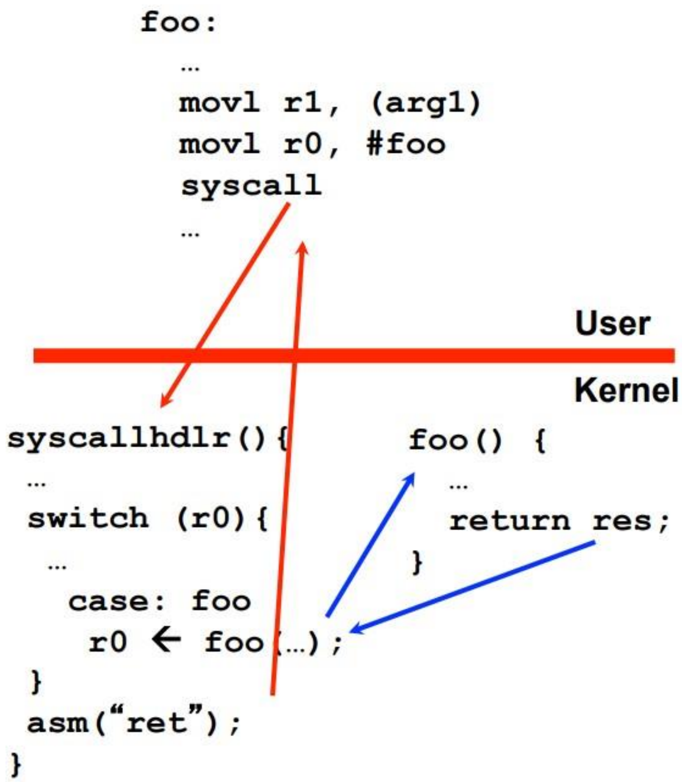
- **System calls**
  - Perform trap instruction-> vector to system call handler
    - Low level code carefully saves CPU state
    - Processor switches to kernel mode
    - Syscall handler checks param and jumps to desired handler
  - Return from system call
    - Result placed in register and low level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

Outline

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

System Call Anatomy (1/5)

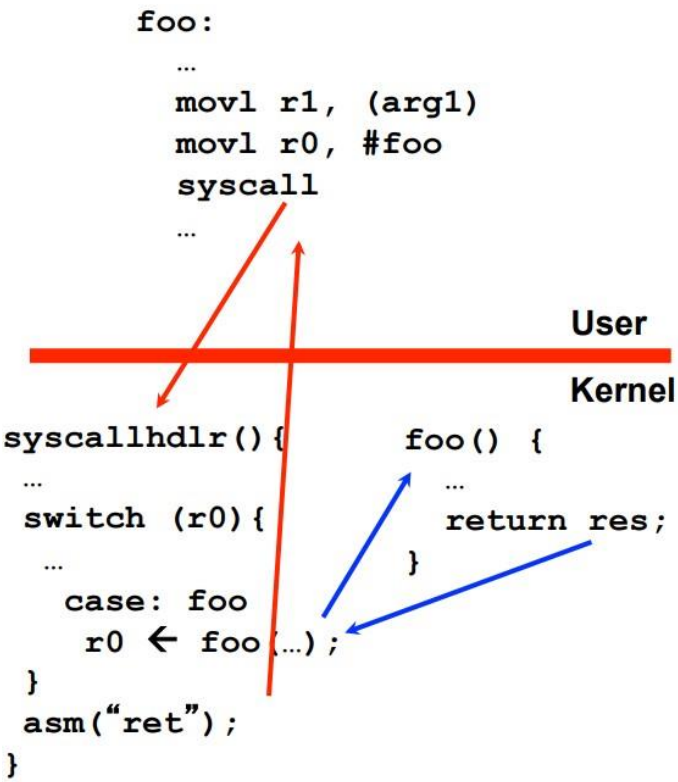
- **Anatomy of a system call**
  - Program puts syscall params in registers
  - Program executes a **trap**
    - Processor state (PC, PSW) pushed on stack
    - CPU switches mode to KERNEL
    - CPU vectors to registered trap handler
      in the OS kernel



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

System Call Anatomy (2/5)

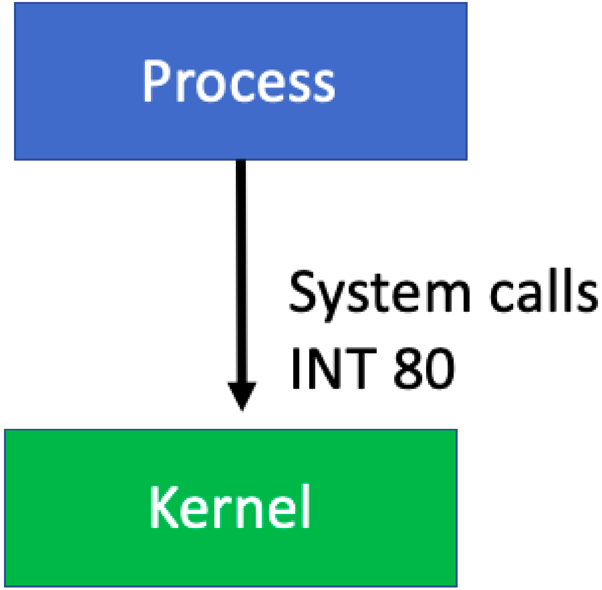
- **Anatomy of a system call**
  - Trap handler uses param to jump to desired handler (e.g. fork, exec, open…)
  - When complete, reserve operation
    - Place return code in register
    - Return from exception



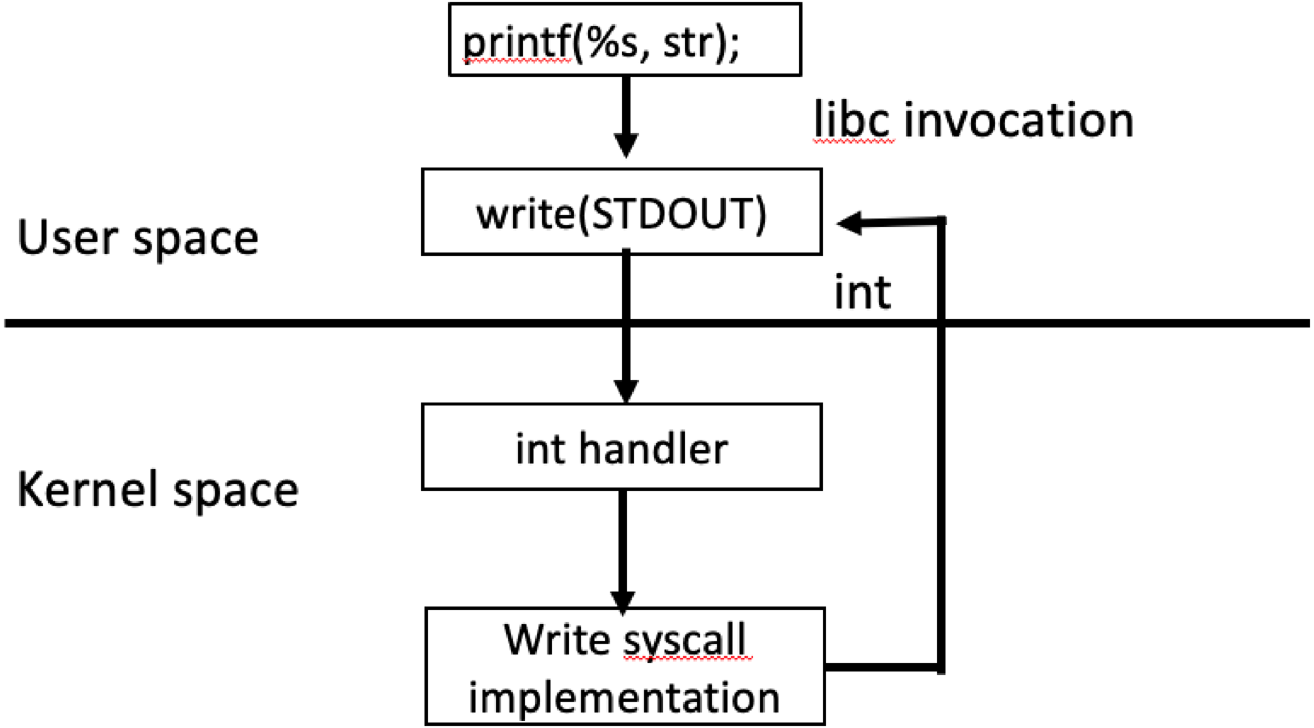
https://my.eng.utah.edu/~cs5460/slides/Lecture02.pdf

System Call Anatomy (3/5)

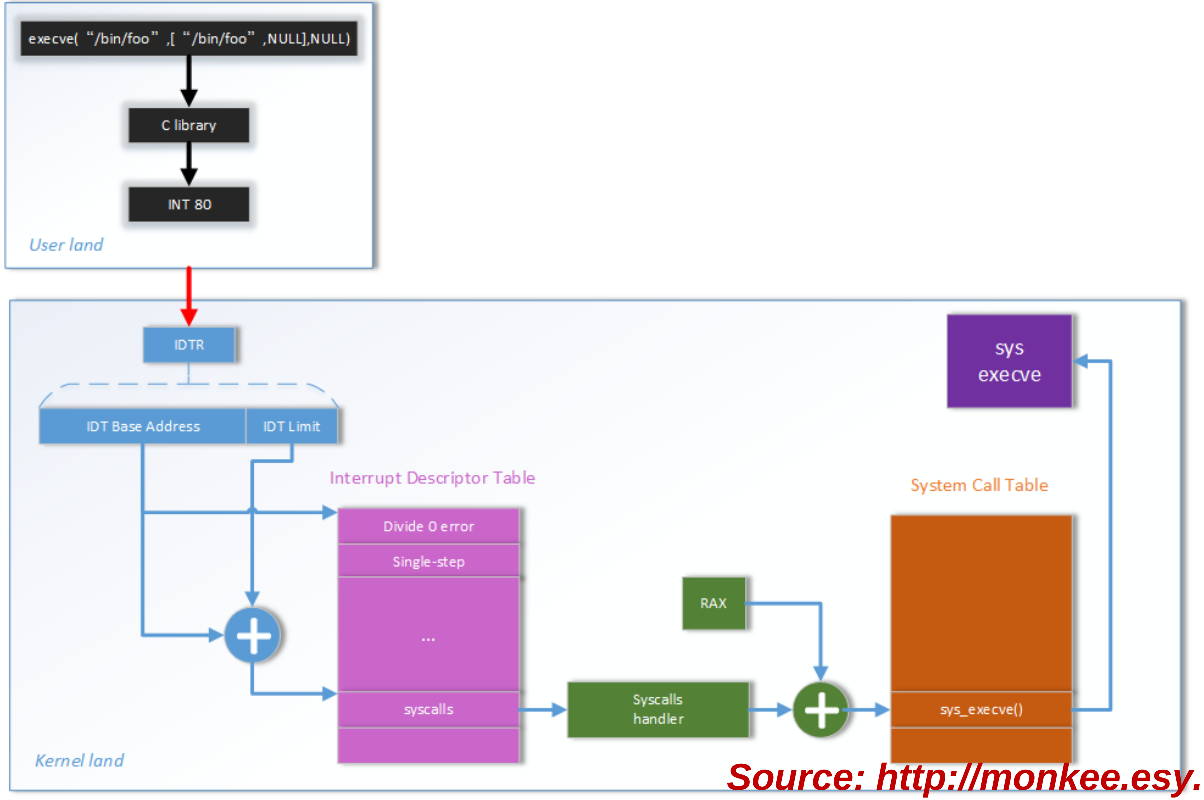
- **Software interrupt** used for implementing system calls
  - **INT** is an assembly language instruction for x86 processors that generates a software interrupt
  - In Linux INT 128 (0x80) (128 is interrupt number) used for system calls



System Call Anatomy (4/5)



System Call Anatomy (5/5)



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

Passing Parameters (1/5)

- **Prototype of a system call**

int system_call (resource_descriptor, parameters)

‘int’ return, sometimes ‘void’

OS resource: file, device, etc. if not specified, generally means the current process

System call specific parameters passed. How are they passed ?

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

Outline

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

Passing Parameters (2/5)

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 (3/5)

- Typical methods
  - Pass by **registers** (e.g. Linux)
    - Pros: fast
    - Cons: limited registers, cannot pass too many params
  - Using **system stack** to store parameters
    - “Push”: store params; “Pop”: load params
    - Pros: can store more parameters; Cons: slow
  - Pass via a **designated memory region**
    - Base address passed to registers

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

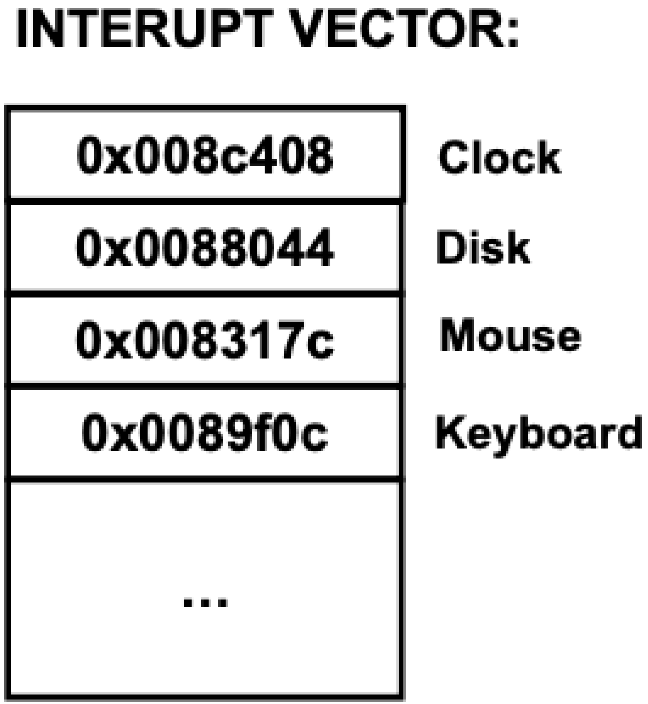
TRAP VECTOR:

0x0082404	Illegal address
0x0084d08	Mem Violation
0x008211c	Illegal instruction
0x0082000	System call
...	

Here, 0x82404 is address of `handle_illegal_addr()`.

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)

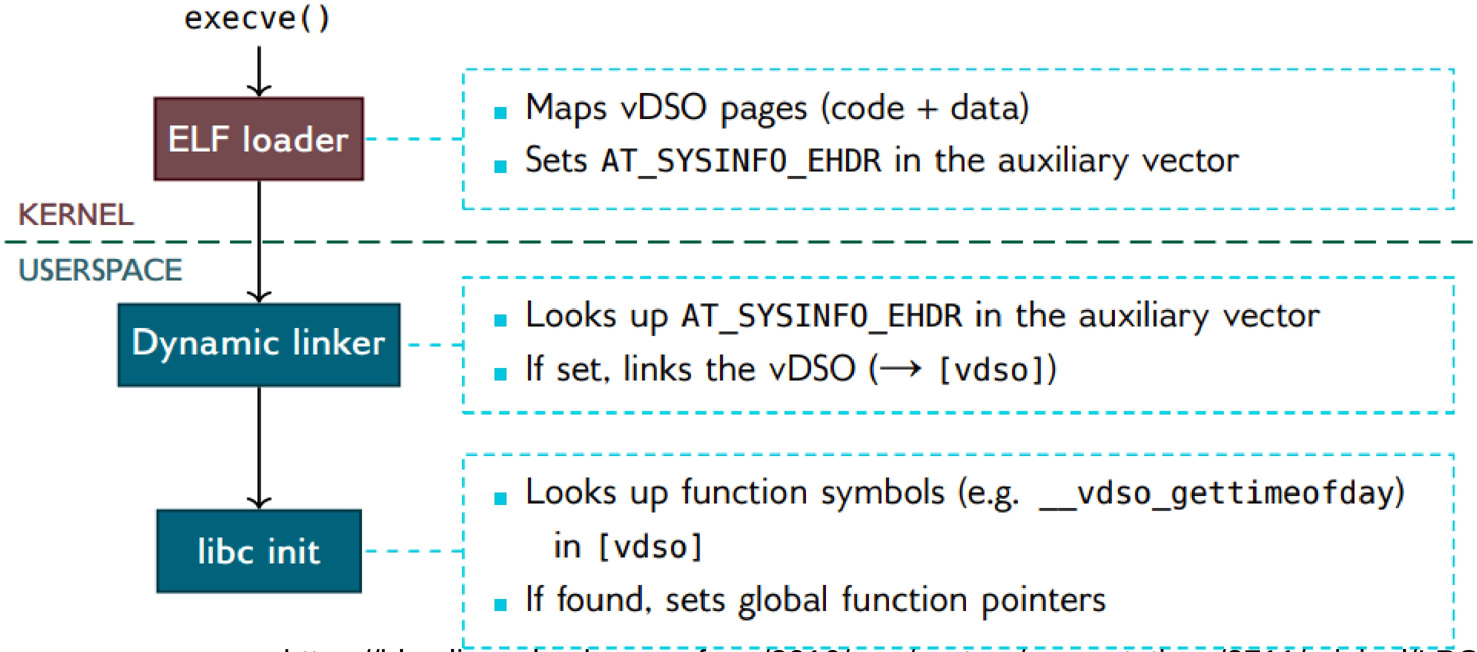
- Virtual system calls (vsyscall)
  - Certain system calls are fast to process
  - The system call itself (kernel enter/exit) causes a significant overhead
  - Certain system calls don’t require much privilege to process
- Solution: vsyscall
  - Map vsyscall data to two virtual memory addresses; write-only 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)

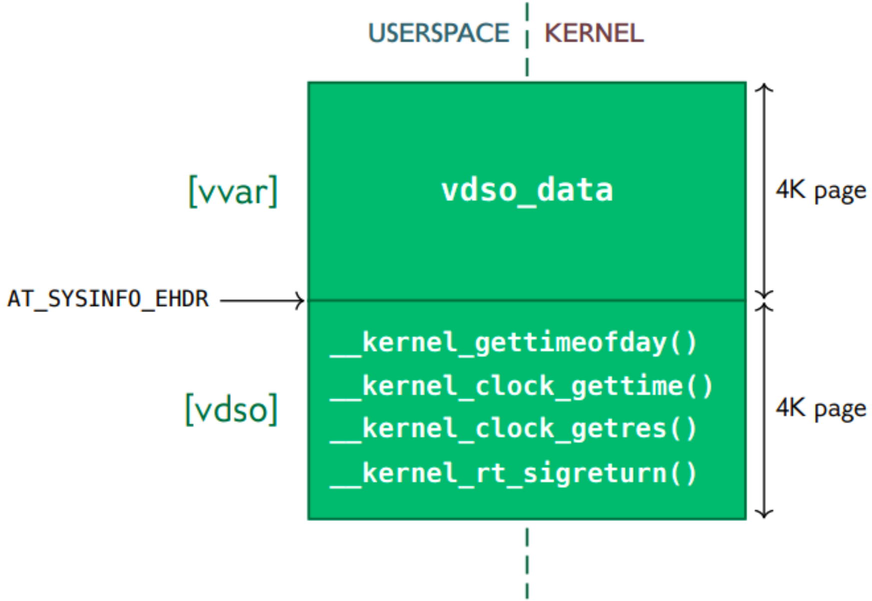
- Kernel and user space setup



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

vDSO & vsyscall (4/5)

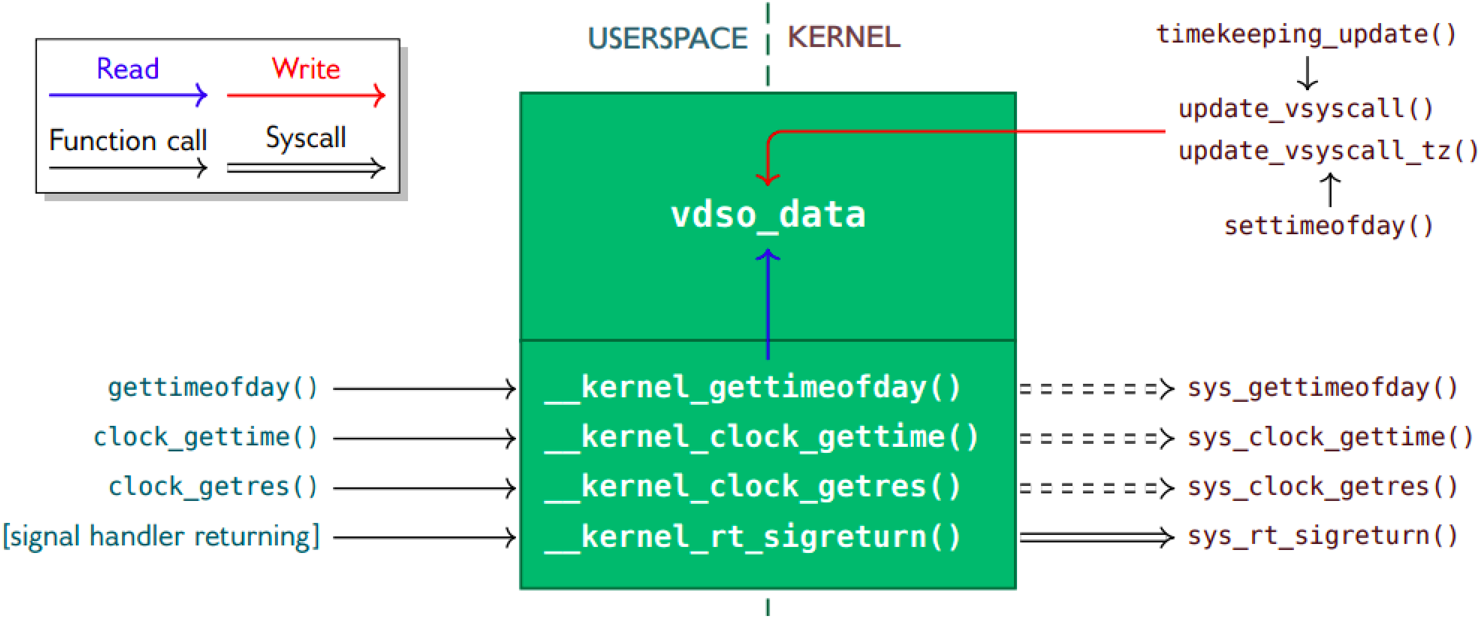
- Anatomy of the vDSO on arm64



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

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

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

```
...
397 common   statx         sys_statx
398 common   hello_world   sys_hello_world
```

Create a System Call (3/5)

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

```
...
                unsigned mask, struct statx __user *buffer);
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