Operating Systems
Lecture 4

OS Interfaces and Syscalls

Prof. Mengwei Xu

Recap of Last Course

IDTR →

IDT

Interrupt number →

Gate

offset

GDT or LDT

Segment Descriptor

base addr
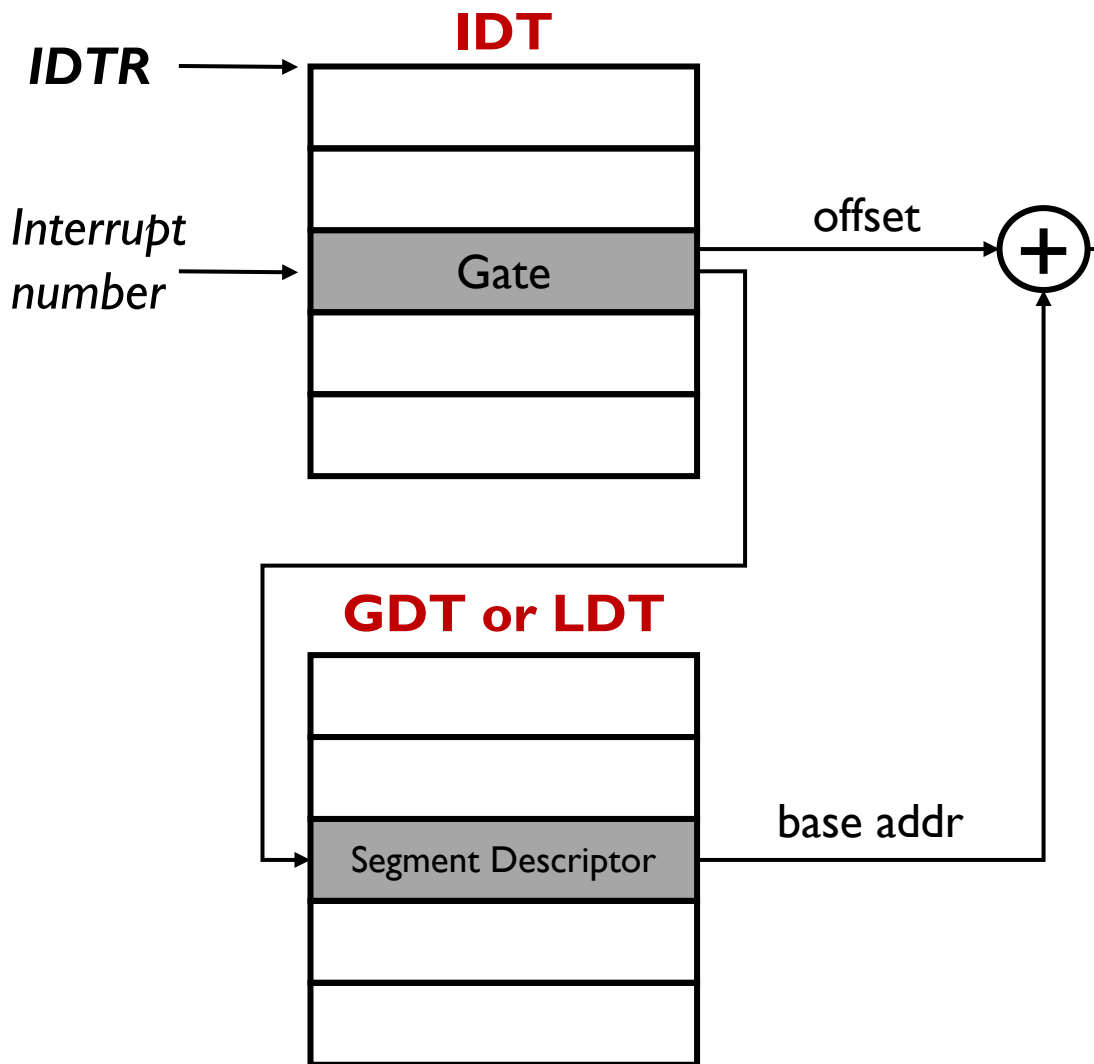
```
void
page_fault_handler(struct Trapframe *tf)
{
    uint32_t fault_va;

    // Read processor's CR2 register to find the faulting address
    fault_va = rcr2();

    // Handle kernel-mode page faults.

    // LAB 3: Your code here.
    if ((tf->tf_cs & 0x3) == 0)
        panic("Unhanlded page fault in kernel: %08x\n", fault_va);
```

Interrupt Procedure

Recap of Last Course

- Interrupt stack (中断栈) is a special stack in kernel memory that saves the interrupt process status.
    - Empty when there is no interrupt (running in user space)
    - Why not directly use the user-space stack?

- *Disable interrupts* and *enable interrupts* are two privileged instructions
    - Maskable interrupts (可屏蔽中断): all software interrupts, all system calls, and partial hardware exceptions
    - Non-maskable interrupts (NMI，不可屏蔽中断): partial hardware exceptions
    - Specified by *eflags* registers

Recap of Last Course

- User-to-Kernel Mode Switch
  - Exception
  - Interrupts
  - Syscalls

- Kernel-to-User Mode Switch
  - New process
  - Resume after an interrupt/exception/syscall
  - Switch to a different process
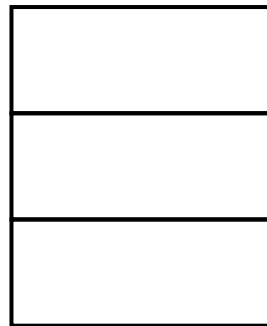   - ❑After a timer interrupt
  - User-level upcall

x86 Mode Transfer

- When an interrupt/exception/syscall occurs, the hardware will:

1. Mask interrupts
2. Save the special register values to other temporary registers
3. Switch onto the kernel interrupt stack
4. Push the three key values onto the new stack
5. Optionally save an error code
6. Invoke the interrupt handler

User-space process

```
foo() {
  x = x + 1;
  y = x + 2;
  …
}
```

Registers

- CS:EIP
- SS:ESP
- EFLAGS
- Others: EAX, EBX..

Kernel

```
handler() {
  pushad;
  …
}
```

User Stack

Interrupt Stack

Before interrupt

x86 Mode Transfer

- When an interrupt/exception/syscall occurs, the hardware will:

1. Mask interrupts
2. Save the special register values to other temporary registers
3. Switch onto the kernel interrupt stack
4. Push the three key values onto the new stack
5. Optionally save an error code
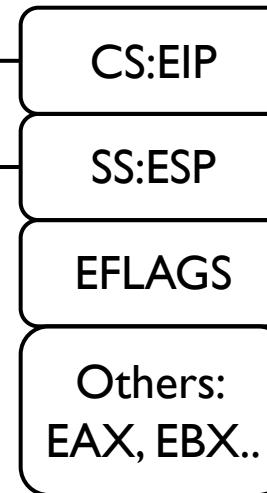6. Invoke the interrupt handler

User-space process
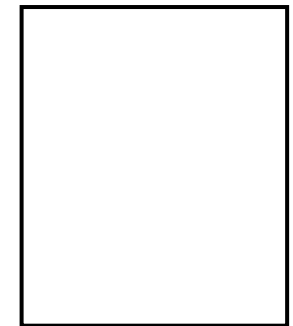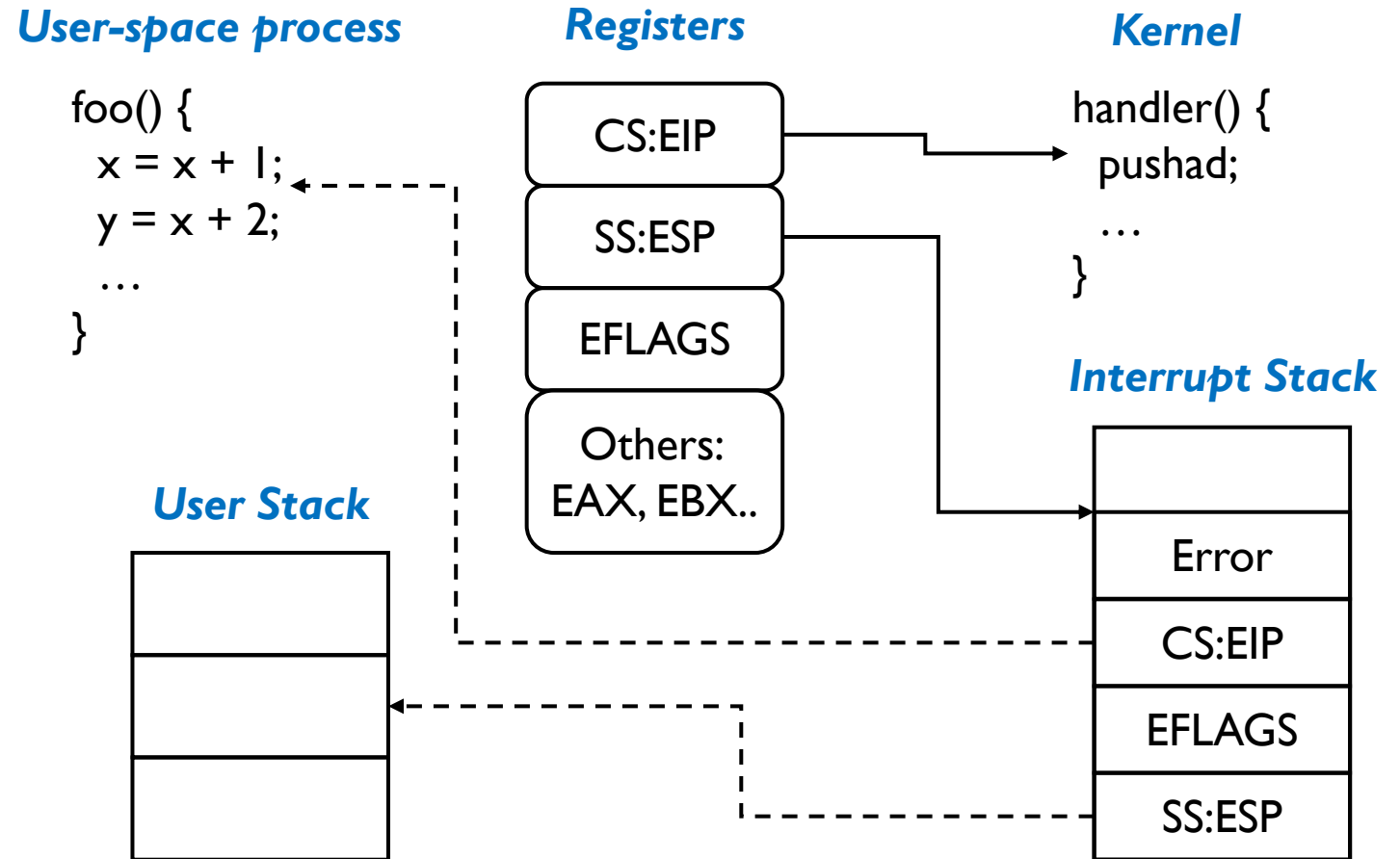
```
foo() {
  x = x + 1;
  y = x + 2;
  …
}
```

Registers

CS:EIP
SS:ESP
EFLAGS
Others: EAX, EBX..

Kernel

```
handler() {
  pushad;
  …
}
```

User Stack

Interrupt Stack

Error
CS:EIP
EFLAGS
SS:ESP

At the beginning of handler

x86 Mode Transfer

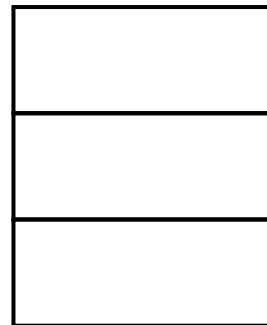- When an interrupt/exception/syscall occurs, the OS will:

1. Save the rest of the interrupted process's state
   - *pusha*/*pushad*
2. Executes the handler
3. Resume the interrupted process
   - popa/popad + pop error code
4. Resume the interrupted process
   - *iret*

User-space process

```
foo() {
  x = x + 1;
  y = x + 2;
  …
}
```

Registers
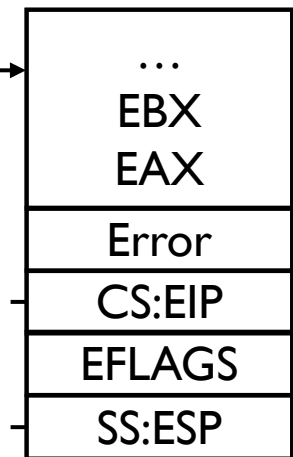
- CS:EIP
- SS:ESP
- EFLAGS
- Others: EAX, EBX..

Kernel

```
handler() {
  pushad;
  …
}
```

Interrupt Stack

User Stack

...
EBX
EAX
Error
CS:EIP
EFLAGS
SS:ESP

During interrupt handler

x86 Mode Transfer

- Who modifies the CPL (2 bits in CS)? Instructions like:
  - int/SYSCALL
  - iret

```
1  #define _GNU_SOURCE
2  #include <unistd.h>
3  #include <sys/syscall.h>
4
5  int main() {
6      const char msg[] = "Hello, Kernel!\n";
7      syscall(SYS_write, STDOUT_FILENO, msg, sizeof(msg) - 1);
8      return 0;
9  }
```

x86 Mode Transfer

- Who modifies the CPL (2 bits in CS)? Instructions like:
  - int/SYSCALL
  - iret

```c
1  #include <stdio.h>
2
3  int main() {
4      /* Define message and file descriptor */
5      const char msg[] = "Hello, Kernel!\n";
6      int fd = 1;   // STDOUT
7
8      /* Inline assembly to trap into kernel */
9      __asm__ volatile (
10         "movl $4, %%eax;"                /* syscall number for sys_write */
11         "movl %0, %%ebx;"                /* file descriptor */
12         "movl %1, %%ecx;"                /* pointer to message */
13         "movl %2, %%edx;"                /* message length */
14         "int $0x80;"                     /* software interrupt */
15         :                                /* no output */
16         : "g"(fd), "g"(msg), "g"(sizeof(msg) - 1)
17         : "eax", "ebx", "ecx", "edx"
18     );
19
20     return 0;
21 }
```

Goals for Today

- OS Programming Interface

- Case Study: Process Management

- Case Study: Input/Output

- System Calls Design

OS Functions to Apps

- Process management

- Input/output

- Thread management

- Memory management

- File systems and storage

- Networking

- Graphics and window management
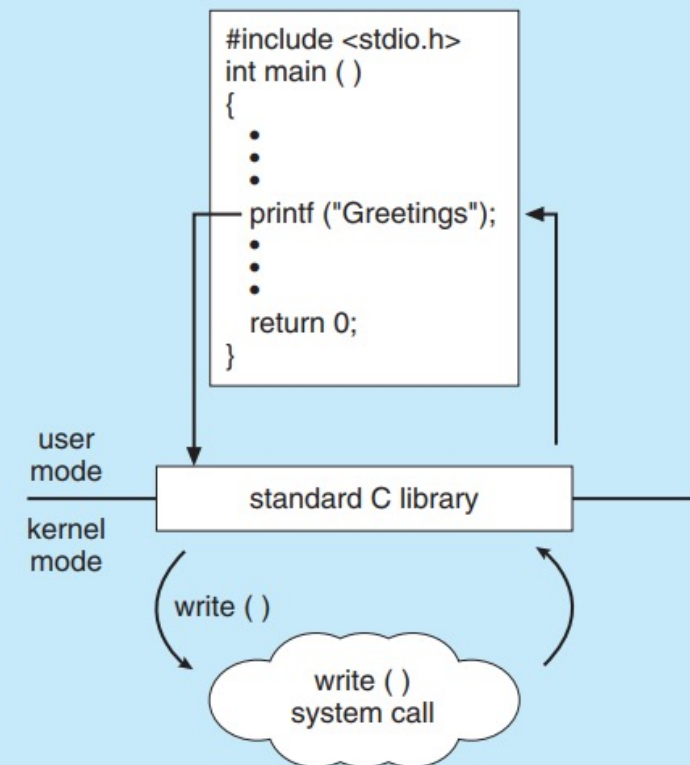
- Authentication and security

OS System Calls

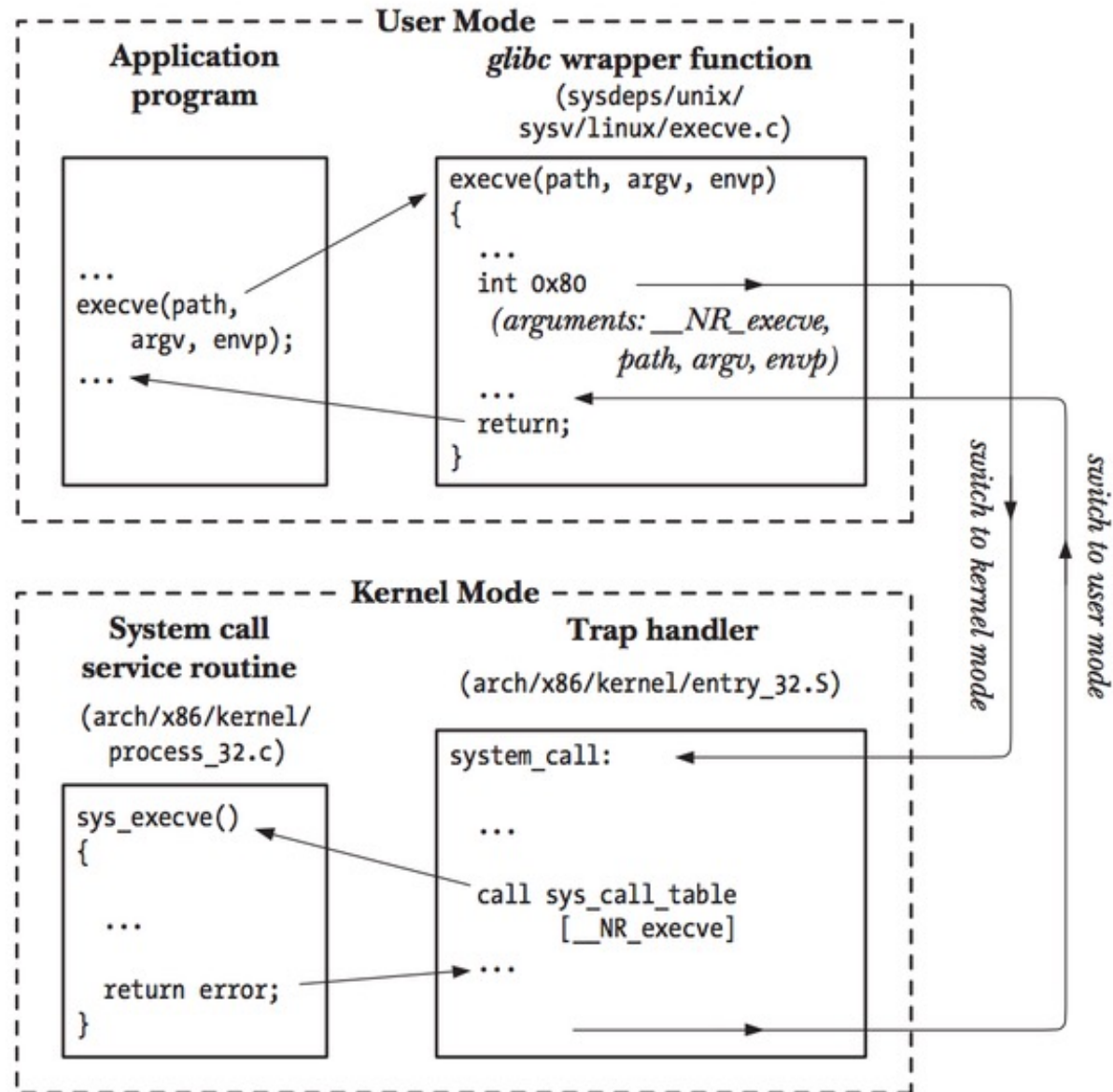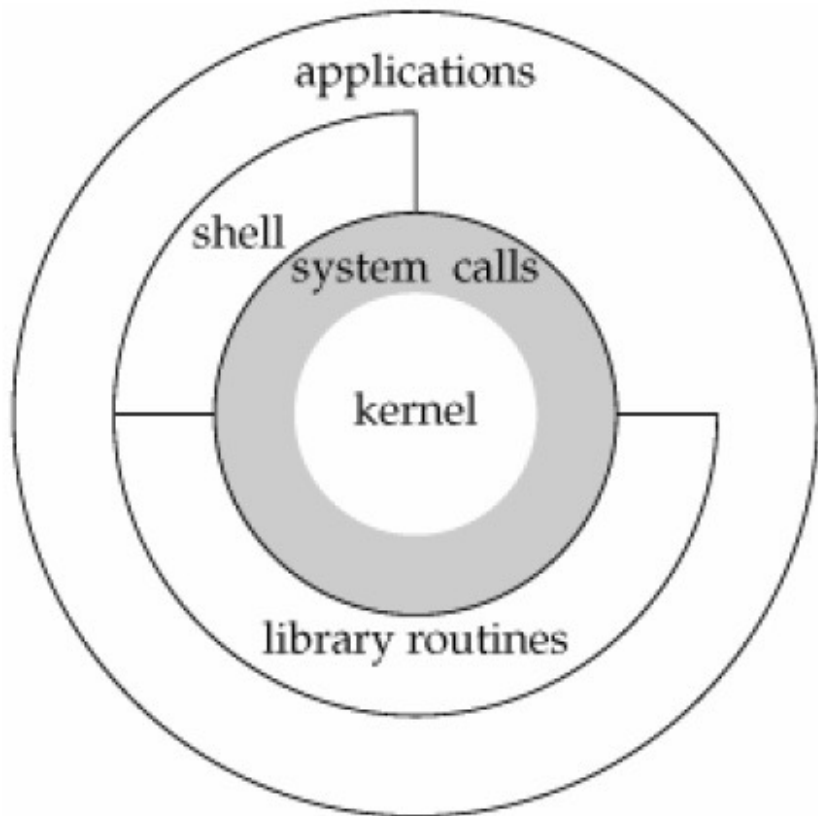|	**Windows**	**Unix**
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

POSIX and libc

- Portable Operating System Interface (POSIX)
  - A standard for UNIX OSes, especially its system calls

- **libc**: overview of standard C libraries
  - POSIX APIs like *getpid()* + standard C functions like *strcpy()*
  - Apps do not directly invoke syscalls
  - glibc: GNU C library

- If a software is written with only dependency to libc, it has good *portability* across OSes/hardware

EXAMPLE OF STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program. This is shown below:

```
#include <stdio.h>
int main ( )
{
    .
    .
    .
    printf ("Greetings");
    .
    .
    .
    return 0;
}
```

user mode / kernel mode

standard C library

write ( )

write ( ) system call

POSIX and libc

POSIX and libc

POSIX-certified [edit]

Current versions of the following operating systems have been certified to conform to one or more of the various POSIX standards. This means that they passed the automated conformance tests[17] and their certification has not expired and the operating system has not been discontinued.[18][19]

- AIX[20]
- HP-UX[21]
- INTEGRITY[22]
- macOS (since 10.5 Leopard)[23][24]
- OpenServer[25]
- UnixWare[26]
- VxWorks[22]
- z/OS[20]

Formerly POSIX-certified [edit]

Some versions of the following operating systems had been certified to conform to one or more of the various POSIX standards. This means that they passed the automated conformance tests. The certification has expired and some of the operating systems have been discontinued.[18]

- EulerOS (exp. 2022)[27]
- Inspur K-UX (exp. 2019)[28]
- IRIX (defunct 2006)[29]
- OS/390 (defunct 2004)[30]
- QNX Neutrino[31]
- Solaris (exp. 2019)[32]
- Tru64 (defunct 2010)[33]

Mostly POSIX-compliant [edit]

The following are not certified as POSIX compliant yet comply in large part:

- Android (Available through Android NDK) [citation needed]
- BeOS (and subsequently Haiku)
- Contiki
- Darwin (core of macOS and iOS)
- DragonFly BSD
- FreeBSD[34]
- illumos
- Linux (most distributions)
- LynxOS
- MINIX (now MINIX3)

- MPE/iX[35]
- NetBSD
- Nucleus RTOS
- NuttX
- OpenBSD
- OpenSolaris[36]
- PikeOS RTOS for embedded systems with optional PSE51 and PSE52 partitions; see partition (mainframe)
- PX5 RTOS[37]
- Redox

- RTEMS – POSIX API support designed to IEEE Std. 1003.13-2003 PSE52
- SerenityOS
- Stratus OpenVOS[38]
- SkyOS
- Syllable
- ULTRIX[39]
- VSTa
- VMware ESXi
- Xenix
- Zephyr[40]

https://en.wikipedia.org/wiki/POSIX

POSIX and libc

```
1  #include <stdio.h>
2  int main() {
3      printf("Hello, World!\n");
4      return 0;
5  }
```

✅ Linux, MacOS, Windows

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4  #include <sys/syscall.h>
5
6  int main() {
7      pid_t tid = syscall(SYS_gettid);
8      printf("Thread ID: %d\n", tid);
9      return 0;
10 }
```

✅ Linux
❌ MacOS, Windows
• gettid: Linux-specific function

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4
5  int main() {
6      pid_t pid = fork();
7      if (pid == 0) {
8          printf("I am the child process.\n");
9      } else if (pid > 0) {
10         printf("I am the parent process.\n");
11     } else {
12         printf("Fork failed.\n");
13     }
14     return 0;
15 }
```

✅ Linux, MacOS
❌ Windows
• Unistd.h: Unix standard header

This is for compilation level. What if they are already compiled..?

The Design Considerations

- Flexibility

- Safety

- Reliability

- Performance
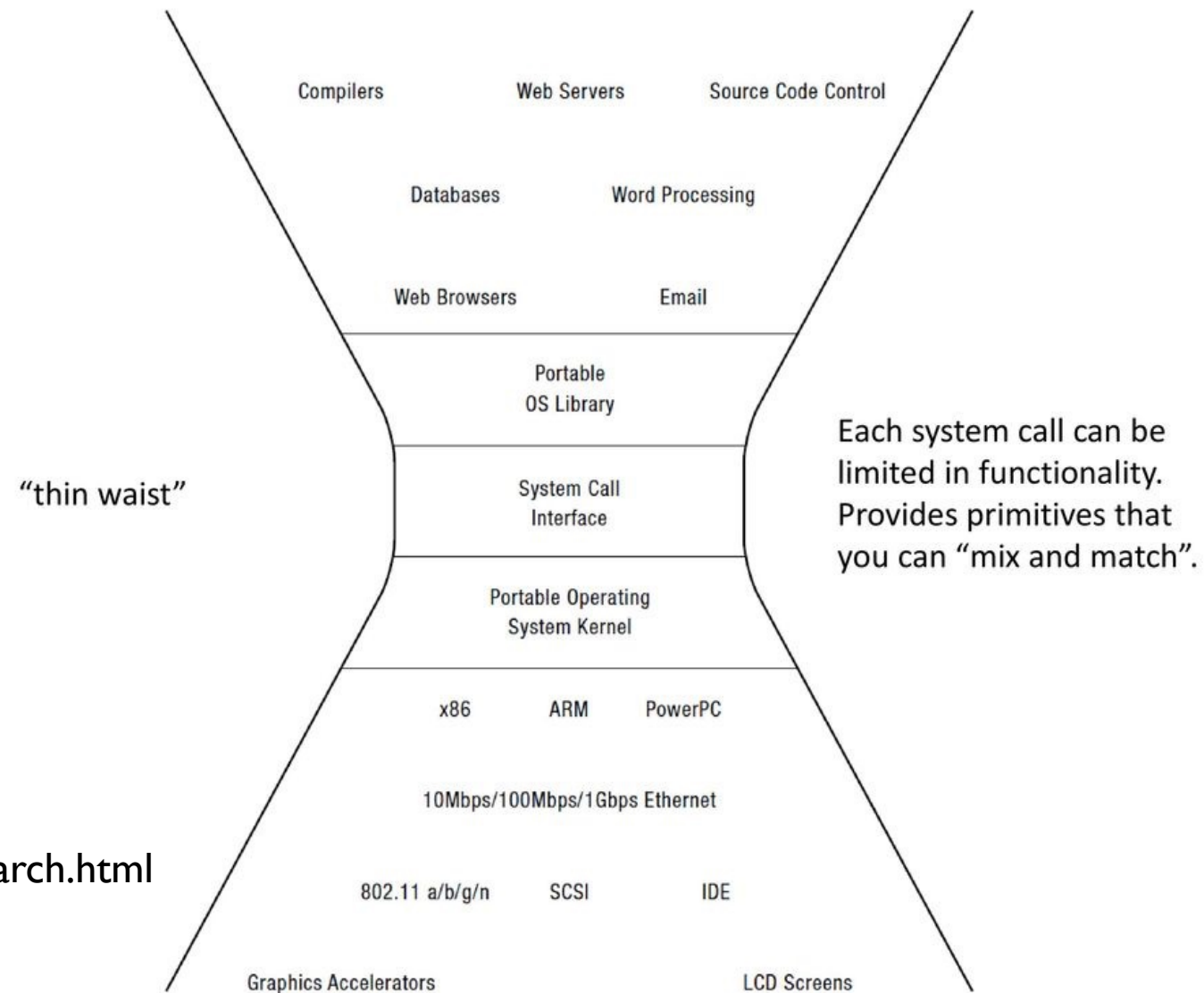
"thin waist"

Compilers   Web Servers   Source Code Control

Databases   Word Processing

Web Browsers   Email

Portable OS Library

System Call Interface

Each system call can be limited in functionality. Provides primitives that you can "mix and match".

Portable Operating System Kernel

x86   ARM   PowerPC

10Mbps/100Mbps/1Gbps Ethernet

802.11 a/b/g/n   SCSI   IDE

Graphics Accelerators   LCD Screens

https://www.oilshell.org/blog/2022/03/backlog-arch.html

Goals for Today

- OS Programming Interface
- Case Study: Process Management
- Case Study: Input/Output
- System Calls Design

The Need for Multi-process

- Early motivation: allow developers to write their own shell command line interpreters

> # shell script
> cc –c sourcefile1.c
> cc –c sourcefile1.c
> ln –o program sourcefile1.o sourcefile2.o

Process in Windows

Boolean CreateProcess(char *prog, char *args)

- Create and initialize the process control block (PCB) in kernel
- Create and initialize a new memory address space
- Load the program prog into the address space
- Copy arguments args into memory in the address space
- Initialize the hardware context to start execution at "start"
- Inform the scheduler that the new process is ready to run

- In reality, it's a bit more complex
  - The parent process (父进程) may specify the child process's (子进程) privileges, where it sends its input and output, what it should store its files, what to use as a scheduling priority, etc.

https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa

fork() in Unix

- fork() and exec(): the Unix way to create new processes
    - Perhaps one of the most controversial design in Unix

SYNOPSIS   top

```
#include <unistd.h>

pid_t fork(void);
```

fork(): create a complete copy of the parent process, except the return value:
- 0 for child process
- The PID of child process for the parent process

SYNOPSIS   top

```
#include <unistd.h>

int execve(const char *pathname, char *const argv[],
                char *const envp[]);
```

exec(): load and execute a program from disk

Note: exec() does not create a new process!

fork() in Unix

- What actually have fork() and exec() done

<table>
<tr>
<td>

fork()
1. Create and initialize PCB
2. Create a new address space
3. Copy the entire memory contents from parent process to the child
4. Inherit the execution content of the parent (e.g., open files)
5. Inform the scheduler that new process is ready to run

</td>
<td>

exec(char *prog, char *args)
1. Load the program prog into the current address space
2. Copy arguments args into memory in the address space
3. Initialize the hardware context to start execution at "start"

</td>
</tr>
</table>

fork() in Unix

- A typical example of how fork() and exec() are used

```
int pid = fork();
if (pid == 0) {
    exec("foo");        ←——————— Child process
} else {
    waitpid(pid, &status, options);
};
```

Parent process

- The memory contents of the child process are copied twice, would that be a waste?

fork() in Unix

- exec() is not always necessary
  - Opens a new page in Google Chrome

- <u>wait(pid):</u> wait for the child process to finish execution
- <u>signal</u>: terminate, stop, resume a process

Some Simple fork() Quizzes

1. How many "OS" printed?

```
int main() {
  fork();
  fork();
  fork();
  printf("OS ");
  return 0;
}
```

2. How many "OS" printed?

```
int main() {
  if (fork() || fork())
    fork();
  printf("OS ");
  return 0;
}
```

3. What is the output
A. I am child, I am parent
B. I am parent, I am child
C. Both are possible

```
int main() {
  int pid = fork();
  if (pid == 0) {
    printf("I am child, ");
  } else {
    printf("I am parent, ");
    return 0;
  }
}
```

4. What are the possible output

```
int main() {
  for (int i = 0; i < 3;
i += 1) {
    pid_t p = fork();
    if (p == 0) {
      i += 1;
    }
    printf("%d", i);
  }
  return 0;
}
```

Some Simple fork() Quizzes

1. How many "OS" printed?
- **<u>8</u>**
```c
int main() {
  fork();
  fork();
  fork();
  printf("OS ");
  return 0;
}
```

2. How many "OS" printed?
- **<u>5</u>**
```c
int main() {
  if (fork() || fork())
    fork();
  printf("OS ");
  return 0;
}
```

3. What is the output
A. I am child, I am parent
B. I am parent, I am child
C. <u>Both are possible</u>

```c
int main() {
  int pid = fork();
  if (pid == 0) {
    printf("I am child, ");
  } else {
    printf("I am parent, ");
    return 0;
  }
}
```

4. What are the possible output

```c
int main() {
  for (int i = 0; i < 3;
i += 1) {
    pid_t p = fork();
    if (p == 0) {
      i += 1;
    }
    printf("%d", i);
  }
  return 0;
}
```

Goals for Today

- OS Programming Interface

- Case Study: Process Management

- Case Study: Input/Output

- System Calls Design

Input/Output in Unix

- Computer systems have very diverse I/O devices
  - Keyboard: individual characters
  - Disk: fixed-sized chunks
  - Network: stream of variable sized packets
  - Mouse: single events

- Having an interface for each device means the OS interface needs to expand whenever a new device is added..

- Unix has one interface for all of them!
  - "Everything is a file" – open, read, write, close

File Descriptor in Unix

- File Descriptor (fd): a number (int) that uniquely identifies an open file in a computer's operating system. It describes a data resource, and how that resource may be accessed.

- Each process has its own file descriptor table
- A file can be opened multiple times and therefore associated with many file descriptors
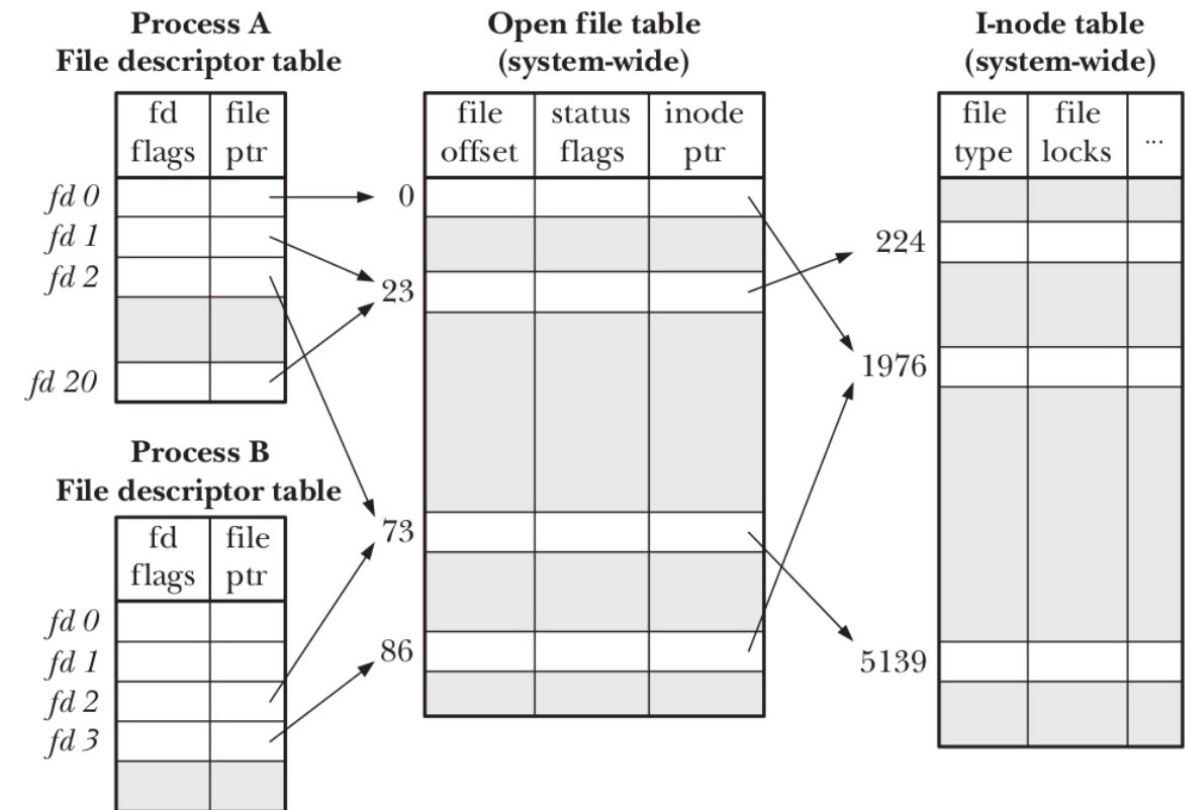- More in filesystem courses

Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

File Descriptor in Unix

- File Descriptor (fd): a number (int) that uniquely identifies an open file in a computer's operating system. It describes a data resource, and how that resource may be accessed.

ls –l /proc/[pid]/fd


```
[root@server-11-170 sentinel]# ls -l /proc/96104/fd
total 0
l-wx------. 1 root root 64 Nov 30 00:05 0 -> /dev/null
l-wx------. 1 root root 64 Nov 30 00:05 1 -> /etc/redis/cluster/nohup.out
lrwx------. 1 root root 64 Nov 30 00:05 10 -> socket:[1003439]
lrwx------. 1 root root 64 Nov 30 00:05 11 -> socket:[1003412]
lrwx------. 1 root root 64 Nov 30 00:05 12 -> socket:[1003413]
lrwx------. 1 root root 64 Nov 30 00:05 13 -> socket:[1003419]
lrwx------. 1 root root 64 Nov 30 00:05 14 -> socket:[1003420]
lrwx------. 1 root root 64 Nov 30 00:05 15 -> socket:[1003421]
lrwx------. 1 root root 64 Nov 30 00:05 16 -> socket:[1001448]
lrwx------. 1 root root 64 Nov 30 00:05 17 -> socket:[1001450]
lrwx------. 1 root root 64 Nov 30 00:05 18 -> socket:[1001452]
lrwx------. 1 root root 64 Nov 30 00:05 19 -> socket:[1001458]
l-wx------. 1 root root 64 Nov 24 03:20 2 -> /etc/redis/cluster/nohup.out
lrwx------. 1 root root 64 Nov 30 00:05 20 -> socket:[1003441]
l-wx------. 1 root root 64 Nov 30 00:05 29 -> /root/appendonly-7005.aof
lr-x------. 1 root root 64 Nov 30 00:05 3 -> pipe:[1002939]
l-wx------. 1 root root 64 Nov 30 00:05 4 -> pipe:[1002939]
lrwx------. 1 root root 64 Nov 30 00:05 5 -> anon_inode:[eventpoll]
lrwx------. 1 root root 64 Nov 30 00:05 6 -> socket:[1002942]
l-wx------. 1 root root 64 Nov 30 00:05 8 -> /root/nodes-7005.conf
lrwx------. 1 root root 64 Nov 30 00:05 9 -> socket:[1003700]
```

File Descriptor in Unix

- Internally, it has everything about an opened file
  - Where it resides
  - Its status
  - How to access it
  - ..

```
                                    github.com/torvalds/linux/blob/master/include/linux/fs.h
checkpoint   AndroidBlog    bupt    course    funding    research    writing    doc    tmp    TODO    serve
 939
 940     struct file {
 941           union {
 942                   struct llist_node        f_llist;
 943                   struct rcu_head          f_rcuhead;
 944                   unsigned int             f_iocb_flags;
 945           };
 946           struct path              f_path;
 947           struct inode             *f_inode;        /* cached value */
 948           const struct file_operations     *f_op;
 949
 950           /*
 951            * Protects f_ep, f_flags.
 952            * Must not be taken from IRQ context.
 953            */
 954           spinlock_t               f_lock;
 955           atomic_long_t            f_count;
 956           unsigned int             f_flags;
 957           fmode_t                  f_mode;
 958           struct mutex             f_pos_lock;
 959           loff_t                   f_pos;
 960           struct fown_struct       f_owner;
 961           const struct cred        *f_cred;
 962           struct file_ra_state     f_ra;
 963
 964           u64                      f_version;
 965  #ifdef CONFIG_SECURITY
 966           void                     *f_security;
 967  #endif
 968           /* needed for tty driver, and maybe others */
 969           void                     *private_data;
```

Input/Output in Unix

- A uniform interface for all I/O
    - Uniformity: open, close, read, and write

```
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);

return value: file descriptor or error code (-1)
pathname: could be a file ("/data/readme.txt") or a
device ("/dev/zero")
```

```
#include <fcntl.h>
int close(int fd);

return value: 0 (success) or -1 (error)
Note: if fd is the last file descriptor referring to the
underlying open file description, the resources
associated with the open file description are freed.
```

Input/Output in Unix

- A uniform interface for all I/O
  - Uniformity: open, close, read, and write

```
#include <fcntl.h>
ssize_t read(int fd, void *buf, size_t count);

It will read up to count bytes from file descriptor fd
into the buffer starting at buf.
return value: the number of bytes read or error (-1)
```

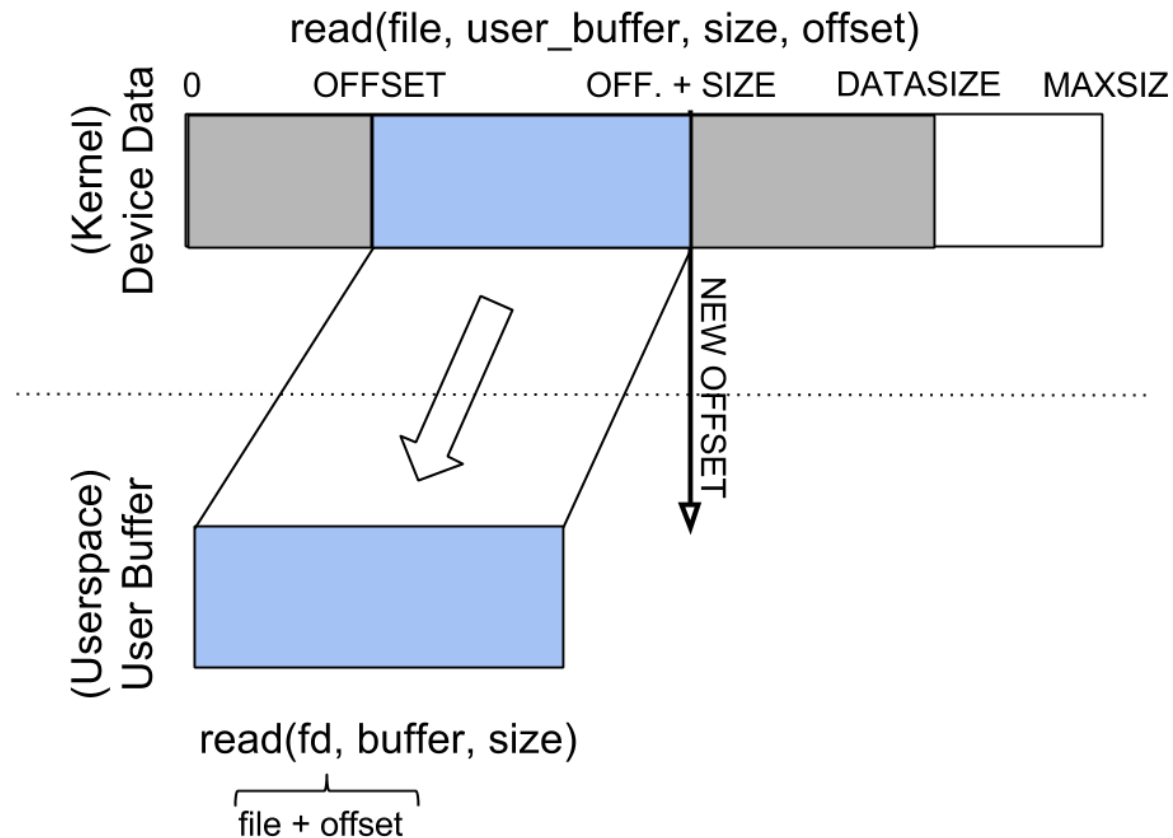```
#include <fcntl.h>
ssize_t write(int fd, const void *buf, size_t count);

It will write up to count bytes from the buffer starting
at buf to the file referred to by the file descriptor fd.
return value: the number of bytes written or -1 (error)
```
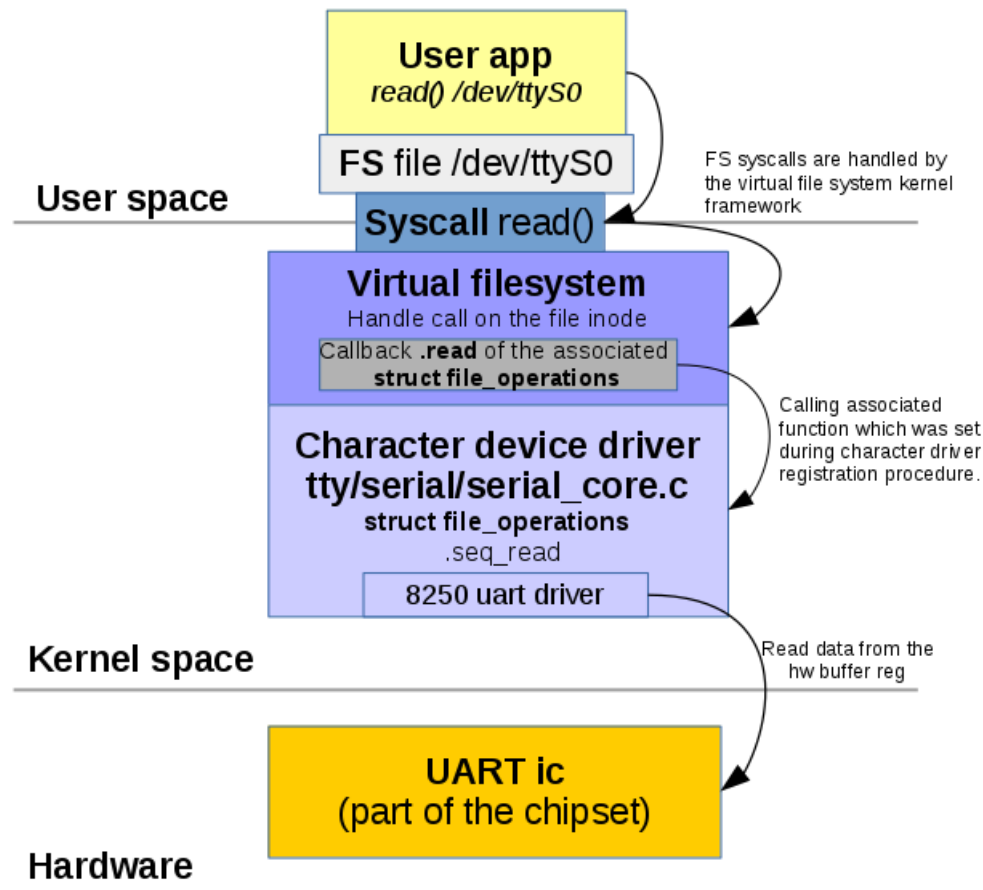
- A uniform interface for all I/O
  - Uniformity: open, close, read, and write



read(file, user_buffer, size, offset)

# Input/Output in Unix

- A uniform interface for all I/O
  - Uniformity: open, close, read, and write



```c
struct file_operations {
  struct module *owner;
  loff_t (*llseek) (struct file *, loff_t, int);
  ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
  ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
  ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
  ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
  int (*iopoll)(struct kiocb *kiocb, bool spin);
  int (*iterate) (struct file *, struct dir_context *);
  int (*iterate_shared) (struct file *, struct dir_context *);
  __poll_t (*poll) (struct file *, struct poll_table_struct *);
  long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
  long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
  int (*mmap) (struct file *, struct vm_area_struct *);
  unsigned long mmap_supported_flags;
  int (*open) (struct inode *, struct file *);
  int (*flush) (struct file *, fl_owner_t id);
  int (*release) (struct inode *, struct file *);
  int (*fsync) (struct file *, loff_t, loff_t, int datasync);
  int (*fasync) (int, struct file *, int);
  int (*lock) (struct file *, int, struct file_lock *);
  ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *,
  unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned
  int (*check_flags)(int);
  int (*flock) (struct file *, int, struct file_lock *);
  ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
  ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
  int (*setlease)(struct file *, long, struct file_lock **, void **);
  long (*fallocate)(struct file *file, int mode, loff_t offset,
          loff_t len);
  void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
  unsigned (*mmap_capabilities)(struct file *);
#endif
  ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
      loff_t, size_t, unsigned int);
  loff_t (*remap_file_range)(struct file *file_in, loff_t pos_in,
          struct file *file_out, loff_t pos_out,
          loff_t len, unsigned int remap_flags);
  int (*fadvise)(struct file *, loff_t, loff_t, int);
} __randomize_layout;
```

Input/Output in Unix

- A uniform interface for all I/O
  - Uniformity: open, close, read, and write
  - Open before use
    - ❑ OS can check permission and do bookkeeping

Input/Output in Unix

- A uniform interface for all I/O
  - Uniformity: open, close, read, and write
  - Open before use
  - Byte-oriented
    - ❑ Even if blocks are transferred, addressing is in bytes

Input/Output in Unix

- A uniform interface for all I/O
  - Uniformity: open, close, read, and write
  - Open before use
  - Byte-oriented
  - Kernel-buffered reads/writes
    - ❑ Streaming and block devices looks the same
    - ❑ Read blocks process, yielding processor to other task
    - ❑ Write does not block (even if it's faster than device receiving data)

Input/Output in Unix

- A uniform interface for all I/O
  - Uniformity: open, close, read, and write
  - Open before use
  - Byte-oriented
  - Kernel-buffered reads/writes
  - Explicit close
    - ❑ Garbage collection of unused kernel data structures

Input/Output in Unix

- Extending the interface to inter-process communication
  - Pipes: a kernel buffer with two file descriptors (reading and writing)
  - Replace file descriptor for the child process
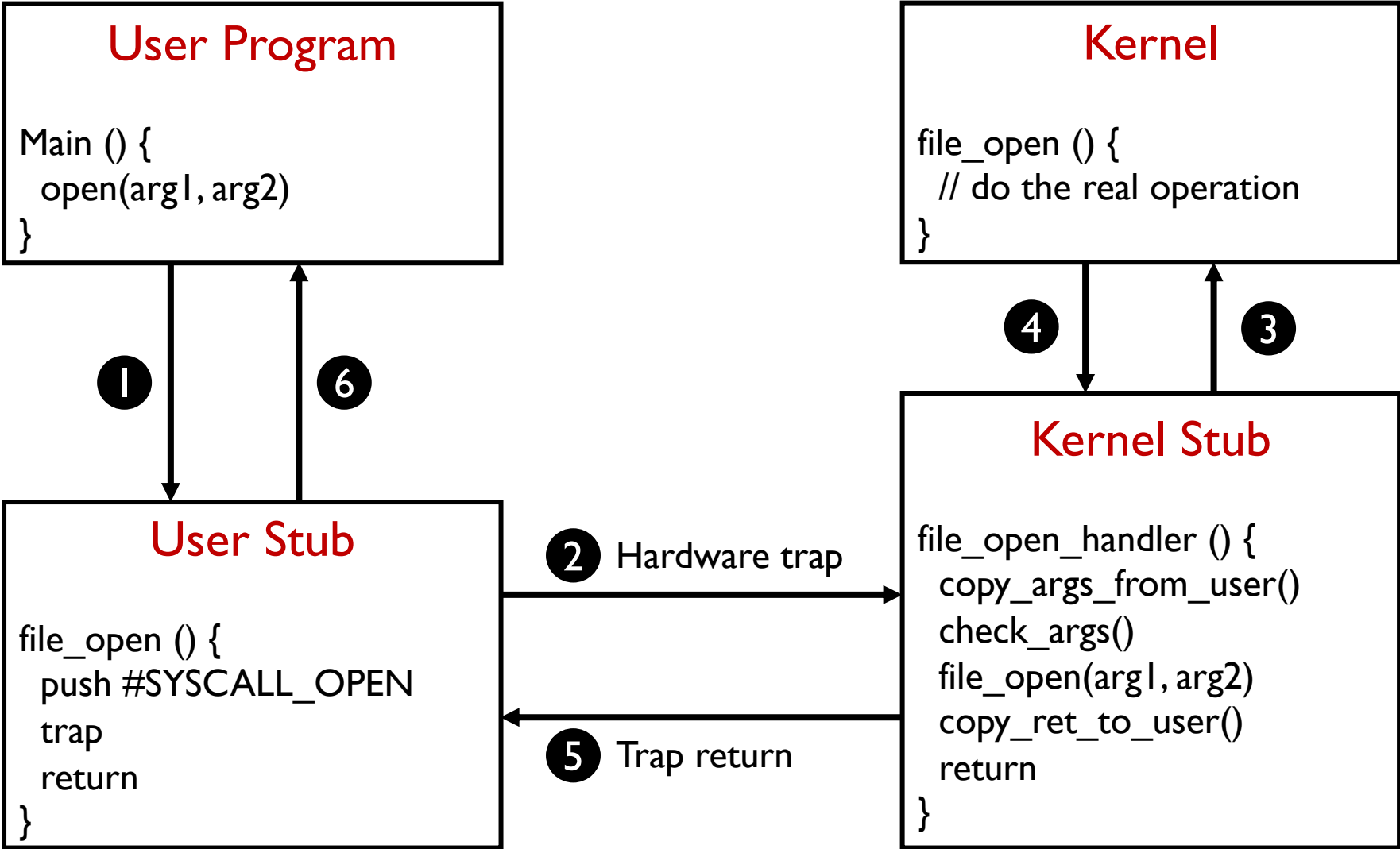   - ❏ Often used in shells
  - Wait for multiple reads
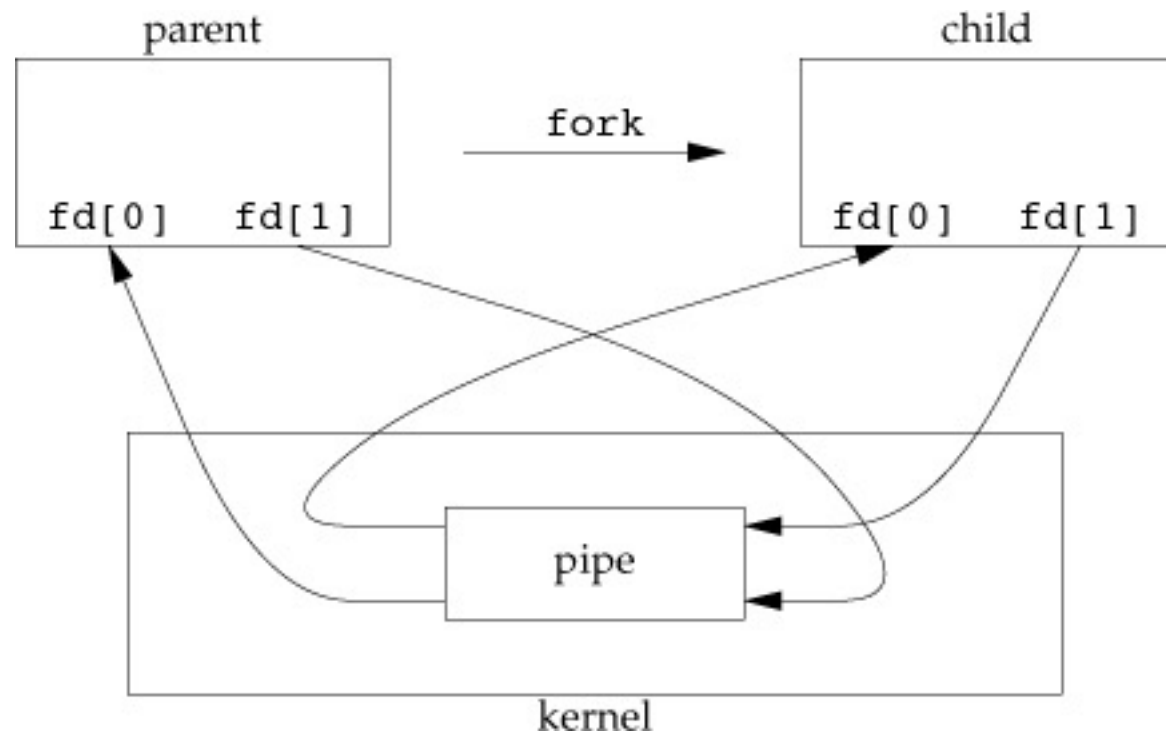
Goals for Today

- OS Programming Interface

- Case Study: Process Management

- Case Study: Input/Output

- System Calls Design

System Calls

- An illusion that kernel is simply a set of library routines
  - Actually, it's not.. They are not even in the same context!
  - Names, arguments, return values

- A key challenge: protection from user-space errors
  - What are to be checked?

System Calls Stubs

User Program

```
Main () {
  open(arg1, arg2)
}
```

Kernel

```
file_open () {
  // do the real operation
}
```

User Stub

```
file_open () {
  push #SYSCALL_OPEN
  trap
  return
}
```

Kernel Stub

```
file_open_handler () {
  copy_args_from_user()
  check_args()
  file_open(arg1, arg2)
  copy_ret_to_user()
  return
}
```

❶

❻

❷ Hardware trap

❺ Trap return

❹

❸

System Calls Stubs

In x86:

open:
// Put the code for the syscall we want into %eax
    movl #SysCallOpen, %eax
// Trap into the kernel
    int #TrapCode
// Return to the caller; the kernel puts the return value in
%eax already
    ret

User Stub

```
file_open () {
  push #SYSCALL_OPEN
  trap
  return
}
```

The *int* instruction:
- Saves the program counter, stack pointer, and eflags on the kernel stack
- Jumps to the system call handler through interrupt vector table
- The kernel handler examines the TrapCode and calls the correct stub

System Calls Stubs

- Can kernel directly access the parameters without copying?

- Why parameters must be copied from user memory to kernel memory?

- Can we check parameters before copying them to kernel memory?

<div style="border: 1px solid black;">

Kernel Stub

```
file_open_handler () {
  copy_args_from_user()
  check_args()
  file_open(arg1, arg2)
  copy_ret_to_user()
  return
}
```

</div>

System Calls Stubs

- Can kernel directly access the parameters without copying?
    - Yes in most OSes, because kernel and user share memory space
- Why parameters must be copied from user memory to kernel memory?
    - Original parameters are stored in user memory stack
    - *copy_from_user* and *copy_to_usr*
- Can we check parameters before copying them to kernel memory?
    - time of check vs. time of use (TOCTOU) attack

Kernel Stub

```
file_open_handler () {
  copy_args_from_user()
  check_args()
  file_open(arg1, arg2)
  copy_ret_to_user()
  return
}
```