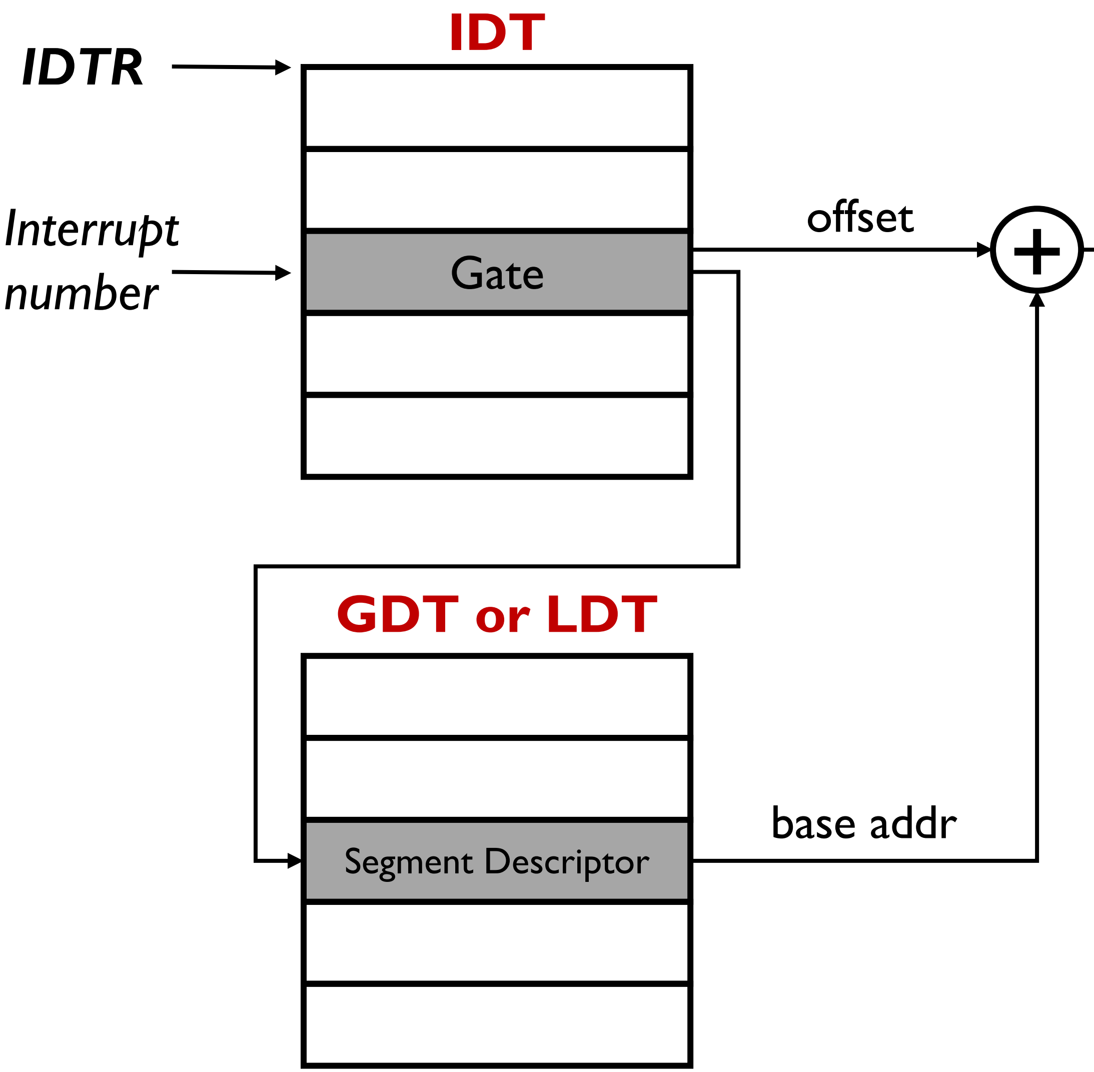
Operating Systems
Lecture 4

OS Interfaces and Syscalls

August 22, 2022

Prof. Mengwei Xu

Recap of Last Course




```c
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

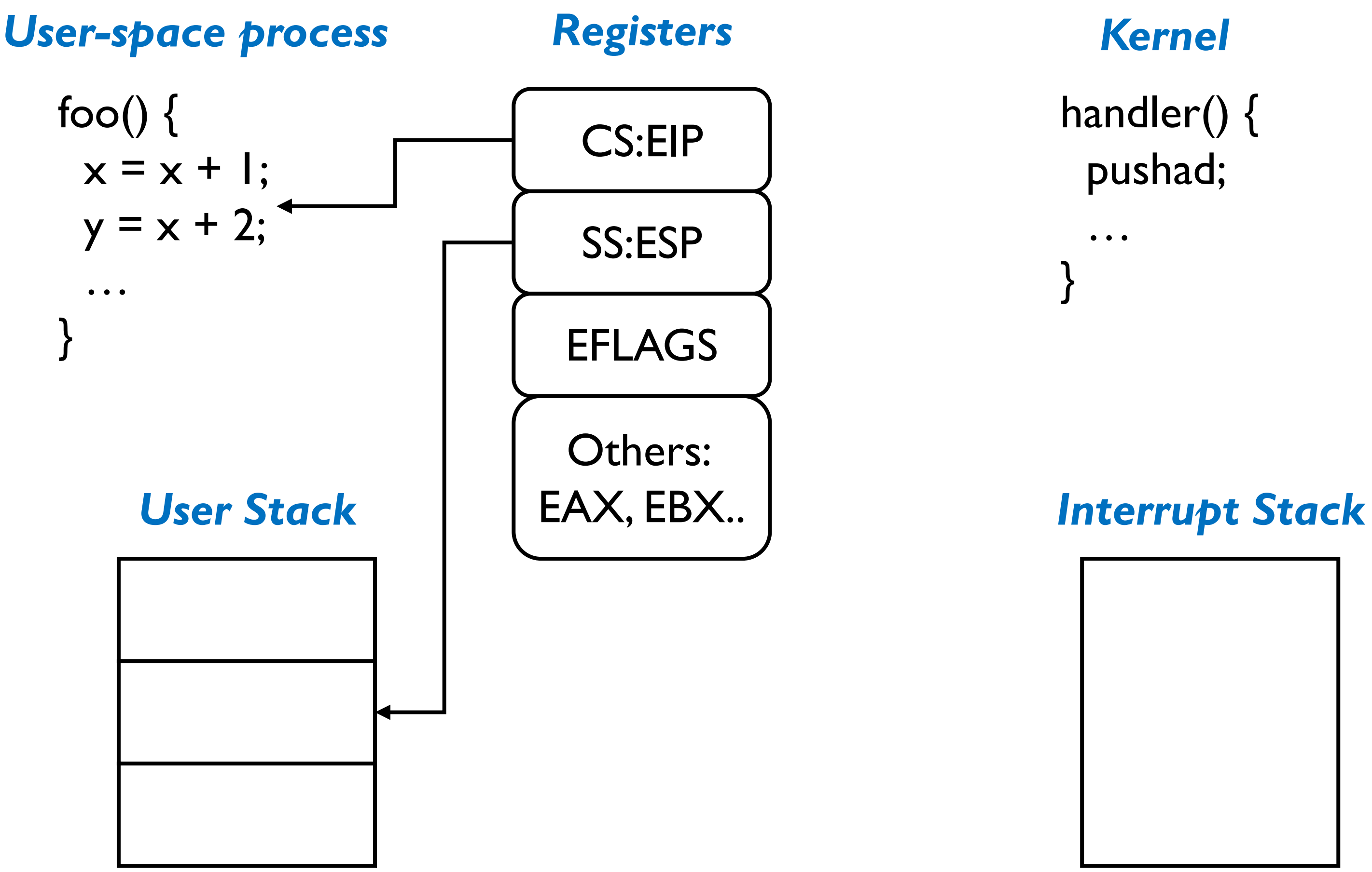
Recap of Last Course

- 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

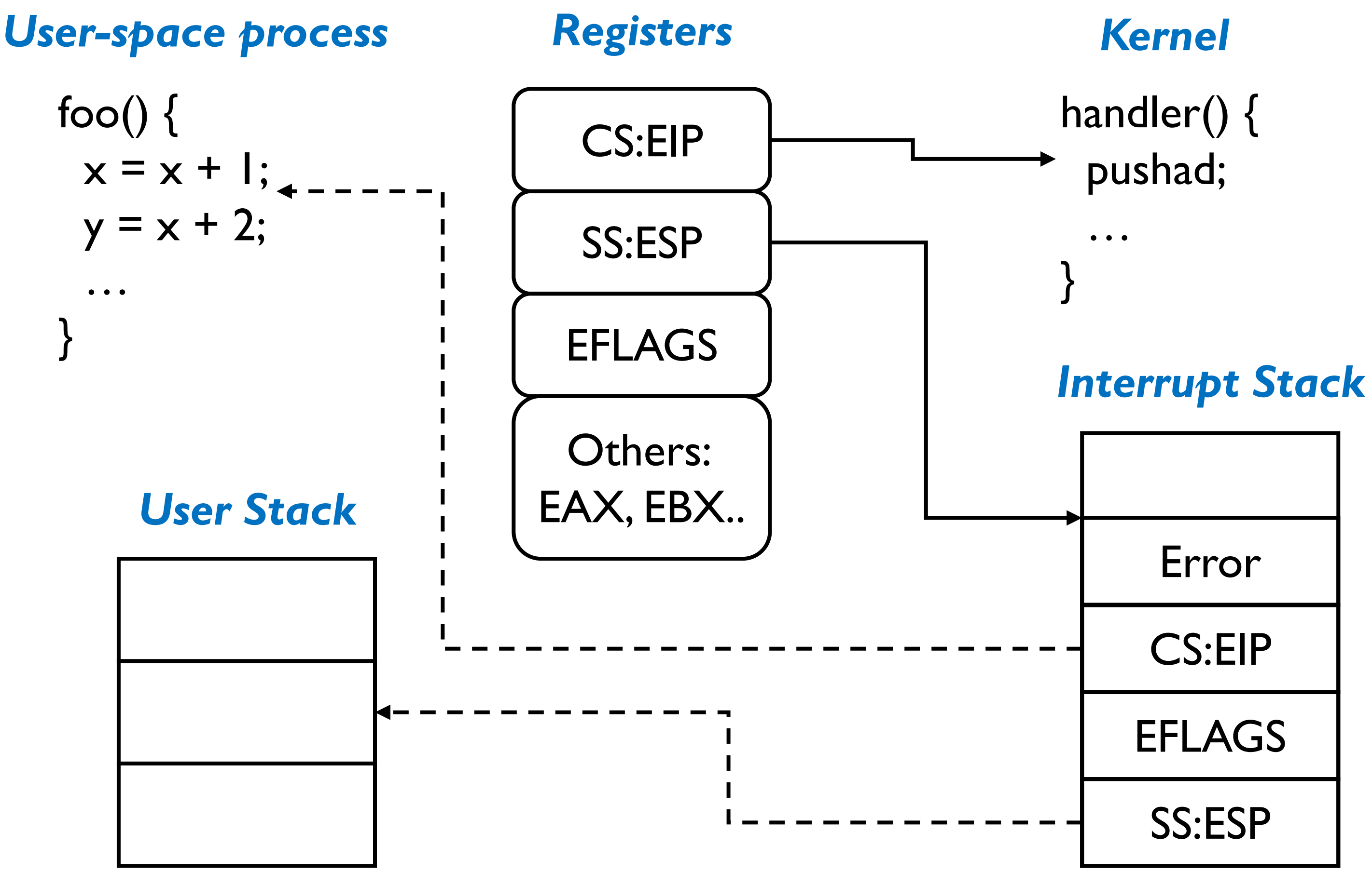


Before interrupt

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

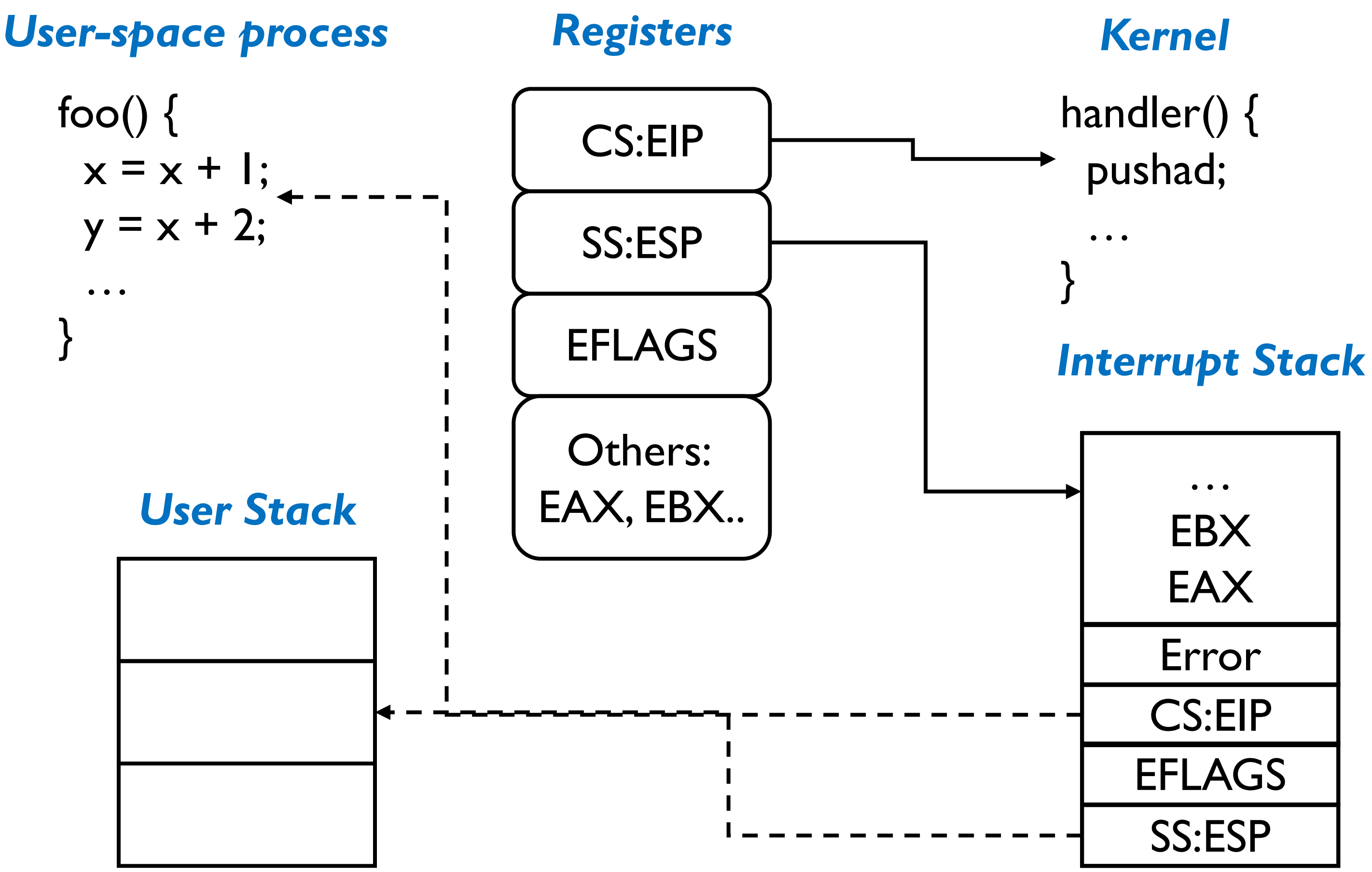


At the beginning of handler

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*



During interrupt handler

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

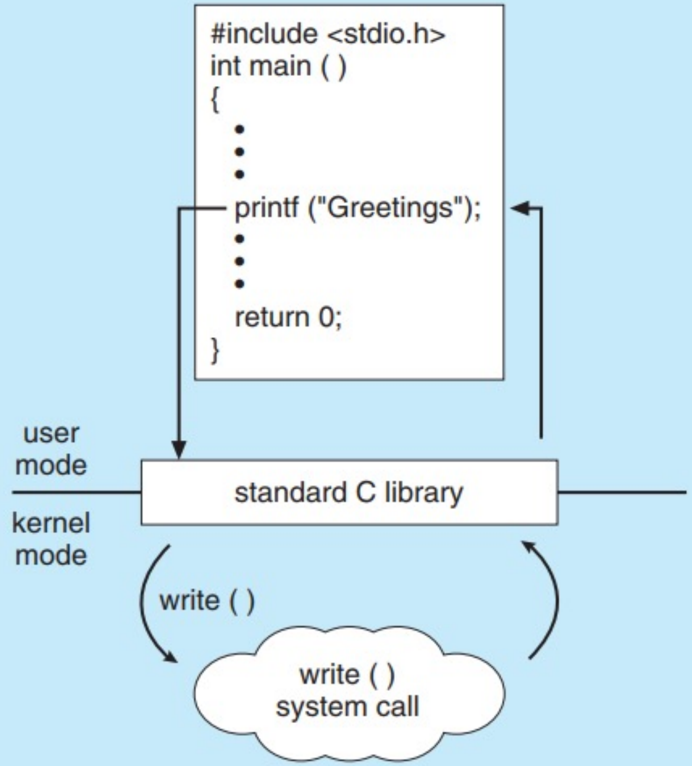
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information 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

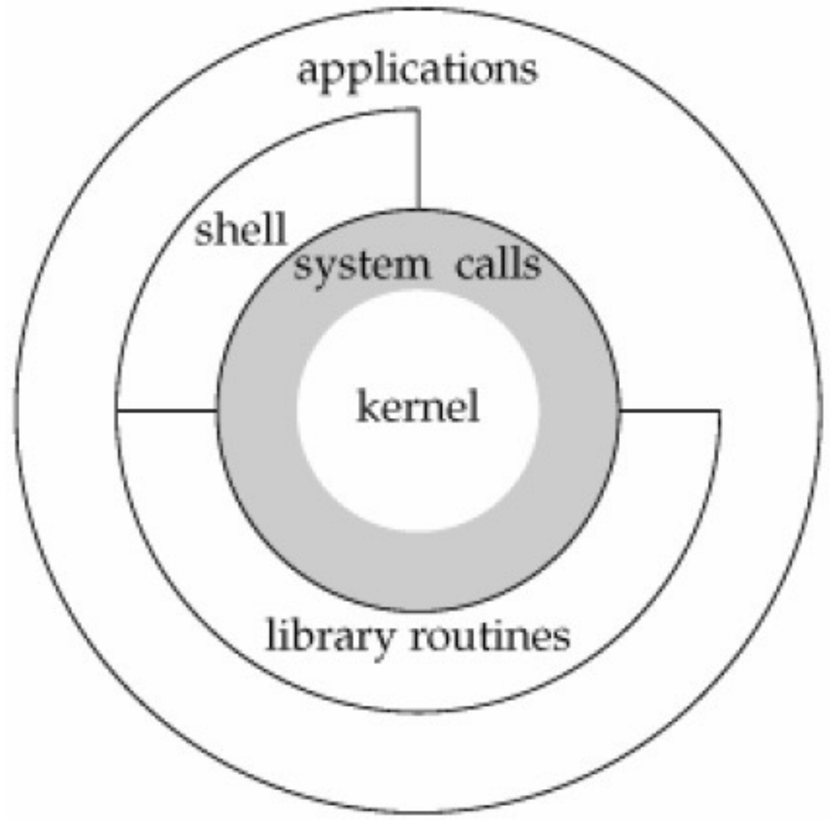
- Portable Operating System Interface (POSIX)
  - A standard for UNIX OSes, especially its system calls
- libc: overview of standard C libraries on Linux
  - POSIX APIs + standard C functions like *strcpy()*
  - Apps do not directly invoke syscalls
  - glibc

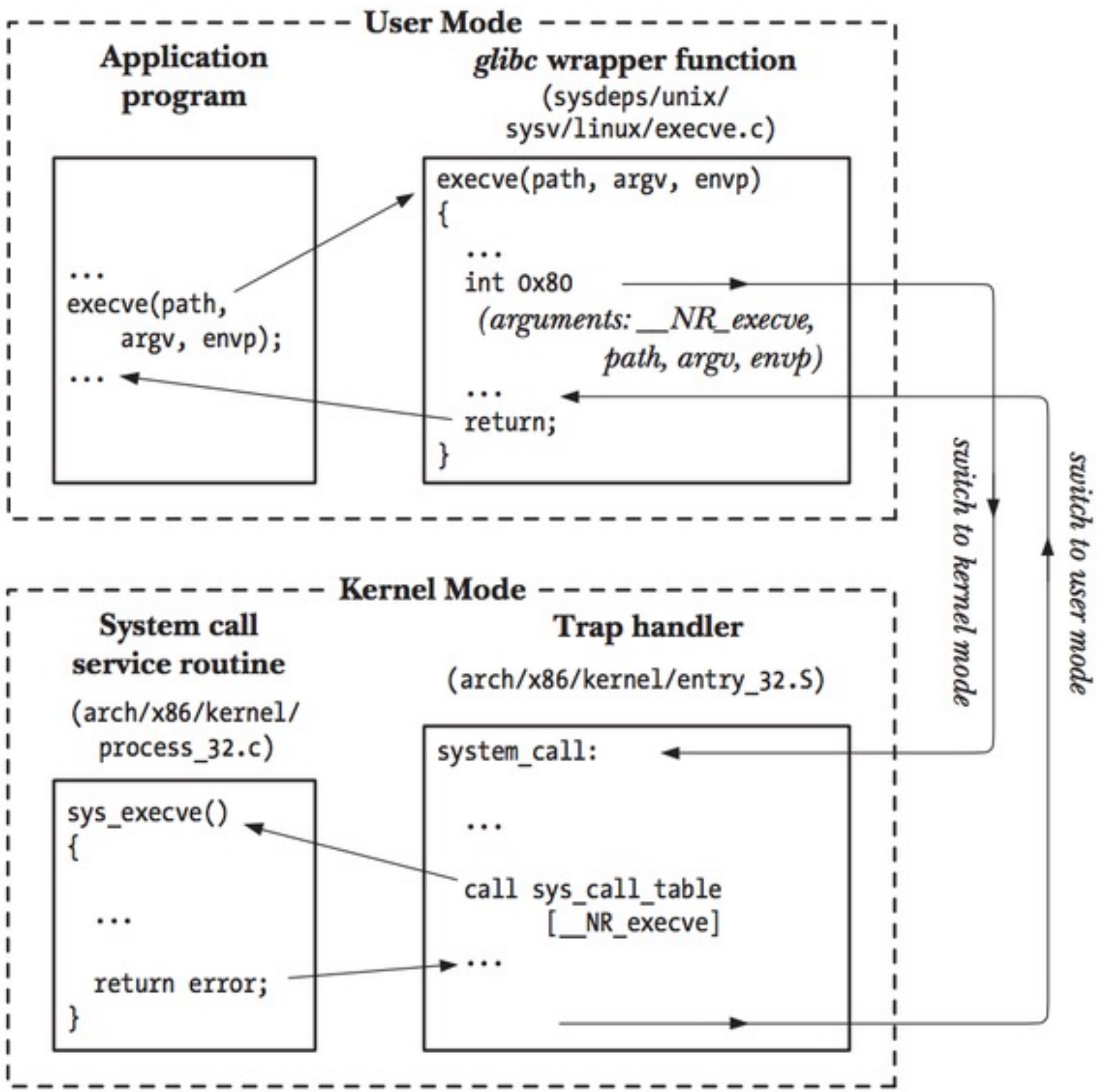
EXAMPLE OF STANDARD C LIBRARY

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:



POSIX and libc





The Ways to Deliver

- In user-level program
- In user-level library
- In kernel, accessed through syscalls
- In a standalone user-mode system process, invoked through syscalls

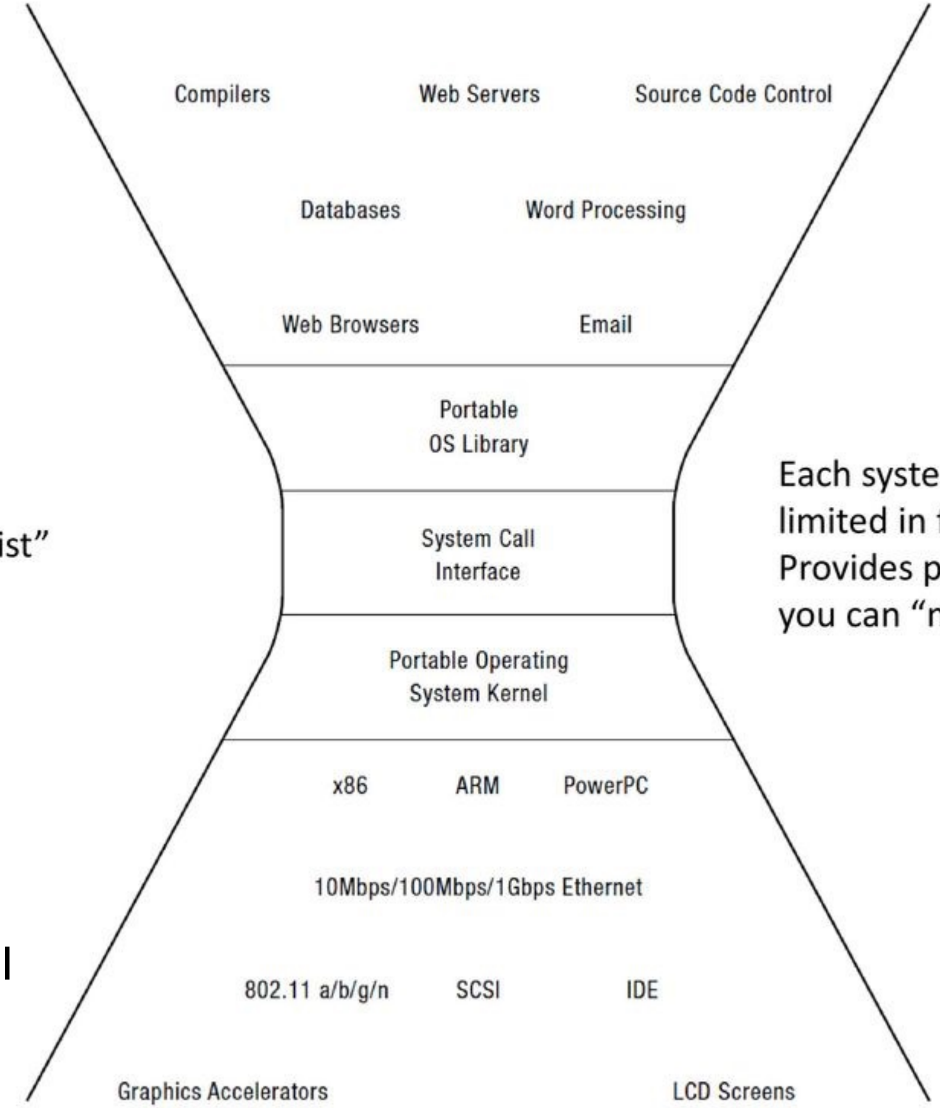
The Ways to Deliver

- In user-level program
  - User login app and control panel app
- In user-level library
  - UI widgets
- In kernel, accessed through syscalls
  - Network stack, memory management
- In a standalone user-mode system process, invoked through syscalls
  - Window manager
  - Extensively used in Android (service)

The Design Considerations

- Flexibility
- Safety
- Reliability
- Performance

“thin waist”



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

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

fork()	exec(char *prog, char *args)
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	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"

fork() in Unix

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

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

Child process

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

- wait(pid): wait for the child process to finish execution
- signal: 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;
}
```

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"

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

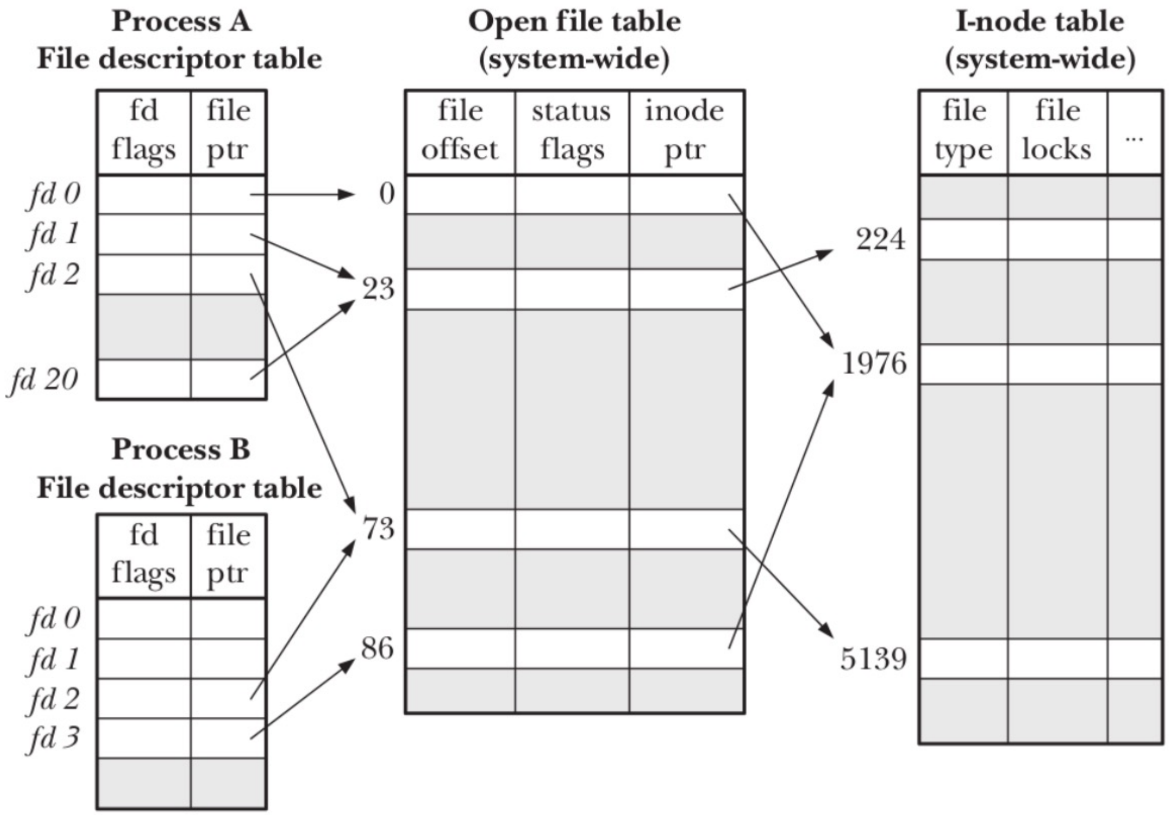


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

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]
[root@server-11-170 sentinel]#
```

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

```c
struct file {
	union {
		struct llist_node	f_llist;
		struct rcu_head		f_rcuhead;
		unsigned int		f_iocb_flags;
	};
	struct path		f_path;
	struct inode		*f_inode;	/* cached value */
	const struct file_operations	*f_op;

	/*
	 * Protects f_ep, f_flags.
	 * Must not be taken from IRQ context.
	 */
	spinlock_t		f_lock;
	atomic_long_t		f_count;
	unsigned int		f_flags;
	fmode_t			f_mode;
	struct mutex		f_pos_lock;
	loff_t			f_pos;
	struct fown_struct	f_owner;
	const struct cred	*f_cred;
	struct file_ra_state	f_ra;

	u64			f_version;
#ifdef CONFIG_SECURITY
	void			*f_security;
#endif
	/* needed for tty driver, and maybe others */
	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

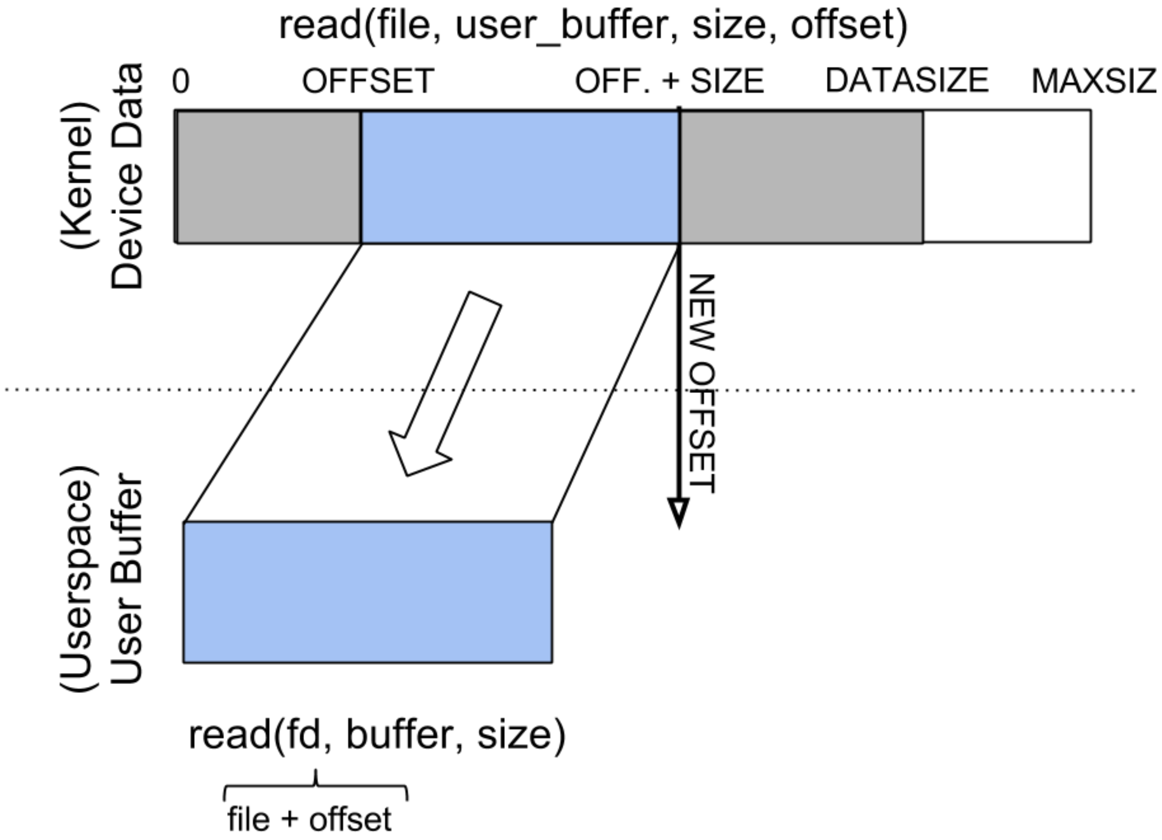
- A uniform interface for all I/O
  - Uniformity: open, close, read, and write

```
#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)
```

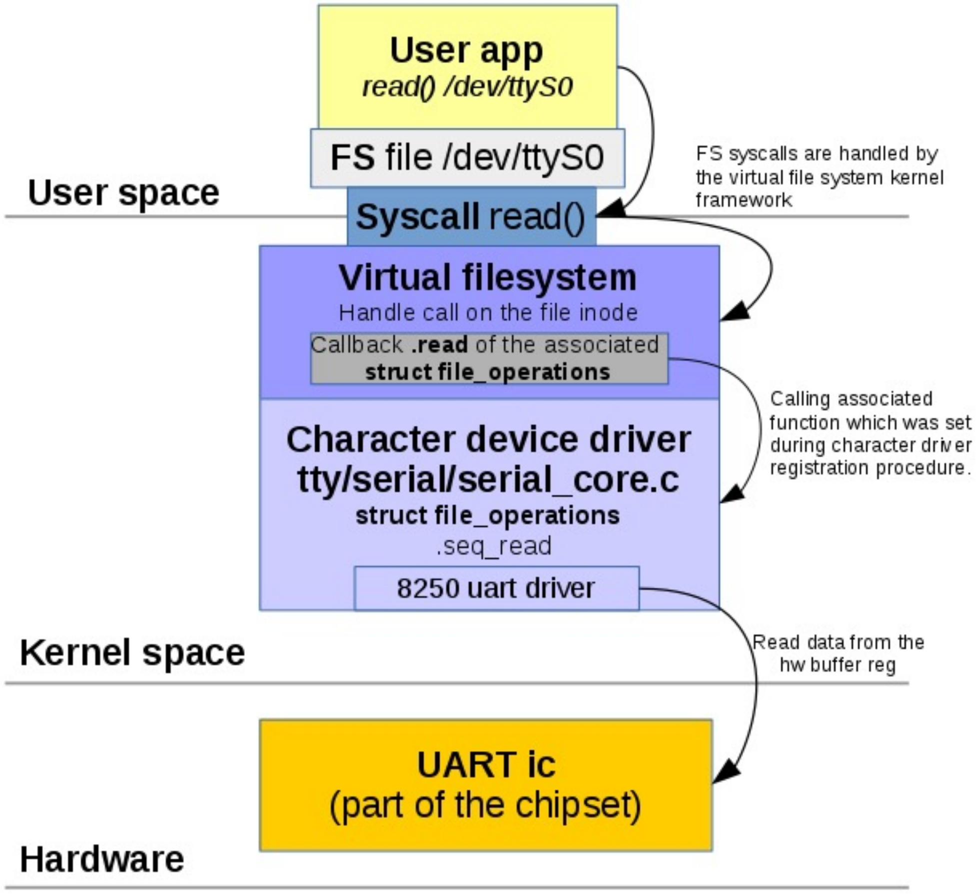
Input/Output in Unix

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



Input/Output in Unix

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




```
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

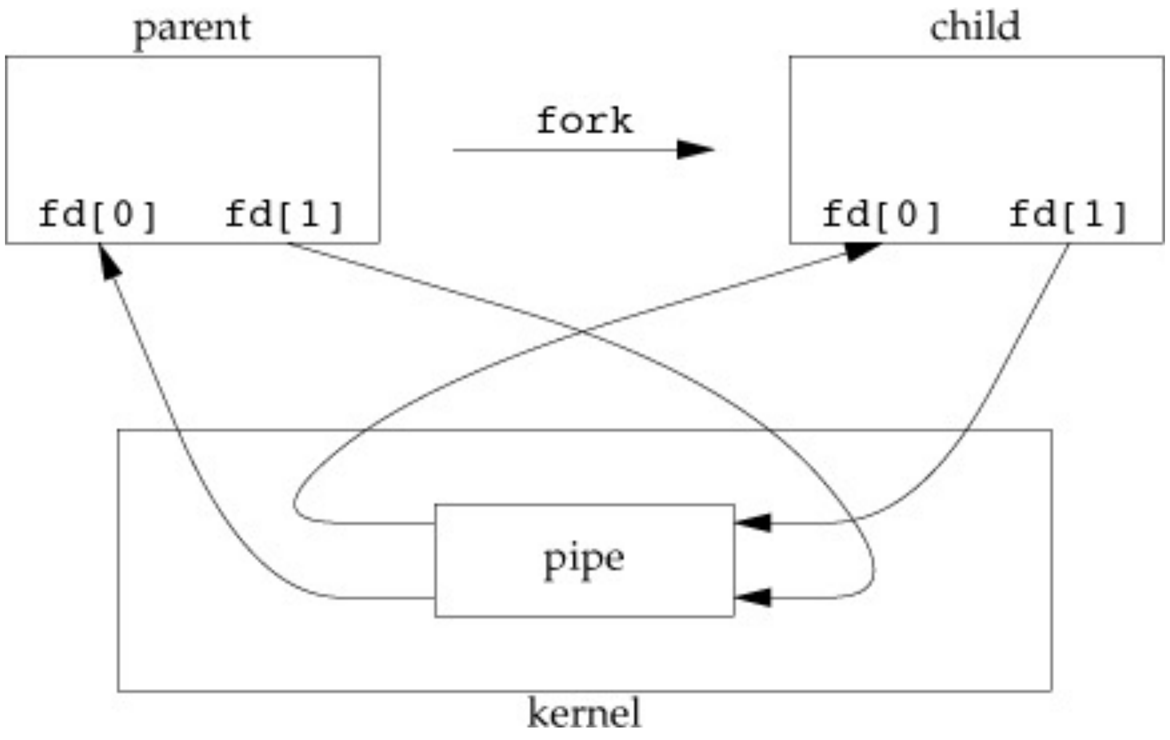
- A uniform interface for all I/O
  - Uniformity: open, close, read, and write
  - Open before use
  - Byte-oriented
  - Kernel-buffered reads/writes
    - ❑ Streaming and block devices looks the same
    - ❑ Read blocks process, yielding processor to other task
    - ❑ Write does not block (even if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

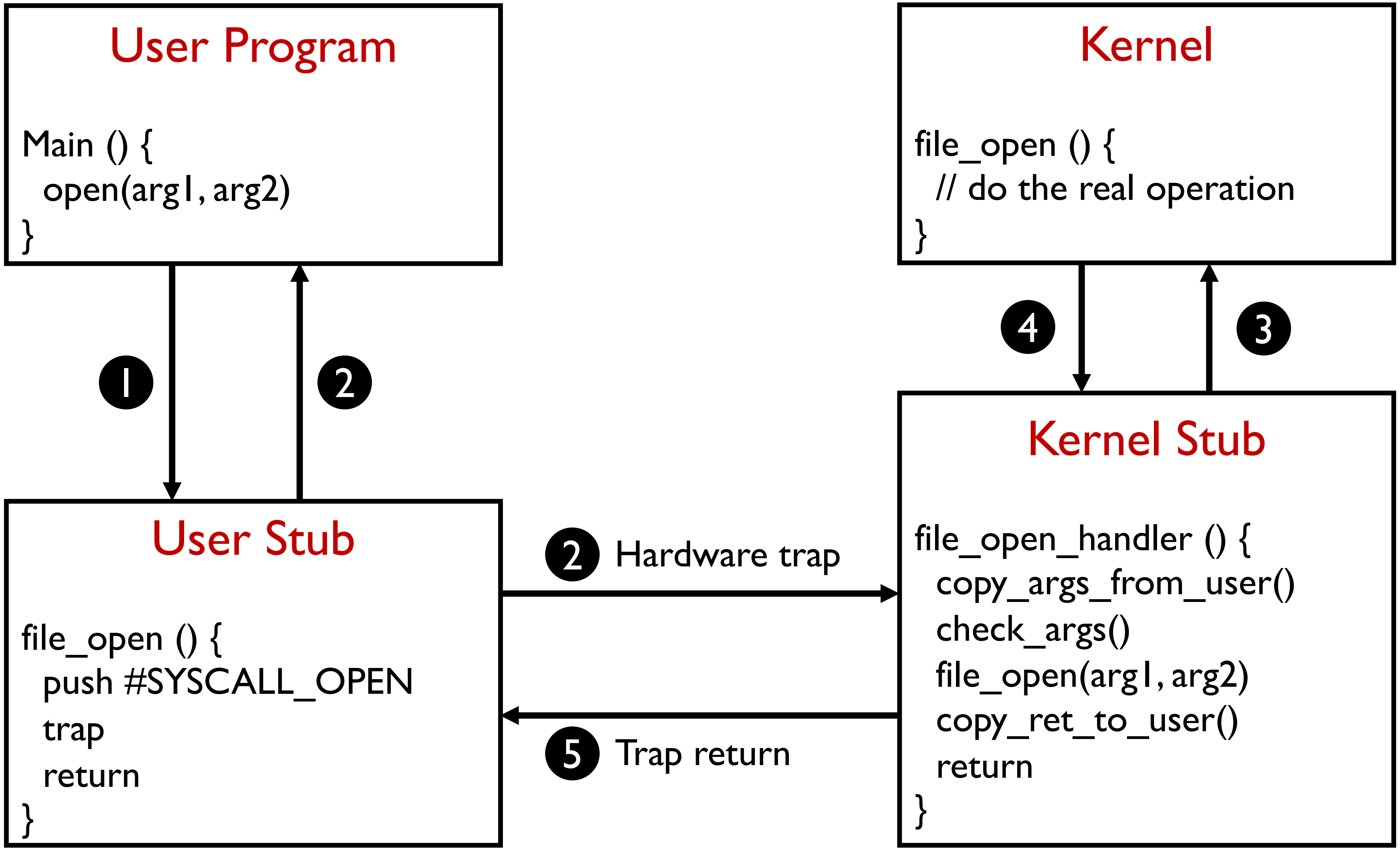
- OS Programming Interface
- Case Study: Process Management
- Case Study: Input/Output
- System Calls Design

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



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

https://developer.ibm.com/articles/l-kernel-memory-access/

- 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?

Kernel Stub

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

System Calls Stubs

https://developer.ibm.com/articles/l-kernel-memory-access/

- 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
}
```