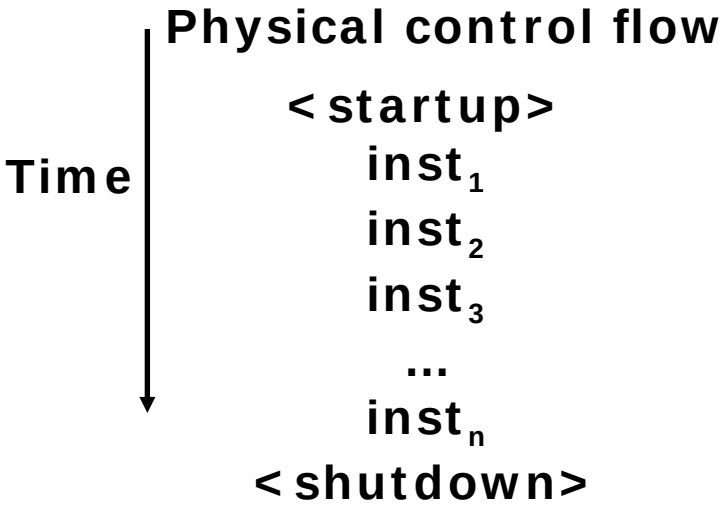
Exceptional Control Flow I

Outline

- Interuption
- Exceptions
- Classes of Exceptions

Control flow

- 从开机到关键， CPU 的工作流程是固定的:
  - While(1) {
    - 读指令；
    - 执行指令；
  - }
- 这个指令序列就是系统的 physical *control flow* (or *flow of control*).



Altering the Control Flow

- 之前我们学过两种改变控制流的方法：
  - Jumps and branches
  - Call and return using the stack discipline.
  - Both react to changes in program state.
- How can we change the control flows among processes (multitasking) ?

Altering the Control Flow

- 此外，还有一些其他需求，需要 CPU 对系统状态改变作出应对：
  - data arrives from a disk or a network adapter.
  - instruction divides by zero
  - user hits ctl-c at the keyboard
- 因此，系统需要异常控制流（exceptional control flow, ECF）的机制
  - 从当前 program 的控制流中跳出，转到其他指令

Exceptional Control Flow

- ECF 的重要用途之一：支撑“进程”这一概念
  - 进程看起来是独享整个机器
  - 但实际上多个进程并发执行，共享计算机
  - 进程之间的切换、交替运行，就是依赖 Exceptional Control Flow 机制

如何让多个进程看起来独享计算机资源？

- 基本思路：
  - 分时复用，时间片切换
- 挑战：
  - #1. 进程如何执行限制指令？（普通程序没有权限）
    - 进行 I/O 操作
    - 分配内存
  - #2. 如何切换进程？

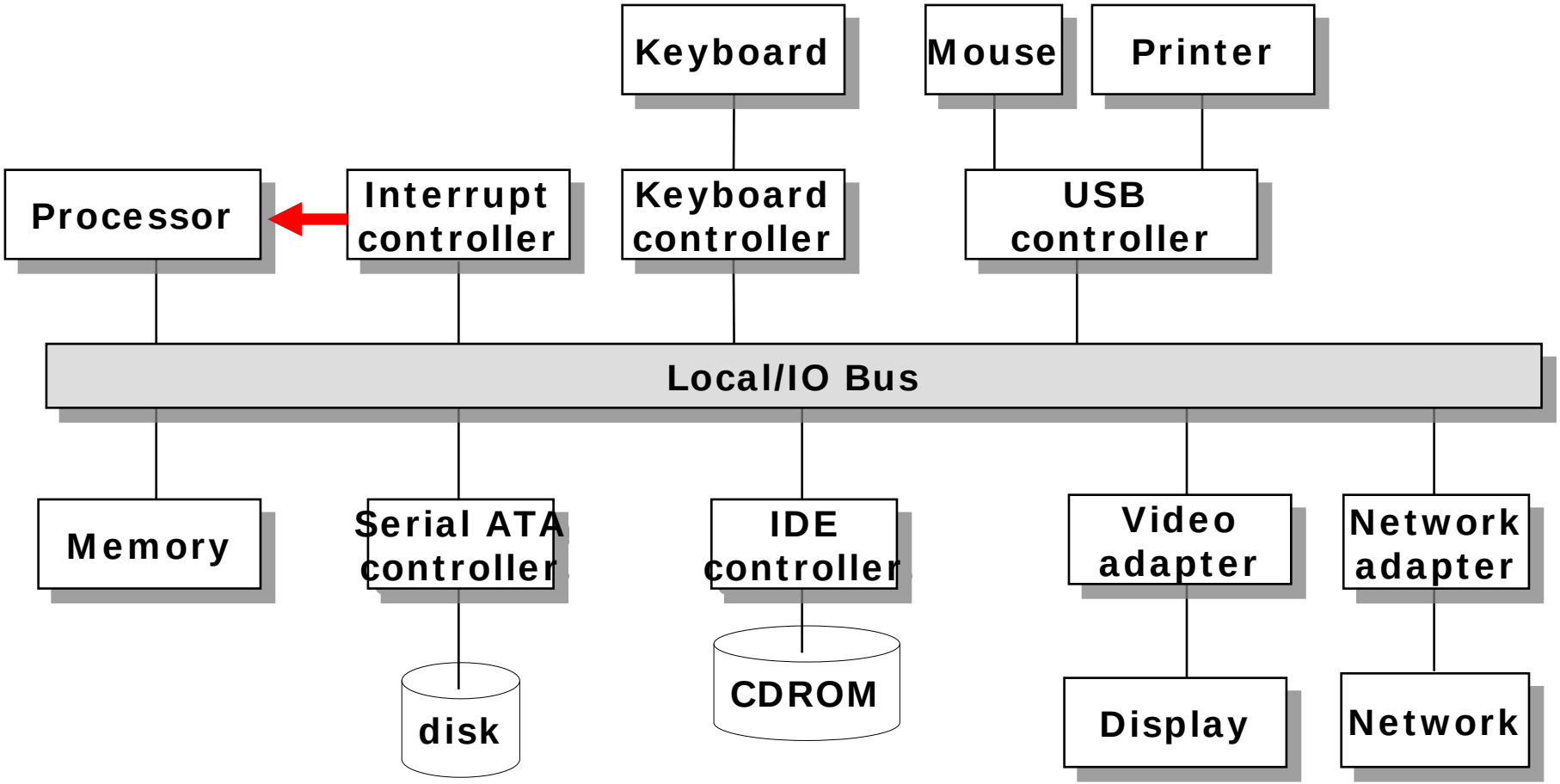
挑战 #1: 进程如何执行限制指令?

- 方案 1
  - 让用户程序都自己做
  - 不合理
    - 访问文件前要做权限检查，如果让用户自己做，那么形同虚设
    - 硬件资源分配，多占用，出错

挑战 #1: 进程如何执行限制指令?

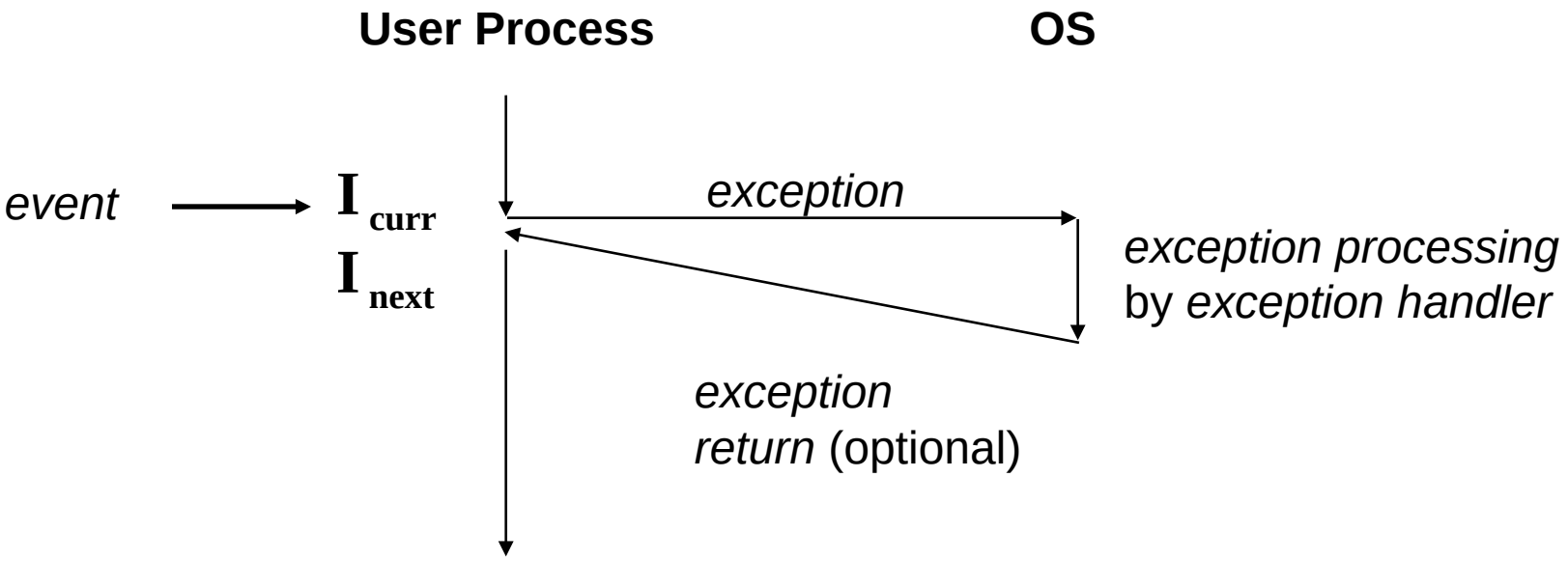
- 方案 2
  - User mode: 有限制，有些指令没有权利执行
    - 例如 I/O 请求
    - 如果这样做了，会抛出异常， OS 会 kill process
  - Kernel mode: OS kernel 有最高权限
    - 限制级指令被包装为 system call
    - 程序调用 system call ，要先切换到 kernel mode 去执行
  - 如何切换 mode ?
    - 中断 / 异常流控制（ CPU 硬件提供支持）
    - 切换到中断处理程序，完成后再返回

System context for exceptions

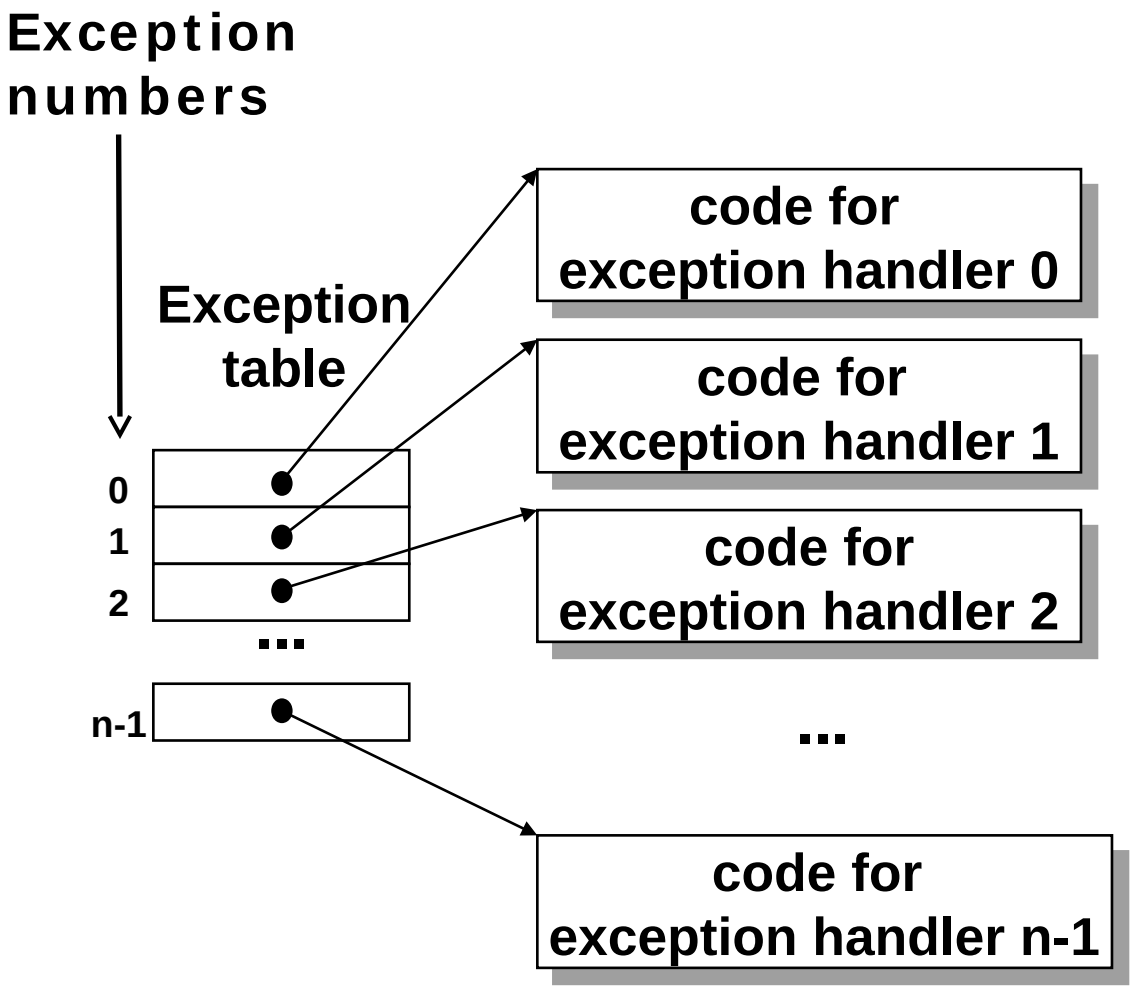


Exceptions

- An *exception* is a transfer of control to the OS in response to some *event* (i.e., change in processor state)



Exception Table

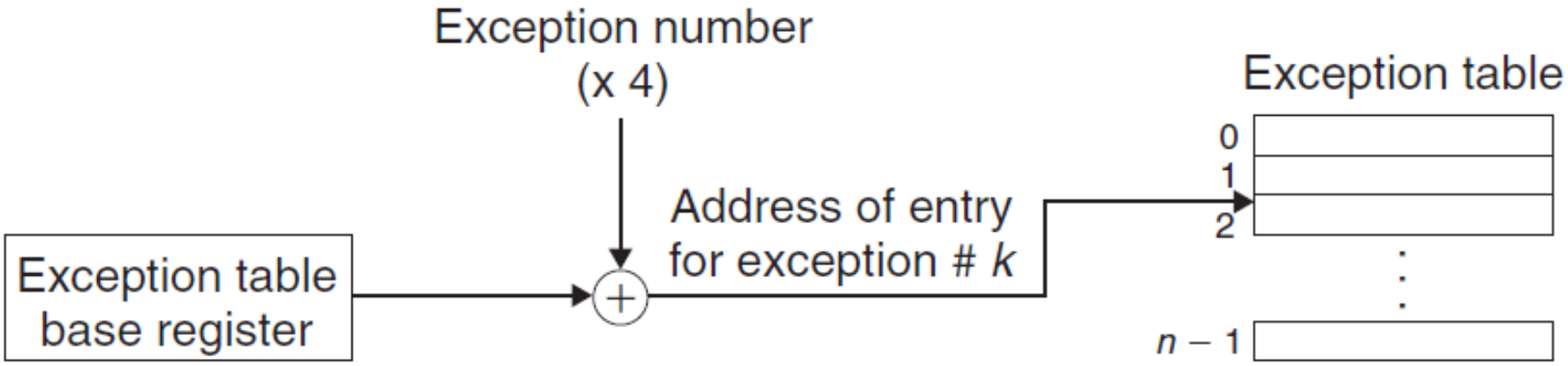


1. Each type of event has a unique exception number k

2. Exception table entry k points to a function (exception handler).

3. Handler k is called each time exception k occurs.

Exception Table



Synchronous exceptions

- Exceptions in IA32 systems

Exception number	Description	Exception class
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32–127	OS-defined exceptions	Interrupt or trap
128 (0x80)	System call	Trap
129–255	OS-defined exceptions	Interrupt or trap

挑战 #1: 进程如何执行限制指令?

- 没有限制，直接执行程序

OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute **call** main()	
	Run main()
	Execute **return** from main
Free memory of process	
Remove from process list	

挑战 #1

- 有限制的直接执行

OS @ boot (kernel mode)	Hardware	Program (user mode)
initialize trap table		
	remember address of... syscall handler	
OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
Create entry for process list		
Allocate memory for program		
Load program into memory		
Setup user stack with argv		
Fill kernel stack with reg/PC		
return-from-trap		
	restore regs from kernel stack	
	move to user mode	
	jump to main	
		Run main()
		...
		Call system call **trap** into OS
	save regs to kernel stack	
	move to kernel mode	
	jump to trap handler	
Handle trap		
Do work of syscall		
return-from-trap		
	restore regs from kernel stack	
	move to user mode	
	jump to PC after trap	
		...
		return from main
		trap (via `exit()`)
Free memory of process		
Remove from process list		

挑战 #2：如何切换进程?

- 方案 1. a cooperative approach: wait for system call
  - E.g., early version of Macintosh OS
  - OS 信任进程，进程执行足够长时间后，主动放弃 CPU；然后 OS 再选择执行其他进程
  - 当进程调用 system call 时，就放弃了 CPU （例如访问文件，或传输网络消息）
    - yield system call: 只是归还控制权
  - 或者进程出错时也会归还控制权，例如除以 0
    - 出错是也会产生 trap

挑战 #2：如何切换进程?

- 方案 2. A non-cooperative approach: The OS takes control
  - 怎么抢回控制权?
  - Timer interrupt (时间片)

挑战 #2：如何切换进程?

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember addresses of...
	syscall handler
	timer handler
start interrupt timer	
	start timer
	interrupt CPU in X ms

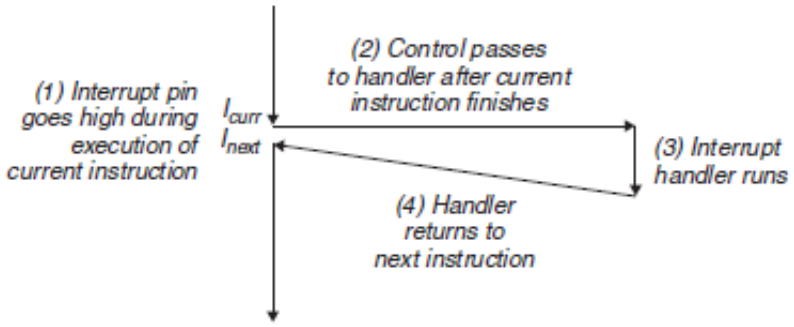
OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	timer interrupt	
	save regs(A) to k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) to proc-struct(A)		
restore regs(B) from proc-struct(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) from k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

Exception 分类

- 中断（interrupt）
- 陷阱（trap）
- 故障（fault）
- 终止（abort）

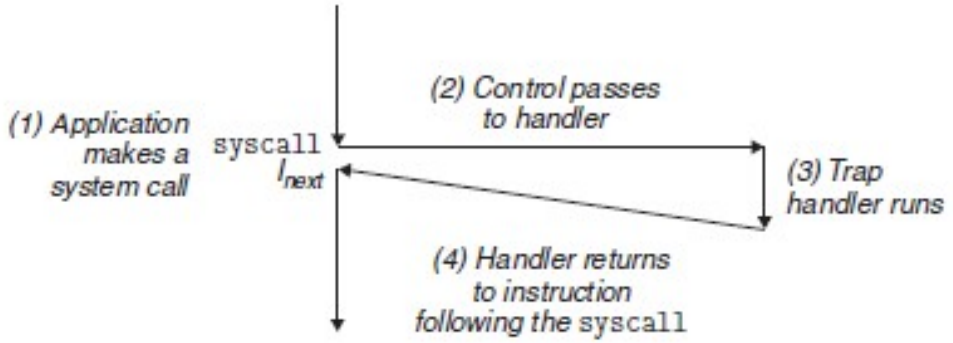
Exception 分类

- 中断（interrupt）
  - 来自 I/O 设备等硬件
  - 异步发生（与指令不对齐，任何时间都可能发生）
  - CPU 有专门的硬件管脚接受中断请求
  - 对应的处理程序叫“中断处理程序”（interrupt handler）
    - 中断不能打断当前指令（指令具有原子性）
    - 当前指令结束后执行中断处理程序
    - 然后返回下一条指令
    - 中断可以[illegible]中断所中断）



Exception 分类

- 陷阱（trap）和系统调用
  - 有意的异常，是执行一条特殊指令（int 指令）的结果，是同步发生的
  - 也叫做 software interrupt
  - trap 最重要的用途是在用户程序和内核之间提供一种接口，即系统调用（system calls）
  - 与中断处理类似，执行完陷阱处理程序后，会返回当前程序的下一条指令



Synchronous exceptions

- Exceptions in IA32 systems

Exception number	Description	Exception class
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32–127	OS-defined exceptions	Interrupt or trap
128 (0x80)	System call	Trap
129–255	OS-defined exceptions	Interrupt or trap

System Calls

- System calls in IA32 systems

Number	Name	Description	Number	Name	Description
1	exit	Terminate process	27	alarm	Set signal delivery alarm clock
2	fork	Create new process	29	pause	Suspend process until signal arrives
3	read	Read file	37	kill	Send signal to another process
4	write	Write file	48	signal	Install signal handler
5	open	Open file	63	dup2	Copy file descriptor
6	close	Close file	64	getppid	Get parent's process ID
7	waitpid	Wait for child to terminate	65	getpgrp	Get process group
11	execve	Load and run program	67	sigaction	Install portable signal handler
19	lseek	Go to file offset	90	mmap	Map memory page to file
20	getpid	Get process ID	106	stat	Get information about file

System Call Example

```
# hello world
1 int main()
2 {
3       write(1, "hello, world\n", 13);
4       exit(0);
5 }
```

System Call Example

```
1       .section .data
2       string:
3            .ascii "hello, world\n"
4       string_end:
5            .equ len, string_end - string
6       .section .text
7       .global main
8 main:
```

System Call Example

```
  First, call write(1, "hello, world\n", 13)
9 movl  $4, %eax          Sdystem call number 4
10      movl  $1, %ebx      stdout has descriptor 1
11      movl  $string, %ecx       Hello world string
12      movl  $len, %edx          String length
13      int   $0x80                  System call code
  Next, call exit(0)
14movl  $1, %eax           System call number 1
15      movl  $0, %ebx                  Argument is 0
16      int   $0x80                  System call code
```

x86_64 使用的 syscall 方式

```
int main(void)
{
  unsigned long syscall_nr = 60;
  long exit_status = 99;

  __asm__ __volatile__ (
  "movq %0, %%rax\n\t"  /* 调用号syscall_nr存入rax寄存器 */
  "movq %1, %%rdi\n\t"  /* 第一个参数返回码exit_status存入rdi寄存器 */
  "syscall"
  :
  :"m"(syscall_nr), "m"(exit_status)
  :"rax", "rdi"
  );
}  // gcc -o asmsyscall_syscall asmsyscall_syscall.c
```

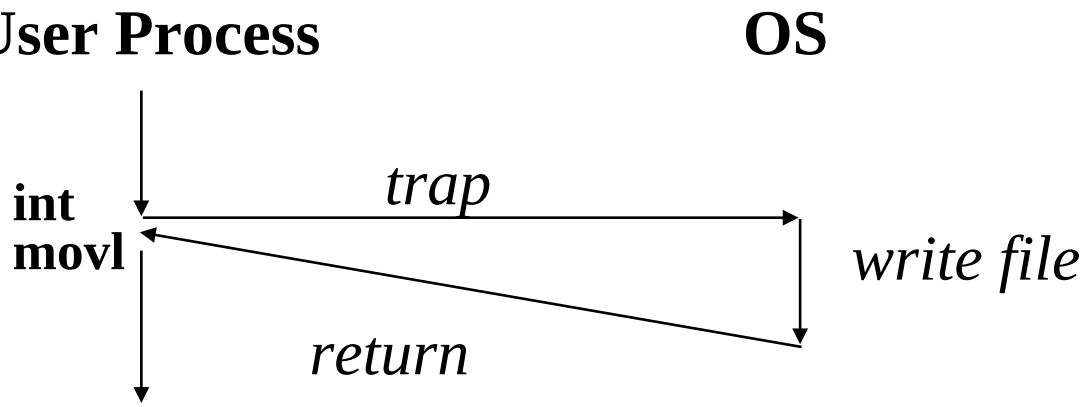
Syscall 指令，系统调用号存放在 rax 寄存器中，参数按顺序存放在 rdi 、 rsi 、 rdx 、 r10 、 r8 、 r9 寄存器中

Syscall 要点

- 应用程序一般无法离开 syscall （系统编程）
  - 文件操作（磁盘）
  - 输入输出（键盘、显示器）
  - 退出程序
  - ……
- 与 Call 不同， Syscall= 升级权限 + 跳转
- 理解汇编，才真正理解 OS
  - 了解硬件

Trap Example

- Writing a File
  - OS must find the file, write the string to it
  - Returns integer whether it is succeeded



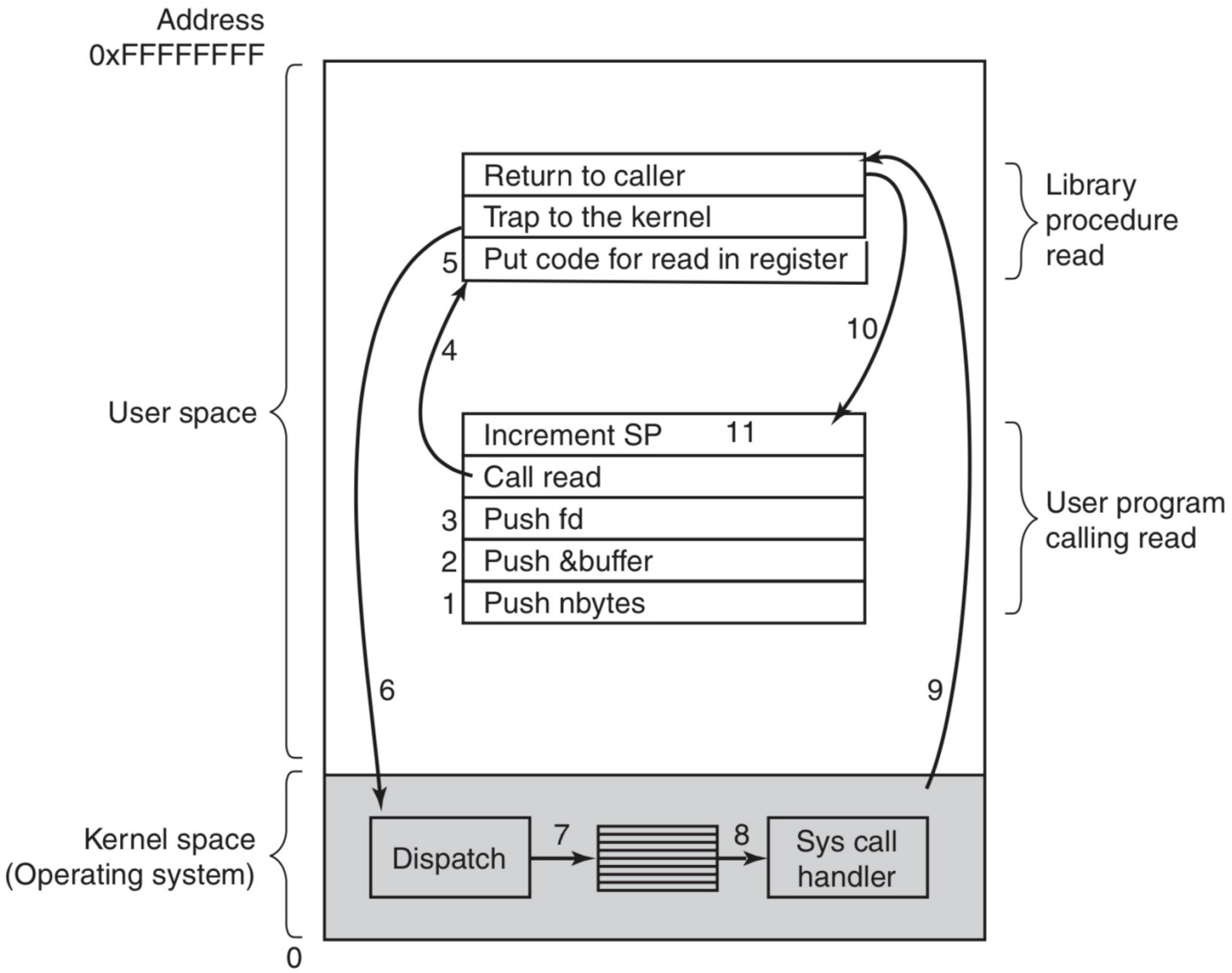
System Call vs. Procedure Call

- 系统调用（例如 open() ）看起来就像函数调用，有什么区别呢?
  - open() 等就是函数调用（库函数）
  - 只是函数内部调用汇编的 int 80 指令等触发 exception，进入内核态，进入真正的 open 系统调用的代码
  - 1 ）系统调用和函数调用很像，主要区别是进入内核态
  - 2 ）系统调用没法指定目标函数的地址，只能传递一个系统调用号码给内核

System Call

- Example: read
  - count = read(fd, buffer, nbytes);
    - #1，文件描述符；
    - #2，读出数据将要放在内存中的位置，内存缓冲区首地址；
    - #3，读多少字节
  - 执行流程：
    - 用户调用 read 的代码（含传参）
    - Read 库函数的代码
    - 内核中 read 系统调用的代码
    - 具体 11 个步骤（下页图）

System Call


Address
0xFFFFFFFF
User space
Kernel space
(Operating system)
0
Return to caller
Trap to the kernel
5
Put code for read in register
Library
procedure
read
10
4
Increment SP
11
Call read
3
Push fd
2
Push &buffer
1
Push nbytes
User program
calling read
6
9
Dispatch
7
8
Sys call
handler

System Call

- 各种类型的系统调用

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc	
s = execve(name, argv, er	
exit(status)	

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
	nto a buffer
	r into a file
	mation

Directory- and file-system management

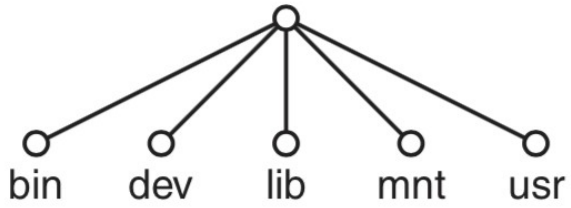
Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	
s = mount(special, name, flag	
s = umount(special)	

Miscellaneous

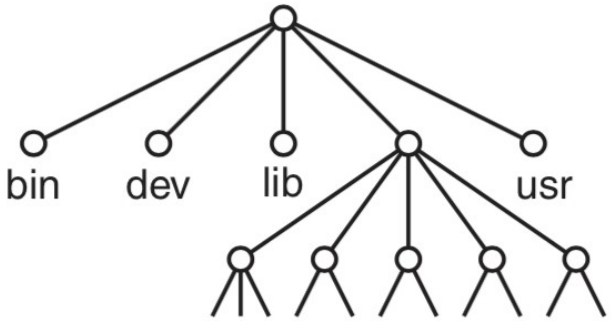
Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

System Call

- Example： 目录操作 mount
  - 把一个**设备**挂载到**文件系统**树的一个**挂载点**（必须是目录）上
  - mount("/dev/sdb0", "/mnt", 0); // 第三个参数表示 read-write 或 read-only



(a)

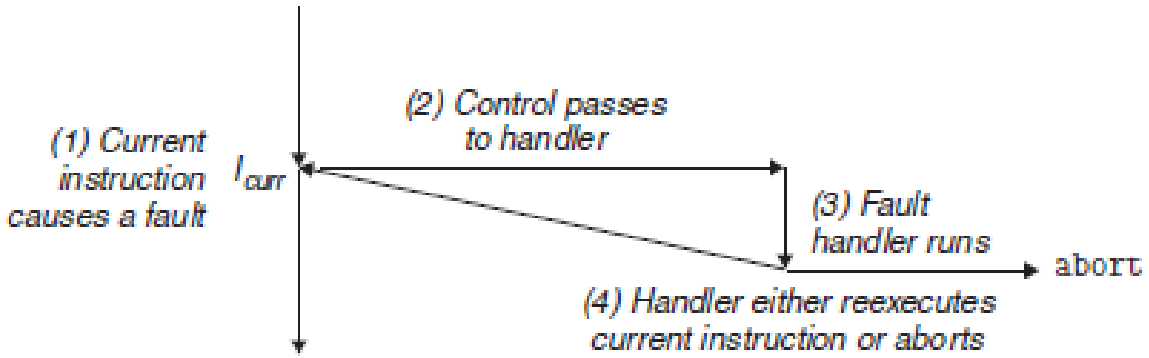


(b)

Fdisk (磁盘分区表)
mkfs
Mount
df -lh

Exception 分类

- 故障（Fault）
  - 故障由错误引起，是某条指令执行引发，因此也是同步发生的
  - 处理器将控制转移给“故障处理程序”
    - 如果能够修复故障，那么返回引起故障的指令，并重新执行
    - 如果不能修复故障，就返回到内核中的 abort 例程， abort 例程会终止引起故障的应用程序



Fault Example #1

- Memory Reference
  - User writes to memory location
  - That portion (page) of user's memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

```
80483b7: c7 05 10 9d 04 08 0d  movl   $0xd,0x8049d10
```

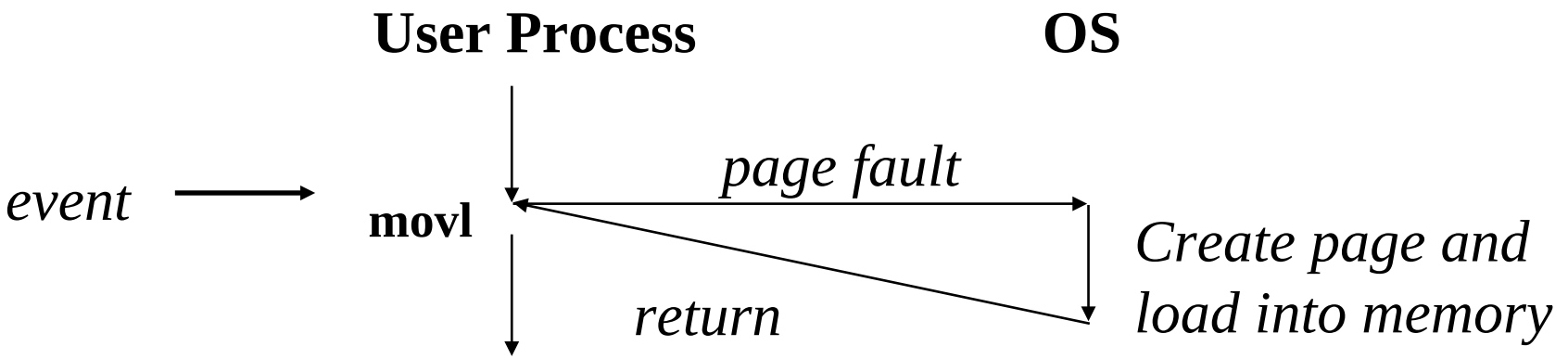
Fault Example #1

- Memory Reference
  - Page fault handler must load page into physical memory
  - Returns to faulting instruction
  - Successful on second try

Fault Example #1

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

```
80483b7: c7 05 10 9d 04 08 0d  movl   $0xd,0x8049d10
```



Fault Example #2

- Memory Reference
  - User writes to memory location
  - Address is not valid

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7: c7 05 60 e3 04 08 0d  movl   $0xd,0x804e360
```

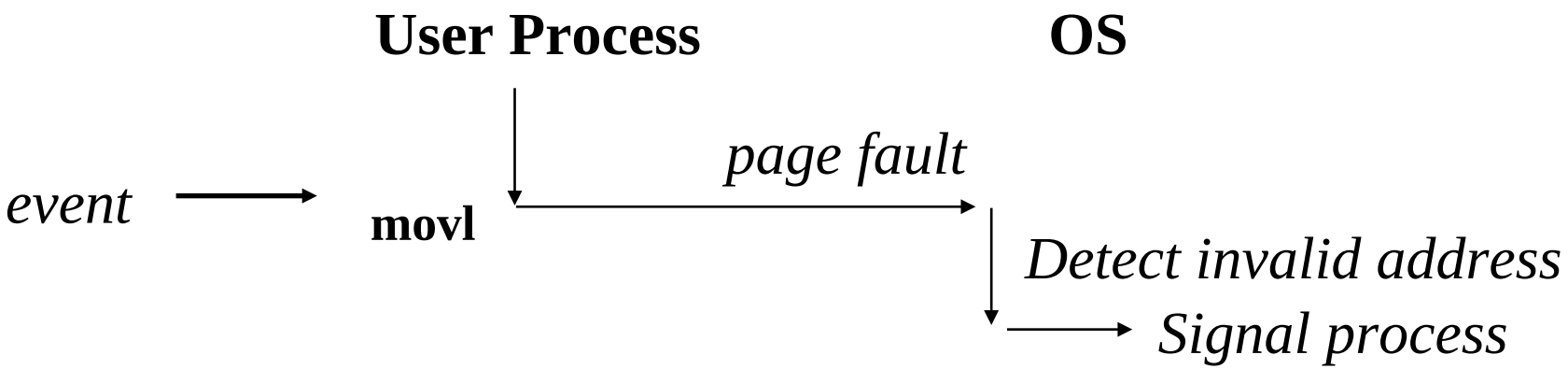
Fault Example #2

- Memory Reference
  - Page fault handler detects invalid address
  - Sends SIGSEG signal to user process
  - User process exits with “segmentation fault”

Fault Example #2

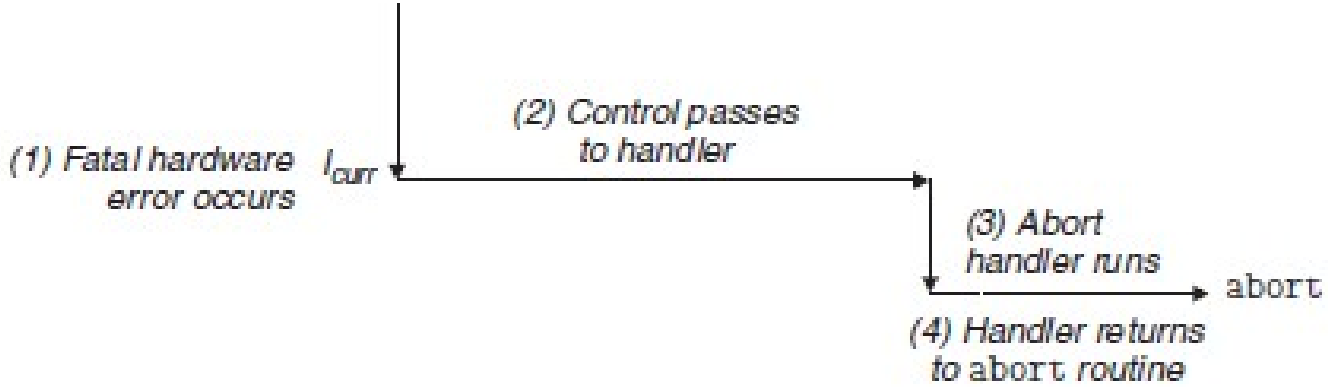
```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7: c7 05 60 e3 04 08 0d  movl   $0xd,0x804e360
```



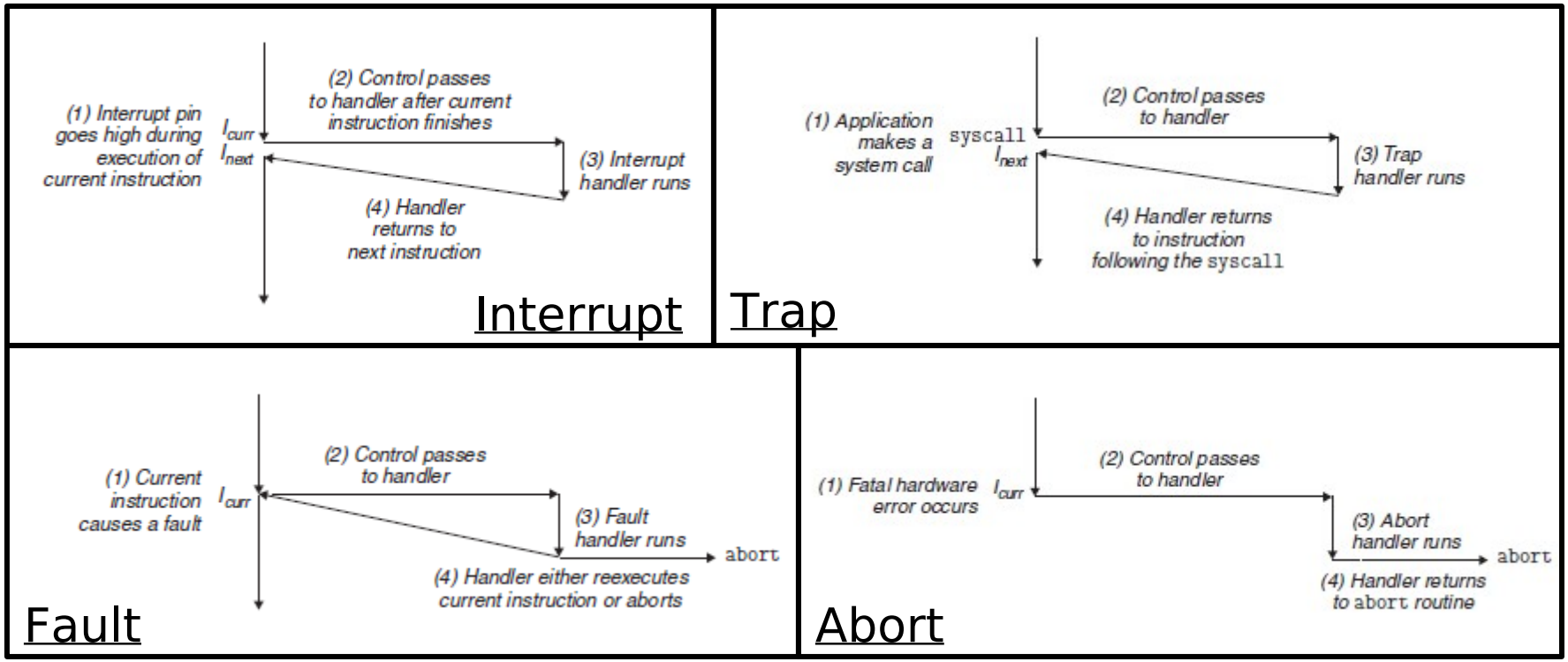
Exception 分类

- 终止（abort）
  - 不可修复的错误造成的结果
  - 一般是硬件错误，比如 DRAM 或 SRAM 位被损坏时发生的奇偶错误
  - 不会将控制权返回给应用程序，而是终止该程序



Exceptions

Class	Cause	Async/Sync	Return behavior
Interrupt	Signal from I/O device	Async	Always returns to next instruction
Trap	Intentional exception	Sync	Always returns to next instruction
Fault	Potentially recoverable error	Sync	Might return to current instruction
Abort	Nonrecoverable error	Sync	Never returns



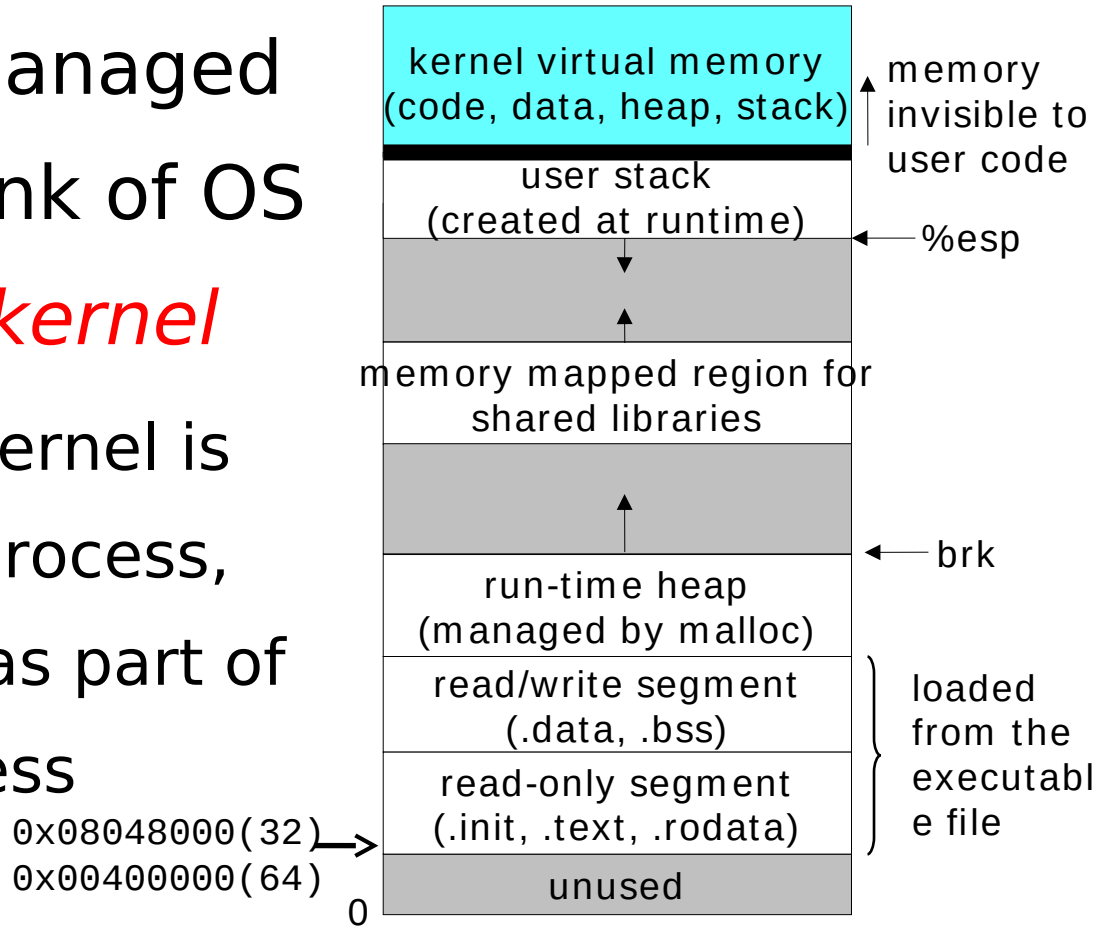
Outline

- Kernel Mode and User Mode
- Context Switch
- System Calls and Error Handling
- Process Control

Kernel and User Mode

User and Kernel Modes

- Processes are managed by a shared chunk of OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some user process



内核线程 vs. 普通进程

- 内核线程，也叫内核任务
  - 一般周期性执行
  - 例如磁盘高速缓存的刷新、网页连接的维护、页面换入换出
- 内核线程与普通进程的区别
  - 内核线程只运行在内核态，而普通进程可以运行在用户态，也可以运行在内核态
  - 内核线程由于只在内核态，只能使用大于PAGE_OFFSET （3G 以上）的内存；而普通进程不论在用户态，还是在内核态，都可以使用 4G 地址空间

Protection

- 限制应用程序
  - 不能执行一些指令（特权指令）
  - 不能访问一些地址空间（kernel address space）
- 一般是在某个控制寄存器中，通过一个 mode bit 来控制
  - Kernel mode: if the bit is set
  - User mode: if the bit is clear

User and Kernel Modes

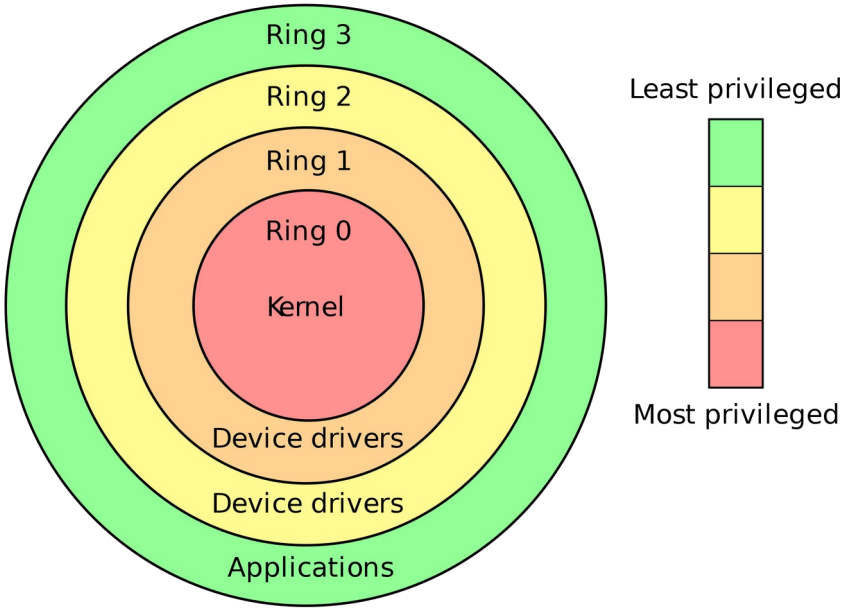
- 运行在 kernel mode 的进程能够
  - 执行 CPU 指令集中的任何指令
  - 能否访问系统中的任何内存地址（物理地址访问）
- 运行在 user mode 的进程
  - 既不能执行特权指令
  - 也不能直接访问地址空间中 kernel 区域的数据和代码
  - 只能通过 system call 接口来做以上事情

User and Kernel Modes

- 一个运行应用代码的进程
  - 初始状态是在 user mode
  - 从用户态到内核态
    - 当 exception 发生、控制权转移到 exception handler 时
  - 从内核态到用户态
    - 当控制权回到应用代码时
- 一个进程从 user mode 变到 kernel mode 的唯一方法是 exception

User and Kernel Modes

- 需要 CPU 硬件的支持和配合
  - Intel x86 CPU 提供 Ring 0~3 不同的特权级别
  - 当陷入内核态时，要先提升 CPU 特权级别，否则缺少很多权限
  - 很多系统只用两级（只需要 1 bit），例如 Windows 7



Context Switches

Context switching

- Kernel 维护了每个进程的上下文（context）
- Context 是指进程在重新启动一个进程时所需要的所有状态信息
- Context contains
  - the program’s code and data stored in memory
  - its stack
  - the contents of its general-purpose registers
  - its program counter
  - environment variables
  - and the set of open file descriptors

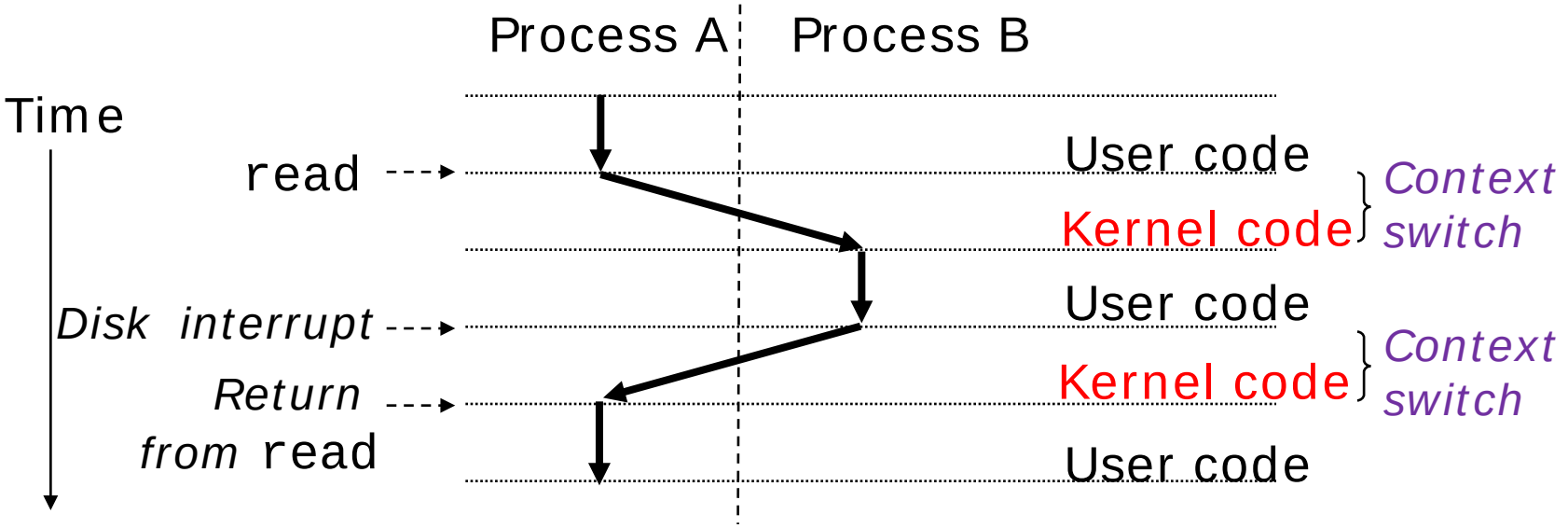
Context switching

- 指进行多任务切换（例如通过时间片）的机制
- Exceptional control flow 的高层次形式
  - 构建于底层 exception 机制之上

Context switching

- Context switch 在什么情况下发生呢?
  - Kernel 正在代表用户执行一个系统调用时，例如
    - **`read`**, **`sleep`** , **`etc.`** which will cause the calling process blocked
    - 即使一个 system call 并不会 block 进程， kernel 代码也要进行一次 context switch ，而不是继续执行原来的调用进程（只要发生 system call ，都要强制触发 context switch ）
  - 作为中断的结果
    - 最常见的是 Timer interrupt
    - I/O 设备完成操作发生中断

Context switching



Context switching

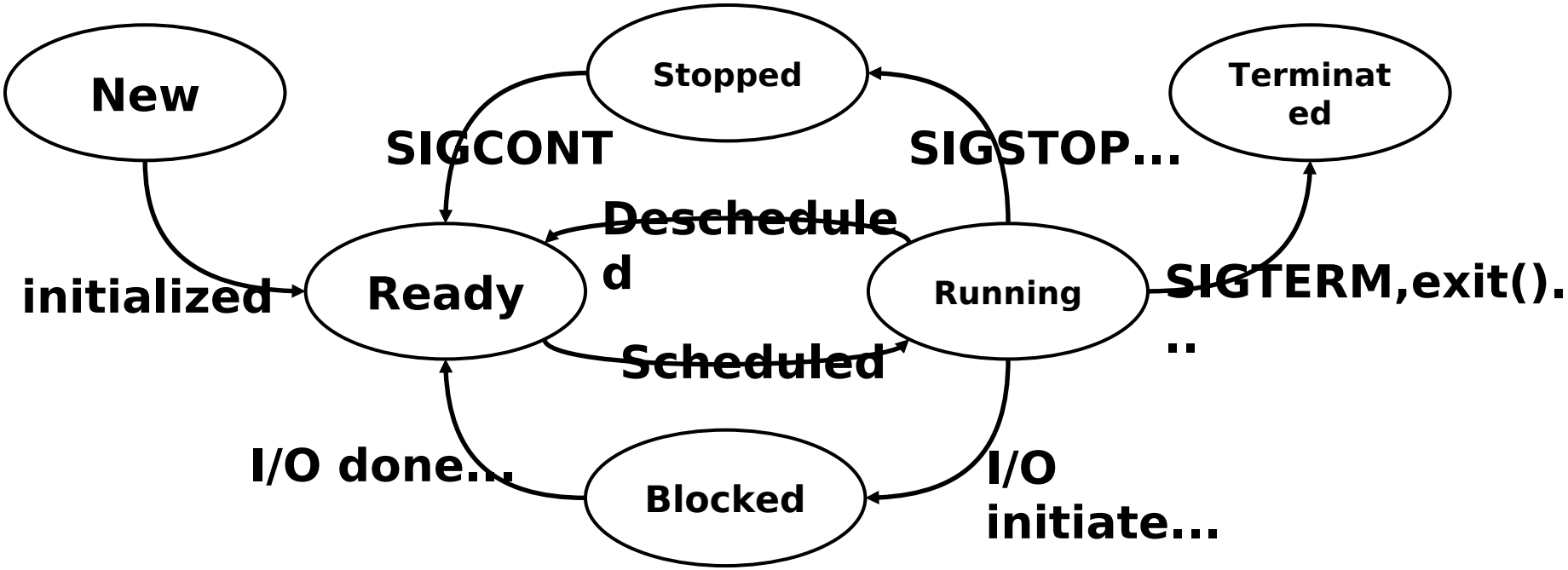
- 调度器（一部分 kernel 代码）会按照如下方法来执行调度：
  - 在执行一个进程的代码时，决定是否要抢占当前进程
  - 选择一个之前被抢占的进程 (scheduled process)，重新启动它
  - 抢占当前进程
    - 保存当前进程的上下文
  - 重启 scheduled process
    - 回复 scheduled process 的上下文
    - 将控制权交给新恢复的进程
  - 选择 scheduled process 的方法叫 CPU 调度算法
    - 将在后面单独讲解

States of a Process

- New (新建)： 进程正在初始化
- Running (运行)： 进程正在 CPU 上执行
- Ready (就绪)
  - 进程等待被执行，且迟早 (也许一万年) 会被调度执行
- Stopped (暂停)
  - 进程暂停执行，且永远都不会被调度 (除非转为 Ready)
- Blocked (阻塞)
  - 进程等待外部事件 (如 I/O 完成) 而停止执行，且永远都不会被调度 (除非事件完成、转为 Ready)
- Terminated (终止)： 进程永久停止执行

States of a process

- 进程的状态转移



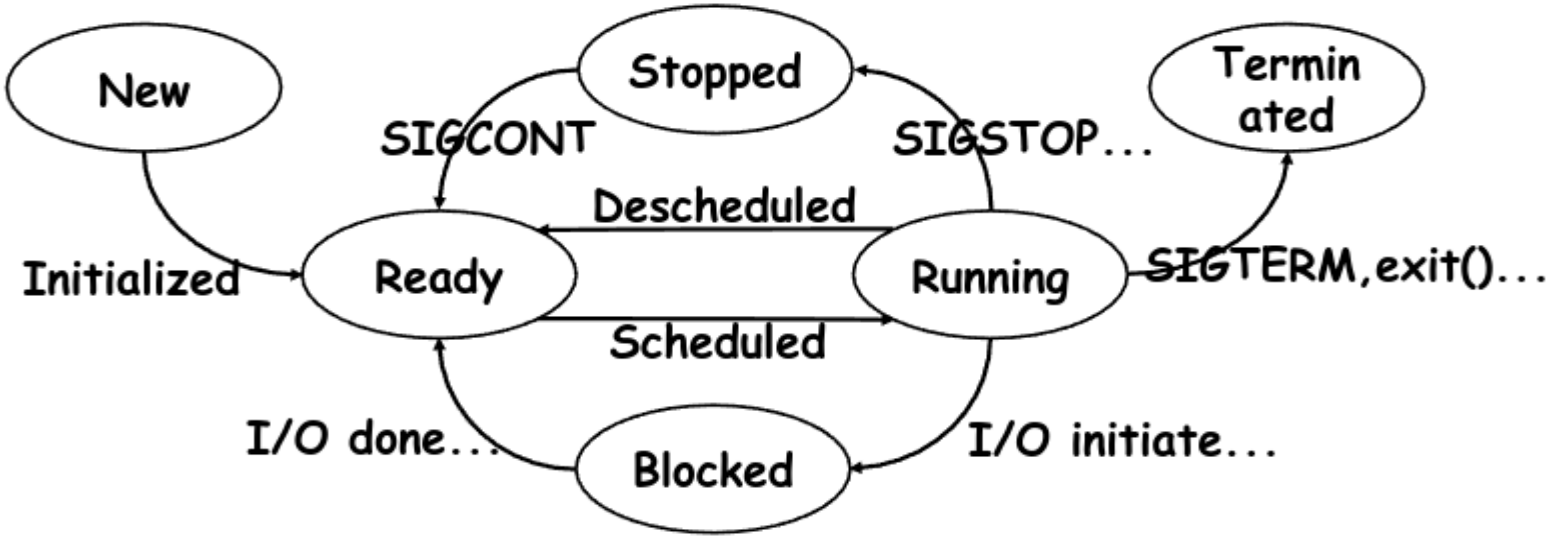
SIGSTOP、 SIGCONT、 SIGTERM 等是 Signal
Signal:OS 提供的一种 software interrupt 机制，之后课程会讲

States of a process

- 任何非终止状态下的进程都可以被终止 (terminated)
- 终止意味着回收进程的一切资源
- 进程如何被终止
  - 接收到一个可以终止进程的 signal，如 SIGTERM
  - 从 main 函数返回
  - 调用 **`exit`** 系统调用
  - 父进程被终止
  - OS 被关闭

States of a process

- 进程不能直接从 Blocked 或 Stopped 状态进入 Running 状态
- 必须先进入 Ready 状态，等待 OS 调度
- 否则进程就可以不进入 ready 队列，这等于跳过了 OS 的进



States of a process

- 进程的状态转移

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

States of a process

- 进程数据结构举例

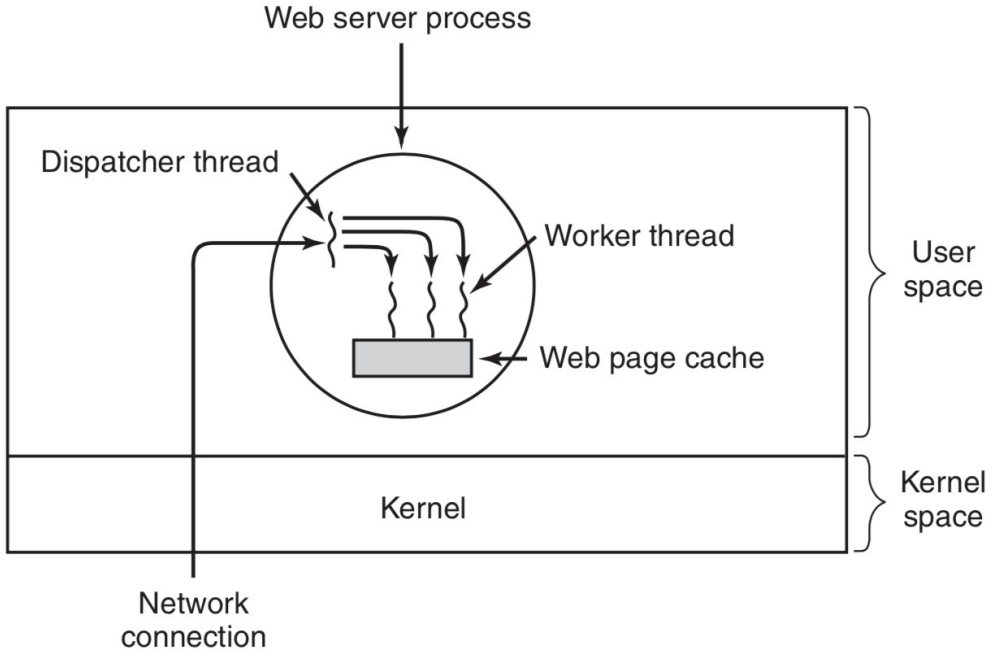
```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent 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
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

States of a process

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Thread

- 在同一个进程空间的多个指令流（控制流）
- 共享很多进程资源，变量可以互相访问
- 切换开销较小
- 将在“并行”部分详细介绍

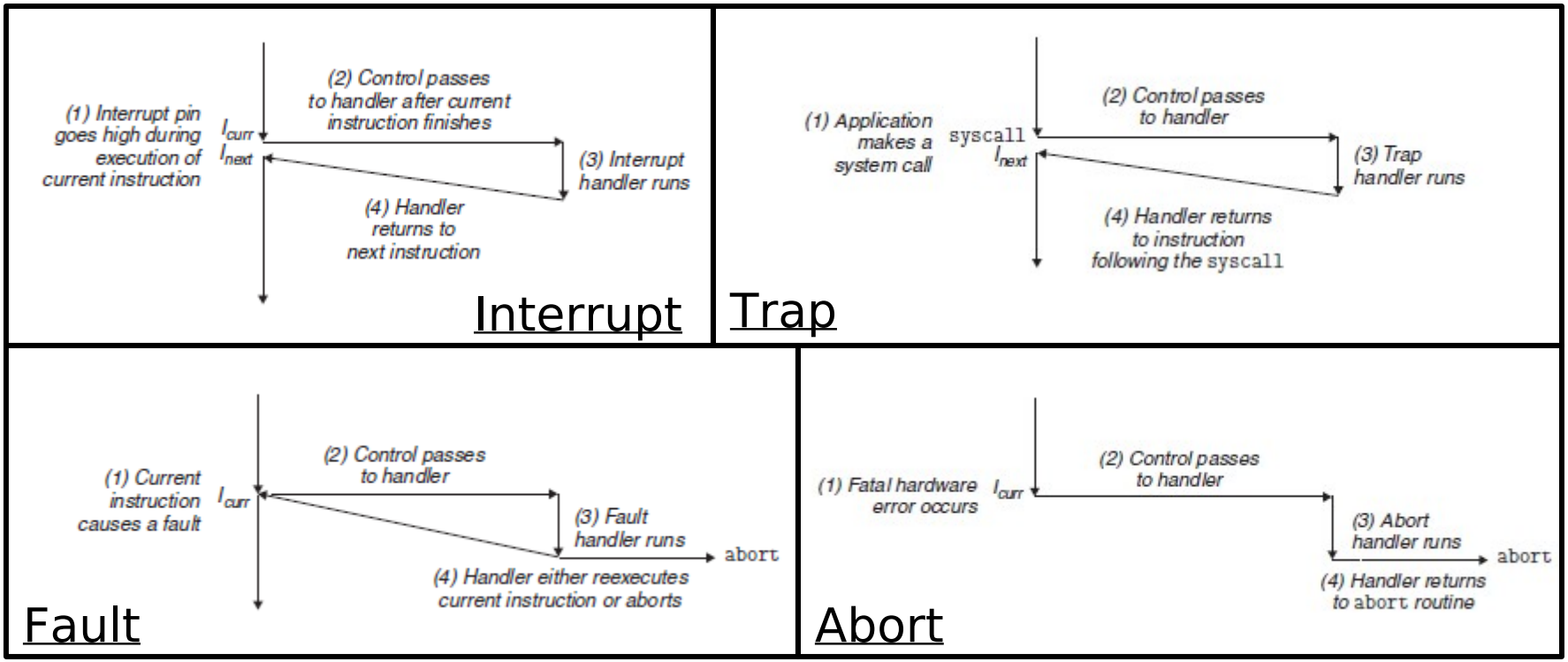


Exceptions

安全 / 权限控制　　多人共享

OS

Class	Cause	Async/Sync	Return behavior
Interrupt	Signal from I/O device	Async	Always returns to next instruction
Trap	Intentional exception	Sync	Always returns to next instruction
Fault	Potentially recoverable error	Sync	Might return to current instruction
Abort	Nonrecoverable error	Sync	Never returns



System Call Error Handling

- Unix system-level functions encounter an error
  - typically return –1
  - set the global integer variable **`errno`**
    - to indicate what went wrong

System Call Error Handling

```
1  if ((pid = fork()) < 0) {
2      fprintf(stderr, "fork error: %s\n",
3              strerror(errno));
3      exit(0);
4  }
```

```
1 void unix_error(char *msg) /* unix-style error */
2 {
3      fprintf(stderr, "%s: %s\n", msg, strerror(errno));
4      exit(0);
5 }
```

System Call Error Handling

```
1 pid_t Fork(void)
2 {
3       pid_t pid;
4
5       if ((pid = fork()) < 0)
6             unix_error("Fork error");
7       return pid;
8 }
```

课堂练习

- 说明下列场景下触发的 exception 的类型，以及是同步还是异步?
  - 程序中访问数组越界
  - 从磁盘中读取一个数据块完成
  - 调用 read() 函数

Homework1

- 3 月 16 日 23:00 前在 obe 提交