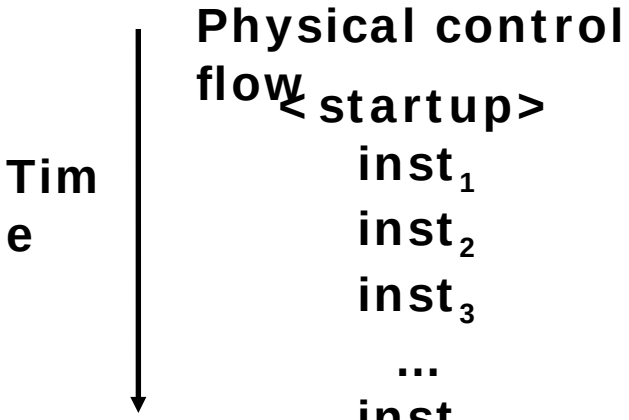
Exceptional Control Flow I
（异常控制流 1）

Outline

- Interuption (中断)
- Exceptions (异常)
- Classes of Exceptions

Control flow（控制流）

- 从开机到关机， 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*



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.

Altering the Control Flow

- 但是如何高效地在进程之间切换 control flow：
  - 多任务 (mul[illegible]解决的问题

Altering the Control Flow

- 此外，一些外部事件需要 CPU 改变 control flow：
  - 磁盘或网卡上要读取 / 接受的数据已经 ready.
  - 执行了一个除以 0 的指令
  - 用户在 shell 上键入了 ctl-c

Exceptional Control Flow

- 这些情况都使得系统要突然改变控制流
- 这种可能突变的控制流称为异常控制流（exceptional control flow, ECF）
  - 从当前 program 的控制流中突然跳出，转到其他指令

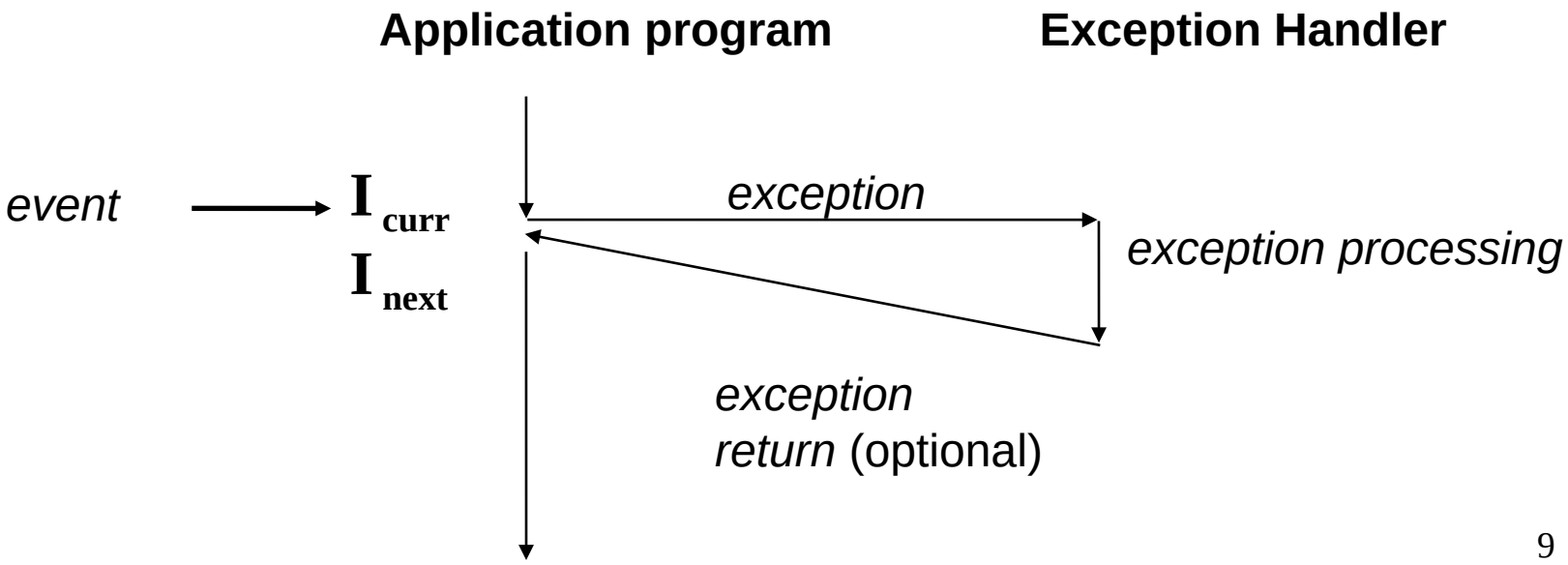
- 硬件和操作系统有一套协作的机制来实现 ECF

Exceptions

- 要理解 ECF，首先需要理解什么是 Exception (异常)
- Exception 在 OS 中是指原本的 Sequencial Control Flow 的突然改变，这种改变是由于 CPU 状态的改变 (event) 导致的
- 和编程语言 (如 C++ 、 Java) 中的 Exception 不是一个东西

Exceptions

- event 发生时， CPU 会从 application 的控制流跳转到 exception handler ，进行 exception processing
- 完成后可能会返回 application 的控制流 (也可能 abort application)



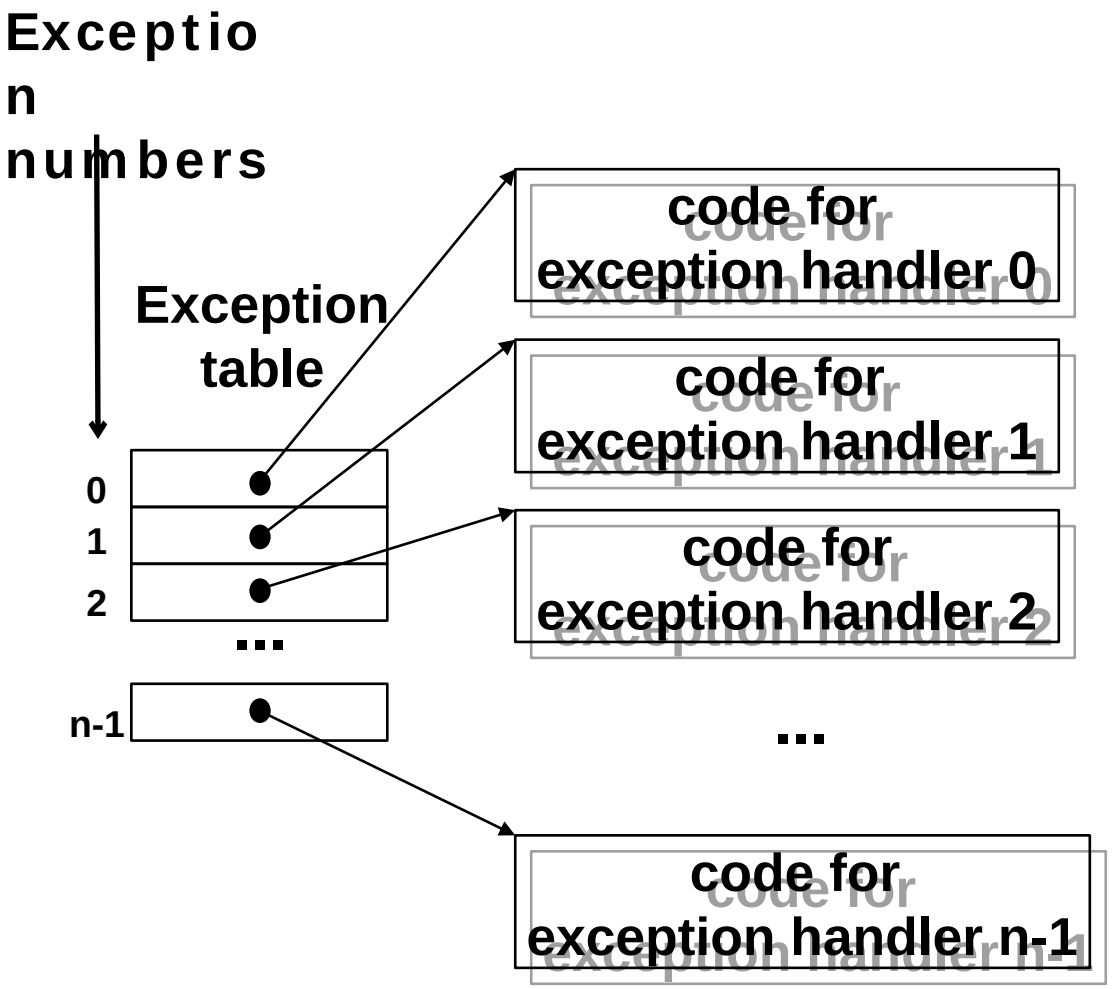
Exceptions

- 系统中每种可能的异常都有一个唯一的正整数的 exception number
- 有些 exception number 在设计 CPU 时就定义好了，有些则可以由 OS Kernel 来分配

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

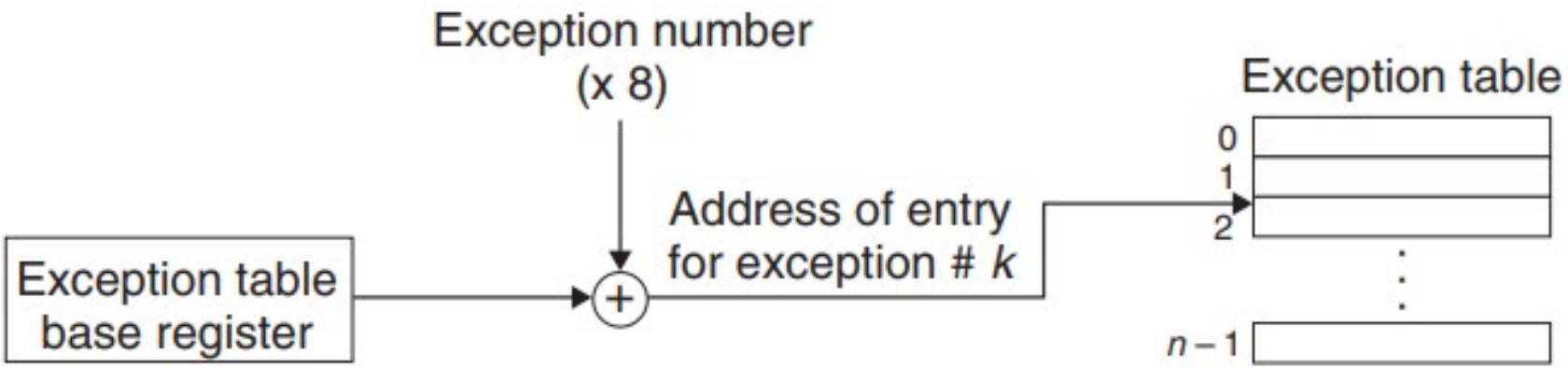
Exception Table

OS 在启动时会创建和初始化 exception table, 每个 exception number 对应其中的一个 entry，存放着相应 exception handler 的指针



1. 每种 event 有唯一的 exception number k
2. Exception table entry k 指向一个 exception handler.
3. Handler k 在 exception k 发生时被调用 .

Exception Table



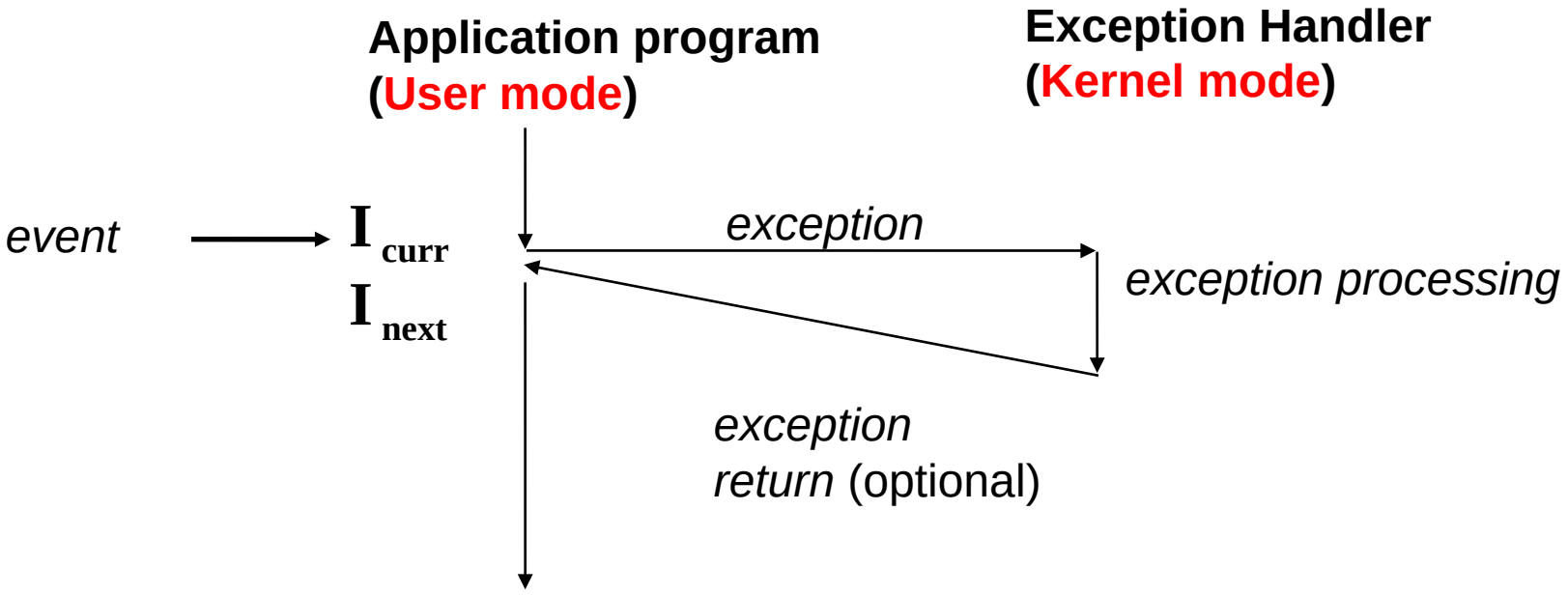
一个 CPU register，存放 exception table 的起始地址

Exception Handler

- Exception handler 由 OS 装载，运行在 kernel mode ，拥有系统中的最高权限
- 因此， Exception 意味着 control flow 在 user mode 和 kernel mode 之间的切换 (transfer) ，即进出内核

- User and kernel modes
  - User mode: 有限制，有些指令没有权利执行
    - 例如 I/O 请求
  - 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

Exception Handler



ECF and Process

- ECF 是支撑“进程”的关键机制
  - 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 ECF
  - 进程对各种事件和故障的处理也依赖 ECF
  - 进程执行 system call 也依赖 ECF

Exception 分类

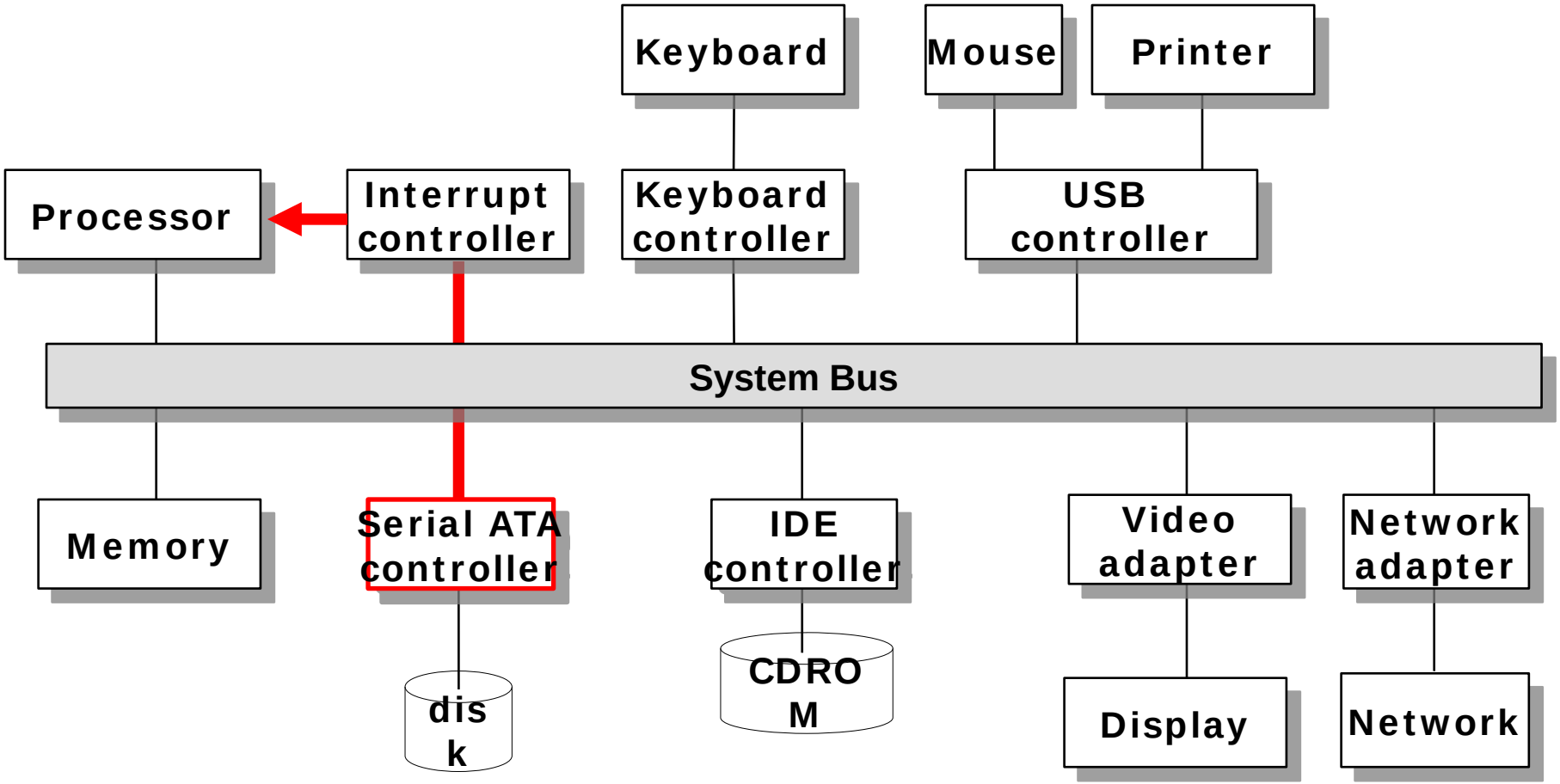
- 中断（interrupt）
- 陷阱（trap）
- 故障（fault）
- 终止（abort）

Interrupts

- 中断（interrupt）
  - 来自 CPU 外部，如 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 (pin) 接受中断请求
  - 对应的 exception handler 也叫 interrupt handler

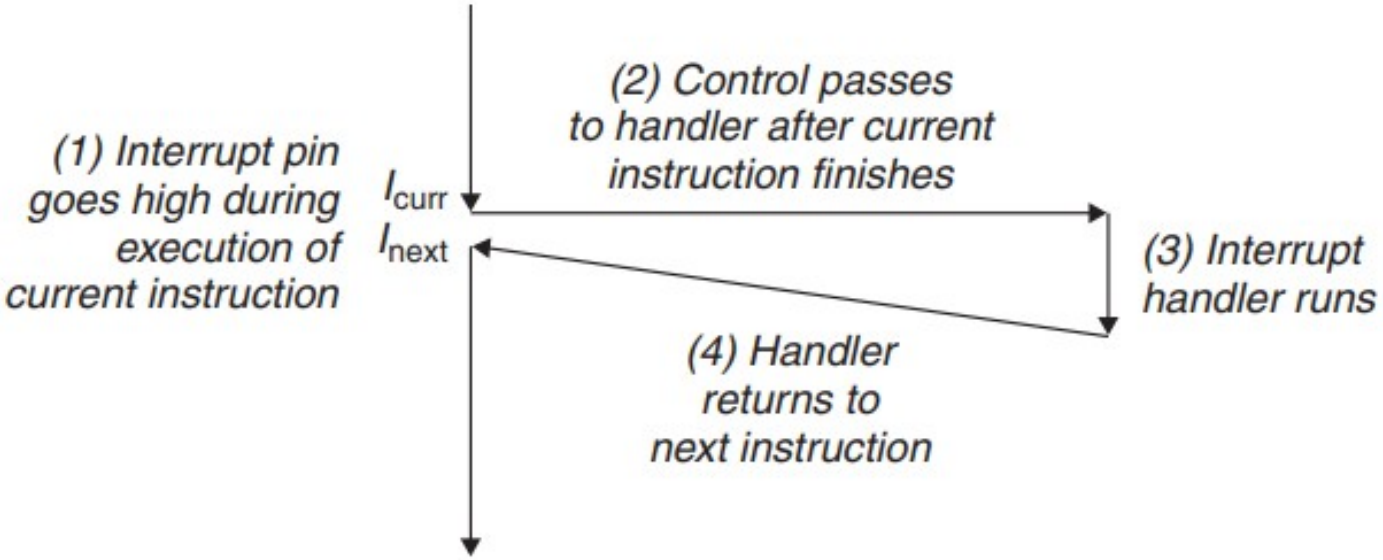
Interrupts

外部设备将 exception number 发到 system bus 上，并通过 interrupt controller 将 CPU 某个 pin 上的电平拉高，以通知 CPU 中断的发生



Interrupts

- 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
- 中断可以嵌套（中断处理程序可被更高级别中断所中断）

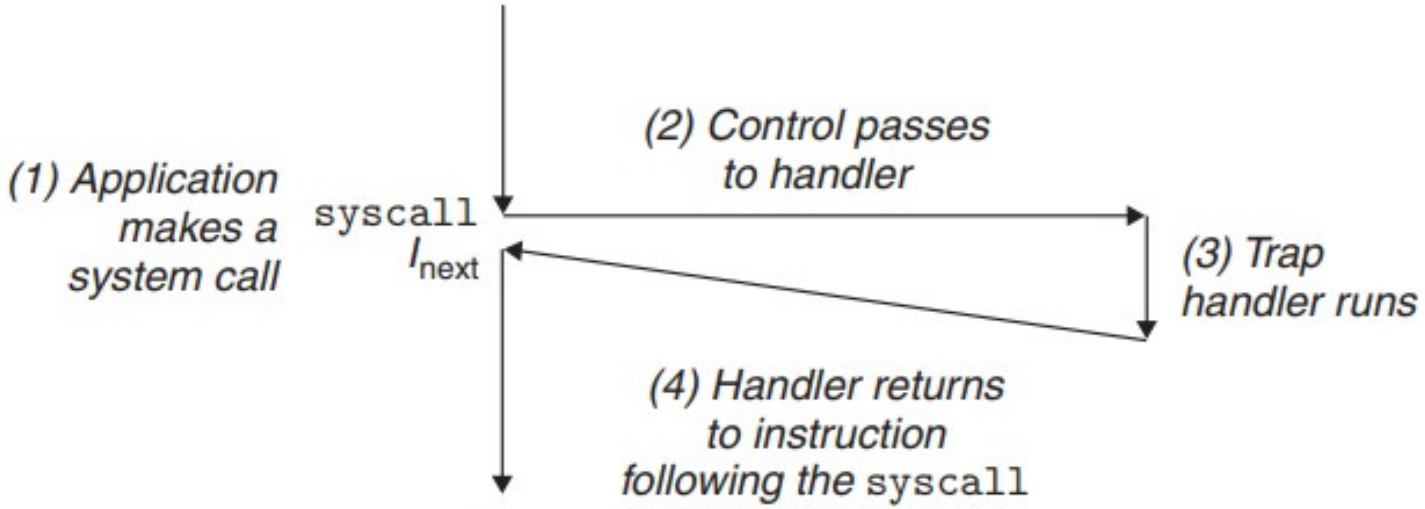


Traps

- 陷阱（trap）
  - 有意的异常，是执行一条特殊指令（syscall 或 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）

Traps

- 陷阱（trap）
  - 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



Traps and System Calls

- Exceptions in x86-64 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

Traps and System Calls

- 每个 system call 有一个由 OS 定义的 syscall number
- System calls in Linux x86-64:

Number	Name	Description	Number	Name	Description
0	`read`	Read file	33	`pause`	Suspend process until signal arrives
1	`write`	Write file	37	`alarm`	Schedule delivery of alarm signal
2	`open`	Open file	39	`getpid`	Get process ID
3	`close`	Close file	57	`fork`	Create process
4	`stat`	Get info about file	59	`execve`	Execute a program
9	`mmap`	Map memory page to file	60	`_exit`	Terminate process
12	`brk`	Reset the top of the heap	61	`wait4`	Wait for a process to terminate
32	`dup2`	Copy file descriptor	62	`kill`	Send signal to a process

Traps and System Calls

- x86 中所有的 system call 都通过 syscall (int 0x80) 指令触发
- Linux 在启动时在 kernel space 中会创建一个类似于 exception table 的查找表，用于根据 syscall number 查找 syscall handler 的入口

System Call Example #1

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

1 是 stdout 的 file descriptor

System Call Example #1

初始化常量、开始执行 main 函数

code/ecf/hello-asm64.sa

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

System Call Example #1

L9 将 syscall number 传入 rax 寄存器， L10-12 将 write 的 3 个参数传入寄存器， L13 调用 syscall wirte ， L14-16 完成调用 syscall exit

```
        First, call write(1, "hello, world\n", 13)
 9      movq $1, %rax           write is system call 1
10      movq $1, %rdi           Arg1: stdout has descriptor 1
11      movq $string, %rsi      Arg2: hello world string
12      movq $len, %rdx         Arg3: string length
13      syscall                 Make the system call

        Next, call _exit(0)
14      movq $60, %rax          _exit is system call 60
15      movq $0, %rdi           Arg1: exit status is 0
16      syscall                 Make the system call
```

code/ecf/hello-asm64.sa

system call 传参通过寄存器，而非 stack 完成

System Call

- 应用程序离不开 system call （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， System call= 升级权限 + 跳转
- 理解汇编可以帮助理解 OS

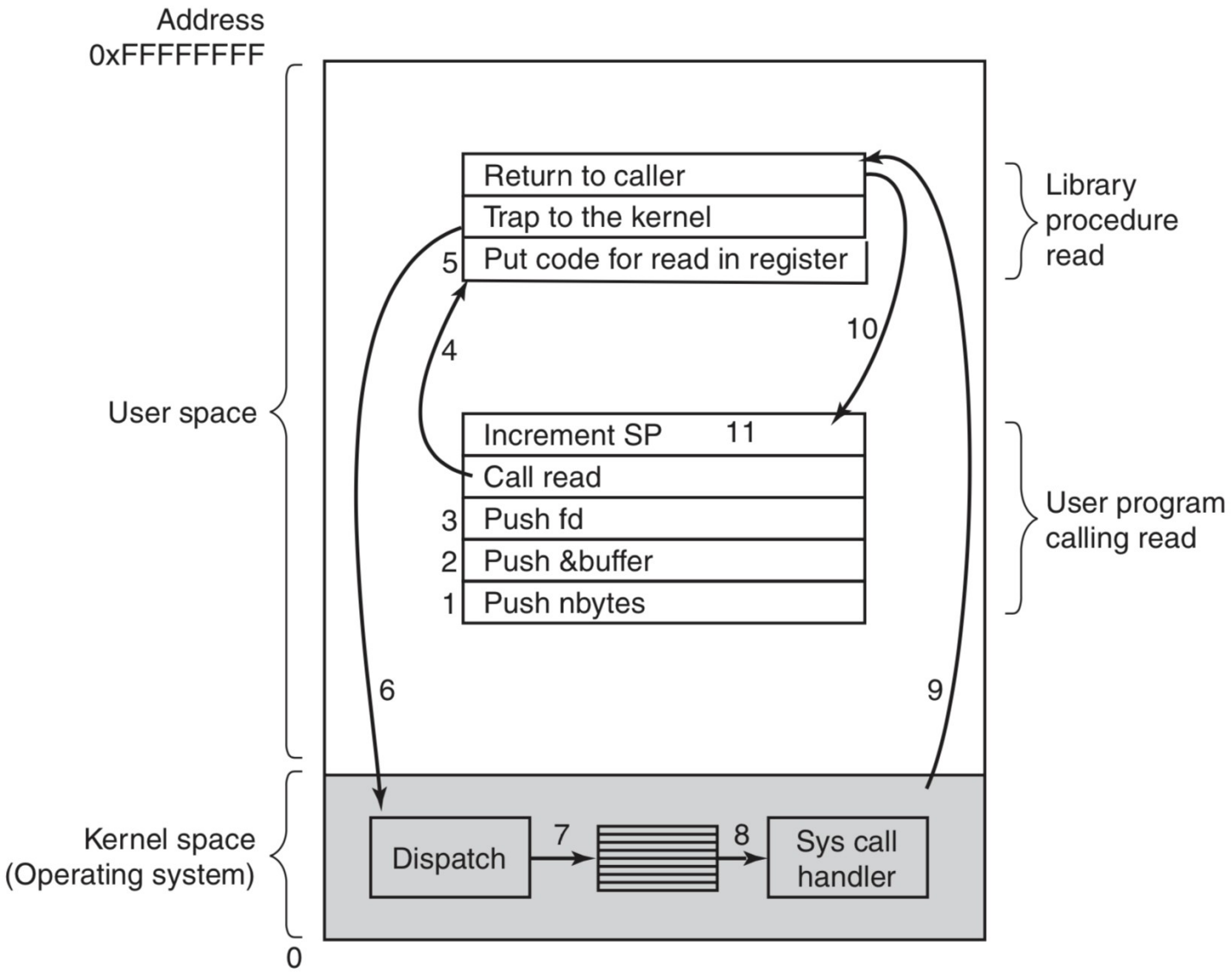
System Call vs. Procedure Call

- 系统调用（例如 open() ）看起来就像函数调用，有什么区别呢?
  - open() 相当于 OS 提供的库函数
  - 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 handler 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 syscall number 给内核

System Call Example #2

- 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 Example #2



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

Lab0. Kernel Lab

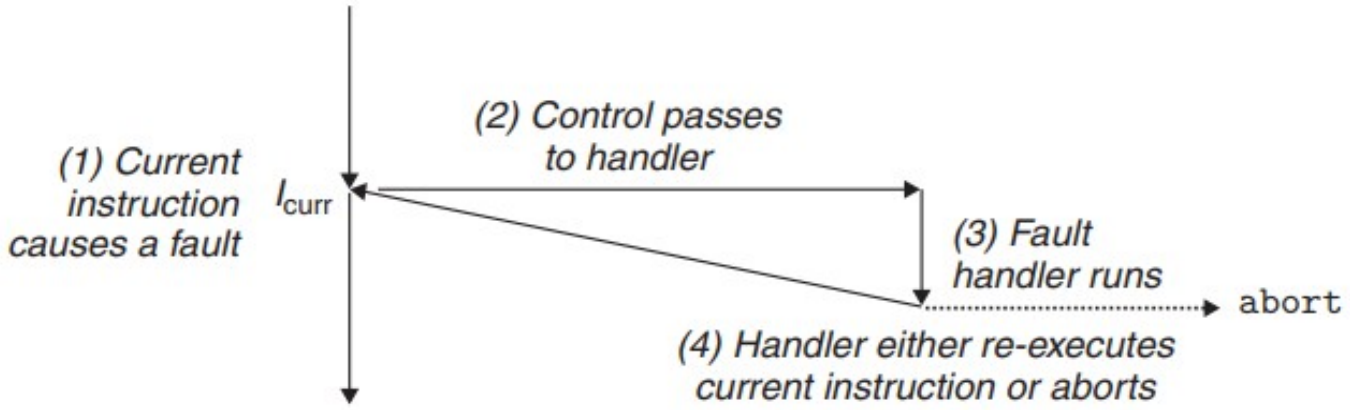
- 在 Linux 中增加一个自己的 system call
  - #0. 搭好 Linux 虚拟机环境;
  - #1. 学会编译内核;
  - #2. 学会用 ftrace 跟踪内核函数调用
  - #3. 在 Linux 内核中增加一个自己定义的系统调用
- 选做，不计分

Faults

- 故障（Fault）
  - 指系统中一种有可能被修正的 error
  - 故障由某条指令的执行引发，因此也是**同步发生**的
  - 相应的 exception handler 也称为 fault handler

Faults

- 故障（Fault）
  - 故障发生时，处理器将控制转移给 fault handler
    - 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 routine(例程)， abort routine 会终止引起故障的应用程序



Fault Example: Page Fault

- Page fault
  - User writes to or reads from 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
```

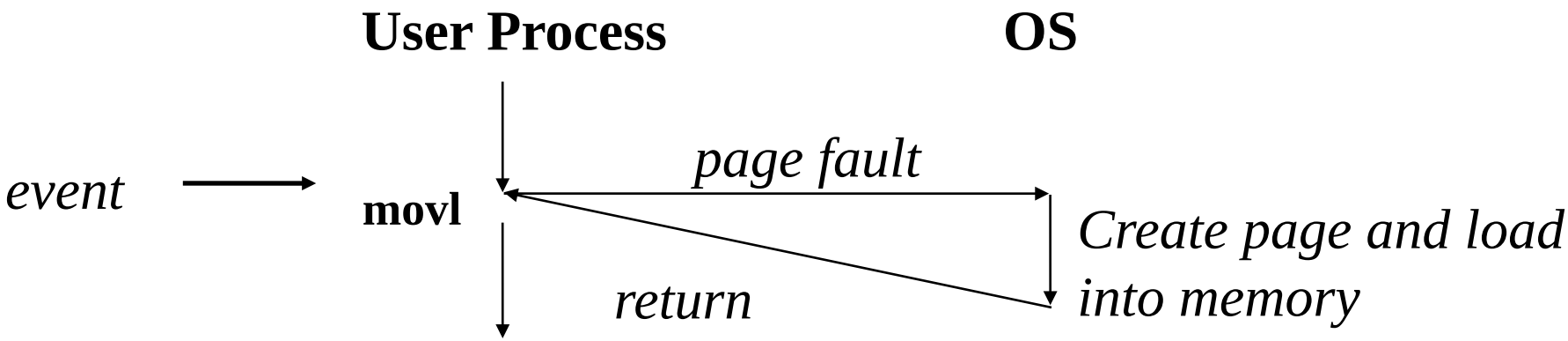
Fault Example: Page Fault

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

Fault Example: Page Fault

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

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



Fault Example: Page Fault

- Page fault
  - 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
```

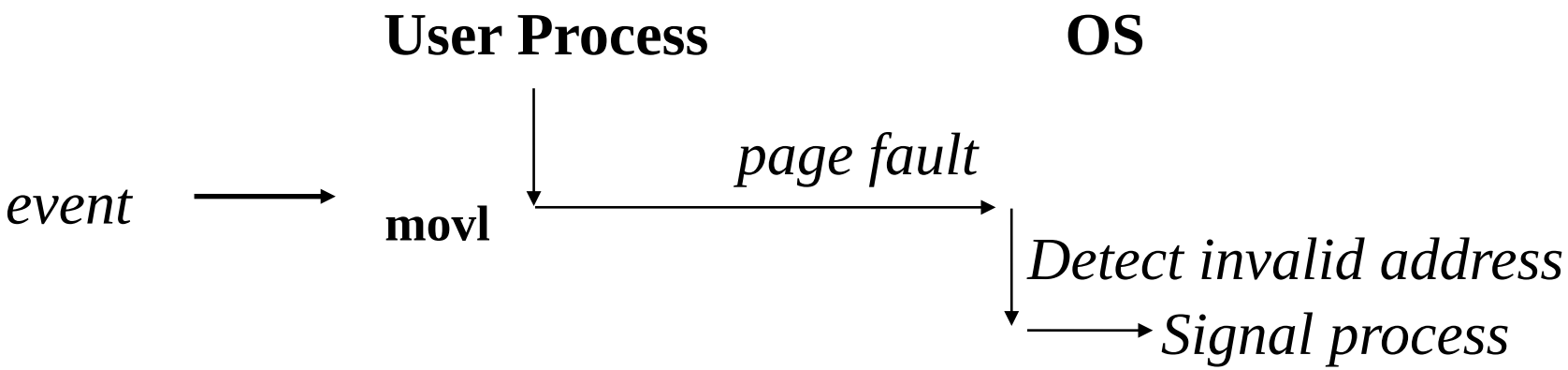
Fault Example: Page Fault

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

Fault Example: Page Fault

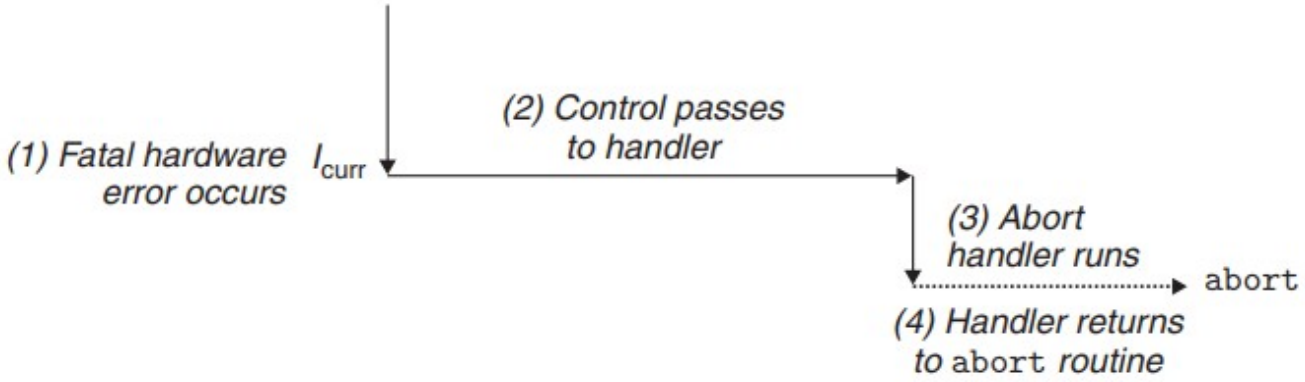
```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

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



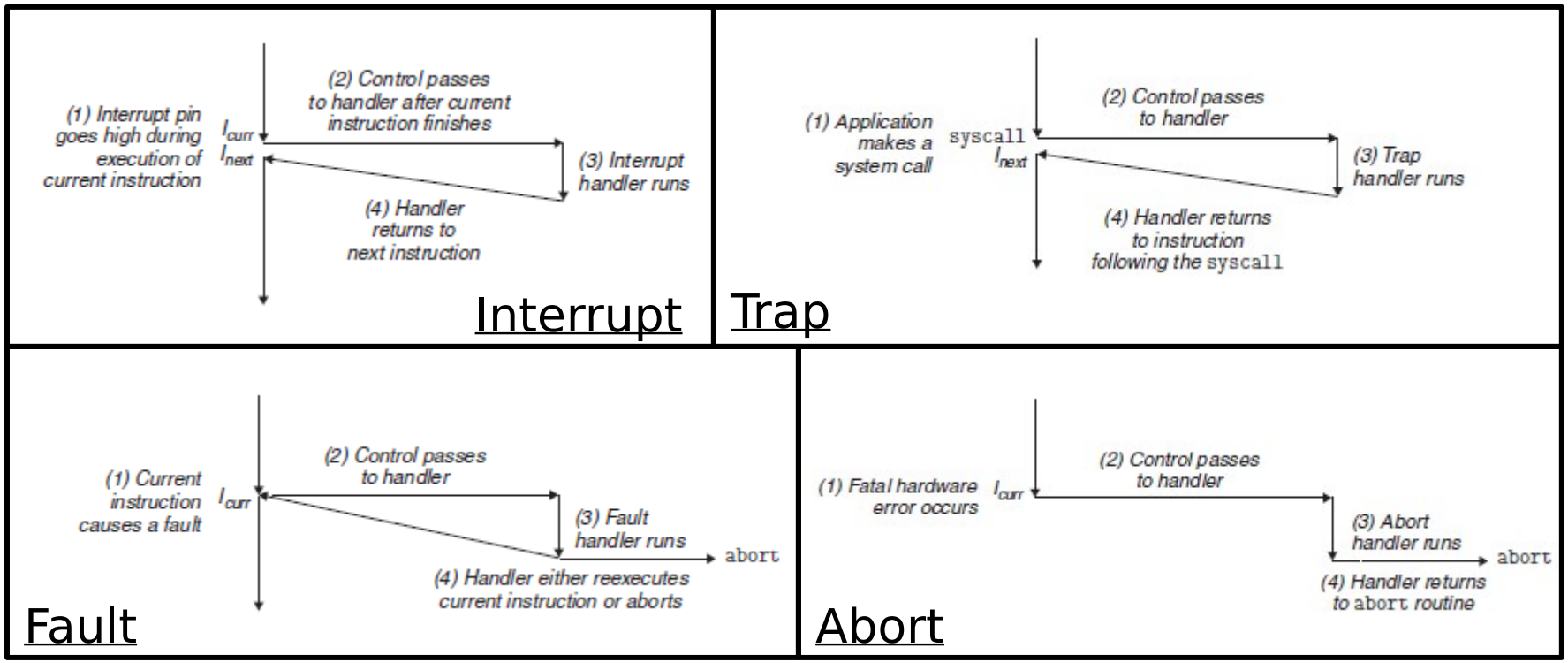
Aborts

- 终止（abort）
  - 不可修复的错误导致
  - 一般是硬件错误，比如 DRAM 或 SRAM 中的位被损坏时发生的奇偶校验错误
  - 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

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

Processes

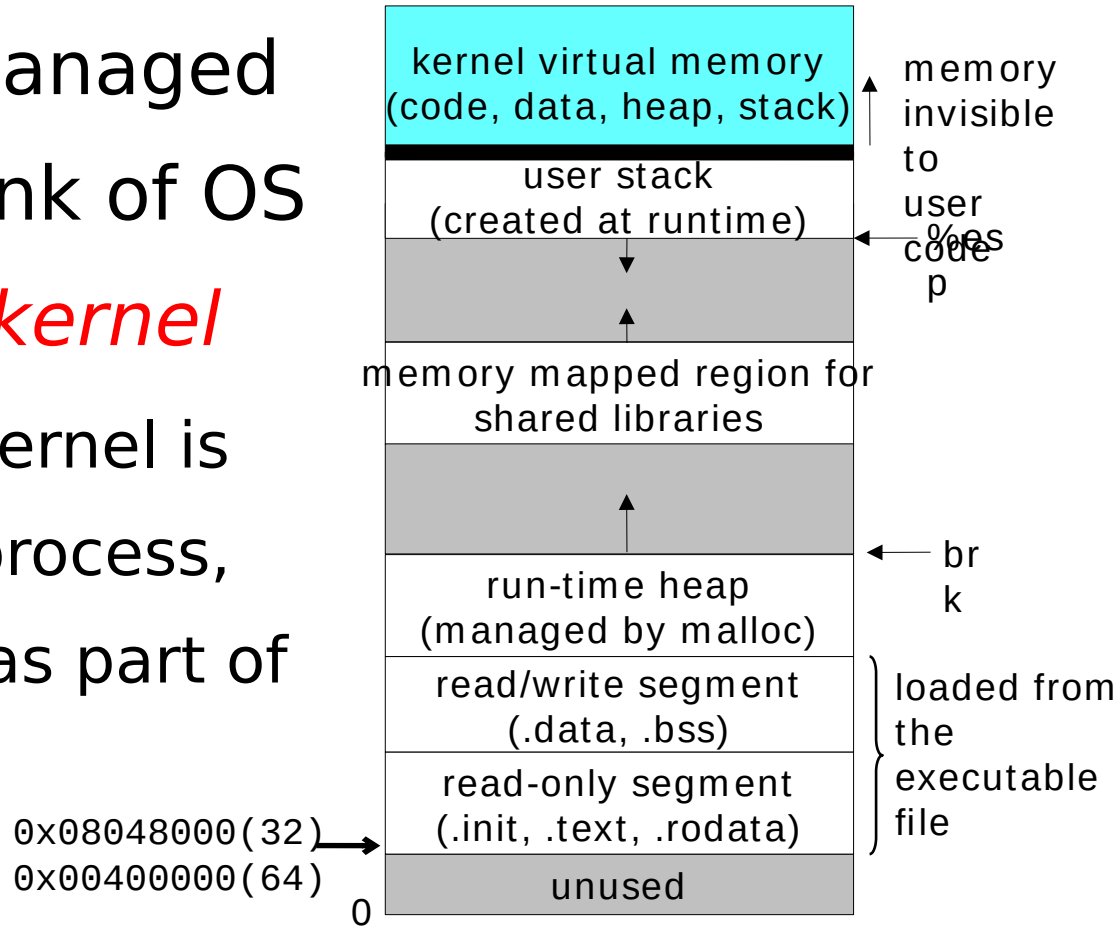
- 定义 : A *process* is an instance of a running program.

- Process 给每个程序提供两个关键的抽象:
  - 逻辑控制流 (logical control flow)
    - 让每个进程感到自己是独自使用 CPU 的
  - 私有的地址空间 (private address space)
    - 让每个进程感到自己是独自使用整个内存的

Kernel and User Modes

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 user process



User and Kernel Modes

- CPU 的某个 control register 中有一个 mode bit
- mode bit 置位时，进程处于 kernel mode
- 否则，进程处于 user mode

User and Kernel Modes

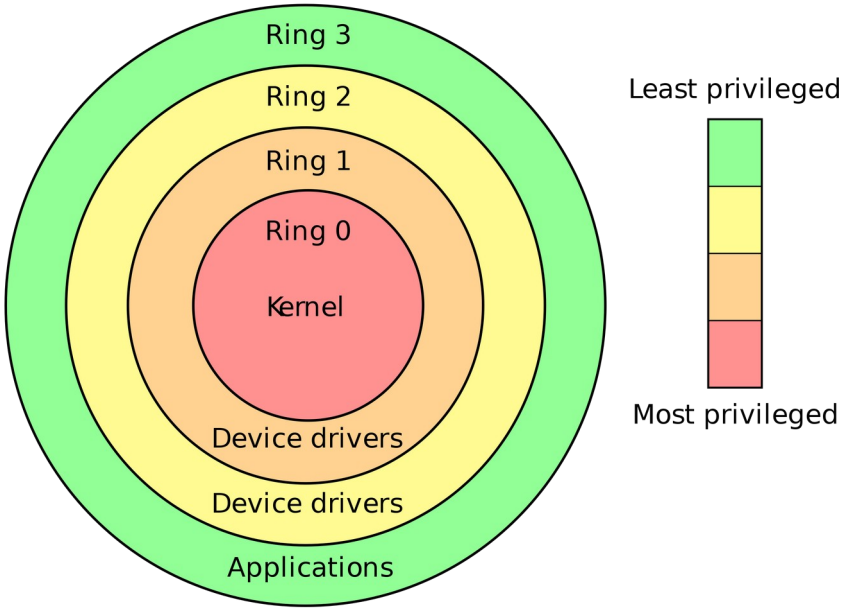
- 运行在 kernel mode 的进程能够
  - 执行 CPU 指令集中的任何指令
  - 能否访问系统中的任何内存地址（物理地址访问）
- 运行在 user mode 的进程
  - 既不能执行特权指令
  - 也不能直接访问地址空间中 kernel 区域的数据和代码

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



Linux Processes

- Linux 在启动时会创建一个 pid 为 0 的特殊进程，该进程会创建 pid 为 1 的 init 进程和 pid 为 2 的 kthreadd 进程
- init 是所有用户进程的父进程
- kthreadd 是所有内核线程的父进程
  - 内核线程约等于内核进程
    - Linux 中线程是作为 lightweight process 实现的，有独立的 pid ，只是 share （而不是 copy-on-wirte ）父进程的地址空间

kernel threads（内核线程）

- 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
- Linux 中内核线程
  - 运行在 kernel mode，具有特权
  - 没有用户地址空间
  - 共享同一个内核地址空间
    - Linux 中所有进程 / 线程都共享同一块内核地址空间

内核线程 vs. 用户进程

- 内核线程与用户进程的区别
  - 内核线程只运行在内核态，而用户进程可以运行在用户态，也可以运行在内核态
  - 内核线程没有用户地址空间，不能使用用户态内存
  - 内核线程和用户进程一样，被 OS 调度
  - 内核线程必须由 kernel 创建，用户可以编写 kernel extension 或者 driver ，在其中创建内核线程

Context Switches

Context switching

- Kernel 维护了每个进程的上下文（context）
- Context 是指进程在恢复运行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

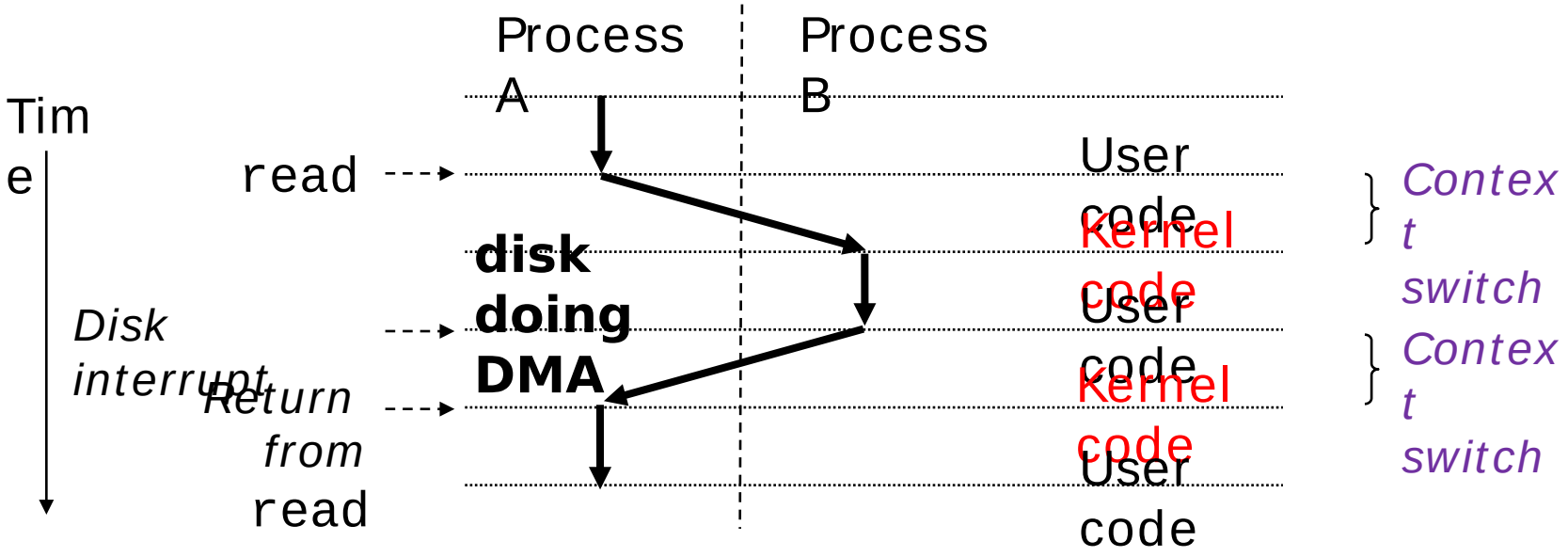
- 指进行多任务切换（例如通过时间片）的机制

- 保存当前进程 A 的 context，恢复另一个进程 B 的 context，将 CPU 控制权转交给 B

Context switching

- Context switch 在什么情况下发生呢?
  - 执行 system call
    - **`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 （只要发生 system call ，都要强制触发 context switch ）
  - 发生 Interrupt
    - 最常见的是时间片轮转中的 Timer interrupt
    - I/O 设备完成操作发生中断
  - 发生 Fault
    - 如常见的 page fault

Context switching



Context switching

- Scheduler （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)
  - 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

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) {
•      unix_error("fork error",
•                 strerror(errno));
3      exit(0);
4  }
```

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

课堂练习

- 思考：
  - 当中断或系统调用把控制转给操作系统时，通常将内核堆栈和被中断进程的用户堆栈分离。为什么？

- **Kmalloc：连续，不超过128KB**
- **Kzmalloc： kmalloc+ 清零**
- **Vmalloc：不连续，大小没限制**

课堂练习

- A computer system has enough room to hold five programs in its main memory. These programs are idle waiting for I/O half the time. What fraction of the CPU time is wasted?

CPU utilization = 1 − *p*^*n* ， *p* *是 I/O 比例， n 是通道数（并发度）。*
从概率角度看，各进程 I/O 阻塞的时间可能碰上，这时 CPU 就闲置了
1-0.5^5 = 96.88%

课堂练习

- A computer has 4 GB of RAM of which the OS kernel occupies 512 MB. The processes are all 256 MB (for simplicity) and have the same characteristics. If the goal is 99% CPU utilization, what is the maximum I/O wait that can be tolerated?

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
  - 访问一块内存空间，对应的内存页不在物理内存中