



计算机系统

# 第8章 (上)

## 异常与进程

湖南大学

《计算机系统》课程教学组

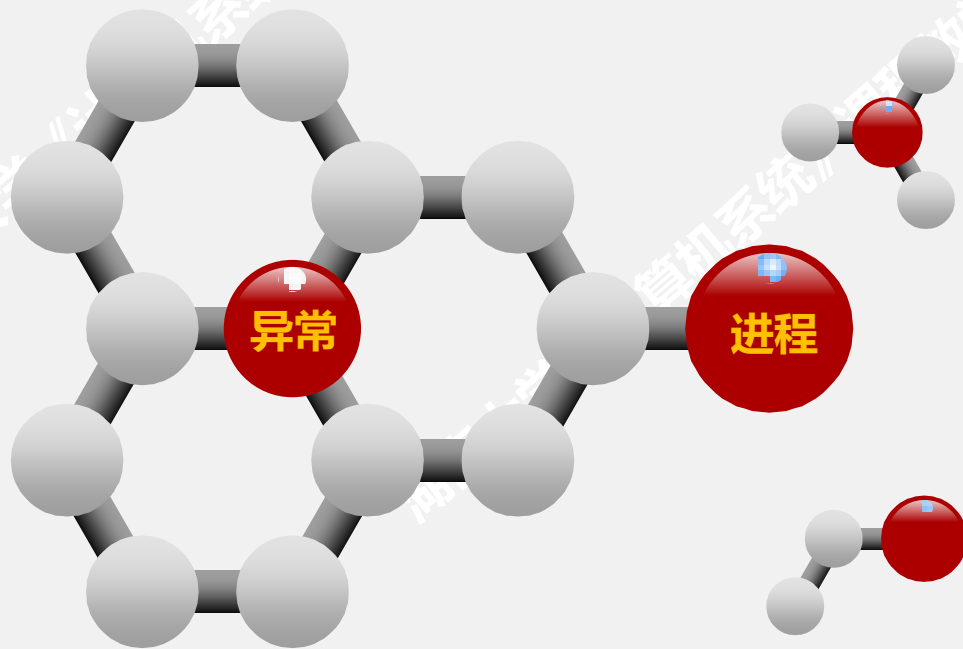
主讲：肖雄仁

# 世界冠军教你什么是异常



- ▶ 世界冠军的拼搏精神
- ▶ 羽毛球比赛的正常接球流程——控制流
- ▶ 羽毛球拍线断了——硬件信号
- ▶ 换拍——异常处理

# 本讲内容

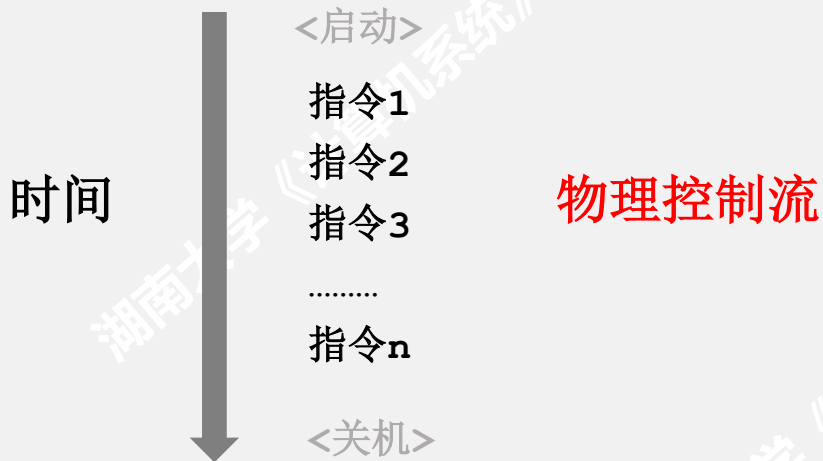


异常控制流

进程与进程控制

## 处理器(CPU)的工作:

- ▶ 从加电开始工作, 到断电停止工作, CPU就是读并执行 (解释) 指令序列:  $a_0, a_1, \dots, a_n$ ,  
一次执行一条指令



- ▶ 这个序列就是处理器的控制流(*control flow*)

# 改变控制流

- 到目前为止，有两种机制可改变控制流
  - 分支与跳转 Jumps and branches
  - 调用与返回 Call and return
  - 上述两者均是响应程序状态的变化
- 对于一个实际应用的系统，需要对系统状态的变化进行响应
  - 来自硬盘或网络的数据已到达
  - 试图除以零的指令
  - 用户在键盘上使用Ctrl-C 组合键盘
  - 系统计时器终止
- 系统需要一种机制来处理上述控制流的突变——“异常控制流 (Exceptional Control Flow, ECF)”

## ▶ ECF存在于计算机系统的各个层次中

## ▶ 相对底层的机制

### ▷ 异常 ( Exceptions )

▶ 响应系统事件，例如改变系统状态导致的控制流变化

### ▷ 由硬件和操作系统软件相结合执行

## ▶ 较高层的机制

### ▷ 进程上下文切换 (Process context switch)

### ▷ 信号量 (Signals)

### ▷ 非局部跳转(Nonlocal jumps: setjmp/longjmp)

### ▷ 或者由操作系统软件执行，如 上下文切换和信号控制

### ▷ 或者由C语言动态库执行，如非本地跳转

# 程序员理解ECF很重要

## ► 帮助理解重要的系统概念

- ▷ 实现 I/O、进程和虚拟内存的基本机制

## ► 帮助理解应用程序是如何与操作系统交互的

- ▷ 陷阱(trap) 或者系统调用(system call)

## ► 帮助编写有趣的新应用程序

- ▷ 创建新进程、等待进程终止、通知其他进程系统中的异常事件
- ▷ 检测和响应这些事件

## ► 帮助理解并发

- ▷ 实现并发的基本机制

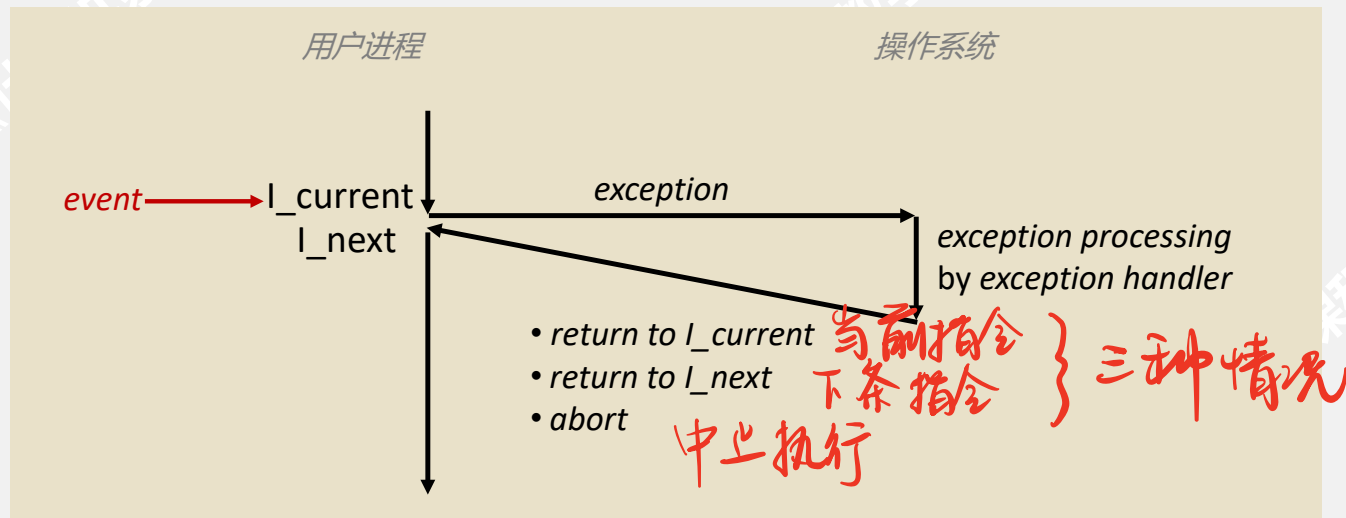
## ► 帮助理解**软件异常**如何工作

- ▷ 进行非本地跳转（即违反通常的调用 / 返回栈规则的跳转）来响应错误情况

## 8.1 异常(Exceptions)

### ► 一个异常是控制流突变

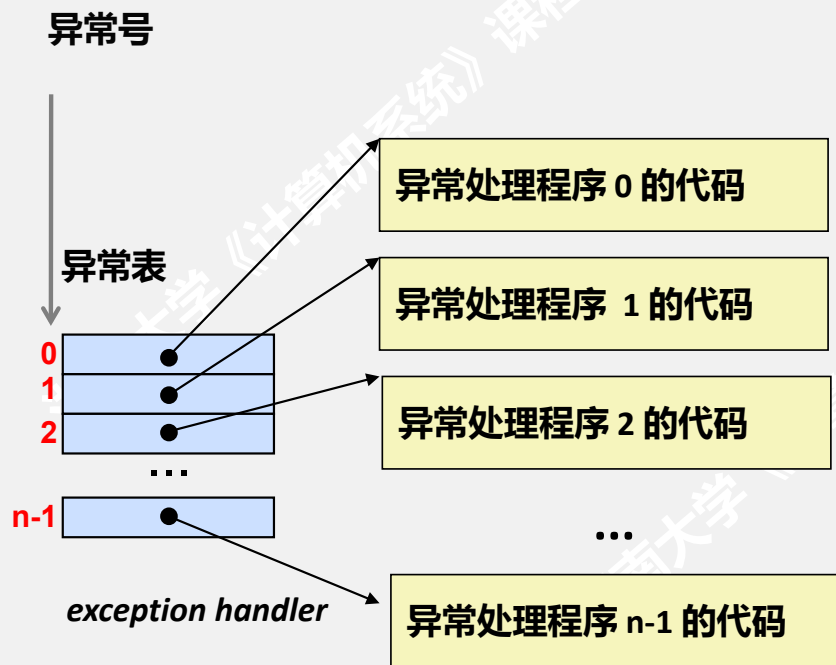
▷ 响应某些事件（如处理器状态的改变）而将控制交给操作系统的控制流转移



### ► 例如:

- ▷ 试图除以零, 算术溢出, 虚拟内存缺页
- ▷ I/O 请求完成, 使用了Ctrl-C组合键





## 硬件和软件紧密合作处理异常

- ▶ 每一种类型的事件都对应一个唯一的**异常号k**
- ▶ 根据 k 值，索引进入异常表
- ▶ 条目 k 包含的异常处理程序调用地址
- ▶ 在每次发生异常K时都会调用异常处理程序K

异常类似过程调用，不过有一些区别

- 返回地址
- 压栈的信息和内核栈
- 内核模式

# IA32异常与中断表

异常号	描述	异常类型
0	除以零	故障 Fault
13	一般保护故障	故障 Fault
14	页故障	故障 Fault
18	机器检查	终止 Abort
32-127	操作系统定义的异常	中断或陷阱 Interrupt or trap
128 (0x80)	系统调用	陷阱 Trap
129-255	操作系统定义的异常	中断或陷阱 Interrupt or trap

异常可以分为四类：

**故障(fault)、终止(abort)、中断(interrupt) 和陷阱(trap)**

可在INTEL官网手册看到更多异常表细节：

[Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A](#)

# 异步异常——中断(Interrupt)

## ► 由处理器的外部事件所引起的异常

- ▷ 通过置位（高电平）处理器的中断引脚信号来触发中断
- ▷ 中断处理程序执行完后返回到被中断的下一条指令

## ► 举例:

### ▷ I/O 中断

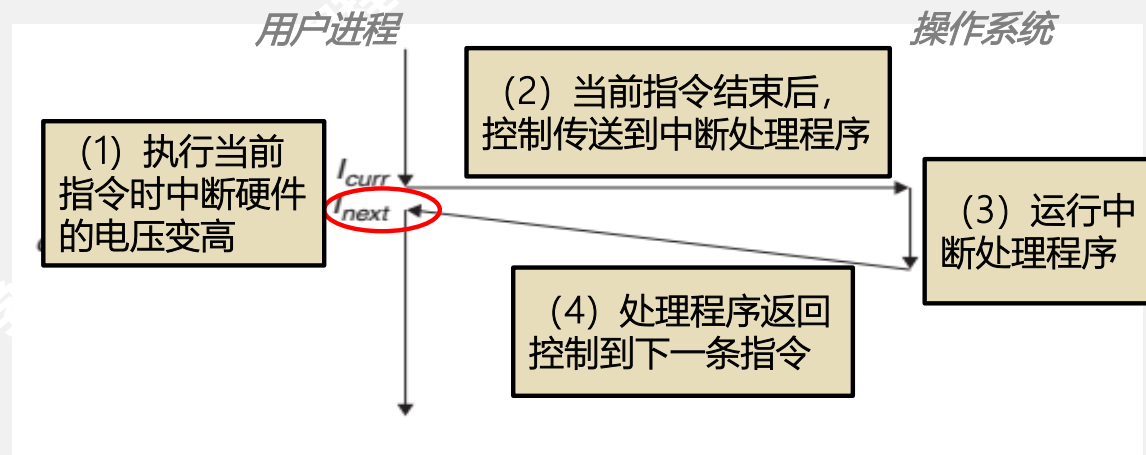
- 在键盘上敲入ctrl-c组合键
- 网络中数据包的到达
- 磁盘中数据扇区的到达

### ▷ 硬复位中断

- 按下复位按钮

### ▷ 软复位中断

- 在PC上键入 Ctrl-Alt-Delete



## ► 引起异常的事件是某条指令的执行结果:

### ▷ 陷阱 *Trap*

- 一种有意产生的异常
- 例如：系统调用, 断点陷阱, 其他特殊指令
- 将控制返回到“下一条”指令

### ▷ 故障 *Fault*

- 非有意产生、可能可恢复的异常
- 例如：页故障 (可恢复), 保护故障(不可恢复), 浮点异常
- 控制返回：或者再次执行“当前”指令或者终止

### ▷ 中止 *Abort*

- 非有意产生, 且不可恢复
- 例如: 奇偶校验错误, 机器校验
- 终止当前程序

### ► 向内核请求服务:

- 读文件 (read)
- 创建进程 (fork)
- 新的程序 (execve)
- 终止当前进程 (exit)

### ► `system_call n`

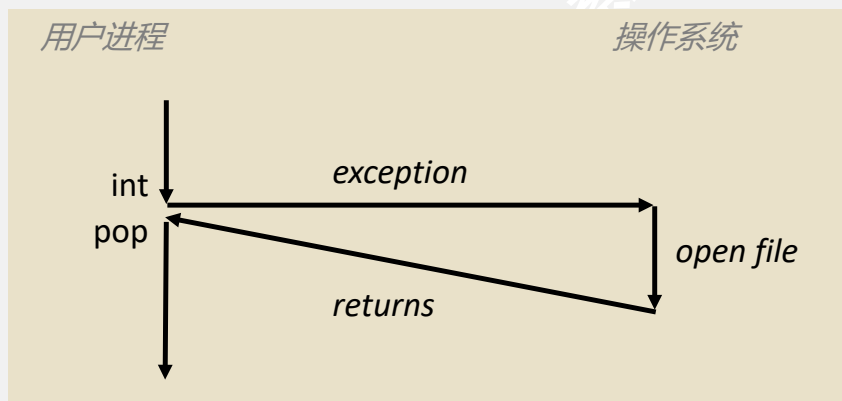
### ► 运行在内核模式

# Trap 示例：打开文件

用户调用：open(filename, options)

- ▶ 函数open执行系统调用指令int

```
0804d070 <__libc_open>:  
. . .  
804d082:  cd 80                int    $0x80  
804d084:  5b                   pop    %ebx  
. . .
```



- ▶ 操作系统必须找到或者创建文件，并使之读写做好准备
- ▶ 返回整型文件描述符

# IA32 系统调用(System Calls)

■ `int n` //中断指令, `n`为IA32异常号 (0-256)

■ `int`指令实现linux系统调用

`mov $5,%eax` //通过`eax`寄存器传递系统调用号

`int $0x80` //执行系统调用初始化

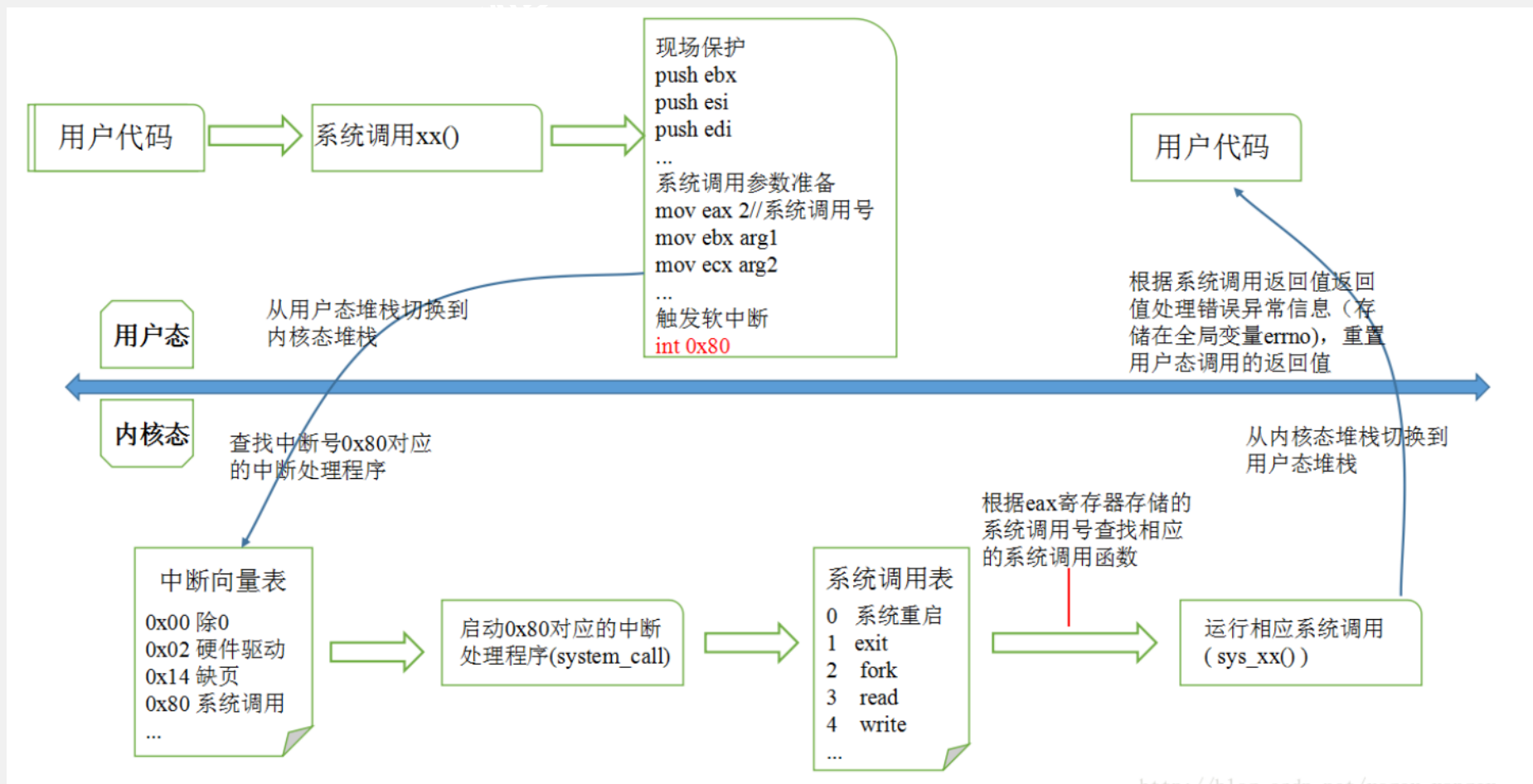
■ 每个IA32系统调用都有一个唯一的**整数号**: 对应内核中系统调用表的偏移量

■ Linux系统调用宏定义:

</usr/include/sys/syscall.h>

Number	Name	Description
1	exit	Terminate process
2	fork	Create process
3	read	Read file
4	write	Write file
5	open	Open file
6	close	Close file
11	execve	Execute a program
37	kill	Send signal to process
128	init_moudule	Init module to system call

# 系统调用示例：创建进程



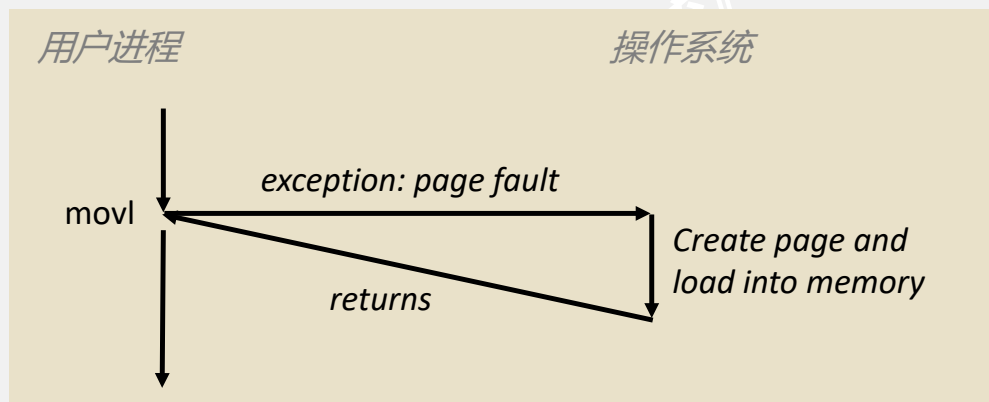
# Fault 示例: 页故障

## 存储器引用

- ▶ 用户写到存储单元
- ▶ 用户请求的存储空间 (page) 当前在磁盘上

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

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



- ▶ 页处理程序必须将页加载到物理主存
- ▶ 返回到发生页故障的指令
- ▶ 在第二次写尝试时成功命中



## Fault 示例：页故障

### ➤ “页故障”事件何时发现？如何发现？

- 执行每条指令都要访存（取指令、取操作数、存结果）
- 在保护模式下，每次访存都要进行逻辑地址向物理地址转换
- 在地址转换过程中会发现是否发生了“页故障”！

### ➤ “页故障”事件是软件发现的还是硬件发现的？

- 逻辑地址向物理地址的转换由硬件（MMU）实现，故“页故障”事件由硬件发现。
- 所有异常和中断事件都由硬件检测发现！

### ➤ 以下几种情况均会引发“页故障”

- 缺页：页表项有效位为0

← 可通过读磁盘恢复故障

- 地址越界：地址大于最大界限

- 访问越级或越权（保护违例）：

} 不可恢复，称为“段故障（segmentation fault）”

- 越级：用户进程访问内核数据（CPL=3 / DPL=0）

- 越权：读写权限不相符（如对只读段进行了写操作）

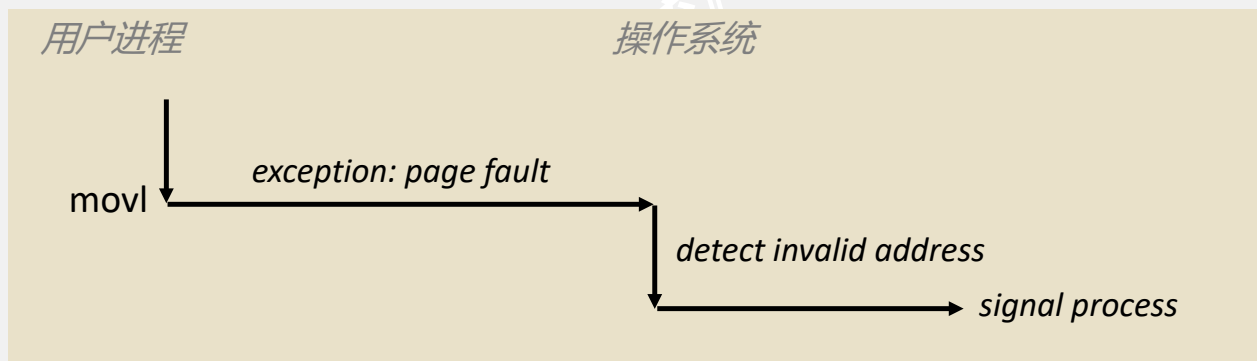
# Fault 示例: 无效的内存引用

## 存储器引用

- ▶ 用户写到存储单元
- ▶ 地址不合法

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

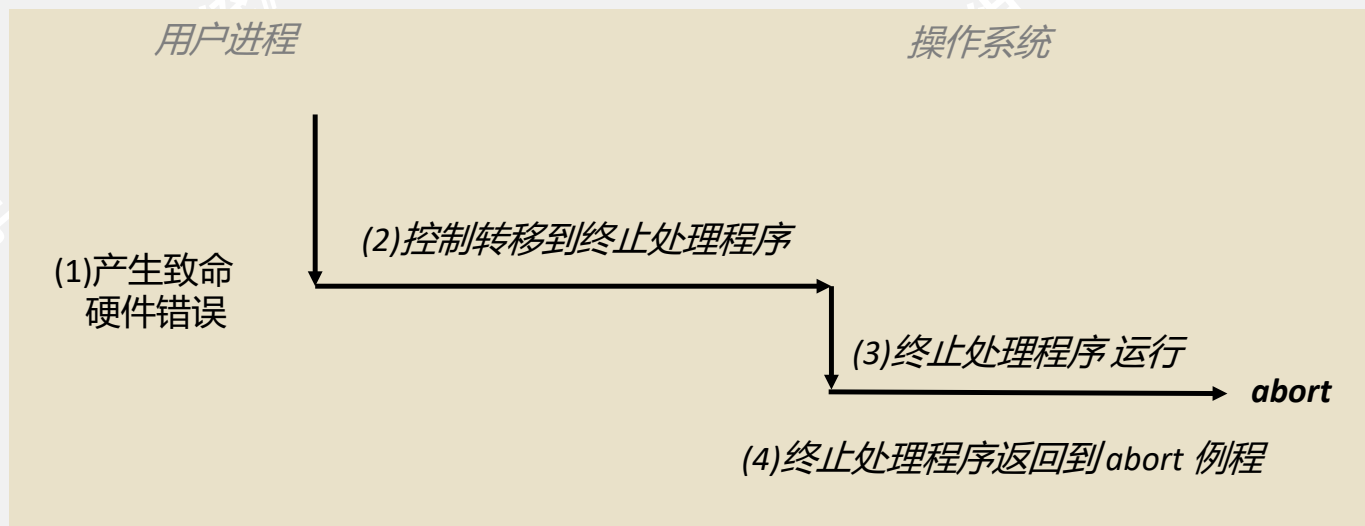
```
80483b7: c7 05 60 e3 04 08 0d movl $0xd,0x804e360
```



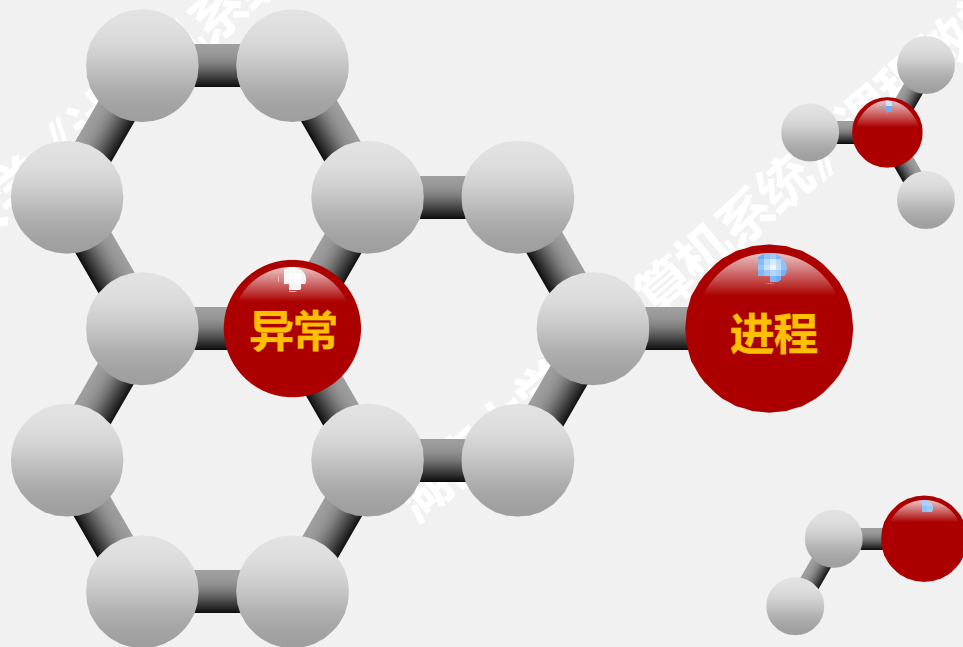
- ▶ 页处理程序检测到无效地址
- ▶ 发送 SIGSEGV 信号到用户进程
- ▶ 用户进程发生段错误 “segmentation fault” 退出

# Abort 示例

- ▶ 终止处理程序将控制传递给内核终止例程，结束当前应用程序



# 本讲内容



异常控制流

进程与进程控制

## 8.2 进程

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

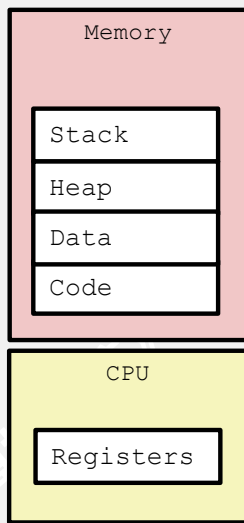
- ▷ 计算机科学领域最深刻最成功的概念之一
- ▷ 与“程序(program)”或“处理器(processor)”是不同的概念
  - 状态, 持久性, 组成

► **进程提供给每个程序两种重要抽象**

- ▷ 独立的逻辑控制流 Logical control flow
  - 每个程序看上去像在独占使用CPU
- ▷ 私有的虚拟地址空间 Private virtual address space
  - 每个程序看上去独占使用主存

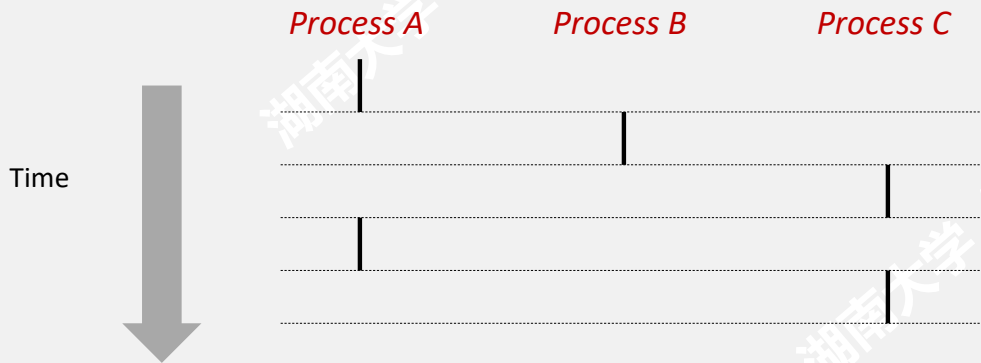
► **这些表象如何实现?**

- ▷ (多任务) 进程交替运行在一个单独的处理器核上
- ▷ 地址空间由虚拟存储器系统管理 (第9章)



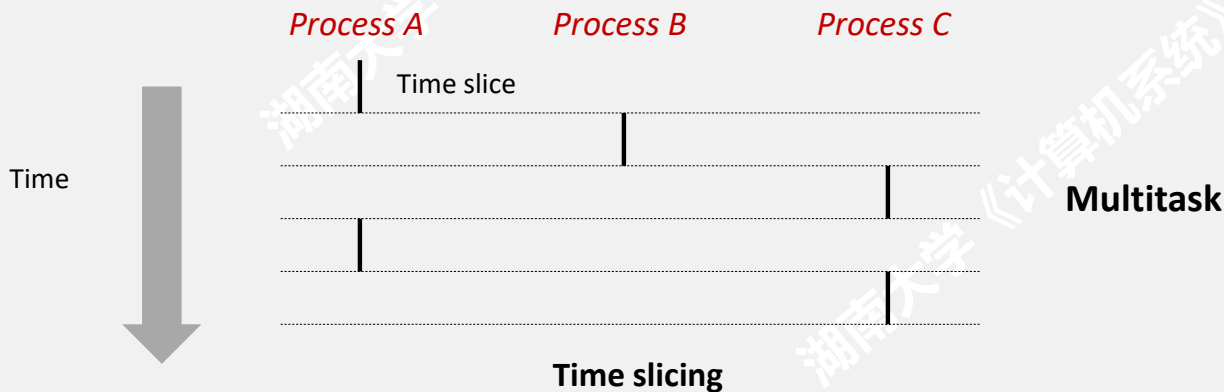
# 逻辑控制流

- ▶ 程序计数器(PC)值
  - ▷ 对应在可执行目标文件中的指令或动态链接到程序的共享对象中的指令
- ▶ PC值的序列叫做逻辑控制流，或者简称逻辑流
- ▶ 每个进程都有自己的逻辑控制流
- ▶ 逻辑流的执行是交错的



# 并发进程

- ▶ 如果两个进程的控制流在时间上重叠，它们**同时/并发**（concurrency）运行，
- ▶ 否则它们就是**顺序的**
- ▶ 例如（在单核上运行）：
  - ▷ 并发: **A & B, A & C**
  - ▷ 顺序: **B & C**



# 从用户角度看并发进程

## ▶ 并发进程的控制流在时间上不相交

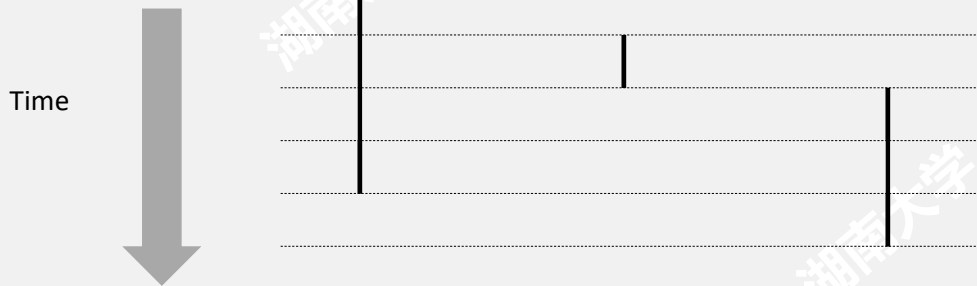
- ▷ 处理器时间片切换时间很短，可将并发进程**视为**彼此之间并行执行
- ▷ 并发流(concurrent flow)与处理器核数与计算机数无关，两个流在时间上重叠

## ▶ 并行流

- ▷ 两个流并发地运行在不同的处理器核或者计算机上
- ▷ 它们并发地运行，且并行地执行

## ▶ 并发 VS 并行

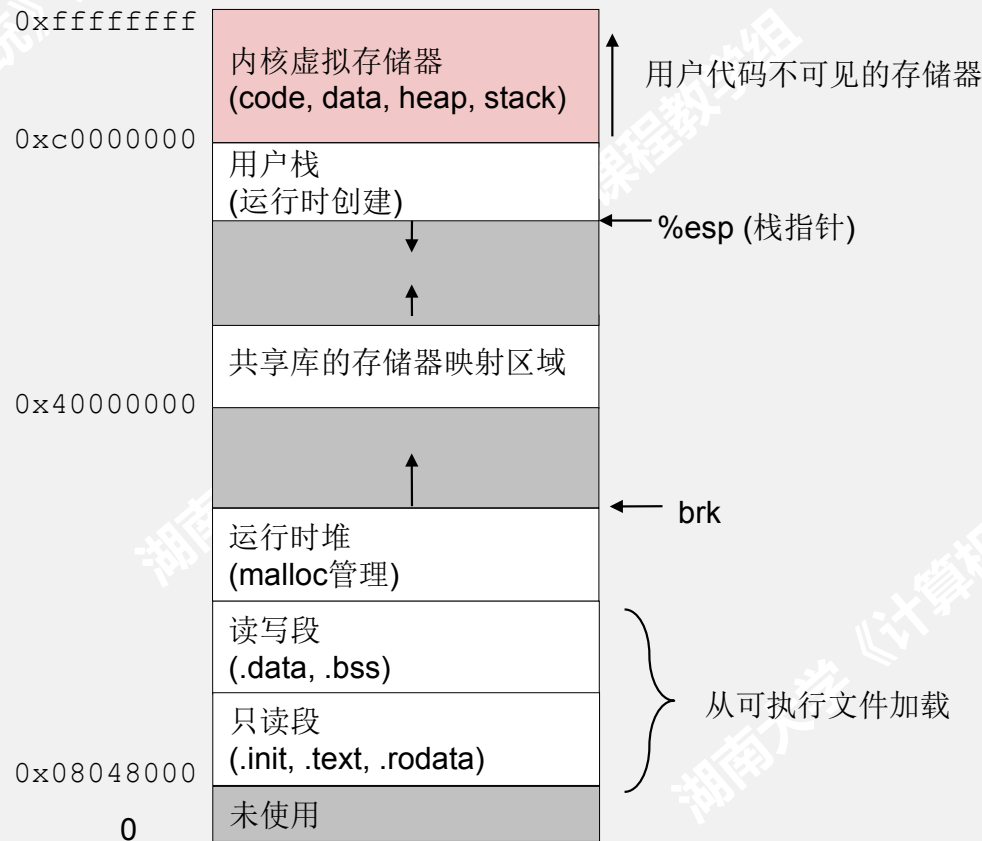
宏观 微观





# 私有地址空间

- ▶ 每个进程都有自己的私有地址空间。



## 处理器限制应用程序可以执行的指令和访问的地址空间范围

- ▶ 处理器通过**控制寄存器**中的一个**模式位**来提供这个功能
  - ▷ 设置了模式位后，进程就运行在内核模式中
  - ▷ 没有设置模式位时，进程运行在用户模式
- ▶ 进程从用户模式转变到内核模式的方法
  - ▷ 通过中断，故障，陷阱系统调用这样的异常
  - ▷ 在异常处理程序中进入内核模式
  - ▷ 退出后返回用户模式
- ▶ Linux提供一种聪明的机制，叫/proc文件系统
  - ▷ 允许用户模式进程访问内核数据结构的内容
  - ▷ 将内核数据结构输出为用户程序可以读的文本文件的层次结构

# 上下文切换

- ▶ 进程由共享的操作系统代码块（**内核-*kernel***）管理
  - ▷ 注意: 内核不是一个单独的进程，而是作为用户进程的一部分来运行
- ▶ 内核中调度器在进程执行时，重新开始一个先前被抢占的进程（调度）
- ▶ 通过**上下文切换**(*context switch*)控制流从一个进程转移到另一个进程。

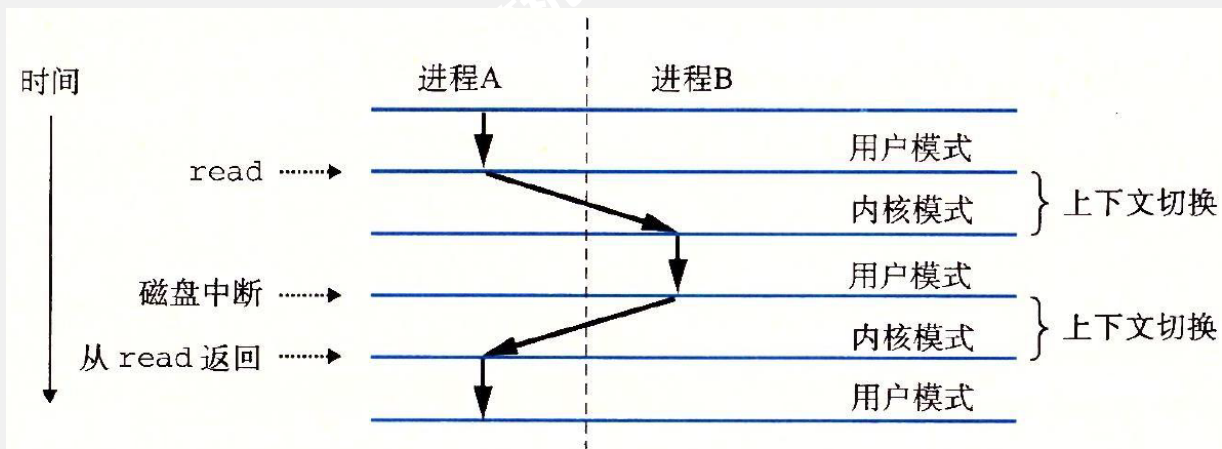


图 8-14 进程上下文切换的剖析

## 8.4 进程控制

### ► 获取进程ID

- ▷ PID是每个进程唯一的进程ID（非零的正数）。
- ▷ getpid()返回调用进程的PID，getppid()返回它的父进程的PID

### ► 进程总是处于三种状态

- ▷ **运行**，进程要么在CPU中执行，要么等待执行，最终被内核调度。
- ▷ **停止**，进程的执行被挂起(suspend)，且不会被调度
  - 收到SIGSTOP,SIGTSTP,SIDTTIN或者SIGTTOU信号，进程停止。
  - 直到收到一个SIGCONT信号，进程再次开始运行。
  - 信号是一种软件中断的形式（8.5节）
- ▷ **终止**，永远停止
  - 收到一个信号，该信号的默认行为是终止进程
  - 从主程序返回
  - 调用exit 函数。

# fork:创建新进程

► `int fork(void)`

▷ 创建一个与调用进程（父进程）相同的新进程（子进程）

▷ 返回0给子进程

▷ 返回子进程的PID（进程ID）给父进程

```
pid_t pid = fork();  
if (pid == 0) {  
    printf("hello from child\n");  
} else {  
    printf("hello from parent\n");  
}
```

► fork 非常有趣(但经常引起混乱)因为被调用一次却返回两次

# 理解fork

进程n



```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

pid = m

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```



```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

子进程m



```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

pid = 0

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```



```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

hello from parent    上述二者谁先完成?    hello from child

# fork 示例1

## ▶ 父进程与子进程运行相同的代码

▷ 通过fork的返回值区别父进程与子进程

## ▶ 开始于相同的状态，但父/子进程都有自己的私有地址空间

▷ 包括共享的输出文件描述符

▷ 它们的打印输出语句的相对顺序未定义

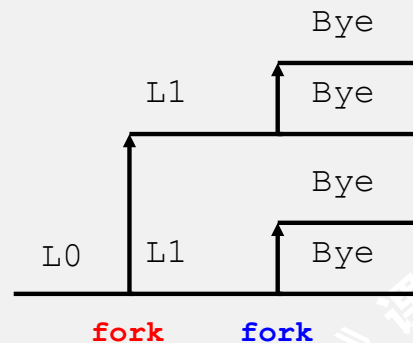
```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

Child has x = 2  
Parent has x=0

## fork示例2

- ▶ 连续两次forks, 父/子进程均可连续调用fork函数

```
void fork2()  
{  
    printf("%d say:L0\n",getpid());  
    fork();  
    printf("%d say:L1\n",getpid());  
    fork();  
    printf("%d say:Bye\n",getpid());  
}
```

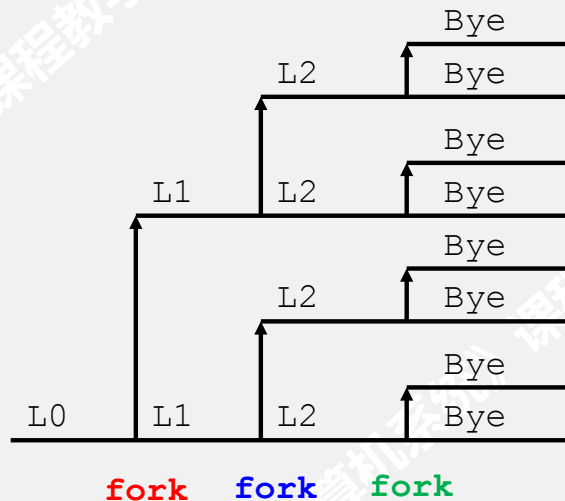




# fork示例3

## ► 连续三次forks

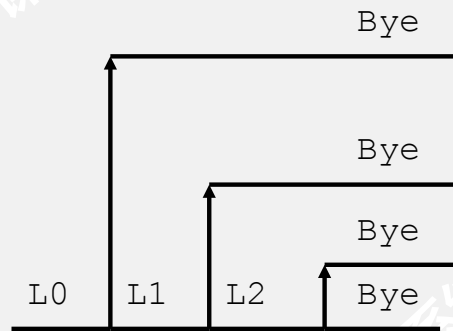
```
void fork3()
{
    printf("%d:L0\n",getpid());
    fork();
    printf("%d:L1\n",getpid());
    fork();
    printf("%d:L2\n",getpid());
    fork();
    printf("%d:Bye\n",getpid());
}
```



# fork示例4

## 父进程嵌套fork

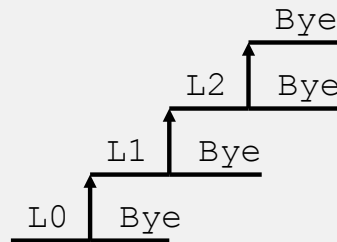
```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```



# fork示例5

## ► 子进程嵌套fork

```
void fork5()  
{  
    printf("L0\n");  
    if (fork() == 0) {  
        printf("L1\n");  
        if (fork() == 0) {  
            printf("L2\n");  
            fork();  
        }  
    }  
    printf("Bye\n");  
}
```



## ► void exit(int status)

### ▷ 退出进程

- 正常退出返回status 0
- 调用一次，不返回

### ▷ atexit() 注册函数记录退出进程时执行的函数

```
void cleanup(void) {  
    printf("cleaning up\n");  
}  
  
void fork6() {  
    atexit(cleanup);  
    fork();  
    exit(0);  
}
```

# 回收子进程——僵死进程(Zombies)

## ▶ 概念

- ▷ 当进程终止后还在消耗系统资源
  - ▶ 操作系统维护各种表的信息
- ▷ 被称为“僵死进程”(Zombie)
  - ▶ 活着的尸体-半死半活

## ▶ 回收

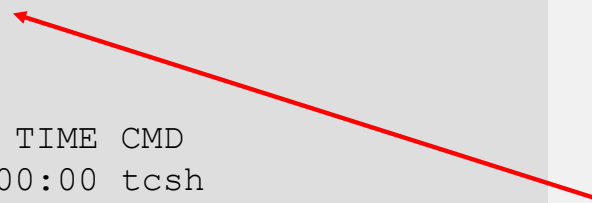
- ▷ 由父进程在子进程结束时执行(using **wait()** or **waitpid()**)
- ▷ 内核将子进程退出状态信息将交给父进程
- ▷ 内核将彻底丢弃子进程

## ▶ 假使父进程不回收会怎么样?

- ▷ 如果父进程终止时没有回收该子进程, 那么子进程会被**init**进程回收(pid = 1)
- ▷ 只有长时间运行的进程需要显式的回收它的僵死子进程
  - ▶ 如 shells 和 servers

# 僵死进程 示例

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 tttyp9      00:00:00 tcsh
 6639 tttyp9      00:00:03 forks
 6640 tttyp9      00:00:00 forks <defunct>
 6641 tttyp9      00:00:00 ps
linux> kill 6639
[1] Terminated
linux> ps
  PID TTY          TIME CMD
 6585 tttyp9      00:00:00 tcsh
 6642 tttyp9      00:00:00 ps
```



```
void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

- ▶ 父进程没有显式回收子进程
- ▶ `ps` 命令显示子进程为“defunct”已停用状态
- ▶ 杀死父进程，则 *init* 进程会回收子进程

# 未终止子进程 示例

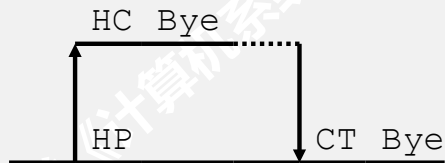
```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9        00:00:00 tcsh
 6676 ttyp9        00:00:06 forks
 6677 ttyp9        00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9        00:00:00 tcsh
 6678 ttyp9        00:00:00 ps
```

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

- ▶ 尽管父进程已经终止，子进程处于活动状态
- ▶ 必须显式地杀死子进程，否则将无限期继续运行

- 父进程可以通过调用wait函数来回收子进程
- `int wait(int *child_status)`
  - 阻塞当前进程，等待直到它的某个子进程终止，回收并收集子进程信息
  - 当`child_status == NULL`，`wait()`返回值是终止的子进程的pid
  - `&child_status`，将子进程退出时的状态取出并存入`child_status`

```
void fork9() {  
    int child_status;  
  
    if (fork() == 0) {  
        printf("HC: hello from child\n");  
    }  
    else {  
        printf("HP: hello from parent\n");  
        wait(&child_status);  
        printf("CT: child has terminated\n");  
    }  
    printf("Bye\n");  
    exit(0);  
}
```





## wait() 示例

- ▶ 如果多个子进程完成，将会被以**任意**顺序执行
- ▶ 可以使用宏指令 **WIFEXITED**、**WEXITSTATUS** 来获得退出状态信息

```
void fork10(){
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

## waitpid():等待特定的进程

► waitpid(pid, &status, options)

▷ 挂起当前进程直到**特定子进程** (pid>1)或**等待集合中的子进程** (pid=-1) 终止

▷ 多个**options** (0,WNOHANG, WUNTRACED,参见教科书P496)

```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) { 顺序
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

## Using wait (fork10)

```
Child 1833 terminated with exit status 103  
Child 1832 terminated with exit status 102  
Child 1831 terminated with exit status 101  
Child 1830 terminated with exit status 100  
Child 1834 terminated with exit status 104
```

## Using waitpid (fork11)

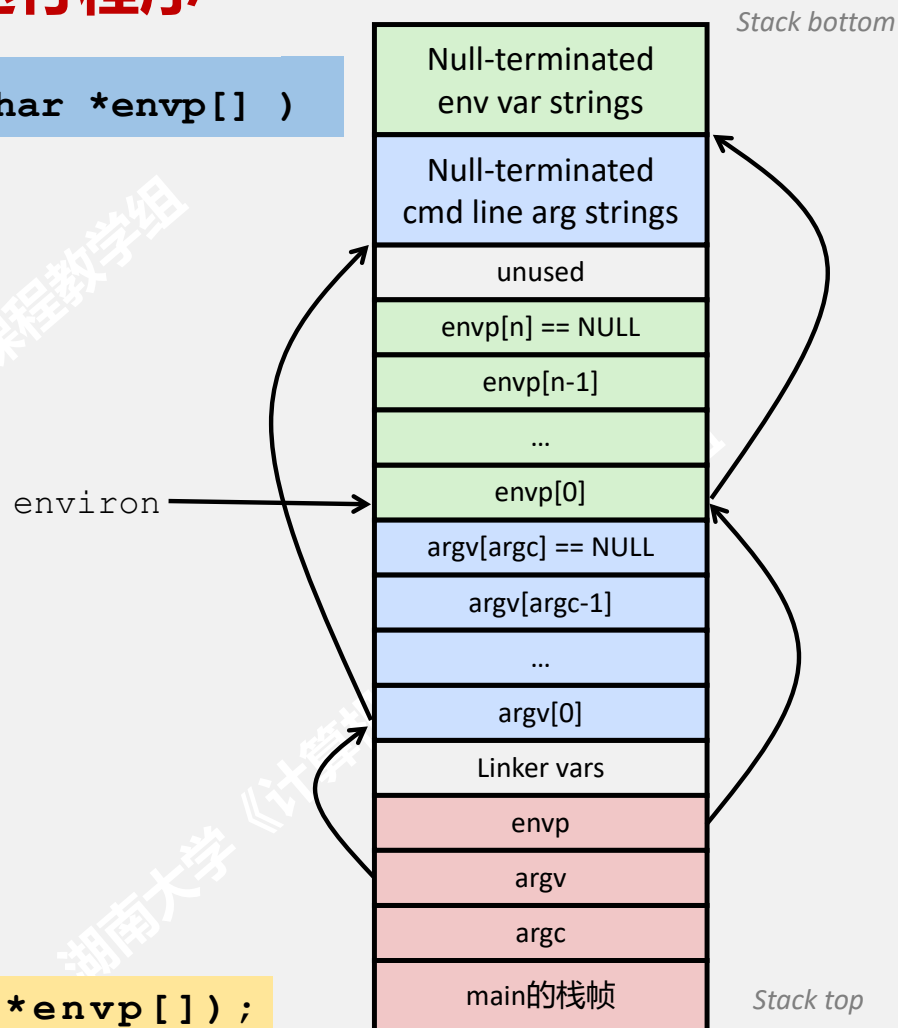
```
Child 1839 terminated with exit status 104  
Child 1838 terminated with exit status 103  
Child 1837 terminated with exit status 102  
Child 1836 terminated with exit status 101  
Child 1835 terminated with exit status 100
```

# execve:加载和运行程序

```
int execve ( char *filename, char *argv[], char *envp[] )
```

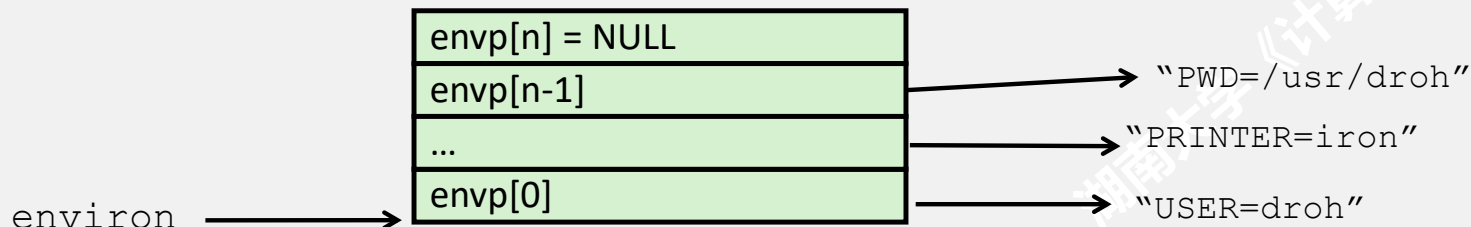
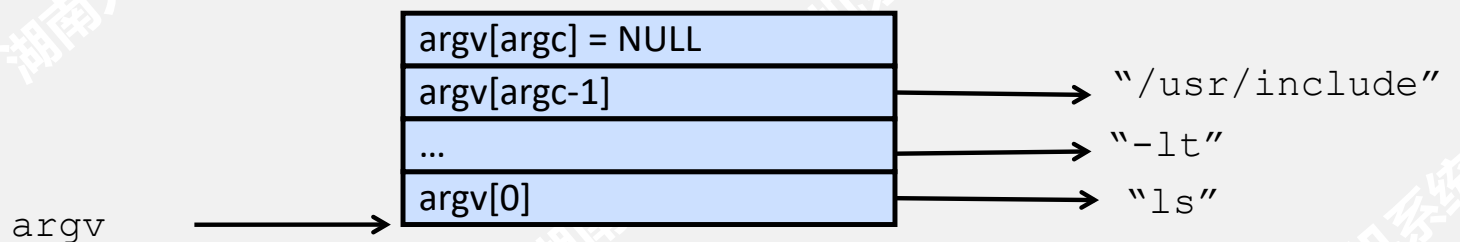
- 在当前进程的上下文中加载并运行：
  - 可执行文件名
  - 参数列表 `argv`
  - 环境变量列表 `envp`
- 不返回 ( 除非出错)
- 覆盖代码、数据和栈
  - 保持pid, 打开的文件 (文件描述符) 和上下文
- 环境变量:
  - “name=value” 字符串
  - `getenv` 与 `setenv`

```
int main(int argc, char *argv[], char *envp[]);
```



## execve 示例

```
if ((pid = Fork()) == 0) { /* Child runs user job */  
    if (execve(argv[0], argv, environ) < 0) {  
        printf("%s: Command not found.\n", argv[0]);  
        exit(0); }  
}
```



# 本讲小结

- 任何时候，系统里面由多个活跃进程
- 一次一个核只能执行一个进程
- 每个进程看起来都独占处理器和私有内存地址空间

- 调用函数 wait 或 waitpid

02 进程

04 回收/等待进程

01 异常

03 创建进程  
终止进程

05 加载和运行程序

- 需要非标准控制流的事件

- 由外部（中断）或内部（陷阱和故障）产生

- 调用函数 fork，一次调用，两个返回值

- 调用函数 exit，一次调用，无返回

- 调用函数 execve

- 一次调用，正常情况无返回