

# Exceptions&Processes

熊婧

# Exceptional Control Flow

- 控制流

从处理器开始运行到停止，程序计数器假设一个值的序列 $a_1, a_2, a_3, \dots$ 。其中 $a_k$ 是某个相应指令 $I_k$ 的地址。每次从 $a_k$ 到 $a_{k+1}$ 的过渡称为控制转移，这样的控制转移序列叫做处理器的控制流。ps: 跳转、函数调用和返回等程序指令。

- 异常控制流ECF

系统状态的变化，不能被内部程序变量捕获，也不一定和程序的执行相关。例如，程序向磁盘请求数据然后休眠，直到被通知数据已经就绪；当子进程终止时，创造这些子进程的父进程必须得到通知。现代系统通过使控制流发生突变来对这些情况作出反应。一般我们把这些突变称为异常控制流。

# Exceptional Control Flow

异常控制流发生在计算机系统的各个层次——

- 硬件层，硬件检测到的事件会触发控制突然转移到异常处理程序。
- 操作系统层，内核通过上下文切换将控制从一个用户进程转移到另一个用户进程。
- 应用层，一个进程可以发送信号到另一个进程，而接收者会将控制突然转移到它的一个信号处理程序。
- 一个程序还可以通过回避通常的栈规则，并执行到其他函数中任意位置的非本地跳转来对错误作出反应。

# Exception

- 异常

控制流中的突变，用来响应处理器状态中的某些变化。

注意：异常≠出错. 异常也可能是为了实现某些功能(中断/陷阱)

- 状态

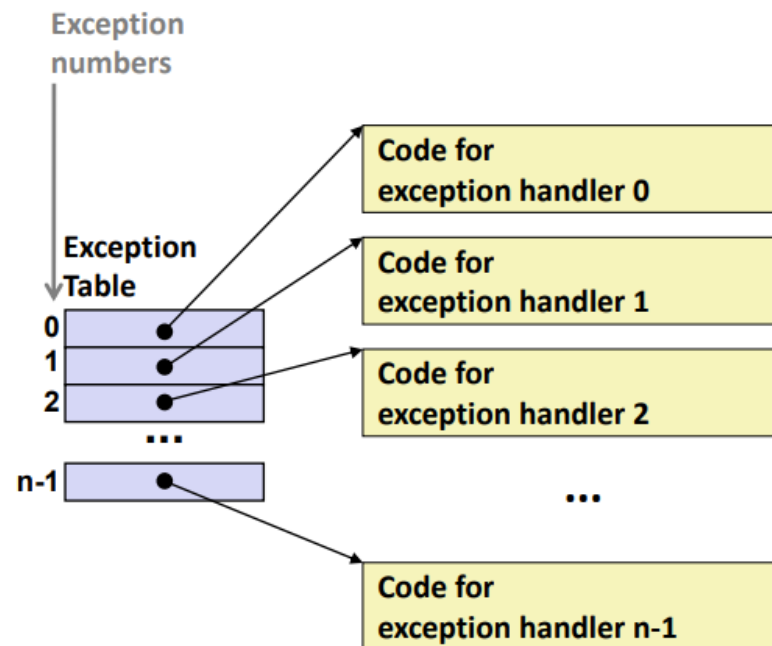
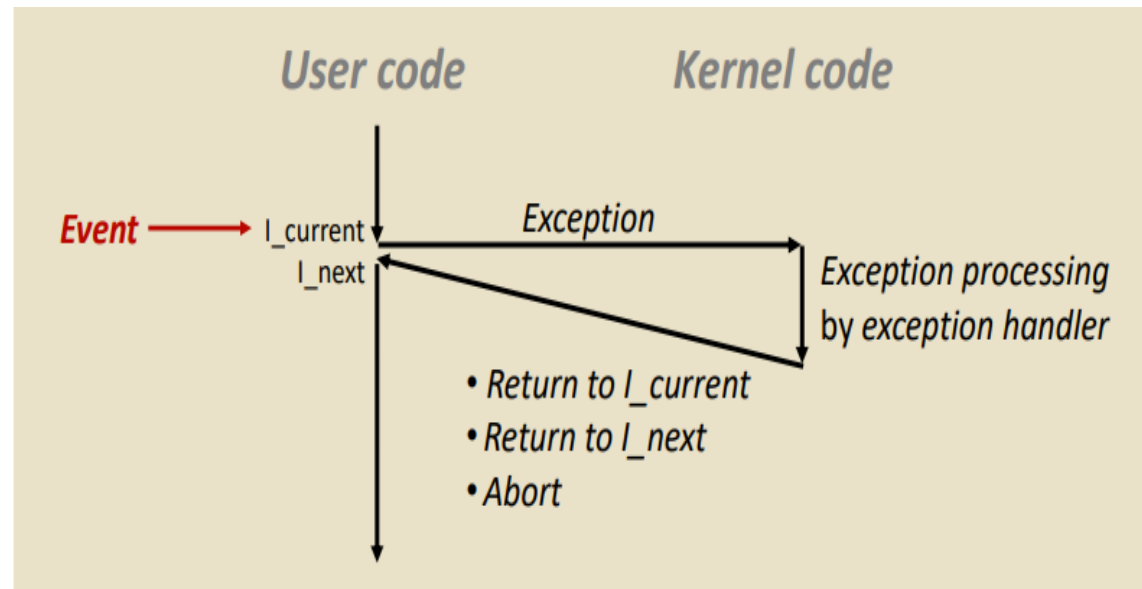
在处理器中，状态被编码为不同的位和信号。

- 事件

处理器状态的变化称为事件。

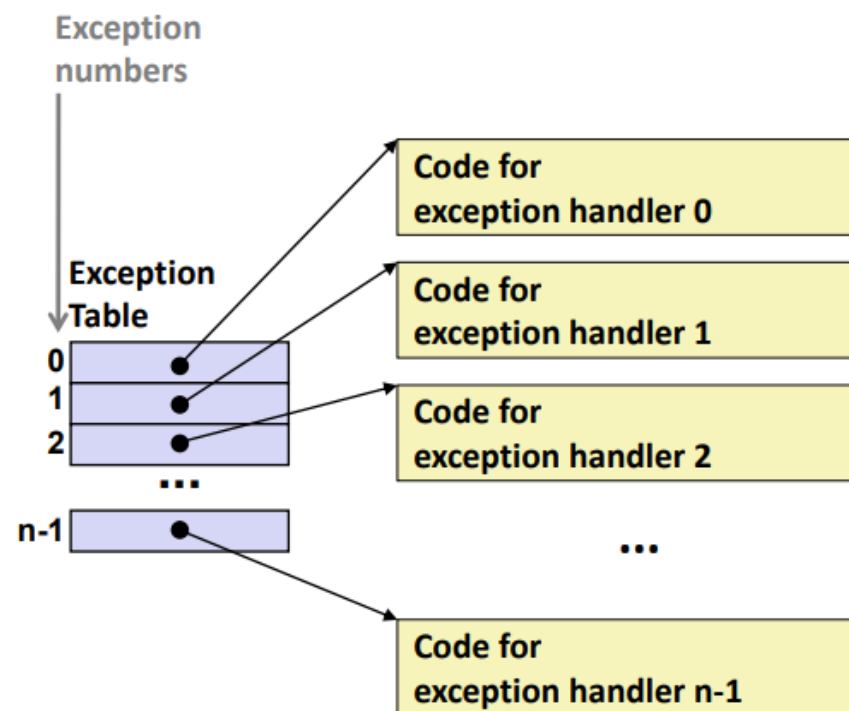
事件可能和当前指令的执行直接相关，例如发生虚拟内存缺页、算术溢出，或者一条指令试图除以0。

也可以和当前指令的执行没有关系，例如一个系统定时器产生信号或者一个I/O请求完成。



# 异常处理

系统中可能的每种类型的异常都分配了一个唯一的非负整数的异常号。一些异常号由处理器的设计者分配，包括被零除、缺页、内存访问违例、断电以及算数运算溢出。其他异常号是由操作系统内核的设计者分配的，包括系统调用和来自外部I/O设备的信号。



# 异常处理

- 运行时，处理器检测到事件发生，确定了对应的异常号后，处理器触发异常，通过异常表执行间接过程调用，获取异常处理程序的地址，转到该程序。
- 异常表的起始地址存在一个叫异常表基址寄存器的特殊CPU寄存器里。

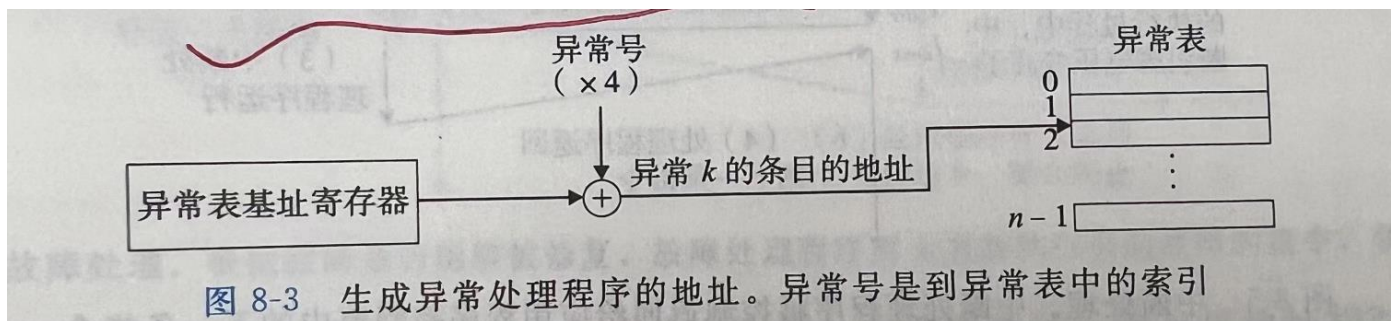


图 8-3 生成异常处理程序的地址。异常号是到异常表中的索引

异常类似过程调用，但有一些重要的不同之处：

- 异常根据其类型，返回地址是当前指令或者下一条指令。
- 处理器还将一些额外的处理器状态也压入栈内，因为返回时重新开始执行被中断的程序需要这些状态。eg:EFLAGS寄存器
- 如果控制从用户程序转移到内核，这些项目都会被压入内核栈中。
- 异常处理程序运行在内核模式，对所有系统资源都有访问权限。

# 异常 (Exception)

- 异常的分类：中断 / 陷阱 / 故障 / 终止
  - 中断：外部I/O设备
  - 陷阱(同步)：故意引发的异常，目的是进行系统调用
  - 故障(同步)：出错了，有可能修复. 例：缺页，除法错误，一般保护故障(段错误)
  - 终止(同步)：致命错误，无法恢复. 例：DRAM/SRAM损坏
- 注意区分：同步/异步？返回行为？

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

**Figure 8.4** **Classes of exceptions.** Asynchronous exceptions occur as a result of events in I/O devices that are external to the processor. Synchronous exceptions occur as a direct result of executing an instruction.

# 中断

- 中断是异步发生的，是来自处理器外部的I/O设备的信号的结果。硬件中断不是任何一条专门的指令造成的，从这个意义上来说是异步的。硬件中断的异常处理程序常常称为中断处理程序。
- **注意：同步发生的异常类型是执行当前指令的结果——故障指令（陷阱、故障和终止）。**
- 系统总线上的异常号标识了引起中断的设备。

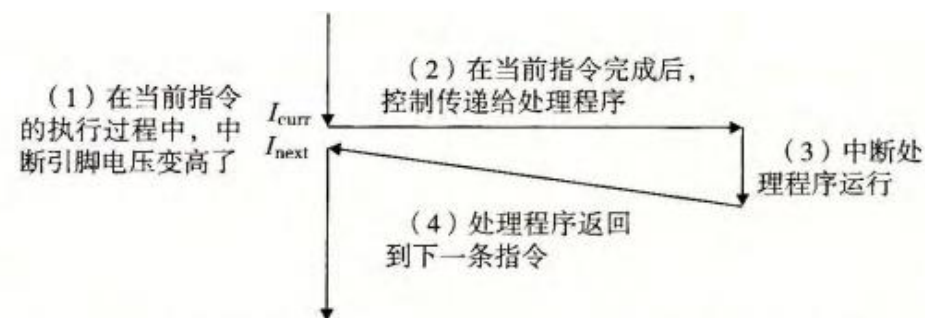


图 8-5 中断处理。中断处理程序将控制返回给应用程序控制流中的下一条指令



# 陷阱和系统调用

- 陷阱是有意的异常，是执行一条指令的结果。就像中断处理程序一样，陷阱处理程序将控制返回到下一条指令。在用户程序和系统内核之间提供了一个像过程一样的接口，叫系统调用。
- 系统调用与函数调用的区别——运行在内核模式中，允许系统调用执行特权指令，并访问定义在内核中的栈。

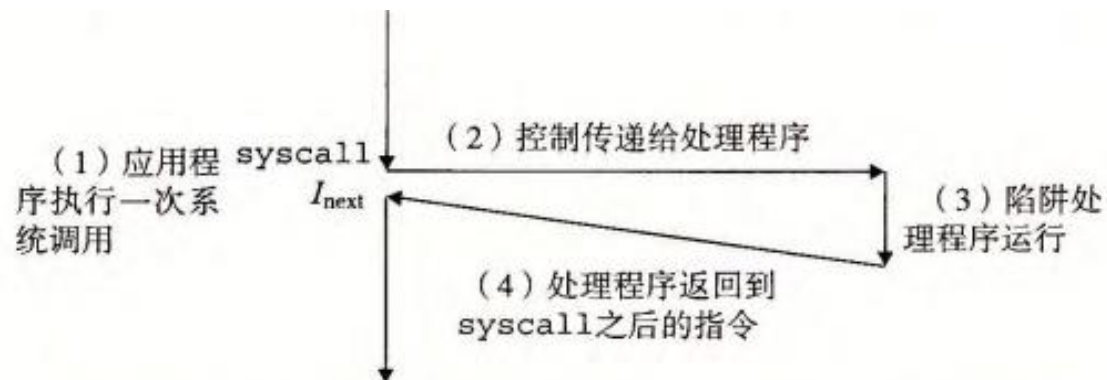


图 8-6 陷阱处理。陷阱处理程序将控制返回给应用程序控制流中的下一条指令

# 故障

- 故障由错误情况引起，可能能够被故障处理程序修正。
- 经典的故障实例是缺页故障，所引用的虚拟地址对应的物理存储不在内存中，就必须从磁盘中取出时，就会发生故障。

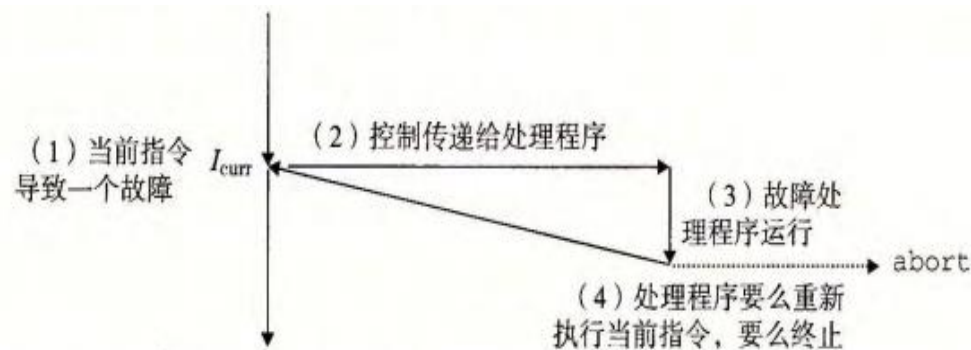
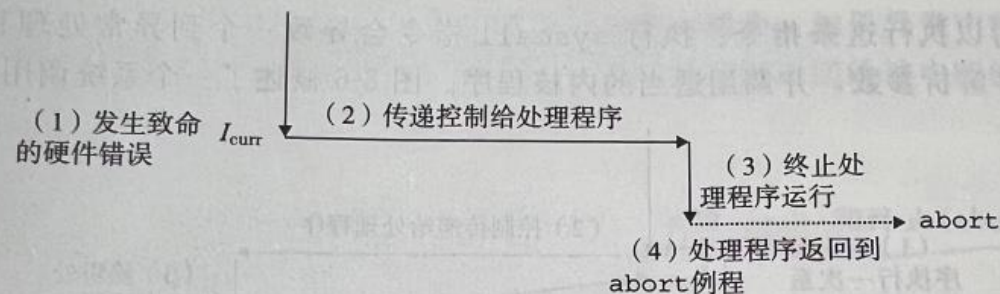


图 8-7 故障处理。根据故障是否能够被修复，故障处理程序要么重新执行引起故障的指令，要么终止

# 终止

终止是不可恢复的致命错误造成的结果，通常是一些硬件错误，比如DRAM或者SRAM为被损坏时发生的奇偶错误。



# 例


7. 关于 x86-64 系统中的异常，下面那个判断是正确的：
- A. 除法错误是异步异常，Unix 会终止程序；
  - B. 键盘输入中断是异步异常，异常服务后会返回当前指令执行；
  - C. 缺页是同步异常，异常服务后会返回当前指令执行；
  - D. 时间片到时中断是同步异常，异常服务后会返回下一条指令执行；

7. C。除法错误是不可恢复故障，是同步的；I/O 中断是异步的，一般会执行完当前指令，再去处理，返回就不需要再处理了；缺页异常当然需要重新执行遇到问题的访存指令；时间片到中断属于时钟中断，属于异步异常。

# 异步vs同步

- 异步异常不是由当前指令的执行引起的，也不是由某条特定的指令引起的；而同步指令是由当前特定指令引起的，例如读内存。
- 异步异常需要等当前指令执行完成后将控制转移给异常处理程序，而同步异常在执行当前指令的过程中将控制转移给异常处理程序。
- 异步异常总是返回到下一条指令，而同步异常存在以下情况：陷阱返回到下一条指令；故障可能返回到当前指令或者调用**abort**例程；终止调用**abort**例程不会返回。

# Linux/x86-64系统中的异常



异常号	描述	异常类别
0	除法错误	故障
13	一般保护故障	故障
14	缺页	故障
18	机器检查	终止
32~255	操作系统定义的异常	中断或陷阱

图 8-9 x86-64 系统中的异常示例

# Linux/x86-64故障和终止

- 除法错误：除以零或者结果对目标操作数来说太大。shell通常将其报告为“浮点异常”。
- 一般保护故障：通常是因为引用了一个未定义的虚拟内存区域或尝试写入一个只读的文本段。系统不会恢复这种故障，shell报“段错误”。
- 缺页：重新执行产生故障的指令的一个异常示例。
- 机器检查：致命的硬件错误。机器检查处理程序从不返回控制给应用程序。



# 系统调用

- 每个系统调用有一个编号，对应内核中的一张跳转表中的条目（不是异常表）
- 触发指令： `syscall` -> 陷入内核
  - `%rax`存放系统调用号/返回值，`%rdi` `%rsi` `%rdx` `%r10` `%r8` `%r9`存放参数

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

**Figure 8.10** Examples of popular system calls in Linux x86-64 systems.

# 进程

- 定义：执行中的程序实例
  - 作用：隔离不同程序，让每个程序都认为系统中只有自己在运行
  - 运行一个新的exe = 创建一个新的进程
- 进程间相互独立：
  - 独立的逻辑控制流
  - 私有的地址空间
- 每个进程都有一个上下文
  - 上下文：程序运行所需的相关状态
    - 栈 / 寄存器 / 程序计数器 / 环境变量 / 打开的文件描述符



# 逻辑控制流

- 逻辑控制流（逻辑流）

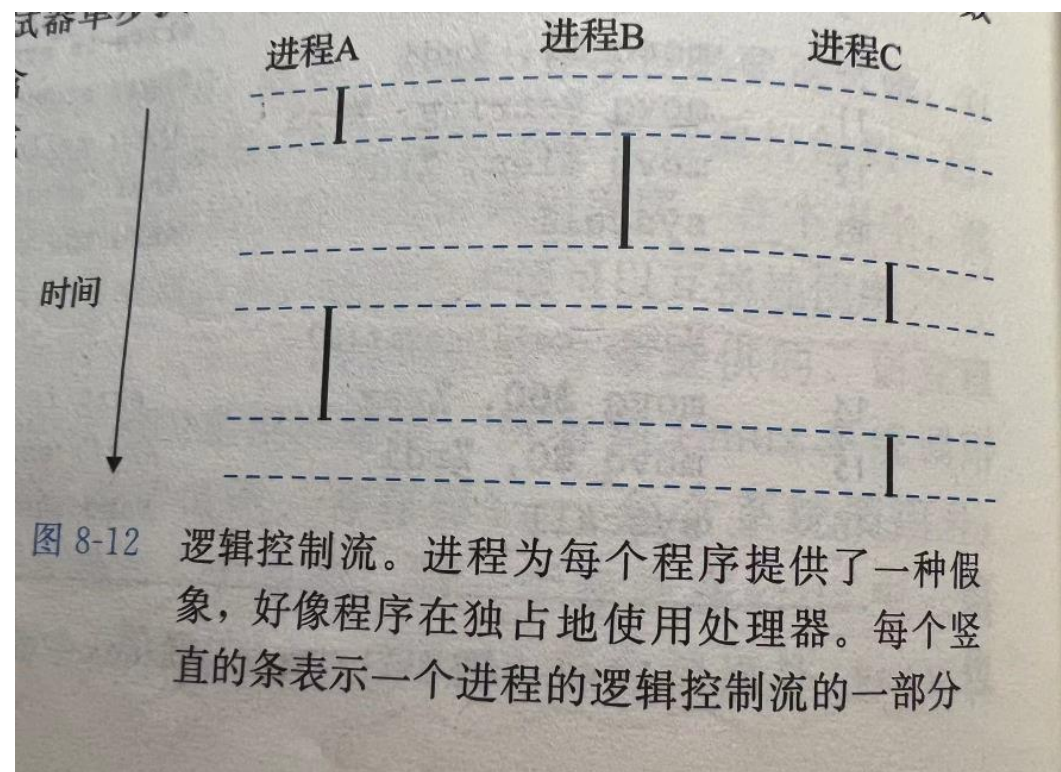
程序计数器PC的值的序列

- 进程轮流使用处理器。每个进程执行它的流的一部分，然后被抢占（暂时挂起），然后轮到其他进程。

- 并发vs并行

并发：多个进程轮流在同一个核上运行。

并行：同一时刻，多个进程在不同核上运行。



# Process进程

- 上下文切换

内核为每个进程维持一个上下文。

在进程执行的某些时刻，内核可以决定抢占当前进程，并重新开始一个之前的进程或开始一个新的进程。这种决策就叫调度，使用的机制叫做上下文切换。

Q: 抢占是一个同步的过程还是异步的过程？

- 流程：
  - 1) 保存当前进程A的上下文(寄存器/PC等)；
  - 2) 恢复已保存的之前被抢占的进程B的状态；
  - 3) 将控制传递给进程B。

# 私有地址空间

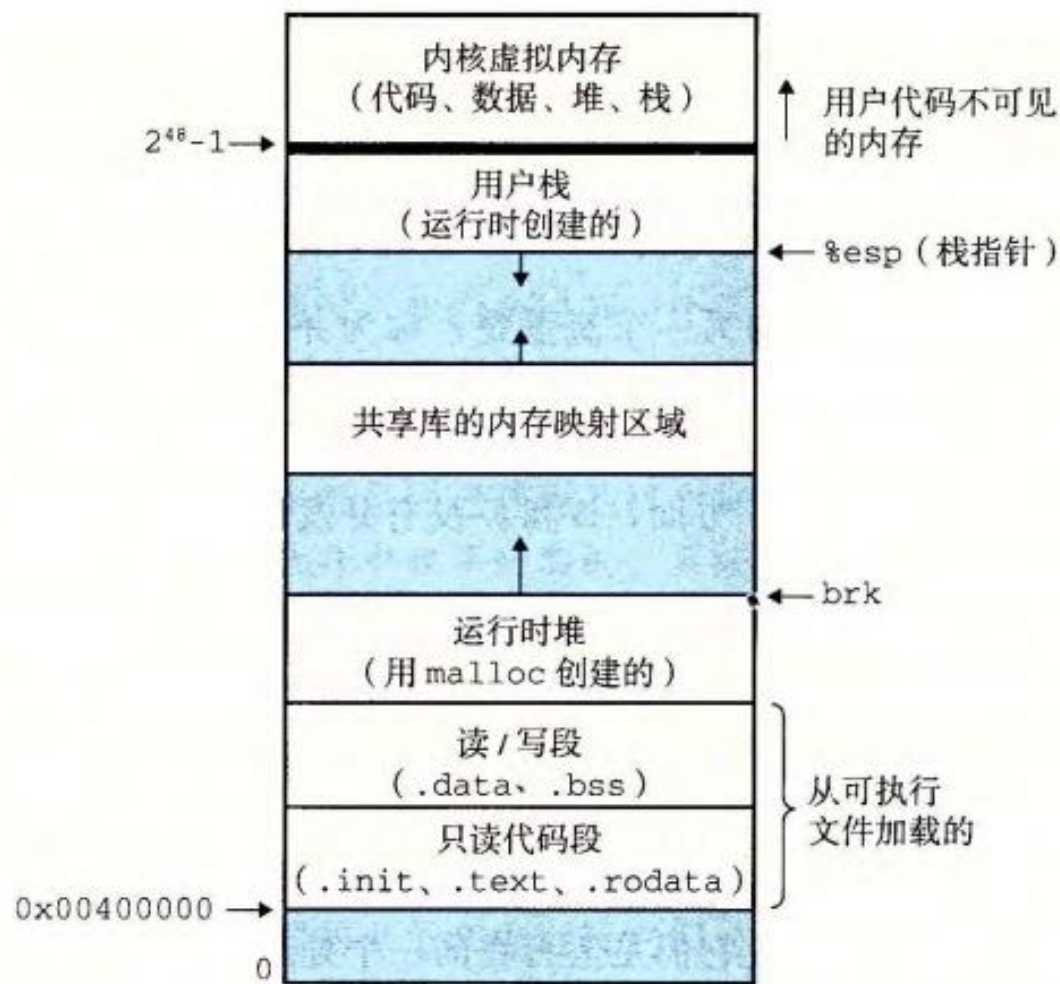


图 8-13 进程地址空间

- 进程的私有地址空间中和某个地址相关联的字节不能被其他进程读写。
- 私有地址空间都有相同的通用结构  
用户:  $0x400000 \sim 2^{48} - 1$   
内核:  $\geq 2^{48}$
- 用户栈从高向低增长, 堆从低向高增长

# Process进程

- 用户模式&内核模式

处理器通常用某个控制寄存器中的一个模式位来描述一个进程的权限。

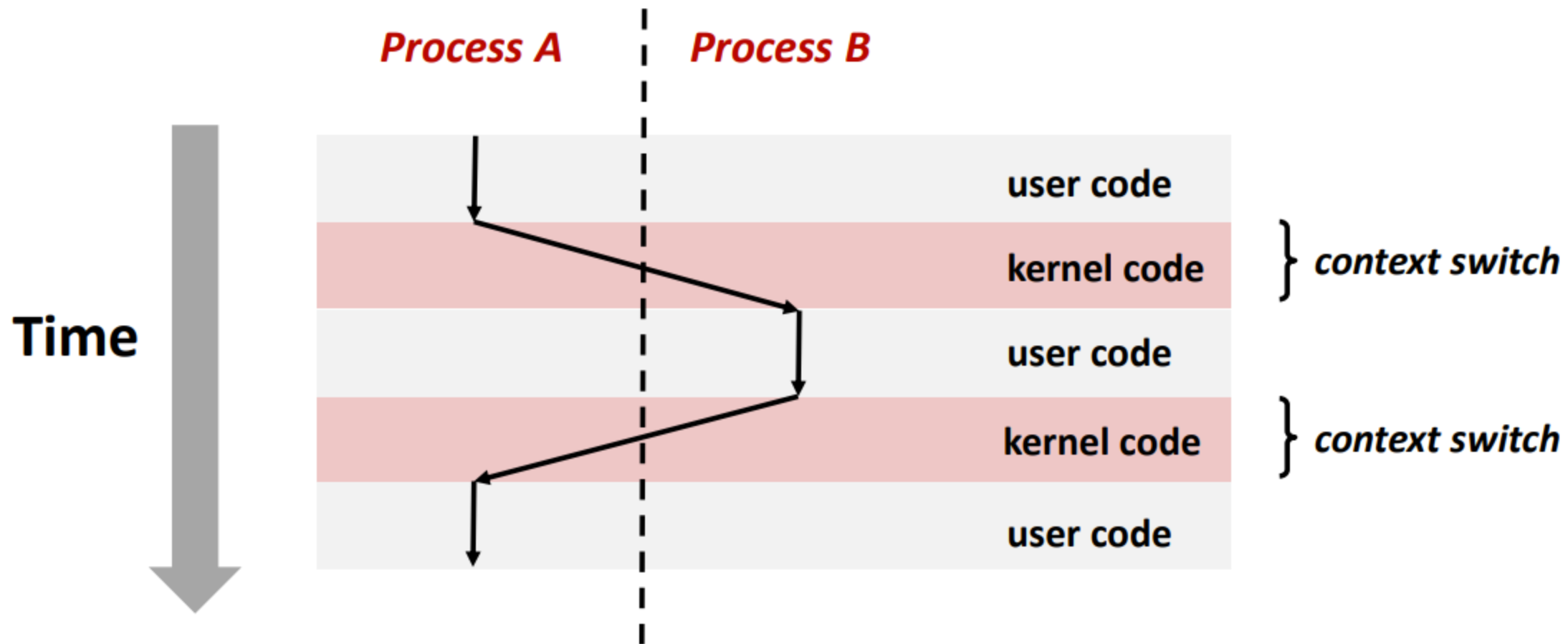
当设置了模式位时，进程运行在内核模式，可以执行任何指令，访问系统中任何位置。

没设置模式位时，进程运行在用户模式。不允许执行特权指令，如停止处理器、改变模式位等，也不允许访问内核代码和数据。

异常发生时，控制传递到异常处理程序，此时处理器将模式从用户模式切换到内核模式，返回时切回去。

注意：/proc文件系统

# Process进程



# 系统调用错误处理

- 错误报告函数

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

简化

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

错误报告函数

```
if ((pid = fork()) < 0)  
    unix_error("fork error");
```

# 系统调用错误处理

- 错误处理包装函数

```
pid_t Fork(void)
{
    pid_t pid;

    if ((pid = fork()) < 0)
        unix_error("Fork error");
    return pid;
}
```

```
pid = Fork();
```

# 进程控制

- 同一时刻，操作系统中有若干个进程
- 进程的三种状态
  - 运行
    - 同一时间可以有若干个进程同时运行
    - 运行  $\neq$  正在CPU上执行
      - 也可能是在等待调度的队列中
  - 停止：进程被挂起，且不会进入等待调度的队列
    - 收到以下4种信号会导致进程停止：SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU
    - 收到SIGCONT后被转为运行状态
  - 终止：进程永不运行，不能被转为运行状态
    - 可能原因：①收到相关信号 ②从主程序返回 ③调用exit函数
    - 一个进程终止后必须被回收

```
#include <stdlib.h>

void exit(int status);
```

该函数不返回。



# 进程控制

- 获取进程ID

每个进程都有一个唯一的正数进程ID。

pid\_t在Linux系统上在types.h里被定义为int。

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

返回：调用者或其父进程的PID。

# 进程控制

- 创建进程fork()

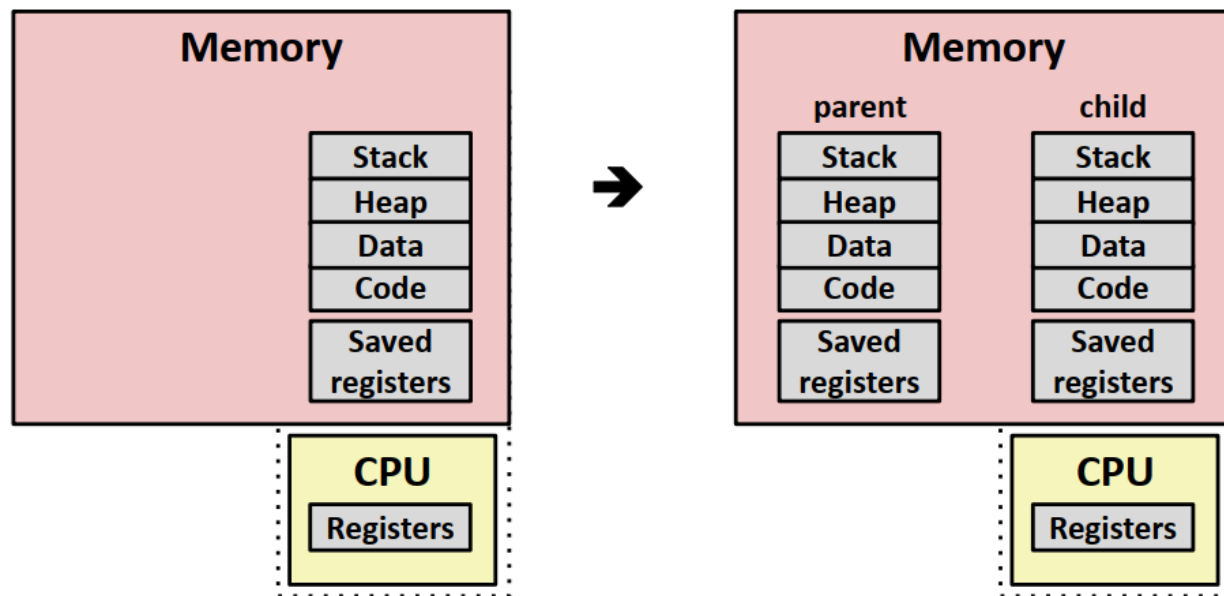
父进程调用fork函数创建一个新的运行的子进程。

- 子进程得到与父进程用户级虚拟地址空间相同且**独立**的一份副本（代码、数据段、堆、共享库、用户栈），并且**共享文件**，但是PID不相同。在父进程中，fork()返回子进程PID；在子进程中返回0。

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

返回：子进程返回0，父进程返回子进程的PID，如果出错，则为-1。



# 进程控制

- fork() 实例

```
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

*fork.c*

函数特点:

- 调用一次，返回两次
- 并发执行

内核能够以任意方式交替执行它们的逻辑控制流中的指令

- 相同但是独立的地址空间

```
linux> ./fork
parent: x=0
child : x=2
```

```
linux> ./fork
child : x=2
parent: x=0
```

```
linux> ./fork
parent: x=0
child : x=2
```

```
linux> ./fork
parent: x=0
child : x=2
```

# 进程控制

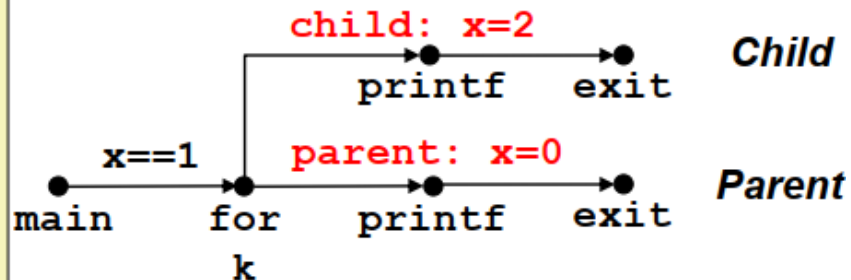
- 进程图：刻画程序语句偏序的一种简单前趋图。
- 顶点：一条程序语句的执行
- 有向边：语句的顺序发生（ $a \rightarrow b$ ：a发生在b之前）
- 顶点拓扑排序：用于表示程序中语句可行的全序排列。

```
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

*fork.c*



# 例

下面代码一共输出几个A?

```
fork() || fork();  
printf( "A\n" );  
exit(0);
```

3个.

如果第一个fork()确定返回值为1, 则第二个fork()不会被运行

# 进程控制

- 一个进程终止后必须被其父进程回收
  - 如果父进程已终止，则安排init进程作为养父
    - init进程：pid=1，系统启动由内核创建的第一个进程
      - 所有的进程都是它衍生出来的
- waitpid()：父进程调用waitpid()等待其子进程终止

# 进程控制

- 僵死进程：终止但未被回收的进程。
- 父进程结束，子进程未结束时，init 进程回收该子进程。像shell等长期运行的程序总应该回收子进程——僵死子进程不运行，但消耗内存资源。

## Zombie Example

```
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 */  
    }  
}
```

```
linux> ./forks 7 &  
[1] 6639  
Running Parent, PID = 6639  
Terminating Child, PID = 6640
```

```
linux> ps  
  PID TTY          TIME CMD  
 6585 ttyp9      00:00:00 tcsh  
 6639 ttyp9      00:00:03 forks  
 6640 ttyp9      00:00:00 forks <defunct>  
 6641 ttyp9      00:00:00 ps  
linux> kill 6639  
[1] Terminated  
linux> ps  
  PID TTY          TIME CMD  
 6585 ttyp9      00:00:00 tcsh  
 6642 ttyp9      00:00:00 ps
```

■ **ps** shows child process as "defunct" (i.e., a zombie)

■ Killing parent allows child to be reaped by **init**

## Non-terminating Child Example

```
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);  
    }  
}
```

```
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
```

■ Child process still active even though parent has terminated

■ Must kill child explicitly, or else will keep running indefinitely

# waitpid()

```
#include <sys/types.h>
```

```
#include <sys/wait.h>
```

```
pid_t waitpid(pid_t pid, int *statusp, int options);
```

返回：如果成功，则为子进程的 PID，如果 WNOHANG，则为 0，如果其他错误，则为 -1。

- 等待集合的成员由参数 `pid` 决定：

`pid > 0`：等待单独子进程，其进程 ID 等于 `pid`

`pid = -1`：等待集合由父进程的所有子进程构成



# waitpid()

- 修改默认行为

(1) 默认情况: **options=0**时, **waitpid**挂起调用进程的执行, 直到它的等待集合中的一个子进程终止。如果等待集合中一个进程在刚调用时就终止, 那么**waitpid**立即返回。成功回收子进程则返回该子进程**PID**。

(2) 将**options**设置为常量及其各种组合。

**WNOHANG**: 等待集合中任何子进程都没有终止, 则立即返回。用途: 在等待子进程终止时, 需要做别的工作。

**WUNTRACED**: 挂起调用进程的执行, 直到等待集合中的一个进程变成已终止或已被停止, 返回的**PID**为导致返回的已终止或被停止子进程的**PID**。用途: 检查已终止或被停止的子进程。

**WCONTINUED**: 挂起调用进程的执行, 直到等待集合中一个正在运行的进程终止或等待集合中一个被停止的进程收到**SIGCONT**信号重新开始执行。

选项还可以进行组合, 例如: **WNOHANG|WUNTRACED**: 立即返回, 如果等待集合中子进程都没有停止或终止, 则返回**0**; 如果有一个停止或终止, 则返回值为该子进程**PID**。

# waitpid()

- 修改默认行为

如果 `statusp` 参数是非空的，那么 `waitpid` 就会在 `status` 中放上关于导致返回的子进程的状态信息，`status` 是 `statusp` 指向的值。`wait.h` 头文件定义了解释 `status` 参数的几个宏：

- `WIFEXITED(status)`：如果子进程通过调用 `exit` 或者一个返回(`return`)正常终止，就返回真。
- `WEXITSTATUS(status)`：返回一个正常终止的子进程的退出状态。只有在 `WIFEXITED()` 返回为真时，才会定义这个状态。
- `WIFSIGNALED(status)`：如果子进程是因为一个未被捕获的信号终止的，那么就返回真。
- `WTERMSIG(status)`：返回导致子进程终止的信号的编号。只有在 `WIFSIGNALED()` 返回为真时，才定义这个状态。
- `WIFSTOPPED(status)`：如果引起返回的子进程当前是停止的，那么就返回真。
- `WSTOPSIG(status)`：返回引起子进程停止的信号的编号。只有在 `WIFSTOPPED()` 返回为真时，才定义这个状态。
- `WIFCONTINUED(status)`：如果子进程收到 `SIGCONT` 信号重新启动，则返回真。

# waitpid()

- 错误条件

调用进程没有子进程: `waitpid`返回-1, 设置`errno`为ECHILD

`waitpid`函数被信号中断: `waitpid`返回-1, 设置`errno`为EINTR

- `wait()`函数

`wait(&status)`等价于调用`waitpid(-1,&status,0)`。

练习题 8.3 列出下面程序所有可能的输出序列：

code/ecf/waitprob0.c

```
1  int main()
2  {
3      if (Fork() == 0) {
4          printf("a"); fflush(stdout);
5      }
6      else {
7          printf("b"); fflush(stdout);
8          waitpid(-1, NULL, 0);
9      }
10     printf("c"); fflush(stdout);
11     exit(0);
12 }
```

code/ecf/waitprob0.c

8.3 我们知道序列 acbc、abcc 和 bacc 是可能的，因为它们对应有进程图的拓扑排序(图 8-48)。而像 bcac 和 cbca 这样的序列不对应有任何拓扑排序，因此它们是不可行的。

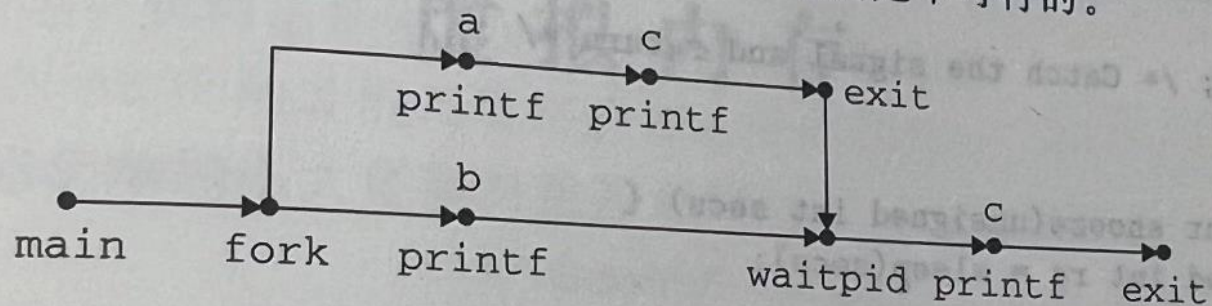


图 8-48 练习题 8.3 的进程图

# 让进程休眠

- `sleep`函数：将进程挂起一段时间，完成则返回0，否则返回剩余秒数。

```
#include <unistd.h>

unsigned int sleep(unsigned int secs);
```

返回：还要休眠的秒数。

- `pause`函数：将调用函数休眠，直到收到信号，返回-1。

```
#include <unistd.h>

int pause(void);
```

总是返回-1。



# execve()

- `int execve(const char *filename, const char *argv[], const char *envp[])`
- 加载并运行可执行目标文件 `filename`
- 参数列表 `argv`，按照惯例，`argv[0]=filename`
- 环境变量列表 `envp`

`getenv(const char*)`: 搜索字符串 `name=value`;

`setenv(const char* name, const char* newvalue, int overwrite)`: 用 `newvalue` 代替 `oldvalue` (第3个参数非0);

`unsetenv(const char*)`: 如果包含 `name=oldvalue` 则删除之。

- 覆盖代码、数据和栈，保存 `pid`
- 出错错误时返回。否则调用一次。从不返回。

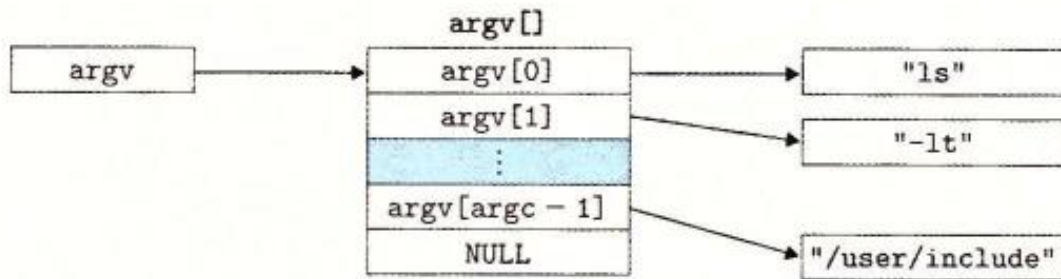


图 8-20 参数列表的组织结构

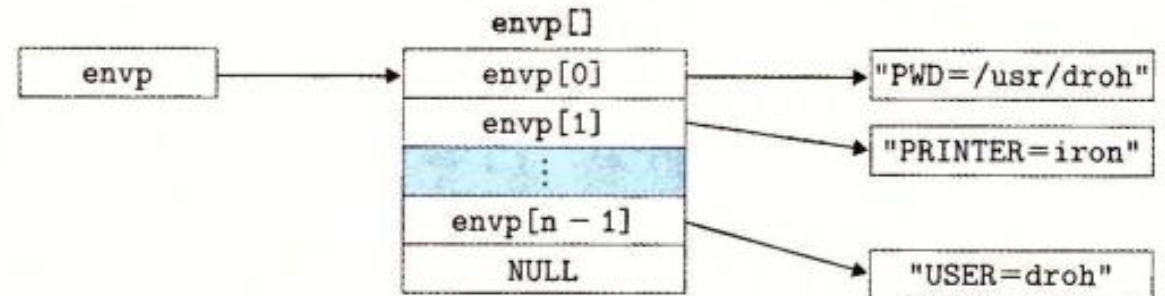


图 8-21 环境变量列表的组织结构

# execve()

- 新程序主函数执行时，用户栈的组织结构

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

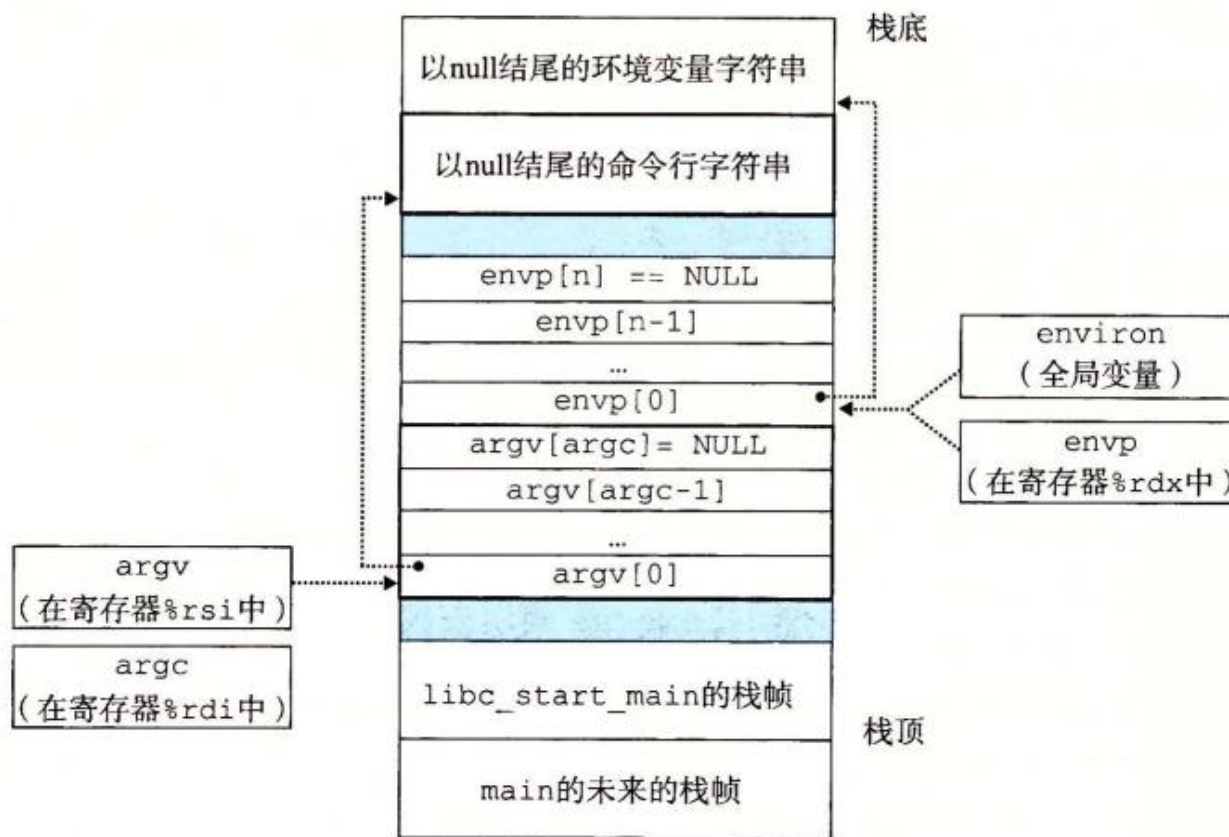


图 8-22 一个新程序开始时，用户栈的典型组织结构

# Process vs Program

- 程序是一堆代码和数据；可以作为目标文件存在于磁盘上，或者作为段存在于地址空间中。
- 进程是执行中程序的一个具体的实例；程序总是运行在某个进程的上下文中。
- `fork()`函数在新的子进程中运行相同的程序，新的子进程是父进程的复制品；`execve()`函数在当前进程的上下文中加载并运行一个新的程序，它会覆盖当前进程的地址空间，但没有创建一个新进程。新的程序有相同的PID，并且继承已打开的所有文件表述符。
- **shell**：交互型应用级程序。从sh程序到csh、tcsh、ksh和bash等变种  
对命令行进行解析和处理——  
第一个参数：内置shell命令or可执行目标文件（创建子进程）  
最后一个参数“&”字符——后台执行否则waitpid等待终止



Thank you for listening!