CS:APP 讨论（十）

异常控制流（上）

马玉坤

2017 年 3 月 5 日

异常控制流

处理器执行的指令间的过渡所形成的控制转移序列称为处理器的**控制流** (Control Flow)。
系统必须能够对系统状态的变化做出反应，例如硬件定时器定期产生的信号。现代系统通过使控制流发生突变来应对这些情况，这些突变称为**异常控制流** (Exceptional Control Flow, ECF)。

各种层次的异常控制流

异常控制流发生在计算机系统的各个层次，比如：

- 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。**异常**
- 操作系统层，内核通过上下文转换将控制从一个用户进程转移到另一个用户进程。**上下文转换**
- 应用层，一个进程可以发送信号到另一个进程，信号接收者会将控制突然转移到它的一个信号处理程序。**信号（本章重点）**

异常 (exception) 是异常控制流的一种形式，它一部分由硬件实现，一部分由操作系统实现。
当处理器状态发生一个重要的变化时（例如除 0），处理器正在执行当前指令 I_{curr}，处理器会通过**异常表** (exception table)，进行间接过程调用（异常），让异常处理程序处理异常，之后可能

- 返回指令 I_{curr}
- 返回指令 I_{next}
- 终止被中断的程序

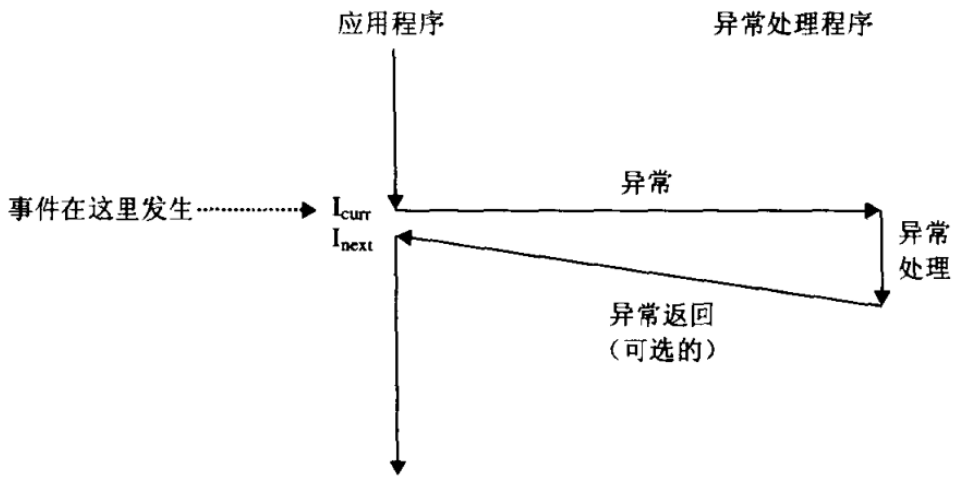


图 8.1 异常的剖析

异常处理

系统中的每种类型异常都分配了一个唯一的非负整数，作为异常号。系统启动时会初始化一张异常表，异常表的起始地址存在一个特殊的 CPU 寄存器里，异常表基址寄存器。

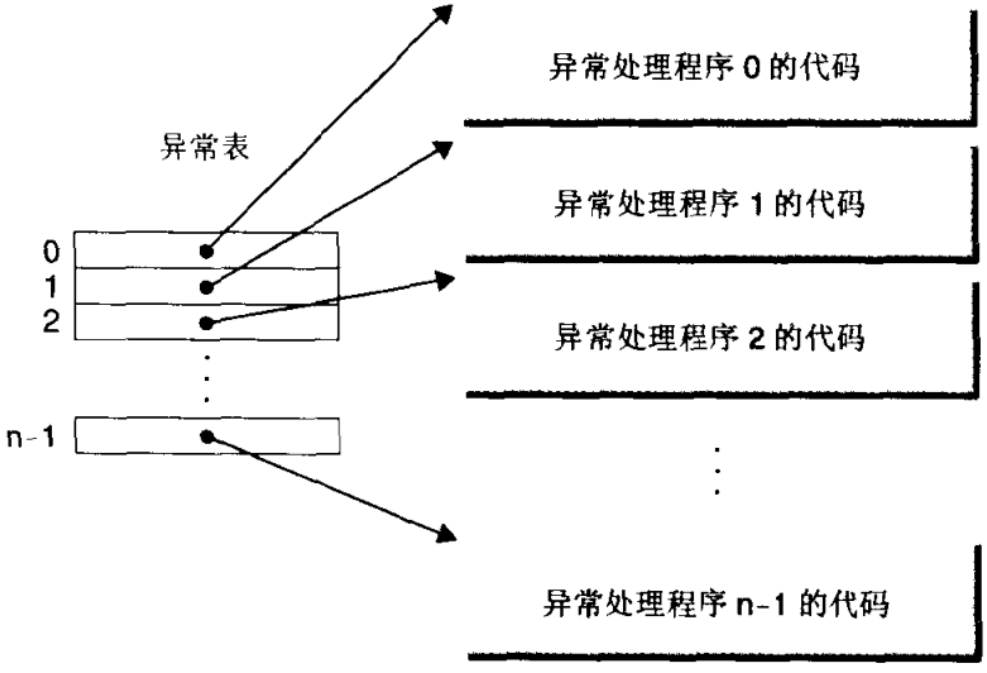


图 8.2 异常表

自定义异常处理程序

将向量表指向的指令地址修改为新程序的地址。（对于 Ring 3 级别的用户程序，权限不够。而且该方法不安全，应尽量使用系统调用。）

四种异常

异常可分为四类：中断 (interrupt)、陷阱 (trap)、故障 (fault) 和终止 (abort)。

类别	原因	异步/同步	返回行为
中断	来自 I/O 设备的信号	异步	总是返回到下一条指令
陷阱	有意的异常	同步	总是返回到下一条指令
故障	潜在可恢复的错误	同步	可能返回到当前指令
终止	不可恢复的错误	同步	不会返回

图 8.4 异常的类别

陷阱与系统调用

用户程序需要向内核请求服务时（例如读文件，创建进程），可以使用 syscall(Linux 系统)，此时异常处理程序会解析参数，调用适当的内核程序，并将结果返回给用户程序。

Linux/x86-64 系统中的异常

x86-64 系统有 256 种不同的异常类型。

- 0-31 号：Intel 架构师定义
- 32-255 号：操作系统定义
- 128 号：Linux 系统调用

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
32～127	操作系统定义的异常	中断或陷阱
128（0x80）	系统调用	陷阱
129～255	操作系统定义的异常	中断或陷阱

Linux/x86-64 系统调用

Linux 提供几百种系统调用，当应用程序想要请求内核服务时可以使用，包括读文件、写文件或是创建一个新进程。但是一般情况下使用 C 语言时不需要使用 syscall，因为 C 语言已经将大部分的 syscall 的功能封装为函数，例如 read()、write()、fork()。

Linux x64 与 Linux x86 系统调用

x86 下，汇编语言使用 sysenter 语句。x64 下，使用 syscall 语句。使用 int $0x80 语句也可以。

逻辑控制流

进程为每个程序提供了一个假象，一个独立的逻辑控制流，好像我们的程序独占地使用处理器。

- 逻辑（控制）流
- 并发流：一个逻辑流的执行在时间上与另一个流重叠，称为并发流
- 并行流：两个流并发的运行在不同的处理器核或者计算机上，那么我们称它们为并行流

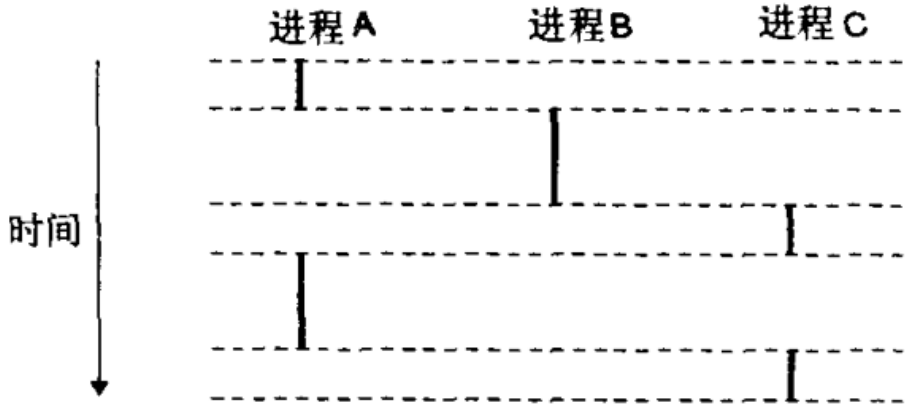


图 8.10 逻辑控制流

私有地址空间

进程为每个程序提供了一个假象，一个私有的地址空间，好像程序独自地使用系统地址空间。

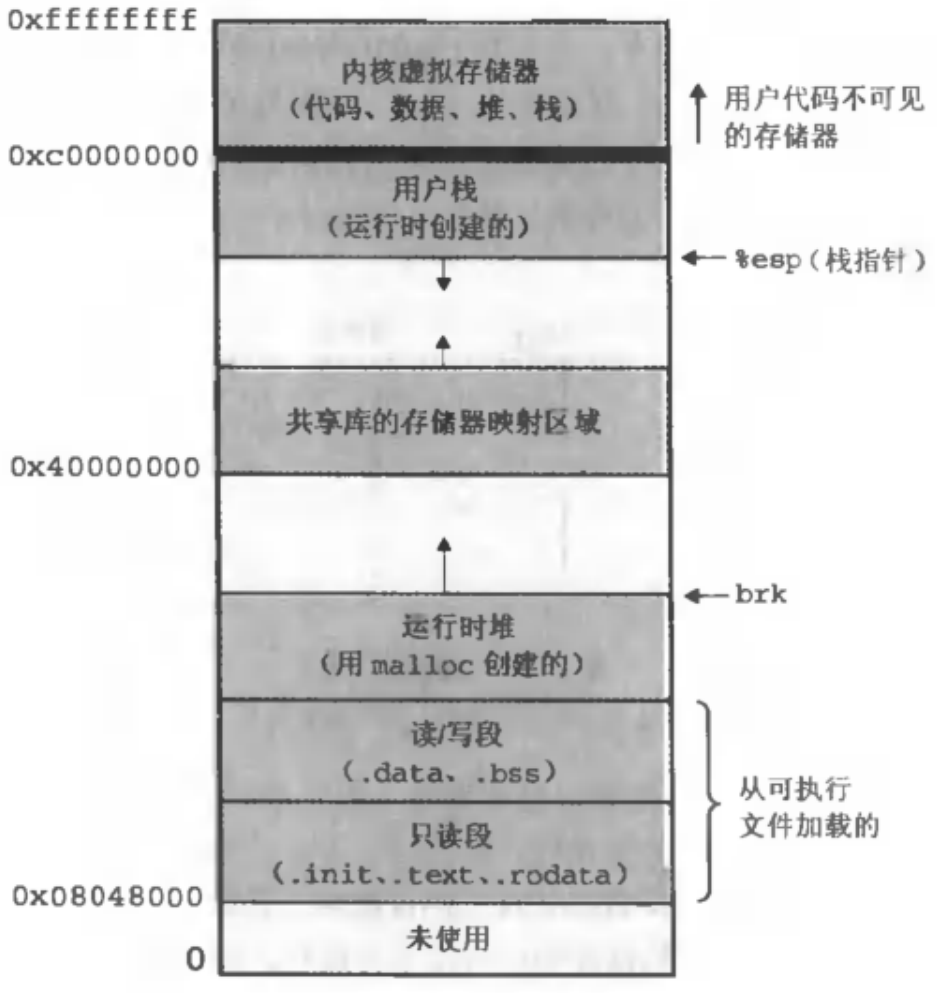


图 8.11 进程地址空间

用户模式与内核模式

处理器通常是用某个控制寄存器中的模式位来限制应用可以执行的指令以及它可以访问的地址空间范围。
没有设置模式位时，进程运行在用户模式中，否则运行在内核模式中。
进程从用户模式变为内核模式的唯一方法是通过异常。

上下文切换

- 内核数据结构
- 页表
- 进程表
- 文件表

errno 与 strerror()

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

- errno: 全局整数变量，储存错误类型
- strerror(int errno): 返回描述 errno 对应错误的字符串

自定义函数 unix_error

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

csapp.h 与 csapp.c

http://csapp.cs.cmu.edu/3e/ics3/code/src/csapp.c
http://csapp.cs.cmu.edu/3e/ics3/code/include/csapp.h
在 Linux 或者伪 Linux 环境下编译。另外由于 csapp.c 里有多线程相关代码，编译时请加上-pthread。

获取进程 ID

```c
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void); /* 调用进程 ID */
pid_t getppid(void); /* 父进程 ID */
```

exit() 与 fork()

```c
#include <stdlib.h>
void exit(int stauts);

#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
/* 子进程返回 0，父进程返回子进程的 PID，如果出错，返回-1
```

fork()

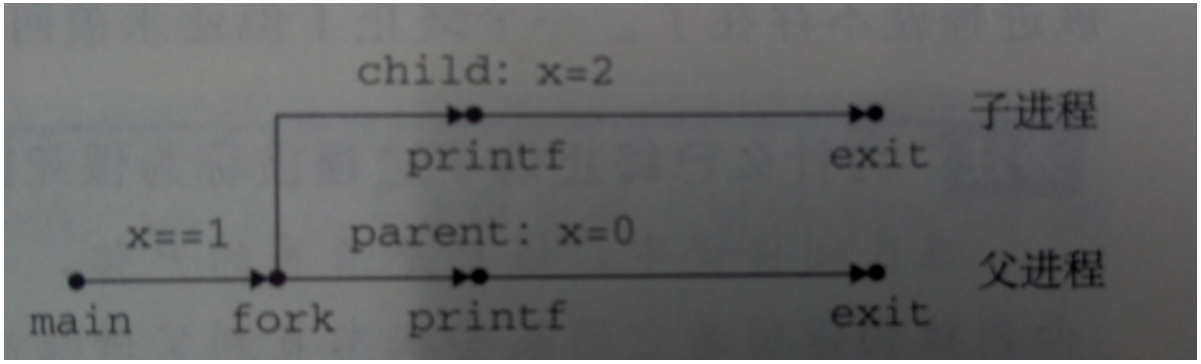
fork() 调用一次，会返回两次

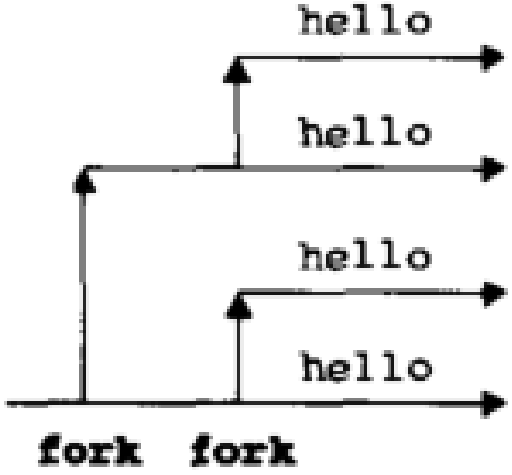
```c
int main() {
  pid_t pid;
  int x = 1;

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

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

parent: x=0
child: x=2


child: x=2
printf
exit
子进程
x==1
parent: x=0
父进程
main
fork
printf
exit


hello
hello
hello
hello
fork
fork

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 */
```

options

- WNOHANG: 如果等待集合中子进程都没终止，立即返回 0。
- WUNTRACED: 挂起调用进程执行，直到等待集合中一个进程终止或停止。
- WCONTINUED: 挂起调用进程执行，直到等待集合中一个正运行的进程终止或者一个被停止的进程收到 SIGCONT 信号重新开始执行。
- 上述常量通过或运算的组合

statusp

如果 statusp 非 NULL，waitpid 就会在 *statusp 放上关于导致返回的子进程的状态信息。

- WIFEXITED(status): 返回子进程是否调用 exit 或者一个返回 (return) 正常终止。
- WEXISTSTATUS(status): 返回一个正常终止的子程序的退出状态。
- WIFSIGNALED(status): 返回子进程是否是因为一个未被捕获的信号终止的。
- WTERMSIG(status): 返回导致子进程终止的信号的编号。
- WIFSTOPPED(status): 返回是否引起返回的子进程当前是停止的。
- WSTOPSIG(status): 返回引起子进程停止的信号的编号。
- WIFCONTINUED(status): 返回子进程是否收到 SIGCONT 信号重新启动。

错误条件

如果调用进程没有子进程，waitpid 返回-1，errno 设置为 ECHILD。如果 waitpid 被一个信号中断，返回-1，设置 EINTR。

回收子进程

```c
#include "csapp.h"
#define N 2

int main()
{
    int status, i;
    pid_t pid[N], retpid;

    /* Parent creates N children */
    for (i = 0; i < N; i++)
        if ((pid[i] = Fork()) == 0)  /* Child */
            exit(100+i);

    /* Parent reaps N children in order */
    i = 0;
    while ((retpid = waitpid(pid[i++], &status, 0)) > 0) {
        if (WIFEXITED(status))
            printf("child %d terminated normally with exit status=%d\n",
                   retpid, WEXITSTATUS(status));
        else
            printf("child %d terminated abnormally\n", retpid);
    }

    /* The only normal termination is if there are no more children */
    if (errno != ECHILD)
        unix_error("waitpid error");

    exit(0);
}
```

让进程休眠

```c
#include <unistd.h>
unsigned int sleep(unsigned int secs);
/* 返回还要休眠的秒数（如未被信号中断会返回 0） */
```

加载并运行程序

```
#include <unistd.h>
int execve(const char *filename, const char *argv[],
const char *envp[]);
/* 如果成功，则不返回，如果错误，则返回-1 */
```

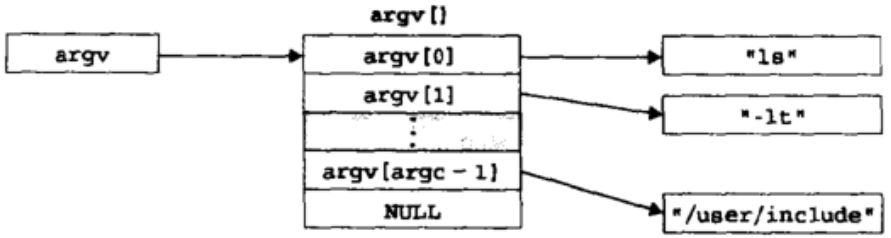


图 8.17　参数列表的组织结构

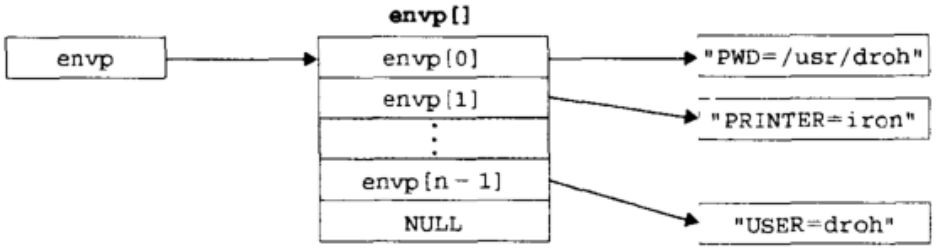


图 8.18　环境变量列表的组织结构