Chapter 08 异常控制流(进程与信号)

- **控制转移(Control Transfer)**
- **控制流(Control Flow)**
- **异常控制流(Exceptional Control Flow, ECF)**: 现代系统通过**使控制流发生突变**来对系统状态的变化做出反应, 这些突变称为ECF.
- **事件(Event)**: 处理器状态的变化叫做**事件**.
- **异常表(Exception Table)**:

  当处理器检测到有事件发生时, 会通过异常表这张跳转表, 进行间接过程调用.

  调用的目标是专门设计用来处理这类事件的操作系统子程序(称作 `异常处理程序`).

  当异常处理完成后, 根据不同情况, 要么返回到产生异常的指令**本身$I_{current}$**, 要么返回到**下一条I_{next}**, 要么直接**终止**程序.
- **异常号(Exception Number)**:

  一个非负整数, 作为异常表中的索引:

  一部分由**处理器**设计者分配, 比如: 被零除, 缺页, 内存访问违例, 断点, 算术运算溢出;

  另一部分由操作系统**内核**设计者分配, 比如: 系统调用, 来自外部I/O设备的信号.
- **异常表的起始地址**放在 `异常表基址寄存器` 这个特殊的CPU寄存器中.
- 在跳转到处理程序之前, 处理器将返回地址压入栈中, 并把一些额外的处理器状态也压入栈中(比如条件码寄存器).
- 异常处理程序运行在 `内核模式` 下, 他们对所有的系统资源都有完全的访问权限.
- 异常的类别: `中断(interrupt)` , `陷阱(trap)` , `故障(fault)` , `终止(abort)`
  - **中断(Interrupt)**:
    - 是**异步**的, 即这个异常不是任何指令造成的, 而是来自外部I/O设备的信号造成的.

      > 具体地, I/O设备通过向处理器的一个引脚发送信号, 并把异常号放在系统总线上, 来触发中断.
      > 而在处理器指令执行完当前指令后, 如果处理器发现 `中断引脚` 的电压升高了,
      > 就会从系统总线读取异常号, 然后调用相应的 `中断处理程序` .

  - **陷阱(Trap)与系统调用**:
    - 陷阱是**有意**引发的异常, 最重要的用途是进行 **`系统调用`** .
    - 所谓系统调用, 是一种在用户程序和内核之间提供一个像过程调用一样的接口.

      这是因为用户程序经常需要**向内核请求服务**(比如读文件read, 创建新进程fork, 加载新程序execve, 终止当前进程exit,

      而这些服务是用户程序自身没有权限进行的.
    - 因此, 处理器提供 `syscall n` 指令, 允许用户程序对内核服务进行 `受控的访问` .
  - **故障(Fault)**:

        - 由错误情况引起, 并且**有可能**被故障处理程序修正.
        - 如果成功修正, 就返回到**当前指令**$I_{current}$重新执行(注意**不是**返回到下一条!);

          如果失败, 则返回到内核中的 `abort` 例程, **终止**引起故障的应用程序.
        - 故障的例子: **缺页异常**: 当指令引用一个虚拟地址, 而该地址对应的物理页面不在内存中, 因此需要从磁盘中取出.

          当成功取出后, 控制将返回给引起故障的当前指令, 让这条指令重新执行.

          这次当然就不会再遇到缺页故障了! 因此是可恢复的故障. 也表明了为什么是返回到 $I_{current}$.
    - **终止(Abort)**:
        - 不可恢复的致命错误.
        - 通常是硬件错误, 比如SRAM或DRAM的位发生损坏造成的奇偶错误.
        - 处理程序将把控制转移给 `abort` 例程.
- `Linux/x86-64` 系统中的异常

  x86-64中有256种不同的异常类型. 其中, 0~31是Intel架构师定义的异常; 32~255是操作系统定义的 `中断` 和 `陷阱` .
    - `Linux/x86-64` 的 **`故障`** 和 **`终止`**
        - **除法错误**: Unix不会试图恢复错误, 而是选择终止程序.
        - **一般保护故障**: 很多种类型, 典型的是引用未定义的虚拟内存区域, 或者向只读文本段写入等等. Linux不会尝试恢复这类故障.
        - **缺页**: 会重新执行产生故障的指令.
        - **机器检查**: 检测到致命的硬件错误. 终止程序.
    - `Linux/x86-64` 的 **`系统调用`**

      包括读文件, 写文件, 创建新进程等几百种系统调用.

      每个系统调用都对应唯一的一个整数号, 对应于一个内核中的 `跳转表` 的偏移量

      (注意, 这个 `跳转表` 当然不是之前提到的异常表! 比如 `这个表` 的0是read.)

      C语言的 `syscall` 函数可以直接调用任何系统调用, 不过一般我们会直接用标准C库提供的 `包装函数` (统称为 `系统级函数`)

      x86-64的 `syscall陷阱` 函数用于实现系统调用.

      注意这个调用的参数必须用寄存器(而不是栈)传递, 而且按照惯例:

      `%rax` 包含系统调用号, `%rdi,%rsi,%rdx,%r10(注意不是%rcx),%r8,%r9` 依次传递6个参数;

      从系统调用返回时, `%rcx` 和 `%r11` 可能会被修改, `%rax` 则存储返回值.

- **上下文(Context)**: 程序正确运行所需要的**状态**的总和.
- **进程(Process)**: 提供给应用程序的关键抽象:
    - **独立的逻辑控制流**: 提供一个假象, 好像程序在独占地使用**处理器**.
    - **私有的地址空间**: 提供一个假象, 好像程序独占地使用**内存系统**.
- **进程管理块(Process Control Block, PCB)**

- **逻辑控制流**:

  多个进程轮流地使用处理器: 每个进程执行它的流的一部分, 然后被 **`抢占(preempted, 暂时挂起)`** , 然后轮到其它进程使用处理器.
- **并发**:
  - **并发(Concurrency)**: 多个流并发地执行的现象.
  - **并发流(Concurrent Flow)**: 一个逻辑流在时间上和另一个逻辑流重叠.
  - **并行流(Parralel)**: 并发流的真子集. 如果两个并发流运行在不同的 `处理器核心` 或不同的 `计算机` 上, 特别地称之为 `并行流` .
  - **多任务(Multitasking)**: 一个进程和其它进程轮流运行.
  - **时间片(Time Slice)**: 一个进程执行它的控制流的一部分时间的每一个时间段.
- **私有地址空间**:

  进程为每个程序提供它的**私有地址空间**. 这个空间中的内存字节一般不能被其它进程读写, 所以说是"私有的".
- **用户模式**和**内核模式**:
  - 是**处理器**提供的一种机制, 用于限制应用程序**可以执行的指令**和**可以访问的地址空间范围**.
  - 处理器使用 `控制寄存器` 的一个 `模式位(mode bit)` 来提供这种机制.
  - 当设置了这个bit时, 进程就运行在 **`内核模式(超级用户模式)`** 中, 否则运行在 **`用户模式`** .
  - 在用户模式下, 不能执行 `特权指令(Previleged Instruction)` , 比如停止处理器, 改变模式位, 发起I/O操作等等.

    也不允许访问内核区域的内存, 否则会直接引发 `致命的保护故障` .

    所以用户模式下必须使用 `系统调用` 间接访问内核的代码和数据.
  - 进程从用户模式变成内核模式的**唯一方式**是通过 `异常` (中断, 故障, 系统调用).

    当异常发生时, 控制传递到异常处理程序, 此时处理器模式切换成 `内核模式` . 而当返回到应用程序时, 切换回用户模式.
  - 在Linux中, 有一种机制叫做 `/proc文件系统` . 这种机制让进程在 `用户模式` 下也可以访问内核数据结构的(部分)内容.

    比如: 查看CPU类型: `/proc/cpuinfo` ; 查看某个进程使用的内存段: `/proc/<process-id>/maps` .
- **上下文切换(Context Switch)**

  内核为每个进程维持一个 **`上下文`** , `上下文` 即内核重新启动一个被抢占的进程所需的 `状态`

  (这些状态包括: 通用寄存器, 浮点寄存器, 程序计数器, 用户栈, 状态寄存器, 内核栈, 页表, 进程表, 文件表 等等).

  在进程执行时, 内核随时可以决定 "抢占当前**进程**, 重新开始一个(之前被抢占了的)**进程**" .

  上下文切换将: (1)**保存**当前进程的上下文 (2)**恢复**某个之前被抢占的的进程所保存的上下文 (3)将**控制传递**给新恢复的进程.
- **何时**发生上下文切换?
  1. 当**内核(代表用户)执行系统调用时**, 可能发生上下文切换:
     1. 比如系统调用可能因为等待某个事件而 **`阻塞`** , 比如 `read(的磁盘访问)` , 又或者如 `sleep()` 显式地请求进程休眠.

2. 即使系统调用没有`阻塞`, 内核也可以决定执行上下文切换, 而不是返回到调用系统调用的应用程序.

2. 当**遇到`中断`异常**时, 可能发生上下文切换: (回忆"中断"特指由外部I/O信号引发的异常)

   所有的系统都会有一个`周期性定时引发中断`的机制, 通常为1ms或10ms.

   当定时器产生中断信号时, 内核判定当前进程已经执行了较长的时间, 应该轮到别的进程执行了.

- 对**系统调用**发生的错误的处理:

  当Unix系统级函数遇到错误时, 通常会返回`-1`, 并设置全局整型变量`errno`来标志出了什么错.

  一个检查fork()函数产生错误的例子:

```
if((pid = fork()) < 0){  // 返回-1说明发生了错误
    fprintf(stderr, "fork()产生错误: %s\n", stderror(errno));
  exit(0);
}
```

  这种检查错误的方式比较臃肿. Unix提供了对`fprintf()`的包装函数`unix_error()`, 这样, 上述例子可以等价地改写为:

```
if((pid = fork()) < 0){
  unix_error("fork()产生错误:");
}
```

  其实可以进一步简化: 假如系统级函数名为`func`, 我们定义包装函数`Func`(将首字母大写), `Func()`和`func()`**具有相同的参数**.

```
// 以fork()函数的包装函数Fork()为例:
pid_t Fork(void){
    pid_t pid;
  if((pid = fork()) < 0){
    unix_error("fork()产生错误:");
  }
  return pid;
}
```

  这样, 以后我们就可以用更简单的方式进行系统调用, 并进行错误检查:

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

> 后文中, 我们会简单地用fork()来代指包含错误检查的Fork().

进程控制---在C语言中利用系统调用操作进程

- `获取进程ID`:
  - 每个**进程**都有一个`唯一的`正数**进程(Process)ID**, 称为`PID`.

- `getpid()`: 返回调用 `getpid()` 的进程的PID.

  `getppid()`: 返回调用 `getpid()` 的进程的 `父进程` 的PID.

  它们的返回值类型为 `pid_t`. 在Linux中, `pid_t` 在 `types.h` 中被定义为 `int`.

- **`创建和终止进程`**:
  - 进程的三种状态:
    - **`运行`**: 正在CPU上执行, 或者等待被内核调度.
    - **`停止(Stop)`**: 进程被 `挂起(suspended)` 了, 即不会被内核调度. (但并没有终止(Terminate))

      停止由 `SIGSTOP/SIGTSTP/SIGTTIN/SIGTTOU` 信号引起;

      而 `SIGCONT` 信号将让这个进程解除停止, 继续会被内核调度运行.
    - **`终止(Terminate)`**: 进程永远地停止了.

      由三种原因引起: (1)收到一个表示终止进程的信号 (2)main()函数执行返回 (3)调用exit()函数
  - `fork()`: 父进程通过调用 `fork()` 函数创建一个**子进程**.
    - 子进程**几乎**与父进程完全相同, 它会得到和父进程的用户级虚拟地址空间的 **`独立的副本`**, 并继承父进程所有的 **`打开文件`**.
    - 子进程和父进程的最大**区别**在于他们拥有不同的**PID**.
    - fork()函数会**"执行两次返回"**:

      一次返回到父进程, 此时返回值为子进程的PID;

      一次返回到子进程, 此时返回值为 `0`.

      程序通过检查返回值是否为0, 可以区分此时是处于子进程还是父进程.
    - 子进程和父进程是**并发地**执行的, 且是独立的两个进程.

      所以我们不能对这两个进程中的指令执行顺序进行任何假设. (因为鬼知道内核先调度谁)
    - 子进程和父进程的地址空间**相同**, 但相互**独立**. 因此一个进程对变量的修改不会影响另一个进程.
    - 子进程继承父进程所有的 `打开文件`.

      比如父进程调用fork()时, `stdout` 文件是打开的并指向屏幕. 因此子进程也可以利用这个文件将信息输出到屏幕.

- **`回收子进程`**:
  - 当一个进程终止时, 内核并不会立即清除它, 直到它的父进程将它**回收(reaped)**.
  - 一个终止了但尚未被回收的进程称为**僵死进程(zombie)**.

    > Linux的 `ps` 指令可以用于查看某个进程是否为僵死进程(被标记为 `defunct`). (ps是process status的缩写)

  - PID为1的 `init进程` 在系统启动时由内核创建. 如果一个进程的父进程先终止, 则init进程负责回收这些子进程.
  - `waitpid()函数`:

将调用这个函数的进程挂起, 直到: (1) `等待集合(wait set)` 中的其中一个子进程**终止**. 或者(2)如果在调用waitpid()时, `pid` 参数指明的等待集合中已经有终止了的子进程了, 就直接返回. 这两种情形的返回值都是已终止的子进程的PID, 且相应的子进程成功被**回收**(所以wait函数不仅仅是wait, 还要reap!). 回收后, 内核才会把它从系统中完全删除.

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

> `wait(&status)` : 等价于 `waitpid(-1, &status, 0)` . 可以理解为是waitpid()的简化版.

- `pid` : 确定等待集合: 如果 `pid > 0` , 那就是确切地等待pid为 `pid` 的那个进程; 如果 `pid == -1` , 那等待集合就是**全体子进程**.
- `option` : 默认为0, 可以通过设置为 `WNOHANG` / `WUNTRACED` / `WCOUNTINUED` 或它们之间的组合来修改 `waitpid()` 的默认行为:
  - `WNOHANG` : 如果等待集合中任何子进程都没有终止, 则**立即返回**, 返回值为0.
  - `WUNTRACED` : 终止**以及停止**的子进程均会引发 `waitpid()` 的返回, 返回值为对应的PID.
  - `WCONTINUED` : 终止**以及本来在停止但收到SIGCONT信号重新开始执行**的子进程均会引发 `waitpid()` 返回对应的PID.
  - `WNOHANG|WUNTRACED` : 如果子进程都没有被停止或终止, 则返回值为0; 如果有, 则返回相应的PID.

    > 这个 `|` 就是让两个功能融合了一下, 注意这里不是 `&` . 考虑位向量中 `1` 表示选择之意, 所以 `|` 才能表示功能的组合.

- `statusp` : 检查被回收的子进程的退出状态, `waitpid()` 会在 `*statusp` 写入导致返回的子进程的状态信息.

  可能写入这些: `WIFEXITED` , `WEXITSTATUS` , `WIFSIGNALED` , `WTERMSIG` , `WIFSTOPPED` , `WSTOPSIG` , `WIFCONTINUED` .
- `waitpid()` 何时返回 `-1` ?

  1. 如果调用函数的进程**没有子进程**. 此时会设置 `errno` 为 `ECHILD` 来标明.
  2. 如果waitpid()函数自身**被一个信号中断**. 此时会设置 `errno` 为 `EINTR` 来标明.(Interrupted)
- `statusp` 用于返回导致 `waitpid()` 返回的子进程的状态. 使用几个宏函数检查statusp.

- **`让进程休眠`**
  - `sleep()` 函数: 让进程挂起一段指定的时间. 如果请求的时间到了, 则返回0; 否则返回还剩下需要休眠的秒数. 之所以会发生后者, 是因为sleep()函数本身可能会被一个信号中断, 从而提前返回.
  - `pause()` 函数: 让进程挂起, 直到该进程收到任意一个信号.

- **`加载并运行程序`**
  - `程序` 与 `进程` 的区别:

    程序是一堆代码和数据; 进程是执行中的某个程序: 程序总是运行在某个进程中.
    - 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. 新的程序仍然有相同的PID.

      因为虽然它会覆盖当前进程的地址空间, 但并没有创建一个新的进程.
  - `execve()` 函数:

    > v: vector, 即命令行参数通过**指针数组**(vector of pointer)的方式传递.
    >
    > e: environment_variables, 即环境变量通过**指针数组**的方式显式传入. 参考wiki

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

    - `execve()` 函数在当前进程中, 加载并运行一个新的程序.
    - 可执行目标文件由 `filename` 给出, 命令行参数由 `argv` 给出, 环境变量由 `envp` 给出.
    - 除非出现错误, 比如filename未知, 否则 `execve` 函数不再会返回到调用它的程序中!
    - 当execve加载了filename程序之后, 它调用启动代码, 设置栈, 将控制传递给新程序的main函数.

- Shell: 利用 `fork()` 和 `execve()` 运行程序
  - **后台执行**与**前台执行**:

    如果最后一个参数是一个 `&` 字符, 那么 `parseline()` 函数将返回1, 表示应该在Shell的后台执行.

    对于前台执行, shell可以简单地使用waitpid函数来等待该作业终止.

    而对于后台执行, 看起来很简单, 那就是直接不使用 `waitpid()` 函数,

    不过, 回忆waitpid()可不仅仅是等待, 更有着回收子进程的重大使命! 如果不使用waitpid, shell就不能实现对后台子进程的回收!

    为了实现对后台子进程的回收, 我们需要进一步了解 `Linux信号` (下一节).

- **`Linux` 信号**: 软件形式的异常, 允许进程和内核中断其他进程.
- **`发送信号`** : **内核**通过更新目的进程的上下文中的某个状态, 发送一个信号给目的进程.

  发送信号的两种原因:

  1. 内核检测到一个系统事件
  2. 一个进程调用kill()函数, 显式地要求发送一个信号给目的进程(也可以是自己).
- **`接收信号`** : 目的进程被内核强迫以 `某种方式` **响应**信号. 响应的方式包括:

  1. **忽略**这个信号
  2. 进程**终止**
  3. 执行 `信号处理程序(signal handler)` 的用户层函数, 称作 **`捕获`** 这个信号
- **`待处理信号(pending signal)`** : 一个发出但没有被接收的信号.

- 在任何时刻, 同种类型的待处理信号最多有一个.
  这是因为内核是通过为每个进程是通过一个 `pending` 位向量来维护待处理信号的, 每种类型分配到了其中一个bit,
  每当传送了一个类型为k的信号, 就会设置第k位为1;
  每当 `接收` 了一个类型为k的信号, 就会清除第k位.
- 一个待处理信号最多只能被接收一次. (由上一行)
- 进程可以选择 **`阻塞接收`** 某个类型的信号, 当这种信号被发送后**不会被接收**, 直到进程取消对这类信号的阻塞.
  内核通过为每个进程维护一个 `blocked` 位向量, 表示被阻塞的信号的集合.

- `进程组`:
  - 每个进程都属于且仅属于一个进程组, 每个 `进程组` 由一个正整数 `进程组ID` 标识.
  - `getpgrp()` 函数返回当前进程的进程组ID. (get process group)
  - 默认地, 一个子进程和它的父进程同属于一个进程组.
    不过, 可以通过 `setpgid(pid, pgid)` 函数改变指定进程(自己or他人)的进程组.
    这个函数很有意思, 如果pid为0, 表示设置自己进程的进程组; 如果pgid为0, 表示把进程组ID设置为指定进程的PID(即PGID:=PID)
- 用 `/bin/kill` 程序给指定进程发送信号(不一定是SIGKILL!)

  > 这又是一个名字和功能不那么一致的东西.

  使用格式: `linux> /bin/kill -type [-]id` ,
  其中 `type` 为信号类型, `[]` 中的 `-` 为可选项, 如果 `-` 存在, 则 `id` 表示 `gpid` , 否则表示 `pid` .
  例如: `linux> /bin/kill -9 -15213` 会给进程组15213的每个进程发送信号9, 即SIGKILL信号.
- **作业(job)**: 指为了对一条命令行求值而创建的进程.
  在任何时刻, Unix Shell 至多有一个 `前台作业` 和0个或多个 `后台作业` .
  shell为每个作业创建独立的进程组, 进程组ID取自作业中父进程中的一个.
  在键盘上输入ctrl+C会让内核发送一个 `SIGINT` 信号给前台进程组中的**每个进程**, 默认效果是终止前台作业;
  在键盘上输入ctrl+Z会让内核发送一个 `SIGTSTP` 信号给前台进程组中的**每个进程**, 默认效果是停止(挂起)前台作业;

  > 注意这两个**都是可以**被捕获的! 不能被捕获的是 `SIGSTP` 和 `SIGKILL` !

- 用 `kill(id, sig)` 函数发送信号.
  和上面说的 `/bin/kill` 的命令行指令很相似:
  如果id>0, 发送信号sig给id;
  如果pid<0, 发送信号给进程组id;
  如果pid==0, 发送信号给调用进程所在进程组的每个进程(包括调用进程自己)
- 用 `alarm(uint secs)` 函数给自己发送信号:

进程可以通过调用 `alarm()` 函数给自己发送 `SIGALRM` 信号.

(这里p530最下面不是特别明白)

- 当内核把进程p从内核模式切换到用户模式时, 会检查进程p的 `未被阻塞的待处理信号` 们(`pending&~blocked`)
  - 如果这个集合是空的, 那就没啥要特别处理的, 内核将控制传递到下一条指令(I_{next})即可;
  - 如果这个集合非空, 那么内核会选择集合中最小的信号k, 强制进程p接收信号k, 而收到这个信号会触发进程采取 `某种行为` .

    直到这个集合变成空的. 才会返回到I_{next}.
- 每个信号都有一个预先定义的 `默认行为` ("默认"暗示我们有时可以修改成别的), 是以下中的一种:
  - 终止进程
  - 终止进程并转储内存(?)
  - 停止(挂起)直到被SIGCONT信号重启.
  - 忽略该信号
- 进程可以通过使用 `signal()` 函数**修改(override)**默认行为.

  不过, `SIGSTP` 和 `SIGKILL` 是例外, 它们的默认行为(即终止和停止)是不可被修改的!

  (不过再次强调, ctrl+C发送的是 `SIGINT` , ctrl+Z发送的是SIG**T**STP, (**T**即Terminal), 均是可以被修改的! 这也是tsh的运作原理之一)

```
#include<signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

  - 如果handler是 `SIG_IGN` , 那么忽略类型为signum的信号
  - 如果handler是 `SGI_DFL` , 那么类型为signum的信号恢复默认行为
  - 否则, 当类型为 `signum` 的信号被接收时, 进程就会调用 `handler()` 程序, 称作 `信号处理程序` .

    正如之前提到过的, **调用**信号处理程序被称作 `捕获信号` ; **执行**信号处理程序被称为 `处理信号` .

    > 信号处理程序运行在用户态.

- **阻塞信号**和解除阻塞:
  - **隐式阻塞机制**: 内核默认阻塞任何当前处理程序(S)正在处理的信号类型(s)的待处理信号.

    比如程序捕获了信号s, 当前正在运行处理程序S, 那么此时发送给该进程另一个信号s, 则s不会被立即接收, 而是变成待处理.

    直到处理程序S返回.
  - **显式阻塞机制**: 使用 `sigprocmask()` 函数和它的 `辅助函数` , 阻塞或解除阻塞选定的信号.

```
int sigprocmask(int how, const sigset_t* set, sigset_t* save);
```

    `how` : `SIG_BLOCK` , `SIG_SETMASK` , `SIG_UNBLOCK` . 前两个比较常用, 常常用于阻塞/恢复.

`save`: 如果不是`NULL`, 则`block`位向量的旧值保存在`*save`里.
`辅助函数`: 用于生成`set`. 包括函数: `sigemptyset`, `sigfillset`, `sigaddset`, `sigdelset`. 字面意思.

- 信号处理程序与主程序并发运行, 共享相同的全局变量.

  > 这里的<并发>只是说<处理程序在逻辑上被插入到主程序中运行>, **而非**指<处理程序和主程序可以来回上下文切换>

- **安全**的信号处理:
  - 处理程序要尽可能简单.

    > 比如设置某个全局标志并立即返回. 所有与接收信号相关的处理让主程序去执行(这需要主程序定期检查这个标志).

  - 只调用`异步信号安全`的函数.
  - 保存和恢复`errno`.

    > 因为异步信号安全的函数在出错时会设置errno, 信号处理程序如果调用了这类函数, 就可能干扰主程序中对errno的检查.

  - 当访问全局数据结构时, 应该阻塞所有的信号.

    > 这需要我们存储之前的blocking位向量为prev, 然后设置blocking为mask_all, 最后还原blocking位向量为prev.

  - 用`volatile`关键字声明全局变量.

    > volatile关键字强迫编译器不要缓存这个变量, 而是去内存中读取. 同样地, 访问这类全局变量也应该阻塞所有的信号.

  - 用`sig_atomic_t`声明全局标志.

    > 这是C语言提供的一种整型数据类型, 保证对这种类型的读/写是`原子的`, 即不会被中断, 从而不需要暂时阻塞信号.
    > 不过要注意, 这种原子性只针对单独的读/写, 如果是flag++这种, 由于需要多条指令, 所以可能被打断!

- **正确**的信号处理:
  - 不要尝试用`信号`来对其它进程中发生的事件计数

    > 因为待处理的(pending)信号存储在`pending`位向量中, 所以只能标记有or没有, 而不能表示数量.

- **可移植**的信号处理:

  Signal包装函数:

  - 只有处理程序当前正在处理的哪种类型的信号被阻塞
  - 信号不会排队等待
  - 只要可能, 被中断的系统调用会自动重启
  - 一旦设置了自定义的信号处理程序, 就会一直保持, 除非显式地带着 `SIG_IGN` 或 `SIG_DFL` 的 `handler` 参数来调用Signal().

- 使用 `sigsuspend()` 函数显式地等待信号.

  为什么要用sigsuspend()?

  考虑一个现实的问题: 比如我们要实现让shell等待前台程序运行完毕, 我们现在有两种**错误或不太好**的方案:

```
// 注: 处理SIGCHLD的handler会设置全局变量pid≠0, 从而让while循环终止.
// 方案1
pid = 0;
while(pid == 0){
    ;
}

// 方案2
pid = 0;
while(pid == 0){
    pause();
}
```

  方案1会浪费大量CPU资源;

  方案2 is BUGGY! 因为如果在 <while测试指令后, pause指令前> 收到了 `SIGCHLD` 信号, pause()将永远睡眠. 因为再也没有信号会打断pause()!

- 我们将基于方案2的思路, 利用一个神奇的系统调用 `sigsuspend` 来解决问题:

  `sigsuspend(mask)` 暂时用mask替换当前的阻塞集合, 然后挂起该进程(类似pause的功能), 直到收到一个信号. 这个函数等价与下述代码的 `<` 原子版本(不可中断的版本) `>` :

```
<
sigprocmask(SIG_SETMASK, &mask, &prev);
pause();
>
sigprocmask(SIG_SETMASK, &prev, NULL);
```

  使用方法是: 进入while循环前仍然是把 `SIG_CHLD` 阻塞, 然后在 `sigsuspend` 中暂时取消对 `SIG_CHLD` 的阻塞.

- **非本地跳转(nonlocal jump)**: 将控制直接从一个函数转移到另一个当前正在执行的函数, 而不经过正常的 `调用–返回` 规则.

通过 `setjmp()` 和 `longjmp()` 函数实现:

```
#include<setjmp.h>
int setjmp(jmp_buf env);
int sigsetjmp(sigjmp_buf env, int savesigs);
```

`setjmp()` 保存当前的 `调用环境` , 存储在 `env` 中; 返回值为0 (表明这是第一次返回, 具体原因见下).

`调用环境` : 包括程序计数器PC, 栈指针%rsp, 和通用目的寄存器.

```
#include<setjmp.h>
void longjmp(jmp_buf env, int retval);
void siglongjmp(sigjmp_buf env, int retval);
```

`longjmp()` 函数从env恢复调用环境, 然后返回到最近一次初始化env的setjmp()的位置,
这时setjmp()的返回值为**非零的**retval, 表示是通过longjmp()返回的, 而不是第一次返回(which返回值==0).
另外, `longjmp()` 是没有返回值的! 因为也没有必要.(虽然错误返回-1好像也有点道理?...)

> 我们可以发现有两个带有sig前缀的函数: sigsetjmp和siglongjmp,
> 它们是可以被信号处理程序使用的版本.

Chapter 10 系统级I/O

- **输入/输出(I/O)**: 主存和外部设备之间复制数据的过程.
  - 输入: `外部设备` 复制到 `主存`
  - 输出: `主存` 复制到 `外部设备`
- **Linux文件**: 一个m个字节的序列.
  - 所有的I/O设备都被模型化为文件.
  - 所有的输入和输出都被当做对相应文件的读/写来执行.
- **Unix I/O**: 把外部设备抽象成文件使得所有的输入输出可以被统一成一致的方式来执行:
- **描述符(file descriptor)**: 内核返回的一个小的非负整数, 在进程后续对此文件的操作中标识这个文件.

内核记录这个文件的信息, 而应用程序只需记住这个描述符.

- Linux **Shell 创建的**进程**在最开始**有3个打开文件: 标准输入(STDIN, 0, 键盘), 标准输出(STDOUT, 1, 屏幕), 标准错误(STDERR, 2).

  > STDERR不需要flush就会输出, 没有缓冲区

- **文件位置k**: 内核维护的一个值, 初始为0, 表示从文件开头开始的一个字节偏移. `seek(k)` 设置文件的当前位置为k.
- **读文件**: (1)从文件复制n个字节到内存 (2)设置文件位置从k变为k+n.
  当文件位置k≥文件总字节数m时, 若执行读操作, 会触发 `EOF` .
  应用程序可以检测到这个条件, 但文件结尾处并没有所谓的"EOF符号".
- **写文件**: 类似读文件
- **关闭文件**: 应用程序可以通知内核关闭某个文件, 内核释放文件打开时创建的数据结构, 并释放这个描述符, 使其变成可用的.
  当一个进程终止时, 内核会关闭这个进程的所有打开的文件.

- 文件的**类型(type)**:
  - 普通文件(regular file): 包括文本文件和二进制文件. 内核并不区分这二者(虽然应用程序常常会区分)
  - 目录(directory): 包含一组 `链接` 的文件, 每个链接将一个 `文件名(filename)` 映射到一个 `文件` . (这个文件可能是另一个目录)
    每个目录至少包含两个条目: `.` (到目录自身的链接) 和 `..` (到父目录的链接).
  - 套接字(socket): 用于与另一个进程进行跨网络通信的文件.
- **根目录**: `/`
- **当前工作目录(current working directory)**
- **绝对路径名(absolute pathname)**: 以斜杠 `/` 开始, 表示从根节点开始的路径.
  相对路径名(relative pathname): 以文件名(包括 `.` 和 `..`)开始, 表示从当前工作目录开始的路径.

- **打开文件**:

```
1  int open(char* filename, int flags, mode_t mode);
```

  - `返回值` : 描述符, 且是在进程中当前没有打开的最小描述符.
  - `flags` : 指明进程打算如何访问这个文件: 只读? 只写? 读写? 如果文件不存在? 如果文件存在? 在结尾append?
  - `mode` : 指定新文件的访问权限. 文件的访问权位限被设置为 `mode& ~umask` , 其中 `umask` 市进程上下文的一部分.
- **关闭文件**:

```
int close(int fd);
```

- `返回值`: 成功则为0, 失败返回-1. (关闭一个已关闭的描述符会出错)

  > 我们一定要检查返回值, 否则会有线程灾难.

- `fd`: 描述符.
- close一个fd, 会让fd指向的 `打开文件表` 中的相应表项的 `引用计数cnt` 减1.

- **读写文件**

```
#include<unistd.h>
ssize_t read(int fd, void* buf, size_t n);  // 成功返回读的字节数, EOF返回0, 失败返回-1
ssize_t write(int fd, const void* buf, size_t n);   // 返回写的字节数, 失败返回-1
```

`ssize_t` 是有符号数, 这是为了在失败的情况下返回-1. 而 `size_t` 则是无符号数.

- 遇到**不足值(shortcount)**: 即read和write实际传送的字节数比应用程序所要求的少. 原因有:
  1. **读**的时候遇到了EOF:

     比如文件位置离末尾还有20字节, 而你却想读50字节, 那么返回值20就是一个不足值.

     如果文件位置已经是末尾了, 这时read会返回0, 这个0也是不足值, 而不是一个错误(which 返回-1).
  2. 从终端**读**文本行:

     如果打开文件是与终端相关联的, 比如键盘和显示器, 那么每次read将传送一个文本行, 不足值为文本行的大小.
  3. **读/写**网络套接字或其它进程间通信时.

  为了创建**可靠的(Robust)**诸如Web服务器的应用, 需要反复调用read和write来处理不足值, 直到所有的字节都传送完毕.

- **健壮I/O包(Robust I/O, RIO)**: 提供自动处理不足值的读/写函数.
- **无缓冲的** `输入/输出` 函数: 直接在内存和文件之间传送数据.

```
ssize_t rio_readn(int fd, void* usrbuf, size_t n);
ssize_t rio_writen(int fd, void* usrbuf, size_t n);
```

- **带缓冲的** `输入` 函数: 用于从文件中读取 `文本行和二进制` 数据. 这些文件的内容缓存在应用级缓冲区中.

这里懒得看, 记得补笔记

- **元数据(metadata)**: 关于文件的信息.
- 读取元数据: stat() 和 fstat() 函数.

  这两个函数的返回值是一样的, 是一个stat结构体类型, 只不过stat以文件名为参数, 而fstat以描述符为参数.

  我们主要关心这个结构体中的两个字段:

- `st_size`: 包含了文件的字节数大小.
- `st_mode`: 编码了文件的访问许可位.
- `st_ino`: i-node, Linux下唯一标识文件的字段.

- **读取目录内容**

  `readdir()` 系列函数用于读取目录的内容. 它以路径名为参数, 返回指向 **目录流** 的指针.

  > 所谓流, 是对条目有序列表的抽象, 在这里特指目录项的列表.

  每次对readdir的调用返回指向流dirp中下一个目录项的指针, 如果没有更多的目录项则返回NULL.

- **共享文件**
  - **描述符表(descriptor table)**: 每个进程都有它自己的独立的描述符表. 它的每个表项指向 **文件表** 中的一个表项.
  - **打开文件表(file table)**: 所有进程共享这张表.

    每个表项中包含 `文件位置` (seek的那个!), `引用计数(reference counter)` 以及一个指向 **v-node表** 表项的指针.
  - **v-node表(v-node table)**: 所有的进程共享. 表项中包含stat结构体中的大部分信息, 包括 `st_mode` 和 `st_size`.

    这个v-node和之前所说的inode是对应的, 也是唯一对应一个磁盘(?)文件.
  - 多个描述符可以通过不同的文件表表项引用**同一个文件**.

    原理是: 比如两个不同的描述符分别为fd=1和fd=4, 二者指向不同的文件表表项, 因此可以有不同的 `文件位置` 字段.

    但这两个表项指向同一个 `v-node表` 表项, 共享同一个磁盘文件.
  - 子进程继承父进程打开文件的原理: 子进程获得父进程的 `描述符表` 的副本. (仅此而已)

- **I/O重定向**

  原理: 更改描述符表中的表项(描述符).

  对dup2的记忆: `dup2(fd1, fd2)`, 其中 `fd1` 是要被复制的对象, 而 `fd2` 是我们指定的 `用于存放复制出来的副本` 的描述符.

  (如果fd2原来是打开的, 则os会隐式关闭它)
-