Lesson 11
ECF 2 & System I/O

ICS Seminar #9

张龄心

Nov 29, 2023

信号

- 由内核/其他进程发送给进程
- 参考P527的表格, 至少要了解常见信号的含义和对应行为
    - SIGINT, SIGILL, SIGFPE, SIGKILL, SIGSEGV, SIGUSR1 & SIGUSR2, SIGALRM, SIGCHLD, SIGCONT, SIGSTOP, SIGTSTP
- SIGSTOP和SIGTSTP
    - SIGSTOP: 不是来自终端的终止信号, 不能被捕获/忽略
    - SIGTSTP: 来自终端的终止信号(Ctrl+Z), 其默认行为可以更改
- 进程收到信号后, 可能:
    - 按默认行为进行 (终止/忽略等)
    - 按自定义行为进行 (可以自定义信号处理程序)

信号

- 多个同类信号的处理
  - 回顾: (单核)操作系统中, 多个进程轮流运行在CPU上
  - 如果进程a正在CPU上运行, 则马上能收到信号并处理
  - 如果进程a在排队等候运行, 则发给a的信号不能马上被处理->pending
    - 轮到a在CPU上运行时, 它才能开始处理这些待处理信号
      - 这时可能a已经收到了一堆待处理信号
  - 待处理信号的保存方式: pending数组, 只有0和1

    ->**进程只能知道”自己收到过某类信号”, 但不能知道总共收到了几次**
- 信号的发送: alarm和kill
  - pid>0针对单个进程, pid<0针对进程组

信号

- 阻塞信号: 进程可以暂时阻塞某类信号, 导致该类信号排队
  - 隐式阻塞(处理中的信号会被阻塞) / 显式阻塞(sigprocmask导致的)
- 信号的阻塞: sigprocmask, 以及set修改相关的函数

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
```

返回：如果成功则为0，若出错则为−1。

```
int sigismember(const sigset_t *set, int signum);
```

返回：若signum是set的成员则为1，如果不是则为0，若出错则为−1。

信号

- 自定义信号处理: signal函数

```
sighandler_t signal(int signum, sighandler_t handler);

// 返回：若成功则为指向前次处理程序的指针，若出错则为 SIG_ERR（不设置 errno）。
```

- 参数handler可以有三种情况:
  - SIG_IGN = ignore
  - SIG_DFL = default
  - 用户自定义的函数地址

信号

信号处理原则：

- 1.处理程序尽可能简单
- 2.只调用异步信号安全函数
- 3.保存恢复errno
- 4.访问全局数据结构时阻塞所有信号
- 5.全局变量使用volatile声明
- 6.标志使用sig_atomic_t声明

信号

- 异步信号安全函数:
  - 要么是可重入的(不使用共享数据), 要么是无法被打断的(原子的atomic)
- Linux中的异步信号安全函数
  - _exit, write, wait, waitpid, sleep, kill
  - 注意: **printf**, sprintf, malloc, exit不是异步信号安全的
- 并发问题: 在12章中进一步介绍
  - sigsuspend: 用某个mask替换当前mask, 等待, 直到收到一个新信号
- setjmp & longjmp
  - setjmp: 保存此时整个进程的状态
  - longjmp: 将进程恢复成之前用setjmp保存时的状态

信号

- 等待信号

错误方法：

```
while (!pid)
    pause();
```

正确但不好的方法：

```
while (!pid)
    ;
```

```
while (!pid)
    sleep(1);
```

正确方法：sigsuspend

sigsuspend 函数等价于下述代码的原子的(不可中断的)版本：

```
1       sigprocmask(SIG_SETMASK, &mask, &prev);
2       pause();
3       sigprocmask(SIG_SETMASK, &prev, NULL);
```

例

判断下列说法正确性:

1. SIGTSTP信号既不能被捕获，也不能被忽略
2. 存在信号的默认处理行为是进程停止直到被SIGCONT信号重启
3. 系统调用不能被中断，因为那是操作系统的工作
4. 在任何时刻，一种类型至多只会有一个待处理信号
5. 信号既可以发送给一个进程，也可以发送给一个进程组
6. SIGTERM和SIGKILL信号既不能被捕获，也不能被忽略
7. 当进程在前台运行时键入Ctrl-C，内核就会发一个SIGINT信号给这个前台进程
8. 子进程能给父进程发送信号，但不能发送给兄弟进程

例

判断下列说法正确性:

1. SIGTSTP信号既不能被捕获，也不能被忽略
2. 存在信号的默认处理行为是进程停止直到被SIGCONT信号重启
3. 系统调用不能被中断，因为那是操作系统的工作
4. 在任何时刻，一种类型至多只会有一个待处理信号
5. 信号既可以发送给一个进程，也可以发送给一个进程组
6. SIGTERM和SIGKILL信号既不能被捕获，也不能被忽略
7. 当进程在前台运行时键入Ctrl-C，内核就会发一个SIGINT信号给这个前台进程
8. 子进程能给父进程发送信号，但不能发送给兄弟进程

FTFT TFTF

例

- 下面关于非局部跳转的描述，正确的是

A. setjmp可以和siglongjmp使用同一个jmp_buf变量

B. setjmp必须放在main()函数中调用

C. 虽然 longjmp 通常不会出错，但仍然需要对其返回值进行出错判断

D. 在同一个函数中既可以出现setjmp，也可以出现longjmp

例

- 下面关于非局部跳转的描述，正确的是

A. setjmp可以和siglongjmp使用同一个jmp_buf变量

B. setjmp必须放在main()函数中调用

C. 虽然 longjmp 通常不会出错，但仍然需要对其返回值进行出错判断

D. 在同一个函数中既可以出现setjmp，也可以出现longjmp

D

系统级I/O

- 文件
  - 普通文件: 包含任意数据
    - 文本文件: 仅包含ascii码
    - 二进制文件: 其他文件(可能包含非ascii码)
  - 目录: 包含将文件名映射到文件（普通文件/目录）的链接
    - 注意区分: 绝对路径 / 相对路径
  - 套接字etc

系统级I/O

- 在进程中打开/关闭文件
  - open函数, 成功返回正整数的文件描述符, 失败返回-1

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

  - flag和mode的含义

掩码	描述
S_IRUSR	使用者（拥有者）能够读这个文件
S_IWUSR	使用者（拥有者）能够写这个文件
S_IXUSR	使用者（拥有者）能够执行这个文件
S_IRGRP	拥有者所在组的成员能够读这个文件
S_IWGRP	拥有者所在组的成员能够写这个文件
S_IXGRP	拥有者所在组的成员能够执行这个文件
S_IROTH	其他人（任何人）能够读这个文件
S_IWOTH	其他人（任何人）能够写这个文件
S_IXOTH	其他人（任何人）能够执行这个文件

- O_RDONLY：只读。
- O_WRONLY：只写。
- O_RDWR：可读可写。
- O_CREAT：如果文件不存在，就创建它的一个截断的(truncated)(空)文件。
- O_TRUNC：如果文件已经存在，就截断它。
- O_APPEND：在每次写操作前，设置文件位置到文件的结尾处。

系统级I/O

- 读写文件: read和write

```
ssize_t read(int fd, void *buf, size_t n);
                返回：若成功则为读的字节数，若 EOF 则为 0，若出错为－1。
ssize_t write(int fd, const void *buf, size_t n);
                返回：若成功则为写的字节数，若出错则为－1。
```

- 不足值
  - 读时遇到EOF
  - 从终端读文本行
  - 读写网络套接字 (11章)
- 其他相关函数: stat/fstat读取文件元数据, dir系列函数读取目录相关内容

RIO

- RIO: robust I/O
- 无缓冲的RIO / 带缓冲的RIO (缓冲区可以减少陷入内核的次数, 从而减小开销)
  - 注意无缓冲/带缓冲的RIO不能混用
  - 例: rio_readlineb和rio_readnb可以混用，但不能和rio_readn混用

```
#include "csapp.h"

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

返回：若成功则为传送的字节数，若 EOF 则为 0(只对 rio_readn 而言)，若出错则为 −1。

```
#include "csapp.h"

void rio_readinitb(rio_t *rp, int fd);
```

返回：无。

```
ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n);
```

返回：若成功则为读的字节数，若 EOF 则为 0，若出错则为 −1。

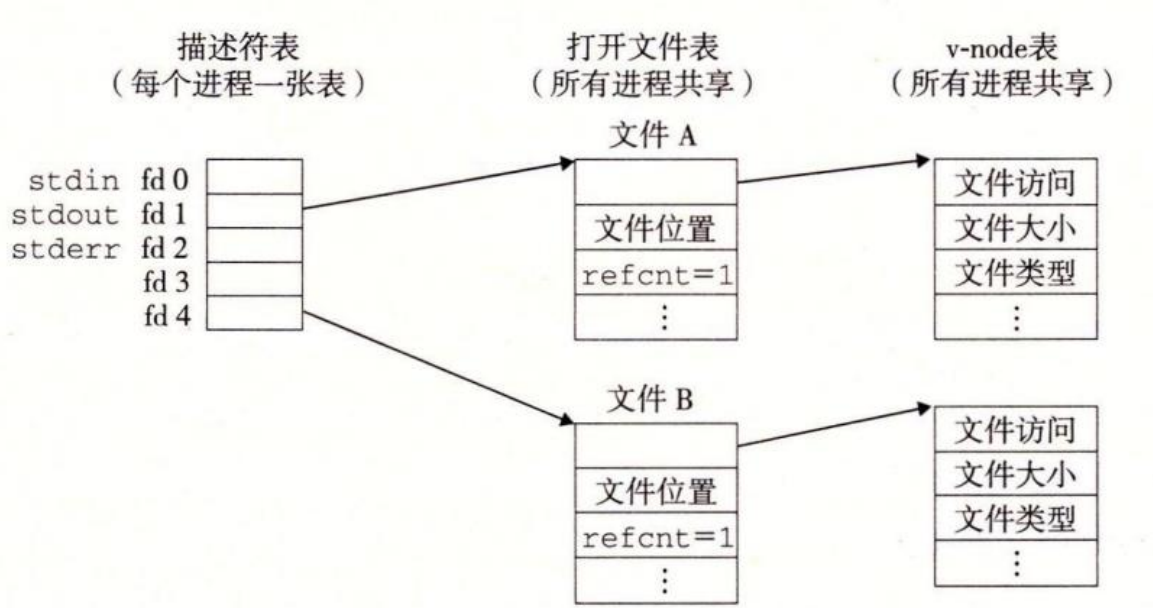
RIO

- 标准I/O: 以FILE*指针作为参数, 而非文件描述符
  - 标准I/O是有缓冲区的, 可以减小内核调用的开销
  - 类型为FILE的流是对文件描述符和缓冲区的抽象

```
#include <stdio.h>
extern FILE *stdin;    /* Standard input (descriptor 0) */
extern FILE *stdout;   /* Standard output (descriptor 1) */
extern FILE *stderr;   /* Standard error (descriptor 2) */
```

共享文件

- 文件描述符表 / 文件表 / V-node表
  - 注意: 对同一个文件多次open? 使用dup/dup2进行IO重定向?



例

1．假设缓冲区足够大，且 stdout 只有在关闭文件、换行与 fflush 的情况下才会刷新缓冲区。程序运行过程中的所有系统调用均成功。

(1)

```
int main() {
    printf("a");
    fork();
    printf("b");
    fork();
    printf("c");
    return 0;
}
```

(2)

```
int main() {
    write(1, "a", 1);
    fork();
    write(1, "b", 1);
    fork();
    write(1, "c", 1);
    return 0;
}
```

(3)

```
int main() {
    printf("a");
    fork();
    write(1, "b", 1);
    fork();
    write(1, "c", 1);
    return 0;
}
```

对于(1)号程序，写出它的一个可能的输出：abcabcabcabc 这个可能的输出是唯一的吗？是。

对于(2)号程序，它的输出中包含1个a，2个b，4个c。输出的第一个字符一定是a。

对于(3)号程序，它的输出中包含4个a，2个b，4个c。输出的第一个字符一定是b。

Thank you!