Linux系统编程-目录

2024年5月18日 11:16

基本概念

2024年5月18日　　11:17

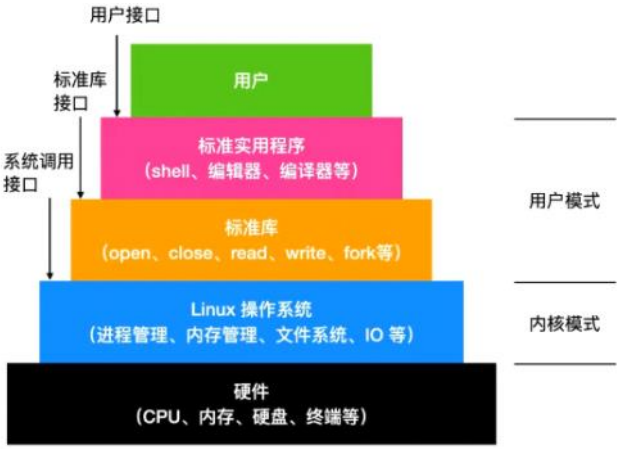
一、 基本概念

1.1 操作系统

操作系统（operating system,OS）：是管理硬件资源和软件资源的计算机程序。

操作系统通俗解释：计算机是由各种硬件组成，如果没有操作系统，开发人员就需要熟悉对所有计算机硬件的使用，掌握所有硬件的细节，这样程序员就不用写代码了。所以就给计算机安装了一个软件，称为操作系统，能为用户程序提供一个更简单，方便的计算机模型。本质上操作系统也是一个程序，最大的作用就是管理硬件资源。

其中管理硬件资源的程序叫做内核。

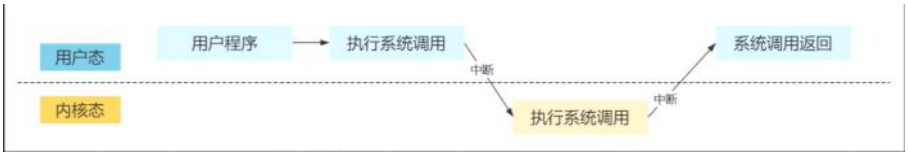


因此，操作系统一般分为两种模式：用户模式（用户态）、内核模式（内核态）

一般情况下，内核态可以访问硬件资源，用户态不能访问硬件资源。

例如：printf()函数在用户态下是无法打印输出的，在函数内部进行了系统调用write()函数，进入了内核态

可以通过系统调用进行内核态。



流程：用户利用程序（开发工具）调用库函数（printf），库函数内部进行系统调用，然后进入内核态，内核态下执行系统调用，返回系统调用，回到用户态。

1.2 库函数和系统调用

本质上都是函数

库函数：是把函数放到库里，方便其他人使用。目的就是把常用的一些函数放到一个文件了，方便重复使用。一般放在

lib文件中。

系统调用：是内核的入口，是为了能够进入内核的一个函数。如果需要进入内核空间，就需要通过系统调用，当程序进行系统调用时，会产生一个中断，CPU会中断当前执行的应用程序，跳转到中断处理程序，也就是开始执行内核程序，内核处理完成后，主动触发中断，把CPU执行权交回应用程序。

使用man手册查看系统调用

补全man手册

```
sudo apt-get install manpages-dev glibc-doc manpages-posixmanpages-posix-dev
```

1.3 内核

内核是操作系统的核心，是应用程序与硬件直接的桥梁，应用程序只关心与内核的交互，不用关系硬件的细节

内核的能力：

（1）进程调度：管理进程、线程，决定哪个进程、线程使用CPU
（2）内存管理：决定内存的分配和回收
（3）提供文件系统：提供文件系统，允许对文件进行创建、获取、更新以及删除等操作。
（4）管理硬件设备：为进程与硬件设备直接提供通信能力
（5）提供系统调用接口：进程可以通过内核入口（也就是系统调用），请求去内核执行各种任务。

内核具有很高的权限，可以控制CUP、内存、硬盘等硬件。而应用程序权限很小，因此大多数操作系统一般将内存分为两个区域：

内核空间：只有内核程序才能访问
用户空间：专门给应用程序使用

因此CUP可以在两种状态下运行：用户态、内核态，

用户态下的CPU，只能访问用户空间的内存
内核态下，内核空间内存与用户空间内存都能访问

1.4 程序和进程

程序：所有的程序都必须能运行。使用的软件就是程序，或者是自己写的代码经过编译和链接处理，得到计算机能够理解和执行的指令（可执行文件）。

进程：进程是正在执行程序的实例。是操作系统进行资源分配和调度的基本单位。在内核看来，进程是一个个实体，内核需要在他们之间共享各种计算机资源。例如内存资源：内核会在程序开始时为进程分配一定的内存空间，并统筹该进程和 整个系统对内存的需求。程序终止时，内核会是否该进程的所有资源，以供其他进程使用。

程序运行起来之后，就叫做进程（运行起来的的程序）。

1.5 虚拟内存

内核是通过虚拟内存来确保进程只访问自己的内存空间的。

进程中，只能访问虚拟内存，操作系统会把虚拟内存翻译为真实的内存地址。这种内存管理方式叫做虚拟内存。

操作系统会给每个进程分配一套独立的虚拟地址。每个进程访问自己的虚拟内存，互不干涉，操作系统会提供虚拟内存和物理地址的映射机制。

段式分配内存：外部内存碎片
页式分配内存：内部内存碎片

1.6 并发与并行

假设电脑里只有一个CPU、并且只有一个核心（core）

并行：是同时进行的。

并发：不是同时进行的，虽然在一个时间段内是一起完成的，但在某一时刻只能有一个进程在执行。
微观上：CPU在进程上来回切换（进程1在阻塞时，就会去执行进程2）
宏观上：多个进程都在运行。

文件IO

2024年5月27日　　15:42

二、 文件IO

2.1 文件描述符

打开的程序，使用进程来描述；而打开的文件，使用文件描述符来描述。（非负整数，从0开始，依次递增）

但一个程序打开之后，默认就有三个文件描述符（0：标准输入、1：标准输出、2：标准错误）

所以，在程序中打开的第一个文件的文件描述符应该是3。

2.2 打开关闭文件（open和close函数）

1、open函数

使用man 2 open 查看 open函数的定义。

```
SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

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

pathname：要打开的文件路径

flags:文件访问模式，有一系列常数值可供选择，可同时选择多个，用按位或(|)连接起来。
必选项:以下三个常数中必须指定一个，且仅允许指定一个
O_RDONLY:只读打开
O_WRONLY:只写打开
O_RDWR:可读可写打开
常用可选项:可以同时指定0个或多个，和必选项按位或起来使用。
O_APPEND:追加，如果文件已有内容，这次打开文件所写的数据附加到文件的末尾而不覆盖原来的内容，
O_CREAT:若此文件不存在则创建它。使用此选项时需要提供第三个参数mode，表示该文件的访问权限。注:文件最终权限:mode&~umask
O_EXCL:如果同时指定了O_CREAT，并且文件已存在，则出错返回，

O_TRUNC:如果文件已存在，将其长度截断为为0字节，
O_NONBLOCK:设置非阻塞模式:
普通文件默认是非阻塞的，内核缓冲区保证了普通文件 I/O 不会阻塞。打开普通文件一般会忽略该参数
设备、管道和套接字默认是阻塞的，以O_NONBLOCK方式打开可以做非阻塞I/O(Nonblockl/O)。

mode：创建文件时设置权限

函数返回值:
成功:返回一个最小且未被占用的文件描述符
失败:返回 -1，并设置 errno 值，

文件描述符指的是打开的文件，而不是文件,文件描述符是由进程维护的。

```
int fdc = open("./c.txt",O_RDONLY | O_CREAT,0644);
```

创建打开c.txt文件，不存在就创建一个，权限为0644（8进制），给的是 用户、用户组、其他人的权限

```
int fdc = open("./c.txt",O_RDONLY | O_CREAT,0666);
```
无法给到其他人读写权限，系统有默认权限，其他人给不到写权限。系统会给一个权限掩码，防止给太高的权限

使用umask即可查看权限掩码
```
weihong@weihong:~/linux/Day_517$ umask
0002
```

系统会将其二进制 0000 0010 取反 1111 1101 与 你给的 权限进行 按位与 &
0000 0110
1111 1101
=0000 0100

2、close函数

```
// close 函数 -- 关闭文件
// int close(int fd);
//
// fd 文件描述符
//
// 函数返回值:
//    成功返回 0
//    失败返回 -1,并设置 errno 值。
//
```

不使用close函数，在进程结束之后也会关闭文件，因为文件描述符是由进程维护的，进程关了，文件也就关了

2.3 文件读写（read和write函数）

1、读文件-- read函数

```
ssize_t read(int fd, void *buf, size_t count);
```

函数参数:
fd: 文件描述符
buf: 读取的数据保存在缓冲区 buf 中 -- 传出参数
count: buf 缓冲区存放的最大字节数
函数返回值:
>0: 读取到的字节数
=0: 文件读取完毕
-1: 出错，并设置 errno

count 应该是 大于等于 ssize_t

2、写文件-- write函数

```
ssize_t write(int fd, const void *buf, size_t count);
```

函数参数:

fd: 文件描述符
buf: 缓冲区，要写入文件或设备的数据
count: buf中数据的长度

函数返回值:

成功: 返回写入的字节数
错误: 返回-1 并设置 errno

ssize_t 等于 count

相同的文件描述符就是追加，不同的文件描述符就是覆盖

2.4 lseek（文件偏移量）

所有打开的文件都有一个当前文件偏移量(currentfile ofset)，也叫读写偏移量和指针文件偏移量通常是一个非负整数，用于表明文件开始处到文件当前位置的字节数(下一个read()或write()操作的文件起始位置)。文件的第一个字节的偏移量为 0。文件打开时，会将文件偏移量设置为指向文件开始(使用O_APPEND 除外)，以后每次read()和write()会自动对其调整，以指向已读或已写数据的下一字节。因此连续的read()和write()将按顺序递进，对文件进行操作。使用 lseek 函数可以改变文件的偏移量。

```
off_t lseek(int fd, off_t offset, int whence);
```

函数参数:

fd:文件描述符
offset:字节数，以 whence 参数为基点解释 offset
whence:解释 offset 参数的基点
SEEK SET:文件偏移量设置为 offset
SEEK CUR:文件偏移量设置为当前文件偏移量加上offset，offset可以为负数
SEEK END:文件偏移量设置为文件长度加上 offset，offset可以为负数

函数返回值:

若lseek 成功执行，则返回新的偏移量。
失败返回 -1 并设置 errno

Iseek 函数常用操作:

文件指针移动到头部

```
lseek(fd, 0, SEEK_SET);
```

获取文件指针当前位置

```
int len = lseek(fd,0,SEEK CUR);
```

获取文件长度

```
int len = lseek(fd,0,SEEK END);
```

lseek 实现文件拓展

```
lseek(fd,n,SEEK_END);// 扩展n个字节
write();// 扩展后需要执行一次写操作才能扩展成功
```

练习--实现自己的cp函数 -- mycp

```c
int main(int argc, char* argv[])
{
    if(argc!=3)
    {
        printf("请输入正确的参数数量: 源文件、目标文件\n");
    }
    // 打开两个文件
    int fd1 = open(argv[1],O_RDONLY);
    // 写入、创建、清空(截断)
    int fd2 = open(argv[2],O_WRONLY|O_CREAT|O_TRUNC,0644);

    // 读取第一个文件的内容
    char buf[10];
    while(1) // 循环读取
    {
        int read_count = read(fd1,buf,sizeof(buf));
        if(read_count == 0)
        {
            break;
        }
        // 将读取的内容写入第二个文件
        int write_count = write(fd2,buf,read_count);
        printf("write_count = %d\n",write_count); // 打印写入字节
    }

    // 关闭两个文件
    close(fd1);
    close(fd2);
    return 0;
}
```

2.5 阻塞、非阻塞

对于进程

进程挂起（切换其他进程执行）的几种情况：

（1）系统给定的时间片用完（时间片轮转算法），

（2）遇到阻塞事件：例如：读取磁盘。

阻塞、非阻塞，强调的是进程或线程的状态（等待中或者执行中）

而对于文件

这里的阻塞、非阻塞是只文件的，而不是进程。该项设置是决定了进程在读写文件的时候，是否要等它（）

O_NONBLOCK:设置非阻塞模式:

（1）普通文件：默认是非阻塞的，内核缓冲区保证了普通文件 I/O 不会阻塞。打开普通文件一般会忽略该参数

（就像等红绿灯，其他路口有是红灯，这儿都没有红绿灯）

（2）设备、管道和套接字：默认是阻塞的，以O_NONBLOCK方式打开可以做非阻塞I/O(Nonblockl/O)。

设备：标准输入输出（终端设备，文件描述符0、1）

①. 比如使用scanf进行输入，库函数内部进行系统调用就是调用了 read(0) 标准输入文件描述符，而他是设置阻塞的，因此在未输入之前是一直等待的。

②. 而使用printf进行输出。就是在库函数内部进行系统调用，write(1) 标准输出文件描述符，设置了非阻塞

管道：用于进程间的通信。

套接字：用于网络间的通信。

```c
// 查看 0 1 文件描述符的默认阻塞状态
int main(int argc, char* argv[])
```

```
{
    char buf[1024];
    // 通过标准输入的文件描述符 0 ，在终端读取数据（类似与scanf）
    // 阻塞的
    int read_count = read(0,buf,sizeof(buf));
    printf("read_count = %d\n",read_count);
    // 通过标准输出的文件描述符 1 ，向终端写出数据（类似与printf）
    // 非阻塞的
    int wr_count = write(1,buf,read_count);
    printf("wr_count = %d\n",wr_count);
    return 0;
}
```

printf不会直接输出到终端，会等到缓冲区刷新才会进行输出，而'\n' 会刷新缓冲区。

```
// 将标准输入设置为非阻塞的
int main(int argc, char* argv[])
{
    // 再次打开 标准输入文件，并将其设置为非阻塞的。fd = 3
    int fd = open("/dev/fd/0",O_RDWR | O_NONBLOCK);
    char buf[1024];
    // 设置为非阻塞的
    int read_count = read(fd,buf,sizeof(buf));
    printf("read_count = %d\n",read_count); // 非阻塞，不会等待，读取失败。
    // 通过标准输出的文件描述符 1 ，向终端写出数据（类似与printf）
    // 非阻塞的
    int wr_count = (lwrite,buf,read_count);
    printf("wr_count = %d\n",wr_count);
    return 0;
}
```

2.6 fcntl （修改以打开文件的状态标志）

通过文件描述符，获取或修改已经打开文件的状态标志。

```
int fcntl(int fd, int cmd, ... /* arg */ );
```

函数参数:

fd:要控制的文件描述符

cmd:不同值对应不同的操作

- cmd为 F_GETFL: 获取文件描述符的 flag 值
- cmd为 F_SETFL: 设置文件描述符的 flag 值
- cmd为 F_DUPFD: 复制文件描述符，与dup函数功能相同

函数返回值:返回值取决于 cmd

成功:

- 若 cmd 为 F_DUPFD，返回一个新的文件描述符
- 若 cmd 为 F_GETFL，返回文件描述符的 flags 值
- 若 cmd 为 F_SETFL，返回0

失败: 返回 -1，并设置 errno 值

获取当前打开文件的状态，以及是否拥有某一flags

```
int main(int argc, char* argv[])
{
    int fd = open("./a.txt",O_RDWR|O_CREAT|O_NONBLOCK,0644);
    printf("fd = %d\n",fd);
    // 返回fd文件描述符所打开文件的状态标志
    int flags = fcntl(fd,F_GETFL);
    // 判断状态中是否有某一标志
    // 将得到的状态标志 & O_NONBLOCK 如果有该状态，就会进入判断
    if(flags & O_NONBLOCK)
    {
        printf("a.txt is NONBLOCK\n");
    }
```

```
    printf("flags = %d\n",flags);
    return 0;
}
```

判断当前文件的访问模式。

(flags & O_ACCMODE) == O_RDWR
因为 O_RDONLY 的值为0，O_WRONLY 的值为1，O_RDWR的值为2，只占两位
而O_ACCMODE的值为3，二进制下为11，
flags & O_ACCMODE 保留了后两位的数值，得到的结果就是 访问模式。

因此 O_WRONLY、 O_RDWR使用上面的直接&也可以。
& 的使用场景就是，用来与11111进行按位与来获得某几位上的数据

怎么用一个变量来表示多个含义。（16位的数，前八位和后八位，表示两个数，用 & 获得前八位或后八位）

```
int main(int argc, char* argv[])
{
    int fd = open("./a.txt",O_RDWR|O_CREAT|O_NONBLOCK,0644);
    printf("fd = %d\n",fd);
    // 返回fd文件描述符所打开文件的状态标志
    int flags = fcntl(fd,F_GETFL);
    // 判断状态访问模式
    if((flags & O_ACCMODE) == O_RDWR)
    {
        printf("a.txt is O_RDWR\n");
    }
    else{
         printf("a.txt is not O_RDWR\n");
    }
    printf("flags = %d\n",flags);
    return 0;
}
```

设置文件状态标志--SETFL

```
// 设置文件状态标志 SETFL
int main(int argc, char* argv[])
{
    int fd = open("./a.txt",O_RDWR);
    printf("fd = %d\n",fd);
    int flags = fcntl(fd,F_GETFL);
    printf("flags = %d\n",flags);
    flags = flags | O_NONBLOCK; // 在之前的基础上添加某个属性
    printf("flags = %d\n",flags);

    flags = fcntl(fd,F_SETFL,flags);
    printf("flags = %d\n",flags);
    // 判断状态访问模式
    if(flags & O_NONBLOCK)
    {
        printf("a.txt is O_NONBLOCK\n");
    }
    else{
         printf("a.txt is not O_NONBLOCK\n");
    }
    printf("flags = %d\n",flags);
    return 0;
}
```

2.7 复制文件描述符（dup、dup2）

一个文件描述符对应一个打开的文件，多个文件描述符也可以对应同一个文件（同一个文件被打开多次）

两个不同的文件标识符打开同一个文件，其文件偏移量是不一样的。一个读一个写，此时是覆盖

两个相同的文件标识符操作同一个打开的文件，文件偏移量相同，一个读一个写，此时是追加。

而复制文件描述符，能将文件偏移量一同复制。对一个文件描述符进行读写以后，另一个文件描述符的文件偏移量也会修改。

int dup(int oldfd);

函数参数:

oldfd: 要复制的文件描述符

函数返回值:

成功: 返回最小且没被占用的文件描述符

失败: 返回-1，设置 errno 值

```
// dup
int main(int argc, char* argv[])
{
    int fd = open("./a.txt",O_RDWR|O_CREAT,0644);
    printf("fd = %d\n",fd);
    int fd2 = dup(fd);
    printf("fd2 = %d\n",fd2);
    char buf[1024];
    int read_count = read(fd,buf,sizeof(buf));
    printf("read_count = %d, buf = %s\n",read_count,buf);
    int wr_count = write(fd2,buf,read_count);
    printf("wr_count = %d, buf = %s\n",wr_count,buf); //  追加的
    close(fd);
    close(fd2);
    ret
```

int dup2(int oldfd, int newfd);

可以自己指定复制后的文件描述符的值，这样就导致一个问题，如果指定的文件描述符已经存在了，那么它会和之前的文件断开，之前的那个打开的文件自动关闭了（没有指向该文件的文件描述符时，就会关闭）。

2.8 文件报错（perror、strerror）

在每一个文件操作之后，都应该进行文件报错处理，用来保证文件打开或者操作失败之后，能够确定是那里出问题。而不是一直运行，直到程序崩溃。

```
// 文件报错
int fdb = open("bbb.txt",O_RDWR);
if(fdb ==-1)
{
    // 打印errno值对应的报错信息
    perror("打开文件失败: ");
    // 将报错码转换为相应报错信息字符串
    printf("文件打开失败: %s\n", strerror(errno));

}
```

当open函数打开失败时，会返回 -1 并设置errno的值。而errno是一个int类型的值，我们可以使用perrno将其转换为报错信息。

2.9 获取文件状态（stat、lstat、fstat）

获取文件属性，Istat和 stat的区别在于如果文件是符号链接，返回的文件属性是符号链接本身。fstat 则是指定文件描述符获取文件信息。

头文件:

#include <sys/stat.h>

函数原型:

```
int stat(const char *pathname, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
```

函数参数:

pthname:要获取属性的文件路径

statbuf:传出参数，指向 struct stat 结构体的指针，用于存储获取到的文件信息

fd:要获取属性的文件描述符

函数返回值:

成功返回 0

失败返回 -1

```
struct stat {
    dev_t     st_dev;       //文件所在的设备号
    ino_t     st_ino;       //inode 号
    mode_t    st_mode;      //文件类型及权限
    nlink_t   st_nlink;     //硬链接计数
    uid_t     st_uid;       //拥有者
    gid_t     st_gid;       //所属用户组
    dev_t     st_rdev;      //如果是设备文件时有效，为设备号
    off_t     st_size;      //文件的字节数
    blksize_t st_blksize;   //文件系统中 block 的大小
    blkcnt_t  st_blocks;    //文件所占扇区数量
    time_t st_atime;        //最后访问时间
    time_t st_mtime;        //最后修改时间
    time_t st_ctime;        //最后改变状态时间
};
```

st_mode 为 16 位整数，低 12 位代表文件权限，只需要了解低9位。高4位代表文件类型
如下列出用以检查权限和文件类型的宏。

检测文件权限的宏

- 检查文件权限的宏:

st_mode 的位数	宏	权限
低 1-3 位	S_IXOTH	其他人执行权限
	S_IWOTH	其他人写权限
	S_IROTH	其他人读权限
	S_IRWXO	其他人读写执行权限
低 4-6 位	S_IXGRP	所属组执行权限
	S_IWGRP	所属组写权限
	S_IRGRP	所属组读权限
	S_IRWXG	所属组读写执行权限
低 7-9 位	S_IXUSR	拥有者执行权限
	S_IWUSR	拥有者写权限
	S_IRUSR	拥有者读权限
	S_IRWXU	拥有者读写执行权限

检查文件类型的宏：(S_IFMT:掩码，过滤st_mode 权限部分，保留文件类型部分)

- 检查文件类型的宏：

  （S_IFMT：掩码，过滤 st_mode 权限部分，保留文件类型部分）

常量	测试宏	文件类型
S_IFREG	S_ISREG()	普通文件
S_IFDIR	S_ISDIR()	目录
S_IFCHR	S_ISCHR()	字符设备
S_IFBLK	S_ISBLK()	块设备
S_IFIFO	S_ISFIFO()	管道
S_IFSOCK	S_ISSOCK()	套接字
S_IFLNK	S_ISLNK()	软链接

```c
#include<fcntl.h> // 读取文件
#include <sys/stat.h> // 读取文件属性
int main(int argc, char* argv[])
{
    // 获取文件属性
    struct stat file_info;
    // 将文件属性读取到结构体 file_info中
    stat("./a.txt", &file_info);
    printf("st_ino = %ld\n",file_info.st_ino);  // inode编号，通过ls -il查看
    printf("st_mode = %d\n",file_info.st_mode);  // 文件类型以及权限
    printf("st_nlink = %ld\n",file_info.st_nlink); // 硬链接计数
    printf("st_uid = %d\n",file_info.st_uid); // 所属用户id
    printf("st_gid = %d\n",file_info.st_gid); // 所属组id
    printf("st_size = %ld\n",file_info.st_size); // 文件大小
    // 文件最后被修改的时间，结构体类型，需要使用ctime将时间结构体转为字符串
    printf("st_mtim = %s\n", ctime(&file_info.st_mtime));
    // 文件类型以及权限是10进制，需要通过宏定义转换
    if(file_info.st_mode & S_IRUSR)
    {
        printf("文件所有者读权限\n");
    }
    if((file_info.st_mode & S_IFMT) == S_IFREG)
    {
        printf("该文件是普通文件\n");
    }
    if(S_ISREG(file_info.st_mode))
    {
        printf("该文件是普通文件\n");
    }
    return 0;
}
```

文件I/O练习

```c
// 练习
/*
1.创建 mc.txt 文件并打开 /etc/passwd 文件
2.后面的操作使用值为3的文件描述符操作/etc/passwd 文件，使用值为7的文件描述符操作 mc.txt 文件
3.将 /etc/passwd 的文件内容从第 10 个字节开始全部复制到 mc.txt中
4.判断 /etc/passwd 文件其他人是否有该文件的读权限
5.判断 mc.txt 文件是否是非阻塞的
所有的系统调用使用后，判断是否报错并打印报错信息
*/
int main(int argc, char* argv[])
{
    int fdmc = open("./mc.txt",O_RDWR|O_CREAT,00644);
    if(fdmc == -1)
    {
        perror("mc文件打开失败");
    }
    int fdp = open("/etc/passwd",O_RDONLY);
    if(fdp == -1)
    {
```

```
        perror("passwd文件打开失败");
    }
    printf("fdmc = %d\n",fdmc);
    printf("fdp = %d\n",fdp);
    int new_fdmc = dup2(fdmc,7);
    if (new_fdmc == -1)
    {
        perror("指定mc文件描述符失败");
    }
    printf("new_fdmc = %d\n",new_fdmc);
    int new_fdp = dup2(fdp,fdmc);
    if (new_fdp == -1)
    {
        perror("指定passwd文件描述符失败");
    }
    printf("new_fdp = %d\n",new_fdp);
    lseek(new_fdp,10,SEEK_SET);
    int len = lseek(new_fdp,10,SEEK_END);
    char buf[len];
    read(new_fdp,buf,sizeof(buf));
    write(new_fdmc,buf,len);

    // 获取文件属性
    struct stat file_info;
    // 将文件属性读取到结构体 file_info中
    stat("/etc/passwd", &file_info);
    if(file_info.st_mode & S_IROTH)
    {
        printf("其他人读权限读权限\n");
    }
    int flags= fcntl(new_fdmc,F_GETFL);
    if(flags & O_NONBLOCK)
    {
        printf("非阻塞\n");
    }else{
        printf("阻塞\n");
    }
    return 0;
}
```

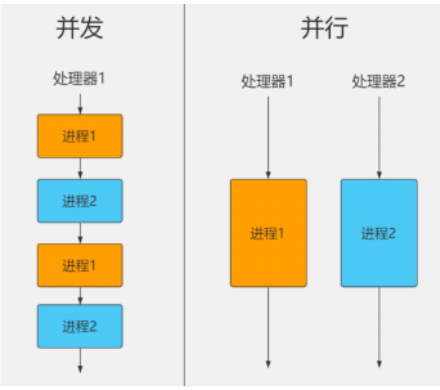
进程

2024年5月29日　　18:30

三、进程

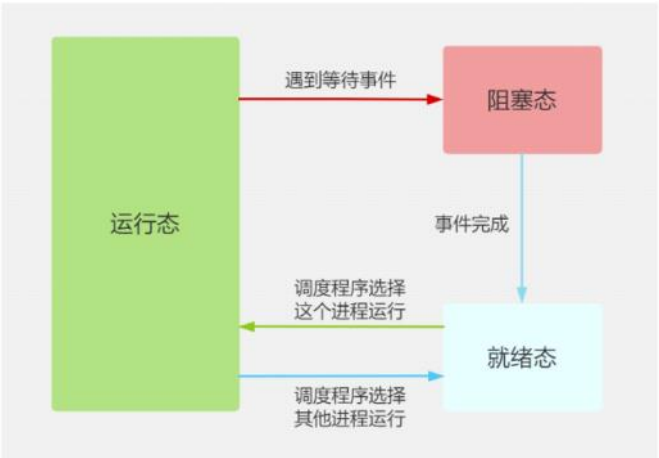
3.1 进程状态

进程就是一个运行程序的实例，是操作系统进行资源分配和调度的基本单位。



当一个进程开始运行时，它可能会经历下面这几种状态：

（1）运行态：运行态指的就是进程实际占用 CPU 时间片运行时
（2）就绪态：就绪态指的是可运行，但因为其他进程正在运行而处于就绪状态
（3）阻塞态：该进程正在等待某一事件发生(如等待输入/输出操作的完成)而暂时停止运行，这时即使给它 CPU 控制权，它也无法运行



状态切换：

（1）运行态到阻塞态：当进程遇到某个事件需要等待时会进入阻塞态。

（2）阻塞态到就绪态：当进程要等待的事件完成时会从阻塞态变到就绪态。
（3）就绪态到运行态：处于就绪态的进程被操作系统的进程调度程序选中后，就分配CPU开始运行。
（4）运行态到就绪态：进程运行过程中，分配给它的时间片用完后，操作系统会将其变为就绪态，接着从就绪态的进程中选择一个运行。

程序调度指的是，决定哪个进程优先被运行和运行多久，这是很重要的一点。已经设计出许多算法来尝试平衡系统整体效率与各个流程之间的竞争需求。

3.2 创建进程

创建进程的方式：

系统初始化:启动操作系统时会创建若干个进程
用户请求创建:例如双击图标启动程序
系统调用创建:一个运行的进程可以发出系统调用创建新的进程帮助其完成工作

fork 函数：

函数描述：

创建进程，新创建的是当前进程的子进程

函数原型：

```
pid t fork(void);
```

函数返回值：

成功:父进程:返回新创建的子进程 ID
子进程:返回 0
失败返回 -1，子进程不被创建

```c
int main(int argc, char* argv[])
{
    // 进程创建 fork，返回一个 进程id，
    int a = 0;
    printf("world\n");
    int pid = fork(); // 创建一个新的进程，新进程从此处开始运行
    printf("hello\n");
    return 0;
}
```

```c
int main(int argc, char* argv[])
{
    // 进程创建 fork，返回一个 进程id，
    int a = 0;
    printf("world\n");
    char* str1 = (char*)malloc(100);
    strcpy(str1, "hello");
    int pid = fork(); // 创建一个新的进程，新进程从此处开始运行
    if(pid == 0) // 子进程id返回值为0
    {
        // 子进程是父进程复制过来的，但内存空间是不同的，虽然显示的虚拟地址相同
        // 但实际上的物理地址是不同的
        // 内存不共享，但磁盘内的东西是可以共享的（文件描述符）
        strcpy(str1, "world");
        printf("子进程: %s,  %p\n",str1,&str1);
        // 获取当前进程id和父进程id
        printf("子进程: pid = %d , ppid = %d\n", getpid(),getppid());
    }
    if(pid > 0) // 父进程
    {
        sleep(1);
        printf("父进程: %s,  %p \n",str1,&str1);
        printf("父进程: pid = %d , ppid = %d\n", getpid(),getppid());
    }
    return 0;
}
```

```c
int main(int argc, char* argv[])
{
    printf("hello"); // 不加 \n 缓冲区就不会刷新，在子进程中也会复制缓冲区内容
    // 进程创建 fork，返回一个 进程id，
    int pid = fork(); // 创建一个新的进程，新进程从此处开始运行
    if(pid == 0) // 子进程id返回值为0
    {
        printf("world\n"); // 输出helloworld
    }
    if(pid > 0) // 父进程
    {
        printf("world\n"); // 输出helloworld
    }
    return 0;
}
```

```c
// 创建15个进程
int main(int argc, char* argv[])
{

    // 进程创建 fork，返回一个 进程id，
    int pid; // 创建一个新的进程，新进程从此处开始运行
    for(int i = 0; i < 15; i++)
    {
        pid = fork();
        if(pid == 0)
        {
            break; // 子进程直接跳出,只在父进程下创建
        }
    }
    printf("end %d\n");
    while(1);
    return 0;
}
```

```c
// 判断以下程序的打印次数
int main(int argc, char* argv[])
{
    // // 1
    // if(fork() && fork())
    // {
    //     fork();
    // }
    // printf("end \n"); // 打印4次
    // 2
    if(fork() || fork())
    {
        fork();
    }
    printf("end\n");
    // // 3
    // if(fork() || fork() && fork())
    // {
    //     fork();
    // }
    // printf("end %d\n");
    while(1);
    return 0;
}
```

3.3 进程终止

进程终止的方式:

正常退出:

从 main 函数返回

调用exit()或_exit()，exit()是库函数，_exit()是系统调用，程序一般不直接调用exit()，而是调用库函数exit()。

异常退出:

被信号终止

头文件:

#include <stdlib.h>

函数原型:

void exit(int status):

函数参数:

进程的退出状态，0表示正常退出，非0值表示因异常退出，保存在全局变量$?中，$?保存的是最近一次运行的进程的返回值，返回值有以下3种情况:

程序中的 main 函数运行结束，$?中保存 main 函数的返回值

程序运行中调用 exit 函数结束运行， $?中保存 exit 函数的参数

程序异常退出 $?中保异常出错的错误号

```c
int main(int argc, char* argv[])
{
    // 进程结束
    // 1、执行完毕，正常退出 返回0
    // 2、exit()
    // 异常退出
    // 信号，杀死进程
    // kill -9 进程id 强制杀死进程
    // exit() 是一个库函数、内部进行了系统调用 _exit()
    // void exit(int status); 状态,一般是0，表示正常退出

    printf("exit\n");
    exit(10); // echo $? 使用该命令查看最近一次进程的返回值
    printf("printf\n"); // 不执行，exit已经将进程结束了
    return 0;
}
```

return 0 和 exit()的区别: return 0 是让主函数结束

exit() 可以在任意位置使用，结束当前进程

3.4 孤儿进程

孤儿进程:**父进程先于子进程结束**，则子进程成为孤儿进程，变成孤儿进程后会有一个专门用于回收的 init 进程成为它的父进程，称 init 进程领养孤儿进程。 （init 一般是 1号进程）

```c
// 当子进程结束后，父进程负责回收子进程
// 子进程死循环，不会结束，而父进程在10秒后结束了，称其为孤儿进程
// 但是，父进程是用来管理子进程的资源的，因此要找一个新的父进程来管理。
// 变成孤儿进程后会有一个专门用于回收的 init 进程成为它的父进程，称 init 进程领养孤儿进程。(一般是1号进程)
int main(int argc, char* argv[])
{
    int pid = fork();
    if(pid == 0)
    {
        while(1) // 子进程死循环，不会结束，而父进程在10秒后结束了，称其为孤儿进程
        {
            printf("子进程: pid = %d ; ppid = %d\n",getpid(),getppid());
            sleep(1);
        }
    }
    sleep(10);
```

```
    printf("父进程:  pid = %d",getpid());
    return 0;
}
```

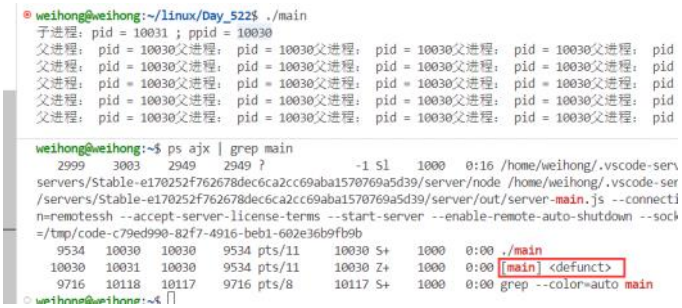
3.5 僵尸进程

僵尸进程: **子进程先终止，父进程尚未回收**，子进程残留资源(PCB)存放于内核中，变成僵尸(Zombie)进程。
特别注意，僵尸进程是不能使用 ki 命令清除掉的。因为 ki 命令只是用来终止进程的而僵尸进程已经终止。可以杀死它的父进程，让 init 进程变成它的父进程，init进程可以回收记。

```
// 僵尸进程
int main(int argc, char* argv[])
{
    int pid = fork();
    if(pid == 0) // 子进程先结束
    {
        sleep(1);
        printf("子进程: pid = %d ; ppid = %d\n",getpid(),getppid());
        return 0;
    }

    while(1) // 父进程不结束
    {
        sleep(1);
        printf("父进程:  pid = %d",getpid());
    }
    return 0;
}
```

使用 ps ajx | grep main 查看进程状态



Z+ 代表 进程的状态 zombie（僵尸进程），子进程已经结束了，但父进程未结束，而父进程负责在结束后回收子进程的资源，此时子进程还有残留的资源未被回收，就成了僵尸进程，而且此时的子进程无法使用kill信号回收，因为他已经结束了，只是残留的资源还在。

3.6 回收进程

僵尸进程是由子进程先结束，而父进程未结束，无法回收导致的，而使用wait函数就可以回收进程

wait 函数

函数描述:父进程调用 wait 函数可以回收子进程终止信息。该函数有三个功能:

1. 阻塞等待，子进程退出（父进程等待子进程结束）
2. 回收子进程残留资源
3. 获取子进程结束状态(退出原因)。

头文件:

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

函数原型:

```
pid t wait(int *status)
```

函数参数:

status 为传出参数，用以保存进程的退出状态

函数返回值:

成功:返回清理掉的子进程 ID:

失败:返回-1(没有子进程)

当进程终止时，操作系统的隐式回收机制:

1.关闭所有文件描述符释放用户空间、分配的内存。内核的 PCB 仍存在。其中保存该进程的退出状态。

2.(正常终止→退出值;异常终止一终止信号)

可使用 wait 函数传出参数 status 来保存进程的退出状态。借助宏函数来进一步判断进程终止的具体原因。宏函数可分为如下三组:

```
1.  WIFEXITED(status)     //为真 → 进程正常结束
    WEXITSTATUS(status)   //如上宏为真，获取进程退出状态 (exit 的参数)
2.  WIFSIGNALED(status)   //为真 → 进程异常终止
    WTERMSIG(status)      //如上宏为真，得使进程终止的那个信号的编号。
*3. WIFSTOPPED(status)    //为真 → 进程处于暂停状态
    WSTOPSIG(status)      //如上宏为真，取得使进程暂停的那个信号的编号。
    WIFCONTINUED(status)  //如上宏为真，进程暂停后已经继续运行
```

waitpid 函数

函数描述:

作用同 wait，但可指定进程 id 为 pid 的进程清理，可以不阻塞,

头文件:

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

函数原型:

```
pid t waitpid(pid t pid, int *status, int options);
```

函数参数:

参数 pid:

pid > 0 : 回收指定ID 的子进程

pid = -1 : 回收任意子进程(相当于 wait)

pid = 0 : 回收和当前调用进程(父进程)一个组的任一子进程

pid < -1 : 设置负的进程组ID，回收指定进程组内的任意子进程

函数返回值:

成功: 返回清理掉的子进程ID

失败: 返回 -1(无子进程)

参数3 为 WNOHANG，且子进程正在运行，返回0。

注意:一次 wait 或 waitpid 调用只能清理一个子进程，清理多个子进程应使用循环,

3.7 exec函数族

在一个程序中调用另一个程序

fork 创建子进程后执行的是和父进程相同的程序(但有可能执行不同的代码分支)，子进程往往要调用一种 exec 函数以执行另一个程序。当进程调用一种 exec 函数时，通过该调用进程能以全新程序来替换当前运行的程序。将当前进程的代码段和数据段替换为所要加载的程序的代码段和数据段，然后让进程从新的代码段第一条指令开始执行，但进程ID不变，换核不换壳。

execl 函数:

函数描述:

加载一个进程， 通过路径+程序名来加载。

函数原型:

```
int execl(const char *path, const char *arg, ...);
```

函数参数:

pathname:可执行文件的路径

arg:可执行程序的参数，对应main()函数的第二个参数(argv)，格式相同，以NULL结束

函数返回值:

成功:无返回

失败:-1

main.c

```c
// execl函数
int main(int argc, char* argv[])
{
    printf("main pid = %d\n",getpid());
    int ret = write(3,"hello\n",6);
    printf("ret = %d\n",ret);
    printf("argc = %d\n",argc);
    printf("argv[0] = %s\n",argv[0]); // 函数名
    printf("argv[1] = %s\n",argv[1]); //
    printf("argv[2] = %s\n",argv[2]);
    printf("argv[3] = %s\n",argv[3]);
    while(1);
    return 0;
}
```

main2.c

```c
int main(int argc, char* argv[])
{
    // 在main2中打开一个文件，在main中使用 文件描述符对文件进行写入
    // 查看资源是否共享---共享的，main中可以访问到main2中的变量
    int fd = open("a.txt",O_RDWR);

    // 会发现，main2的id与  main的id一样，而且 main2结束 未打印
    // 因为 execl函数会将main中的得内容复制到main2，本质上还是main2在执行，
    // 在main 中 return 0; main2 也结束了，不会执行到 打印main2结束 的语句
    //
    // 执行execl函数后，进程id不变，但是进程名字变了
    printf("main2 pid = %d\n",getpid());

    // 休眠10秒，使用 ps ajx | grep “main” 查看 进程变化
    sleep(10);
    // 函数路径、函数名、传入的参数（aaa，bbb）、null（参数结束标志）
    execl("./main","./main","aaa","bbb",NULL);
    printf("main2 结束\n");
    return 0;
}
```

执行后，进程名会改变

结论：

（1）执行excel函数后，进程id不变（还是main2的id），本质上还是main2在运行，不过是在执行main中的代码

（2）执行excel函数后，虽然进程id不变，但是进程名字会变

（3）执行excel函数后，进程使用的内存依旧是main2的内存，在main中也可以访问main2中的变量

execlp 函数:该函数通常用来调用系统程序。如:Is、date、cp、cat等命令

函数描述:
加载一个进程，借助 PATH 环境变量:
函数原型:
int execlp(const char *file, const char *arg, ...);
函数参数:
file:可执行文件的文件名，系统会在环境变量 PATH 的目录列表中寻找可执行文件
arg:可执行程序的参数
函数返回值:
成功:无返回
失败:-1
该函数通常用来调用系统程序。如:Is、date、cp、cat等命令

```
// execlp 函数
int main(int argc, char* argv[])
{
    // 执行 ls -l
    execlp("ls","ls","-l",NULL);
    return 0;
}
```

练习：写一个程序完成Is/>>a.txt 指令的功能

>> 输出重定向（追加）

> 是覆盖添加
>> 是追加

```
int main(int argc, char* argv[])
{
    // 打开a.txt
    int fd = open("./a.txt",O_RDWR);
    // 利用dup2 将标准输入的文件描述符改为 a.txt的文件描述符
    dup2(fd,1);
    // 利用execlp函数 执行 ls / 查看根目录下的文件
    execlp("ls","ls","/",NULL);
    // 此时，输出到终端的内容就输出到a.txt 中了
    return 0;
}
```

进程通信

2024年5月28日　　20:05

四、进程通信

进程间通信简称IPC(Inter process communication)，进程间通信就是在不同进程之间传播或交换信息。

由于各个运行进程之间具有独立性，这个独立性主要体现在数据层面，而代码逻辑层面可以私有也可以公有(例如父子进程)，因此各个进程之间要实现通信是非常困难的。

各个进程之间若想实现通信，一定要借助第三方资源，这些进程就可以通过向这个第三方资源写入或是读取数据，进而实现进程之间的通信，这个第三方资源实际上就是操作系统提供的一段内存区域。

进程间通信的目的：

数据传输：一个进程需要将它的数据发送给另一个进程，

资源共享：多个进程之间共享同样的资源，

通知事件：一个进程需要向另一个或一组进程发送消息，通知它(它们)发生了某种事件，比如进程终止时需要通知其父进程。(信号)

进程控制：有些进程希望完全控制另一个进程的执行(如 Debug 进程)，此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变

4.1 管道

管道 默认是阻塞的，普通文件默认不会阻塞。

4.1.1 匿名管道

匿名指的是文件名，通过pipe创建的管道文件，在磁盘中是没有实体的文件的。

匿名管道只能在有血缘关系的进程之间通讯（父子进程之间）

pipe 函数

函数描述:

创建匿名管道

函数原型:

```
int pipe(int pipefd[2]);
```

函数参数:

pipefd 是一个传出参数，用于返回两个指向管道读端和写端的文件描述符

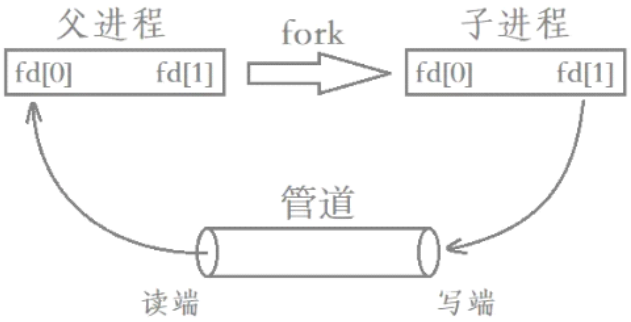
pipefd[0]:管道读端的文件描述符

pipefd[1]:管道写端的文件描述符

函数返回值:

成功返回 0

失败返回 -1，设置 errno



注意:

管道只能够进行单向通信，因此当父进程创建完子进程后，需要确认父子进程谁读谁写然后关闭相应的读写端。

从管道写端写入的数据会被存到内核缓冲，直到从管道的读端被读取。

（1）管道中没有数据：write 返回成功写入的字节数，读端进程阻塞在read 上
（2）管道中有数据没满：write 返回成功写入的字节数，read 返回读取的字节数
（3）管道已满：写端进程阻塞在 write 上，read 返回读取的字节数
读取一定大小的内容后，写端才会接着写

（4）写端全部关闭：read 正常读，返回读到的字节数(没有数据返回0，不阻塞)
（5）读端全部关闭：写端进程 write 会异常终止进程(被信号 SIGPIPE 杀死的)

练习

借助管道和 execlp 函数，实现ls/ |wc -l

```
// 练习：借助管道和 execlp 函数，实现ls / |wc -l
int main(int argc, char* argv[])
{
    // 创建匿名管道--0 读端、1 写端
    int pipefd[2];
    pipe(pipefd);

    // 创建子进程
    int pid = fork();
    if(pid == 0) // 子进程写入数据
    {
        close(pipefd[0]); // 关闭读端
        dup2(pipefd[1],1); // 将 标准输出 给到 管道写端
        execlp("ls","ls","/",NULL);
    }
    else // 父进程读取数据
    {
        close(pipefd[1]); // 关闭写端
        dup2(pipefd[0],0); // 将标准输入 给到 管道读端
        execlp("wc","wc","-l",NULL);
    }
    return 0;
}
```

命名管道

匿名管道只能用于具有共同祖先的进程(具有亲缘关系的进程)之间的通信，通常，一个管道由一个进程创建，然后该进程调用fork，此后父子进程之间就可应用该管道。如果要实现两个毫不相关进程之间的通信，可以使用命名管道来做到。命名管

道就是一种特殊类型的文件，两个进程通过命名管道的文件名打开同一个管道文件，此时这两个进程也就看到了同一份资源，进而就可以进行通信了。命名管道和匿名管道一样，都是内存文件，只不过命名管道在磁盘有一个简单的映像，但这个映像的大小永远为 0，因为命名管道和匿名管道都不会将通信数据刷新到磁盘当中。

线程

2024年6月1日　　21:26

五、线程

5.1 创建线程