

# Linu系统编程-目录

2024年5月18日 11:16

## 一、基本概念

- [1.1 操作系统](#)
- [1.2 库函数和系统调用](#)
- [1.3 内核](#)
- [1.4 程序和进程](#)
- [1.5 虚拟内存](#)
- [1.6 并发与并行](#)

## 二、文件IO

- [2.1 文件描述符](#)
- [2.2 打开关闭文件（open和close函数）](#)
- [2.3 文件读写（read和write函数）](#)
- [2.4 lseek（文件偏移量）](#)
- [练习--实现自己的cp函数 -- mycp](#)
- [2.5 阻塞、非阻塞](#)
- [2.6 fcntl（修改以打开文件的状态标志）](#)
- [2.7 复制文件描述符（dup、dup2）](#)
- [2.8 文件报错（perror、strerror）](#)
- [2.9 获取文件状态（stat、lstat、fstat）](#)
- [文件I/O练习](#)

## 三、进程

- [3.1 进程状态](#)
- [3.2 创建进程](#)
- [3.3 进程终止](#)
- [3.4 孤儿进程](#)
- [3.5 僵尸进程](#)
- [3.6 回收进程](#)
- [3.7 exec函数族](#)

## 四、进程通信

- [4.1 管道](#)
  - [4.1.1 匿名管道](#)

#### 4.1.2 命名管道

练习：两个进程使用 fifo 互发消息聊天

#### 4.2 内存映射

#### 4.3 消息队列

### 五、线程

#### 5.1 创建线程

#### 5.2 线程终止

#### 5.3 线程的链接与分离

### 六、线程同步

#### 6.1 互斥锁（互斥量）

#### 6.2 读写锁

#### 6.3 条件变量

#### 6.4 信号量

# 基本概念

2024年5月18日 11:17

## 一、基本概念

- 1.1 操作系统
- 1.2 库函数和系统调用
- 1.3 内核
- 1.4 程序和进程
- 1.5 虚拟内存
- 1.6 并发与并行

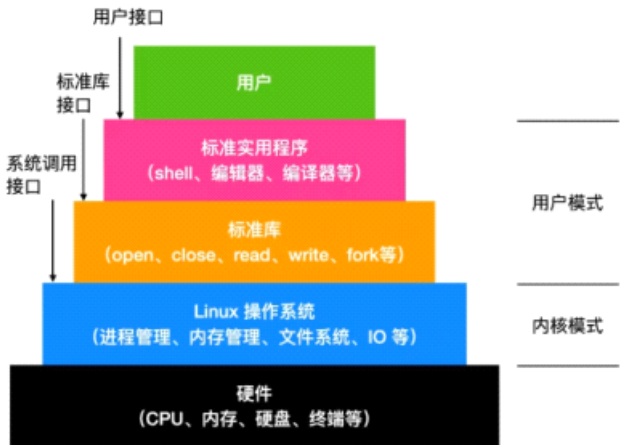
## 一、 基本概念

### 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

```
1 可执行程序或 shell 命令
2 系统调用(内核提供的函数)
3 库调用(程序库中的函数)
4 特殊文件(通常位于 /dev)
5 文件格式和规范, 如 /etc/passwd
6 游戏
7 杂项(包括宏包和规范), 如 man(7), groff(7), man-pages(7)
8 系统管理命令(通常只针对 root 用户)
Manual page man(1) line 1 (press h for help or q to quit)
```

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I/O

2024年5月27日 15:42

## 二、文件I/O

### [2.1 文件描述符](#)

### [2.2 打开关闭文件（open和close函数）](#)

### [2.3 文件读写（read和write函数）](#)

### [2.4 lseek（文件偏移量）](#)

### [练习--实现自己的cp函数 -- mvcp](#)

### [2.5 阻塞、非阻塞](#)

### [2.6 fcntl（修改以打开文件的状态标志）](#)

### [2.7 复制文件描述符（dup、dup2）](#)

### [2.8 文件报错（perror、strerror）](#)

### [2.9 获取文件状态（stat、lstat、fstat）](#)

### [文件I/O练习](#)

## 二、文件I/O

### 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 (Nonblock I/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 offset), 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

### lseek 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



```

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 (Nonblock I/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。

// 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

怎么用一個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，能够确定是哪里出问题。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perror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，lstat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：

pathname: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

```
#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_mtim));
    // 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

// 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

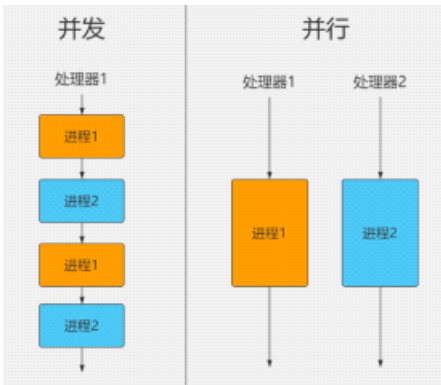
## 三、进程

- 3.1 进程状态
- 3.2 创建进程
- 3.3 进程终止
- 3.4 孤儿进程
- 3.5 僵尸进程
- 3.6 回收进程
- 3.7 exec函数族

## 三、进程

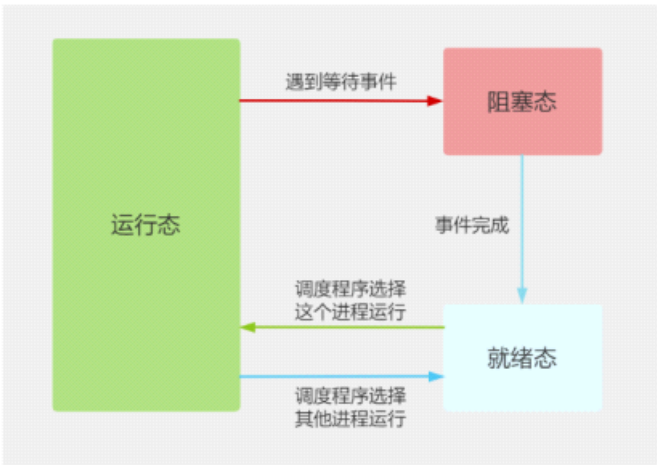
### 3.1 进程状态

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

```
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;
}
```

```
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;
}
```

```

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;
}

```

```

// 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;
}

```

```

// 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 \$?中保存异常出错的错误号

```
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）

```
// 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，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);
}
```



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会:

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→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
// 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
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

```
weihong@weihong:~/linux/Day_608/读写锁$ ps ajx | grep "main"
2602 2608 2522 2522 ? -1 Sl 1000 1:10 /home/weihong/.vscode-serv
52f762678dec6ca2cc69aba1570769a5d39/server/node /home/weihong/.vscode-server/cli/servers/Stab
9aba1570769a5d39/server/out/server-main.js --connection-token=remotessh --accept-server-licen
nable-remote-auto-shutdown --socket-path=/tmp/code-0926c159-abc6-44c4-bc25-0425a2f1e878
13494 18111 18111 13494 pts/11 18111 S+ 1000 0:00 ./main2
13502 18132 18131 13502 pts/12 18131 S+ 1000 0:00 grep --color=auto main
weihong@weihong:~/linux/Day_608/读写锁$ ps ajx | grep "main"
2602 2608 2522 2522 ? -1 Sl 1000 1:10 /home/weihong/.vscode-serv
52f762678dec6ca2cc69aba1570769a5d39/server/node /home/weihong/.vscode-server/cli/servers/Stab
9aba1570769a5d39/server/out/server-main.js --connection-token=remotessh --accept-server-licen
nable-remote-auto-shutdown --socket-path=/tmp/code-0926c159-abc6-44c4-bc25-0425a2f1e878
13494 18111 18111 13494 pts/11 18111 R+ 1000 0:00 ./main aaa bbb
13502 18188 18187 13502 pts/12 18187 S+ 1000 0:00 grep --color=auto main
```

结论:

- (1) 执行execl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execl函数后, 虽然进程id不变, 但是进程名字会变
- (3) 执行execl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:ls、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:ls、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ls/>>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;
}
```

四、进程通信

4.1 管道

4.1.1 匿名管道

4.1.2 命名管道

练习：两个进程使用 fifo 互发消息聊天

4.2 内存映射

4.3 消息队列

四、进程通信

进程间通信简称IPC(Inter process communication)，进程间通信就是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。

因为没有文件名，到不同的程序中，没有文件名就找不到这个管道文件。

因此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

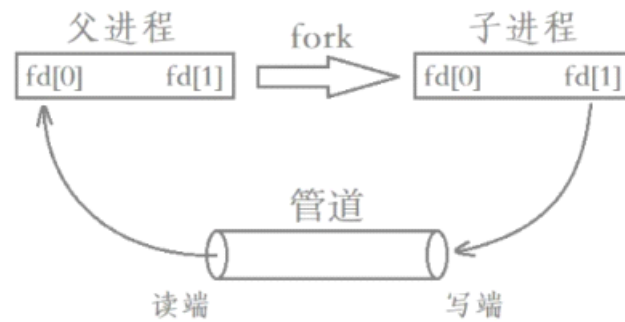
pipefd[1]:管道写端的文件描述符



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;
}
```

## 4.1.2 命名管道

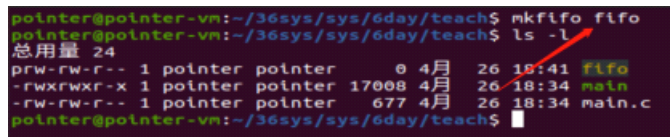
匿名管道只能用于具有共同祖先的进程(具有亲缘关系的进程)之间的通信,通常,一个管道由一个进程创建,然后该进程调用fork,此后父子进程之间就可应用该管道。如果要实现两个毫不相关进程之间的通信,可以使用命名管道来做到。命名管道就是一种特殊类型的文件,两个进程通过命名管道的文件名打开同一个管道文件,此时这两个进程也就看到了同一份资源,进而就可以进行通信了。命名管道和匿名管道一样,都是内存文件,只不过**命名管道在磁盘有一个简单的映像,但这个映像的大小永远为 0**,因为**命名管道和匿名管道都不会将通信数据刷新到磁盘当中**。

命名管道,命名指的就是文件名。

### 1、创建命名管道

#### (1) 系统命令创建

mkfifo fifo



```
pointer@pointer-vm:~/36sys/sys/6day/teach$ mkfifo fifo
pointer@pointer-vm:~/36sys/sys/6day/teach$ ls -l
总用量 24
prw-rw-r-- 1 pointer pointer 0 4月 26 18:41 fifo
-rwxrwxr-x 1 pointer pointer 17008 4月 26 18:34 main
-rw-rw-r-- 1 pointer pointer 677 4月 26 18:34 main.c
pointer@pointer-vm:~/36sys/sys/6day/teach$
```

#### (2) 函数创建---mkfifo 函数

**函数描述:**程序中创建命名管道

**头文件:**

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

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

**函数原型:**

```
int mkfifo(const char *pathname, mode_t mode),
```

**函数参数:**

pathname:表示要创建的命名管道文件

mode:表示创建命名管道文件的默认权限

**函数返回值:**

成功, 返回 0

失败, 返回 -1

### 命名管道在父子进程间通信

```
int main(int argc, char* argv[])
{
    // 创建命名管道
    mkfifo("./fifo",0644);
    // 打开命名管道
    int fd = open("./fifo",O_RDWR);
    // 创建子进程
    int pid = fork();
    if(pid == 0) // 子进程, 向命名管道写入内容
    {
        write(fd,"hello fifo",11);
    }
    else
    {
        // 父进程读取管道中的内容
        char buf[64];
        int ret = read(fd,buf,sizeof(buf));
        printf("ret = %d,buf = %s\n",ret,buf);
    }
    return 0;
}
```

### 命名管道在没有血缘关系之间的进程间通信

```
write.c
int main(int argc, char* argv[])
{
    // 以写入方式, 打开命名管道
    int fd = open("./fifo",O_WRONLY);
    // 打印写入时的文件描述符
    printf("wr_fd = %d\n",fd);
    // 向命名管道写入内容
    write(fd,"hello fifo",11);
}
```

```

    int ret = write(fd, "hello fifo1", 12);
    // 打印写入的字符
    printf("wr_ret = %d\n", ret);
    sleep(2);
    printf("111\n");
    return 0;
}

```

#### read.c

```

int main(int argc, char* argv[])
{
    // 以读取方式, 打开命名管道
    int fd = open("./fifo", O_RDONLY);
    // 打印读取的文件描述符
    printf("read_fd = %d\n", fd);
    // 读取管道内容
    char buf[64];
    int ret = read(fd, buf, sizeof(buf));
    // 打印读到的信息
    printf("rd_ret = %d, buf = %s\n", ret, buf);
    return 0;
}

```

在没有写入内容时，读端会一直阻塞，等待写端写入。  
 在没有读取时，写入也会一直阻塞，直到有读端打开

这里读写的阻塞是根据打开管道文件时的权限来判断是读还是写操作的。因此，在打开命名管道时，**一定要确定好打开时的读写方式。**

#### 命名管道的打开规则：

- 读进程打开 FIFO，并且没有写进程打开时：
  - 没有O\_NONBLOCK:阻塞直到有写进程打开该 FIFO
  - 有O\_NONBLOCK:立刻返回成功
- 写进程打开 FIFO，并且没有读进程打开时：
  - 没有O\_NONBLOCK:阻塞直到有读进程打开该FIFO
  - 有O\_NONBLOCK:立刻返回失败，错误码为ENXIO

#### 练习：两个进程使用 fifo 互发消息聊天

1. 在终端输入和打印消息
2. 能一直发消息和接收消息
3. 互发(两个管道，两个进程)

#### wfifo.c

```

int main(int argc, char* argv[])
{
    char buf[1024];
    int ret;
    int pid = fork();
    printf("pid = %d", pid);
    if(pid == 0) // 写
    {
        int fd = open("./wr_fifo", O_WRONLY);
        while(1)
        {
            int read_count = read(0, buf, sizeof(buf));
            write(fd, buf, read_count);
        }
    }
    else // 读
    {
        int fd1 = open("./rd_fifo", O_RDONLY);
        while(1)
        {
            int ret = read(fd1, buf, sizeof(buf));
            write(1, buf, ret);
        }
    }
}

```

```

    return 0;
}

rfifo.c
int main(int argc, char* argv[])
{
    char buf[1024];
    int ret;
    int pid = fork();
    if(pid == 0) // 写
    {
        int fd = open("./rd_fifo", O_WRONLY);
        while(1)
        {
            int read_count = read(0, buf, sizeof(buf));
            write(fd, buf, read_count);
        }
    }
    else // 读
    {
        int fd1 = open("./wr_fifo", O_RDONLY);
        while(1)
        {
            int ret = read(fd1, buf, sizeof(buf));
            write(1, buf, ret);
        }
    }
    return 0;
}

```

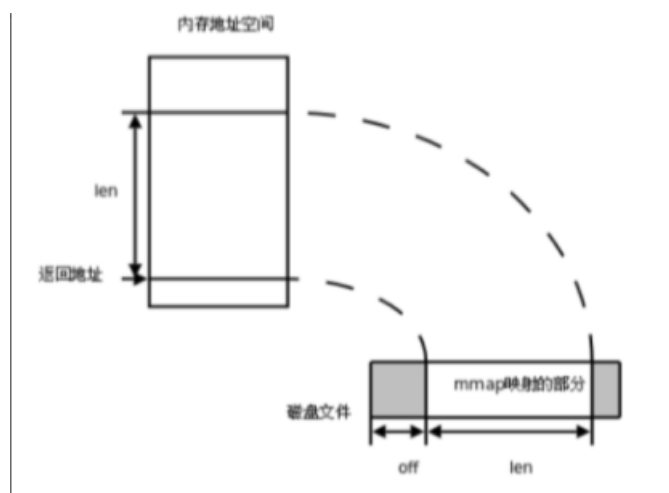
## 4.2 内存映射

管道的本质还是文件，不同进程通过访问管道文件来进行通信，但管道是在内存上的文件，并不在磁盘进行更新。

我们能否自定义一个内存空间，来让不同的进程来访问这块空间，来实现共享内存，实现进程通信呢？

内存映射即可解决上面的问题。

内存映射，**实际上是拿一个磁盘文件，将其映射到内存上（物理内存）**，然后将这块内存给到不同的进程，以实现内存共享。



内存映射 (Memory-mapped I/O) 是将磁盘文件的数据映射到内存，用户通过修改内存就能修改磁盘文件。

**映射分为两种：**

**文件映射：**将文件的一部分映射到调用进程的虚拟内存中。对文件映射部分的访问转化为对相应内存区域的字节操作。映射页面会按需自动从文件中加载。

**匿名映射：**一个匿名映射没有对应的文件。其映射页面的内容会被初始化为 0。

**一个进程所映射的内存可以与其他进程的映射共享，共享的两种方式：**

两个进程对同一文件的同一区域映射。

fork() 创建的子进程继承其父进程的映射。

## mmap() 函数

### 函数描述:

在调用进程的虚拟地址空间中创建一个新内存映射。

### 头文件:

<sys/mman.h>

### 函数原型:

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

### 函数参数:

addr: 指向欲映射的内存起始地址，通常设为NULL，代表系统自动选定地址

length: 映射的长度。

prot: 映射区域的保护方式:

PROT\_READ: 映射区域可读取

PROT\_WRITE: 映射区域可修改

flags: 影响映射区域的特性。必须指定MAP\_SHARED(共享) 或 MAP\_PRIVATE(私有)

MAP\_SHARED: 创建共享映射。对映射的写入会写入文件里，其他共享映射的进程可见

MAP\_PRIVATE: 创建私有映射。对映射的写入不会写入文件里，其他映射进程不可见

MAP\_ANONYMOUS: 创建匿名映射。此时会忽略参数fd(设为-1)，不涉及文件，没有血缘关系的进程不能共享

fd: 要映射的文件描述符，匿名映射设为 -1

offset: 文件映射的偏移量，通常设置为0，代表从文件最前方开始对应，offset必须是分页大小(4k)的整数倍。

### 函数返回值:

若映射成功则返回映射区的内存起始地址，

失败返回MAP\_FAILED(-1)，错误原因存于 errno 中。

## munmap() 函数

### 函数描述:

解除映射区域

### 头文件:

<sys/mman.h>

### 函数原型:

```
int munmap(void *addr, size_t length);
```

### 函数参数:

addr: 指向要解除映射的内存起始地址

length: 解除映射的长度

## 父子进程间通信——匿名映射

// 匿名映射

```
int main(int argc, char* argv[])
{
    // void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
    // 创建匿名映射
    char* str = (char*)mmap(NULL, 1024, PROT_READ | PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0);
    int pid = fork();
    if(pid == 0)
    {
        strcpy(str, "hello mmap");
        printf("child str = %s\n", str);
    }
    else
    {
        // 内存映射没有阻塞，读端得加 sleep()，确保读者之间就写入了。
        sleep(2);
        printf("father str = %s\n", str);
    }
    return 0;
}
```

## 不同进程间通信——命名映射

#### w\_mmap.c

```
// 命名映射-写端
int main(int argc, char* argv[])
{
    int fd = open("./mmap.txt", O_RDWR | O_CREAT, 0644);
    // 这里会报错, 因为要将磁盘上的文件映射到1024大小的内存上, 但是文件刚刚创建, 大小是0, 无法映射
    // 因此要扩容
    // 截断函数, 截断到某一位位置, 0就是清除内容。
    ftruncate(fd, 1024);
    char* str = (char*)mmap(NULL, 1024, PROT_WRITE, MAP_SHARED, fd, 0);
    // 映射失败报错
    if(str == MAP_FAILED)
    {
        perror("mmap error");
        exit(1);
    }
    int i = 0;
    while(1)
    {
        // 格式化打印, 打印内容到指定位置, 而不是终端
        sprintf(str, "---hello %d---\n", i++);
        sleep(1);
        // 每秒打印 ---hello i--- 到内存映射中
    }

    return 0;
}
```

#### r\_mmap.c

```
// 命名映射-读端
int main(int argc, char* argv[])
{
    int fd = open("./mmap.txt", O_RDWR);
    char* str = (char*)mmap(NULL, 1024, PROT_READ, MAP_SHARED, fd, 0);
    while(1)
    {
        // 每秒从内存映射中读取一次
        printf("read str = %s", str);
        sleep(1);
    }
    return 0;
}
```

## 4.3 消息队列

消息队列是保存在内核中的消息链表, 消息队列是面向消息进行通信的, 一次读取一条完整的消息, 每条消息中还包含一个整数表示优先级, 可以根据优先级读取消息。进程A可以往队列中写入消息, 进程 B读取消息。并且, 进程A写入消息后就可以终止, 进程B在需要的时候再去读取。

每条消息通常具有以下属性:

- (1) 一个表示优先级的整数
- (2) 消息数据部分的长度
- (3) 消息数据本身

### 消息队列函数

头文件:

```
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>
```

打开和关闭消息队列:

```
mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);
```

```

    int mq_close(mqd_t mqdes);
获取和设置消息队列属性:
    int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
    int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr, struct mq_attr *oldattr);
在队列中写入和读取一条消息:
    int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio);
    ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio);
删除消息队列名:
    int mq_unlink(const char *name);

```

函数参数和返回值:

name:消息队列名  
 oflag:打开方式, 类似 open 函数。  
 必选项:O\_RDONLY, O\_WRONLY, O\_RDWR  
 可选项:O\_NONBLOCK, O\_CREAT, O\_EXCL  
 mode:访问权限, oflag 中含有 O\_CREAT 且消息队列不存在时提供该参数  
 attr:队列属性, open 时传 NULL 表示默认属性  
 mqdes:表示消息队列描述符  
 msg\_ptr:指向缓冲区的指针  
 msg\_len:缓冲区大小  
 msg\_prio:消息优先级

返回值:

成功返回 0, open 返回消息队列描述符, mq\_receive 返回写入成功字节数  
 失败返回 -1

### 注意

在编译时报 undefined reference to mq\_open、undefined reference to mq\_close 时, 除了要包含头文件 #include <mqqueue>#include <fcntl>外, 还需要加上编译选项 -lrt。

### 消息队列的关闭与删除

使用 mq\_close 函数**关闭消息队列**, 关闭后消息队列并不从系统中删除。一个**进程结束**会自动调用关闭打开着的消息队列。

使用 mq\_unlink 函数**删除一个消息队列名**, 使**当前进程无法使用该消息队列**。并将队列标记为在所有进程关闭该队列后删除该队列。

Posix 消息队列具备随内核的持续性。所以即使当前没有进程打开着某个消息队列, 该队列及其上的消息也将一直存在, 直到调用 mq\_unlink 并最后一个进程关闭该消息队列时, 将会被删除。操作系统会维护一个消息队列的引用计数, 记录有多少进程正在使用该消息队列。只有当引用计数减为零时, 消息队列才会被真正删除。

### 查看消息队列的状态

```

int main(int argc, char* argv[])
{
    // 打开消息队列
    mqd_t mqd = mq_open("/mymsg", O_RDONLY);
    if(mqd == -1)
    {
        perror("mq_open error\n");
        exit(1);
    }

    // 接收消息队列状态
    struct mq_attr attr;
    int ret = mq_getattr(mqd, &attr);
    if(ret == -1)
    {
        perror("mq_getattr error\n");
        exit(1);
    }

    printf("mq_curmsgs = %ld\n", attr.mq_curmsgs); // 现有消息数量
    printf("mq_msgsize = %ld\n", attr.mq_msgsize); // 消息队列大小
    printf("mq_maxmsg = %ld\n", attr.mq_maxmsg); // 最大消息数量
}

```

```
    return 0;  
}
```



## 五、线程

### [5.1 创建线程](#)

### [5.2 线程终止](#)

### [5.3 线程的链接与分离](#)

## 五、线程

与进程类似，线程(thread)是允许程序并发执行多任务的一种机制。**一个进程可以包含多个线程**，每个线程会独立执行相同的程序代码，且共享同一份全局内存区域。

同一进程中的多个线程可以并发执行。**在多处理器环境下，多个线程可以同时并行**。如果一个线程因等待 I/O 操作而遭阻塞，那么其他线程依然可以继续运行。

### 线程相对进程的优势：

进程间的信息难以共享，需要进程间通信。线程间能方便、快速的共享信息，不过要避免多个线程同时修改同一数据的情况。

调用fork() 创建进程的代价较高，需要复制父进程的内容。线程比进程的创建要快很多。

### 5.1 创建线程

启动程序时，产生的进程只有单条线程，称之为初始(initial)或主(main)线程。函数pthread create() 负责创建一条新线程。新线程通过调用带有参数 arg 的函数 start routine(即 start routine(arg))而开始执行。调用 pthread create() 的线程会继续执行该调用之后的语句。

进程内部的每个线程都有一个唯一标识，称为线程 ID。线程获取自己的线程 ID 使用pthread self() 函数。

使用命令 ps -elf 查看线程

ubuntu20 编译有线程的代码可能失败，使用 gcc 编译，加参数 -l pthread

### pthread\_create函数

#### 函数描述：

创建一个新线程。类似进程中的fork() 函数，

#### 头文件：

```
#include <pthread.h>
```

#### 函数原型：

```
int pthread_create(pthread_t*thread,
                   const pthread_attr_t *attr,
                   void*(*start_routine)(void *),
                   void *arg);
```

#### 函数参数：

thread:传出参数，保存系统为我们分配好的线程ID

attr:通常传 NULL，表示使用线程默认属性。若想使用具体属性也可以修改该参数。

start\_routine:函数指针，指向线程主函数(线程体)，该函数运行结束，则线程结束

线程主函数的参数:arg

#### 函数返回值：

成功返回 0  
失败返回错误号

### pthread\_self函数

#### 函数描述:

获取线程 ID。类似于进程中的getpid()函数。线程ID 是进程内部的识别标志(两个进程间, 允许线程ID 相同)

#### 函数原型:

```
pthread_t pthread_self(void),
```

#### 函数返回值:

成功返回线程ID, pthread\_t为无符号整型(%lu)  
失败无

## 5.2 线程终止

### 终止线程的方式:

- (1)线程 start\_routine 函数执行 return 。
- (2)线程调用 pthread\_exit()终止线程。
- (3)调用 pthread\_cancel(thread)取消指定线程
- (4)任意线程调用了 exit(), 或者主线程执行了return 语句(在 main()函数中), 都会导致进程中的所有线程立即终止

pthread\_exit()函数将终止调用线程, 且其返回值可由另一线程通过调用 pthread\_join()来获取。调用 pthread\_exit()相当于在线程的 start routine 函数中执行return, 不同之处在于可在线程 start routine 函数所调用的任意函数中调用 pthread\_exit()都能够直接终止线程。

### pthread\_exit函数

#### 函数描述:

终止线程

#### 函数原型:

```
void pthread_exit(void *retval);
```

#### 函数参数:

retval:表示线程退出状态, 通常传 NULL

### 调用pthread\_exit()线程结束:

```
int a = 0;
void* thread1(void* arg)
{
    printf("thread1 id = %lu\n", pthread_self());
    while(a < 10)
    {
        printf("a = %d\n", a++);
        sleep(1);
        pthread_exit(NULL);
    }
    return NULL;
}
int main(int argc, char* argv[])
{
    pthread_t mpd;

    pthread_create(&mpd, NULL, thread1, NULL);
    printf("main id = %lu\n", pthread_self());
    sleep(10);

    return 0;
}
```

### 主函数return, 线程终止

```
// 主函数return, 线程终止
int a = 0;
void* thread1(void* arg)
```

```

{
    printf("thread1 id = %lu\n", pthread_self());
    while(a < 10)
    {
        printf("a = %d\n", a++);
        sleep(5);
    }
    return NULL;
}
int main(int argc, char* argv[])
{
    pthread_t mpd;

    pthread_create(&mpd, NULL, thread1, NULL);
    printf("main id = %lu\n", pthread_self());

    return 0;
    // exit(1);
}

```

## pthread\_cancel函数

### 函数描述:

向指定线程发送一个取消请求。发出取消请求后，函数pthread\_cancel()当即返回不会等待目标线程的退出。被请求取消的线程不会立即取消，需要等待到达某个取消点取消点通常是一些系统调用，也可以使用pthread\_testcancel 函数手动创建一个取消点，被取消的线程返回值是 PTHREAD\_CANCELED(-1)。

### 函数原型:

int pthread\_cancel(pthread\_t thread).

### 函数参数:

thread: 要取消的线程ID

### 函数返回值:

成功返回 0

失败返回错误号

```

// pthread_cancel函数
int a = 0;
void* thread1(void* arg)
{
    while(1)
    {
        // a 在0~10循环
        if(a==10)
        {
            a = 0;
        }
        a++;

        //printf("hello\n"); // 取消点
        pthread_testcancel(); //取消点
    }
    return NULL;
}
int main(int argc, char* argv[])
{
    pthread_t mpd;

    pthread_create(&mpd, NULL, thread1, NULL);
    // 取消线程，不会立即取消，得等待线程1中执行取消点（系统调用）
    int ret = pthread_cancel(mpd);
    if(ret != 0)
    {
        printf("pthread_cancel error, ret = %d\n", ret);
    }
    else
    {
        printf("pthread_cancel success, ret = %d\n", ret);
    }
    while(1)
    {
        printf("ret = %d\n", ret);
        sleep(1);
    }
}

```

```

    return 0;
}

```

### 5.3 线程的链接与分离

#### pthread\_join 函数

##### 函数描述:

等待指定线程终止并回收, 这种操作叫做连接(joining)。未连接的线程会产生僵尸线程, 类似僵尸进程的概念。

##### 函数原型:

```
int pthread_join(pthread_t thread, void **retval);
```

##### 函数参数:

thread: 要等待的线程 ID  
retval: 存储线程返回值

##### 函数返回值:

成功返回 0  
失败返回错误号

#### pthread\_detach函数

##### 函数描述:

默认线程是可连接的(ioinable)当线程退出时, 其他线程可以通过调用pthread\_join() 获取其返回状态。有时, 程序员并不关心线程的返回状态, 只是希望系统在线程终止时能够自动清理并移除。这时可以调用 pthread\_detach() 函数, 将线程标记为分离(detached)状态。

##### 函数原型:

```
int pthread_detach(pthread_t thread);
```

##### 函数参数:

thread: 要分离的线程ID  
retval: 存储线程返回值

##### 函数返回值:

成功返回 0  
失败返回错误号

### 练习

定义一个全局变量 a, 创建 10 个线程, 每个线程对a进行自增(a++)100万次, 所有线程自增结束主线程打印 a 的值。

```

int a = 0;
void* thread1(void* arg)
{
    printf("thread1 id = %lu\n", pthread_self());
    for(int i = 0; i < 1000000; i++)
    {
        a++;
    }
    return NULL;
}
int main(int argc, char* argv[])
{
    pthread_t mpd;
    pthread_t mpd_Arr[10];
    for(int i = 0; i < 10; i++)
    {
        pthread_create(&mpd_Arr[i], NULL, thread1, NULL);
    }
    for(int i = 0; i < 10; i++)
    {
        pthread_join(mpd_Arr[i], NULL);
    }
}

```

```
printf("main id = %lu\n", pthread_self());  
// sleep(10);  
printf("a = %d\n", a); // a!=100w  
return 0;  
}
```

## 六、线程同步

### [6.1 互斥锁（互斥量）](#)

### [6.2 读写锁](#)

### [6.3 条件变量](#)

### [6.4 信号量](#)

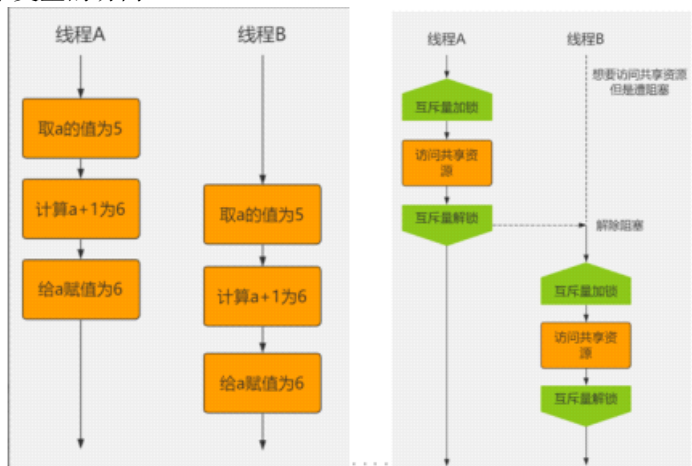
## 六、线程同步

子线程没有独立的地址空间，大部分数据都是共享的，如果同时访问数据，就会造成混乱，所以要进行控制，线程之间要协调好先后执行的顺序。**同步就是协同步调，按预定的先后次序进行运行。**如：你说完，我再说。**这里的同步千万不要理解成那个同时进行，应是指协同、协助、互相配合。**线程同步是指多线程通过特定的设置(如互斥量，条件变量等)来控制线程之间的执行顺序(即所谓的同步)也可以说是在线程之间通过同步建立起执行顺序的关系，如果没有同步，那线程之间是各自运行各自的！线程互斥是指对于共享的进程系统资源，在各单个线程访问时的排它性。当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。

### 6.1 互斥锁（互斥量）

线程的主要优势在于能够通过全局变量来共享信息。这种便捷的共享是有代价的：必须确保多个线程不会同时修改同一变量，或者某一线程不会读取正由其他线程修改的变量

互斥量可以保护对共享变量的访问。



**a++需要执行3个步骤：**

1. 取a的值
2. 计算 a+1
3. a+1 赋值给 a

某一时刻，全局变量a的值为5，线程A和线程B中都要执行a++，如果线程A在执行a+1 赋值给a之前，这时线程B执行了a++，线程B取到a的值仍是5，这时，线程A和 B赋给a的值都是 6。值为5的变量a经过两次a++结果却是6，显然出现了错误。

**多线程竞争操作共享变量的这段代码叫做临界区。**多个进程同时操作临界区会产生错误所以这段代码应该互斥，当一个线程执行临界区时，应该阻止其他线程进入临界区。

为避免线程更新共享变量时出现问题，可以使用互斥量(mutex是mutualexclusion 的缩写)来**确保同时仅有一个线程可以访问某项共享资源**。

互斥量的作用类似于一个“锁”，当一个线程访问共享资源时，它必须先尝试获取互斥量的锁。如果互斥量当前没有被其他线程占用，那么该线程将成功获取锁，并可以安全地访问共享资源:如果互斥量已经被其他线程占用，那么该线程将被阻塞，直到互斥量的锁被放。

互斥量有两种状态:**已锁定(locked)**和**未锁定(unlocked)**。至多只有一个线程可以锁定该互斥量，试图对已经锁定的某一互斥量再次加锁将会阻塞线程。一旦线程锁定斥量，随即成为该互斥量的所有者，只有所有者才能给互斥量解锁。

### pthread\_mutexinit 函数

**函数描述:**

初始化一个互斥量

**函数原型:**

```
int pthread_mutex_init(pthread_mutex_t*mutex, const pthread_mutex_attr_t* mutexattr);
```

**函数参数:**

mutex:指向互斥量的指针，下面几个函数都有该参数，不一一介绍

mutexattr:指向定义互斥量属性的指针，取默认值传 NULL

**函数返回值:**

成功返回 0

失败返回错误号

### pthread\_mutex\_lock和pthread\_mutex\_unlock函数

**函数描述:**

给互斥量加锁和解锁，解锁的函数在解锁的同时唤醒阻塞在该互斥量上的线程，默认先阻塞的先唤醒。

**函数原型:**

```
int pthread_mutex_lock(pthread_mutex_t* mutex),  
int pthread_mutex_unlock(pthread_mutex_t* mutex),
```

**函数返回值:**

成功返回 0

出现错误返回错误号。加锁不成功，线程阻塞

### pthread\_mutex\_destroy 函数

**函数描述:**

销毁一个互斥量

**函数原型:**

```
int pthread_mutex_destroy(pthread_mutex_t*mutex);
```

**函数返回值:**

成功返回 0

失败返回错误号

### pthread\_mutex\_trylock函数

**函数描述:**

尝试给互斥量加锁，加锁不成功直接返回错误号(EBUSY)，不会阻塞，其他与pthread\_mutex\_lock相同。

```
// 创建互斥量  
pthread_mutex_t mtx;  
int a = 0;  
void* thread1(void* arg)  
{  
    printf("thread1 id = %lu\n", pthread_self());  
    for(int i = 0; i < 10000000; i++)  
    {  
        pthread_mutex_lock(&mtx); // 添加锁  
        a++;  
        pthread_mutex_unlock(&mtx); // 解锁  
    }  
    return NULL;  
}  
int main(int argc, char* argv[])  
{
```

```

// 初始化互斥量
pthread_mutex_init(&mtx, NULL);
pthread_t mpd;
pthread_t mpd_Arr[10];
for(int i = 0; i < 10; i++)
{
    pthread_create(&mpd_Arr[i], NULL, thread1, NULL);
}
// 连接线程, 等待
for(int i = 0; i < 10; i++)
{
    pthread_join(mpd_Arr[i], NULL);
}

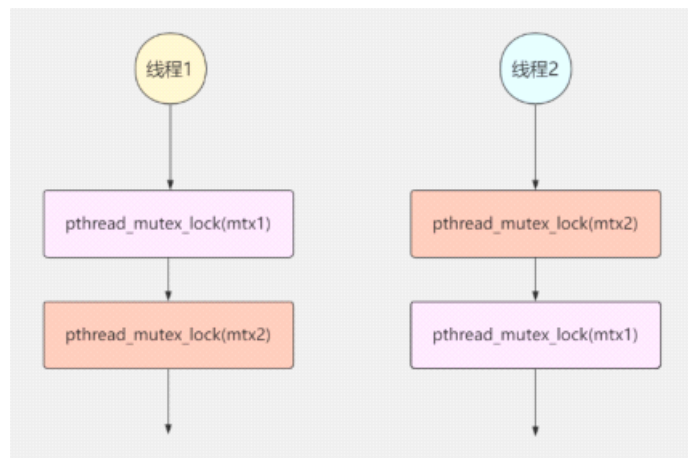
printf("main id = %lu\n", pthread_self());
// sleep(10);
printf("a = %d\n", a);

pthread_mutex_destroy(&mtx); // 销毁锁
return 0;
}

```

## 死锁现象

当多个线程中为了保护多个共享资源而使用了多个互斥锁，如果多个互斥锁使用不当，就可能造成，多个线程一直等待对方的锁释放，这就是死锁现象。



### 死锁产生的四个必要条件：

- (1) 互斥条件: 资源只能同时被一个进程占用
- (2) 持有并等待条件: 线程1已经持有资源 A，可以申请持有资源 B，如果资源B已经被
- (3) 线程 2 持有，这时线程1持有资源并等待资源 B不可剥夺条件: 一个线程持有资源，只能自己释放后其他线程才能使用。其他线程不能强制收回该资源
- (4) 环路等待条件: 多个线程互相等待资源，形成一个环形等待链

### 避免死锁: 破坏其中一个必要条件就可以避免死锁，常用的方法如下：

- (1) 锁的粒度控制: 破坏请求与保持条件，尽可能减少持有锁的时间，降低发生死锁的可能性
- (2) 资源有序分配: 破坏环路等待条件，规定线程使用资源的顺序，按规定顺序给资源加锁
- (3) 重试机制: 破坏不可剥夺条件，如果尝试获取资源失败，放弃已持有的资源后重试

```

pthread_mutex_t mtx1;
pthread_mutex_t mtx2;

void* thread1(void* arg)
{
    while (1)
    {
        printf("子线程获取 mtx1中。。。 \n");
        pthread_mutex_lock(&mtx1);
        printf("子线程获取 mtx1成功! \n");
        sleep(1);
        printf("子线程获取 mtx2中。。。 \n");
    }
}

```



```

        int ret = pthread_mutex_trylock(&mtx2);
        if((ret != 0) && (ret == EBUSY))
        {
            printf("mtx2被占用, 已成功释放! \n");
            pthread_mutex_unlock(&mtx2);
            sleep(1);
        }
        printf("子线程获取 mtx2成功! \n");
        pthread_mutex_unlock(&mtx2);
        pthread_mutex_unlock(&mtx1);
        return NULL;
    }
}

int main(int argc, char* argv[])
{
    pthread_mutex_init(&mtx1, NULL);
    pthread_mutex_init(&mtx2, NULL);
    pthread_t ptid;
    pthread_create(&ptid, NULL, thread1, NULL);
    printf("主线程获取 mtx2中。。。 \n");
    pthread_mutex_lock(&mtx2);
    printf("主线程获取 mtx2成功! \n");
    sleep(1);
    printf("主线程获取 mtx1中。。。 \n");
    pthread_mutex_lock(&mtx1);
    printf("主线程获取 mtx1成功! \n");
    pthread_mutex_unlock(&mtx1);
    pthread_mutex_unlock(&mtx2);

    printf("线程结束! ! \n");
    while(1);

    return 0;
}

```

## 练习

### 模拟选座系统

有 10个空座, 12个用户(线程)同时选座, 系统从空座中随机选择  
打印出当前用户的序号和选中座位序号(线程创建顺序就是用户序号顺序)。  
输出正确结果应为:10个用户成功, 2个失败, 没有重复座位

```

#define SEAT_NUM 10 // 座位数量
#define USER_NUM 12 // 选座人数
pthread_mutex_t mtx;
// 座位数组
int seat[SEAT_NUM];
// 空座数量
int empty_seat_count = SEAT_NUM;
int chooseSeat()
{
    if(empty_seat_count > 0)
    {
        // 生成随机座位号—座位数组的下标
        int rand_index = rand()%empty_seat_count;
        // 获取座位号
        int seat_num = seat[rand_index];
        // 更新座位数组
        seat[rand_index] = seat[empty_seat_count-1];
        // 更新空座数量
        empty_seat_count--;
        return seat_num;
    }
    else
    {
        return -1;
    }
}

void* thread1(void* arg)
{
    .

```

```

{
    pthread_mutex_lock(&mtx); // 加锁
    int seat_num = chooseSeat(); // 选座
    pthread_mutex_unlock(&mtx); // 解锁
    if(seat_num!=-1)
    {
        printf("用户 %d 选座失败, 座位已售罄! \n",*(int*)arg);
    }
    else
    {
        printf("用户 %d 选座成功, 座位号是 %d\n",*(int*)arg, seat_num);
    }
}
}
int main(int argc, char* argv[])
{
    // 设置随机种子
    srand(time(NULL));
    // 初始化互斥锁
    pthread_mutex_init(&mtx, NULL);
    // 初始化座位号
    for(int i = 0; i < SEAT_NUM; i++)
    {
        seat[i] = i+1;
    }
    pthread_t pArr[USER_NUM]; // 存储线程id的数组
    for(int i = 0; i < USER_NUM; i++) // 给每个用户创建线程
    {
        int* pi = (int*)malloc(sizeof(int));
        *pi = i+1;
        pthread_create(&pArr[i], NULL, thread1, pi);
    }
    // 设置等待
    for(int i = 0; i < USER_NUM; i++)
    {
        pthread_join(pArr[i], NULL);
    }
    // 销毁互斥锁
    pthread_mutex_destroy(&mtx);
    return 0;
}

```

## 6.2 读写锁

读写锁，由读锁和写锁两部分组成，读取资源时用读锁，修改资源时用写锁。其特性为：**写独占，读共享(读优先锁)**。

**读写锁适合读多写少的场景。**

### 读写锁的工作原理

没有线程持有写锁时，所有线程都可以一起持有读锁

有线程持有写锁时，所有的读锁和写锁都会阻塞

读优先锁:有线程持有锁，这时有一个读线程和一个写线程想要获取锁，读线程会优先获取锁，就是读优先锁，反过来就是写优先锁。

### 读写锁函数

```

int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr),
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock),
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock)
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock),
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock)

```

### 场景分析

- (1) 持有读锁时，申请读锁：全部直接加锁成功，不需要等待
- (2) 持有写锁时，申请写锁：申请的写锁阻塞等待，写锁释放再申请加锁
- (3) 持有读锁时，申请写锁：写锁阻塞
- (4) 持有写锁时，申请读锁：读锁阻塞
- (5) 持有写锁时申请写锁和读锁：申请的读锁和写锁都会阻塞，当持有的写锁释放时，读锁先加锁成功
- (6) 持有读锁时申请写锁和读锁：申请的写锁阻塞，读锁加锁成功，写锁阻塞到读锁全部解锁才能加锁在此期间可能一直有读锁申请，会导致写锁一直无法申请成功，造成饥饿

#### 读优先锁：

线程 1 要加写锁，线程2要加写锁，线程3要加读锁：  
线程1直接成功加锁，线程2和线程3阻塞  
线程1释放锁  
由于是读优先锁，线程3加锁成功，线程2继续阻塞  
线程1 要加读锁，线程2要加写锁，线程3要加读锁：  
线程1成功加锁  
由于是读优先锁，线程 3 加锁成功，线程2继续阻塞条件变量

#### 练习

#### 使用读写锁模拟银行账户管理：

1. 有一个变量 balance 代表账户余额，存取钱就是对 balance 进行修改
2. 存5 次钱，在执行程序时传入5个整数代表钱的数额，sleep1 秒
3. 查询 10 次余额，查余额时随机 sleep1到3秒
4. 每一次的存钱和查余额操作都使用线程完成

```
/*
使用读写锁模拟银行账户管理
1. 有一个变量 balance 代表账户余额，存取钱就是对 balance 进行修改
2. 存5次钱，在执行程序时传入5个整数代表钱的数额，sleep1秒
3. 查询 10 次余额，查余额时随机 sleep1到3秒
4. 每一次的存钱和查余额操作都使用线程完成
*/
pthread_rwlock_t rwl;
int balance = 0;
// 存钱操作
int SetBalance(int money)
{
    sleep(1);
    balance += money;
    return balance;
}
// 查看操作
int GetBalance()
{
    int rand_time = rand()%3+1;
    sleep(rand_time);
    return balance;
}
// 存钱线程
void* threadSet(void* arg)
{
    pthread_rwlock_wrlock(&rwl);
    int money = atoi((char*)arg);
    int ret = SetBalance(money);
    printf("存钱成功，现有余额为: %d\n", ret);
    pthread_rwlock_unlock(&rwl);
    return NULL;
}
// 查看线程
void* threadGet(void* arg)
{
    pthread_rwlock_rdlock(&rwl);
    int ret = GetBalance();
    printf("读取余额成功，现有余额为: %d\n", ret);
}
```

```

pthread_rwlock_unlock(&rw1);
return NULL;
}
int main(int argc, char* argv[])
{
    if(argc !=6)
    {
        printf("输入的参数有误, 请输入5个存取的数额! \n");
        exit(1);
    }
    // 设置随机种子
    srand(time(NULL));
    // 初始化读写锁
    pthread_rwlock_init(&rw1, NULL);
    pthread_t pArr[15]; // 存储查看线程id的数组
    for(int i = 0; i < 5; i++) // 给存取钱创建线程
    {
        pthread_create(&pArr[i], NULL, threadSet, argv[i+1]);
    }
    for(int i = 5; i < 15; i++) // 给查看操作创建线程
    {
        pthread_create(&pArr[i], NULL, threadGet, NULL);
    }
    for(int i = 0; i < 15; i++) // 设置等待
    {
        pthread_join(pArr[i], NULL);
    }
    pthread_rwlock_destroy(&rw1);

    return 0;
}

```

结果：因为是读优先锁，因此，一旦有读锁进来，就得一直读。

```

weihong@weihong:~/linux/Day_613$ ./main 100 100 100 100 -100
存钱成功, 现有余额为: 100
读取余额成功, 现有余额为: 100
读取余额成功, 现有余额为: 100
读取余额成功, 现有余额为: 100
读取余额成功, 现有余额为: 100
读取余额成功, 现有余额为: 100
读取余额成功, 现有余额为: 100
读取余额成功, 现有余额为: 100
读取余额成功, 现有余额为: 100
读取余额成功, 现有余额为: 100
读取余额成功, 现有余额为: 100
存钱成功, 现有余额为: 200
存钱成功, 现有余额为: 300
存钱成功, 现有余额为: 400
存钱成功, 现有余额为: 300
weihong@weihong:~/linux/Day_613$

```

## 6.3 条件变量

条件变量，通知状态的改变。条件变量允许一个线程就某个共享变量的状态变化通知其他线程，并让其他线程等待(阻塞于)这一通知。条件变量总是结合互斥量使用。条件变量就共享变量的状态改变发出通知，而互斥量则提供对该共享变量访问的互斥。

### 演示程序：

定义一个变量 a，线程1当a为0时对 a+1，线程2当a为1时对 a-1，循环 10次，a的结果应该是 0和1交替。

两个线程需要不断的轮询结果，造成 CPU浪费。可以使用条件变量解决:线程1不满足运行条件时，先休眠等待，其他线程运行到满足线程1的运行条件时，通知并唤醒线程2继续执行。

### 条件变量函数

`int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr),`  
参数 `attr` 表示条件变量的属性，默认值传 `NULL`

`int pthread_cond_destroy(pthread_cond_t *cond);`  
`int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`  
阻塞等待一个条件变量，释放持有的互斥锁，这两步是原子操作  
被唤醒时，函数返回，解除阻塞并重新申请获取互斥锁

`int pthread_cond_signal(pthread_cond_t *cond);`  
唤醒一个阻塞在条件变量上的线程

`int pthread_cond_broadcast(pthread_cond_t *cond),`  
唤醒全部阻塞在条件变量上的线程

`int pthread_cond_timedwait(pthread_cond_t *cond,  
pthread_mutex_t *mutex  
const struct timespec *abstime),`

限时等待一个条件变量

参数 `abstime` 是一个 `timespec` 结构体，以秒和纳秒表示的绝对时间

// 0-1交替 -- 条件变量

```
pthread_mutex_t mtx;  
pthread_cond_t cont;  
int a = 0;  
void* thread1(void* arg)  
{  
    for(int i = 0; i < 10; i++)  
    {  
        pthread_mutex_lock(&mtx);  
        // 这里只能用while, 不能用if  
        // 因为如果是if的话, 他只会判断一次, 就阻塞在锁下面了。  
        // 如果, 多个线程同时执行的话, 因为不在判断条件 (a=1), 就有可能之间抢到锁,  
        // 接着++ 导致a=2, 不是0-1交替了。  
        // 因此这里只能使用while判断, 加锁失败, 阻塞之后, 想再次加锁, 得再次判断是否满足条件  
        while(a!=0)  
        {  
            // a != 0 时, 阻塞等待, 并将 thread1解锁释放, 然后thread2执行  
            // 如果被唤醒, 则重新加锁。  
            pthread_cond_wait(&cont, &mtx);  
        }  
        printf("a = %d\n", ++a);  
        pthread_cond_broadcast(&cont); // 唤醒  
        pthread_mutex_unlock(&mtx);  
    }  
    return NULL;  
}  
void* thread2(void* arg)  
{  
    for(int i = 0; i < 10; i++)  
    {  
        pthread_mutex_lock(&mtx);  
        while(a!=1)  
        {  
            // a != 1 时, 阻塞等待, 并将 thread2解锁释放, 然后thread1执行  
            // 如果被唤醒, 则重新加锁。  
            pthread_cond_wait(&cont, &mtx);  
        }  
        printf("a = %d\n", --a);  
        pthread_cond_broadcast(&cont); // 唤醒  
        pthread_mutex_unlock(&mtx);  
    }  
    return NULL;  
}  
int main(int argc, char* argv[])  
{  
    pthread_mutex_init(&mtx, NULL);  
    pthread_cond_init(&cont, NULL);
```

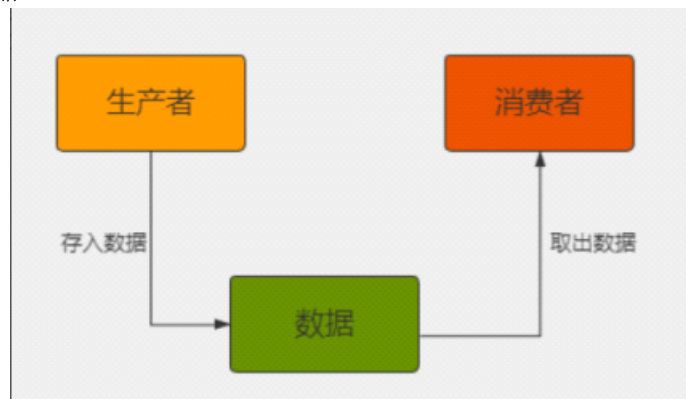
```

pthread_t ptid1;
pthread_t ptid2;
pthread_t ptid3;
pthread_t ptid4;
pthread_create(&ptid1, NULL, thread1, NULL);
pthread_create(&ptid2, NULL, thread2, NULL);
pthread_create(&ptid3, NULL, thread1, NULL);
pthread_create(&ptid4, NULL, thread2, NULL);
pthread_join(ptid1, NULL);
pthread_join(ptid2, NULL);
pthread_join(ptid3, NULL);
pthread_join(ptid4, NULL);
return 0;
}

```

## 生产者消费者模型

线程同步典型的案例即为生产者消费者**模型**，而借助条件变量来实现这一模型，是比较常见的一种方法。假定有两个线程，一个模拟生产者行为，一个模拟消费者行为。两个线程同时操作一个共享资源(一般称之为汇聚)，生产者向其中添加产品，消费者从中消费掉产品。



相较于互斥量而言，条件变量可以减少竞争。如直接使用互斥量，除了生产者、消费者之间要竞争互斥量以外，消费者之间也需要竞争互斥量，但如果汇聚(链表)中没有数据消费者之间竞争互斥量是无意义的。有了条件变量机制以后，只有生产者完成生产，才会引起消费者之间的竞争。提高了程序效率。

场景:一个线程产生随机数放入链表中，一个线程从链表中取出一个随机数打印

```

pthread_mutex_t mtx;
pthread_cond_t cond;
// 生产者消费者模型
// 场景:一个线程产生随机数放入链表中，一个线程从链表中取出一个随机数打印
// 定义结构体
typedef struct Node
{
    int number;
    struct Node* next;
}Node;
// 定义头节点
Node* head;
// 添加节点的函数
Node* AddNode(int number)
{
    // 申请新的节点
    Node* new_node = (Node*)malloc(sizeof(Node));
    // 将传入的数据，给到节点
    new_node->number = number;
    // 将指针域指向头节点
    new_node->next = head;
    // 将头节点指向申请的节点
    head = new_node;
    return head;
}

```

```

}
// 获取头结点元素的函数
int PopNode()
{
    if(head == NULL)
    {
        return -1;
    }
    Node* temp = head;
    int number = temp->number;
    head = temp->next;
    free(temp);
    return number;
}

// 生产者生成随机数，添加到链表
void* producer(void* arg)
{
    while(1)
    {
        pthread_mutex_lock(&mtx); // 加锁
        int number = rand() % 500 + 1; // 生成随机数
        AddNode(number); // 添加节点
        pthread_cond_signal(&cond); // 条件信号，添加完节点后，通知消费者读取数据
        pthread_mutex_unlock(&mtx);
        sleep(1);
    }
    return NULL;
}

// 消费者，取出头节点的随机数
void* consumer(void* arg)
{
    while(1)
    {
        pthread_mutex_lock(&mtx);

        while(head == NULL)
        {
            // 如果链表为空，阻塞等待，并释放持有的锁，让生产者去加锁，添加数据
            pthread_cond_wait(&cond, &mtx);
        }
        int num = PopNode();
        printf("num = %d\n", num);
        pthread_mutex_unlock(&mtx);
    }
    return NULL;
}

int main(int argc, char* argv[])
{
    // 设置随机种子
    srand(time(NULL));
    // 初始化读写锁
    pthread_mutex_init(&mtx, NULL);
    pthread_cond_init(&cond, NULL);
    pthread_t ptid1;
    pthread_t ptid2;
    pthread_create(&ptid1, NULL, producer, NULL);
    pthread_create(&ptid2, NULL, consumer, NULL);
    pthread_join(ptid1, NULL);
    pthread_join(ptid2, NULL);
    free(head);
    return 0;
}

```

## 6.4 信号量

信号量是操作系统提供了一种协调共享资源访问的方法。信号量是由内核维护的整型变量(sem)，它的值表示可用资源的数量。

互斥量就是可用资源数量为一的信号量

## 对信号量的原子操作:

### P 操作:

如果有可用资源( $\text{sem} > 0$ ), 占用一个资源( $\text{sem} - 1$ )

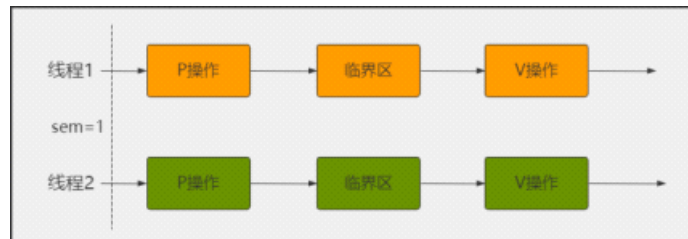
如果没有可用资源( $\text{sem} = 0$ ), 进程或线程阻塞, 直到有资源

### V 操作:

如果没有进程或线程在等待资源, 释放一个资源( $\text{sem} + 1$ )

如果有进程或线程在等待资源, 唤醒一个进程或线程

P 操作和V操作成对出现, 进入临界区前进行P操作, 离开临界区后进行V操作



POSIX提供两种信号量, 命名信号量和无名信号量, 命名信号量一般是用在进程间同步, 无名信号量一般用在线程间同步。

## 命名信号量和无名信号量通用函数:

```
int sem_wait(sem_t*sem);
```

P操作, 信号量大于零将信号量减一。否则进程阻塞

```
int sem_post(sem_t*sem);
```

V操作, 有阻塞进程会唤醒阻塞进程。否则信号量加一

```
int sem_getvalue(sem_t*sem, int *sval);
```

## 无名信号量函数:

```
int sem_init(sem_t*sem, int pshared, unsigned int value);
```

sem: 要进行初始化的信号量

pshared: 等于0用于同一进程下多线程的同步

pshared: 大于0用于多个相关进程间的同步(即fork产生的)

ovalue: 信号量的初始值

```
int sem_destroy(sem_t* sem);
```

## 命名信号量函数:

```
sem_t* sem_open(const char * name, int oflag, mode_t mode, unsigned int value);
```

打开或创建信号量

参数 name: 信号量名字

参数 oflag:

oflag为 0: 打开信号量

oflag为 O\_CREAT: 如果 name 不存在就创建一个信号量

oflag为 O\_CREAT | O\_EXCL: 如果 name 存在, 会失败

参数mode和value: 参数 oflag有 O\_CREAT时, 需要传这两个参数, mode 代表权限value 代表信号量的初始值。

返回值: 指向 sem\_t值的指针, 后续通过这个指针操作新打开或创建的信号量。

```
sem_close(sem_t*sem);
```

```
int sem_unlink(const char*name);
```

## 例子1: a自增

// 无名信号量-自增1000000次

```
sem_t s;
```

```
int a = 0;
```

```
void* thread1(void* arg)
```

```
{
```

```
    for(int i = 0; i < 1000000; i++)
```

```
    {
```

```
        sem_wait(&s); // P操作
```

```
        a++;
```



```

        sem_post(&s); // v操作
    }
    return NULL;
}
int main(int argc, char* argv[])
{
    // 无名信号量
    // 0 : 同一进程下的信号量
    // 1 : 信号量的初始值
    sem_init(&s, 0, 1);
    pthread_t ptid1;
    pthread_t ptid2;
    pthread_create(&ptid1, NULL, thread1, NULL);
    pthread_create(&ptid2, NULL, thread1, NULL);
    pthread_join(ptid1, NULL);
    pthread_join(ptid2, NULL);
    printf("a = %d\n", a);
    return 0;
}

```

## 例子2：0-1交替

```

// 01交替—信号量
/*
    这里加锁是对 ++和一操作加锁，所以需要两个信号量
    而不是对a进行加锁。
    如果是对a进行加锁，那么++a之后，就自动解锁了，还能接着++a，就不是0-1交替了
    正确的做法是
    加操作的信号量（s_add）初始值为1，可以进行++a。（因为a的初始值为0，得先进行加操作）
    减操作的信号量（s_sub）初始值为0，开始就阻塞，等加操作结束。

    此时加操作进行p操作。s_add = 0;之后就不能进行加操作了。
    ++a结束之后，对s_sub 进行v操作，s_sub = 1;
    此时 加操作阻塞，减操作的信号量为1，可以执行减操作。减操作信号量进行p操作，s_sub = 0;
    --a结束之后，对s_add 进行 v操作，s_add = 1; 阻塞的加操作可以执行。
*/
sem_t s_add; // 加操作信号量
sem_t s_sub; // 减操作信号量
int a = 0;
void* add(void* arg)
{
    while(1)
    {
        sem_wait(&s_add); // 加操作执行 p 操作，s_add = 0;
        printf("a = %d\n", ++a); // ++a
        sleep(1);
        sem_post(&s_sub); // 减操作执行 v 操作，s_add = 1;
    }
    return NULL;
}
void* sub(void* arg)
{
    while(1)
    {
        sem_wait(&s_sub); // 减操作执行 p 操作，s_sub = 0;
        printf("a = %d\n", --a);
        sleep(1);
        sem_post(&s_add); // 加操作执行 v 操作，s_add = 1;
    }
    return NULL;
}
int main(int argc, char* argv[])
{
    sem_init(&s_add, 0, 1); // 加操作信号量 初始值为1
    sem_init(&s_sub, 0, 0); // 减操作信号量 初始值为0
    pthread_t ptid1;

```

```

pthread_t ptid2;
pthread_create(&ptid1, NULL, add, NULL);
pthread_create(&ptid2, NULL, sub, NULL);
pthread_join(ptid1, NULL);
pthread_join(ptid2, NULL);
return 0;
}

```

例子3:信号量实现生产者消费者模型，生产者线程产生随机数存入数组，消费者线程从数组取出一个随机数打印。

```

// 例子3:信号量实现生产者消费者模型,
// 生产者线程产生随机数存入数组, 消费者线程从数组取出一个随机数打印。
sem_t arr; // 数组的信号量
sem_t s; // 消费者的信号量
int num_Arr[3]; // 数组
void* producer(void* arg)
{
    int index = 0;
    while(1)
    {
        sem_wait(&arr); // 数组信号量加锁
        int rand_num = rand()%500; // 产生随机数
        num_Arr[index%3] = rand_num; // 赋值
        printf("producer num_Arr[%d] = %d\n", index%3, num_Arr[index%3]);
        index++; // 改变下标
        sleep(1);
        sem_post(&s);

    }
    return NULL;
}
void* consumer(void* arg)
{
    int index = 0;
    while(1)
    {
        sem_wait(&s);
        printf("consumer num_Arr[%d] = %d\n", index%3, num_Arr[index%3]);
        num_Arr[index%3] = -1;
        index++;
        sleep(1);
        sem_post(&arr);
    }

    return NULL;
}
int main(int argc, char* argv[])
{
    srand((unsigned int)time(NULL));
    sem_init(&arr, 0, 3);
    sem_init(&s, 0, 0);
    pthread_t ptid1;
    pthread_t ptid2;
    pthread_create(&ptid1, NULL, producer, NULL);
    pthread_create(&ptid2, NULL, consumer, NULL);
    pthread_join(ptid1, NULL);
    pthread_join(ptid2, NULL);
    sem_destroy(&arr);
    sem_destroy(&s);
    return 0;
}

```