



APUE 总结

秋招快速复习之 Linux 系统编程



刘洋洋

2018-7-15

CHINA
南三舍 425

目录

0.进程基础.....	3
0.1 写时复制技术.....	3
0.2 fork 函数与文件共享	5
0.3.exec 系列函数的本质	7
0.4 wait 系列函数与僵尸进程.....	7
1.信号	9
1.1、信号处理函数是如何被调用的.....	9
1.2、可重入函数.....	10
2. Linux IPC 常用手段.....	12
2.1、IPC 分类	12
2.2、无名管道 pipe	13
2.3、有名管道.....	14
2.4、System V 共享内存	14
2.41、shmat 函数原理.....	16
2.42、shmctl 函数.....	16
2.5、System V 消息队列.....	17
2.6、System V 信号量.....	18
3.进程间的关系.....	23
3.1 进程扇与进程链.....	23
3.2、进程组.....	23
3.21 控制终端.....	24
3.3.孤儿进程与孤儿进程组.....	25
3.31、孤儿进程.....	25
3.32、孤儿进程组.....	26
3.33、守护进程.....	26
4.线程基础.....	28
4.1、进程与线程的区别：	28
4.2、线程的创建与终止.....	29
4.3、线程的清理.....	29
5.线程互斥与同步.....	31
5.1、多线程互斥.....	31
5.11、互斥锁.....	31
5.12、读写锁.....	31
5.13、自旋锁.....	31
5.2、多线程同步.....	32
5.21、线程同步.....	32
6.线程控制.....	36
6.1 线程属性.....	36
6.11 互斥量的共享属性.....	36
6.12 线程安全函数/可重入函数	37
6.13 多线程与信号.....	38
6.14 多线程与 fork	40

7.高级 IO-多路复用.....	41
7.1 问题的引入.....	41
7.2 fd_set 容器.....	41
7.3 select 函数.....	42
7.4 poll 函数.....	43
1、struct pollfd 类型.....	44
2、timeout 参数.....	44
3、poll 和 select 区别.....	45
4、epoll 函数.....	45
8.高级 IO-同步 IO 与异步 IO	46
8.1 引导	47
8.2 POSIX 异步 IO 概念	47
遗留问题:	48
8.21、异步 IO 的 aiocb 结构（类似于 PCB 进程控制块）	48
8.22、异步通知.....	49
9、网络编程基础.....	50
9.1 字节序.....	50
9.11 网络字节序.....	50
9.2 IPv4 地址.....	51
9.3 地址和端口号.....	51

0. 进程基础

0.1 写时复制技术

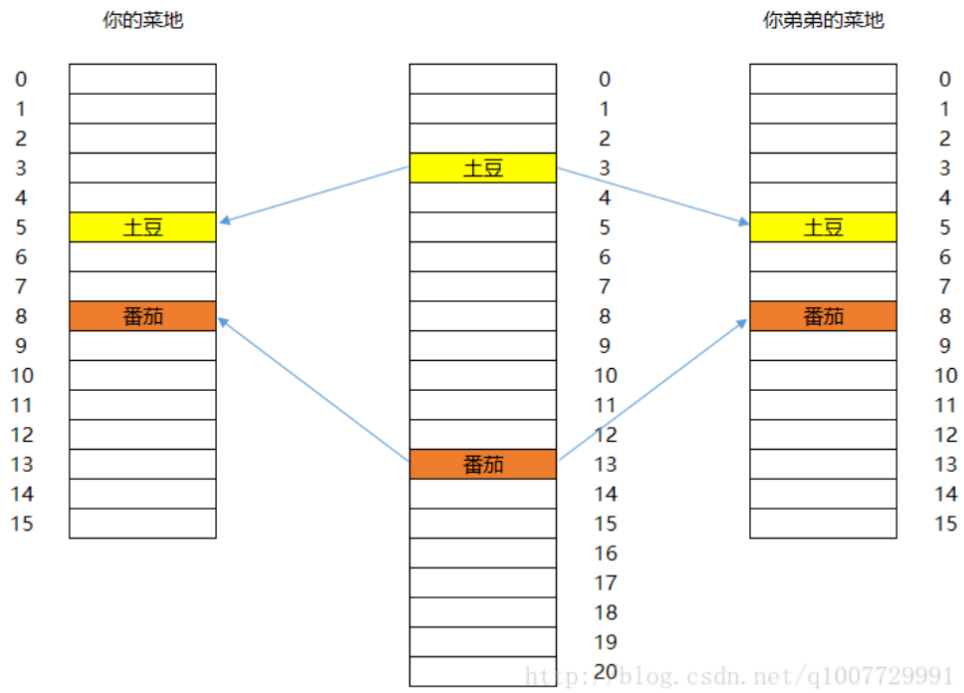


图6 读时共享

0.2 fork 函数与文件共享

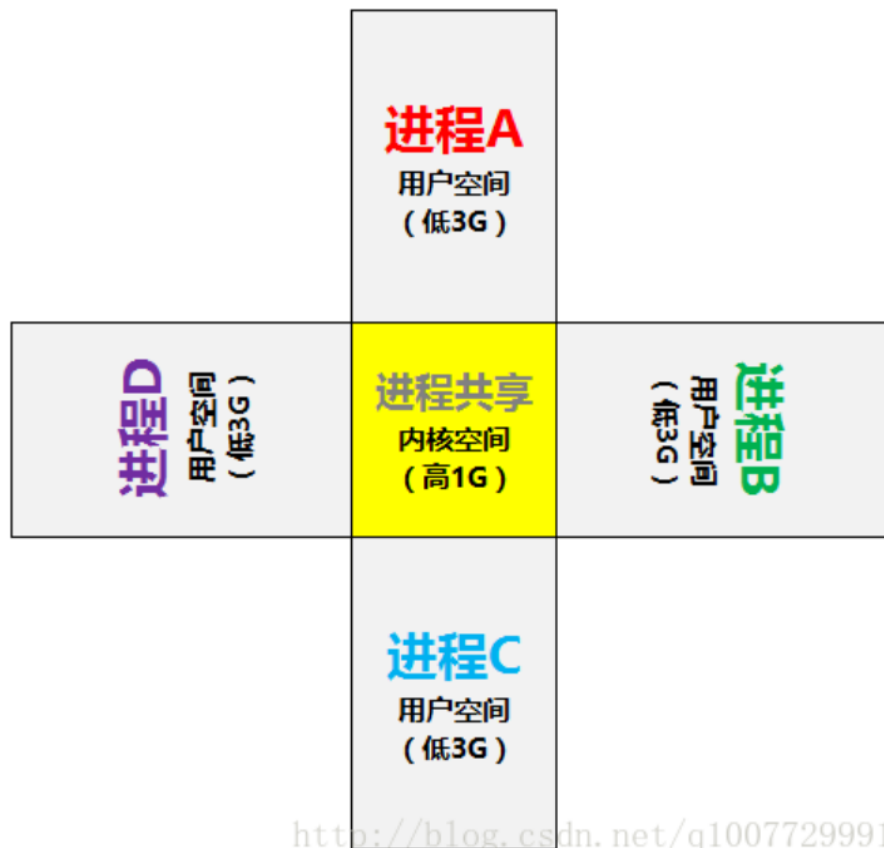


图 1 进程的用户空间和内核空间

进程 4GB 空间并不是完全隔离的。

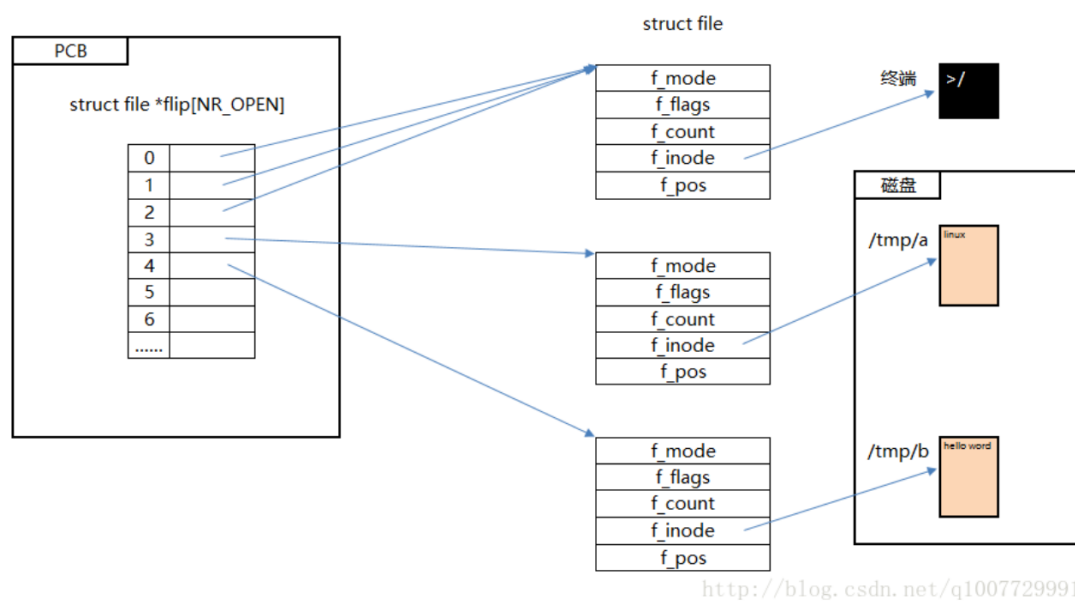
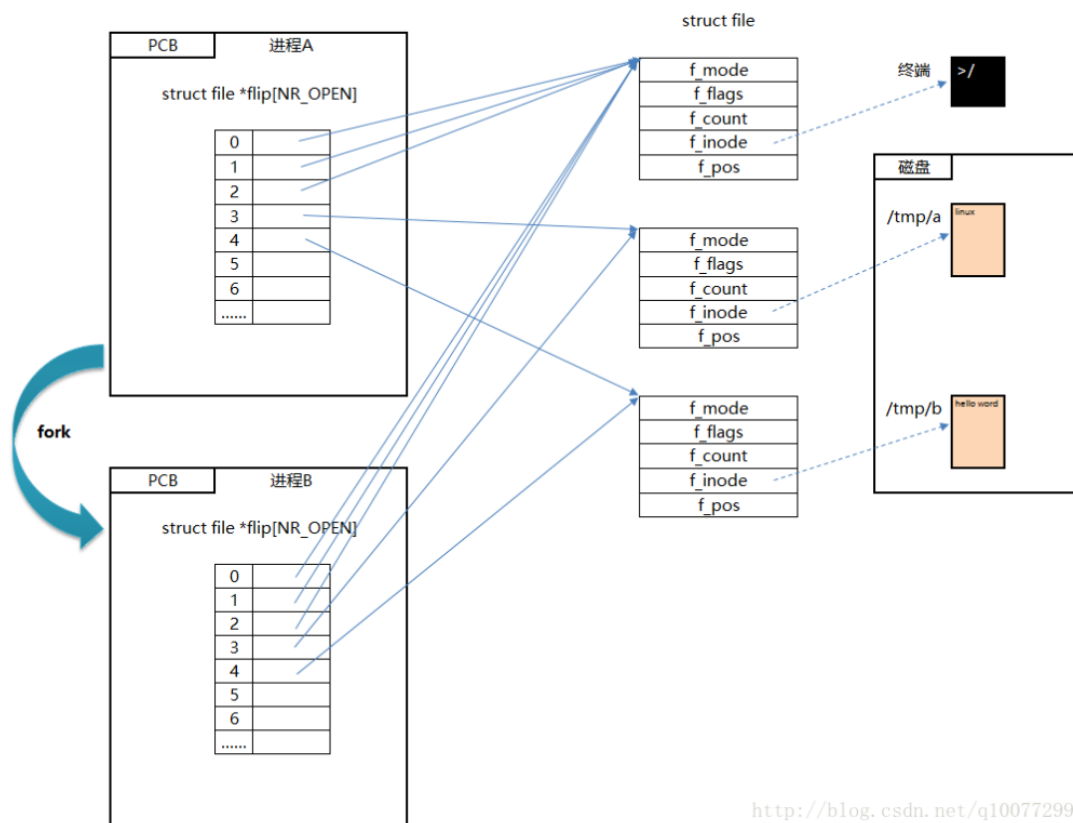


图2 进程打开的文件



<http://blog.csdn.net/q1007729991>

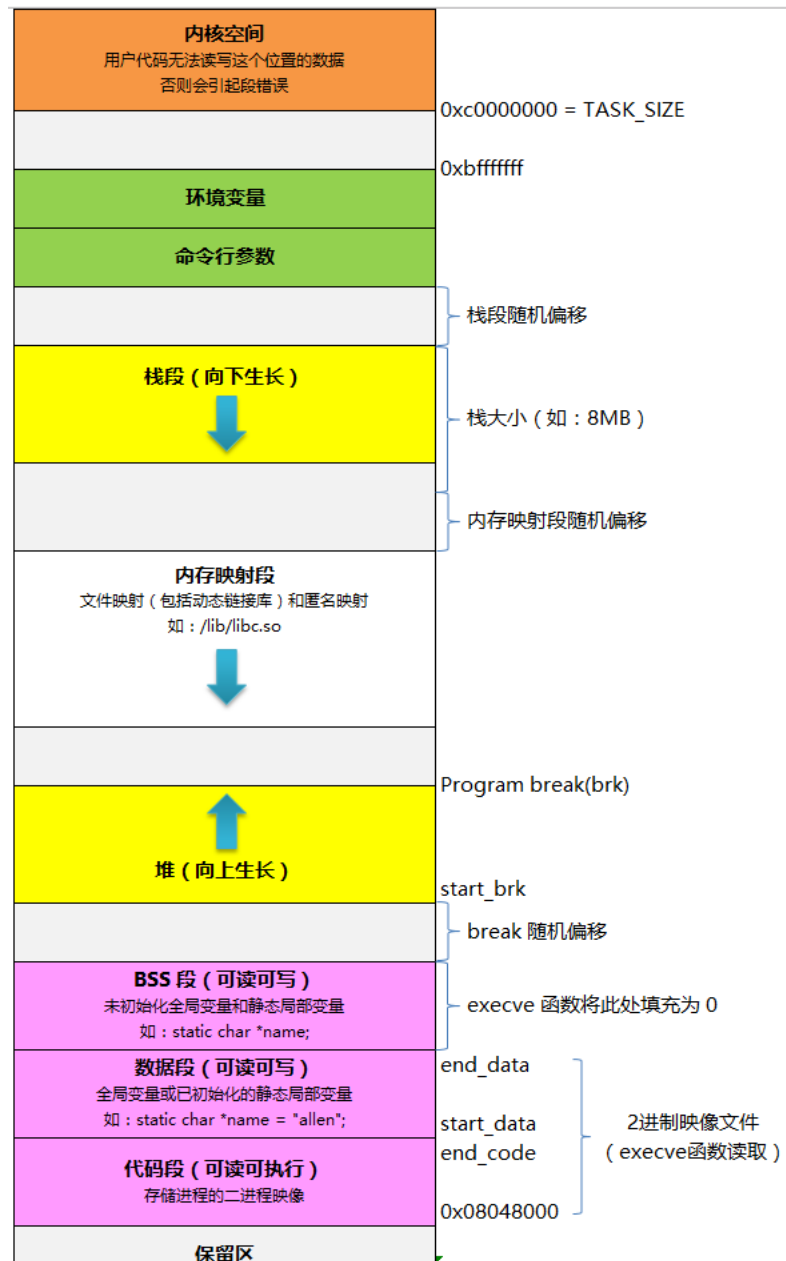
图3 fork 后的两个进程打开的文件

这时候，`struct file` 中的 `f_count` 都会自增 1。

父子进程共享 `struct file` 结构，因为该结构在内核空间。

注意：除了打开的文件外，父进程的很多其他性质也会被子进程共享，比如各种 ID 号、当前工作目录、根目录、资源的限制、信号屏蔽字、进程环境、文件打开执行时关闭标志、共享存储段。

0.3.exec 系列函数的本质



进程的地址空间。

exec 系列函数的本质就是替换代码段和数据段，然后从新的入口点重新执行。对应到图 2 中，就是替换整个粉红色的区域。

0.4 wait 系列函数与僵尸进程

如果子进程运行结束了而父进程还没有，就会产生僵尸进程

子进程在死的时候会通知父亲（给父进程发送 SIGCHLD 信号）。

所以父进程只要妥善处理好子进程的通知就行了，需要用到的函数就是 `wait`。

// 参数保存子进程退出通知码，返回 `-1` 表示没有子进程或者错误。否则返回子进程的进程 `id` 号。

```
pid_t wait(int *status);
```

只要子进程发生了变化，它就会通知父进程（信号），父进程可以选择处理该信号或者忽略信号。

不过当子进程在正常退出时给父进程发送信号，如果父进程还忽视，子进程自然就变僵尸了。

增强版的 `wait` 函数：`pid_t waitpid(pid_t pid, int *status, int options);`

`options` 的三个可组合选项如下（可以理解成开关组合）：

- `WNOHANG` (设置非阻塞，即使子进程全部正常运行，`waitpid` 也会立即返回 0)
- `WUNTRACED` (可获取子进程暂停状态，也就是可获取 `stopped` 状态)
- `WCONTINUED` (可获取子进程恢复执行的状态，也就是可获取 `continued` 状态)

1. 信号

1.1、信号处理函数是如何被调用的

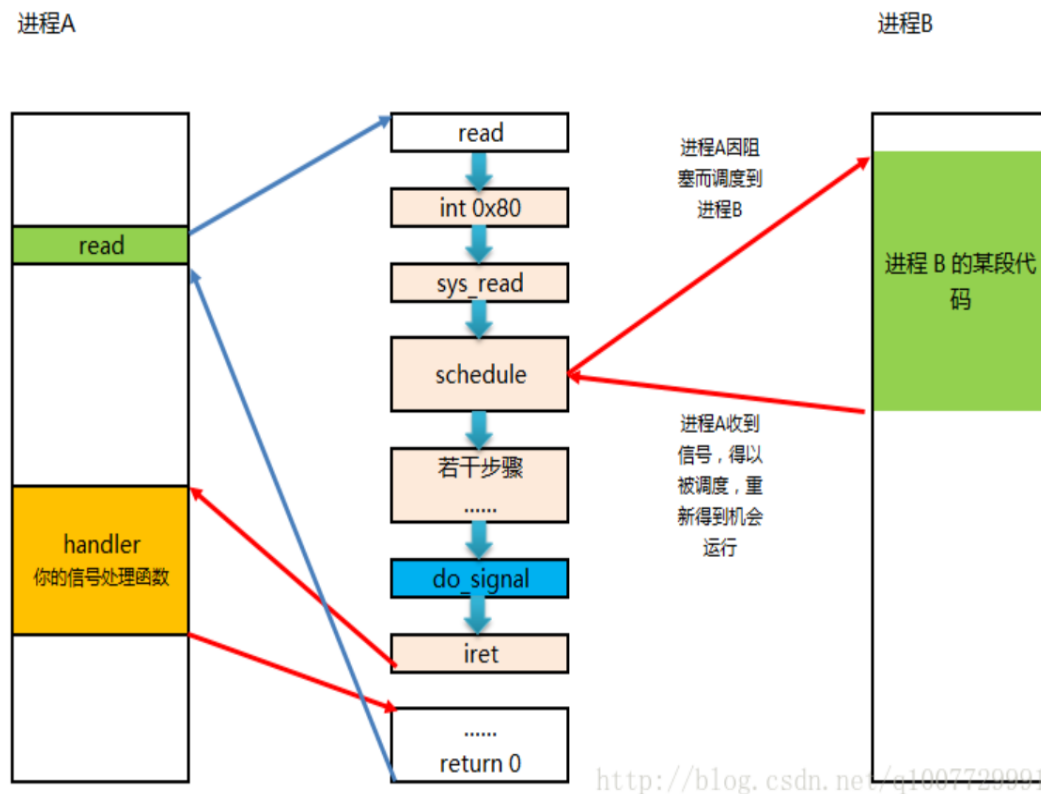


图2 信号处理函数是如何被调用的

下面具体分析下执行流程：

- 进程 A 执行到 `read` 里的时候（题外话，本质上 `read` 函数只是个外壳而已，真正的系统调用是 `sys_read`，这已经位于内核了），然后执行了中断指令 `int 0x80`，进入到了内核态。
- 接下来执行真正的系统调用 `sys_read`，由于请求的资源没有到来，`sys_read` 调用了 `schedule` 函数，这是真正的进程调度函数。进程调度函数执行到某个位置后，找到了下一个要运行的进程（进程 B），然后切换进程上下文，转而执行进程 B。
- 进程 B 运行到某个位置，也发生了和进程 A 一样的遭遇，也调用了某个系统调用而阻塞而切换到了进程 A

注意：如果进程 A 的资源没有准备好，同时进程 A 没有收到信号是不会被调度到的。只要进程 A 收到信号了，同时该信号未被阻塞，进程 A 就会被调度到，资源没有准备好也没关系。

- 当再次回到进程 A 的时候，进程 A 接着从 `schedule` 函数切到进程 B 那行代码的下一行继续执行，完成 `sys_read` 的功能。`sys_read` 完成后，准备退出内核态。
- 在退出内核态前，会执行 `do_signal` 函数检查是否有信号，如果有信号 `do_signal` 会找到信号对应的信号处理函数（你使用 `signal` 或者 `sigaction` 注册的）。
- 接下来执行 `do_signal`
 - 如果进程没有收到信号，`do_signal` 函数直接返回，执行 `iret` 后直接返回到 `read` 函数。图中没有画线条，因为这种情况比较简单。
 - 执行如果进程收到了信号，`do_signal` 函数会修改用户栈，为信号处理函数创建一个函数执行现场，这将导致系统调用在执行 `iret` 函数的时候不是返回到 `read` 函数，而是返回到信号处理函数！！
- `do_signal` 函数创建完对应的信号处理函数的执行现场后，执行 `iret`，返回到了信号处理函数。信号处理函数执行完后，才返回到 `read` 函数。
- 接下来，`read` 函数返回。

实际上，系统进程内核态，除了执行系统调用，也可以通过时钟中断进入内核。只不过这种方式是被动的。进入时钟中断后，会检查进程时间片是否用完，如果用完，照样会执行 `schedule` 调度程序转移到别的进程。

1.2、可重入函数

不妨看下面的一个函数。

```
int a = 0; // 全局变量

int fun() {

    ++a;

    return a;
}
```

试想一下，当你在执行 `fun()` 函数的 `return a` 的时候（假设这时候 `a` 的值已经为 1），你的代码突然由于**信号的打断**而跳转到另一段代码运行。然而十分不巧的是，那段代码把 `fun` 函数执行了一遍（此时 `a` 的值已经变成了 2），当重新回到你的代码时，你的 `fun` 函数的返回值已经不再是你期望的 1，而是 2。

产生这种现象的本质在于，该函数引用了全局变量 `a`。

除此之外，使用静态局部变量也会出现这种问题。所以，我们把所有引用了**全局变量或静态变量**的函数，称为**不可重入函数**，不可重入函数都不是信号安全的，也不是线程安全的。（有关线程，后面会慢慢涉及）。

反过来说，如果一个函数对于信号处理来说是可重入的，则称其为**异步信号安全函数**。

注意：线程安全的函数，不一定是异步信号安全的。

有一点需要注意的是，如果一个函数使用了不可重入函数，那么该函数也会变成不可重入的。这意味着，你不能在信号处理函数中使用不可重入函数。

2. Linux IPC 常用手段

2.1、IPC 分类

最后总结一下，Linux 实现 IPC 主要有以下几种方式：

- 无名管道（**pipe**）及有名管道（**named pipe**）：无名管道只能用于有亲缘关系的进程，有名管道用于任意两进程间通信。
- 信号（**signal**）：上一个专题讲的很详细了。
- 消息（**message**）队列：包括 **Posix** 消息队列 **system V** 消息队列。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。
- 共享内存（**share memory**）：使得多个进程可以访问同一块内存空间，是最快的可用 IPC 形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。
- 信号量（**semaphore**）：主要作为进程间以及同一进程不同线程之间的同步手段。
- 套接口（**Socket**）：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由 **Unix** 系统的 **BSD** 分支开发出来的，但现在一般可以移植到其它类 **Unix** 系统上：Linux 和 **System V** 的变种都支持套接字。

2.2、无名管道 pipe

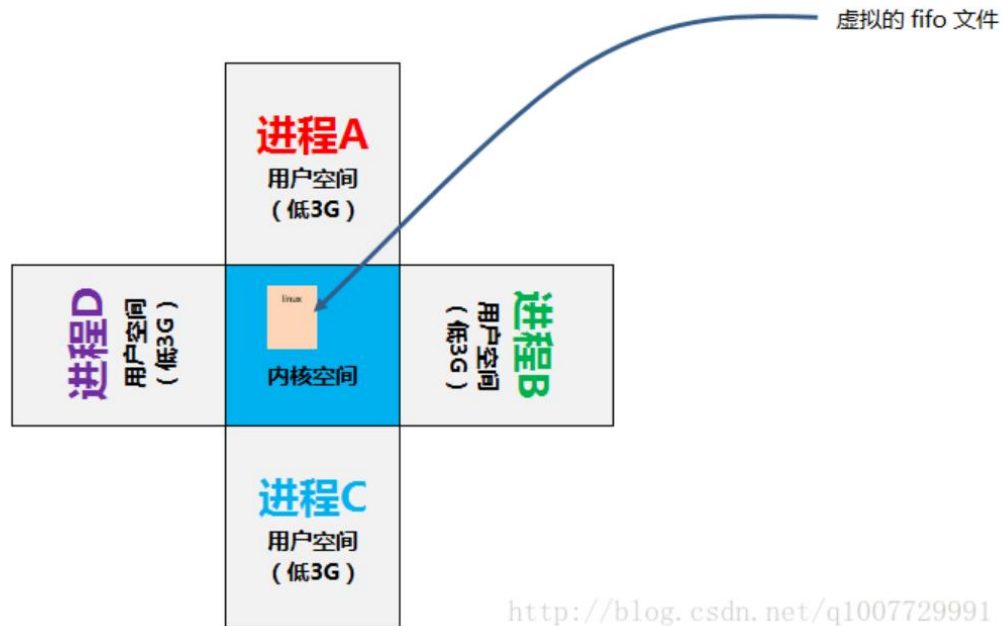
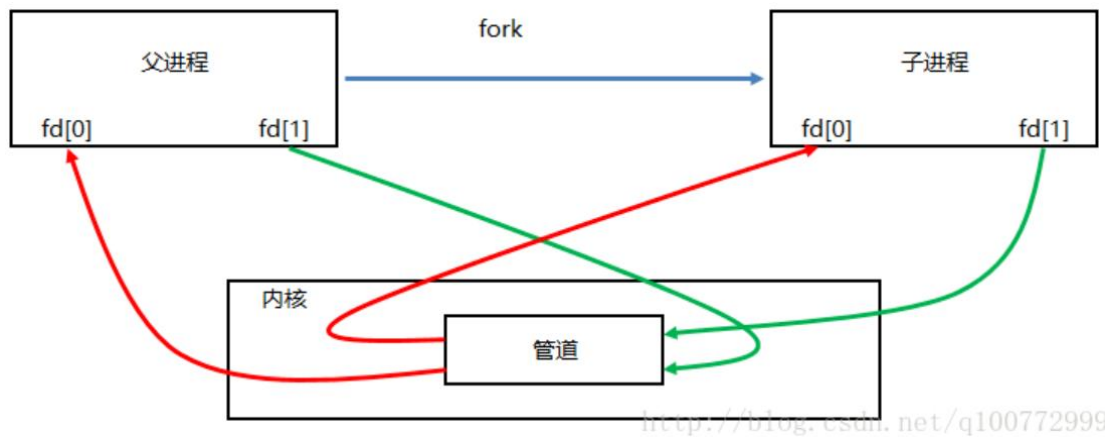


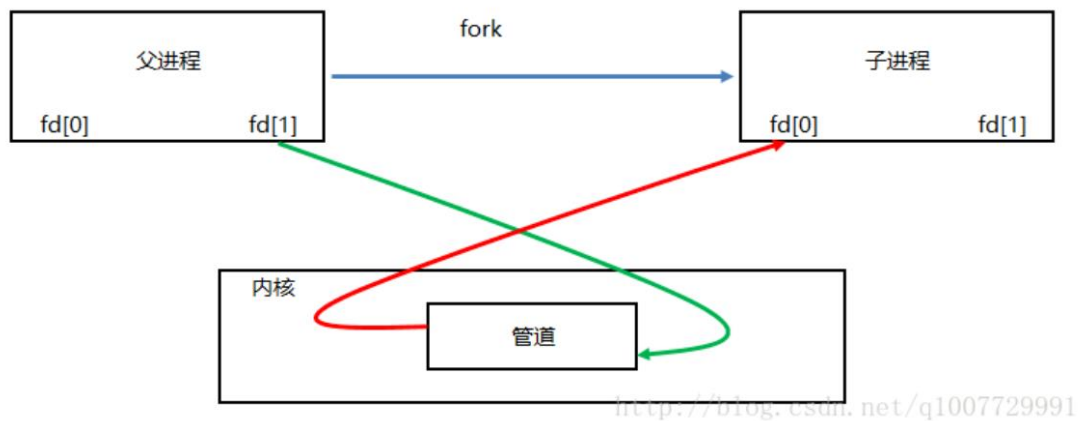
图1 假想的位于内核的管道文件

Pipe 使用图示举例:

- 步骤一: fork 子进程



- 步骤二: 关闭父进程读端, 关闭子进程写端



要想实现全双工通信（既能发又能收）怎么办？你可以再使用 `pipe` 打开两个描述符，或者使用其它的 `ipc` 手段。

2.3、有名管道

有名管道打破了无名管道的限制，进化出了一个实实在在的 **FIFO** 类型的文件。这意味着即使没有亲缘关系的进程也可以互相通信了。所以，只要不同的进程打开 **FIFO** 文件，往此文件读写数据，就可以达到通信的目的。

不过 **FIFO** 文件与我们最开始讲的本地文件通信还是有着本质的区别，它长着普通文件的脑袋，却有着无名管道的基因。

FIFO 文件特性：

- 文件属性前面标注的文件类型是 `p`，代表管道
- 文件大小是 0
- `fifo` 文件需要有读写两端，否则在打开 `fifo` 文件时会阻塞

2.4、System V 共享内存

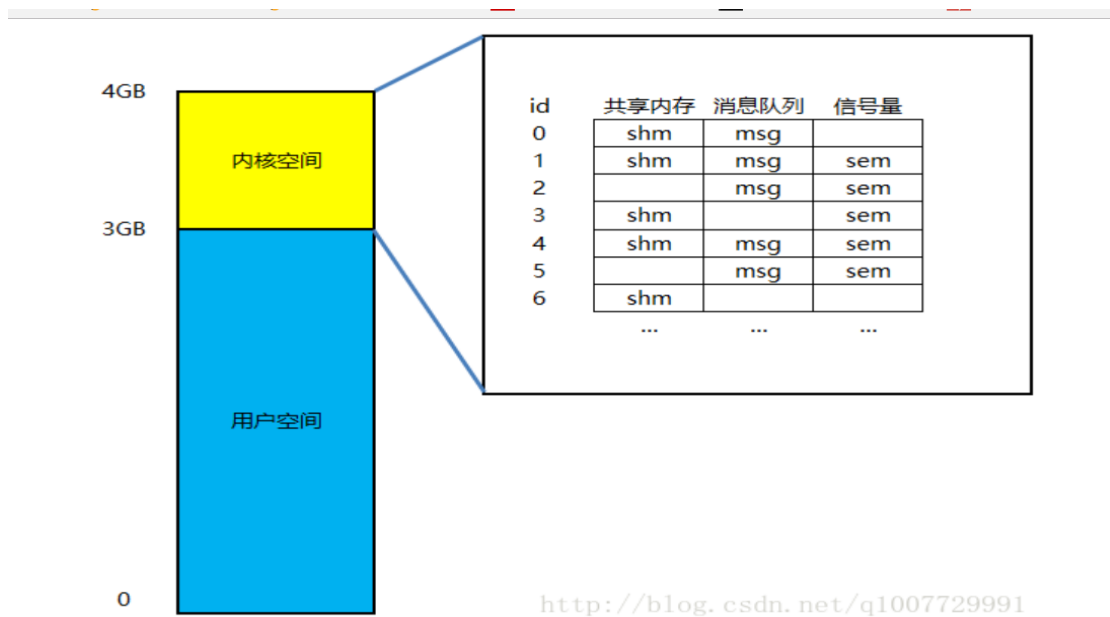


图1 IPC 内核对象示意图

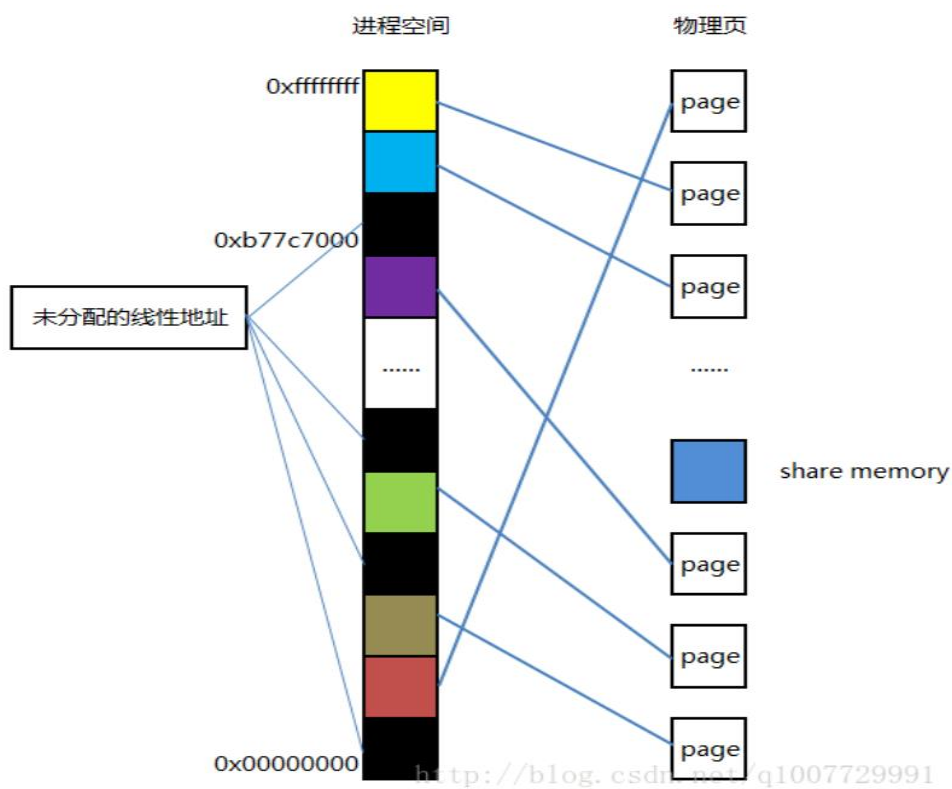


图1 进程空间与物理页

黑色部分表示的是未分配的线性地址。

当你使用 `shmget` 创建出共享内存后，系统就会为你分配一个（或多个）物理页，具体分配多少，就看你当时使用 `shmget` 第二个参数指定的大小了，一般来说，至少会分配一个物理页(4KB 大小)。分配完成后，操作系统就会在 System V 共享内存的内核对象里进行记录。

2.41、shmat 函数原理

刚刚分配完物理页后，当然无法直接使用，因为没有任何线性地址映射到这个物理页，所以 `shmat` 函数就是干这个事的，它从进程空间中选择一个合适的或者你自己指定的线性地址（根据 `shmat` 第二个参数是否为 0 来决定，如果是 0，系统自动选择），将其挂接到共享内存物理页上。

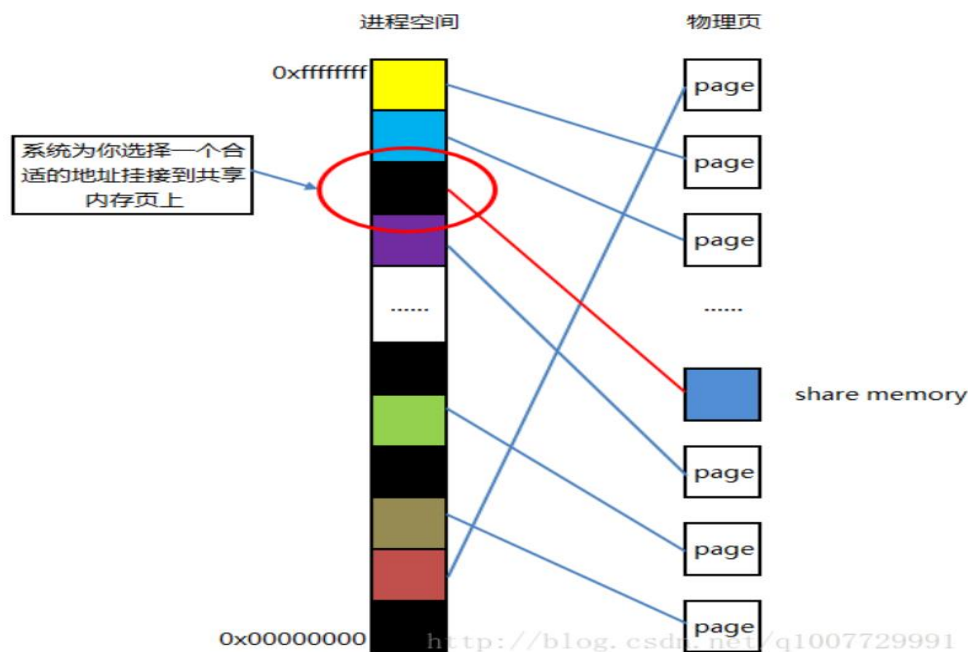


图2 shmat 函数将线性地址挂接到物理页

`shmat` 函数完成挂接后会返回那个位置的线性地址，这时候，用户就可以往这里存取数据了。`shmat` 的第三个参数，`shmflg` 用来控制读写权限，如果指定为 0，表示可读写，如果指定为 `SHM_RDONLY`，表示只读。

如果你理解了这个过程，`shmdt` 自然就解惑了。它所做的操作只是和 `shmat` 相反罢了。

2.42、shmctl 函数

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

`shmid_ds` 结构体

有时候，我们需要获取 `IPC` 内核对象的相关信息，这时候要怎么办呢？毕竟 `ipc` 内核对象它位于内核空间中，普通用户程序根本无法直接访问，为此 `linux` 提供了这样的结构体 `shmid_ds` 给用户态的程序使用。`shmctl` 函数可以将位于内核空间的 `ipc` 内核对象的信息拷贝一份给用户空间的 `shmid_ds`，也可以通过将此结构体中的信息拷贝到内核空间用来设置 `ipc` 内核对象，具体是哪种情况依赖于参数 `cmd`。

2.5、System V 消息队列

在有了共享内存的基础后，消息队列对你来说就是 A piece of cake 了。同样我们需要创建消息队列的内核对象，获取它的 id，然后操控它。

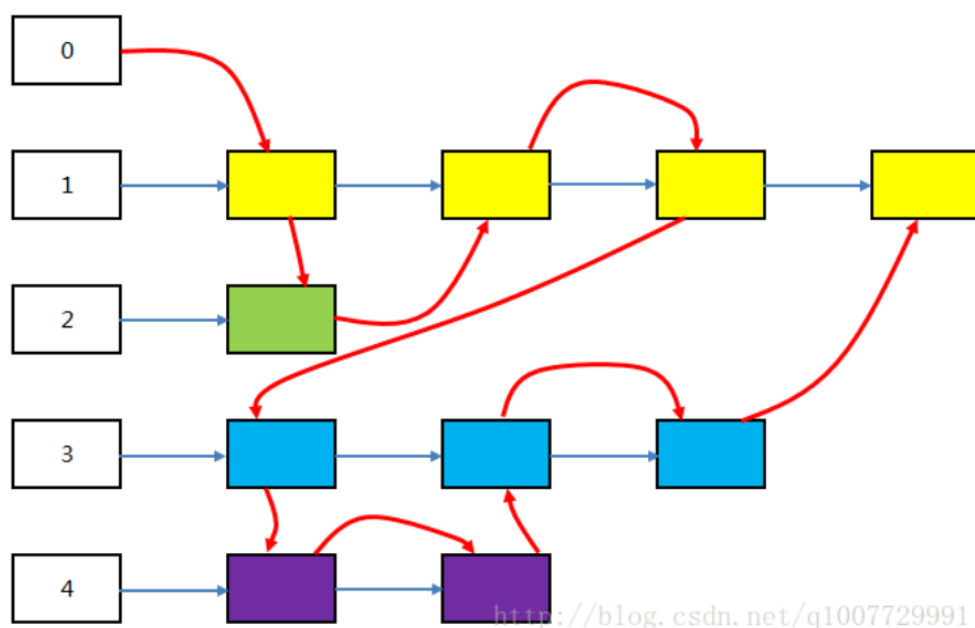


图1 位于内核空间的消息队列

其中数字 1 表示类型为 1 的消息，数字 2、3、4 类似。彩色块表示消息数据，它们被挂在对应类型的链表上。

值得注意的是，刚刚说过没有消息类型为 0 的消息，实际上，消息类型为 0 的链表记录了所有消息加入队列的顺序，其中红色箭头表示消息加入的顺序。

```
// 创建和获取 ipc 内核对象
int msgget(key_t key, int flags);
// 将消息发送到消息队列
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
// 从消息队列获取消息
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
// 查看、设置、删除 ipc 内核对象（用法和 shmctl 一样）
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

消息数据的格式：

```
struct Msg{
```

```

    long type; // 消息类型。这个是必须的，而且值必须 > 0，这个值被系统使用
    // 消息正文，多少字节随你而定
    // ...
}

```

只要你保证首 4 字节(32 位 linux 下的 long)是一个整数就行了

```

Msg msg[10] = {
    {1, {"Luffy", 17}},
    {1, {"Zoro", 19}},
    {2, {"Nami", 18}},
    {2, {"Usopo", 17}},
    {1, {"Sanji", 19}},
    {3, {"Chopper", 15}},
    {4, {"Robin", 28}},
    {4, {"Franky", 34}},
    {5, {"Brook", 88}},
    {6, {"Sunny", 2}}
}; 注意类型的对应

```

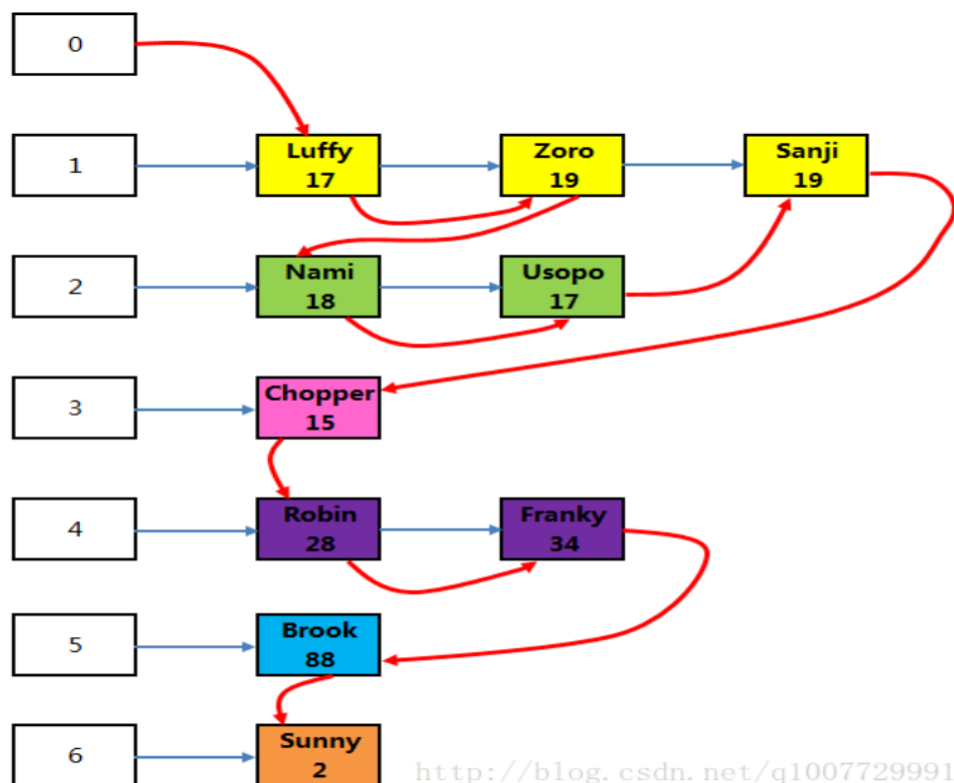


图1 第一次执行完 `msg_send` 后的消息队列

2.6、System V 信号量

不同于 System V 共享内存和消息队列，这两者都是用于交换数据，而 System V 信号量通常用来做进程同步（与互斥量不同，互斥量是互斥）。

信号量就做了类似的工作，它可以让一系列的指令要么一次全部执行完成，要么一条都不执行。

生产者与消费者模型

PV 原语

- **P(S)**表示将资源 **S** 的数量减 1， $S=S-1$ ；如果 $S \leq 0$ ，则该进程进入等待状态
- **V(S)**表示将资源 **S** 的数量加 1， $S=S+1$ ；如果 $S > 0$ ，则进程继续执行，否则释放一个正在等待的进程

上面的资源 **S** 是信号量，是非负整数，互斥量只能为 0/1；

PV 操作是原子操作，原子操作的意思是一次性执行若干条指令，中间不能被打断。如果一个操作不是原子的，就可能会产生竞态错误。

生产者与消费者问题描述

假设有一个缓冲区，该缓冲区最多只能存放 5 个蛋糕。有若干生产者(producer)进程，每个生产者一次生产一个蛋糕，如果发现缓冲区有空位，就将蛋糕放进去。另外还有若干消费者(consumer)进程，每个消费者如果发现缓冲区有蛋糕，就拿走一个蛋糕。

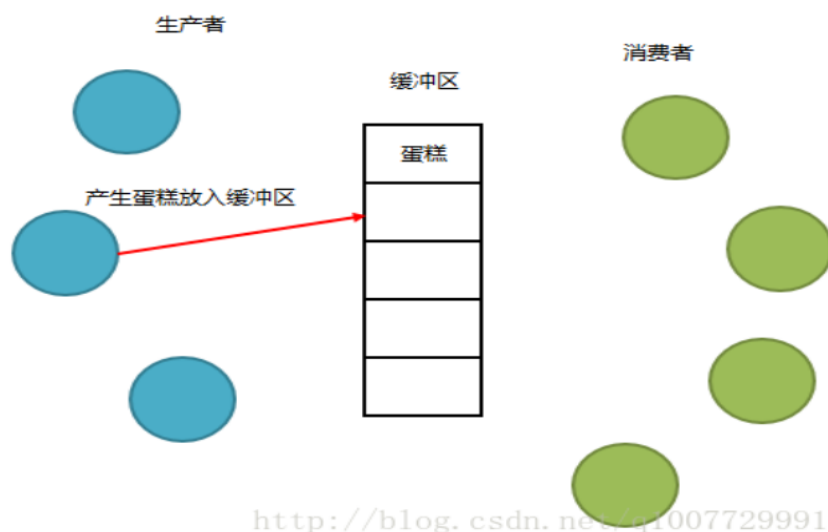


图1 3 个生产者和 4 个消费者

这里的 **cake** 明显是竞争资源，在任意一个时刻，只能有一个进程去操作它

- 生产者进程

```
while(1) {
```

```

P(MUTEX);

if (cake < 5) {

    cake++;

}

V(MUTEX);

}

```

- 消费者进程

```

while(1) {

    P(MUTEX);

    if (cake > 0) {

        cake--;

    }

    V(MUTEX);

}

```

上面改进的代码里引入了信号量 **MUTEX**，用来表示资源 **cake** 是否被占用，**MUTEX** 的初始值为 1。上面的程序就可以达到对 **cake** 互斥访问的目的了。

信号量 **MUTEX** 的初始值为 1，它是内核对象，所有进程共享的！所以会产生以上的互斥效果（要理解为什么信号量也被叫做进程间通信的机制，实际上只是进程间共享信号量，从而通过信号量对公共访问区域进行加锁！）

改进使 CPU 的效率更高

我们希望做到，如果蛋糕数为 0，消费者进程就去睡觉吧！别再被操作系统调度了，什么时候有蛋糕了，再唤醒它们！

这里再次引入信号量 **FULL** 和 **EMPTY**。**FULL** 表示蛋糕的个数，初始值为 0。**EMPTY** 表示空缓冲区的个数，初始值为 5。

改进前面的程序：

- 生产者进程

```
while(1) {  
  
    P(EMPTY); // 减少一个空缓冲区个数  
  
    P(MUTEX);  
  
    if (cake < 5) {  
  
        cake++;  
  
    }  
  
    V(MUTEX);  
  
    V(FULL); // 增加一个蛋糕个数
```

- 消费者进程

```
while(1) {  
  
    P(FULL); // 减少一个蛋糕个数  
  
    P(MUTEX);  
  
    if (cake > 0) {  
  
        cake--;  
  
    }  
  
    V(MUTEX);  
  
    V(EMPTY); // 增加一个空缓冲区个数  
  
}
```

如果用来描述蛋糕个数的信号量 $FULL \leq 0$ 了, 消费者执行到 $P(FULL)$ 就会立即被投入等待状态, 不再被 OS 调度。什么时候 $FULL > 0$ 了, 才可能会被 OS 调度。

3. 进程间的关系

3.1 进程扇与进程链

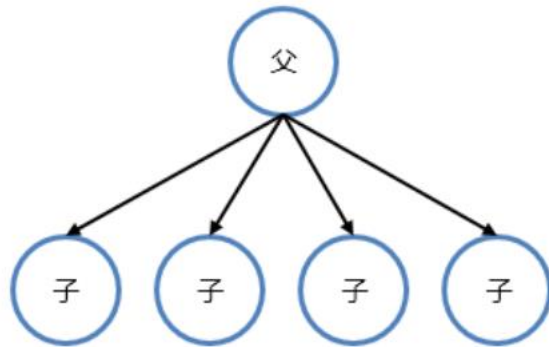


图1 进程扇

按照进程扇的定义，子进程是没有子进程的。



图4 进程链

因为按照进程链定义，一个父进程只能产生一个子进程。

3.2、进程组

- 进程组的生命期

从被创建开始，到其内所有进程终止或离开该组。

- 信号与进程组

使用 `kill -n -pgid` 可以将信号 `n` 发送到进程组 `pgid` 中的所有进程。例如命令 `kill -9 -4115` 表示杀死进程组 `4115` 中的所有进程。

进程组可以没有组长。（被杀死）

前面已经学过，进程组是一个或多个进程的集合。而会话(session)，是一个或多个进程组的集合。

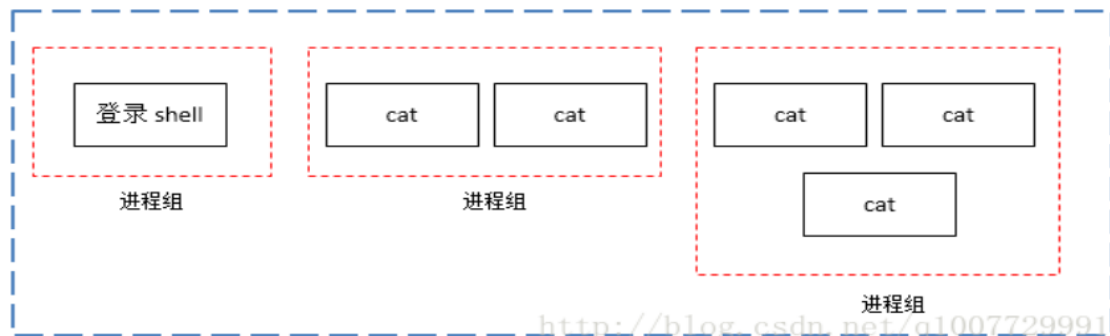


图1 进程组和会话中的进程安排

图 1 中的会话，可以使用以下命令形成：

```
$ cat | cat &
```

```
$ cat | cat | cat
```

3.21 控制终端

一般来说，在 `bash` 中运行的进程，都会伴随着一个控制终端。

控制终端是进程的属性，它保存在进程 `PCB` 中。而 `bash` 进程在启动的时候，会自动建立一个控制终端。所以由 `bash` 进程启动(fork)的进程都会继承控制终端。所以在同一个会话中，控制终端都是一样的。

如果一个会话有控制终端，则它有一个**前台进程组**，以及多个**后台进程组**。

当在终端输入或者终端产生信号时，都会发送到前台进程组

执行下面的命令：

```
$ cat | cat &
```

```
$ cat | cat | cat
```

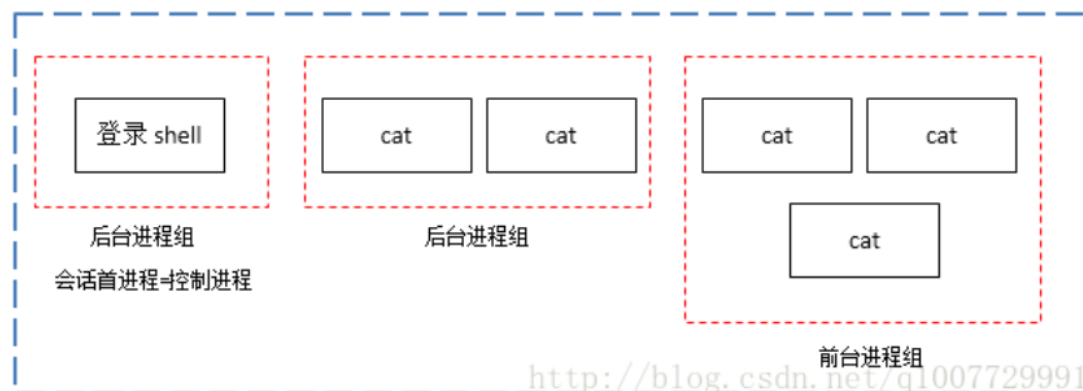


图2 前台进程组，后台进程组，控制进程

如果一个会话有控制终端，那么该会话就有一个前台进程组。此时在终端中的输入以及终端产生的信号默认都发送给前台进程组中的所有进程。

前台进程组是可以通程序进行设置的。在同一个会话中，调用 `tcsetpgrp` 函数可以设置某个进程组为前台进程组。

3.3.孤儿进程与孤儿进程组

之前有学习过僵尸进程，它指的是“未老先衰”的子进程先于父进程去逝，而父进程未对它进行回收(wait)所产生的。

本文来谈谈，另一个对立的观念——孤儿进程，以及孤儿进程组。

3.3.1、孤儿进程

如果一个进程，它的父进程先终止了，它就会变成孤儿进程。此后，该进程的父进程变成 `init` 进程（1 号进程）

需要注意的是，孤儿进程不像僵尸进程，它是无害的，也不需要回收。

3.32、孤儿进程组

进程组中的每一个进程的父进程要么在本进程组，要么在其他会话。

该进程组的每个成员的父进程要么是该组的成员，要么在其它会话中。

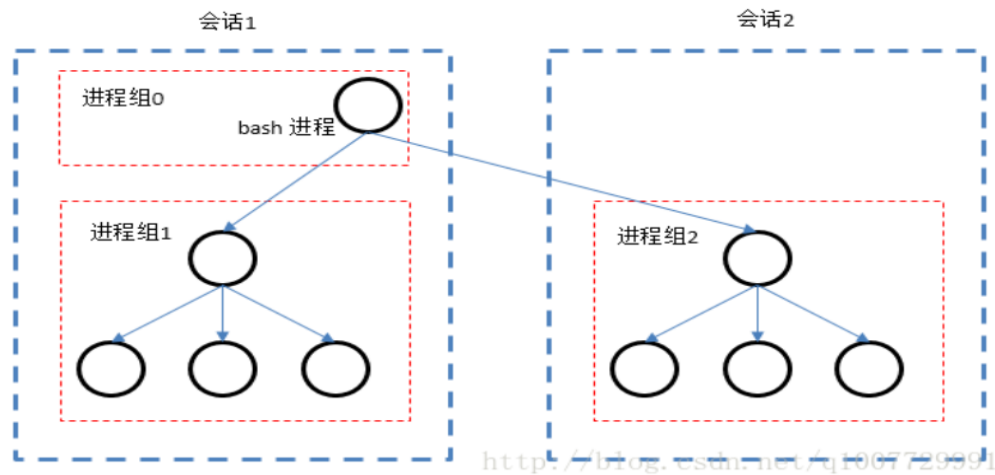


图 1 非孤儿进程组与孤儿进程组

图 1 中，按照孤儿进程组的定义，进程组 1 不是孤儿进程组，因为进程组 1 中有一个进程的父进程不属于进程组 1，也不在另一个会话中。

进程组 2 是孤儿进程组，因为该组中的每个成员满足定义：每个成员的父进程要么在本组中，要么在其它会话中。

3.33、守护进程

在我们之前写的大多数程序中，如果你关闭控制终端，就很可能导致在该控制终端中运行的进程也退出。

控制终端关闭时，该控制终端符合下面条件的进程会收到 **SIGHUP** 信号：

- 前台进程组的所有进程

某个所以，并不是所有情况下控制终端中的进程都会收到 **SIGHUP** 信号。

有时候，我们希望进程不和控制终端挂钩，即使你关闭终端对它也没有任何影响。另外我们不希望这种进程向终端读写数据（需要关闭标准输入、标准输出和标准错误），它会一直在后台运行着，像这种进程，称之为守护进程，或者叫精灵进程（**daemon process**）。

- 后台进程组中存在**停止的进程**（ps ajx 命令中，进程状态显示为 T）。控制终端关闭，导致该后台进程组**成为孤儿进程组**，则该孤儿进程组中的所有进程都收到 SIGHUP 信号。

4. 线程基础

4.1、进程与线程的区别：

当一个进程被创建的时候，自动就有一个线程在运行，该线程就是主控线程，简称主线程（也就是 `main` 函数运行的那个线程）。我们可以在主线程里调用创建线程的函数 `pthread_create`，当然你也可以在其它线程里调用线程创建函数。


我们需要修正一下以前我们对进程的理解。一直以来我们认为进程就是程序的执行体，实际上，真正的执行体是进程中的线程。进程被创建出来，自动的就包含了一个主线程，真正的执行体是这个线程。

所以，进程是一种空间上的概念，它为所有这些执行体（线程）提供必要的资源（内存、文件描述符、代码等）。而线程，是时间上的概念，它是抽象的、假想的、动态的指令执行过程。

我们可以把进程理解成工厂以及工厂里的各种设备和资源，而线程就是工厂里一个个干活的工人。

线程的资源：

在同一个进程中，大部分资源都是所有线程所共享的，比如：

- 
- 可执行程序的代码
 - 程序的全局内存
 - 堆内存
 - 栈
 - 文件描述符

但是线程还有一些独有的资源，包含：

- 线程 ID
- 线程自己的一套寄存器值
- 线程运行栈
- 调度优先级和策略
- 信号屏蔽字
- `errno` 变量
- 线程私有数据

4.2、线程的创建与终止

线程创建:

```
typedef void *(*start_routine) (void *);
```

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

线程终止:

分为主动终止和被动终止。

- 主动终止
 - 线程过程函数执行 `return` 正常返回 返回值是线程的退出码
 - 线程过程函数执行 `pthread_exit` 函数退出，其参数是线程退出码
- 被动终止
 - 在其它线程中调用 `pthread_cancel` 函数
 - 任意线程调用 `exit`、`_Exit` 或者 `_exit` 导致整个进程终止

4.3、线程的清理

有时候我们希望线程退出时能够自动的执行某些函数，为了能达到此目的，OS 提供了两个函数帮我们完成这个功能：

```
void pthread_cleanup_push(void (*rtn)(void*), void *arg);
```

```
void pthread_cleanup_pop(int execute);
```

如果想要你的线程在退出时能够执行清理函数，你需要使用 `pthread_cleanup_push` 对你的清理函数进行注册，如下：

```
void clean(void *arg) {  
    // ...  
}
```

```
void *th_fn(void *arg) {  
    // push 和 pop 必须对对出现  
    pthread_cleanup_push(clean, // 清理函数 clean 的参数);  
    // ...  
    pthread_cleanup_pop(1);  
}
```

线程清理函数的调用时机：

有三种情况线程清理函数会被调用：

- 线程还未执行 `pthread_cleanup_pop` 前，被 `pthread_cancel` 取消
- 线程还未执行 `pthread_cleanup_pop` 前，主动执行 `pthread_exit` 终止
- 线程执行 `pthread_cleanup_pop`，且 `pthread_cleanup_pop` 的参数不为 0.

注意：如果线程还未执行 `pthread_cleanup_pop` 前通过 `return` 返回，是不会执行清理函数的。

- 清理函数的执行顺序，是按照注册时候相反的顺序执行的。

5. 线程互斥与同步

多线程会产生竞态错误

5.1、多线程互斥

- 互斥锁 mutex
- 读写锁 rwlock
- 自旋锁 spinlock

互斥所要做的事，即我做某件事的时候你不允许做，而你做某件事的时候也不允许我做。

5.11、互斥锁

为了确保同一时间只有一个线程访问数据，在访问共享资源前需要对互斥量上锁。一旦对互斥量上锁后，任何其他试图再次对互斥量上锁的线程都会被阻塞，即进入等待队列。

```
lock(&mutex);  
// 访问共享资源  
unlock(&mutex);
```

上锁函数我们一般使用阻塞版本的，非阻塞版本的会浪费 CPU 资源

5.12、读写锁

为了提高读并发！

分为：读加锁状态，写加锁状态，不加锁状态！

注意：有读的人进行读加锁后，想写的人不能进行写加锁，且想写的人后面的想读的人，也不能进行读加锁了，这是为了防止想写的线程处于一种饥饿的状态（长时间不能加写模式的锁）！

注意读写锁和互斥锁的区别！以及读写锁是如何防止写饥饿的！

5.13、自旋锁

自旋锁是一种非阻塞锁，也就是说，如果某线程需要获取自旋锁，但该锁已经被其他线程占用时，该线程不会被挂起，而是在不断的消耗 CPU 的时间，不停的试图获取自旋锁。

互斥量是阻塞锁，当某线程无法获取互斥量时，该线程会被直接挂起，该线程不再消耗 CPU 时间，当其他线程释放互斥量后，操作系统会激活那个被挂起的线程，让其投入运行。

两种锁适用于不同场景：

1、如果是多核处理器，如果预计线程等待锁的时间很短，短到比线程两次上下文切换时间要少的情况下，使用自旋锁是划算的。

2、如果是多核处理器，如果预计线程等待锁的时间较长，至少比两次线程上下文切换的时间要长，建议使用互斥量。

3、如果是单核处理器，一般建议不要使用自旋锁。因为，在同一时间只有一个线程是处在运行状态，那如果运行线程发现无法获取锁，只能等待解锁，但因为自身不挂起，所以那个获取到锁的线程没有办法进入运行状态，只能等到运行线程把操作系统分给它的时间片用完，才能有机会被调度。这种情况下使用自旋锁的代价很高。

4、如果加锁的代码经常被调用，但竞争情况很少发生时，应该优先考虑使用自旋锁，自旋锁的开销比较小，互斥量的开销较大。

5.2、多线程同步

同步与互斥稍稍不同，具体差别在于两个字——顺序。意思是说，只有当我做完了某件事 A，你才能做另一件事 B。

互斥不关心顺序，谁先做 A，谁先做 B，都行。实际上在互斥中 A，B 是同一件事。

5.21、线程同步

5.211、轮询实现线程同步

问题：

学生线程写作业，老师线程检查作业。要求：只有学生线程写完作业了，老师线程才能检查作业。

在此问题中，有两个线程：学生线程和老师线程，和以往的线程互斥不一样的是，线程互斥

之间没有明确的执行顺序上的要求。而线程同步，有了顺序上的要求，即有先后关系：只有学生线程完成了作业以后，老师线程才能够去运行！

在没有信号量和条件变量的时候，如何使用互斥来完成线程同步

轮询是指：老师线程可以不断的去询问学生作业到底有没有完成作业。

具体做法是采用全局变量 `finished` 做标记，初始化为 0，表示学生还未完成作业。学生如果完成了作业，会将 `finished` 变量设置为 1。

另一方面，老师线程不断的检查 `finished` 变量是否为 1 来判断学生完成了作业。

很显然，全局变量 `finished` 属于共享资源，需要使用互斥手段对其进行互斥访问，否则容易出现错误。

轮询效率低，极度浪费 CPU 资源

5.212、条件变量实现线程同步

Pthread 库为线程同步提供了一种机制---条件变量。它允许线程在睡眠的情况下等待特定条件的发生，从而被唤醒。

等待条件发生的函数：

// 阻塞版本

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

函数语义：

假设线程 B 为学生写作业线程，线程 A 为老师批改作业线程。

当 A 询问 B 作业是否做完后，B 如果没有做完，A 将进入等待队列，并将互斥量 `mutex` 解锁（因为 A 询问 B 时对互斥量加锁了，那么 A 走的时候要解锁以供 B 可以继续使用写作业），红色划线部分的两步是原子操作！

在所有进入等待队列的线程，都会等待一个条件变量 `cond` 的成立。所谓 `cond` 条件成立，意思是其他线程（B 写完作业）通过 `pthread_cond_signal` 函数，将等待队列中的某一个线程（A）唤醒，A 被唤醒后，会再次对传入的 `mutex` 变量加锁！

5.213、死锁的理解：

注意：上面的 A 在进入等待队列之前如果不释放锁，那么 B 就无法使用锁，那么 B 就无法访问临界区，那么 B 就一直无法去唤醒 A，A 一直在等待队列等待，死锁。

`pthread_cond_wait` 为什么需要传入互斥量 `mutex` (最后的解释应该是对的, 下面的解释貌似是解释 `pthread_cond_wait` 内部实现需要互斥量):

如果把 `pthread_cond_wait` 函数分成 3 步,

// 线程 A 中

`pthread_mutex_unlock(&lock);` // a1

// 询问 B, B 没有完成

`pthread_cond_wait(&cond);` // a2

// 既然 B 没有完成我就进入到等待队列并释放锁

`pthread_mutex_lock(&lock);` // a3

// B 把 A 唤醒后, A 将互斥量加锁, 要进行批改作业了。。。。

假设上面语句 a1 执行完后, 线程 B 被调用, B 进入临界区 (锁已经被释放) 去执行代码, 如果 B 完成作业, 然后执行 `pthread_cond_signal` 函数给等待队列发信号, 但是此时 A 还没有进入等待队列, B 已经执行完了, 然后 A 执行 a2 进入等待队列, 以后 B 也不会再给 A 发信号了, 那么 A 就永远等不到条件的到来, 将永远休眠。

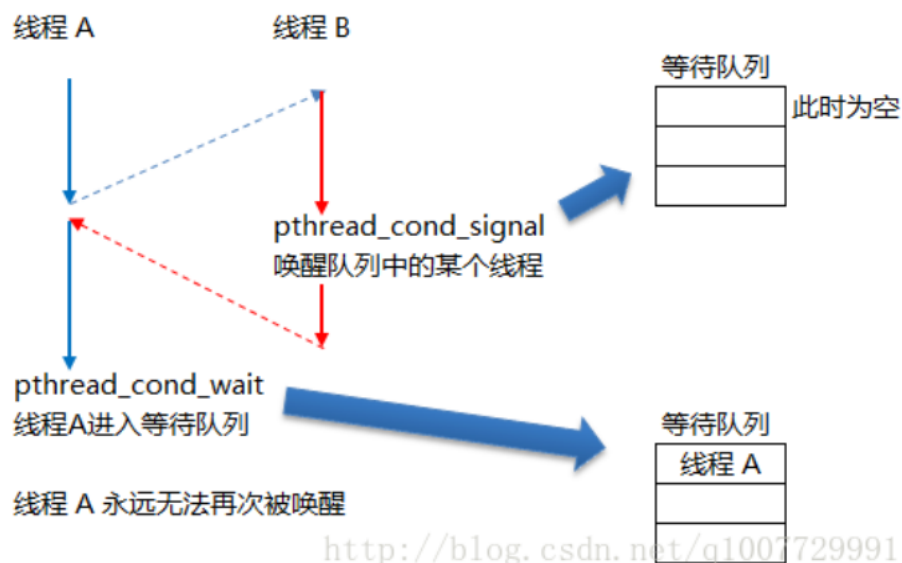


图1 未加锁时对条件变量的访问导致线程 A 永远等待下去

所以说, 语句 a1, 和语句 a2, 一定要是原子操作, 即一次性执行完。

在介绍 `pthread_cond_wait` 的资料中常说传入的这个 `mutex` 是用来保护“条件变量”的, 但是这个条件变量并不是你传入的那个“`pthread_cond_t`”类型的变量。

因为用户写程序的时候通常是发现当前一些变量的值比如说变量 a 并不满足期待的条件，所以选择调用 `cond_wait` 把当前线程挂起，期待别的线程修改 a 的值。变量 a 自然是多个线程之间共享的，所以本意是让你用这个 mutex 保护 a 的，而不是说 `pthread_cond_wait` 的内部实现需要用这个 mutex 保护什么。

5.213、屏障 barrier

直接说它的用途吧：barrier 就是管理员，只有参观者达到了一定的人数，比如 30 人，管理员才允许放行。

换作线程，请求 barrier 的线程会阻塞，直到所有请求 barrier 的线程达到指定的数量。

`pthread_join`，本质上它也是一个 barrier，只不过它只需要等待一个线程运行结束后就返回。

barrier 的数据类型是 `pthread_barrier_t`。

- 初始化和回收

```
int pthread_barrier_init(pthread_t *barrier,  
  
    const pthread_barrier_t *attr, unsigned int count);  
  
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

- 等待函数

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

每一个线程调用等待函数，barrier 计数都会加 1，直到计数值等于初始化函数中的参数 count。当最后一个线程让 barrier 计数值达到了 count，所有等待的线程都会被唤醒。

所有线程中，只有一个线程会让 `pthread_barrier_wait` 返回 `PTHREAD_BARRIER_SERIAL_THREAD`，而其它线程调用此函数会返回 0。

6. 线程控制

6.1 线程属性

一直以来创建线程的时候，都将线程属性参数 `pthread_attr_t` 设置为 `NULL`。通常来说这都能满足需求。本文将简要介绍 `pthread_attr_t` 中包含的属性。

`pthread_create` 函数其中一个参数为此类型，表示将创建指定类型的线程，通常，该类型是一个结构体，包含下面这些属性：

- `Detach state`: 线程的分离属性
- `Scope`: 线程竞争域
- `Inherit scheduler`: 调度继承性
- `Scheduling policy`: 调度策略
- `Scheduling priority`: 线程优先级
- `Guard size`: 警戒缓冲区大小，默认是一个 `PAGE_SIZE(4096KB)`
- `Stack address`: 栈起始地址（栈最低内存地址）
- `Stack size`: 栈大小

我们只关心线程的分离属性：`Detach state`

分离属性是指线程结束的时候，是否回收资源（比如它使用的栈，线程结构体，线程过程函数的返回值等）。

分离属性有两种情况：

- `PTHREAD_CREATE_JOINABLE`: 这种情况是默认的，表示结束的时候不回收资源，需要调用 `pthread_join` 函数显式回收，这种情况我们早已经学会了。
- `PTHREAD_CREATE_DETACHED`: 表示线程结束了，直接释放所占用的资源，这种情况下不可再使用 `pthread_join` 函数去 `wait` 它了。

6.11 互斥量的共享属性

用于线程互斥的互斥量也有相应的属性 `pthread_mutexattr_t`，这里只讨论三个方面：

- 共享属性

- 鲁棒属性
- 互斥量的递归类型

共享属性：

除了互斥量有共享属性外，其它的线程互斥同步对象如读写锁、自旋锁、条件变量、屏障都有共享属性。

该属性有两种情况：

- `PTHREAD_PROCESS_PRIVATE`：这种是默认的情况，表示互斥量只能在本进程内部使用。
- `PTHREAD_PROCESS_SHARED`：表示互斥量可以在不同进程间使用。

对于第一种情况，我们早已经学会。第二种情况，需要结合前面的进程间通信技术才有用。一般需要在共享内存中分配互斥量，然后再为互斥量指定 `PTHREAD_PROCESS_SHARED` 属性就可以了。

鲁棒性：健壮性

递归类型的互斥锁：

一般情况下，我们在同一个线程中对同一个互斥量加两次锁，就会死锁。如果将互斥量类型属性设置为递归类型 `PTHREAD_MUTEX_RECURSIVE` 就不会出现此问题。

递归互斥量内部维护着一个计数器，当互斥量未上锁时，计数器值为 0。只有计数器为 0 的情况下，线程才能够获得锁。只有获得锁的线程，才能持续对互斥量加锁，每加一次锁，计数器的值加 1，每解一次锁，计数器的值减 1。

- 信号处理函数中，避免使用互斥量，如果确实要使用，使用递归类型互斥量

6.12 线程安全函数/可重入函数

线程函数分为：

- 对线程可重入
- 对信号可冲入

1.1 对线程可重入

如果一个函数对线程重入是安全的，则称其为**线程安全函数**。它的意思是说，在相同的时间点，可以被多个线程安全地调用。这种函数的特征：

- 不包含全局变量或静态变量
- 如果包含全局变量或静态变量，需要加互斥锁

1.2 对信号可重入

如果一个函数对异步信号处理程序的重入是安全的，则称其为**异步信号安全函数**。它的意思是说，在相同时间点，可以在执行信号处理函数的时候，可被新的信号打断，再次进入信号处理函数。

这一种，就是在信号章节中所讲的可重入函数，当时规定，它里面不能有全局的或者是静态变量。即它具备下面的特征：

- 不包含全局变量或静态变量

对比线程安全函数，我们可以看到异步信号安全函数少了一条规则，就是不允许出现互斥锁。信号处理函数中，不推荐使用互斥量（容易死锁），如果真的要，只能使用递归类型的互斥量。

注意：线程安全函数不一定是异步信号安全函数

errno 变量是多线程安全的

线程的私有变量：线程私有容器中的键值对

6.13 多线程与信号

回顾进程：

进程的阻塞信号集与未决信号集

它们在进程的 PCB 结构中

阻塞信号集可以被修改，未决信号集不能被修改

阻塞信号集：当某一个进程不想收到某一个信号时，可以将该信号加入到阻塞信号集。阻塞

信号集的概念是：一个信号加入阻塞信号集，该信号的信号处理函数就不会被调用。

未决信号集：说的通俗点就是你的进程已经收到信号，只是还没有被信号处理函数处理的那些函数。

当收到一个信号，且该信号被阻塞，则该信号进入未决信号集，当该信号被信号处理函数处理，则该信号就从未决信号集中被删除。

上面的是进程中的信号，下面是将线程的信号，注意区别

6.131 多线程程序中的信号

在每一个线程中都有自己的阻塞信号集与未决信号集。当一个线程派生另一个线程时，派生出来的线程会继承父线程的阻塞信号集，但是会将未决信号集清除。

- 每个线程有自己的阻塞信号集与未决信号集
- 线程会继承父线程的阻塞信号集，新线程会清空未决信号集
- `pthread_kill` 发送信号给指定线程
- `kill` 发送信号给所有线程

6.132 sigwait 函数

如果未决信号集中没有信号，我们不会一直去取信号，这样太浪费 CPU。

如果在线程中调用 `sigwait` 函数，线程会一直等待未决信号集中出现它所等待的信号。一旦该信号出现，`sigwait` 函数就会将它取出并返回。并将该信号从未决信号集中删除。

如果多个线程调用 `sigwait` 函数，只有一个线程会从 `sigwait` 中返回。

如果 `sigwait` 等待的信号被捕获，要么 `sigwait` 函数返回，要么被信号处理函数处理。

那么这种带有 `sigwait` 的信号处理方式（等待特定信号）与传统的信号处理方式有何区别？

采用这种方式的好处在于防止信号打断线程。安排专用的线程处理函数，不需要担心信号处理程序中哪些函数是安全的，哪些是不安全的。因为这些函数调用都来自于同一个线程上下文，而非传统的会中断线程正常执行流程的信号处理程序。

6.14 多线程与 fork

在多线程程序中使用 `fork`，可能会导致一些意外：

- 子进程中只剩下一个线程，它是父进程中调用 `fork` 的线程的副本构成。这意味着在多线程环境中，会导致“线程蒸发”，莫名其妙的失踪！
- 因为线程蒸发，它们所持有的锁也可能未释放，这将导致子进程在获取锁时进入死锁。

解决死锁：

初始的解决方案是在 `fork` 前先请求所有的锁，然后再 `fork`，另外我们希望 `fork` 完成后，再对所有锁进行解锁。

6.141 `atfork`

```
int pthread_atfork(void (*prepare)(void), void (*parent)(void), void (*child)(void));
```

上面三个回调函数，分别为 `prepare` 函数，`parent` 函数，`child` 函数。这些函数的调用时机如下：

- `prepare` 函数是在 `fork` 还会产生子进程时调用
- `parent` 和 `child` 是在产生子进程后调用
- `parent` 在父进程中调用，`child` 在子进程中调用

利用这个函数可以解决 `fork` 死锁问题，原理与[上面](#)说的一样

7. 高级 IO-多路复用

7.1 问题的引入

IO 多路传输，将多个通道（比如文件描述符，在 socket 中是多个套接字连接）合并为一个通道（将多个描述符放在一个数组里进行监管查询）进行查询（select）（可以避免使用多进程，多线程，也没有使用非阻塞 IO）

7.2 fd_set 容器

fd_set 类型是利用整型数组实现的，每个元素中的每个 bit 位被置 1 就表示该位置保存了描述符，如果为 0 就表示没有该描述符。

arr[0]								arr[1]								...				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	...

图1 一种 fd_set 的实现

在图 1 中，数组类型是 char，占 8 个 bit，可以看到，在第 5 位和第 14 位中 bit 位被置 1，说明这个 fd_set 中保存了 5 号描述符和 14 号描述符。

OS 提供四个宏函数来增删查和初始化。

```
#include <sys/select.h>
// 判断描述符 fd 是否在集合中
int FD_ISSET(int fd, fd_set *fdset);

// 将描述符 fd 从集合中删除
int FD_CLR(int fd, fd_set *fdset);

// 将描述符 fd 添加到集合中
int FD_SET(int fd, fd_set *fdset);

// 将集合清空（所有 bit 置 0）
int FD_ZERO(fd_set *fdset);
```

7.3 select 函数

7.3.1 select 基础

```
int select(int nfd, fd_set *readfds, fd_set *writefds,  
           fd_set *exceptfds, struct timeval *timeout);
```

1、select 参数

这里的 select 函数提供的参数比较多，其中有三个参数都是 fd_set 类型，先讲这个：

- readfds，你想监听这个集合中的描述符中是否有数据可读。
- writefds，你想监听这个集合中的描述符是否可写，写 IO 当然也会发生阻塞，比如缓冲区满了。
- exceptfds，你想监听这个集合中的描述符是否发生异常，有两种情况，一种常见的是在网络编程中出现，另一种不太常见的是在伪终端中出现。

参数 nfd 表示，在传入参数的那三个集合中，最大的描述符的值 + 1。比如 readfds 中保存了 1、3、5 号描述符，writefds 中保存了 2、3、6，exceptfds 为空，那么 nfd 就应该设置成最大的描述符的值(6) + 1，也就是 nfd = 7。

最后一个参数是等待时间，如果传空，表示永远等待，直到指定的三个集合中的描述符有事件（有数据可读、有数据可写、有异常事件发生）到来。

如果不传空值，则表示最长愿意等待多久。

它的结构如下：

```
struct timeval {  
  
    long    tv_sec;        /* 秒 */  
  
    long    tv_usec;       /* 微秒 */  
  
};
```

如果给三个集合都设置成空，`nfds = 0`，只给时间参数传值，那么 `select` 函数就相当于一个加强版本的 `sleep` 函数，因为它提供的时间精度是微秒。

2、select 返回值

Select 返回值体现在两个方面：

- 函数返回值
- 修改的三个描述符集合的参数

返回值：

(1) 返回值 < 0 ，这种一般表示函数执行出错，比如使用了不可用的描述符。有时候，`select` 会被信号打断，也会返回 < 0 的值。因为 `select` 函数是不支持自动重启动的，所以被信号打断会立即返回，然后把 `errno` 的值设置成 `EINTR`。

(2) 返回值 $= 0$ ，这种只对设置了超时时间的方式才会出现，如果返回 `0`，表示超时时间到了，还没有事件发生。

(3) 返回值 > 0 ，表示监听的描述符中，有几个事件发生。也就是读集合中的事件数 + 写集合中的事件数 + 异常集合中的事件数。就算不同集合间有重复的描述符，也会累计。

修改的参数：

对于 `select` 函数，如何才能知道哪些描述符上发生了事件，主要是修改了三个传入的描述符集合。

如果某个集合中的描述符上有事件到来，在 `select` 返回时，会保留该描述符，所有其它未发生事件的描述符全部清除。

对于超时参数来说，如果在超时时间到达前发生异常或有事件到来，则该参数会被更新为剩余时间。

使用 `select` 函数的时候，要处理被信号中断的情况

7.4 poll 函数

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

`poll` 函数的第一个参数是一个类型为 `struct pollfd` 的数组，第二个参数 `nfds`（一般是个无符号的整型）是这个数组的大小，第三个参数就是超时时间了。

1、struct pollfd 类型

```
struct pollfd {  
    int    fd;        /* 文件描述符 */  
    short  events;     /* 监听的事件，比如可读事件，可写事件 */  
    short  revents;    /* poll 函数的返回结果，是可读还是可写 */  
};
```

和 `select` 函数不同，它没有将可读可写的描述符分成两个集合存放，而是都放在一个结构体里进行统一监管。

1.成员 `fd`，表示要监管哪个描述符，如果 `fd<0`，则忽略。

2.对 `events` 来说，它表示你要监听的事件是可读还是可写的还是什么。通常用位或操作对其赋值。在 linux 中，`events` 有下面的可选值：

- `POLLIN`: 监听是描述符是否可读，相当于 `select` 中的读集合参数。
- `POLLPRI`: 监听是否有紧急数据可读（比如 TCP 套接字上的 `out-of-band(OOB)` 数据）
- `POLLOUT`: 监听是描述符是否可写。
- `POLLRDHUP`: 这种事件只有 Linux 2.6.17 内核后才支持，它监听流式套接字对端是否关闭右半关闭。

如果监听的事件到来，`poll` 函数会将它保存到 `revents` 成员中。比如监听到了有数据可读，则 `revents` 的 `bit` 位中就会将 `POLLIN` 保存。如果监听到了有数据可写，那 `revents` 的 `bit` 位中会保存 `POLLOUT`。

还有一些事件，即使你不主动监听它也会发生，并保存到 `revents` 中，这些事件一般都是异常事件：

- `POLLERR`: 这种情况极少见，很抱歉后面的实验没演示。一般是硬件上的问题吧。
- `POLLHUP`: 对端挂断，比如对于有名管道，其中一端关闭了。
- `POLLNVAL`: 使用了未打开的描述符

2、timeout 参数

- `timeout = -1`，表示永远等待，直到有事件发生。
- `timeout = 0`，不等待，立即返回。
- `timeout > 0`，等待 `timeout` 毫秒。

3、poll 和 select 区别

- select 使用 `fd_set` 来存放描述符，poll 使用结构体数组。
- select 能够一次监听的描述符数量是受 `fd_set` 集合的限制的，通常这个集合最多只能放 1024 个描述符。而 poll 一次能够监听的描述符个数是根据数组大小来决定的，这要看 `nfds_t` 被定义成什么类型了，如果是 `unsigned long`，4 字节宽度的情况下，poll 能监听 $2^{32}-1$ 个描述符。

4、epoll 函数

理解 epoll 的三个函数的用法，以及 IO 事件的概念，以及 epoll 相对于 poll 和 select 的优点是什么。

epoll 基础（IO 事件的引入）：epoll 基础

epoll 函数（三个函数使用）：epoll 函数

4.1 epoll (触发模式)

触发是针对 IO 事件的。分为边沿触发 `edge-triggered` 和水平触发 `level-triggered`

边沿触发：只有 IO 缓冲区发生变化（数据变多或者变少），才会导致 IO 事件产生，`epoll_wait` 函数才会返回。

水平触发：IO 缓冲区发生变化肯定导致 IO 事件产生，但是 IO 缓冲区不发生变化，也有可能产生 IO 事件，比如对于读 IO，只要缓冲区有数据可读（即使此时缓冲区数据没有变化），也会产生 IO 事件。对于写 IO，只要缓冲区有空位可写（即使缓冲区没有发生变化），就会产生 IO 事件。`epoll_wait` 函数就会返回。

如何设置 epoll 的触发方式？

前面已经非常详细的说明了 `struct epoll_events` 的 `events` 成员了，它除了设置监听什么事件以外，还可以设置触发方式，如果想要设置 `edge_triggered` 触发方式，只要执行下面的语句就行了：

```
ev.events |= EPOLLET;
```

注意：边沿触发因为是只有缓冲区发生变化才会产生 IO 事件，所以容易使 `read` 阻塞，一

般我们使用边沿触发+非阻塞的方式。

两种触发模式的优缺点：

假设每次每次 read 数据都不能一次性将 IO 缓冲区中的数据读完。

首先从程序的运行角度来说：

- 1.水平触发+阻塞读，每次读只要缓冲区有数据，epoll_wait 就会返回，并 read。
- 2.边沿触发+非阻塞读，对 read 加一个 while 循环，只触发一次 epoll_wait 函数返回，再读缓冲区数据没有发生变化它就不返回了。然后 read 循环读。

那么效率就体现在：

- `while { epoll_wait + read }`
- `epoll_wait + while {read}`

看起来像是边沿触发效率高，实际上这种是无法确定的。[知乎关于 ET 和 LT 谁更快的讨论](#)

8. 高级 IO-同步 IO 与异步 IO

8.1 引导

迄今为止，所见所学都是同步 IO。其主要特征是使用类似 `read`、`write` 等系统调用。这包含了阻塞 IO，与非阻塞 IO，另外 IO 多路复用技术，也都是同步 IO。

网上有帖子讨论：[同步 IO 异步 IO 讨论](#)

8.2 POSIX 异步 IO 概念

注意：POSIX 只是一种版本。还有 System V, BSD 等版本。

● 同步 IO

假设对于从缓冲区读数据来说。无论如何，不管你是阻塞的还是非阻塞的，都需要最后自己主动调用 `read`（或其他同步 `read` 函数）函数去讲内核缓冲区的数据复制到 `read` 函数的接收缓冲区中。就算你是非阻塞 IO，最后还是需要自己去主动 `read` 数据。

对于写 IO 来说，不管你是阻塞的，还是非阻塞的。最后都需要自己主动调用 `write`（或其他同步 `write` 函数）函数将 `write` 函数缓冲区的数据复制到内核缓冲去中。

由以上定义：`read`，`write`，`select`，`poll`，`epoll`，都是同步 IO。

● 异步 IO

对于读数据来说。不管你是阻塞的还是非阻塞的，`read` 请求后。最后都是操作系统（内核）在某个未知的时刻将数据复制到你指定的缓冲区中的。

对于写数据来说也是一样。

综上，同步 IO 于异步 IO 的区别；

到底是主动调用 `read` 或 `write` 函数在用户缓冲区和内核缓冲区中复制数据（你确切的知道是你自己主动调动的）。还是仅仅由操作系统（内核）在某个位置时刻，帮你从两个缓冲区间复制数据。（注意：是复制数据的动作不是你主动发起的，你只是发起了一个请求）

注意：非阻塞 IO 不是异步 IO。

在处理 IO 的时候，阻塞和非阻塞都是同步 IO。

只有使用了特殊的 API 才是异步 IO。

同步	IO multiplexing (select/poll/epoll)		
	阻塞	非阻塞	
异步	Linux	Windows	.NET
	AIO	IOCP	BeginInvoke/EndInvoke

遗留问题：

那么这里就有一个问题，因为异步 IO 的数据读写完成时操作系统完成的，而且是在位置时刻完成的。那么就需要有一种机制，来通知用户程序异步 IO 操作已经完成或者失败等。问题先放这里，先看一些关于异步 IO 函数的概念：

8.21、异步 IO 的 aiocb 结构（类似于 PCB 进程控制块）

POSIX 异步 IO，它对应的是 aio 开头的一系列函数：

函数	语义
aio_read	请求异步读操作
aio_error	检查异步请求的状态
aio_return	获得完成的异步请求的返回状态
aio_write	请求异步写操作
aio_suspend	阻塞调用进程，直到一个或多个异步请求已经完成（或失败）
aio_cancel	取消异步 I/O 请求
lio_listio	发起一系列 I/O 操作

```
#include <aio.h>
```

```
struct aiocb {  
    /* 下面所有字段依赖于具体实现 */  
  
    int aio_fildes; /* 文件描述符 */  
    off_t aio_offset; /* 文件偏移 */  
    volatile void *aio_buf; /* 缓冲区地址 */  
    size_t aio_nbytes; /* 传输的数据长度 */  
    int aio_reqprio; /* 请求优先级 */  
    struct sigevent aio_sigevent; /* 通知方法 */  
    int aio_lio_opcode; /* 仅被 lio_listio() 函数使用 */  
};
```

```
/* Various implementation-internal fields not shown */  
};
```

- **aio_fildes**: fildes 就是 file descriptor 的缩写，这个字段，就相当于你使用 read 或 write 函数时的第一个 fd 参数，表示你想操作哪个文件描述符上的 IO。
- **aio_offset**: 文件偏移指针，这个字段，表示你想从文件的哪个位置开始操作。比如你要从文件的第 10 个字节开始读，那这里你就设置成 10。
- **aio_buf**: 这个位置，缓冲区的地址。
- **aio_nbytes**: 表示要传输多少字节的数据。

8.22、异步通知

前面留下来的问题：[遗留问题：](#)

前面一直使用 while 循环和 aio_error 函数不断的去测试异步操作有没有完成。这种方法很笨。

我们可以通过设置 aiocb 的成员 **sigevent** 结构体中的 **sigev_notify** 成员来设置在异步 IO 完成操作后，用什么方式通知用户程序。

两种通知方式：信号，线程。

信号: 当异步 IO 完成后，用户程序会收到你指定的信号，然后执行相应的处理函数来处理。

线程: 当异步 IO 完成后，内核会创建一个新线程，新线程执行你指定的线程函数。（注意此时不是自己创建线程，只是自己定义线程函数。）

9、网络编程基础

9.1 字节序

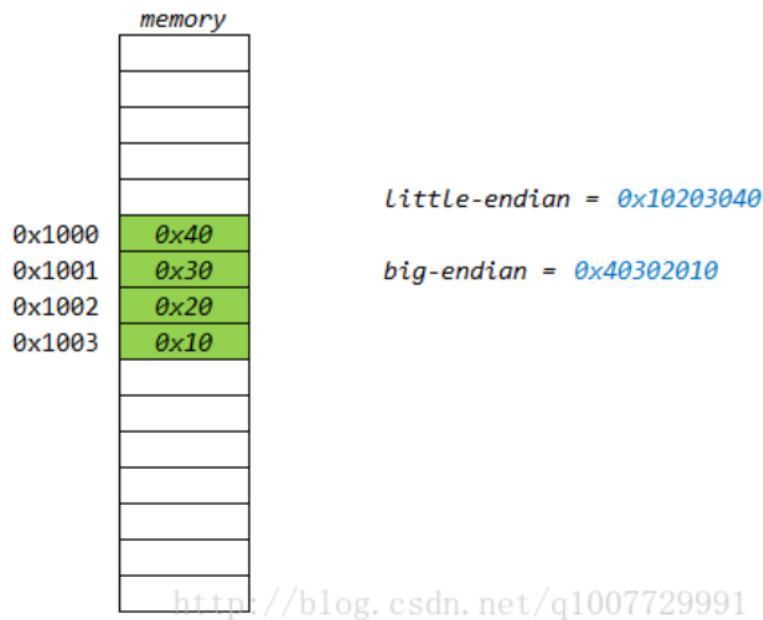


图1 内存中的数据

小端机器：低位（低字节）数据存储在内存中的低地址，数据的高位保存在内存中的高地址处。

大端机器：相反。

大部分情况下，我们的使用都是小端机器，Intel 处理器和 AMD 处理器基本上都是小端的。但是也有一些处理器是大端的。

9.11 网络字节序

为了实现不同机器之间的通信，需要定义网络字节序。在进行不同机器（位于不同主机的进程）之间的通信时，传输的数据都是以网络字节序保存的。

网络字节序是大端字节序。

POSIX 提供了 4 个函数（也可能是用宏来实现的），可以让本机字节序和网络字节序之间进行互转。它们分别是：

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);

uint16_t htons(uint16_t hostshort);

uint32_t ntohl(uint32_t netlong);

uint16_t ntohs(uint16_t netshort);
```

9.2 IPv4 地址

IPv4 地址，本质上是 32 位无符号整数。

9.3 地址和端口号

网络通信的本质仍然是进程间通信。

理解为什么不能用 ip: pid 来表示网络通信中的地址，。因为 pid 不稳定，可能会改变。

使用 ip: port 这样的一对值表示套接字地址。用套接字地址表示网络通信的进程的地址。

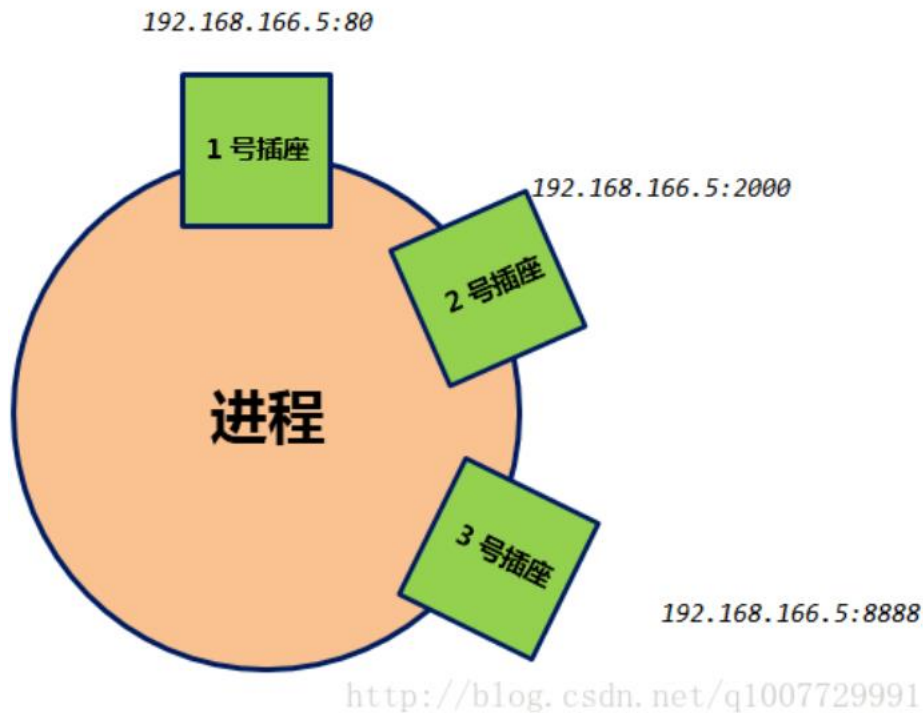


图1 某个进程上绑定了很多插座

套接字的结构:

```
struct sockaddr_in {  
    sa_family_t    sin_family; /* 这个值固定写 AF_INET */  
    in_port_t      sin_port;   /* 网络字节序的端口号 */  
    struct in_addr sin_addr;    /* in_addr 类型的 ip 地址 */  
};
```

Connect 函数: 创建套接字地址并将该套接字地址 **bind** 上我们指定的 **socket** 上, 然后发起连接请求。

主动 socket, 是指未来你将要用此 **socket** 主动发起连接, 即将要把它传递给 **connect** 函数。

被动 socket, 也被称为监听 **socket**, 它不能用作 **connect** 函数的参数。这个套接字, 只能用来接受连接请求。

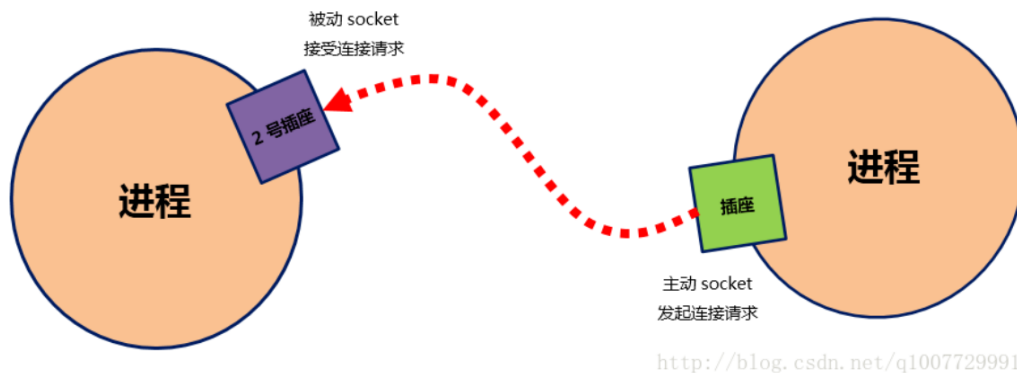


图1 主动 socket 与被动 socket

即使我们使用 `socket` 创建了一个套接字并 `bind` 到了套接字地址上，这个进程也无法接受请求，除非我们将此 `socket` 转换成被套接字。

如何将主动 `socket` 设置成被动 `socket` 的呢？对应到图 1 中，就是那个紫色的插座，它原先是个绿色的，要怎么把它变成紫色的？

函数 `listen` 可以完成这个操作。

```
int listen(int sockfd, int backlog);
```

1) 如果套接字 `sockfd` 还未绑定套接字地址，`listen` 会选择本地地址，并随机选择一个端口号绑定到 `sockfd` 上。（这一条规则蕴含的意思是，作为服务器，可以不调用 `bind` 函数，但是通常不这么干。）

2) `listen` 函数将主动套接字 `sockfd` 转换成被动套接字。参数 `backlog` 用来指定未决连接队列大小。

当 `connect` 函数成功返回后，著名的“三次握手”就算完成了。

对于服务器进程来说，`connect` 完成三次握手后，还不能进行通信。此时建立的连接，还是处于服务器进程的未决队列中，还需要从队列中取出一个连接进行真正的通信。此过程由函数 `accept` 函数完成。

```
int accept(int sockfd, struct sockaddr* addr, socklen_t *addrlen);
```

第一个参数是被动 `socket`，第二个参数是对端进程的套接字地址。

函数语义：

`accept` 函数从未决队列中取出一个连接，并创建一个新的 `socket`，并将其描述符返回。

注意 `socket` 是阻塞 IO，当未决队列为空，`accept` 函数会移植阻塞，直到队列中出现新的连接。

关键点: **accept** 函数 创建新的 **socket**, 服务器进程利用这个新创建的进程与客户端进程进行通信。

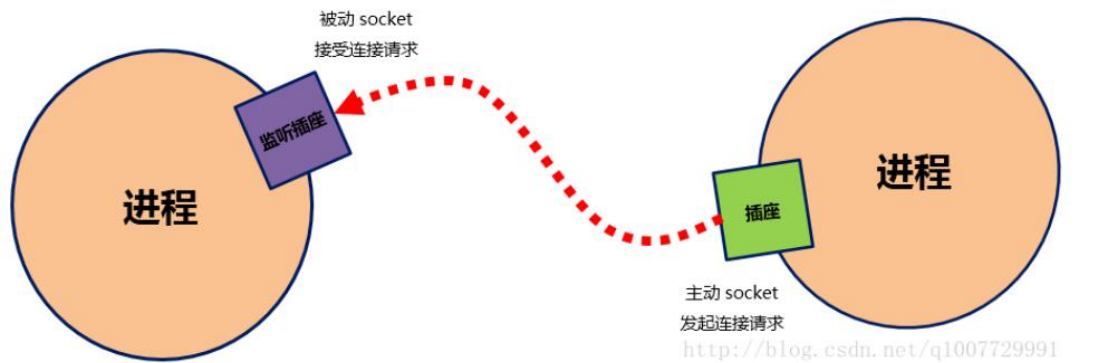


图1 接受连接

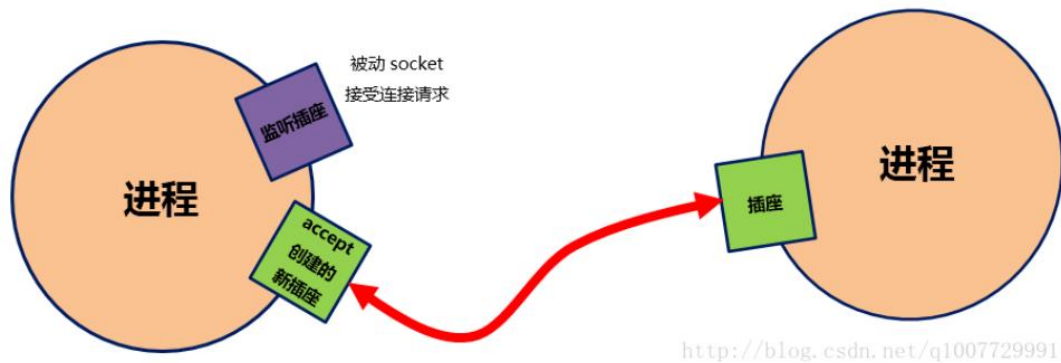


图2 建立通信链路

UDP: (无连接, 所以服务端不需要 **listen** 和 **accept**, 客户端不需要 **connect**)

- 服务器端程序编写步骤

- (1) 创建 **socket** (**socket** 函数)
- (2) 将 **socket** 与套接字地址绑定 (**bind** 函数)
- (3) 使用 **recvfrom** 函数接收数据
- (4) 使用 **sendto** 函数发数据
- (5) 关闭套接字

- 客户端程序编写步骤

- (1) 创建 **socket** (**socket** 函数)
- (2) 使用 **sendto** 函数发数据
- (3) 使用 **recvfrom** 函数接收数据
- (5) 关闭套接字

客户端中不需要 **bind**, **sendto** 后, 系统自动分配一个套接字地址绑定该 **socket**。端口号随机。