# 进程操作

## 常规操作

### fork

原型：pid\_t fork(void)

功能：创建子进程

返回值：fork调用一次，返回两次，有三种不同的返回值

1. 父进程中，fork返回新进程的子进程PID
2. 子进程中，fork返回0
3. 如果出现错误，fork返回一个负值

注意：使用此函数创建的子进程，其数据空间、堆栈空间都会从父进程得到一个拷贝，而不是共享。

在Linux系统中，用户创建进程的唯一方法就是使用系统调用fork，其大致操作：

1. 分配表项，一个用户的进程项是有限的；
2. 创建子进程的进程标识号；
3. 复制父进程中的项目给子进程；
4. 与父进程相连的文件表和索引值加1，与子进程相连；
5. 内核为子进程创建用户级上下文；
6. 生成进程的动态部分。

### vfork

vfork与fork创建子进程的区别：

1. fork子进程拷贝父进程的数据段，vfork子进程与父进程共享数据段
2. fork父子进程的执行次序不确定，vfork子进程先执行，然后执行父进程

### setsid

当进程是会话的领头进程时，setsid调用成功返回新的会话ID，失败并返回-1。调用sedsid函数的进程称为新的会话领头进程，并与其父进程的会话组和进程组脱离。由于会话对控制终端的独占性，进程同时与控制终端脱离。

setsid后子进程不受终端影响，终端退出，不影响子进程，即设置守护进程（我们在SecureCRT启动该进程时，它可以单独运行，退出控制终端后也不会停止）。

编写守护进程的一般步骤：

1. 在父进程中执行fork并exit退出；
2. 在子进程中调用setsid函数创建新的会话；
3. 在子进程中调用chdir函数，让根目录“/”成为子进程的工作目录；
4. 在子进程中调用umask函数，设置进程的umask为0；
5. 在子进程中关闭任何不需要的文件描述符。

### exec

exec系统调用是一个函数蔟：

int execl(const char \*path,const char \*arg);

int execlp(const char \*file,const char \*arg);

int execle(const char \*path,const char \*arg, char \*const envp[]);

int execv(const char \*path,const char \*arg[]);

int execvp(const char \*file,const char \*arg[]);

exec函数最大的作用就是可以取代调用进程的内容，也是Linux可以执行新程序的关键。

exec与fork区别：

1. fork创建一个新的进程，产生一个新的PID；
2. exec启动一个新的程序，替换原来的进程，进程的PID不会改变。

当fork完后调用exec便会运行新的进程，这也是为何fork赋值出来的子进程执行之后与父进程不一样的根本原因。

### exit

函数exit就是进程结束函数，就是一个进程结束的标识。

exit与\_exit函数区别：

1. exit定义在stdlib.h中，\_exit定义在unistd.h中；
2. 它们都是写入内存，但是\_exit只有在满足特点的条件下才能写入文件，如何要保存数据的完整性，一般使用exit函数。

### getpid/getppid

pid\_t getpid(void) //获取本进程ID

pid\_t getppid(void) //获取父进程ID

### wait

原型：pid\_t wait(int \*status)

参数：status

在wait中，这是一个指向一个整型数据的指针，用来存放子进程退出时的状态，后来又定义了一个WIFEXITED（status）的宏来完成这个工作，这里的status就是一个指针指向的函数。

功能：阻塞该进程，直到某个子进程退出（用于结束子进程）。

进程同步：通过wait在父进程中等待子进程的返回值。

僵尸进程：当一个进程调用exit之后，该进程并非马上消息还留下一些残余的信息。可以使用wait结束僵尸进程，进程一旦调用wait，就会立即阻塞自己，当分析到当前进程的子进程已经exit，便会收集这个子进程的信息，然后彻底销毁，如果没有找到这样的子进程，就会一直阻塞在这里，直到有一个出现。

### sleep

函数sleep就是挂起进程指定的秒数，时间到了返回0。

### kill

## 特殊操作

### setuid

设置特权进程的函数：setsid、setgid。setsid设置用户ID位，setgid设置组ID位。

当一个程序一旦设置了这个标志以后，运行该程序的进程将拥有该程序所有者同样的权限。

setsid提升使用者的权限，普通用户可以执行该命令，使自己升级为root（比如chmod）。

### setgid

与setsid类似，使得使用者在执行该文件时，都绑定了文件所有组的权限，单独setgid的文件非常少用，通常都是setsid的同时setgid。

### setpgrp

### setpgid

### chdir

### chroot

### nice

# 进程间通信

## 概述

进程间通信方式：管道、FIFO、消息队列、锁、信号、信号量、共享内存、远程过程调用

线程间通信方式：锁、信号、信号量、共享内存

### 进程通信

同一主机上进程的通信方式：

1. Unix进程间通信方式：管道PIPE、有名管道FIFO和信号signal
2. SystemV进程间通信方式：信号量（Semaphore）、消息队列（Message Queue）和共享内存（shared Memory）

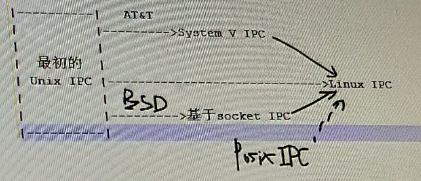
### 网络通信

网络主机间的通信方式：

1. RPC：Remote Protocol Call远程过程调用（thrift、grpc、qpid）
2. Socket：基于TCP/IP协议

## 分类

这几种IPC的关系：



### Unix IPC

UnixIPC包括：管道、FIFO、信号

### System IPC

SystemV IPC包括：SystemV消息队列、SystemV信号量、SystemV共享内存

### Posix IPC

Posix IPC包括：Posix消息队列、Posix信号量、Posix共享内存区

### Linux IPC

## 消息传递

### 管道

#### PIPE

管道是一种半双工的通信方式（有的系统可能支持全双工），数据只能单向流动，而且只能在具有亲缘关系（父子进程）的进程之间使用。

管道传输的是无格式字节流，并且管道缓冲区的大小是有限的（管道缓冲区存在于内存中，在管道创建时，为缓冲区分配一个页面大小）。

#### popen

popen创建一个管道并启动另外一个进程，该进程要么从该管道读出标准输入，要么从该管道写入标准输出。

#### pclose

#### 代码

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#define MAX\_LEN 128

int main(void)

{

/\*0为读，1为写\*/

int fd[2] = {0}; //描述符

pid\_t pid = 0;

char line[MAX\_LEN] = {0};

int n = 0;

/\*创建管道，需要传入两个文件描述符\*/

if(pipe(fd) < 0)

{

perror("create pipe failed\n");

return -1;

}

/\*fork子进程\*/

if((pid = fork()) < 0)

{

perror("fork failed\n");

return -1;

}

/\*父进程\*/

else if(pid > 0)

{

/\*关闭管道的写描述符\*/

close(fd[1]);

/\*从管道读取数据\*/

n = read(fd[0],line,MAX\_LEN);

printf("read %d bytes from pipe :%s\n",n,line);

}

/\*子进程\*/

else

{

/\*关闭管道的读描述符\*/

close(fd[0]);

/\*向管道写入数据\*/

write(fd[1],"www.yanbinghu.com",sizeof("www.yanbinghu.com"));

}

return 0;

}

在程序中，我们创建了一个管道，父进程关闭了写通道，子进程关闭读通道；子进程向管道内写入字符串，而父进程从管道中读取字符串并输出。

运行结果：

read 18 bytes from pipe :www.yanbinghu.com

### FIFO

由于管道是一种无形、无名的文件，它就只能通过fork的过程创建在“近亲”的进程之间，而不可能成为可以在任意两个进程之间建立通信的机制，更不可能称为一般的、通用的进程间通信模型。

同时，管道机制的这种缺点本身就强烈暗示人们，只有用有名、有形的文件来实现管道，就可以克服这种缺点。这里所谓的有名是指这样一个文件应该有文件名，使得任何进程都可以通过文件名或者路径与这个文件挂上钩。在这里，“有形”是指文件的inode应该存在于磁盘或者其他文件系统上，使得任何进程在任意时间（不仅仅是在fork时）都可以建立或断开与这个文件的联系，所以命名管道的出现是必然的。

为了实现“命名管道”，在普通文件，块设备文件，字符设备文件之外又建立了一个文件类型，称为FIFO文件。由于这种文件的访问，严格遵守先进先出的原则，而不允许在文件内移动读写指针位置的lseek操作。

有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

FIFO也常常有以下两个用途：

无需创建中间临时文件，复制输出流

多客户-服务进程应用中，通过FIFO作为汇聚点，传输客户进程和服务进程之间的数据

#### mkfifo

#### open

#### fopen

#### fork

#### 代码

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <errno.h>

#include <fcntl.h>

#define FIFO "/tmp/fifo"

#define MAX\_LEN 128

int main(void)

{

int writeFd;

char line[MAX\_LEN] = {0};

if(mkfifo(FIFO,S\_IRUSR|S\_IWUSR) < 0 && (errno != EEXIST))

{

perror("make fifo failed:");

return -1;

}

/\*关闭管道的读描述符\*/

writeFd = open(FIFO,O\_WRONLY,0);

/\*向管道写入数据\*/

write(writeFd,"www.yanbinghu.com",sizeof("www.yanbinghu.com"));

close(writeFd);

return 0;

}

它首先创建了一个FIFO，并且打开后，往里面写入字符串，然后关闭退出。

读进程代码如下：

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <errno.h>

#include<fcntl.h>

#define FIFO "/tmp/fifo"

#define MAX\_LEN 128

int main(void)

{

int readFd,n;

char line[MAX\_LEN] = {0};

/\*打开FIFO，这里打开可能失败，应该要对返回值处理\*/

readFd = open(FIFO,O\_RDONLY,0);

/\*从FIFO读取数据\*/

n = read(readFd,line,MAX\_LEN);

printf("read %d bytes from pipe :%s\n",n,line);

close(readFd);

/\*删除FIFO\*/

unlink(FIFO);

return 0;

}

它先打开一个已知的FIFO，然后从FIFO中读取数据。

在一个终端先运行写进程，然后运行读进程，结果如下：

read 18 bytes from pipe :www.yanbinghu.com

我们可以看到，两个没有亲缘关系的进程可以通过FIFO进行通信。

### 消息队列

消息队列是由消息的链表，存储在内核中并由消息队列标识符标识。消息队列克服了信号传递信号量少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

消息队列传递的信息还是比较大的，管道信息传递较少，信号更少。为了解决管道、有名管道的弊端，出现了消息队列，在某个进程往另一个进程写入消息之前，并不需要另外某个进程在该队列上等待消息的到达。

与管道和FIFO不同，进程可以在没有另外一个进程等待读的情况下进行写。另外一方面，管道和FIFO一旦相关进程都关闭并退出后，里面的数据也就没有了，但是对于消息队列，一个进程往消息队列中写入数据后退出，另外一个进程仍然可以打开并读取消息。消息队列与UNIX域套接字相比，在速度上没有多少优势。

## 信号同步

同步也是一种通信，但是这种通信的目的是为了实现同步。而消息传递、socket等之类的IPC是为了交换数据，而不仅仅为了实现通信。

同步是进程和线程间共有的通信方式。

### 锁

### 信号

信号用于通知接收进程某个事件已经发生。

#### 概述

信号是与一定的进程相联系的，也就是说，一个进程可以决定在进程中对哪些信息进行怎么样的处理。

信号（signal）是Linux进程间通信的一种机制，全程为软中断信号，也被称为软中断。信号本质上是在软件层次上对硬件中断机制的一种模拟。

信号表示一种事件，也可能异步发生，也就是并不与程序执行过程的任何事件同步。如果程序并未安排怎么处理一个特定的信号，那么该信号出现时程序就作出一个缺省的反应。标准未定义这个缺省反应是什么，但绝大多数编译器选择终止程序。另外，程序可以调用signal函数，或者忽略这个信号，或者设置一个信号处理函数，当信号发生时程序就调用这个函数。

与其他进程间通信方式相比，信号所能传递的信息比较粗糙，只是一个整数。但正是由于传递的信息量少，信号也便于管理和使用，可以用于系统管理相关的任务，例如通知进程终结、中止或恢复等。

信号可能在任何时刻发生，所以由信号处理函数修改的变量的值可能会在任何时候发生改变，因此，不能指望这些变量在两条相邻的程序语句中肯定具有相同的值。Volatile关键字就是告诉编译器这个事实，防止它以一种可能修改程序含义的方式“优化”程序。

#### 原理

信号由内核管理，产生方式多种多样：

1. 可以由内核自身产生，比如出现硬件错误、内存读取错误、分母为0除法等，内核需要通知相应进程；
2. 也可以由其他进程产生并发送给内核，再由内核传递给目标进程。

信号传递的过程：

1. 内核中针对每一个进程都有一个表来保存信号；
2. 当内核需要将信号传递给某个进程时，就在该进程对应的表中写入信号，这样就生成了信号；
3. 当该进程由用户态陷入内核态，再次切换到用户态之前，会查看表中的信号。如果有信号，进程就会首先执行信号对应的操作，此时叫做执行信号；
4. 从生成信号到将信号传递给对应进程这段时间，信号处于等待状态；
5. 我们可以编写代码，让进程阻塞（block）某些信号，也就是让这些信号始终处于等待的状态，直到进程取消阻塞（unblock）或者忽略信号。

#### 信号名

SIGABRT 程序请求异常终止

SIGFPE 发生一个算术错误

SIGILL 检测到非法指令

SIGSEGV 检测到对内存的非法访问

SIGINT 收到一个交互性注意信号

SIGTERM 收到一个终止程序的请求

注：前几个都是同步的，SIGINT、SIGTERM是异步的，它们在程序外部发生，同城是由程序的用户触发。

#### 函数

##### abort

SIGABRT是由函数abort引发的信号，用于终止程序。

##### raise

函数raise用于显式地引发一个信号（由参数指定），程序对这类信号的反应和哪些自主发生的信号都是相同的。可以调用这个函数，对信号处理函数进程测试。

##### signal

当一个信号发生时，程序可以使用三种方式对它作出反应。缺省的反应是由编译器定义的，通常是终止程序。程序也可以指定其他行为对函数作出反应，信号可以被忽略或者设置一个信号处理函数。

##### sigaction

原型：int sigaction(int sig, const struct sigaction \*act, struct sigaction \*oact);

说明：

1. 阻塞：sigaction函数具有阻塞功能，比如SIGINT信号来了，进入信号处理函数，默认情况下，在信号处理函数未完成之前，如果又来了一个SIGINT信号，其将被阻塞，只有信号处理函数处理完毕，才会对后来的SIGINT再进程处理，同时后续无论来多少个SIGINT，仅需要一个SIGINT，sigaction会对后续SIGINT进行排队合并处理；
2. sa\_mask，信号屏蔽集，可以通过函数sigemptyset/sigaddset等来清空和增加需要屏蔽的信号；
3. sa\_flag，如果取值为0，则表示默认行为。

sigaction与signal区别：

signal比sigaction简单，但是signal注册的信号在sa\_handler被调用之前会把信号的sa\_handler指针恢复，而sigaction注册的信号在处理信号时不会恢复sa\_handler指针。所以，用signal函数注册的信号处理函数只会被调用一次，之后收到这个信号将按照默认处理方式处理，如果想一直处理这个信号的话就得在信号处理函数中再用signal注册一次，一般都在信号处理函数开始处调用signal注册一次这个信号，虽然这样可以一直能处理这个信号，但是可以看出，在sa\_handler指针恢复到再次调用signal注册信号期间如果收到这个信号，那么这个信号就按照默认处理方式，如果是INT之类的信号，进程就有可能退出。虽然这种概率很小，但是还是存在风险的。因此更好的做法是，除了SIG\_IGN、SIG\_DFL之外，最好用sigaction来代替signal注册信号。

##### sigemptyset

##### sigaddset

##### sigprocmask

## 信号量

### 概述

信号量（semaphore），有时称为信号灯，是多线程环境下使用的一种设施，可以用来保证两个或多个关键代码段不被并发调用（就是具有原子性的计数器）。

信号量是一个计数器，可用来控制多个进程对共享资源的访问。它常常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因为，作为进程间以及同一进程内不同线程之间的同步手段。

信号量用于多进程或者同一进程的不同线程之间的通信。在进入一个关键代码之前，线程必须获取一个信号量，一旦该关键代码段完成了，那么该线程必须释放信号量，其他想进入该关键代码的线程必须等待直到第一个线程释放信号量。

### 流程

它的主要流程如下：

检查控制该资源的信号量

如果信号量值大于0，则资源可用，并且将其减1，表示当前已被使用

如果信号量值为0，则进程休眠直至信号量值大于0

也就是说，它实际上是提供了一个不同进程或者进程的不同线程之间访问同步的手段。

### 分类

Unix中涉及的信号量包括：

1. Posix有名信号量（又称有名信号量）：使用Posix IPC名字标识，可用于进程或线程间的同步；
2. Posix基于内存的信号量（又称无名信号量）：存放在共享内存区中，可用于进程或线程间的同步；
3. SystemV信号量：在内核中维护，可用于进程或线程间的同步。

**整型信号量：**信号量是整数

**记录型信号量：**每个信号量s处于一个整数值s.value（计数）外，还有一个进程等待队列sL，其中是在阻塞在该信号量的各个进程的标识

**二进制信号量：**只允许信号量取0或1的值

## 函数

### 初始化

初始化也叫建立（create），原型如下：

int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

### 等信号

等信号也叫挂起（suspend），原型如下：

int sem\_wait(sem\_t \*sem);

### 给信号

给信号（signal）或者发信号（psot），原型如下：

int sem\_post(sem\_t \*sem);

### 清理

## 共享内存

共享内存就是映射一段能被其他进程所访问的内存（一旦这样的内存区域映射到共享它的进程的地址空间，这些进程间数据的传递就不再涉及内核，往共享内存区域存放信息或从中取走消息的进程间通常需要某种形式的同步），这段共享内存由一个进程创建，但多个进程都可以访问。

共享内存是最快的IPC方式，它是针对其他进程间通信方式效率低而专门设计的，它往往与其他通信机制如信号量配合使用，来实现进程间的同步和通信。

Posix提供了两种在无亲缘关系进程间共享内存的方法：

1. 内存映射文件：open函数打开，mmap函数把得到的描述符映射到当前进程地址空间中的一个文件；
2. 共享内存区对象：shm\_open打开一个IPC name，所返回的描述符由mmap映射到当前进程的地址空间。

### open

### shm\_open

### shm\_unlink

### ftruncate

## RPC

## 套接字

# 进程锁文件