进程

1. 基本概念

程序

程序是包含一系列信息的文件，这些信息描述了如何在运行时创建一个进程：

- 二进制格式标识：每个程序文件都包含用于描述可执行文件格式的元信息。内核利用此信息来解释文件中的其他信息。（ELF可执行连接格式）
- 机器语言指令：对程序算法进行编码。
- 程序入口地址：标识程序开始执行时的起始指令位置。
- 数据：程序文件包含的变量初始值和程序使用的字面量值（比如字符串）。
- 符号表及重定位表：描述程序中函数和变量的位置及名称。这些表格有多重用途，其中包括调试和运行时的符号解析（动态链接）。
- 共享库和动态链接信息：程序文件所包含的一些字段，列出了程序运行时需要使用的共享库，以及加载共享库的动态连接器的路径名。
- 其他信息：程序文件还包含许多其他信息，用以描述如何创建进程。

进程

- 进程是正在运行的程序的实例。是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。
- 它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。
- 可以用一个程序来创建多个进程，进程是由内核定义的抽象实体，并为该实体分配用以执行程序的各项系统资源。
- 从内核的角度看，进程由用户内存空间和一系列内核数据结构组成，其中用户内存空间包含了程序代码及代码所使用的变量，而内核数据结构则用于维护进程状态信息。记录在内核数据结构中的信息包括许多与进程相关的标识号（IDs）、虚拟内存表、打开文件的描述符表、信号传递及处理的有关信息、进程资源使用及限制、当前工作目录和大量的其他信息。
- 对于一个单 CPU 系统来说，程序同时处于运行状态只是一种宏观上的概念，他们虽然都已经开始运行，但就微观而言，任意时刻，CPU 上运行的程序只有一个。
- 在多道程序设计模型中，多个进程轮流使用 CPU。而当下常见 CPU 为纳秒级，1秒可以执行大约 10 亿条指令。由于人眼的反应速度是毫秒级，所以看似同时在运行。

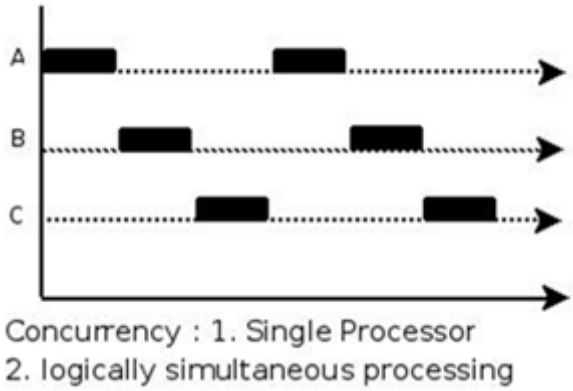
单道多道程序设计

- 单道程序，即在计算机内存中只允许一个的程序运行。
- 多道程序设计技术是在计算机内存中同时存放几道相互独立的程序，使它们在管理程序控制下，相互穿插运行，两个或两个以上程序在计算机系统中同处于开始到结束之间的状态, 这些程序共享计算机系统资源。引入多道程序设计技术的根本目的是为了提高 CPU 的利用率。
- 对于一个单 CPU 系统来说，程序同时处于运行状态只是一种宏观上的概念，他们虽然都已经开始运行，但就微观而言，任意时刻，CPU 上运行的程序只有一个。
- 在多道程序设计模型中，多个进程轮流使用 CPU。而当下常见 CPU 为纳秒级，1秒可以执行大约 10 亿条指令。由于人眼的反应速度是毫秒级，所以看似同时在运行。

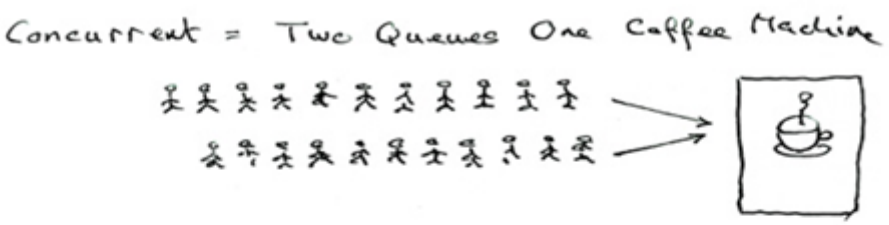
并行和并发

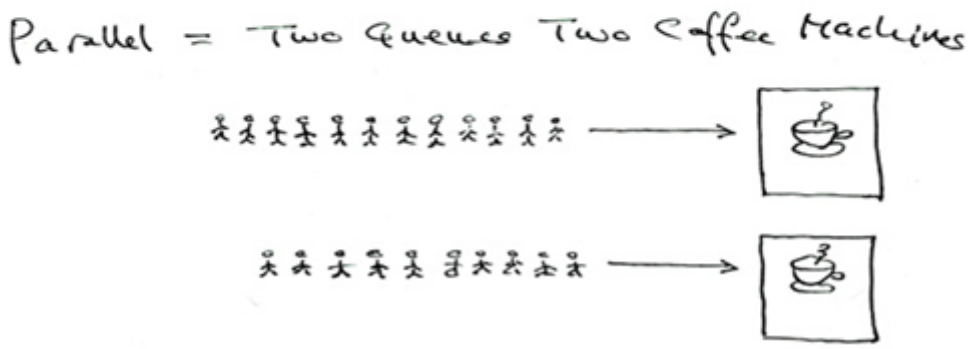
- 并行(parallel)：指在同一时刻，有多条指令在多个处理器上同时执行。

- 并发(concurrency)：指在同一时刻只能有一条指令执行，但多个进程指令被快速的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，使多个进程快速交替的执行。



并发是两个队列交替使用一台咖啡机。并行是两个队列同时使用两台咖啡机。





进程控制块（PCB）

为了管理进程，内核必须对每个进程所做的事情进行清楚的描述。内核为每个进程分配一个PCB(Processing Control Block)进程控制块，维护进程相关的信息，Linux 内核的进程控制块是task_struct 结构体。

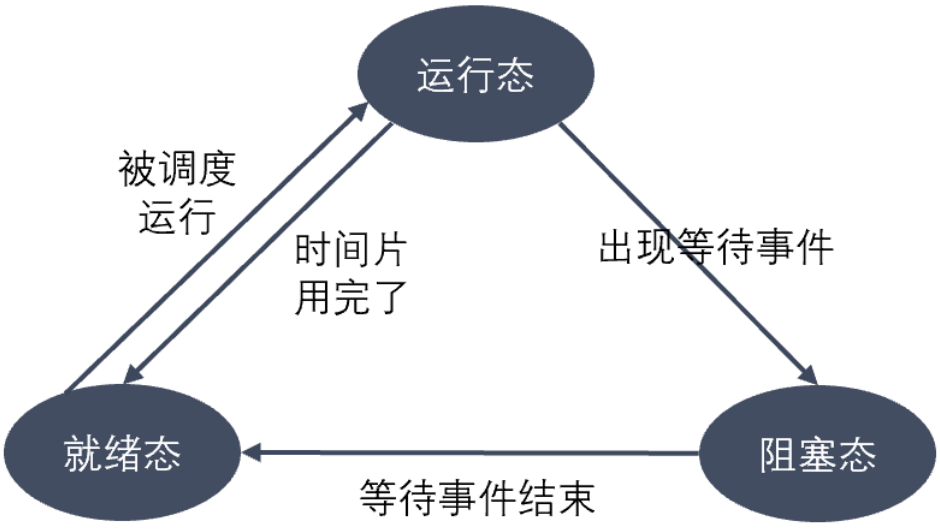
在 /usr/src/linux-headers-xxx/include/linux/sched.h 文件中可以查看 struct task_struct 结构体定义。其内部成员有很多，我们只需要掌握以下部分即可：

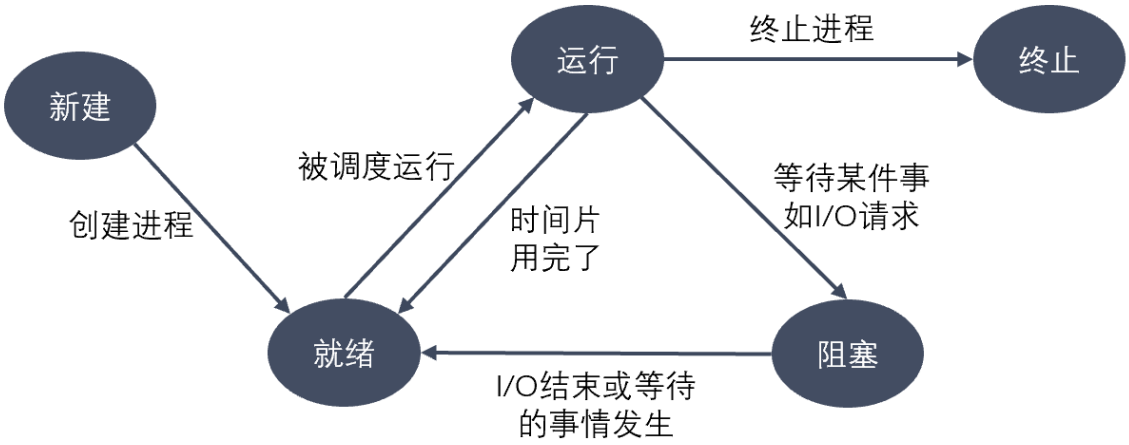
- 进程id：系统中每个进程有唯一的 id，用 pid_t 类型表示，其实就是一个非负整数
- 进程的状态：有就绪、运行、挂起、停止等状态
- 进程切换时需要保存和恢复的一些CPU寄存器
- 描述虚拟地址空间的信息
- 描述控制终端的信息
- 当前工作目录（Current Working Directory）
- umask 掩码
- 文件描述符表，包含很多指向 file 结构体的指针
- 和信号相关的信息
- 用户 id 和组 id
- 会话（Session）和进程组
- 进程可以使用的资源上限（Resource Limit）

2. 进程的状态

进程状态反映进程执行过程的变化。这些状态随着进程的执行和外界条件的变化而转换。在三态模型中，进程状态分为三个基本状态，即就绪态，运行态，阻塞态。在五态模型中，进程分为新建态、就绪态，运行态，阻塞态，终止态。

- 运行态：进程占有处理器正在运行
- 就绪态：进程具备运行条件，等待系统分配处理器以便运行。当进程已分配到除CPU以外的所有必要资源后，只要再获得CPU，便可立即执行。在一个系统中处于就绪状态的进程可能有多个，通常将它们排成一个队列，称为就绪队列
- 阻塞态：又称为等待(wait)态或睡眠(sleep)态，指进程不具备运行条件，正在等待某个事件的完成





- 新建态：进程刚被创建时的状态，尚未进入就绪队列
- 终止态：进程完成任务到达正常结束点，或出现无法克服的错误而异常终止，或被操作系统及有终止权的进程所终止时所处的状态。进入终止态的进程以后不再执行，但依然保留在操作系统中等待善后。一旦其他进程完成了对终止态进程的信息抽取之后，操作系统将删除该进程。

3. 进程相关指令

查看进程

```
ps aux / ajx
a：显示终端上的所有进程，包括其他用户的进程
u：显示进程的详细信息
x：显示没有控制终端的进程
j：列出与作业控制相关的信息
```

STAT参数意义：

```
D          不可中断 Uninterruptible（usually IO）
R          正在运行，或在队列中的进程
S          处于休眠状态
T          停止或被追踪
Z          僵尸进程
W          进入内存交换（从内核2.6开始无效）
X          死掉的进程
<          高优先级
N          低优先级
s          包含子进程
+          位于前台的进程组
```

实时显示进程动态

```
top
```

可以在使用 top 命令时加上 -d 来指定显示信息更新的时间间隔，在 top 命令执行后，可以按以下按键对显示的结果进行排序：

- M 根据内存使用量排序
- P 根据 CPU 占有率排序
- T 根据进程运行时间长短排序

- U 根据用户名来筛选进程
- K 输入指定的 PID 杀死进程

杀死进程

```
kill [-signal] pid
kill -l 列出所有信号
kill -SIGKILL 进程ID
kill -9 进程ID
killall name 根据进程名杀死进程
```

4. 进程号相关函数

每个进程都由进程号来标识，其类型为 pid_t（整型），进程号的范围：0～32767。进程号总是唯一的，但可以重用。当一个进程终止后，其进程号就可以再次使用。

任何进程（除 init 进程）都是由另一个进程创建，该进程称为被创建进程的父进程，对应的进程号称为父进程号（PPID）。

进程组是一个或多个进程的集合。他们之间相互关联，进程组可以接收同一终端的各种信号，关联的进程有一个进程组号（PGID）。默认情况下，当前的进程号会当做当前的进程组号。

进程号和进程组相关函数：

```
pid_t getpid(void);
pid_t getppid(void);
pid_t getpgid(pid_t pid);
```

5. 进程创建

系统允许一个进程创建新进程，新进程即为子进程，子进程还可以创建新的子进程，形成进程树结构模型。

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

返回值：

成功：子进程中返回 0，父进程中返回子进程 ID

失败：返回 -1

失败的两个主要原因：

当前系统的进程数已经达到了系统规定的上限，这时 errno 的值被设置为 EAGAIN

系统内存不足，这时 errno 的值被设置为 ENOMEM

父子进程虚拟地址空间

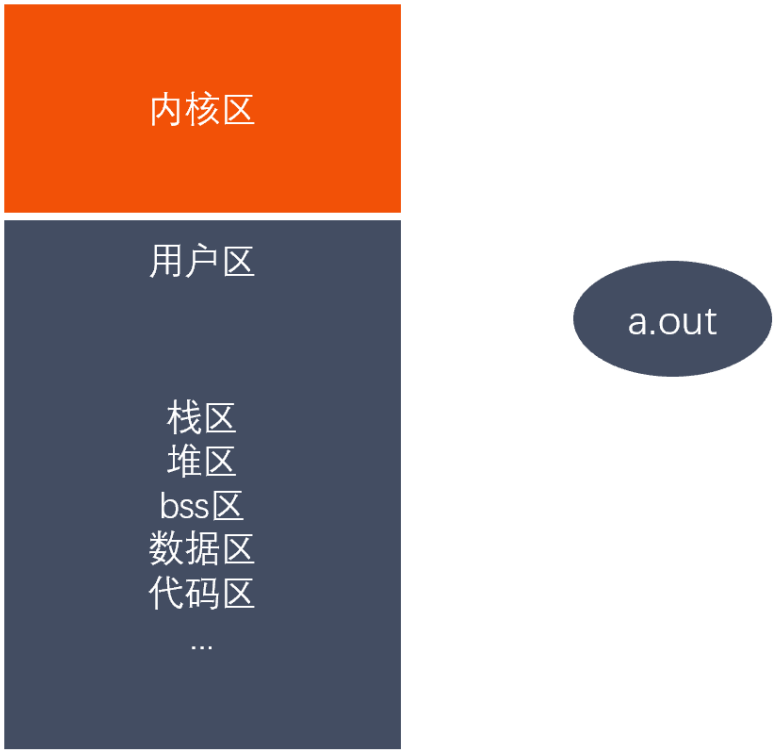
实际上，更准确来说，Linux 的 fork() 使用是通过写时拷贝 (copy- on-write) 实现。写时拷贝是一种可以推迟甚至避免拷贝数据的技术。内核此时并不复制整个进程的地址空间，而是让父子进程共享同一个地址空间。只用在需要写入的时候才会复制地址空间，从而使各个进行拥有各自的地址空间。也就是说，资源的复制是在需要写入的时候才会进行，在此之前，只有以只读方式共享。

注意：fork之后父子进程共享文件，fork产生的子进程与父进程相同的文件文件描述符指向相同的文件表，引用计数增加，共享文件偏移指针。

6. exec 函数族介绍

exec 函数族的作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的内容，换句话说，就是在调用进程内部执行一个可执行文件。

exec 函数族的函数执行成功后不会返回，因为调用进程的实体，包括代码段，数据段和堆栈等都已经被新的内容取代，只留下进程 ID 等一些表面上的信息仍保持原样，颇有些神似“三十六计”中的“金蝉脱壳”。看上去还是旧的躯壳，却已经注入了新的灵魂。只有调用失败了，它们才会返回 -1，从原程序的调用点接着往下执行。



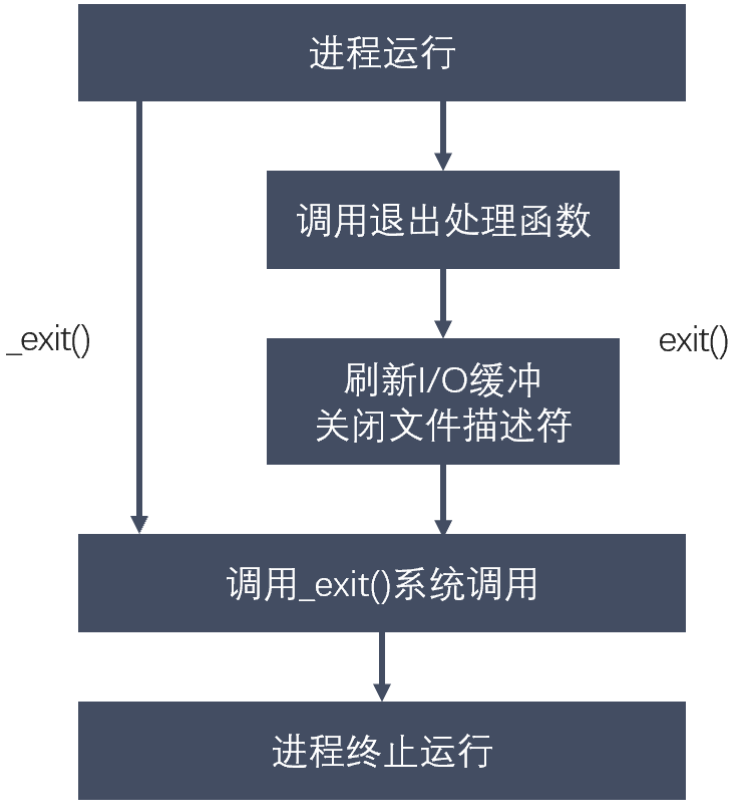

```
int execl(const char *path, const char *arg, .../* (char  *) NULL */);
int execlp(const char *file, const char *arg, ... /* (char  *) NULL */);
int execle(const char *path, const char *arg, .../*, (char *) NULL, char * const
envp[] */);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
int execve(const char *filename, char *const argv[], char *const envp[]);
```

- l(list)　　参数地址列表，以空指针结尾
- v(vector)　　存有各参数地址的指针数组的地址
- p(path)　　按 PATH 环境变量指定的目录搜索可执行文件
- e(environment)　存有环境变量字符串地址的指针数组的地址

7. 进程控制

进程退出

```
#include <stdlib.h>
void exit(int status);
#include <unistd.h>
void _exit(int status);
```



孤儿进程

父进程运行结束，但子进程还在运行（未运行结束），这样的子进程就称为孤儿进程（Orphan Process）。

每当出现一个孤儿进程的时候，内核就把孤儿进程的父进程设置为 init ，而 init 进程会循环地 wait() 它的已经退出的子进程。这样，当一个孤儿进程凄凉地结束了其生命周期的时候，init 进程就会代表党和政府出面处理它的一切善后工作。因此孤儿进程并不会有什么危害。

僵尸进程

每个进程结束之后, 都会释放自己地址空间中的用户区数据，内核区的 PCB 没有办法自己释放掉，需要父进程去释放。

进程终止时，父进程尚未回收，子进程残留资源（PCB）存放于内核中，变成僵尸（Zombie）进程。

僵尸进程不能被 kill -9 杀死，这样就会导致一个问题，如果父进程不调用 wait() 或 waitpid() 的话，那么保留的那段信息就不会释放，其进程号就会一直被占用，但是系统所能使用的进程号是有限的，如果大量的产生僵尸进程，将因为没有可用的进程号而导致系统不能产生新的进程，此即为僵尸进程的危害，应当避免

进程回收

在每个进程退出的时候，内核释放该进程所有的资源、包括打开的文件、占用的内存等。但是仍然为其保留一定的信息，这些信息主要主要指进程控制块PCB的信息（包括进程号、退出状态、运行时间等）。

父进程可以通过调用 wait 或 waitpid 得到它的退出状态同时彻底清除掉这个进程。

wait() 和 waitpid() 函数的功能一样，区别在于，wait() 函数会阻塞，waitpid() 可以设置不阻塞，waitpid() 还可以指定等待哪个子进程结束。

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。

退出信息相关宏函数

```
WIFEXITED(status)      非0，进程正常退出
WEXITSTATUS(status)    如果上宏为真，获取进程退出的状态（exit的参数）

WIFSIGNALED(status)    非0，进程异常终止
WTERMSIG(status)       如果上宏为真，获取使进程终止的信号编号

WIFSTOPPED(status)     非0，进程处于暂停状态
WSTOPSIG(status)       如果上宏为真，获取使进程暂停的信号的编号
WIFCONTINUED(status)   非0，进程暂停后已经继续运行
```

8. 进程间通信

进程间通讯概念

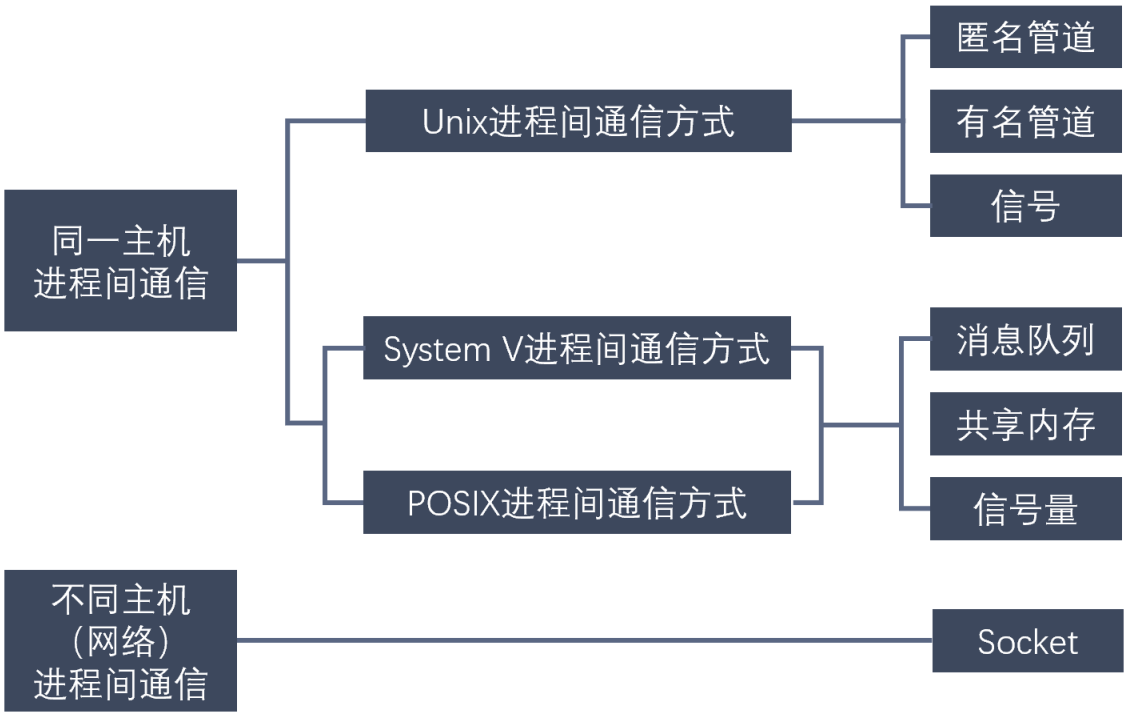
进程是一个独立的资源分配单元，不同进程（这里所说的进程通常指的是用户进程）之间的资源是独立的，没有关联，不能在一个进程中直接访问另一个进程的资源。

但是，进程不是孤立的，不同的进程需要进行信息的交互和状态的传递等，因此需要进程间通信(IPC: Inter Processes Communication)。

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。
- 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（如进程终止时要通知父进程）。
- 资源共享：多个进程之间共享同样的资源。为了做到这一点，需要内核提供互斥和同步机制。
- 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。

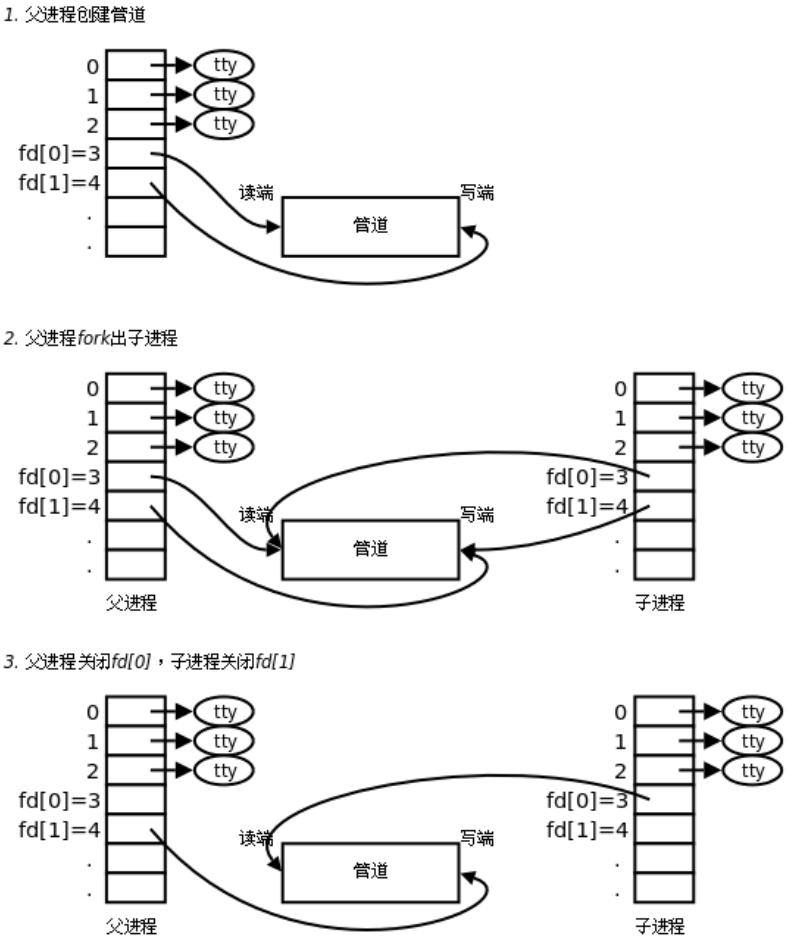
Linux 进程间通信的方式



匿名管道（管道）

管道也叫无名（匿名）管道，它是是 UNIX 系统 IPC（进程间通信）的最古老形式，所有的 UNIX 系统都支持这种通信机制。

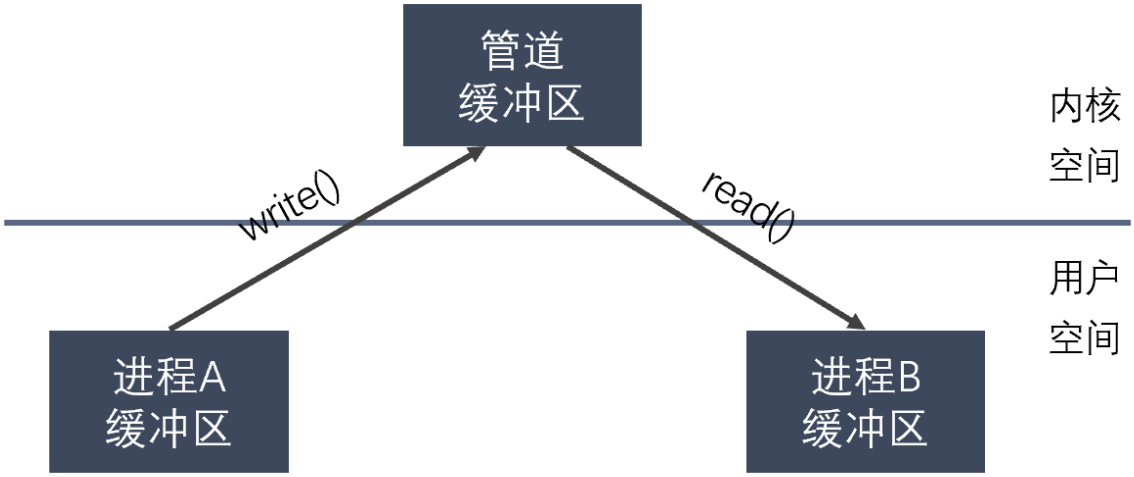
统计一个目录中文件的数目命令：ls | wc –l，为了执行该命令，shell 创建了两个进程来分别执行 ls 和 wc。



管道的特点

- 管道其实是一个在内核内存中维护的缓冲器，这个缓冲器的存储能力是有限的，不同的操作系统大小不一定相同。
- 管道拥有文件的特质：读操作、写操作，匿名管道没有文件实体，有名管道有文件实体，但不存储数据。可以按照操作文件的方式对管道进行操作。
- 一个管道是一个字节流，使用管道时不存在消息或者消息边界的概念，从管道读取数据的进程可以读取任意大小的数据块，而不管写入进程写入管道的数据块的大小是多少。
- 通过管道传递的数据是顺序的，从管道中读取出来的字节的顺序和它们被写入管道的顺序是完全一样的。
- 在管道中的数据的传递方向是单向的，一端用于写入，一端用于读取，管道是半双工的。

- 从管道读数据是一次性操作，数据一旦被读走，它就从管道中被抛弃，释放空间以便写更多的数据，在管道中无法使用 lseek() 来随机的访问数据。
- 匿名管道只能在具有公共祖先的进程（父进程与子进程，或者两个兄弟进程，具有亲缘关系）之间使用。



匿名管道的使用

创建匿名管道

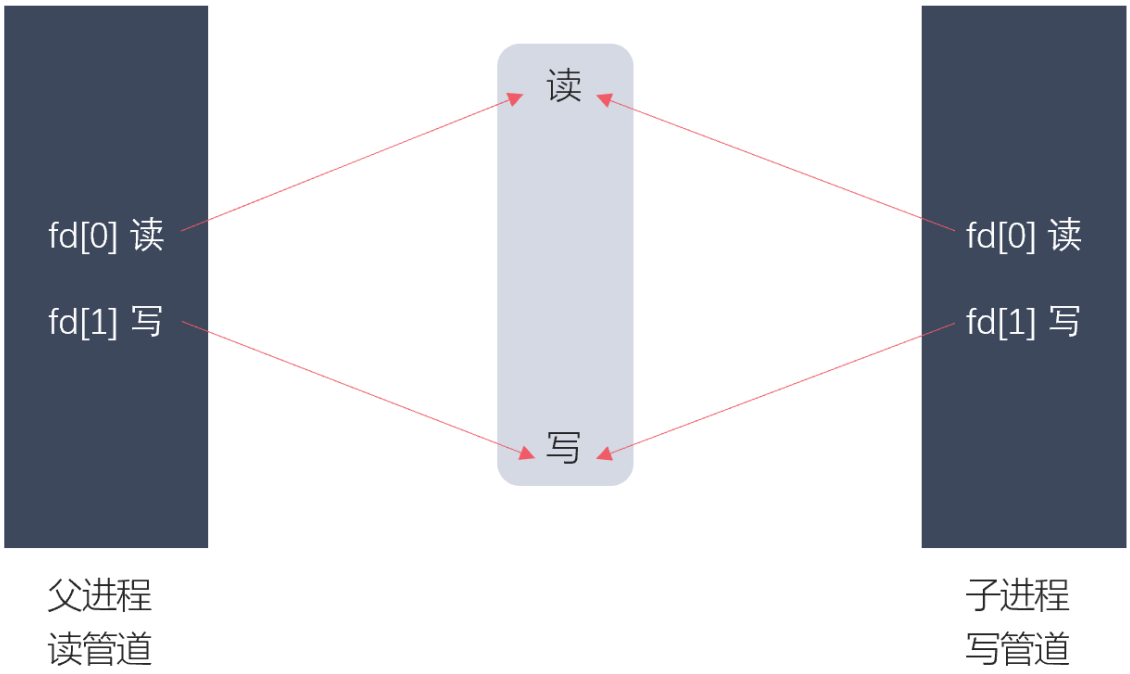
```
#include <unistd.h>
int pipe(int pipefd[2]);
```

查看管道缓冲大小命令

```
ulimit -a
```

查看管道缓冲大小函数

```
#include <unistd.h>
long fpathconf(int fd, int name);
```



有名管道（命名管道）

介绍

1. 匿名管道，由于没有名字，只能用于亲缘关系的进程间通信。为了克服这个缺点，提出了有名管道（FIFO），也叫命名管道、FIFO文件。
2. 有名管道（FIFO）不同于匿名管道之处在于它提供了一个路径名与之关联，以 FIFO 的文件形式存在于文件系统中，并且其打开方式与打开一个普通文件是一样的，这样即使与 FIFO 的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过 FIFO 相互通信，因此，通过 FIFO 不相关的进程也能交换数据。
3. 一旦打开了 FIFO，就能在它上面使用与操作匿名管道和其他文件的系统调用一样的I/O系统调用了（如read()、write()和close()）。与管道一样，FIFO 也有一个写入端和读取端，并且从管道中读取数据的顺序与写入的顺序是一样的。FIFO 的名称也由此而来：先入先出。
4. 有名管道（FIFO)和匿名管道（pipe）有一些特点是相同的，不一样的地方在于：
    - FIFO 在文件系统中作为一个特殊文件存在，但 FIFO 中的内容却存放在内存中。
    - 当使用 FIFO 的进程退出后，FIFO 文件将继续保存在文件系统中以便以后使用。
    - FIFO 有名字，不相关的进程可以通过打开有名管道进行通信。

使用

通过命令创建有名管道

```
mkfifo 名字
```

通过函数创建有名管道

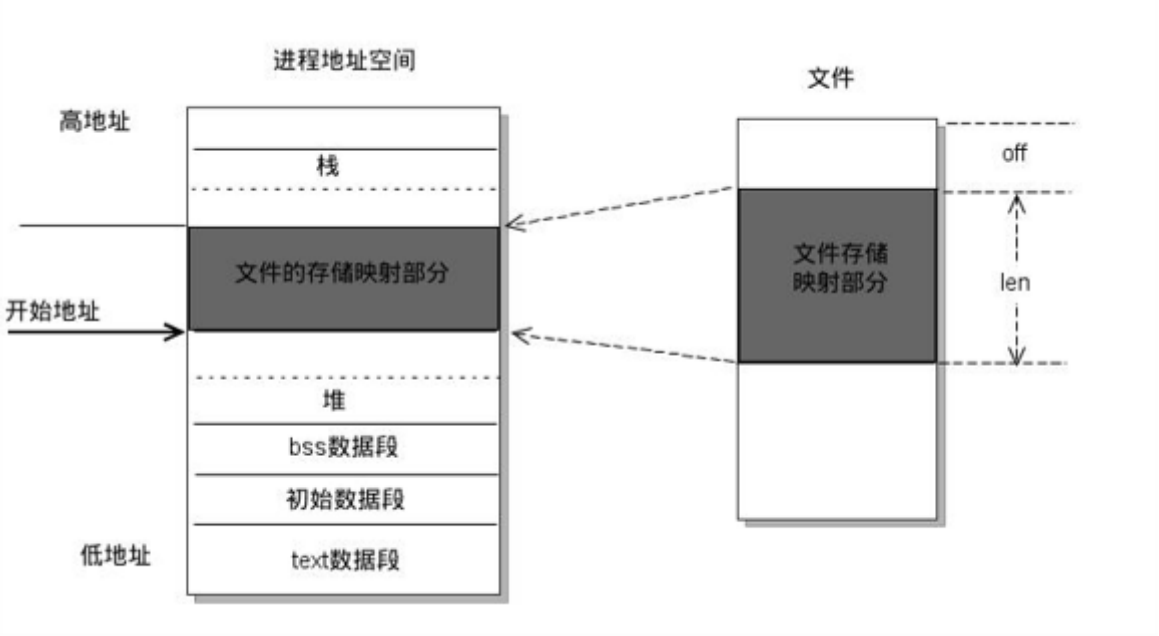
```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

一旦使用 mkfifo 创建了一个 FIFO，就可以使用 open 打开它，常见的文件I/O 函数都可用于 fifo。如：close、read、write、unlink 等。

FIFO 严格遵循先进先出（First in First out），对管道及 FIFO 的读总是从开始处返回数据，对它们的写则把数据添加到末尾。它们不支持诸如 lseek() 等文件定位操作。

内存映射

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。




```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t
offset);
int munmap(void *addr, size_t length);
```

信号

概述

信号是 Linux 进程间通信的最古老的方式之一，是事件发生时对进程的通知机制，有时也称之为软件中断，它是在软件层次上对中断机制的一种模拟，是一种异步通信的方式。信号可以导致一个正在运行的进程被另一个正在运行的异步进程中断，转而处理某一个突发事件。

发往进程的诸多信号，通常都是源于内核。引发内核为进程产生信号的各类事件如下：

- 对于前台进程，用户可以通过输入特殊的终端字符来给它发送信号。比如输入Ctrl+C 通常会给进程发送一个中断信号。
- 硬件发生异常，即硬件检测到一个错误条件并通知内核，随即再由内核发送相应信号给相关进程。比如执行一条异常的机器语言指令，诸如被 0 除，或者引用了无法访问的内存区域。
- 系统状态变化，比如 alarm 定时器到期将引起 SIGALRM 信号，进程执行的 CPU 时间超限，或者该进程的某个子进程退出。
- 运行 kill 命令或调用 kill 函数。

使用信号的两个主要目的是：

- 让进程知道已经发生了一个特定的事情。
- 强迫进程执行它自己代码中的信号处理程序。

信号的特点：

- 简单
- 不能携带大量信息
- 满足某个特定条件才发送
- 优先级比较高

查看系统定义的信号列表：kill –l

前 31 个信号为常规信号，其余为实时信号。

Linux 信号一览表

编号	信号名称	对应事件	默认动作
1	SIGHUP	用户退出shell时，由该shell启动的所有进程将收到这个信号	终止进程
2	**SIGINT**	当用户按下了<Ctrl+C>组合键时，用户终端向正在运行中的由该终端启动的程序发出此信号	终止进程
3	**SIGQUIT**	用户按下<Ctrl+>组合键时产生该信号，用户终端向正在运行中的由该终端启动的程序发出些信号	终止进程
4	SIGILL	CPU检测到某进程执行了非法指令	终止进程并产生core文件
5	SIGTRAP	该信号由断点指令或其他 trap指令产生	终止进程并产生core文件
6	SIGABRT	调用abort函数时产生该信号	终止进程并产生core文件
7	SIGBUS	非法访问内存地址，包括内存对齐出错	终止进程并产生core文件
8	SIGFPE	在发生致命的运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为0等所有的算法错误	终止进程并产生core文件
9	**SIGKILL**	无条件终止进程。该信号不能被忽略，处理和阻塞	终止进程，可以杀死任何进程
10	SIGUSE1	用户定义的信号。即程序员可以在程序中定义并使用该信号	终止进程
11	**SIGSEGV**	指示进程进行了无效内存访问(段错误)	终止进程并产生core文件
12	SIGUSR2	另外一个用户自定义信号，程序员可以在程序中定义并使用该信号	终止进程
13	**SIGPIPE**	Broken pipe向一个没有读端的管道写数据	终止进程
14	SIGALRM	定时器超时，超时的时间 由系统调用alarm设置	终止进程
15	SIGTERM	程序结束信号，与SIGKILL不同的是，该信号可以被阻塞和终止。通常用来要示程序正常退出。执行shell命令Kill时，缺省产生这个信号	终止进程
16	SIGSTKFLT	Linux早期版本出现的信号，现仍保留向后兼容	终止进程

编号	信号名称	对应事件	默认动作
17	**SIGCHLD**	子进程结束时，父进程会收到这个信号	忽略这个信号
18	**SIGCONT**	如果进程已停止，则使其继续运行	继续/忽略
19	**SIGSTOP**	停止进程的执行。信号不能被忽略，处理和阻塞	为终止进程
20	SIGTSTP	停止终端交互进程的运行。按下<ctrl+z>组合键时发出这个信号	暂停进程
21	SIGTTIN	后台进程读终端控制台	暂停进程
22	SIGTTOU	该信号类似于SIGTTIN，在后台进程要向终端输出数据时发生	暂停进程
23	SIGURG	套接字上有紧急数据时，向当前正在运行的进程发出些信号，报告有紧急数据到达。如网络带外数据到达	忽略该信号
24	SIGXCPU	进程执行时间超过了分配给该进程的CPU时间，系统产生该信号并发送给该进程	终止进程
25	SIGXFSZ	超过文件的最大长度设置	终止进程
26	SIGVTALRM	虚拟时钟超时时产生该信号。类似于SIGALRM，但是该信号只计算该进程占用CPU的使用时间	终止进程
27	SGIPROF	类似于SIGVTALRM，它不公包括该进程占用CPU时间还包括执行系统调用时间	终止进程
28	SIGWINCH	窗口变化大小时发出	忽略该信号
29	SIGIO	此信号向进程指示发出了一个异步IO事件	忽略该信号
30	SIGPWR	关机	终止进程
31	SIGSYS	无效的系统调用	终止进程并产生core文件
34~64	SIGRTMIN ~ SIGRTMAX	LINUX的实时信号，它们没有固定的含义（可以由用户自定义）	终止进程

信号的 5 种默认处理动作

查看信号的详细信息：man 7 signal

信号的 5 中默认处理动作

- Term 终止进程
- Ign 当前进程忽略掉这个信号
- Core 终止进程，并生成一个Core文件

- Stop　暂停当前进程
- Cont　继续执行当前被暂停的进程

信号的几种状态：产生、未决、递达

SIGKILL 和 SIGSTOP 信号不能被捕捉、阻塞或者忽略，只能执行默认动作。

信号相关的函数

```
int kill(pid_t pid, int sig);
int raise(int sig);
void abort(void);
unsigned int alarm(unsigned int seconds);
int setitimer(int which, const struct itimerval *new_val, struct itimerval
*old_value);
```

信号捕捉函数

```
sighandler_t signal(int signum, sighandler_t handler);
int sigaction(int signum, const struct sigaction *act, struct sigaction
*oldact);
```

信号集

许多信号相关的系统调用都需要能表示一组不同的信号，多个信号可使用一个称之为信号集的数据结构来表示，其系统数据类型为 sigset_t。

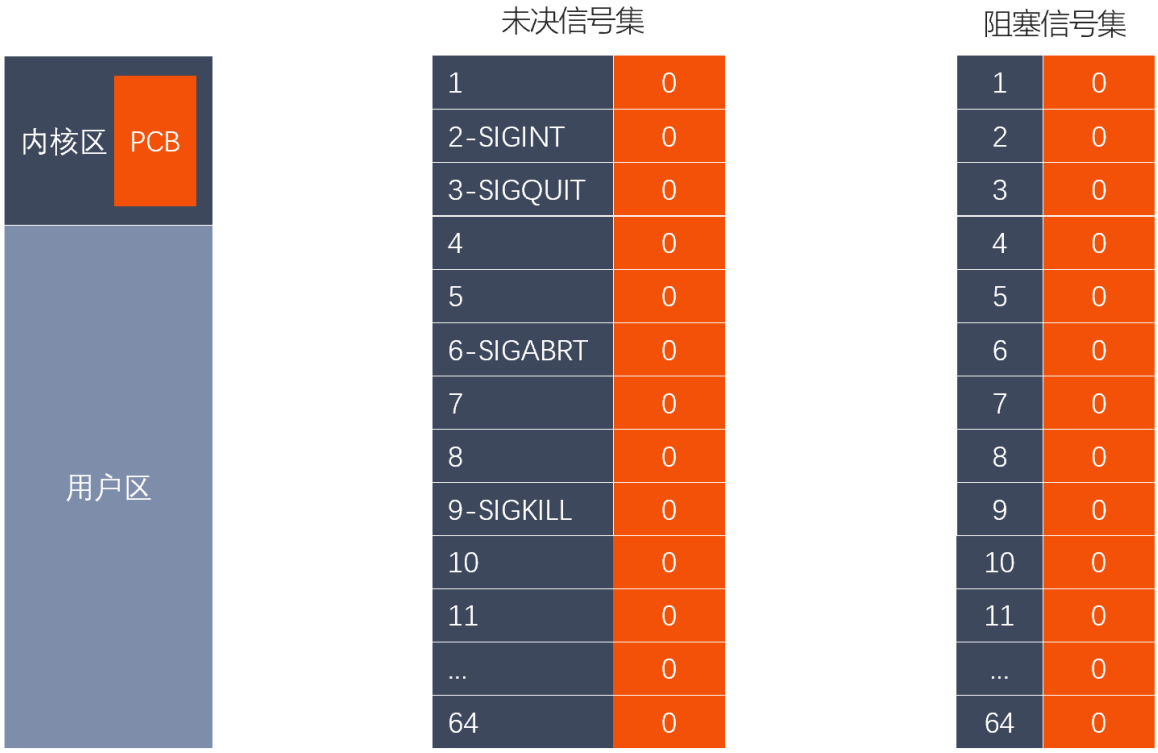
在 PCB 中有两个非常重要的信号集。一个称之为 “阻塞信号集” ，另一个称之为 “未决信号集” 。这两个信号集都是内核使用位图机制来实现的。但操作系统不允许我们直接对这两个信号集进行位操作。而需自定义另外一个集合，

借助信号集操作函数来对 PCB 中的这两个信号集进行修改。

信号的 “未决” 是一种状态，指的是从信号的产生到信号被处理前的这一段时间。

信号的 “阻塞” 是一个开关动作，指的是阻止信号被处理，但不是阻止信号产生。

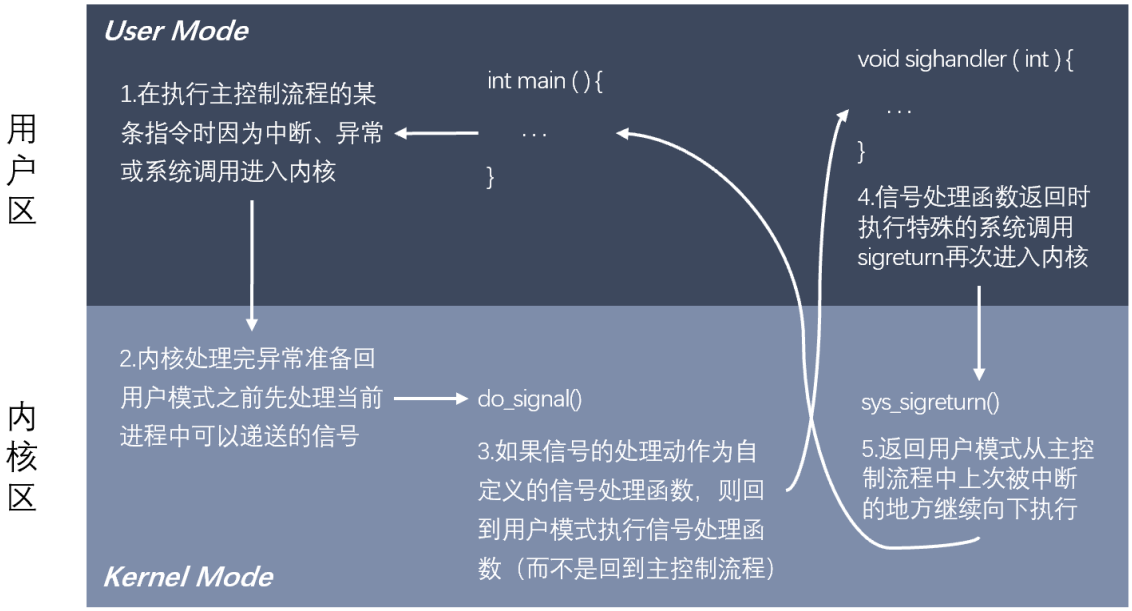
信号的阻塞就是让系统暂时保留信号留待以后发送。由于另外有办法让系统忽略信号，所以一般情况下信号的阻塞只是暂时的，只是为了防止信号打断敏感的操作。



信号集相关操作函数:

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

内核实现信号捕捉的过程



SIGCHILD信号

SIGCHLD信号产生的条件

- 子进程终止时
- 子进程接收到 SIGSTOP 信号停止时
- 子进程处在停止态，接受到SIGCONT后唤醒时

以上三种条件都会给父进程发送 SIGCHLD 信号，父进程默认会忽略该信号

共享内存

共享内存允许两个或者多个进程共享物理内存的同一块区域（通常被称为段）。由于一个共享内存段会称为一个进程用户空间的一部分，因此这种 IPC 机制无需内核介入。所有需要做的就是让一个进程将数据复制进共享内存中，并且这部分数据会对其他所有共享同一个段的进程可用。

与管道等要求发送进程将数据从用户空间的缓冲区复制进内核内存和接收进程将数据从内核内存复制进用户空间的缓冲区的做法相比，这种 IPC 技术的速度更快。

使用步骤

- 调用 shmget() 创建一个新共享内存段或取得一个既有共享内存段的标识符（即由其他进程创建的共享内存段）。这个调用将返回后续调用中需要用到的共享内存标识符。
- 使用 shmat() 来附上共享内存段，即使该段成为调用进程的虚拟内存的一部分。
  此刻在程序中可以像对待其他可用内存那样对待这个共享内存段。为引用这块共享内存，程序需要使用由 shmat()
- 调用返回的 addr 值，它是一个指向进程的虚拟地址空间中该共享内存段的起点的指针。
- 调用 shmdt() 来分离共享内存段。在这个调用之后，进程就无法再引用这块共享内存了。这一步是可选的，并且在进程终止时会自动完成这一步。
- 调用 shmctl() 来删除共享内存段。只有当当前所有附加内存段的进程都与之分离之后内存段才会销毁。只有一个进程需要执行这一步。

相关函数

```
int shmget(key_t key, size_t size, int shmflg);
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
key_t ftok(const char *pathname, int proj_id);
```

共享内存操作命令

ipcs 用法

- ipcs -a　　// 打印当前系统中所有的进程间通信方式的信息
- ipcs -m　　// 打印出使用共享内存进行进程间通信的信息
- ipcs -q　　// 打印出使用消息队列进行进程间通信的信息
- ipcs -s　　// 打印出使用信号进行进程间通信的信息

ipcrm 用法

- ipcrm -M shmkey　// 移除用shmkey创建的共享内存段
- ipcrm -m shmid　　// 移除用shmid标识的共享内存段
- ipcrm -Q msgkey　// 移除用msqkey创建的消息队列
- ipcrm -q msqid　// 移除用msqid标识的消息队列
- ipcrm -S semkey　// 移除用semkey创建的信号
- ipcrm -s semid　// 移除用semid标识的信号