**2. 操作系统**

**2.1 进程间通信方式**

**管道( pipe )：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。**

**命名管道 (FIFO) ： 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。**

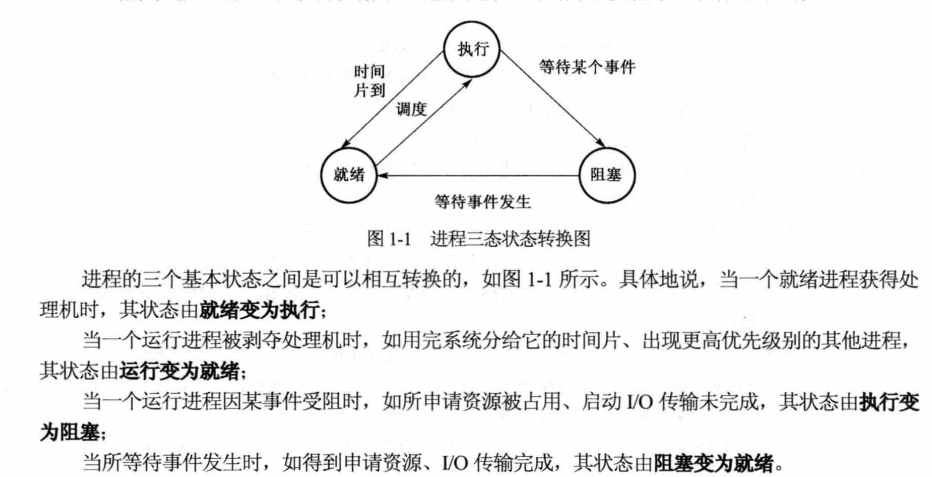
**信号量：信号量用于实现进程间的互斥与同步，也可以用在线程上，主要有posix信号量和System V信号量，posix信号量一般用在线程上，System V信号量一般用在进程上，posix信号量的函数一般都在下划线。**

**消息队列( message queue ) ： 消息队列是由消息的链表，存放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息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。(优先级，大小)**

**共享内存( shared memory ) ：共享内存就是映射一段能被其他进程所访问的内存，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运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。**

**信号 ( sinal ) ： 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生，常见的信号。**

**套接字( socket ) ： 套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信。**



**2.2 常见信号有哪些？**

SIGINT，SIGKILL(不能被捕获)，SIGSTOP(不能被捕获)、SIGTERM(可以被捕获)，SIGSEGV，SIGCHLD，SIGALRM

**2.3 标准io和文件io的区别**

1.定义

　　标准ＩＯ：具有一定的可移植性。标准IO库处理很多细节。例如缓存分配，以优化长度执行IO等。标准的IO提供了三种类型的缓存。

（1）全缓存：当填满标准IO缓存后才进行实际的IO操作。

（2）行缓存：当输入或输出中遇到新行符时，标准IO库执行IO操作。

（3）不带缓存：stderr就是了。

　　文件ＩＯ：文件ＩＯ称之为不带缓存的IO（unbuffered I/O)。不带缓存指的是每个read，write都调用内核中的一个系统调用。也就是一般所说的低级I/O——操作系统提供的基本IO服务，与os绑定，特定于Unix平台。

2.区别

　　首先：两者一个显著的不同点在于，标准I/O默认采用了缓冲机制，比如调用fopen函数，不仅打开一个文件，而且建立了一个缓冲区（读写模式下将建立两个缓冲区），还创建了一个包含文件和缓冲区相关数据的数据结构(FILE \*)。低级I/O一般没有采用缓冲，需要自己创建缓冲区，不过其实在linux系统中，都是有使用称为内核缓冲的技术用于提高效率，读写调用是在内核缓冲区和进程缓冲区之间进行的数据复制。使用标准IO就不需要自己维护缓冲区了，标准IO库会根据stdin/stdout来选择缓冲类型，也就是说当你使用标准IO的时候，要清楚它的stdin/stdou是什么类型以及其默认的缓冲模式，如果不合适，你需要用setvbuf先设置，再使用，例如协同进程的标准输入和输出的类型都是管道，所以其默认的缓冲类型是全缓冲的，如果要使用标准IO，就需要现设置行缓冲。对于文件IO，只要你自己能维护好缓冲区，完全可以不用标准IO。

　　其次从名字上来区分，文件I/O主要针对文件操作，读写硬盘等，标准I/O，主要是打印输出到屏幕等。因为他们设备不一样，文件io针对的是文件，标准io是对控制台，操作的是字符流。对于不同设备得特性不一样，必须有不同api访问才最高效。

**2.4 wait和waitpid区别**

　wait会令调用者阻塞直至某个子进程终止；

　　waitpid则可以通过设置一个选项来设置为非阻塞，另外waitpid并不是等待第一个结束的进程而是等待参数中pid指定的进程。

**waitpid中pid的含义依据其具体值而变**：

　　pid==-1 等待任何一个子进程，此时waitpid的作用与wait相同

　　pid >0   等待进程ID与pid值相同的子进程

　　pid==0   等待与调用者进程组ID相同的任意子进程

　　pid<-1   等待进程组ID与pid绝对值相等的任意子进程

**waitpid提供了wait所没有的三个特性**：

1 waitpid使我们可以等待指定的进程

2 waitpid提供了一个无阻塞的wait

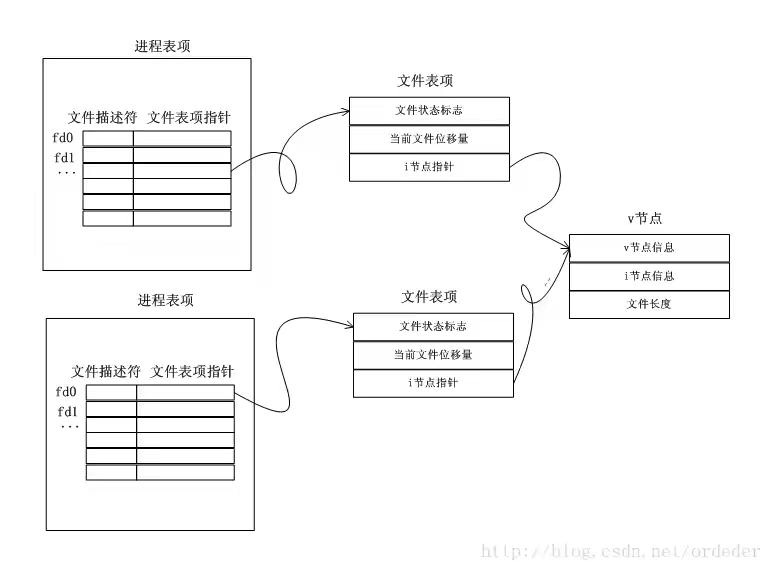
3 waitpid支持工作控制

**2.5 父进程fork后父子进程共享的内容**

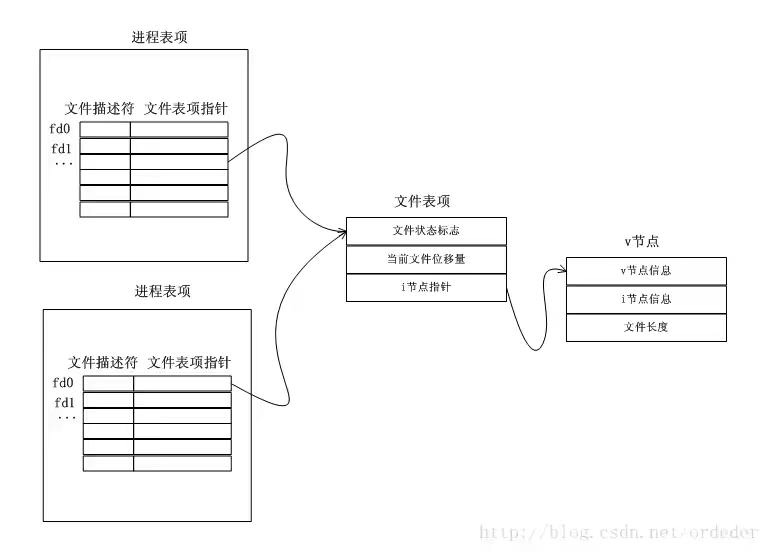
fork之后，子进程会拷贝父进程的数据空间、堆和栈空间（实际上是采用写时复制技术），二者共享代码段。 所以在子进程中修改全局变量（局部变量，分配在堆上的内存同样也是）后，父进程的相同的全局变量不会改变。

共享fd，以及fd对应的文件表项。

不同进程打开同一个文件



进程的fork与文件描述符的拷贝 进程的所打开文件和在fork后的结构图如下所示，子进程是共享父进程的文件表项。



**2.6 ELF文件的理解**

**section 是被链接器使用的，但是 segments 是被加载器所使用的。加载器会将所需要的 segment 加载到内存空间中运行。**

1) 可重定位的对象文件(Relocatable file)(没有segments)

这是由汇编器汇编生成的 .o 文件。后面的链接器(link editor)拿一个或一些 Relocatable object files 作为输入，经链接处理后，生成一个可执行的对象文件 (Executable file) 或者一个可被共享的对象文件(Shared object file)。我们可以使用 ar 工具将众多的 .o Relocatable object files 归档(archive)成 .a 静态库文件。

2) 可执行的对象文件(Executable file)

3) 可被共享的对象文件(Shared object file)

这些就是所谓的动态库文件，也即 .so 文件。

在ELF文件里面，每一个 sections 内都装载了性质属性都一样的内容，比方：

1) .text section 里装载了可执行代码；

2) .data section 里面装载了被初始化的数据；

3) .bss section 里面装载了未被初始化的数据；

4) 以 .rec 打头的 sections 里面装载了重定位条目；

5) .symtab 或者 .dynsym section 里面装载了符号信息；

6) .strtab 或者 .dynstr section 里面装载了字符串信息；

7) 其他还有为满足不同目的所设置的section，比方满足调试的目的、满足动态链接与加载的目的等等。

把带有相同属性(比方都是只读并可加载的)的 section 都合并成所谓 segments(段)。最重要的是三个 segment：代码段，数据段和堆栈段。

**2.7 linux程序启动过程**

当你在 shell 中敲入一个命令要执行时，内核会帮我们创建一个新的进程，它在往这个新进程的进程空间里面加载进可执行程序的代码段和数据段后，也会加载进动态连接器(在Linux里面通常就是 /lib/ld-linux.so 符号链接所指向的那个程序，它本省就是一个动态库)的代码段和数据。在这之后，内核将控制传递给动态链接库里面的代码。动态连接器接下来负责加载该命令应用程序所需要使用的各种动态库。加载完毕，动态连接器才将控制传递给应用程序的main函数。如此，你的应用程序才得以运行。(过程链接表（PLT）,   Global Offset Table（GOT）)

**2.8 产生死锁的四个必要条件：**

（1） 互斥条件：一个资源每次只能被一个进程使用。

（2） 占有且等待：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

（3）不可强行占有:进程已获得的资源，在末使用完之前，不能强行剥夺。

（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之

一不满足，就不会发生死锁。

**2.9 处理死锁的基本方法**：

\***死锁预防**：通过设置某些限制条件，去破坏死锁的四个条件中的一个或几个条件，来预防发生死锁。但由于所施加的限制条件往往太严格，因而导致系统资源利用率和系统吞吐量降低。

\***死锁避免**：允许前三个必要条件，但通过明智的选择，确保永远不会到达死锁点，因此死锁避免比死锁预防允许更多的并发。

\***死锁检测**：不须实现采取任何限制性措施，而是允许系统在运行过程发生死锁，但可通过系统设置的检测机构及时检测出死锁的发生，并精确地确定于死锁相关的进程和资源，然后采取适当的措施，从系统中将已发生的死锁清除掉。

\***死锁解除**：与死锁检测相配套的一种措施。当检测到系统中已发生死锁，需将进程从死锁状态中解脱出来。常用方法：撤销或挂起一些进程，以便回收一些资源，再将这些资源分配给已处于阻塞状态的进程。死锁检测盒解除有可能使系统获得较好的资源利用率和吞吐量，但在实现上难度也最大。

**2.10 vfork函数**

vfork() 子进程与父进程共享数据段.  vfork()保证子进程先运行，在她调用exec或\_exit之后父进程才可能被调度运行。如果在调用这两个函数之前子进程依赖于父进程的进一步动作，则会导致死锁。

**2.11 线程进程的区别体现在几个方面**

1.因为进程拥有独立的堆栈空间和数据段，所以每当启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，系统开销比较大，而线程不一样，线程拥有独立的堆栈空间，但是共享数据段，它们彼此之间使用相同的地址空间，共享大部分数据，切换速度也比进程快，效率高，但是正由于进程之间独立的特点，使得进程安全性比较高，也因为进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。一个线程死掉就等于整个进程死掉。

2.体现在通信机制上面，正因为进程之间互不干扰，相互独立，进程的通信机制相对很复杂，譬如管道，信号，消息队列，共享内存，套接字等通信机制，而线程由于共享数据段所以通信机制很方便。。

3.属于同一个进程的所有线程共享该进程的所有资源，包括文件描述符。而不同过的进程相互独立。

4.线程必定也只能属于一个进程，而进程可以拥有多个线程而且至少拥有一个线程；

* 进程是资源分配的最小单位，线程是程序执行的最小单位。
* 进程有自己的独立地址空间，每启动一个进程，系统就会为它分配地址空间，建立数据表来维护代码段、堆栈段和数据段，这种操作非常昂贵。而线程是共享进程中的数据的，使用相同的地址空间，因此CPU切换一个线程的花费远比进程要小很多，同时创建一个线程的开销也比进程要小很多。
* 线程之间的通信更方便，同一进程下的线程共享全局变量、静态变量等数据，而进程之间的通信需要以通信的方式（IPC)进行。
* 但是多进程程序更健壮，多线程程序只要有一个线程死掉，整个进程也死掉了，而一个进程死掉并不会对另外一个进程造成影响，因为进程有自己独立的地址空间。

**2.12 进程与线程的选择取决以下几点**

1、需要频繁创建销毁的优先使用线程；因为对进程来说创建和销毁一个进程代价是很大的。

2、线程的切换速度快，所以在需要大量计算，切换频繁时用线程，还有耗时的操作使用线程可提高应用程序的响应

3、因为对CPU系统的效率使用上线程更占优，所以可能要发展到多机分布的用进程，多核分布用线程；

4、并行操作时使用线程，如C/S[架构](http://lib.csdn.net/base/architecture)的服务器端并发线程响应用户的请求；

5、需要更稳定安全时，适合选择进程；需要速度时，选择线程更好。

**2.13 守护进程**

守护进程是运行在后台的一种特殊进程, 不受终端控制，Linux系统的大多数服务器就是通过守护进程实现的。一个守护进程的父进程是init进程。

1）创建子进程，父进程退出

2）在子进程中创建新会话

3）改变当前目录为根目

4）重设文件权限掩码

5) 关闭文件描述符

**2.14 Linux开机流程**

加载BIOS的硬件信息与进行自我测试，并依据设置取得第一个可启动设备；

读取并执行第一个启动设备内MBR（主引导分区）的Boot Loader（即是gurb等程序）；

依据Boot Loader的设置加载Kernel，Kernel会开始检测硬件与加载驱动程序；

在硬件驱动成功后，Kernel会主动调用init进程（/sbin/init），而init会取得runlevel信息；

init执行/etc/rc.d/rc.sysinit文件来准备软件的操作环境（如网络、时区等）；

init执行runlevel的各个服务的启动（script方式）；

init执行/etc/rc.d/rc.local文件；

        init执行终端机模拟程序mingetty来启动login程序，最后等待用户登录。