信号量不是信号，信号不是信号量----这两个是完全不同的东西...
进程信号:
作用:通知别人，发生了某件事情，尽快的去处理这件事情(操作系统通知进程发生了某个事件，打断进程当前操作，去处理这个事件)
是什么:软件中断
信号想要成为一个中断，首先我们必须认识这个信号,并且知道如何去处理它
事件多种多样，因此信号也是多种多样

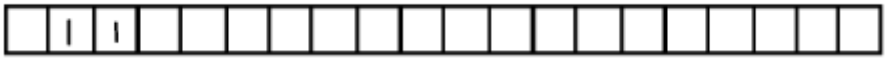
查看操作系统中定义好的信号:使用kil -l命令可以查看信号种类:
用户所能看到并使用的信号共有62种，两大分类:
1~31号信号:每个信号都有具体对应的事件---- 非可靠信号
34~64号信号:在操作系统中当前就没有具体对应的事件了，因此命名也稍微草率一点---- 可靠信号

信号的生命周期:信号的产生->信号在进程中注册->信号在进程中注销->处理信号　　信号的阻塞
信号的产生:
硬件: ctrl+c ctrl+z
软件:
命令: kill -signum pid
kill命令能够杀死一个进程，主要是因为ill命令功能是向进程发送一个信号，默认发送的15号终止信号函数:
int kl(pid_t pid, int sig);向指定的进程发送指定的信号
int raise(int sig):向调用进程自身发送指定的信号
void abort(void);向调用进程自身发送SIGABRT信号,使-个异常的进程退出
unsigned int alarm(unsigned int seconds); seconed秒之后给调用进程发送一个SIGALRM信号， 告诉进程时间到了。

core dump:程序异常退出时，操作系统会保存这个进程的运行信息，便于这个进程的事后调试(默认是关闭的)
ulimit -C 1024　开启core dump,并将core文件最大大小设置为1024kb
gdb ./loop. test　->　core- file core.pid(产生的core文件) - >常规的gdb调试步骤

信号在进程中的注册:让进程知道自己收到了这么一个信号

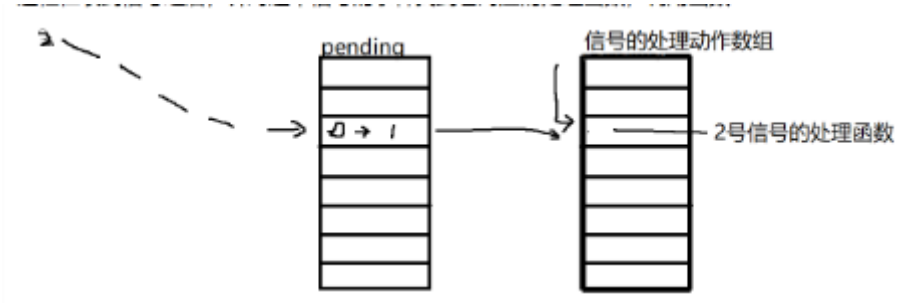
进程就是一个pcb, linux下是一 个task struct结构体;在pcb结构体中定义了一个信号的集合(位图) ;
若给进程发送一个信号， 则将这个信号对应位置的二进制位置1,进程通过查看位图判断是否有信号到来，进而去处理



pending信号集合:未决信号集合:
未决是一个状态，指的是信号产生了，但是还没有被处理的一一个区间状态
这个位图实际上是-个sigset t{unsigned long int_ _val[SIGSET. NWORDS]}结构体;这个数组被用于实现位图
位图这个信号集合，只能用于标记进程是否收到了这个信号,无法确定这个信号收到了多少次
因此在内核中其实还有一个链表 sigqueue{....siginfo_t...}
信号的注册，就是组织一个信号的信息， 添加到信号链表中，并且将信号pending位图进行置位
非可靠信号的注册:在注册信号时，判断当前信号是否已经注册过(位图是否已经为1) ;若没有注册，则添加节点，修改位图;反之，若已经注册过了，则什么也不干(这种信号在链表中就永远顶多只有一个节点,后来到达的信号就会被丢弃 _事件丢失)
可靠信号的注册:在注册信号时，每次针对新到来的信号都会创建一个节点添加到链表中，并且位图置1; (链表中有可能有多 个相同节点)

信号在进程中的注销:抹除信号存在痕迹
非可靠信号:删除节点，位图置0 (因为非可靠信号只会注册一次，顶多只有一个节点)
可靠信号:删除节点，判断是否还有相同节点，若没有，则位图置0;表示没有这个信号了，反之则位图依然为1;表示还有信号待处理

信号的处理:信号的递达
进程处理一个事件，事件可以理解为就是一一个功能，在程序中-一个功能的实现单位就是-一个函数进程在收到信号之后，针对这个信号的事件找到它对应的处理函数，调用函数



在操作系统中，每个信号都有其已经定义好的默认处理动作---信号的默认处理方式

信号的处理方式:

默认处理---系统中已经定义好的默认函数

忽略处理--处理的动作就是什么都不做(依然能够信号注册，只是处理动作中什么也不做而已)

自定义处理-- -用户自己定义信号回调函数，然后使用这个函数的地址替换原有信号动作数组中的函数地址也就是说，替换了信号处理动作中的回调函数

信号处理方式的修改:

sighandler_t signal(int signum, sighandler_t handler); //修改一个信号的处理动作中的回调函数

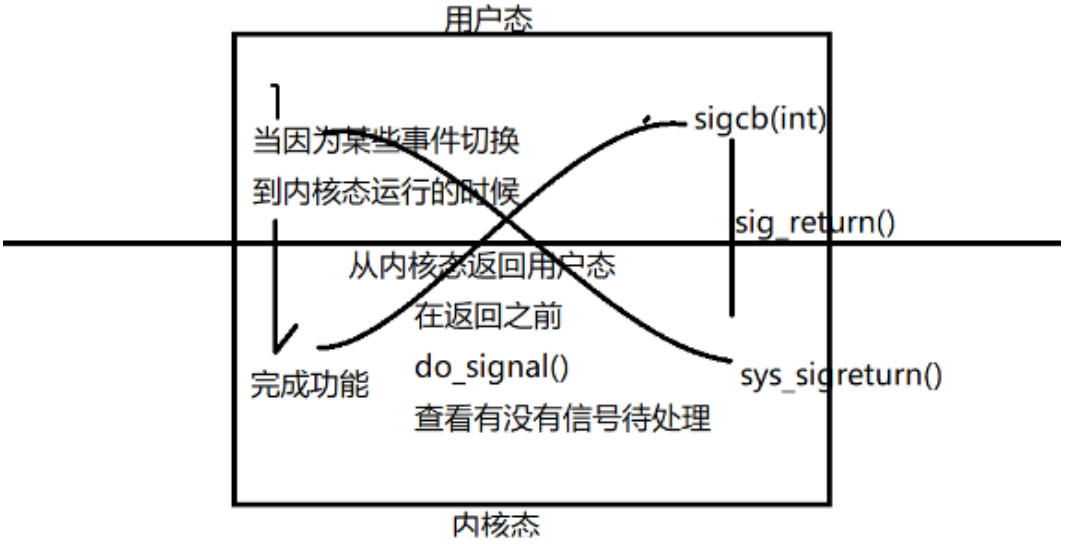
signum:信号值

handler:函数指针sighandler_t--函数指针类型

typedef void (*sighandler_t) (int signo)

SIG_ IGN--信号进行忽略处理　　SIG_ DFL---信号的默认处理

signo:当信号到来时，操作系统向回调函数中传入的信号值---告诉用户本次调用这个函数，是哪一个信号触发的自定义处理方式的信号捕捉流程:



信号的处理，是当前进程运行从内核态切换用户态之前进行处理;

问题:程序如何从用户态切换到内核态:中断，异常，系统调用接口

A:程序数据运算中5/0　B fwrite　C write　D中断

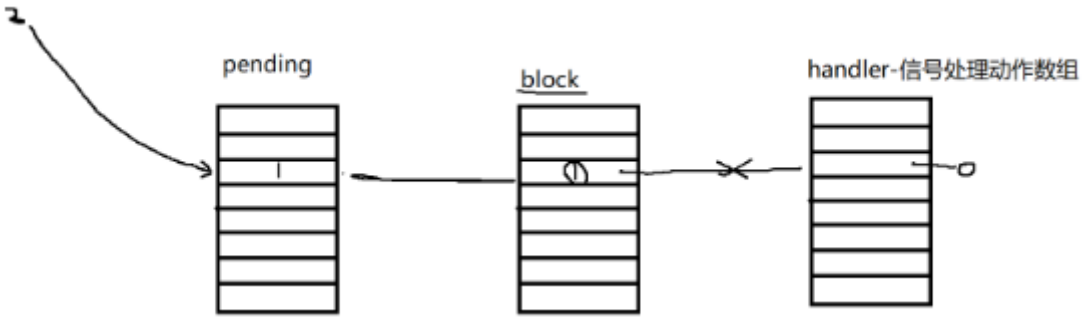
模拟一下异常带来的信号:

稍微调研一下:为什么内存访问错误只有一次， 但是段错误的信号却不断的在给进程发送

上节课讲到的管道特性:

所有读端关闭后，写端继续write会触发异常---这里的异常就是操作系统检测到管道没人读了，因此继续写入没有意义，因此操作系统给进程发送一个SIGPIPE信号 ,进程收到这个信号知道管道崩坏，退出进程

信号的阻塞:在进程中标记，哪些信号注册之后，暂时不去处理，直到信号解除阻塞
阻塞一个信号被递达
进程pcb中有一个信号block集合----用户可以在这个集合中标记哪些信号将被阻塞



如何阻塞一个信号: 将信号添加到pcb的block集合中，则表示文个信号将会被阻塞;到来则暂时吋不被处理
代码中如何阻塞一个信号:
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
how:即将要对pcb中block集合所作的操作
SIG_BLOCK 将第二个参数set集合中信号添加到block集合中/将set集合中信号加入阻塞
block = block I set
SIG_UNBLOCK 将第二个参数set集合中信号从block集合中移除/将set集合中信号解除阻
塞block = block &(~set); block = 00010001 set 00010000 -> 11101111
SIG_SETMASK 使用第二个参数set集合中信号替换block集合中的数据/ block = set
set:信号集合(位图)
oldset:每当block集合要友生改变的吋候，都会将原block集合中的数据拷贝到oldset中返回给用户， 便于后期还原

0.定义几个信号的处理方式
signal(SIGINT, sigcb); signal(SIGQUIT, sigcb);
1.定义一个集合---用户向其中添加信号，即将要阻塞这些信号 sigset tset; <-- SIGINT SIGQUIT

2.阻塞上边定义的信号集合中的信号
sigprocmask(SIG_BLOCK, &set, NULL);
3.向进程发送这些信号，看看是否能够收到这些信号，是否处理了这些信号
getchar()---等待用户一个回车，然后流程才会继续向下
4.解除阻塞，然后双察解除阻塞后信号会不会被处理
sigprocmask(SIG_UNBLOCK, &set, NULL);

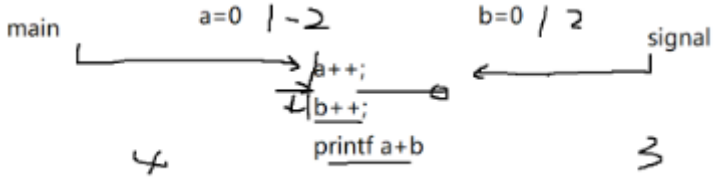
1.信号的阻塞与解除阻塞过程
2.可靠信号的注册与非可靠信号的注册
3.在所有信号中有两个信号不可被阻塞，不可被忽略，不可被自定义修改--- 9-SIGKILL 19-SIGSTOP

信号阻塞是在干什么
如何进行信号阻塞
代码操作

阻塞中的注意事项:
可靠/非可靠
哪些信号不会被阻塞

关键字: volatile ---防止编译器过度优化，保持内存可见性- --每次访问数据都需要从内存中重新获取
对一个变量的数据进行操作的时候: 1. 将变量数据从内存加载到cpu寄存器
若编译程序的时候，进行了代码优化，则会对使用度极高的数据，直接加载到寄存器，以后访问的时候直接从寄存器获取，提高程序性能，
这时候若我们修改了变量的值，在内存中数据已经改变，但是cpu这时候并不重新到内存中获取这个数据

函数的可重入于不可重入:



a++和b++的过程并非一次完成　　这个操作并非是原子操作

原子操作: 一件事情要么没干，要么就一次性完成，中间不会被打断

竞态条件:多个执行流同时竞争执行

在竞态条件下,一个函数的运行如果不会出现数据二义/逻辑混乱，则这个函数是一个可重入函树

在竞态条件下，一个函数的运行有哦可能会出现数据二义/逻辑， 则这个函数是一个不可重入

函数 (一旦重入就会出错)

函数的重入，在多个执行流中，同时进入一个函数执行功能

函数的可重入与不可重入关键点:这个函数重是否对全局数据进行了不受保护的操作(非原子操作)

mallo/free,不可重入函数

函数的重入

不可重入

可重入

判断的基准点

当我用使用一个别人的接口的时候,或者自己设计接口的时候,就需要考虑这种函数的是否可重

入情况

SIGCHLD信号:

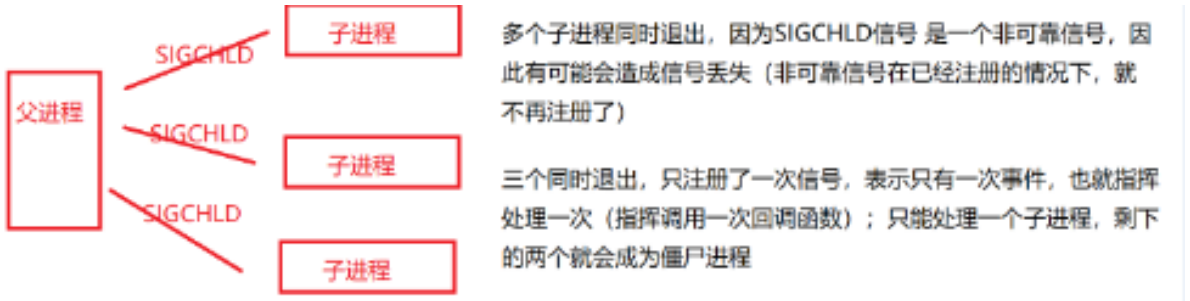
子进程先于父进程退出，操作系统会通知父进程，但是父进程若没有关注子进程的退出状态，则子进程成为僵尸进程

通知:操作系统就是通过SIGCHLD信号通知父进程的，

但是: SIGCHLD信号, 默认的处理动作就是一个忽略处理，导致父进程无法及时的或者，因此只能使用进程等待一直阻塞等待才可以若父进程知道什么时候信号到来了，然后再去调用wait/waitpid接口,则可以不需要过多等待

因此修改SIGCHLD信号的处理方式，在回调函数中调用waitpid接口; ---当子进程退出时，会向父进程发送SIGCHLD信号，收到信号,操作进程就会自动的去回调信号处理函数，调用其中的wait/waitpid接口实现回收资源

但是我们说了1~31号信号都是非可靠信号



多个子进程同时退出，因为SIGCHLD信号是一个非可靠信号，因此有可能会造成信号丢失(非可靠信号在已经注册的情况下，就不再注册了) .
三个同时退出，只注册了一次信号，表示只有一次事件,也就指挥处理一次(指挥调用一次回调函数) ;只能处理一个子进程， 剩下的两个就会成为僵尸进程

因此最好就在一次回调函数中，能够将所有的僵尸进程都处理掉----循环调用waitpid接口，直到没有子进程才退出信号回调函数
waitpid(pid_ t pid, int *status, int option)-返回值:出错小于0;没有子进程退出等于0;有子进程退出返回处理的子进程pid>0
但是默认情况下，waitpid是阻塞的， 如果没有子进程退出了，则循环会卡在waitpid这块，信号回调函数无法返回;程序主控流程停滞
因此应该将waitpid设置为非阻塞; option= WNOHANG

复习

进程信号:
信号处理:
信号的处理方式最终也就是进程收到信号后执行信号的回调函数完成功能
方式:默认处理/忽略处理/自定义处理
修改信号处理方式:sighandler_t signal(int signum, sighandler_t hander);
sighandler_t: typedef void (*sighandler_t)(int);

自定义处理方式的信号捕捉流程
1.程序运行在用户态主控流程，在中断/异常/系统调用的情况下，进程切换到内核态运行
2.完成内核功能之后，在即将返回用户态之前调用do_ signal接口去处理信号
3.其中默认处理/忽略处理都是在内核中完成，但是自定义接口是用户自己定义的函数运行的用户态

4.因此进程从内核态切换到用户态运行的是信号自定义回调函数，去处理信号事件
5.在信号处理函数运行完毕后，调用sigreturn返回内核态
6.当没有信号待处理，则调用sys_sigreturn返回用户态主控流程从之前运行的地方开始继续运行

信号的阻塞:阻止信号被处理(在进程中标记信号，这些信号到来注册之后，暂时不处理，直到解除阻塞)
如何阻塞信号:进程pcb中有一-个block信号集合;将集合种信号相应为置1,就表示要阻塞这个信号了int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
how: SIG_BLOCK/SIG _UNBLOCK/SIG_SETMASK
在所有信号中，有两个信号无法被阻塞，无法被自定义,无法被忽
路: 9/19 SIGKILL/SIGSTOP
什么情况下一个进程无法被杀死:僵尸进程，处于停止状态的进程，信号被阻塞/忽略处理的进程

volatile关键字:保持内存可见性，防止编译器过度优化---修饰一个变量，让cpu每次访问数据的时候都重新从内存中获取
2014-- gcc -02 -弱鸡不配使用代码优化

函数的可重入与不可重入:
函数的重入:多个执行流同时执行进入同一个函数
不可重入函数:函数重入之后有可能造成数据二义或者逻辑混乱
可重入函数:函数重入之后不会造成数据二义或者逻辑混乱
函数是否可重入的关键点: 一个函数重是否对一个全局数据进行了不受保护的操作

SIGCHLD:子进程退出时，操作系统发送给父进程的通知信号
默认忽略处理:因此子进程在不等待的情况下才会成为僵尸进程，因为父进程没有关注到这个信号
因此父进程必须一直等待，才能避免产生僵尸进程
修改信号的处理方式之后:在信号到来时，进程自动回调信号回调函数，在回调函数重调用进程等待接口处理子进程的退出是一个非可靠信号:若同时退出多个子进程，有可能会造成事件丢失;
因此回调函数中循环进行进程等待，直到没有子进程退出
sigcb(int signo) { while(waitpid(- 1, NULL WNOHANG)> 0) ;}