## 进程与线程

gcc -v abc.c 显示编译的信息

运行时的加载器：加载器是操作系统中的程序，当我们去执行一个程序时（./a.out ,譬如代码中用exec族函数来运行）加载器负责将这个程序加载到内存中去执行这个程序。

程序如何结束

正常运行结束、异常退出、接受到信号终止

atexit（） 注册终结处理函数

atexit() 函数可以注册多个，先注册的后执行，和栈一样

Return和exit一样，都是会执行atexit（）函数

\_exit()终止进程时，并不执行atexit()注册的进程终止函数

#inlcude<stdio.h>

#include<stdlib.h>

Void func1(void){

Printf("func1\n");

}

Int main(void)

{

Printf("hello world.\n");

atexit(func2);

atexit(func1);

printf("my name is zhaweiwie");

exit(0);

}

### 进程环境

环境变量

export命令查看环境变量

进程环境表：每一个进程都有一份所有环境变量构成的表格，也就是说我们当前进程中可以直接使用这些环境变量

程序中使用envison全局变量使用环境变量

#include<stdio.h>

Int main (){

Extern char \*\*envison; //申明就能使用

Int i =0;

While (NULL !=envison[i]){

Printf("%s\n",envison[i]);

i++;

}

Return 0;

}

我们写的程序中可以无条件直接使用环境变量，所以一旦用到了环境变量，那么程序就和操作系统环境有关了。

获取指定环境变量可以使用getenv()函数 man getenv

getpid

getppid

getuid 获取用户id

geteuid 获取有效用户id

getgid 获取组id

getegid 获取有效组id

#include <stdio.h>

#include <sys/types.h>

#include<unistd.h>

int main(){

pid\_t p1=-1,p2=-1;

printf("hello.\n");

p1=getpid();

printf("pid =%d.\n",p1);

p2=getppid();

printf("ppid =%d.\n",p2);

return 0;

}

gcc 文件名.c

./ 文件名.out

fork 创建子进程

1. 进程的分裂生长模式，如果操作系统需要一个新进程来运行一个程序，那么操作系统会用一个现有的进程来复制生成一个新进程，老进程叫父进程，复制生成的进程叫子进程

#include<stdio.h>

#include<sys/types.h>

#include<unistd.h>

int main(void){

pid\_t p1=-1;

p1=fork();

if(p1==0){

//这里一定是子进程

}

if(p1>0){

//这里是父进程

}

if(p1<0){

//这里一定是fork出错了

}

//

printf("Hello World.pid=%d.\n",getpid());

}

fork函数调用一次会返回两次，返回值等于0的就是我们的子进程，返回值大于零的就是父进程。

典型的使用fork（）的方法，使用fork后然后用if判断返回值，并且返回值大于0的时候就是父进程。

fork的返回值在子进程中等于0，在父进程中返回值是子进程号。

父进程执行完成后，自身结束前会把自己的子进程移交给自己的父进程，如init进程

关于子进程

子进程有自己独立的PCB

子进程被内核同等调度

父子进程对文件的操作

子进程继承父进程中打开的文件

上下文：父进程先open（）打开一个文件得到fd，然后fork（）创建子进程。之后在父进程中各自write向fd中写入内容。

测试结论是：接续写。实际上本质原因是父子进程之间的fd对应的文件指针是彼此关联着的。

实际测试的时候会看到只有一个，有点像分别写。但实际不是，原因是一个进程写完后关闭了文件，导致另一个进程写不进去。

#include<stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

#include <stdlib.h>

int main(){

//首先打开一个文件

int fd=-1;

fd=open("1.txt",O\_PDWR |O\_TRUNC);

if(fd<0){

perror("open");

return -1;

}

//fork创建子进程

pid=fork();

if (pid>0){

//父进程中

printf("parent.\n");

write(fd,"hello",5);

sleep(1); //防止父进程或者子进程运行完毕后把文件关闭，导致另一个进程写不进去。

}

else if(pid==0){

//子进程

printf("child.\n")

prinf(fd,"world",5);

sleep (1);

}

else{

perror("fork");

exit(-1);

}

}

父子进程各自独立打开同一文件实现共享。

父进程open打开1.txt然后写入，子进程打开1.txt然后写入，结论是：分别写，fork（）之后才打开的文件。此时PCB已经独立了，文件表独立。

#include<stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

#include <stdlib.h>

int main(){

//首先打开一个文件

int fd=-1;

//fork创建子进程

pid=fork();

if (pid>0){

//父进程中

printf("parent.\n");

fd=open("1.txt",O\_PDWR);

if(fd<0){

perror("open");

return -1;

}

write(fd,"hello",5);

sleep(1); //防止父进程或者子进程运行完毕后把文件关闭，导致另一个进程写不进去。

}

else if(pid==0){

//子进程

printf("child.\n")

fd=open("1.txt",O\_PDWR);

if(fd<0){

perror("open");

return -1;

}

write(fd,"world",5);

sleep (1);

}

else{

perror("fork");

exit(-1);

}

}

open时候使用O\_APPEND标志可以将父子进程对同一文件的操作关联起来，达到接续写的作用

#include<stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

#include <stdlib.h>

int main(){

//首先打开一个文件

int fd=-1;

//fork创建子进程

pid=fork();

if (pid>0){

//父进程中

printf("parent.\n");

fd=open("1.txt",O\_PDWR |O\_APPEND);

if(fd<0){

perror("open");

return -1;

}

write(fd,"hello",5);

sleep(1); //防止父进程或者子进程运行完毕后把文件关闭，导致另一个进程写不进去。

}

else if(pid==0){

//子进程

printf("child.\n")

fd=open("1.txt",O\_PDWR|O\_APPEND);

if(fd<0){

perror("open");

return -1;

}

write(fd,"world",5);

sleep (1);

}

else{

perror("fork");

exit(-1);

}

}

### 进程的诞生和消亡

进程的诞生

1. 进程0和进程1
2. fork()
3. vfork()

进程的消亡

1. 正常终止和异常终止
2. 进程在运行时需要消耗系统资源（内存、IO）进程终止时候理应完全释放这些资源。否则会造成资源丢失。
3. Linux系统设计时规定：每一个进程退出时，操作系统会自动回收这个进程的涉及的所有资源。（譬如malloc申请的内容没有free时，当前进程结束时，这个内存会被释放。譬如open打开的文件没有close，在程序终止时也会被关闭。）但是操作系统知识回收了这个进程工作时候消耗的内存和IO，而并没有回收这个进程本身占用的内存（8KB，主要是task\_struct（进程描述信息）和栈的内存）
4. 因为进程本身的8KB内存操作系统不能回收，需要别人来辅助回收，因此我们每个进程都需要一个帮助他收尸的人，这个人就是这个进程的父进程。

僵尸进程

1、子进程先于父进程结束，子进程结束后父进程此时候不应定能立即就能帮子进程收尸，在这一段（子进程已经结束且父进程尚未帮其收尸）子进程就被称为僵尸进程

2、子进程除了task\_struct和栈外其余内存空间皆以清理

3、父进程也可以显示调用wait或waitpid以回收子进程的剩余待回收内存资源并且获取子进程退出状态。

4、父进程也可以不适用wait和waitpid回收子进程，此时父进程结束时一样会回收子进程剩余待回收的内存资源。（这样设计是为了防止父进程忘了显示调用wait或者waitpid来回收子进程而造成内存泄露）

孤儿进程

1、父进程先于子进程结束。子进程成为一个孤儿进程。

2、Linux系统规定，所有的孤儿进程自动都成为一个特殊的进程（进程1，也就是init进程）的子进程。

父进程调用wait回收子进程

wait的工作原理

1. 子进程结束时候，系统向其父进程发送SIGCHILD信号
2. 父进程调用wait函数时阻塞
3. 父进程被SIGCHILD信号唤醒后然后去回收僵尸子进程
4. 父子进程之间是异步的，SIGCHILD信号机制就是为了解决父子进程之间的异步通信问题， 然父进程可以及时的去回收僵尸子进程

5、若父进程没有任何子进程则wait返回错误

wait的参数status status用来返回子进程结束时候的状态，父进程通过wait得到的status后就可以知道子进程的一些结束信息状态。

wait的返回值pid\_t，这个返回值就是本次wait回收的子进程PID，当前进程有可能有多个子进程，wait函数阻塞知道其中一个子进程结束wait就会返回。wait的返回值可以用来判断到底是哪一个子进程本次被回收了

#include<stdio.h>

#include<unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main(void){

pid\_t pid =-1;

pid\_t ret=-1;

pid=fork();

if (pid>0){

//父进程

//sleep(1);wait 本身就会阻塞

printf("parent.\n");

ret= wait(&status);

printf("子进程已经被回收，子进程pid=%d.\n",ret);

printf("子进程是否正常退出%d.\n"，WIFEXITED(status));

printf("子进程是否被信号终止退出%d\n",WIFSIGNALED(status));

printf("子进程终止的返回值%d.\n",WEXITSTATUS(status));

}

if(pid==0){

//子进程

printf("child.\n");

}

if(pid<0){

perror("fork");

return -1;

}

}

WIFEXITED 宏用来判断子进程是否正常退出。

WIFSIGNALED 宏用来判断子进程是否被信号中断（异常终止）和上面的值正好相反。

WEXITSTATUS 宏用来返回子程序正常终止的时候用户自定义的返回值

waitpid介绍

wait和waitpid函数基本功能一样，都是用来回收子进程的

waitpid可以回收指定PID的子进程，wait不行

waitpid可以阻塞式或者非阻塞式两种工作模式

ret=waitpid(pid,&status,0);

-1表示不等待某个特定的子进程而是回收任意一个子进程，0表示默认的方式（阻塞式）来进行等待，返回值ret式本次回收的子进程PID

ret=waitpid(pid,&status,0);

等待回收PID为pid的这个子进程，如果当前进程没有一个PID为pid的进程，则返回至为-1；如果成功回收了子进程，则返回值为该子进程的PID；

ret=waitpid(pid,&status, WNOHANG)；

直接去会后PID为pid的这个子进程，如果子进程没有结束，则直接返回-1；成功回收则返回子进程的PID

#include<stdio.h>

#include<sys/types.h>

#include<sys/wait.h>

int main(void){

pid\_t pid=-1;

pid\_t ret =-1;

int status=-1;

p1=fork();

if (pid>0){

//父进程

printf("parent.\n");

//ret=waitpid(pid,&status,0);

ret=waitpid(pid,&status, WNOHANG);

printf("子进程已经被回收，子进程pid=%d.\n",ret);

printf("子进程是否正常退出%d.\n"，WIFEXITED(status));

printf("子进程是否被信号终止退出%d\n",WIFSIGNALED(status));

printf("子进程终止的返回值%d.\n",WEXITSTATUS(status));

}

else if(pid==0){

//子进程

printf("child.\n");

sleep(1);用于让WNOHANG方式回收不了直接返回；

}

else(pid<0){

//

perror("fork");

}

}

竞态全程是：竞争状态，多进程环境下，多个进程同时抢占资源（内存、CPU、文件IO）

#include<stdio.h>

#include<sys/types.h>

#include<sys/wait.h>

int main(void){

pid\_t pid=-1;

pid\_t ret =-1;

int status=-1;

p1=fork();

if (pid>0){

//父进程

printf("parent.\n");

//ret=waitpid(pid,&status,0);

ret=waitpid(pid,&status, WNOHANG);

}

else if(pid==0){

//子进程

printf("child.\n");

unsigned int count=0;

while(1){

printf("cnt=%d.\n",cnt)

cnt++;

sleep(1);

}

}

else(pid<0){

//

perror("fork");

}

}

fork创建子进程是为子进程执行新的程序（fork创建了子进程后，子进程和父进程可以同时被OS调度执行，因此子进程可以单独的执行一个程序，这个程序在宏观上和父进程同时进行）

可以直接在if中写入子程序的代码,这样可以，但不够灵活，因为我们只能把子进程程序的源代码贴过来执行（必须知道源代码，而且源代码太长了也不好控制）譬如我们希望子进程

执行ls -la 命令就不行了（没有源代码，只有编译好的可执行程序）

使用exec()族函数运行新的可执行程序（exec族函数可以直接把一个编译好的可执行程序直接加载运行）

我们有了exec族函数后，典型的父子进程程序是这样的，子进程需要运行的程序被单独编写，单独编译成一个可执行程序（叫hello），（项目是一个多进程的项目）主程序中为父进程，fork创建了子进程后在子进程中用exec来执行hello，达到父子进程分别作不同程序同时（宏观上）运行的效果

execl和execv execl是把参数列表（本质上是多个字符串，必须是NULL结尾）依次排列而成（l就是list的缩写），execv是把参数列表事先放入一个字符串数组，再把这个字符串数组传给execv函数

execlp 和execvp ，加了个p，较上面两个执行程序的区别是，上面两个必须要传递文件的全路径，而这两个传递的可以file（也可以是path，兼容了file，首先找file，找到就执行，找不到则会去环境变量PATH所指定的目录下继续找，找到执行，找不到报错）

execle和execvpe 加了个e，e代表环境变量environment，和基本版本exec的区别是，执行可执行程序时候会多传一个环境变量

#include<stdio.h>

#include<sys/types.h>

#include<sys/wait.h>  
#include<unistd.h>

int main(void){

pid\_t pid=-1;

pid\_t ret =-1;

int status=-1;

p1=fork();

if (pid>0){

//父进程

printf("parent.\n");

//ret=waitpid(pid,&status,0);

ret=waitpid(pid,&status, WNOHANG);

}

else if(pid==0){

//子进程

printf("child.\n");

//execl("/bin/ls", "ls", "-a", "-l" , NULL);

char \* const arg[]={"ls","-a","-l"};

execv("/bin/ls",arg);

}

else(pid<0){

//

perror("fork");

}

}

#include<stdio.h>

int main(void){

printf("Hello World");

return 0;

}

编译运行一下

gcc hello.c -o hello

上面程序改成execl("./hello",NULL);

#include<stdio.h>

int main(int argc,char\*\*argv){

printf("argc=%d.\n" argc);

int i =0;

while(NULL!=argv[i]){

printf("argv[%d]=%s",i,argv[i]);

i++;

}

return 0;

}

gcc hello.c -o hello

上面改成execl("./hello","aaa","bbb",NULL);

execlp("ls","-a","-l",NULL);自动查询环境变量所在路径

env 就是环境变量字符串数组，平时省略，没有传递

int main(int argc,char\*\*argv,char \*\*env)

第三个参数是环境变量数组，如果用户没有传递，会从操作系统直接继承环境变量。如果使用execle或者execvpe传递一个字符串数组，则程序中的实际环境变量是我们传递的那一份，覆盖从父进程继承的环境变量。

#include<stdio.h>

#include<sys/types.h>

#include<sys/wait.h>  
#include<unistd.h>

int main(void){

pid\_t pid=-1;

pid\_t ret =-1;

int status=-1;

p1=fork();

if (pid>0){

//父进程

printf("parent.\n");

//ret=waitpid(pid,&status,0);

ret=waitpid(pid,&status, WNOHANG);

}

else if(pid==0){

//子进程

printf("child.\n");

char\* const envp[]={"AA=aaa","BB=abcd","CC=ccc",NULL};

execle("./hello",NULL,envp);

}

else(pid<0){

//

perror("fork");

}

}

#include<stdio.h>

int main(int argc,char\*\*argv){

printf("argc=%d.\n" argc);

int i =0;

while(NULL!=argv[i]){

printf("argv[%d]=%s",i,argv[i]);

i++;

}

int j=0;

while(NULL!=argv[j]){

printf("argv[%d]=%s",j,arg[j]);

}

return 0;

}

system函数简介

system函数=fork+exec

system 是原子操作，一旦开始就不被打断的执行完毕。而fork+exec可以被打断

进程关系

无关系

父子进程关系

进程组：由若干进程构成一个进程组

会话：会话就是进程组的组。

守护进程

ps -ajx 偏向于显示各种有关的ID号

ps -aux 偏向于显示进程占用的各种资源

向进程发送信号指令 kill

kill -信号编号 进程ID 向进程发送一个信号

kill -9 xxx,将向xxx这个进程发送信号9，也就是要结束进程

守护进程

daemon 守护进程，简称为d(进程名后面带d基本就是守护进程)

长期运行（一般从开机一直运行到关机）

与控制台脱离（普通进程都和运行该进程的控制台相绑定，表现为如果中端被强制关闭则这个终端中运行的所有进程都会被关闭，背后的问题还在于会话）

常见守护进程

syslogd 系统日志守护进程

cron,cron 进程用来实现操作系统的时间管理，linux中实现定时执行程序的功能就要用到 cron。

编写一个守护进程

任何一个进程都可以将自己实现为守护进程

create damon 函数要素

子进程要等待父进程退出

子进程使用setid创建新的会话期，脱离控制台

调用chdir将当前工作目录设置/

unmask设置为0以取消任何文件权限屏蔽

关闭所有文件描述符

将0、1、2定位到/dev/null

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int main(void){

create\_daemon();

while(1){

printf("I am running .\n");

sleep(1);

}

}

//函数的作用就是把调用该函数的进程变成守护进程

void create\_damon(){

pid\_t pid=0;

pid =fork();

if (pid<0){

perror("fork");

exit(-1);

}

if(pid>0){

exit(0); //父进程直接退出

}

//执行到这里一定是子进程

setid将当前进程设置为新的一个会话期session，目前就是让当前进程脱离控制台

pid=setsid();

if(pid<0){

perror("setsid");

exit(-1);

}

//将当前工作目录设置为/

chdir("/"); //用来设置当前进程的工作目录

umask设置为0确保将来文件有最大的文件操作权限

umask(0);

//关闭所有的文件描述符

//先要获取当前系统中所允许打开的最大的文件描述符的数目

cnt=sysconf(\_SC\_OPEN\_MAX);

int i=0;

for(i=0;i<cnt;i++){

close(i);

}

//把所有输入输出错误输出丢入垃圾桶，控制台不再输出

open("/dev/null",O\_RDWR); 打开标准输入

open("/dev/null",O\_RDWR); 打开标准输出

open("/dev/null",O\_RDWR); 打开标准错误输出

}

使用syslog记录调试信息

openlog、syslog、closelog

一般log信息都在操作系统/var/log/messages这个文件中存着，但是ubuntu是在

/var/log/syslog文件中

#include<stdio.h>

#include<syslog.h>

#include <sys/types.h>

#include <unistd.h>

int main(){

printf("my pid =%d.\n",getpid());

openlog("a.out",LOG\_PID|LOG\_CONS,LOG\_USER);

syslog(LOG\_INFO,"this is my log");

syslog(LOG\_INFO,"this is another log info %d.\n" ,23);//格式类似printf

closelog();

}

操作系统中有一个守护进程syslogd（开机运行，关机才结束），这个守护进程syslogd负责进行日志文件的写入和维护

syslogd是独立于我们任意一个进程而运行的，我们当前进程和syslogd进程本来没有任何关系，但是我们当前进程可以通过调用openlog打开一个和syslogd相连接的通道，然后通过syslog向syslogd发消息。然后由syslogd来将其写入到日志文件中。

syslogd其实就是一个日志文件系统服务器进程，提供日志服务。任何需呀写日志的进程都可以通过openlog，syslog，closelog这三个函数来利用syslogd提供的日志服务。这就是操作系统服务式的设计。

让程序不能被多次运行

问题

因为守护进程是长时间运行而不退出，一次./a.out执行一次就会有一个进程，执行多次就会有多个守护进程。

这样并不是我们想要的。守护进程一般都是服务器，服务器程序只需要运行一次就可以了，多次重复运行并没有意义，甚至会带来错误。

因此希望我们的程序具有一个单例运行的功能，一次就是说当我们./a.out去运行程序时候，如果之前没有运行就运行它，如果已经有了，则本次运行直接退出（提示程序已经在运行）。

实现方法：

常用方法：用一个文件的存在与否做标志，具体的做法就是程序在执行之初去判断一个特定的文件是否存在，如果存在说明进程在运行，如果不存在说明进程不在运行，然后运行程序时候去创建这个文件。文件名一定要特殊。当程序运行结束的时候一定要删除文件。

#include<stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <errno.h>

#define FILE "/var/aston\_test\_single"

void delete\_file(void)

int main(void){

//程序执行之初先去判断文件是否存在

open(FILE,O\_RDWR |O\_TRUNC|O\_CREAT|O\_EXCL,0664);

if(fd<0){

if(errno==EEXIST){

printf("进程已经存在，请不要重复执行\n");

return -1;

}

}

atexit(delete\_file); //程序在终止的时候执行该函数注册的函数。

printf("打开成功\n");

int i=0;

while(i<10){

printf("I am running ...\n");

sleep(1);

i++;

}

return 0;

}

void delete\_file(void){

remove(FILE);

}

进程间通信（IPC）指的是两个任意进程之间的通信。

多种进程间通信机制

无名管道和有名管道

SystemV IPC：信号量、消息队列、共享内存

Socket域套接字

信号

管道（无名管道）

管道通信的原理：内核维护的一块内存，有读端和写端，管道是单向通信的

父进程创建管理fork子进程，子进程继承父进程的管道fd

管道通信的限制：只能在父子进程之间通信、半双工

管道通信函数：pipe、write、read、close

有名管道（fifo）

有名管道的原理：实质是内核维护的一块内存，表现形式为一个有名字的文件。

固定一个文件名，2个进程分别使用mkfifo创建fifo文件，然后分别open打开获取到fd，然后一个读一个写。

管道通信限制：半双工

管道通信函数：mkfifo、open、write、read、close

消息队列：本质上是一个队列FIFO，（内核维护的一个FIFO）

工作时候A和B两个进程通信，A向队列放入消息，B从队列中读出消息。

信号量：实质是一个计数器（用来计数的变量，可以理解为int a）实现互斥和同步

共享内存：大片内存直接映射 类似于LCD显示时的显存方法。