1. fork函数
2. 概述

fork函数在一个进程的基础上创建一个新的子进程。

1. 函数原型

pid\_t fork(void);

1. 函数返回值

创建成功时，返回两次。父进程返回子进程的id，子进程返回0.。

1. getpid

获得当前进程的id（pid）。

pid\_t getpid(void);

1. getppid

获得当前进程的父进程的id（ppid）。

pid\_t getppid(void);

1. 使用示例

#include<stdio.h>

#include<unisted.h>

#include<stdlib.h>

int main(){

printf(“Begin\n”);

pid\_t pid = fork();

if(pid < 0){

perror(“fork err”);

exit(1);

}

//子进程

if(pid == 0){

printf(“I am a child, pid=%d, ppid=%d\n”, getpid(), getppid());

}

else if(pid > 0){

printf(“childpid=%d, self=%d, ppid=%d\n”, pid, getpid(), getppid());

sleep(1);

}

printf(“End … \n”);

return 0;

}

1. 进程操作
2. 查看进程信息

init进程是所有进程的祖先。

ps aux

ps ajx 可以追溯进程之间的父子关系、

1. 杀死进程

kill -l 查看信号相关的信息

给进程发送一个信号，杀死进程的信号为 SIGKILL 9号信号。

kill -SIGKILL pid

pid为进程的id

1. 创建多个子进程

上面已经讲解过了创建单个子进程，直接使用fork函数创建子进程即可。那么，在创建多个子进程时，如果直接使用for循环创建多个子进程就会出现子进程会接着循环创建自己的子进程，即如果循环5次，那么会创建32个子进程，而不是5个子进程。

要想解决这个问题，就要在确认是新创建的子进程处通过break离开循环。

#include<stdio.h>

#Include<unisted.h>

#include<stdlib.h>

int main(){

int n =5;

int i = 0;

pid\_t pid = 0;

for(I = 0; I < 5; ++i){

pid = fork();

if(pid == 0){

printf(“I am child, pid=%d, ppid=%d\n”, getpid(), getppid());

break;

}

else if(pid > 0){

printf(“fater, pid=%d, ppid=%d\n”, getpid(), getppid());

}

}

return 0;

}

1. exec函数族
2. execlp函数
3. 用途

借助PATH环境变量，加载一个进程。

1. 函数原型

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

1. 函数返回值

成功不返回，失败返回-1

1. 函数参数

file:要加载的程序的名称，该函数需要配合PATH环境变量使用，在环境变量的目录中搜索有无参数1，通常用于调用系统程序，如：ls, cp…

arg:参数列表，参数列表最后需要一个NULL作为结尾

1. execl函数
2. 用途

通过路径 + 程序名 来加载一个进程。

1. 函数原型

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

1. 函数返回值

成功无返回，失败返回-1

1. 函数参数

参数与execlp函数一样，只不过path要加上路径名。

如： 加载 ls 命令并带有 -l,-F 参数

execlp(“ls”, “ls”, “-l”, “-F”, NULL);

在PATH的目录下搜索ls 程序名称

execl(“/bin/ls”, “ls”, “-l”, “-F”, NULL);

在绝对路径中搜索ls 程序名称

1. 孤儿进程与僵尸进程
2. 孤儿进程

该进程的父进程被kill了，该进程被init进程给领养了。

1. 僵尸进程

子进程被kill了，父进程没有回收子进程的资源，则该子进程成为了僵尸进程。

那么僵尸进程如何被回收呢？通过kill父进程，init进程领养子进程，init进程负责回收子进程。

1. wait函数
2. 概述

wait函数一般用于回收子进程的资源，如果子进程没有被Kill，则wait函数一直阻塞等待着子进程的死亡，并且能够知晓子进程的死亡原因。

1. 作用
2. 阻塞等待
3. 回收子进程资源
4. 查看死亡原因
5. 函数原型

pid\_t wait(int \* status);

status： 传出参数

1. 函数返回值

成功返回 终止的子进程ID

失败返回 -1

1. 子进程死亡原因
2. 正常死亡 WIFEXITED

如果WIFEXITED为真，则使用WEXITSTATUS得到退出状态

1. 非正常死亡 WIFSIGNALED

如果WIFSIGNALED为真，则使用WTERMSIG得到信号

1. waitpid函数
2. 概述

回收子进程。

1. 函数原型

pid\_t waitpid(pid\_t pid, int \* status, int options);

pid

< -1 传递的是组id

-1 回收任意

0 回收和调用进程组id相同组内的子进程

>0 回收指定的pid

options

0和wait相同，也会阻塞

设置WNOHANG,如果当前没有子进程退出，会立刻返回

（三） 函数返回值

如果设置了WNOHANG,那么如果没有子进程退出，则返回0；如果有子进程退出，则返回退出的pid

失败返回-1（没有子进程）