

系列课程 —Linux系统编程

第四章

Linux进程管理

讲师：任继梅

课前提问

1. 什么是进程？
2. 进程的内存空间如何布局？
3. 如何在代码里从内存里获取一定的空间？
4. `main`函数参数含义是什么？

本章内容

- ✓ 4.1 Linux进程管理
- ✓ 4.2 进程环境
- ✓ 4.3 进程控制
- ✓ 4.4 进程关系
- ✓ 4.5 守护进程
- ✓ 4.6 信号

本章目标

✓ 掌握Linux进程管理



✓ 理解进程环境



✓ 掌握进程控制



✓ 理解进程关系



✓ 熟悉守护进程



✓ 熟悉信号管理



第一节

进程管理

进程管理

什么是进程？

- ❑ 程序（program）：存储在外存上，处于某个目录中的一个可执行文件
 - ☞ 由源代码编译连接后生成
 - ☞ 由exec系列系统调用函数读入内存后执行
- ❑ 进程（process）：程序在内存中执行的实例
 - ☞ 是操作系统task的一种
 - ☞ 操作系统为管理方便给每个进程一个唯一的数字标识符——进程ID
 - ☞ 是系统分配资源（代码段、数据区、栈、文件描述符等）的基本单元
 - ☞ 能够获得CPU时间片运行代码的执行单元

进程的基本元素

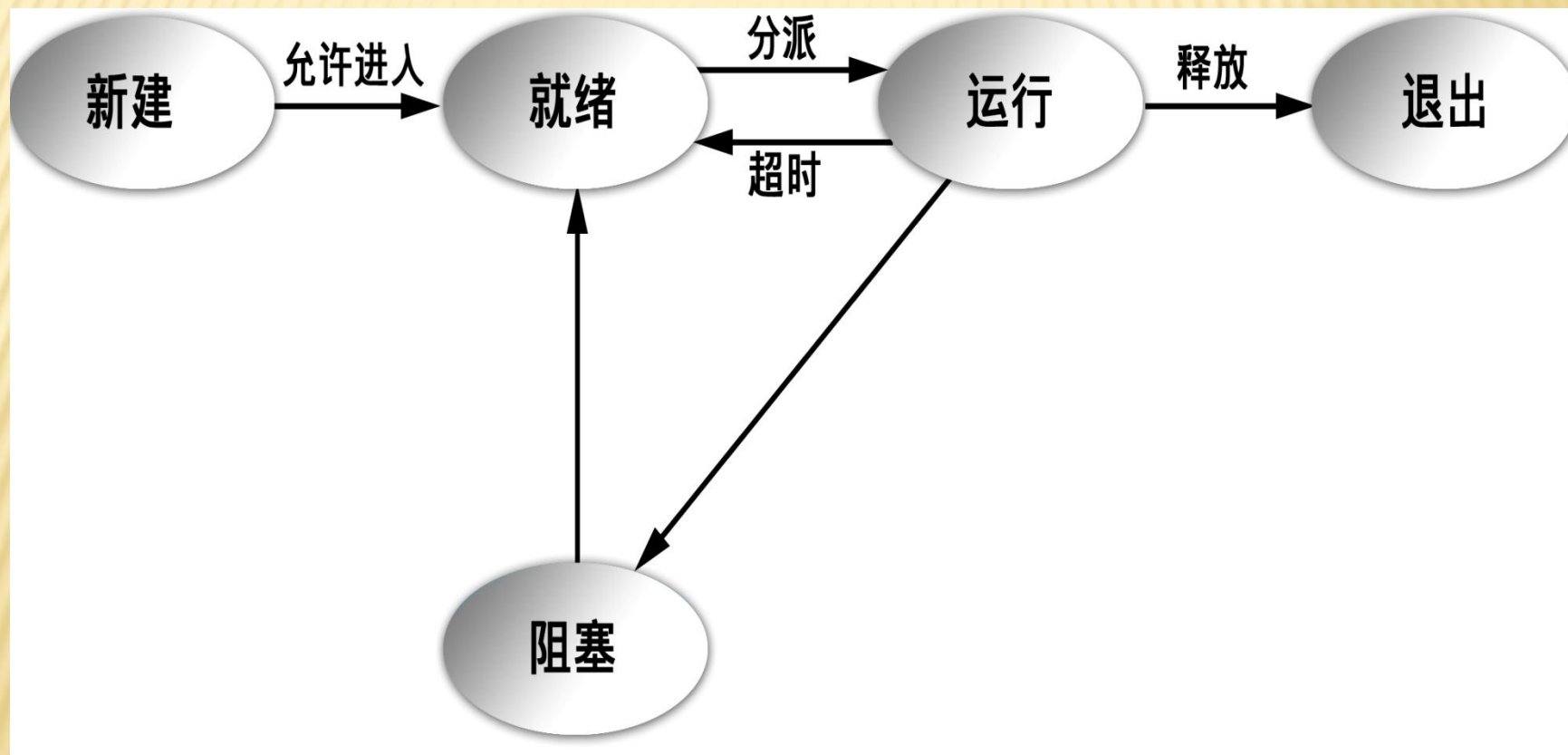
👤 OS如何描述一个进程实体?

- ❑ Linux内核代码实现里没有进程、线程的概念，只认task

👤 进程控制块(PCB): struct task_struct 主要成员:

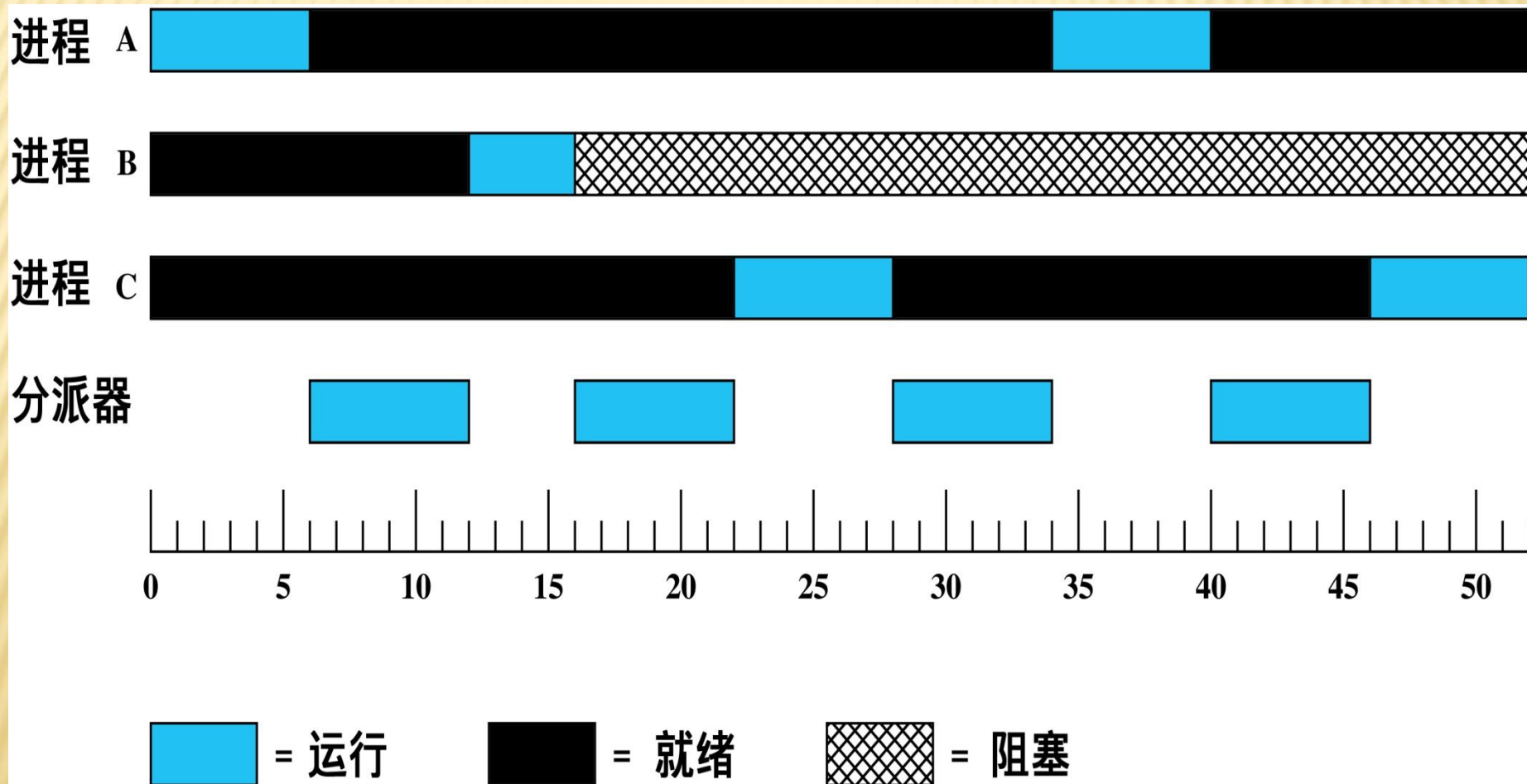
- ❑ 标志符
- ❑ 状态
- ❑ 优先级
- ❑ 程序计数器
- ❑ 内存指针
- ❑ 上下文数据(context)
- ❑ IO的状态信息
- ❑ 审计信息

进程状态



进程管理

例子：



进程管理

查看进程

- 命令 `ps -efl`
- 命令 `ps -ef -o pid,ppid,pgid,sid,command`

进程标示符 (pid)

- 32位正整数
- 创建进程时由OS内核统一分配
- 唯一标识一个进程
- 取值范围 (2~32767)

进程管理

● 所有进程不是同时创建

- ❑ 一个一个一次创建新的进程

● 父进程和子进程

- ❑ Linux第一个启动的应用进程是init
- ❑ 除init进程外，其它所有进程都是由我们称之为“父进程”的进程启动
- ❑ 被父进程启动的进程就叫子进程
- ❑ init进程的pid为1，作为后续进程的祖先进程，它的父进程id为零
- ❑ 进程调度器的pid为0，其实就是内核本身
- ❑ 进程树

进程管理

获取进程ID

```
#include <sys/types.h>
#include <unistd.h>
```

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

- ❏ getpid() 返回当前进程ID
- ❏ getppid() 返回父进程ID

进程管理

获取进程ID例子

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    printf("The proces ID is %d\n", getpid());
    printf("The parent process ID is %d\n", getppid());

    return 0;
}
```

第二节

进程控制

进程控制

👤 程序里运行命令：system()

```
#include <stdlib.h>
```

```
int system(const char *cmd);
```

- ❏ 功能：运行以cmd参数传递给它的程序，并等待该程序的完成
- ❏ 返回值： 正常返回执行命令的进程的终止状态
出错返回-1

进程控制

system()函数例子:

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("system() start...\n");
    system("ls -l");

    printf("system() end...\n");

    return 0;
}
```

进程控制

👤 替换进程：exec系列系统调用

```
#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);

int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

- ⊕ 功能：替换掉当前进程，即用一个全新的程序替换当前进程存储区，对当前进程做些清理工作后，重置PCB许多项。

进程控制

包含p字母: PATH

- ❏ execvp和execp
- ❏ 程序名作为参数，在PATH中搜索
- ❏ 其他的必须指明全路径

包含l字母: List

- ❏ execl,execp,execle
- ❏ 接受可长变参数

包含v字母: vars

- ❏ 字符串数组作为调用程序的参数，以NULL结束数组

进程控制

👤 exec族函数使用例子:

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    char *const ls_argv[] = {"ls", "-al", NULL};

    execl("/bin/ls", "ls", "-al", NULL);
    execlp("ls", "ls", "-al", NULL);

    execv("/bin/ls", ls_argv);
    execvp("ls", ls_argv);

    return 0;
}
```

进程控制

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    printf("start...\n");

    execlp("ls", "ls", "-al", NULL);

    printf("end...\n");

    return 0;
}
```


进程控制

创建新进程

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

- ❏ 功能：复制自己当前进程，产生一个和自己一模一样的进程
- ❏ 返回值：fork()返回两次
 - ❏ 父进程返回子进程的pid
 - ❏ 子进程返回0

进程控制

fork()例1:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    pid_t pid;
    pid = fork();

    printf("hello fork()...\n");

    return 0;
}
```

进程控制

👤 fork()例2:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    pid_t pid;
    pid = fork();
    if(pid > 0)
    {
        printf("i am parent, child pid = %d\n", pid);
    }
    else if(pid == 0)
    {
        printf("i am child\n");
    }else{
        perror("fork()");
    }
    return 0;
}
```


进程控制

fork()例3:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int n = 1;
    pid_t pid;
    pid = fork();
    if (pid > 0) {
        //parent
        sleep(3);
        printf("n = %d\n", n);
    }
    else if (pid == 0)
    {
        n = 3;
    }
    else
    {
        perror("fork()");
    }
    return 0;
}
```

进程控制

fork()技术总结

- ❑ 子进程和父进程继续执行fork后的代码
- ❑ 子进程是父进程的副本，子进程获得父进程数据区、堆和栈的副本，而共享代码区
- ❑ 父进程中所有打开的文件描述符都被复制到子进程中
- ❑ fork后对文件描述符的处理：
- ❑ 父进程等待子进程完成
- ❑ 父、子进程各自执行不同的程序段，这时各自关闭文件描述符
- ❑ fork常见的两种用法：
 1. 一个父进程希望复制自己，使父子执行不同的代码段。网络编程中经常用来处理服务请求
 2. 一个进程要执行一个不同的程序，即子进程执行exec

进程控制-vfork

vfork() 技术

- ❑ 由于fork完整地拷贝了父进程的整个地址空间，因此执行速度是比较慢的。为了提高效率，有些Unix系统设计者创建了vfork。
- ❑ vfork也创建新进程，但它不产生父进程的副本。
- ❑ 它通过允许父子进程可访问相同物理内存从而伪装了对进程地址空间的真实拷贝，当子进程需要改变内存中数据时才拷贝父进程。
- ❑ 这就是著名的“写时拷贝”(copy-on-write)技术

进程控制

人 僵尸进程

- ❖ 考虑一种情况：如果子进程在父进程之前终止，那么父进程又如何能在做相应检查后得到子进程的终止状态呢？
- ❖ 答：内核为每个终止子进程保存了一定量的信息，可以通过某些系统调用函数得到这些信息，得到这些信息后父进程释放子进程仍占用的资源
- ❖ 子进程的善后处理是由其父进程进行的，这些善后包括获取已终止子进程的有关信息 和 释放子进程仍占用的资源
- ❖ 一个已经终止、但其父进程尚未对其进行善后的子进程称为“僵尸进程”，此时该子进程处于僵死状态（zombie）
- ❖ 僵死前，内核就向其父进程发送SIGCHLD信号

进程控制

等待子进程终止

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);

pid_t waitpid(pid_t pid, int *status, int options);
```

- ❑ 功能：如果有子进程成为僵尸，则对子进程进行善后后返回此子进程的pid，否则函数阻塞到有僵尸子进程为止，通俗地讲就是等待子进程终止。
- ❑ @status：保存子进程终止时的信息，不关心时置NULL
- ❑ @pid：指定等待的子进程
- ❑ @options：指定一些特殊处理方式

进程控制

waitpid详解

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);

pid_t waitpid(pid_t pid, int *status, int options);
```

pid 参数详解:

- ❑ -1 等待任一子进程, 这方面等效于wait
- ❑ >0 等待其PID与pid相等的子进程
- ❑ 0 等待同组的任一子进程
- ❑ < -1 等待其组ID等于pid绝对值的任一子进程

options: 0 等效于wait

- ❑ WNOHANG waitpid不阻塞立即返回, 成为一个实时获取子进程状态函数, 此时函数返回值可判断是否为僵尸进程, 为0表示无僵尸进程

进程控制

● 等待子进程结束示例

● 如何避免僵尸进程？

进程控制

❧ 孤儿进程

- ❧ 考虑一下这种情况：父进程已经终止，子进程仍就运行中，此时子进程的父进程应该由谁来担当？
- ❧ 进程的父进程已经消亡，在没有被其它进程“领养”前的进程叫孤儿进程。
- ❧ 孤儿进程最后都由进程号为1的init进程领养。

第三节

进程环境

进程环境

进程的启动

- ❑ C程序总是从main函数开始执行
- ❑ `int main(int argc, char * argv[]);`
- ❑ 要启动程序的父进程调用fork+exec时内核会调用一个启动例程
- ❑ 这个启动例程作为新进程的起始地址
- ❑ 这个启动例程准备好命令行参数和环境变量表，然后调用main

进程环境

进程的终止

- 有8种方式使进程终止

- 5种正常终止

- 从main函数返回 (return)
- 调用exit
- 调用_exit
- 最后一个线程从其启动例程返回
- 最后一个线程调用pthread_exit

- 3种异常终止

- 调用abort
- 接到一个信号并终止
- 最后一个线程对取消请求作出响应

进程环境

exit、_exit:

```
#include <stdlib.h>

void exit(int status);


#include <unistd.h>

void _exit(int status);
```

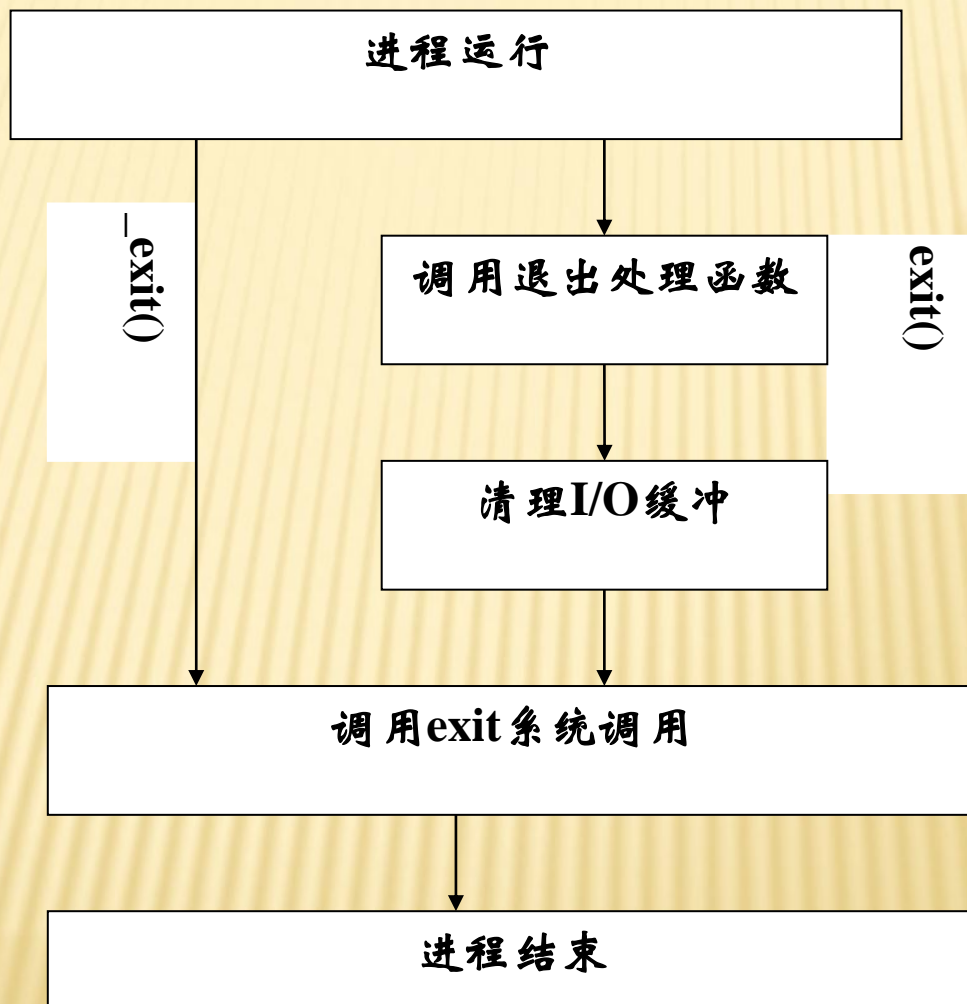
exit: C库函数

-  调用atexit登记的终止处理程序
-  刷新I/O流
-  关闭标准I/O流
-  调用_exit

_exit: 系统调用

-  内核里执行一些清理动作

进程环境-exit和_exit的区别



进程环境-exit和_exit的区别

```
int main()
{
    printf("Using exit...\n");
    printf("This is the end");
    exit(0);
}
```

[root@js]# ./exit_test2

Using exit...

This is the end

[root@js]#

```
int main()
{
    printf("Using _exit...\n");
    printf("This is the end");
    _exit(0);
}
```

[root@js]# ./_exit

Using _exit...

[root@js]#

进程环境

abort(), atexit()

- ⊕ 功能：终止调用的进程
- ⊕ 参数：status：返回给exec系列函数的进程终止状态

```
#include <stdlib.h>
```

```
void abort(void);
```

- ⊕ 退出时登记函数





```
#include <stdlib.h>
```

```
int atexit(void (*function)(void));
```


进程环境

exit、_exit、abort的区别

exit: C库函数

-  调用atexit登记的终止处理程序
-  刷新I/O流
-  关闭标准I/O流
-  调用_exit

_exit: 系统调用

-  内核里执行一些清理动作

abort: C库函数

-  异常终止
-  产生SIGABRT信号

进程环境

人 环境变量表

- ❖ 老式main函数原型:

```
int main(int argc, char *argv[], char * envp[]);
```

- ❖ 现代的方案使用全局变量 `extern char **environ;`

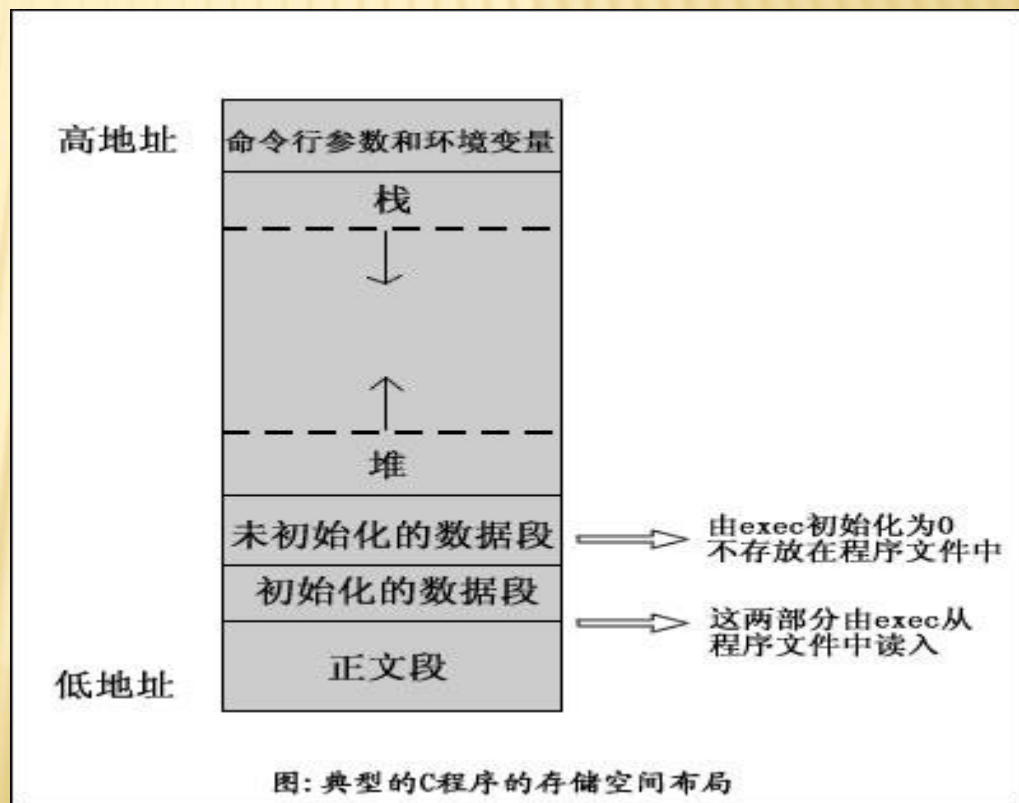
- ❖ 环境变量的内容由” name=value”形式的字符串组成

- ❖ 通常使用`getenv`和`putenv`函数来操作环境变量，而不直接使用全局变量

进程环境

进程的存储空间布局

- 堆区与数据区相邻
- 代码段（正文段）与数据区不相邻
- 堆向上增长
- 栈向下增长



进程环境

共享库和静态库

共享库：又叫动态库 .so .dll

- ❑ 函数库不被包含在可执行文件中
- ❑ 函数库在内存里只有一个副本
- ❑ 所有调用这个函数库的进程共享这段代码段
- ❑ 运行上增加了一些时间开销，发生在第一次被调用时
- ❑ 可执行文件大为缩小的同时节约了内存空间
- ❑ 方便更新函数库的新版本

静态库：.a .lib

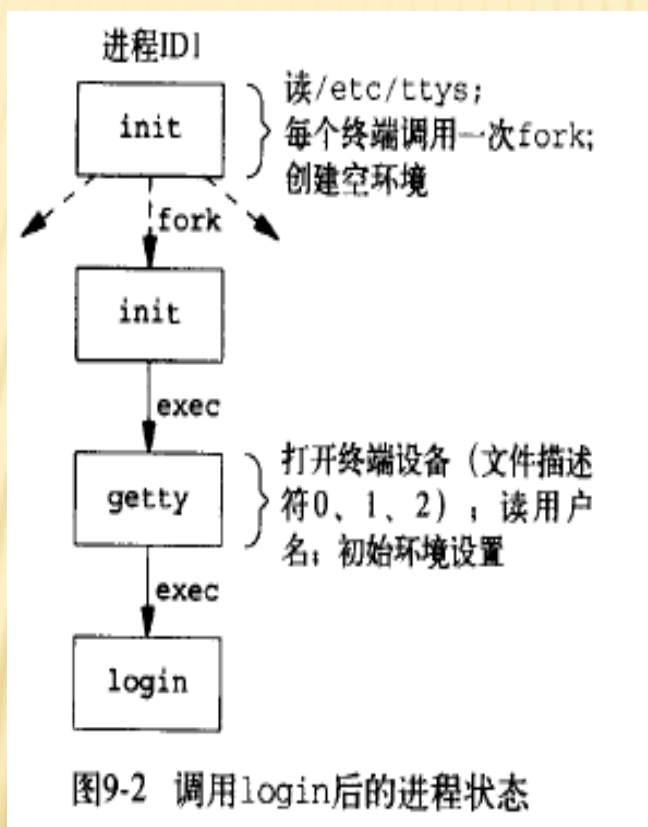
- ❑ 目标文件的简单集合
- ❑ 函数库被包含在可执行文件中
- ❑ 造成内存中存在多个一样的代码段副本
- ❑ 没有额外时间开销
- ❑ 函数库更新必须重新生成可执行文件

第四节

进程关系

进程关系

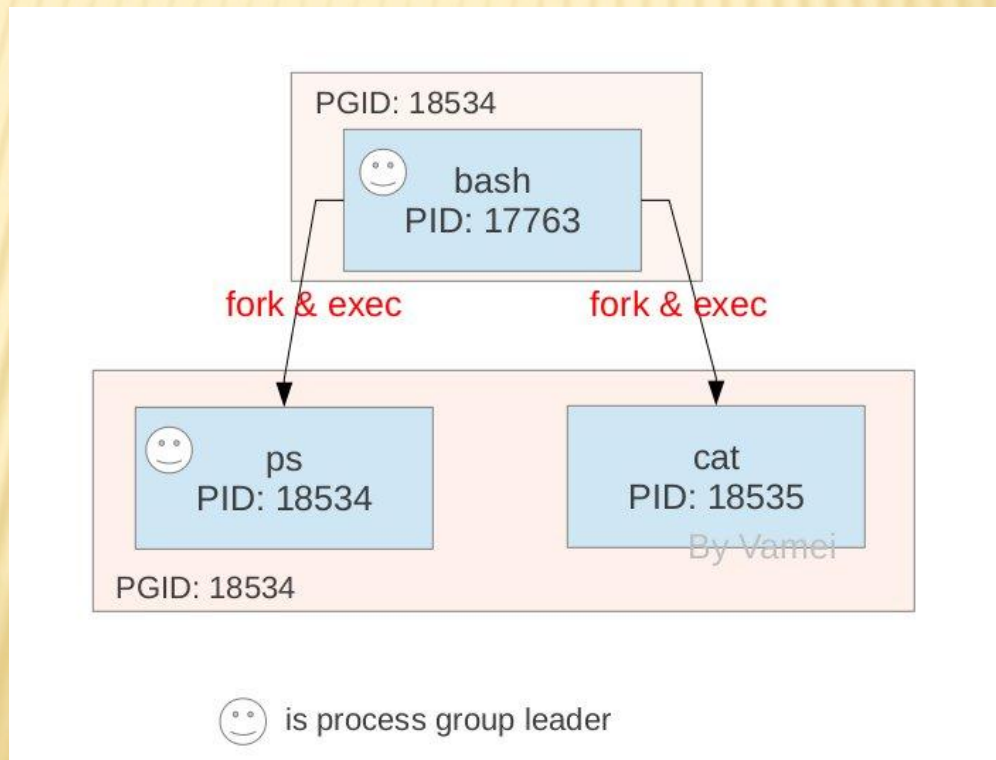
Linux 字符界面命令行启动过程



进程关系

进程组

- 每个进程除了有一个父进程之外，还属于一个进程组
- 进程组是一个或多个进程的集合
- 每个组有个唯一的进程组ID，组长进程的pid等于组ID



进程关系

进程组

- ❏ `pid_t getpgrp(void);`
- ❏ 获取组id

- ❏ `int setpgid(pid_t pid, pid_t pgid);`
- ❏ 设置自己和其子进程的组ID或创建一个新的进程组

进程关系

会话 Session

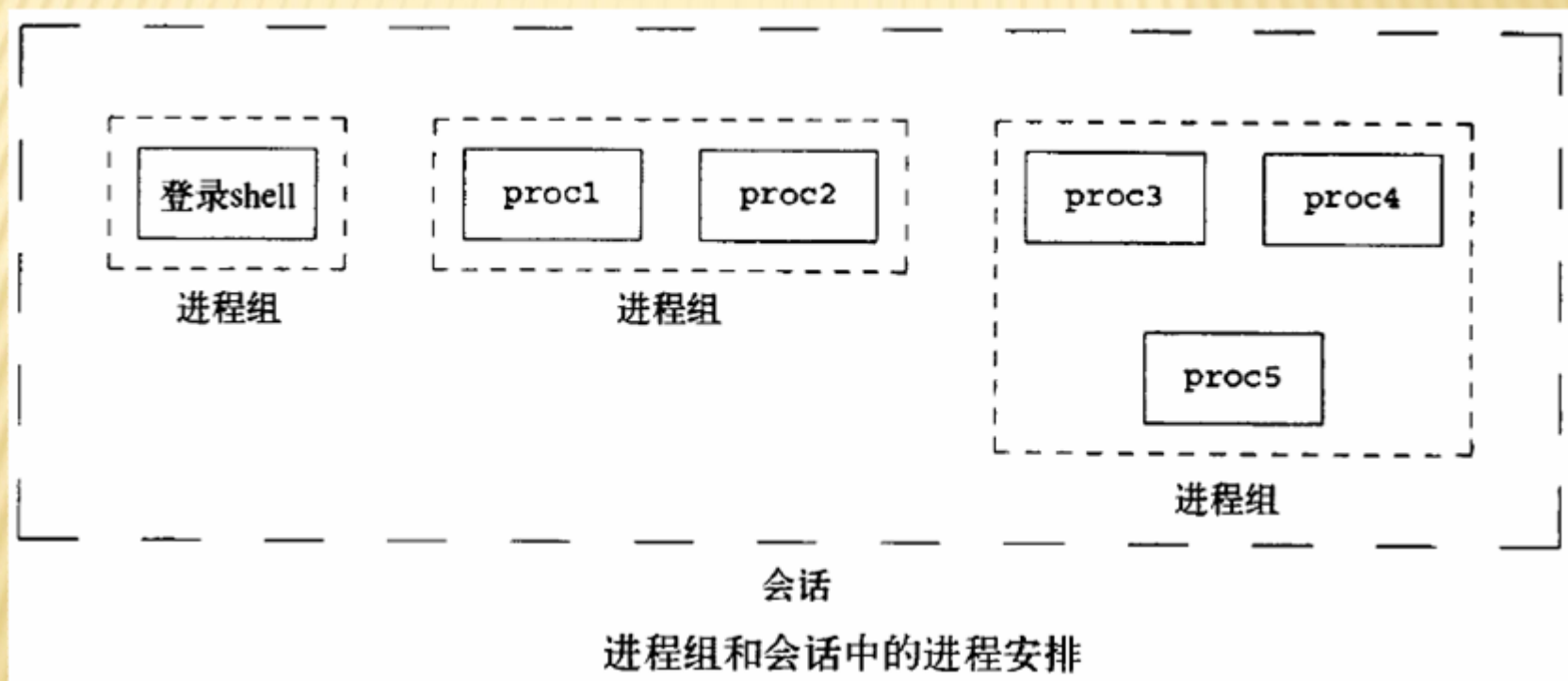
- 会话是一个或多个进程组的集合
- `pid_t setsid(void);` 建立新会话
- 调用进程成为新会话的会话首进程 (session leader)
- 调用进程成为一个新进程组的组长进程

控制终端

- 一个会话可以有一个控制终端
- 具有控制终端的会话其首进程此时叫控制进程
- 一个会话的几个进程组可以分为一个前台进程组和多个后台进程组
- 按键 (如`ctrl+c`中断和`ctrl+\`退出) 产生的信号将发送给前台进程组中的所有进程

进程关系

控制终端的进程组和会话



第五节

守护进程

守护进程

什么是守护进程？

- ❏ `$ps -efjc` 或者 `ps -axj`
- ❏ 守护进程也称为精灵进程（daemon），是一种生存期较长的进程
- ❏ 特征：
 - ❏ 守护进程没有控制终端
 - ❏ 无法在屏幕上打印信息也无法接收键盘输入
 - ❏ 后台运行
 - ❏ 常常在系统自举时启动，在系统关闭时才终止
 - ❏ 进程组的组长进程以及会话的首进程，也是唯一进程
 - ❏ 大多数守护进程的父进程是init进程
 - ❏ 守护进程一般都是以超级用户身份特权运行

守护进程

守护进程编码规则

- 1. 调用umask将文件模式创建屏蔽字设置为0，这个屏蔽字继承自父进程
- 2. 调用fork，然后父进程退出（exit）。保证子进程不会是组长进程
- 3. 调用setsid创建一个新会话，于是子进程，成为新会话的首进程，成为一个新进程组的组长进程
- 4. 将当前工作目录更改为根目录（防止当前文件系统是mount上的分区，导致没法umount）
- 5. 关闭不再需要的文件描述符
- 6. 使用syslog（/dev/log）输出调试信息

守护进程-创建子进程 父进程退出

创建子进程 父进程退出

- ❑ 第一步完成以后，子进程就在形式上做到了与控制终端的脱离调用fork，然后父进程退出（exit）。保证子进程不会是组长进程
- ❑ 由于父进程已经先于子进程退出，子进程变成孤儿进程

❑ pid = fork();

```
if (pid > 0) /*父进程退出*/  
{  
    exit(0);  
}
```

守护进程-在子进程中创建新会话

在子进程中创建新会话

- setsid函数作用
- setsid函数用于创建一个新的会话，并使得当前进程成为新会话组的组长
- setsid函数能够使进程完全独立出来，从而脱离所有其他进程的控制。

```
#include <unistd.h>
pid_t setsid(void);
```

返回值：若成功则返回进程组ID，若出错则返回-1

```
#include <unistd.h>
pid_t getsid(pid_t pid);
```

返回值：若成功则返回会话首进程的进程组ID，若出错则返回-1

守护进程-改变当前目录为根目录

改变当前目录为根目录

- ❑ 通常的做法是让“/”或“/tmp”作为守护进程的当前工作目录。
- ❑ 在进程运行过程中，当前目录所在的文件系统是不能卸载的。
- ❑ chdir函数可以改变进程当前工作目录

守护进程-重设文件全下掩码

重设文件全下掩码

- ❑ 文件权限掩码是指文件权限中被屏蔽掉的对应位。把文件权限掩码设置为0，可以增加该守护进程的灵活性。
- ❑ 设置文件权限掩码的函数是umask
- ❑ 通常的使用方法为umask(0)

守护进程-关闭文件描述符

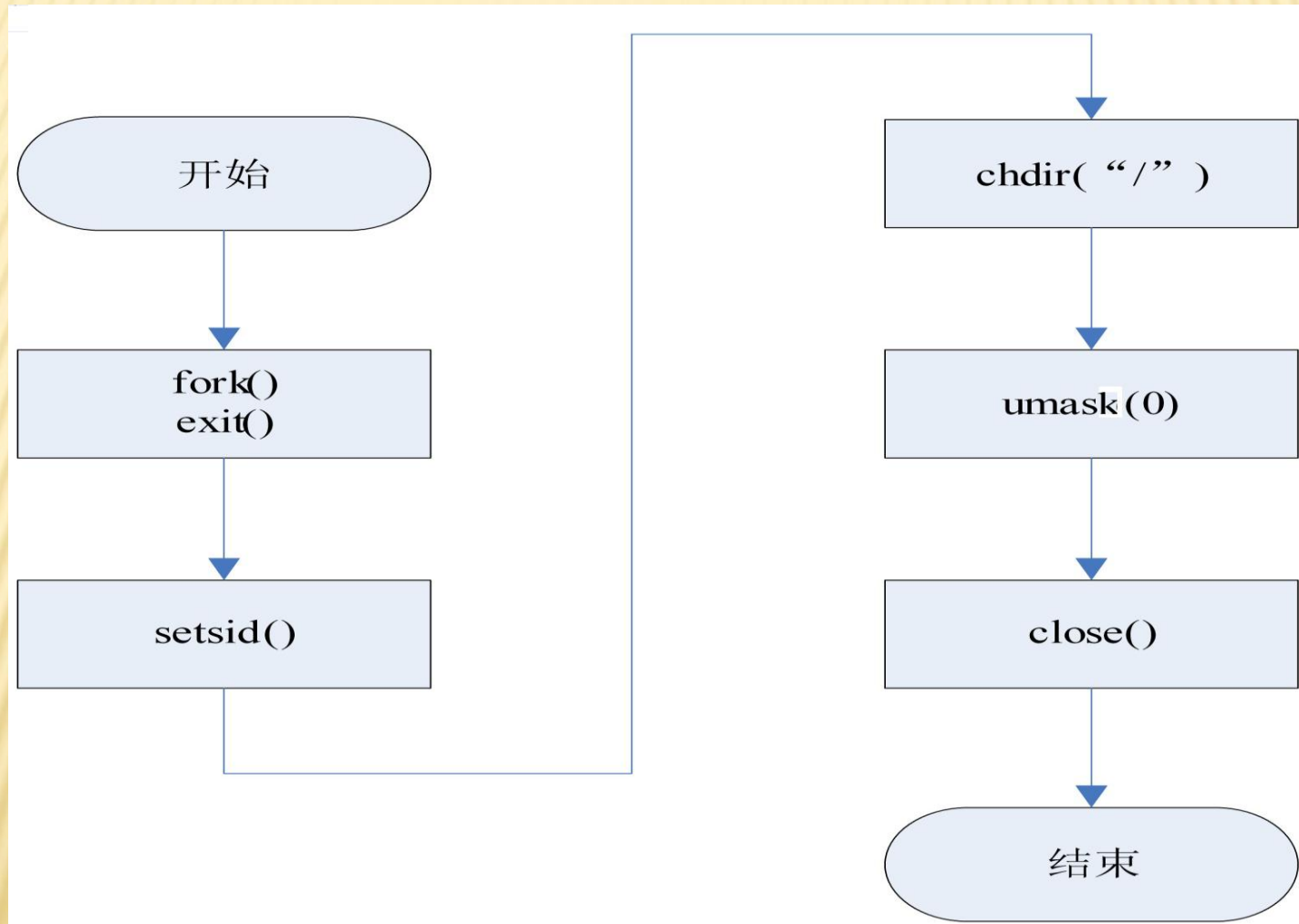
重设文件全下掩码

- 新建的子进程会从父进程那里继承所有已经打开的文件。
- 在创建完新的会话后，守护进程已经脱离任何控制终端，应当关闭用不到的文件



```
for (i=0; i<MAXFILE; i++)  
{  
    close(i);  
}
```

守护进程-创建流程



守护进程

示例代码

 testdaemon.c

第六节 信号

信号

什么是信号？

- 生活中经商定作为采取一致行动的暗号
- Unix家族操作系统中：是一种进程间通讯的有限制的方式
- 是一种异步的通知机制，用来提醒进程一个事件已经发生
- 信号只是用来通知某进程发生了什么事，并不给该进程传递任何数据
- 当一个信号发送给一个进程，操作系统中断了进程正常的控制流程，此时，任何非原子操作都将被中断
- 如果进程定义了信号的处理函数，那么它将被执行，否则就执行默认的处理函数
- 一个进程可以给其它进程和自己发送信号
- 内核也可以给所有进程发送信号

信号

进程对收到的信号处理方法：

- ❑ 第一种是类似中断的处理程序，对于需要处理的信号，进程可以指定处理函数，由该函数来处理。
- ❑ 忽略某个信号，对该信号不做任何处理，就象未发生过一样，这也叫信号屏蔽，这由PCB32位的信号屏蔽成员控制
- ❑ 对该信号的处理保留系统的默认值，这种缺省操作，对大部分的信号的缺省操作是使得进程终止

信号

常见的信号:

❑ SIGHUP	1	A	终端挂起或者控制进程终止
❑ SIGINT	2	A	键盘中断 (如break键被按下)
❑ SIGQUIT	3	C	键盘的退出键被按下
❑ SIGILL	4	C	非法指令
❑ SIGABRT	6	C	由abort(3)发出的退出指令
❑ SIGFPE	8	C	浮点异常
❑ SIGKILL	9	AEF	Kill信号
❑ SIGSEGV	11	C	无效的内存引用
❑ SIGPIPE	13	A	管道破裂: 写一个没有读端口的管道
❑ SIGALRM	14	A	由alarm(2)发出的信号
❑ SIGTERM	15	A	终止信号

信号

常见的信号：

- 处理动作一项中的字母含义如下：
 - A 缺省的动作是终止进程
 - B 缺省的动作是忽略此信号
 - C 缺省的动作是终止进程并进行内核映像转储 (dump core)
 - D 缺省的动作是停止进程
 - E 信号不能被捕获
 - F 信号不能被忽略

信号

通过命令发信号给指定进程

给指定的进程发信号

- kill -num pid

- kill -XXX pid //XXX, 不带SIG信号名称如KILL、ALRM等等

给所有可执行文件名叫指定名称的进程发信号

- killall -num 程序名

- killall -XXX 程序名

信号

在代码给其它进程发信号

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

- 功能：给指定的进程发信号
- 返回值：成功返回0，否则< 0
- 参数：
 - signo参数可以直接使用序号，或预定义好的常量宏如SIGINT（推荐）
 - pid > 0 表示接受信号的进程的PID
 - pid == 0 给同组（进程组）进程发送信号
 - pid < 0 给进程组ID等于pid绝对值的进程发送信号
 - pid == -1 发给本进程有权限给对方发信号的所有进程

信号

给进程自身发信号

```
#include <signal.h>

int raise(int sig);
```

- 返回值：成功返回0，否则< 0
- 参数：signo参数可以直接使用序号，或预定义好的常量宏如SIGINT\
- 完全等价于kill(getpid(),signo)

信号-发送和捕捉

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>

int main()
{
    pid_t pid;
    pid = fork();
    if(pid < 0) /*创建一子进程*/
    {
        perror("fork");
        exit(1);
    }
    if(pid == 0)
    {
        raise(SIGSTOP); /*发出SIGSTOP信号*/
        printf("child process exit...\n");
        exit(0);
    }
}
```

信号-发送和捕捉

```
else    /*在父进程中检测子进程的状态, 调用kill函数*/
{
    printf("pid = %d\n", pid);
    if((waitpid(pid, NULL, WNOHANG)) == 0)
    {
        kill(pid, SIGKILL);
        printf("kill %d\n", pid);
    }
}

return 0;
}
```

信号-发送和捕捉

alarm()和pause()

- alarm()也称为闹钟函数，它可以在进程中设置一个定时器。当定时器指定的时间到时，内核就向进程发送SIGALARM信号。

```
#include <unistd.h>
```

```
unsigned int alarm(unsigned int seconds);
```

- 成功：如果调用此alarm()前，进程中已经设置了闹钟时间，则返回上一个闹钟时间的剩余时间，否则返回0。
- 失败返回-1

- pause()函数是用于将调用进程挂起直到收到信号为止。

```
#include <unistd.h>
```

```
int pause(void);
```


信号-发送和捕捉

👤 alarm()和pause()测试

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    int ret;
    /*调用alarm定时器函数*/
    ret = alarm(5);

    pause();
    printf("I have been waken up.\n");

    return 0;
}
```

信号-信号的处理

人 信号处理方式

- ❑ 第一种是类似中断的处理程序，对于需要处理的信号，进程可以指定处理函数，由该函数来处理。
- ❑ 第二种方法是，忽略某个信号，对该信号不做任何处理，就象未发生过一样。
- ❑ 第三种方法是，对该信号的处理保留系统的默认值，这种缺省操作，对大部分的信号的缺省操作是使得进程终止。进程通过系统调用`signal`来指定进程对某个信号的处理行为。

人 指定信号处理方式

- ❑ 使用简单的`signal()`函数
- ❑ 使用信号集函数组

信号-信号的处理

👤 信号处理绑定函数

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

- signum: 指定信号
- handler: 信号处理函数
 - SIG_IGN: 忽略该信号。
 - SIG_DFL: 采用系统默认方式处理信号。
 - 自定义的信号处理函数指针。
- 返回值: 成功: 设置之前的信号处理方式, 失败返回-1

信号-信号的处理

```
void my_func(int signo)
{
    if (signo == SIGINT)
    {
        printf("I have got SIGINT\n");
    }
    else if (signo == SIGQUIT)
    {
        printf("I have got SIGQUIT\n");
    }
}

int main()
{
    printf("waiting for signal SIGINT or SIGQUIT\n ");

    signal(SIGINT, my_func);
    signal(SIGQUIT, my_func);

    while(1)
        ;

    return 0;
}
```

信号

检查或修改与指定信号关联的处理动作

```
#include <signal.h>
```


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

- 返回值：成功返回0，否则< 0
- 参数：
 - act：非空时修改对应信号处理动作
 - oact：非空时保存对应信号的上一个动作

```
struct sigaction  
{  
    void (*sa_handler)(int); /*新的信号处理函数指针*/  
    void (*sa_sigaction)(int, siginfo_t *, void *);  
    sigset_t sa_mask; /*新的进程信号屏蔽字*/  
    int sa_flags; /*信号处理的相关操作, SA_SIGINFO */  
    void (*sa_restorer)(void);  
};
```

信号

 sigaction 示例

 testsig.c

信号

👤 信号集

```
#include <signal.h>

int sigemptyset(sigset_t *set);           /*清空信号集*/

int sigfillset(sigset_t *set);            /*信号集里包含所有信号*/

int sigaddset(sigset_t *set, int signum); /*向信号集里增加信号*/

int sigdelset(sigset_t *set, int signum); /*从信号集里删除信号*/

int sigismember(const sigset_t *set, int signum); /*判断信号集是否存在指定信号*/
```

信号

👤 sigaction 示例

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

- ❖ sigprocmask 设定对信号屏蔽集内的信号的处理方式(阻塞或不阻塞)。
- ❖ 参数：how：用于指定信号修改的方式，可能选择有三种SIG_BLOCK或SIG_UNBLOCK与SIG_SETMASK 赋值
- ❖ set：为指向信号集的指针，在此专指新设的信号集，如果仅想读取现在的屏蔽值，可将其置为NULL。
- ❖ oldset：也是指向信号集的指针，在此存放原来的信号集。可用来检测信号掩码中存在什么信号。
- ❖ 返回说明：成功执行时，返回0。失败返回-1，errno被设为EINVAL。

课程总结

本节课程内容

- ▣ 进程管理
- ▣ 进程环境
- ▣ 进程控制
- ▣ 进程关系
- ▣ 守护进程
- ▣ 信号

下节课程

- ▣ 文件共享
- ▣ 临界同步
- ▣ 互斥资源
- ▣ 死锁饥饿

联系方式

QQ: 59189174

E-mail: yumeifly@sohu.com