

第十章 进程运行与监控

授课教师

电子邮箱：



主要内容

- **Linux进程控制块**
- 进程的启动
- 进程的运行控制
- 进程的监测



进程在内核中的表现形式：进程控制块 (PCB)

进程ID
用户ID
进程状态
调度信息
文件管理
虚拟内存管理
信号（进程间通信机制）
时间和定时器
.....



Linux进程控制块: task_struct结构



```
pid_t pid;  
uid_t uid,euid;  
gid_t gid,egid;  
volatile long state;  
int exit_state;  
unsigned int rt_priority;  
unsigned int policy;  
struct list_head tasks;  
struct task_struct *real_parent;  
struct task_struct *parent;  
struct list_head children,sibling;
```



Linux进程控制块: task_struct结构

```
struct fs_struct *fs;    /fs用来表示进程与文件系统的联系，包括当前目录和根目录
struct files_struct *files; /files表示进程当前打开的文件
struct mm_struct *mm;
struct signal_struct *signal;
struct sighand_struct *sighand;
cputime_t utime, stime;
struct timespec start_time; /start_time/real_start_time进程创建时间，real_start_time还包含了进程睡眠时间
struct timespec real_start_time;
```



task_struct: 进程状态

■ 进程状态:

- volatile long state;

■ state成员的可能取值如下:

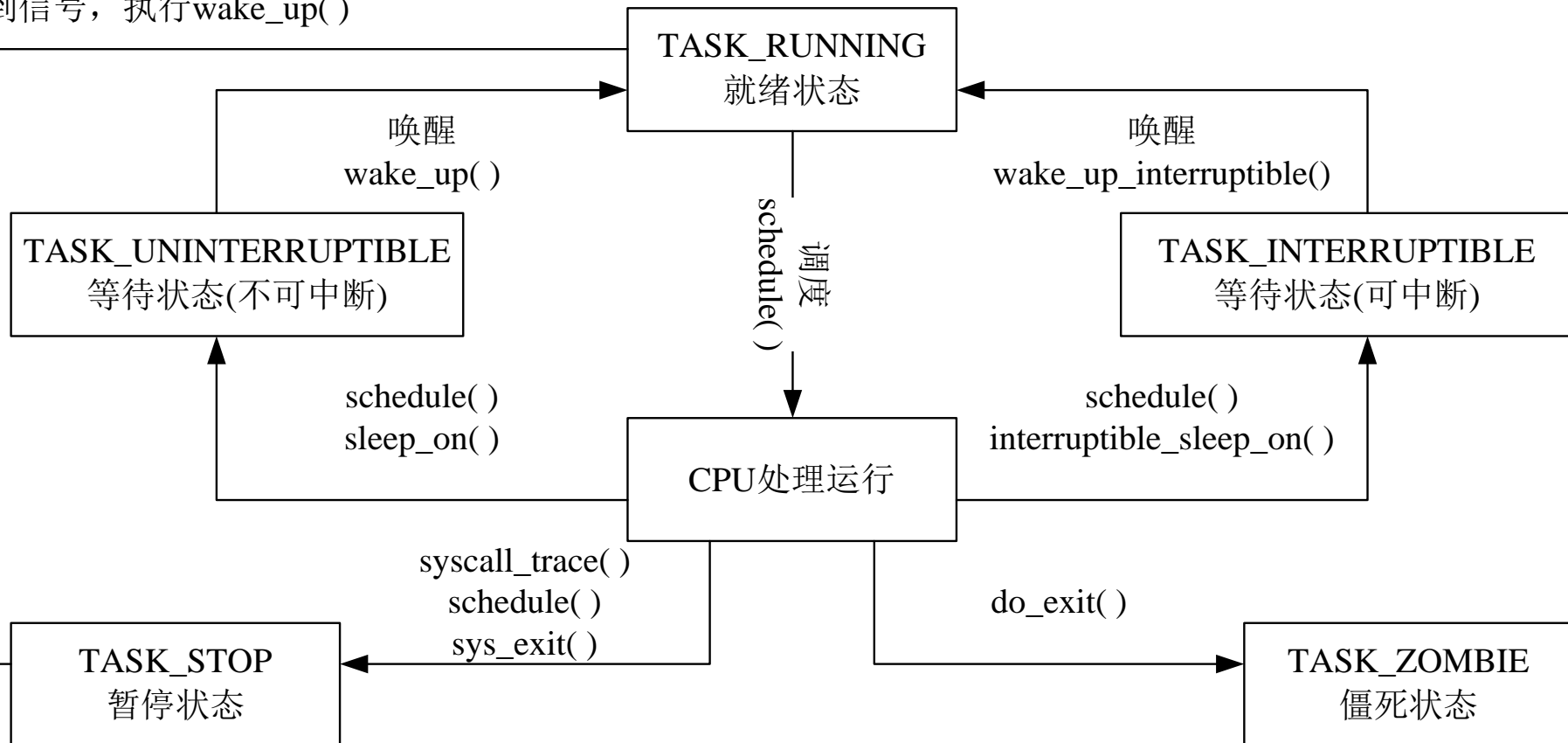
- #define TASK_RUNNING 0
- #define TASK_INTERRUPTIBLE 1
- #define TASK_UNINTERRUPTIBLE 2
- #define TASK_ZOMBIE 4
- #define TASK_STOPPED 8



进程状态切换

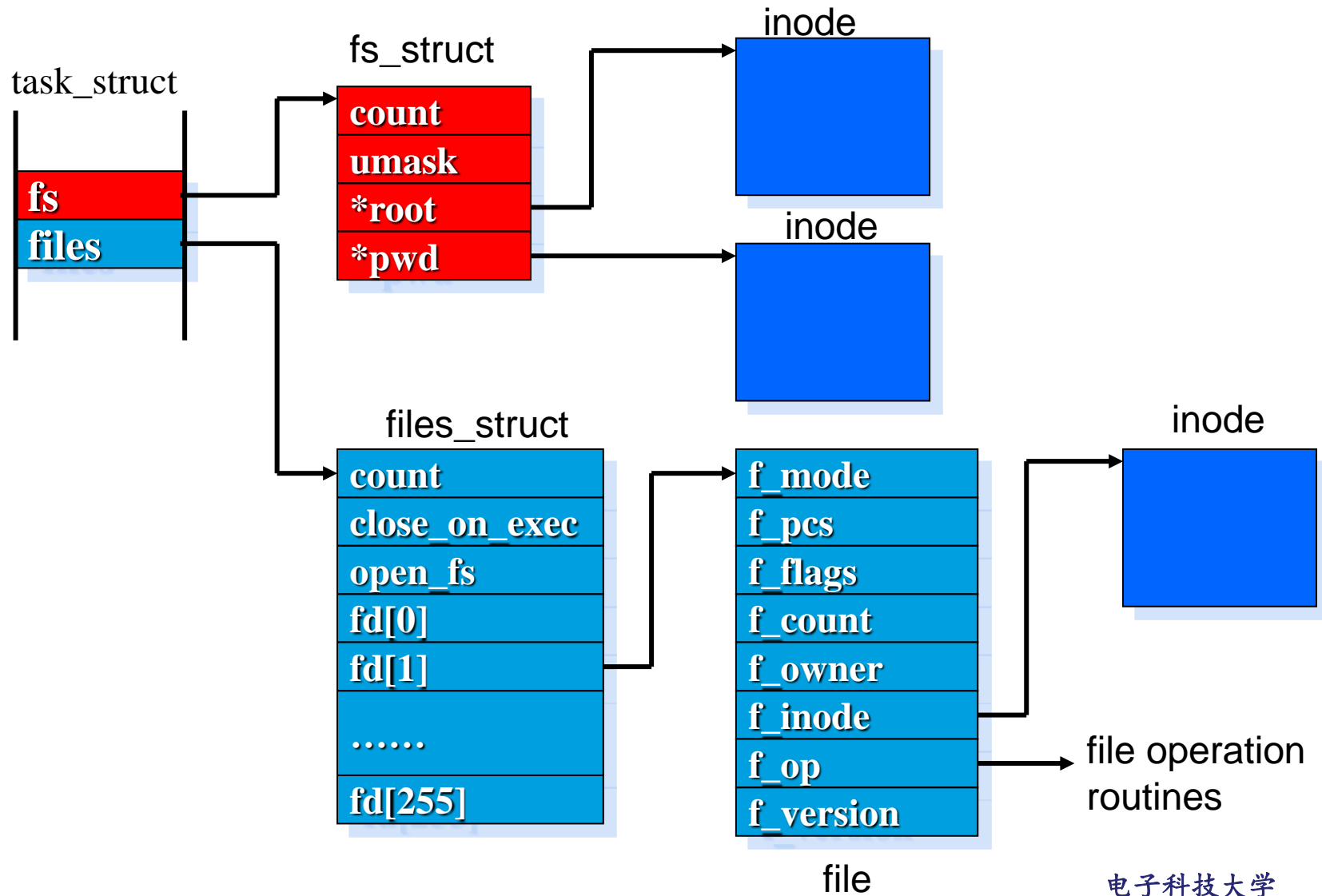


收到信号，执行wake_up()





task_struct: 文件管理





主要内容

- Linux 进程控制块
- 进程的启动
- 进程的运行控制
- 进程的监测



创建进程

- UNIX&Linux中创建进程的方式:
 - 在shell中执行命令或可执行文件
 - 由shell进程调用fork函数创建子进程
 - 在代码中（已经存在的进程中）调用fork函数创建子进程
 - fork创建的进程为子进程
 - 原进程为父进程



创建进程

- Linux系统中进程0（PID=0）是由内核创建其他所有进程都是由它创建的
- Linux系统中进程0在（init进程）后，进程0创建进程
- 进程1（init进程）是共同祖先

```
root@ubuntu:~# pstree
init--ModemManager--2*[{ModemManager}]
    --NetworkManager--dhclient
                        dnsmasq
                        3*[{NetworkManager}]
    --accounts-daemon--2*[{accounts-daemon}]
    --acpid
    --avahi-daemon--avahi-daemon
    --bluetoothd
    --colord--2*[{colord}]
    --cron
    --cups-browsed
    --cupsd
    --dbus-daemon
    --6*[{getty}]
    --gnome-keyring-d--6*[{gnome-keyring-d}]
    --kerneloops
    --lightdm--Xorg
                --lightdm--init--at-spi-bus-laun--dbus-daemon
                                                3*[{at-spi-bus-laun}]
                --at-spi2-registr--{at-spi2-registr}
                --bamfdaemon--3*[{bamfdaemon}]
                --2*[{dbus-daemon}]
                --dbus-launch
                --2*[{dconf-service}--2*[{dconf-service}]]
                --evolution-calen--4*[{evolution-calen}]
                --evolution-sourc--2*[{evolution-sourc}]
                --gconfd-2
                --gnome-session--compiz--4*[{compiz}]
                                --deja-dup-monito--3*[{deja-dup-m+}]
                                --nautilus--3*[{nautilus}]
                                --nm-applet--2*[{nm-applet}]
                                --polkit-gnome-au--2*[{polkit-gno+}]
                                --telepathy-indic--2*[{telepathy-+}]
                                --unity-fallback--2*[{unity-fall+}]
                                --update-notifier--3*[{update-not+}]
                                --zeitgeist-datah--3*[{zeitgeist-+}]
                                3*[{gnome-session}]
                --gnome-terminal--bash--sudo--bash--sudo--su--++
                                --gnome-pty-helpe
                                3*[{gnome-terminal}]
                                --gvfs-afc-volume--2*[{gvfs-afc-volume}]
```



- **函数原型**
 - **头文件: `unistd.h`**
 - **`pid_t fork(void);`**
- **返回值**
 - **`fork`函数被正确调用后, 将会子进程中和父进程中分别返回!!**
 - **在子进程中返回值为0 (不合法的PID, 提示当前运行在子进程中)**
 - **在父进程中返回值为子进程ID (让父进程掌握所创建子进程的ID号)**
 - **出错返回-1**



创建子进程示例

```
int main(void) {  
    pid_t pid;  
    pid=fork();  
    if(pid==-1)  
        printf(“fork error\n” );  
    else if(pid==0) {  
        printf(“the returned value is %d\n”,pid);  
        printf(“in child process!!\n” );  
        printf(“My PID is %d\n”,getpid());}  
    else{  
        printf(“the returned value is %d\n”,pid);  
        printf(“in father process!!\n” );  
        printf(“My PID is %d\n”,getpid());}  
    return 0;  
}
```

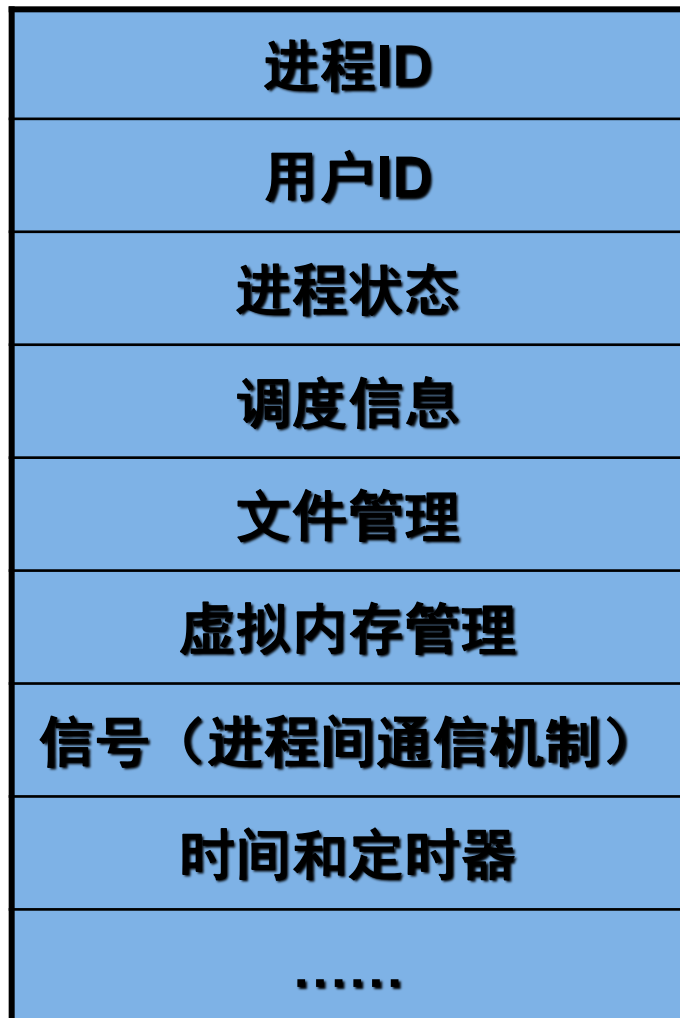
```
wuxiaolin@wuxiaolin-virtual-machine: ~/LinuxC/Progress  
wuxiaolin@wuxiaolin-virtual-machine:~/LinuxC/Progress$ ./progress  
The returned value is 2547  
In father process!!  
My PID is 2546  
The returned value is 0  
In child process!!  
My PID is 2547
```

父进程

子进程



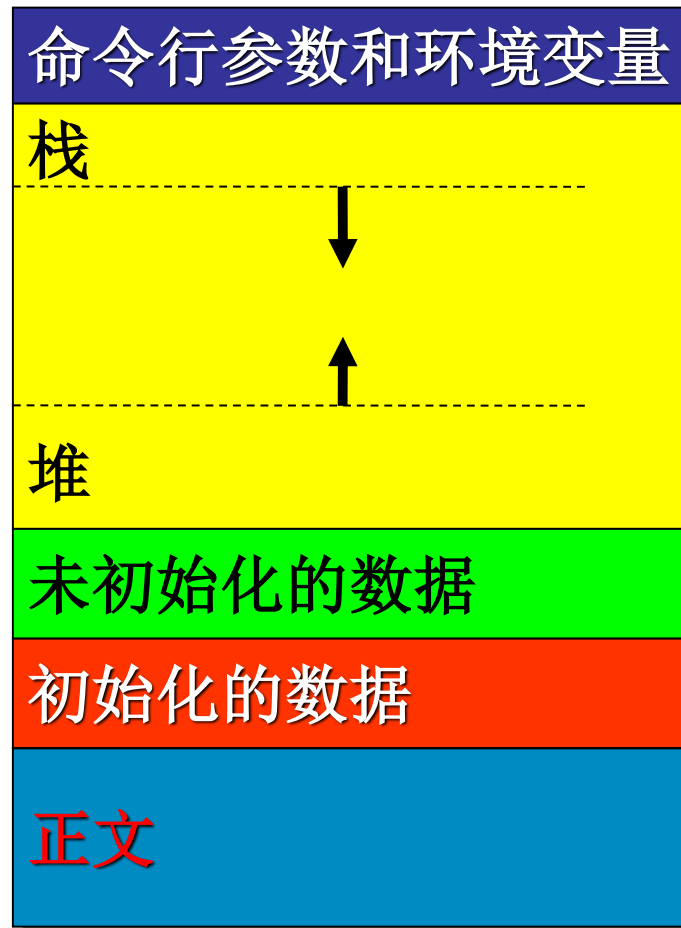
进程内存空间布局



内核空间（PCB）

High address

Low address



用户空间



进程内存空间

程序/Disk

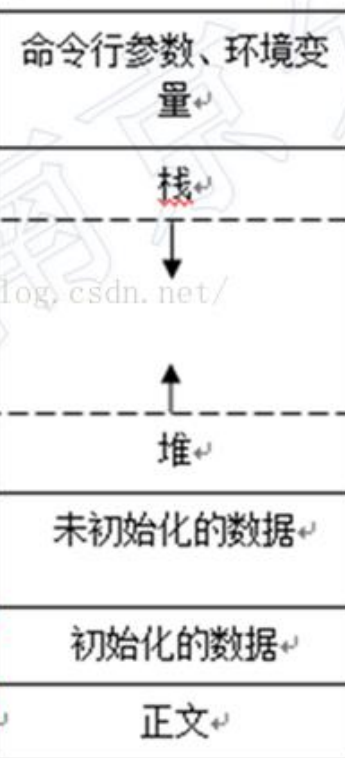
文件格式



进程/RAM

内存地址空间

高地址



```
struct task_struct {  
    volatile long state;  
    ...  
    pid_t pid;  
    pid_t tgid;  
    ...  
};
```

+

低地址



进程内存空间布局

High address

命令行参数和环境变量

栈



堆

未初始化的数据

初始化的数据

正文

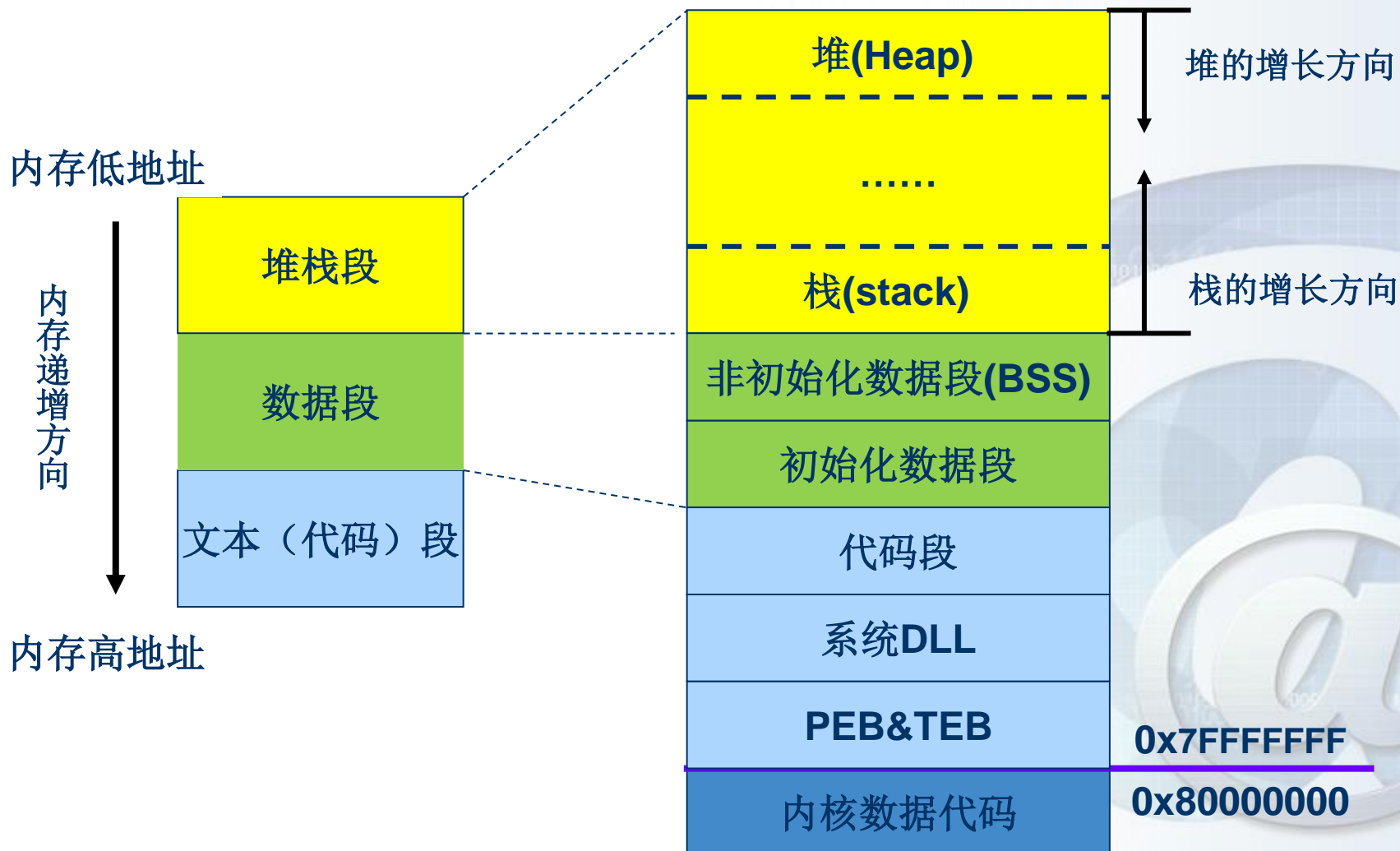
Low address

主要用于支撑函数调用
存放参数、局部变量等

用于动态分配内存

全局变量 `long sum[1000];`

补充：程序在内存中的映像



补充：PEB&TEB



- ❖ **PEB** (**Process Environment Block**, 进程环境块) 存放进程信息, 每个进程都有自己的 **PEB** 信息。位于用户地址空间。
- ❖ 在 **Win 2000** 下, 进程环境块的地址对于每个进程来说是固定的, 在 **0x7FFDF000** 处, 这是用户地址空间, 所以程序能够直接访问。在用户态下 **WinDbg** 中可用命令 **\$proc** 取得 **PEB** 地址。

补充：PEB&TEB

- ❖ **TEB (Thread Environment Block, 线程环境块)** 系统在此**TEB**中保存频繁使用的线程相关的数据。位于用户地址空间，在比 **PEB** 所在地址低的地方。
- ❖ 进程中的每个线程都有自己的一个**TEB**。一个进程的所有**TEB**都以堆栈的方式，存放在从 **0x7FFDE000** 开始的线性内存中，每**4KB**为一个完整的**TEB**，不过该内存区域是向下扩展的。

栈

❖ 栈是一块连续的内存空间

- 先入后出
- 生长方向与内存的生长方向正好相反, 从高地址向低地址生长

❖ 每一个线程有自己的栈

- 提供一个暂时存放数据的区域

❖ 使用**POP/PUSH**指令来对栈进行操作

❖ 使用**ESP**寄存器指向栈顶, **EBP**指向栈帧底

栈内容



- ❖ 函数的参数
- ❖ 函数返回地址
- ❖ **EBP**的值
- ❖ 一些通用寄存器(**EDI,ESI...**)的值
- ❖ 当前正在执行的函数的局部变量

三个重要的寄存器



❖ SP(ESP)

- 即栈顶指针，随着数据入栈出栈而发生变化

❖ BP(EBP)

- 即基地址指针，用于标识栈中一个相对稳定的位置。
通过BP,可以方便地引用函数参数以及局部变量

❖ IP(EIP)

- 即指令寄存器，在将某个函数的栈帧压入栈中时，其中就包含当前的IP值，即函数调用返回后下一个执行语句的地址

函数调用过程



- ❖ 把参数压入栈
- ❖ 保存指令寄存器中的内容，作为返回地址
- ❖ 放入堆栈当前的基址寄存器
- ❖ 把当前的栈指针(**ESP**)拷贝到基址寄存器，作为新的基地址
- ❖ 为本地变量留出一定空间，把**ESP**减去适当的数值



函数调用中栈的工作过程

❖ 调用函数前


- 压入栈
 - 上级函数传给A函数的参数
 - 返回地址(EIP)
 - 当前的EBP
 - 函数的局部变量

❖ 调用函数后

- 恢复EBP
- 恢复EIP
- 局部变量不作处理




例子



```
int main()
{AFunc(5,6); return 0;}
```

```
int AFunc(int i,int j)
{
    int m = 3;
    int n = 4;
    m = i;
    n = j;
    BFunc(m,n);
    return 8;
}
```

```
int BFunc(int i,int j)
{
    int m = 1;
    int n = 2;
    m = i;
    n = j;
    return m;
}
```



函数调用中栈的工作过程

main()

AFunc(5,6);

push 6

push 5

call _AFunc

add esp+8

_AFunc 当前EBP →

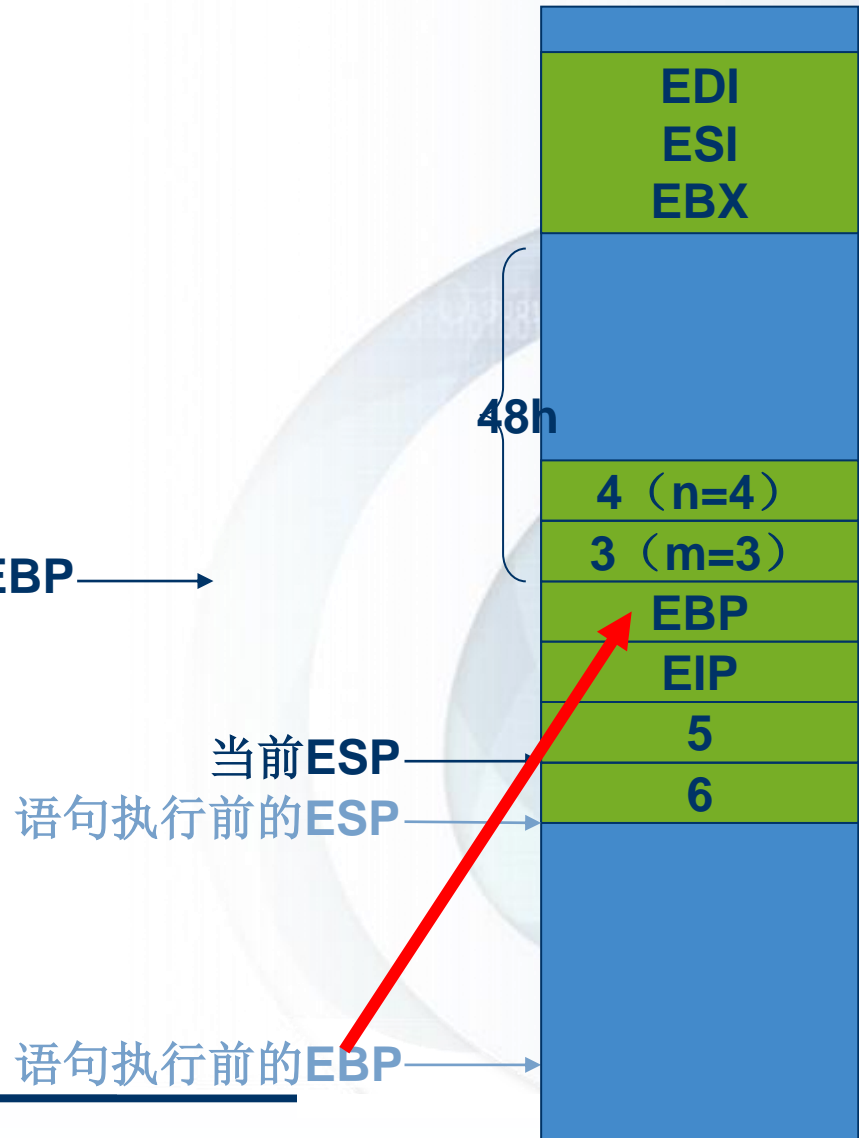
push ebp

mov ebp,esp

sub esp,48h

//压入环境变量

//为局部变量分配空间



函数调用中栈的工作过程

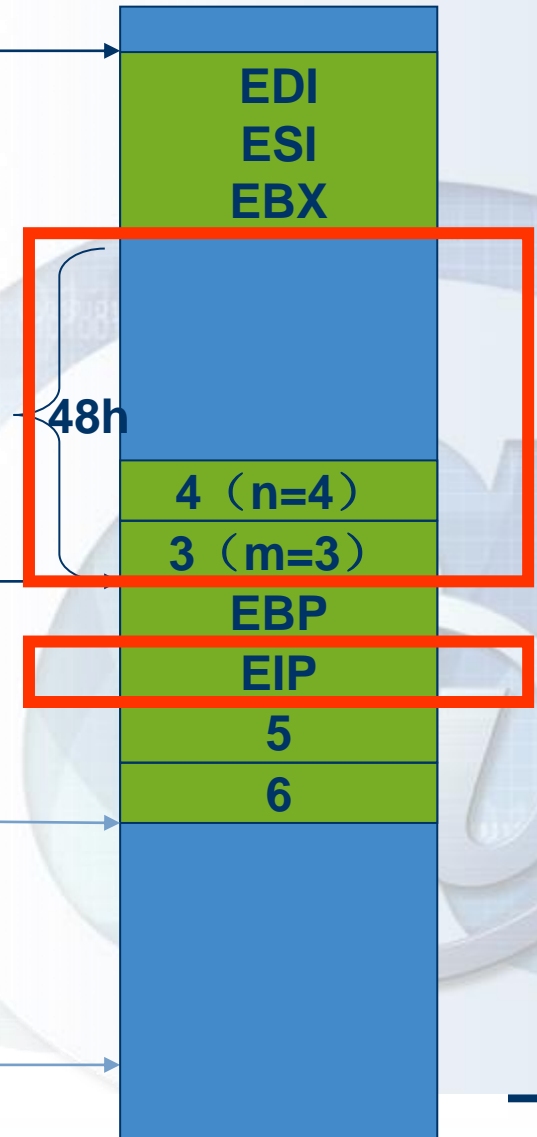
AFunc(5,6);
.....
call _AFunc
add esp+8
_AFunc
{.....return 0;}
pop edi
pop esi
pop ebx
add esp,48h
//栈校验
pop ebp
ret

当前ESP

当前EBP

语句执行前的ESP

语句执行前的EBP





命令行参数

- `ls [参数] <路径或文件名>`
 - `ls -l /home`
- `mkdir [参数] <目录名>`
 - `mkdir -p /home/xiaokun/src`
- `cp [参数] <源文件路径> <目标文件路径>`
 - `cp -r /usr/local/src /root`



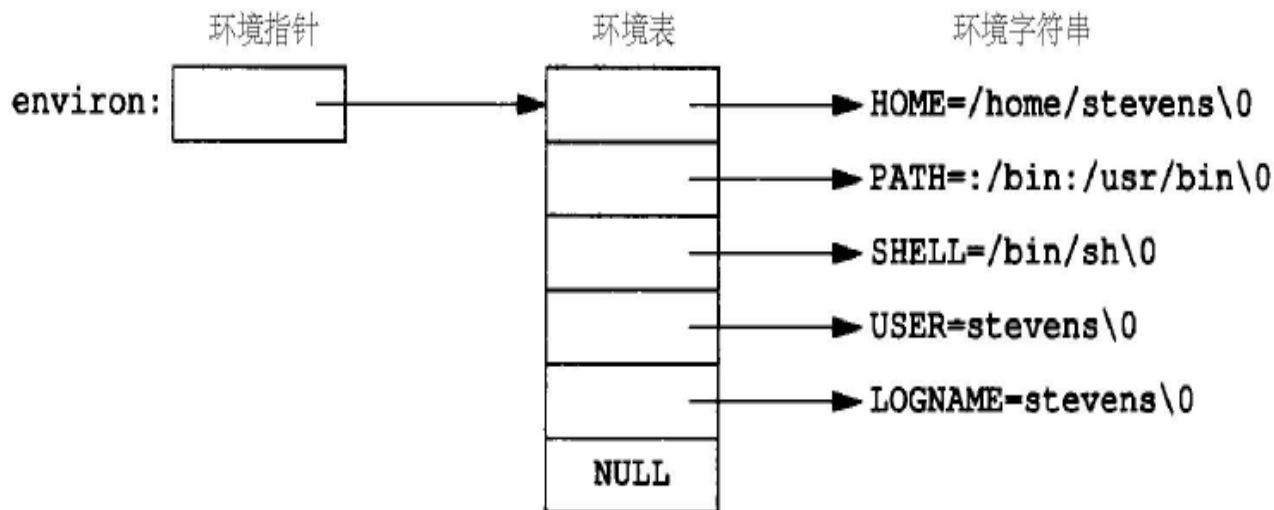
环境变量

- 环境变量一般是指操作系统中指定操作系统运行环境的一些参数。它相当于一个指针，想要查看变量的值，需要加上“\$”。
- 每个进程都有一张环境变量表，环境变量表是一个字符指针数组，每个指针指向一个以‘\0’结尾的环境字符串。
- Main函数的第三个参数就是环境表地址。



环境变量表

- 通过全局的环境指针（environ）可以直接访问环境变量表（字符串数组）
 - 头文件unistd.h
 - `extern char **environ;`
- 环境变量字符串形式为“name=value”，name是环境变量名称，value为环境变量赋值





- 设置环境变量的三种方法：
 - **putenv**
 - **setenv**
 - **unsetenv**
- **putenv**函数将环境变量字符串放入环境变量表中；若该字符串已经存在，则覆盖
- **头文件：stdlib.h**
- **int putenv(char *str);**



▪ **setenv**

- 头文件: `stdlib.h`
- `int setenv(const char* name, const char* value, int rewrite);`
- `setenv`将指定环境变量的值设置为参数指定值（更改环境变量字符串）
- 若`name`已经存在
 - `rewrite`不等于0，则删除其原先的定义
 - `rewrite`等于0，则不删除其原先的定义

▪ **unsetenv**

- 头文件: `stdlib.h`
- `int unsetenv(const char* name);`
- 删除指定的环境变量字符串



▪ 子进程是父进程的副本

- 子进程复制/拷贝父进程的PCB、用户空间（数据段、堆和栈）
- 父子进程共享正文段（只读）

▪ 父进程继续执行fork函数调用之后的代码

，子进程也从fork函数调用之后的代码开始执行

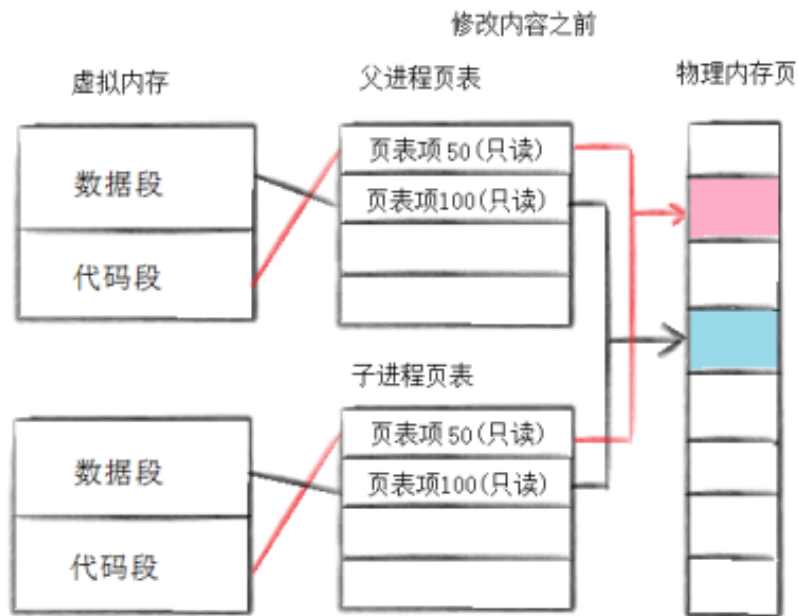
▪ 为了提高效率，fork后并不立即复制父进程数据段、堆和栈，采用了写时复制机制（Copy-On-Write）

- 当父子进程任意之一要修改数据段、堆、栈时，进行复制操作，并且仅复制修改区域

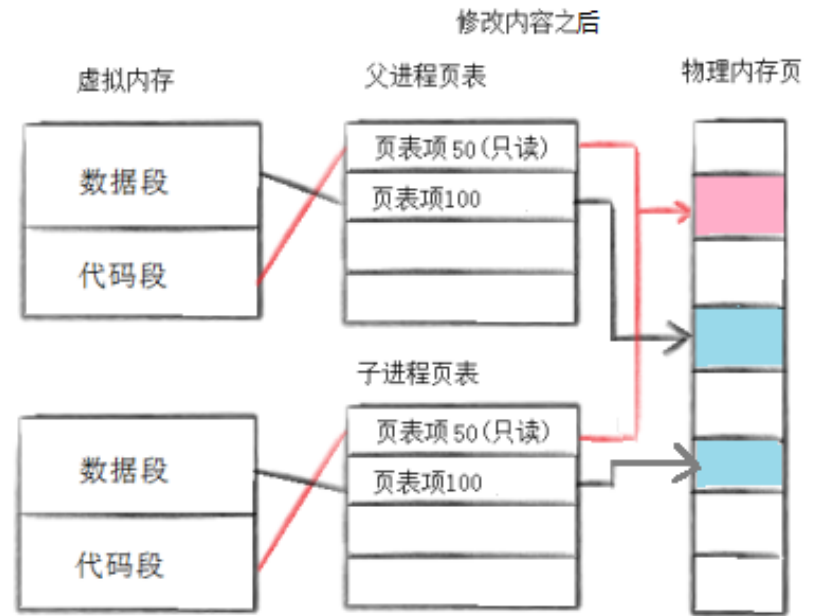


fork函数工作流程

1. 父子进程均无写操作时



2. 父进程或者子进程对数据有修改操作时，子进程或父进程将新产生一份进程的数据拷贝，然后再修改



<https://blog.csdn.net/zztong77>



fork函数执行后父子进程的主要异同

■ 父子进程相同

- 真实用户ID，真实组ID
- 有效用户ID，有效组ID
- 环境变量
- 堆
- 栈
- 打开的文件

■ 父子进程不同

- fork的返回值
 - 进程ID及父进程ID
- 子进程的
tms_utime,
tms_stime,
tms_cutime,
tms_ustime值被设置为 0



父子进程共享文件

▪ 子进程复制父进程的进程控制块

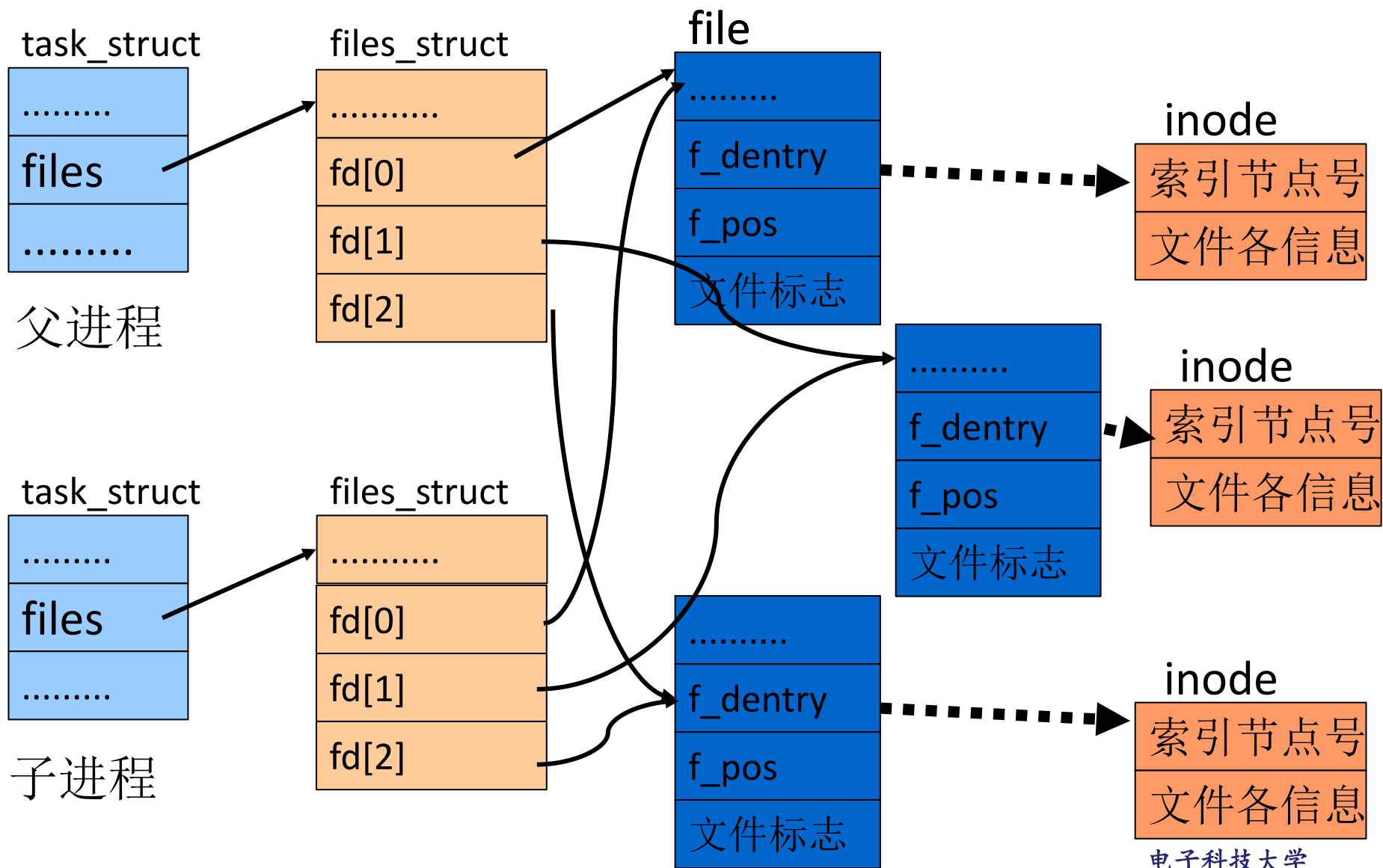
- 父进程的文件描述符表被子进程复制，父子进程的同一文件描述符指向同一个文件表
- 父子进程对同一文件访问基于相同的文件当前位置

▪ 父子进程对共享文件的常见处理方式：

- 父进程等待子进程完成。当子进程终止后，文件当前位置已经得到了相应的更新
- 父子进程各自执行不同的程序段，各自关闭不需要的文件



父子进程共享文件

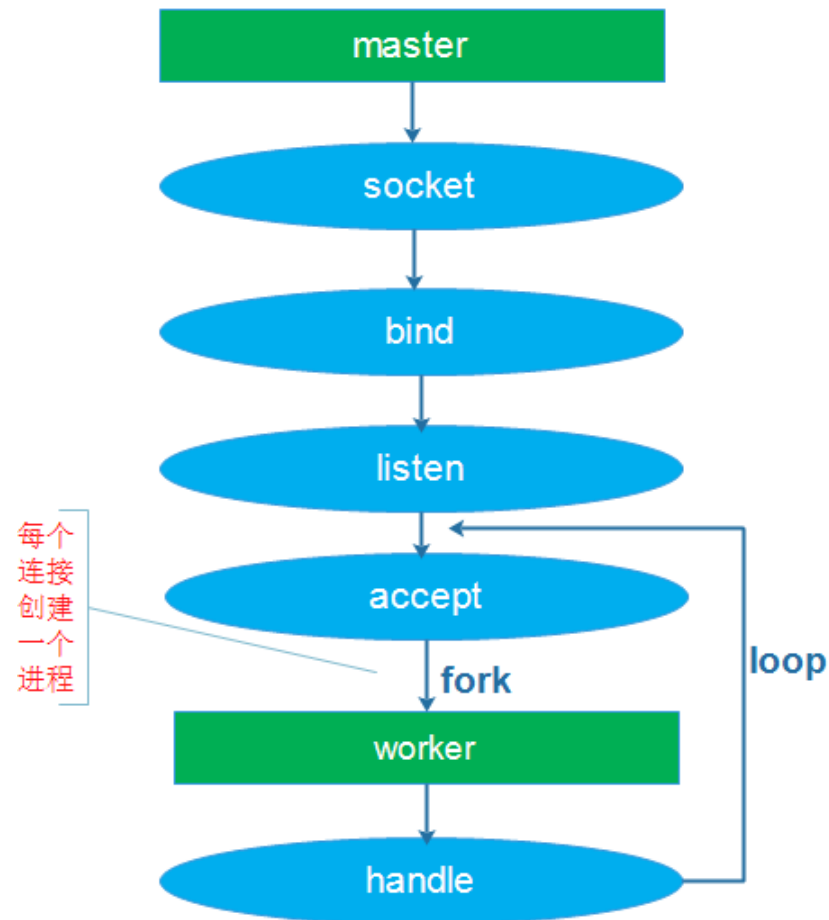




fork的用法

- 父进程希望复制自己（共享代码，复制数据空间），但父子进程执行相同代码中的不同分支

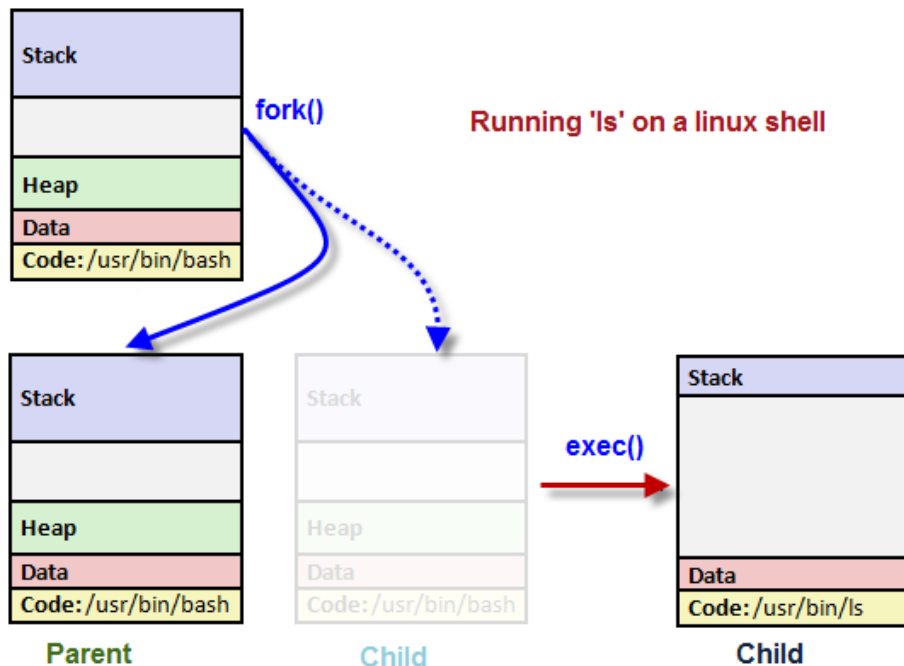
网络并发服务器中，父进程等待客户端的服务请求。当请求达到，父进程调用fork创建子进程处理该请求，而父进程继续等待下一个服务请求





fork的用法

- 父子进程执行不同的可执行文件（父子进程具有完全不同的代码段和数据空间）
 - 子进程从fork返回后，立即调用exec类函数执行另外一个可执行文件





vfork函数

- **vfork用于创建新进程，而该新进程的目的在于执行另外一个可执行文件**
 - 由于新程序将有自己的地址空间，因此子进程并不复制父进程的地址空间
 - 子进程在调用exec或exit之前，在父进程的地址空间中运行
 - vfork函数保证子进程先执行，在它调用exec或者exit之后，父进程才会继续被调度执行（父进程处于TASK_UNINTERRUPTIBLE状态）





主要内容

- Linux 进程控制块
- 进程的启动
- 进程的运行控制
- 进程的监测



exec系列函数

- 进程调用exec系列函数在进程中加载执行另外一个可执行文件
- exec系列函数替换了当前进程（执行该函数的进程）的正文段、数据段、堆和栈（来源于加载的可执行文件），但并不修改PCB！
- 执行exec系列函数后从加载可执行文件的main函数开始重新执行
- exec系列函数并不创建新进程，所以在调用exec系列函数后其进程ID并未改变，已经打开的文件描述符不变

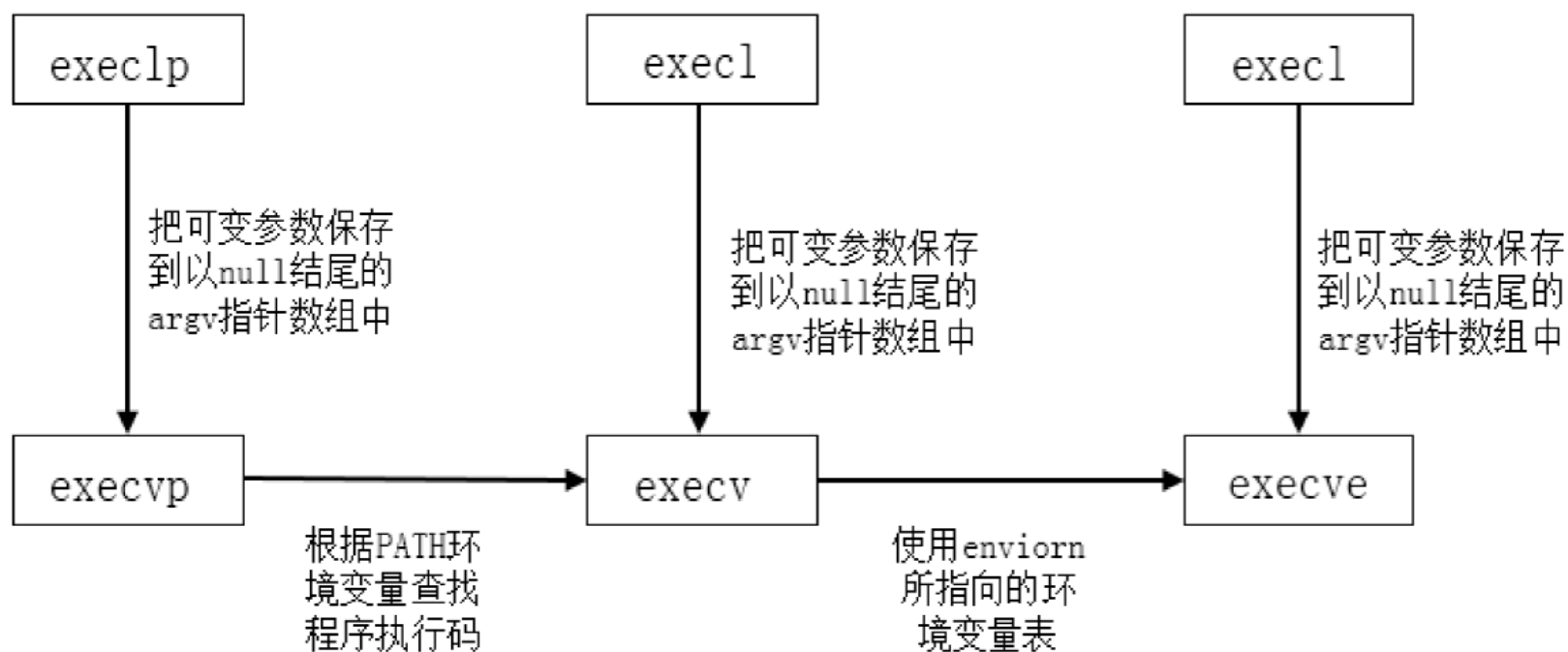


exec系列函数

- **exec****l** **exec****e** **exec****p** **exec****v** **exec****ve** **exec****vp**
- 六个函数开头均为exec，所以称为exec系列函数
 - **l**：表示list，每个命令行参数都说明为一个单独的参数
 - **v**：表示vector，命令行参数放在数组中
 - **e**：表示由函数调用者提供环境变量表
 - **p**：表示通过环境变量PATH来指定路径，查找可执行文件



exec系列函数调用关系





■ 函数原型

- 头文件: unistd.h
- `int execl(const char *pathname, const char *arg0, ..., NULL);`

■ 参数

- `pathname`: 要执行程序的绝对路径名
- 可变参数: 要执行程序的命令行参数, 以空指针结束

■ 返回值

- 出错返回-1
- 成功该函数不返回!



示例代码: execl

```
int main(void)
{
    printf("entering main process---\n");
    if(fork()==0){
        execl("/bin/ls","ls","-l",NULL);
        printf("exiting main process ----\n");
        return 0;
    }
```

```
[zxy@test unixenv_c]$ cc execl.c
[zxy@test unixenv_c]$ ./a.out
entering main process---
total 104
-rwxrwxr-x. 1 zxy zxy          5976 Jul 12 22:54 a.out
-rw-r--r--. 1 zxy zxy           527 Jul 12 15:48 atexit02.c
-rw-r--r--. 1 zxy zxy           426 Jul 12 15:59 atexit03.c
-rw-r--r--. 1 zxy zxy           287 Jul 12 15:44 atexit.c
-rw-r--r--. 1 zxy zxy           472 Jul 10 12:39 creathole.c
```



■ 函数原型

- 头文件: `unistd.h`
- `int execv(const char *pathname, char *const argv[]);`

■ 参数

- `pathname`: 要执行程序的绝对路径名
- `argv`: 数组指针维护的程序命令行参数列表, 该数组的最后一个成员必须为空指针

■ 返回值

- 出错返回-1
- 成功该函数不返回



示例代码：execv



```
int main(void)
```

```
{
```

```
    int ret;
```

```
    char *argv[] = {"ls", "-l", NULL};
```

```
    printf("entering main process---\n");
```

```
    if(fork()==0) {
```

```
        ret = execv("/bin/ls",argv);
```

```
        if(ret == -1) perror("execl error");
```

```
        printf("exiting main process ----\n");}
```

```
    return 0;
```

```
}
```

```
[zxy@test unixenv_c]$ cc execl.c
[zxy@test unixenv_c]$ ./a.out
entering main process---
total 104
-rwxrwxr-x. 1 zxy zxy          6231 Jul 12 23:09 a.out
-rw-r--r--. 1 zxy zxy          527 Jul 12 15:48 atexit02.c
-rw-r--r--. 1 zxy zxy          426 Jul 12 15:59 atexit03.c
-rw-r--r--. 1 zxy zxy          287 Jul 12 15:44 atexit.c
-rw-r--r--. 1 zxy zxy          472 Jul 10 12:39 creathole.c
```



■ 函数原型

- 头文件: unistd.h
- `int execl(const char *pathname, const char *arg0, ... NULL, char *const envp[]);`

■ 参数

- `pathname`: 要执行程序的绝对路径名
- 可变参数: 要执行程序的命令行参数, 以空指针结束
- `envp`:: 指向环境字符串数组的指针, 该数组的最后一个成员必须为空指针

■ 返回值

- 出错返回-1
- 成功该函数不返回



示例代码: execl

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char *envp[]={ "PATH=/tmp" , "USER=shan" ,NULL};
    if(fork()==0)
    {
        if(execl( "/bin/l" , "l" , "-l" ,NULL, envp)<0)
            perror("execl error!");
    }
    return 0;
}
```



■ **execve函数**

- `int execve(const char *pathname, char *const argv[], char *const envp[]);`

■ **execlp函数**

- `int execlp(const char *filename, const char *arg0, ..., NULL);`
- filename参数可以是相对路径（路径信息从环境变量PATH中获取）
- 例如默认环境变量中包含的
PATH=/bin:/usr/bin:/usr/local/bin/

■ **execvp函数**

- `int execvp(const char *filename, char *const argv[]);`



主要内容

- Linux进程控制块
- 进程的启动
- 进程的运行控制
- 进程的监测



进程启动与终止

- **进程启动：**子进程和父进程共享代码段，从fork函数执行之后的代码处开始执行；exec类函数会让进程从可执行文件的main函数开始重新执行
- **进程退出：**
 - 1. 正常终止：
 - 从main函数中执行return返回
 - 在任意代码中调用exit函数或_exit函数
 - 2. 异常终止：
 - 在任意代码中调用abort函数
 - 接收到终止信号



进程终止

(1) exit和return 的区别:

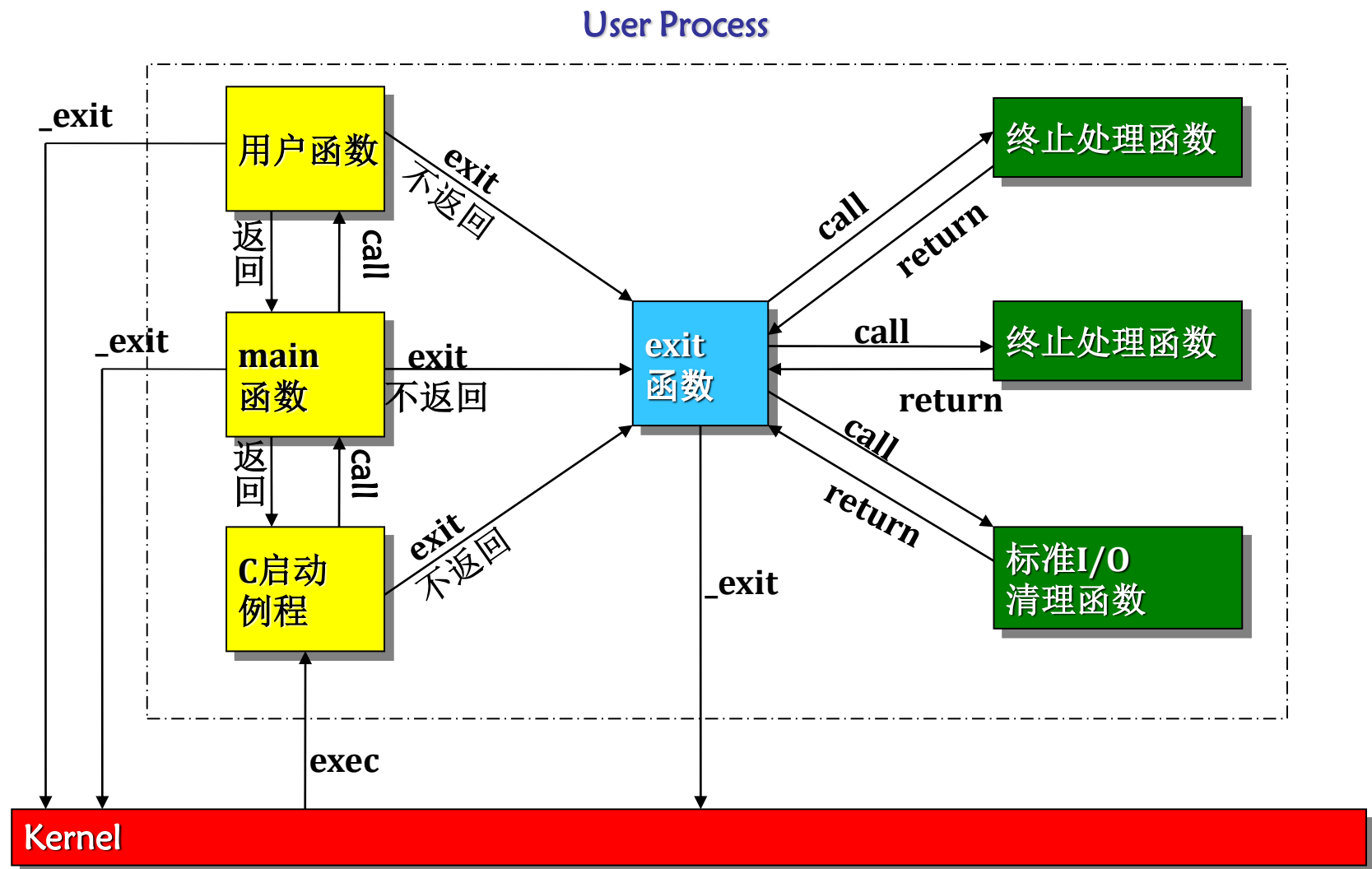
- **exit是一个函数，有参数。exit执行完后把控制权交给系统。**
- **return是函数执行完后的返回。return执行完后把控制权交给调用函数。**

(2) exit和abort的区别:

- **exit是正常终止进程，abort是异常终止。**



进程的启动与终止





终止进程函数

- 头文件`stdlib.h`，函数定义：`void exit(int status)`
- 头文件`unistd.h`，函数定义：`void _exit (int status)`
- 调用这两个函数均会正常地终止一个进程
- 调用`_exit` 函数将会立即返回内核
- 调用`exit` 函数：
 - 执行一些预先注册的终止处理函数
 - 执行文件I/O操作的善后工作，使得所有缓冲的输出数据被更新到相应的设备
 - 返回内核



■ 主动获取

- 调用wait或waitpid函数等待子进程状态信息改变，并获取其状态信息

■ 异步通知

- 当一个进程发生特定的状态变化（进程终止、暂停以及恢复）时，内核向其父进程发送SIGCHLD信号
- 父进程可以选择忽略该信号，也可以对信号进行处理（默认处理方式忽略该信号）



僵尸进程

- 进程在退出之前会释放进程用户空间的所有资源，但PCB等内核空间资源不会被释放
 - 当父进程调用wait或waitpid函数后，内核将根据情况关闭该子进程打开的所有文件，释放PCB（释放内核空间资源）
- 子进程先于父进程终止，但父进程尚未对其调用wait或waitpid函数的子进程（TASK_ZOMBIE状态），称为僵尸进程

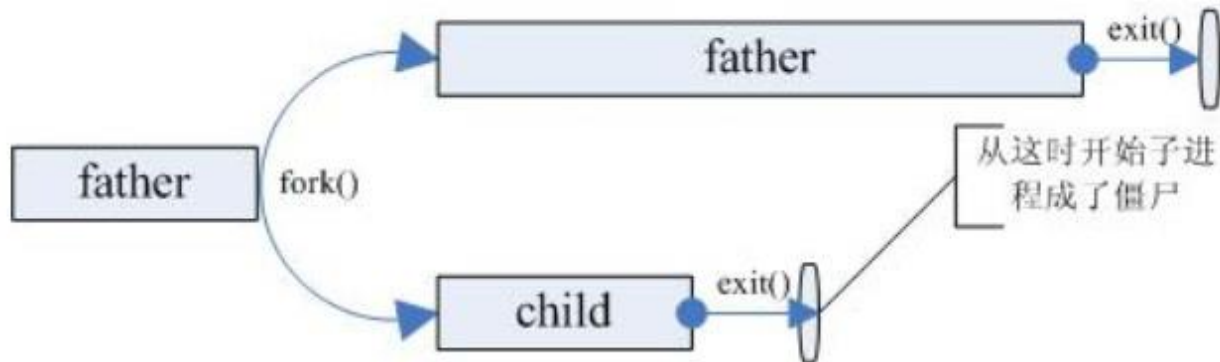


图2 僵尸进程



孤儿进程

- 如果父进程在子进程终止之前终止，则子进程的父进程将变为init进程，保证每个进程都有父进程，由init进程调用wait函数进行善后

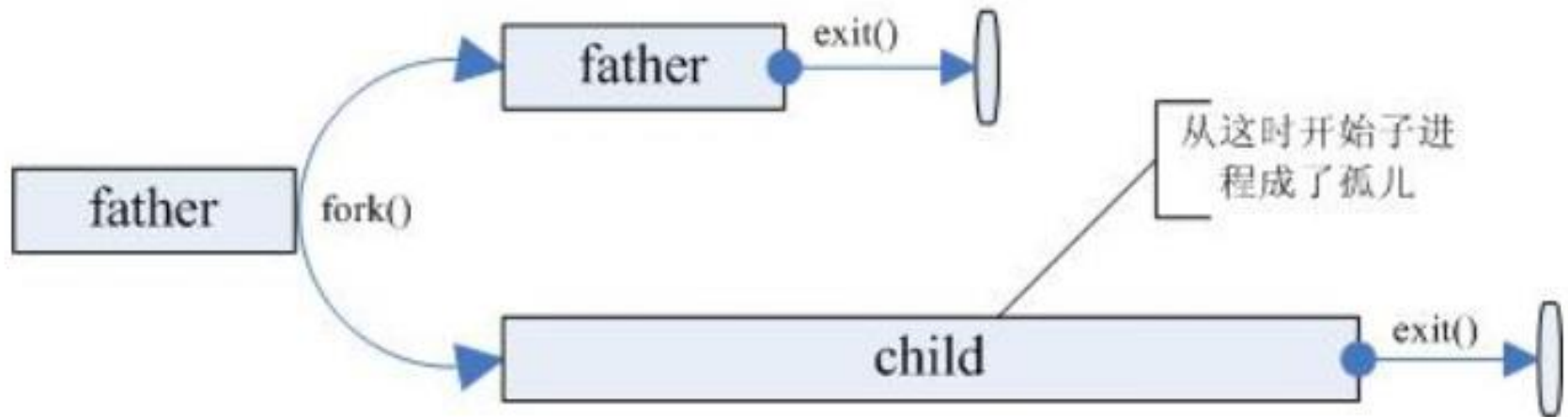


图1 孤儿进程



wait函数

- **功能：获取任意子进程的状态改变信息（如果是终止状态则对子进程进行善后处理）**
- **函数原型**
 - 头文件： `sys/wait.h`
 - `pid_t wait(int *statloc);`
- **参数与返回值**
 - `statloc`：用于存储子进程的状态改变信息
 - 返回值：若成功返回状态信息改变的子进程ID，出错返回-1



▪ 参数statloc

- statloc可以为空指针，此时父进程不需要具体了解子进程的状态变化，只是为了防止子进程成为僵尸进程，或者因为同步原因需等待子进程终止
 - 若statloc不是空指针，则内核将子进程状态改变信息存放在它指向的存储空间中
-
- **子进程状态改变信息**包含了多种类型的信息，可以通过系统提供的宏来快速解析子进程的状态



解析子进程状态改变信息的宏



宏	功能说明
WIFEXITED(statloc)	当子进程正常终止时该宏为真，对于这种情况可进一步执行WEXITSTATUS(statloc)，获取子进程传递给exit、_exit函数参数的低8位
WIFSIGNALED(statloc)	当子进程异常终止时该宏为真，对于这种情况可进一步执行WTERMSTG(statloc)，获取使子进程终止的信号编号
WIFSTOPPED(statloc)	当子进程暂停时该宏为真，对于这种情况可进一步执行WSTOPSIG(statloc)，获取使子进程暂停的信号编号
WIFCONTINUED(statloc)	若子进程在暂停后已经继续则该宏为真



- **调用wait函数之后，父进程可能出现的情况**
 - 如果所有子进程都还在运行，则父进程被阻塞（TASK_INTERRUPTIBLE状态），直到有一个子进程终止或暂停，wait函数才返回
 - 如果已经有子进程进入终止或暂停状态，则wait函数会立即返回
 - 若进程没有任何子进程，则立即出错返回-1



等待特定子进程状态改变

- 如果一个进程有几个子进程，那么只要有一个子进程状态改变，wait函数就返回
- 如何才能使用wait函数等待某个特定子进程的状态改变？
 1. 调用wait，然后将其返回的进程ID和所期望的子进程ID进行比较
 2. 如果ID不一致，则保存该ID，并循环调用wait函数，直到等到所期望的子进程ID为止



waitpid函数

- **功能：等待某个特定子进程状态改变**
- **函数原型**

头文件：sys/wait.h

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

- **返回值**

成功返回终止子进程ID，失败返回-1



waitpid函数

■ 参数

■ pid

- $\text{pid} == -1$: 等待任意子进程状态改变 (同wait)
- $\text{pid} > 0$: 等待进程ID为pid的子进程状态改变
- $\text{pid} == 0$: 等待其组ID等于调用进程组ID的任意子进程
- $\text{pid} < -1$: 等待其组ID等于pid绝对值的任意子进程

■ statloc: 用于存储子进程的状态改变信息

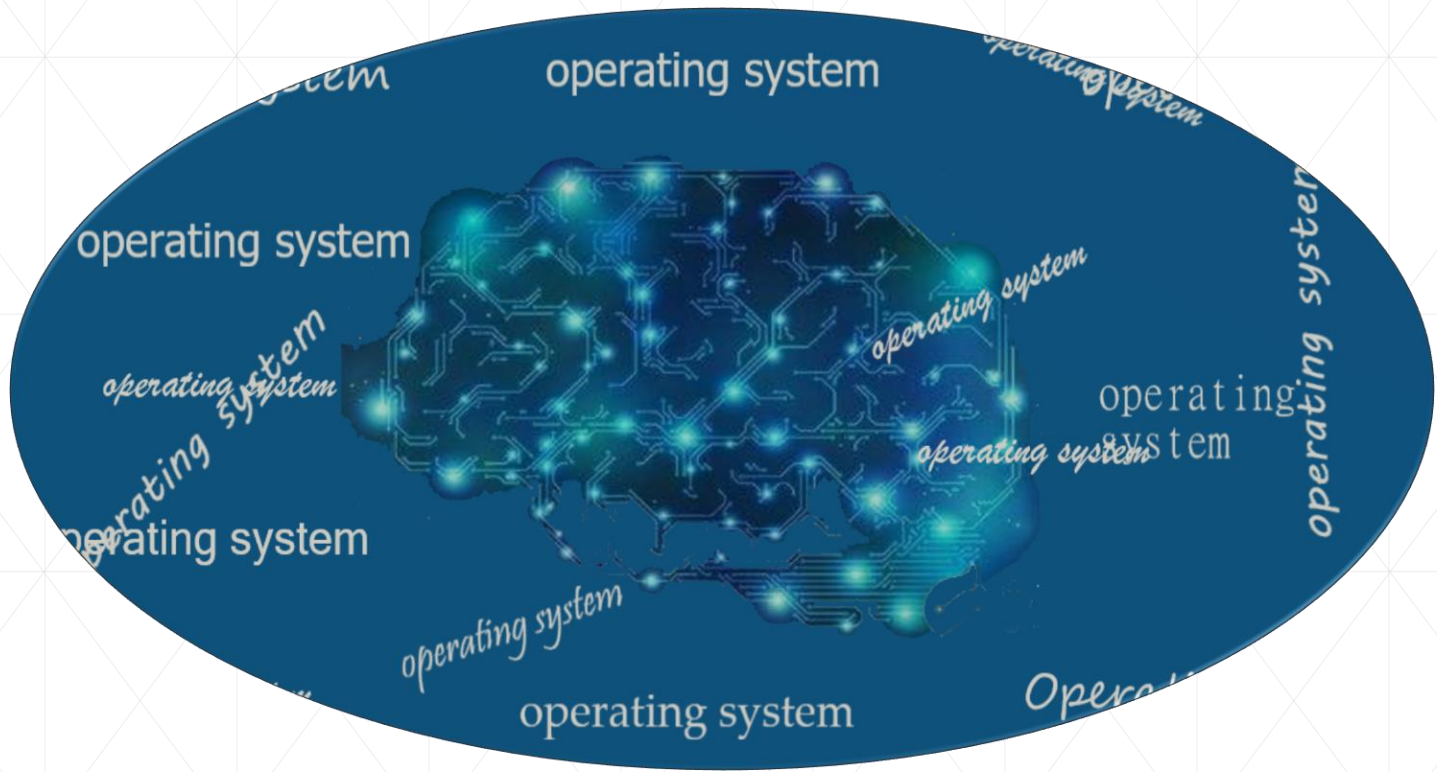
■ options: 可以为0, 也可以是以下常量或常量的或

- WNOHANG: 如果没有任何已经终止的子进程则马上返回, 函数不阻塞, 此时返回值为0
- WUNTRACED: 用于跟踪调试



waitpid的特有功能

- **waitpid可等待一个特定的进程的状态改变信息**
- **waitpid可以实现非阻塞的等待操作，有时希望取得子进程的状态改变信息，但不希望阻塞父进程等待子进程状态改变**
- **waitpid支持作业控制（进程组控制）**



感谢观看!

授课教师:

电子邮箱: