



第5章 进程与内存管理实例

5.1 Linux简介

5.2 Linux进程管理

5.3 Linux内存管理

5.4 Windows 2000/XP的进程与内存管理

5.5 小结



5.1 Linux 简介

1987年，MINIX操作系统：

Andrew S. Tanenbaum设计，类Unix OS，用于教学

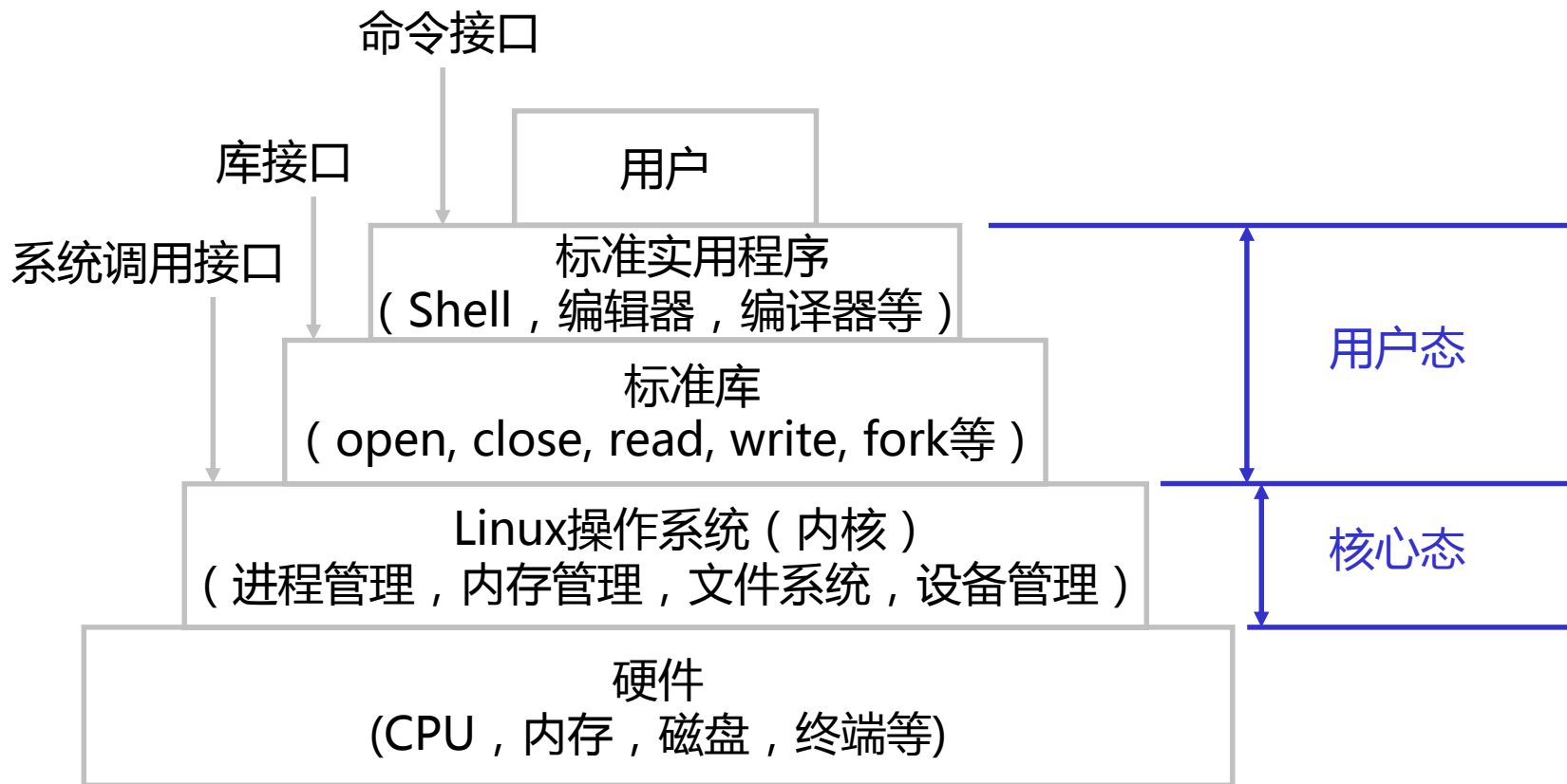
1991年，Linux操作系统

芬兰赫尔辛基大学生Linus Torvalds开发

- ✓ 与Unix兼容：具有全部Unix特征，遵从POSIX标准
- ✓ 自由软件，源代码公开
- ✓ 多用户、多任务的32位操作系统

5.1 Linux 简介

一、Linux的层次结构





Linux的层次结构

1. 进程执行的2个级别(模式)

(1) 用户态

用户态下的进程只能存取自己的指令和数据，不能执行特权指令

(2) 核心态

核心态下的进程可以访问OS核心和用户进程的地址空间

说明：

- ✓ 虽然系统在执行时处于两种状态之一，但核心是为用户进程运行的；
- ✓ **核心不是与用户进程平行运行的孤立的进程集合，而是每个用户进程的一部分。**
- ✓ **用户进程通过系统调用由用户态切换到核心态。**



Linux的层次结构

2. Linux的3个接口

(1) 系统调用接口

用户在程序中（一般用汇编语言）调用OS提供的一些功能

(2) 库接口

以高级语言（C语言）标准库函数的方式，为程序员调用OS提供的功能

(3) 命令接口

在操作系统与计算机用户之间提供的双向通信机制

接口包括：一组联机命令，终端处理程序，命令解释程序等。

实现方式：

- ✓ 命令行方式：要求用户记忆命令格式。
- ✓ 图形用户接口（GUI）方式：用户可利用鼠标对屏幕上的图标进行操作，完成与操作系统的交互，从而减少记忆内容，方便用户使用。



Linux的层次结构

3. 系统调用

(1) 什么是系统调用？

用户在程序中调用操作系统所提供的一些功能；

系统调用是用户请求OS服务的途径。

- ✓ 系统调用本身也是由若干指令构成的过程（函数），但与一般过程的区别在于：系统调用运行在核心态，一般过程运行在用户态
- ✓ 系统调用发生在用户进程通过特殊函数（如open）请求OS内核提供服务的时候
- ✓ 系统调用负责对内核所管理资源的访问
- ✓ 每个操作系统都提供许多种系统调用



系统调用

(2) 系统调用的方式

为了保证OS不被用户程序破坏，不允许用户程序访问OS的系统程序和数据。

那么，**如何得到系统服务呢？**

通过执行一条**陷入指令**（或称访管指令），**由用户态切换到核心态**，转入执行相应的处理程序（陷入处理程序）。

说明:

- ✓ 系统调用是一个低级过程，只能由汇编语言直接访问；
- ✓ 系统调用是操作系统提供给程序设计人员的唯一接口；
- ✓ C语言为每个系统调用提供对应的标准库函数，应用程序可以通过C语言库函数来使用系统调用。



系统调用

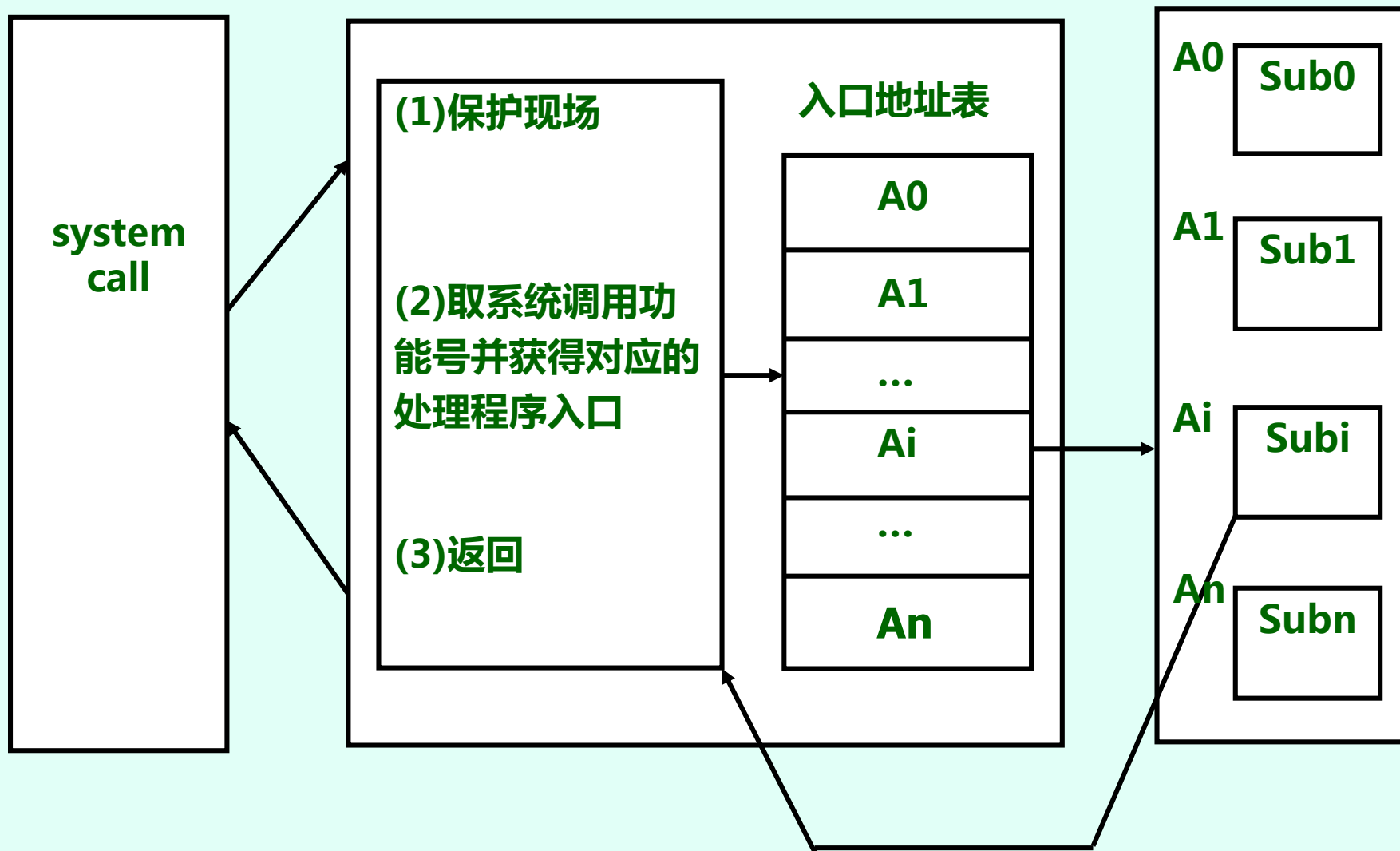
(3) 系统调用的处理过程

- ✓ 每个系统调用对应一个事先给定的功能号，如0、1、2、3等
- ✓ 系统为实现系统调用功能的子程序（过程）设置入口地址表
- ✓ 每个入口地址与相应的陷入处理程序对应
- ✓ 进入系统调用处理前，保护处理机现场
- ✓ 在系统调用返回后，要恢复处理机现场
- ✓ 在系统调用处理结束后，用户程序可以利用系统调用的返回结果继续执行

用户程序

陷入处理机构

系统子程序



系统调用的处理过程



系统调用

怎样实现用户程序和系统程序间的参数传递？

常用的3种实现方法：

①由陷入指令自带参数

陷入指令的长度是有限的，且还要携带系统调用功能号，只能自带有限的参数

②通过有关通用寄存器来传递参数

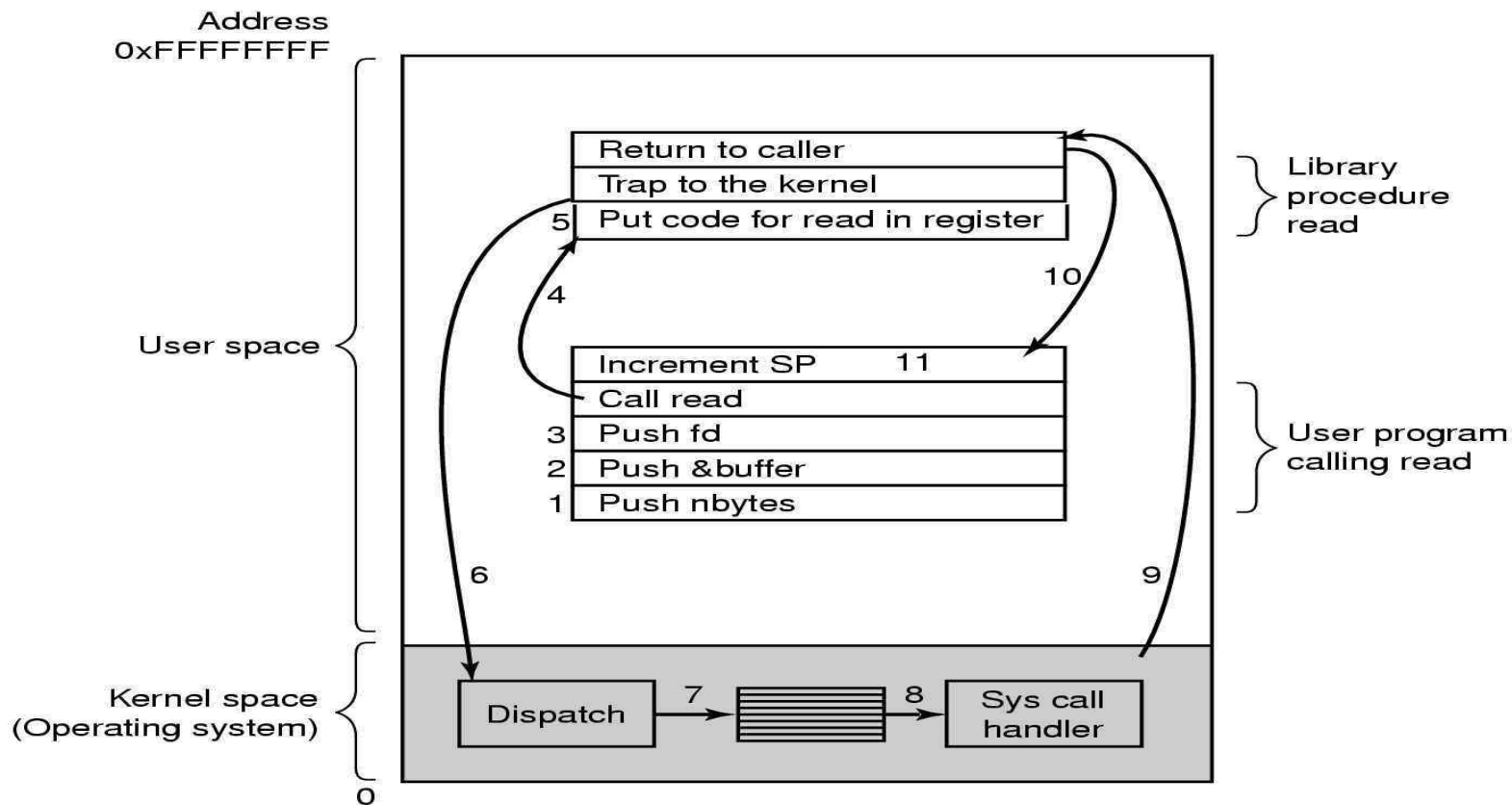
这些寄存器应是系统程序 and 用户程序都能访问的，

由于寄存器长度和个数有限，无法传递较多的数据

③大多在内存中开辟专用的区域（如堆栈区）来传递参数

系统调用

read (fd, buffer, nbytes)





系统调用

(4) 系统调用的分类

系统调用分为两大类：系统自身所需要的；作为服务提供给用户的

- ✓ 进程管理系统调用
- ✓ 内存管理系统调用
- ✓ 文件和目录管理系统调用
- ✓ I/O设备管理系统调用



系统调用

(5) 系统调用与一般过程调用的比较

相同点：

- ✓ 改变指令执行流程
- ✓ 是为实现某个特定功能而设计的，可多次调用
- ✓ 执行完成后返回



系统调用与一般过程调用的比较

系统调用与一般过程调用的不同：

① 运行在不同的系统状态

一般的用户过程运行在用户态

系统调用运行在核心态

② 返回问题

在抢占式调度的系统中，系统调用在返回时，OS将对所有就绪进程进行优先级分析。如果调用进程仍有最高优先级，则返回到调用进程执行，否则，引起重新调度，让优先级最高的进程执行。此时，系统把调用进程放入就绪队列。



系统调用与一般过程调用的不同

③ 嵌套或递归调用

对于系统调用，一般不允许在同一个进程中发生嵌套或递归（不同进程可以重入同一个系统调用）

④ 进入和返回方式不同

利用int或trap指令进行系统调用；

利用call指令进入普通的过程调用；

一般过程的返回用ret指令

系统调用的返回用iret指令

核心态与用户态的转换由系统在int指令与iret指令内部自动完成，

没有用一条单独的专门指令，好处在于：有效地防止在核心态下执行用户程序



系统调用与一般过程调用的不同

- CALL指令的内部实现过程
 - 返回地址压栈（即该CALL指令下一条指令的地址）
 - 将该CALL指令中所含的地址（即被调用代码所在地址）送入PC
- RET指令的内部实现过程
 - 从栈顶弹出返回地址送入程序计数器PC
- INT指令和IRET指令的执行过程中
 - 要处理程序状态字PSW
 - INT指令中要保存用户程序的老PSW
 - IRET指令中要在返回用户程序前恢复用户程序的老PSW



5.1 Linux 简介

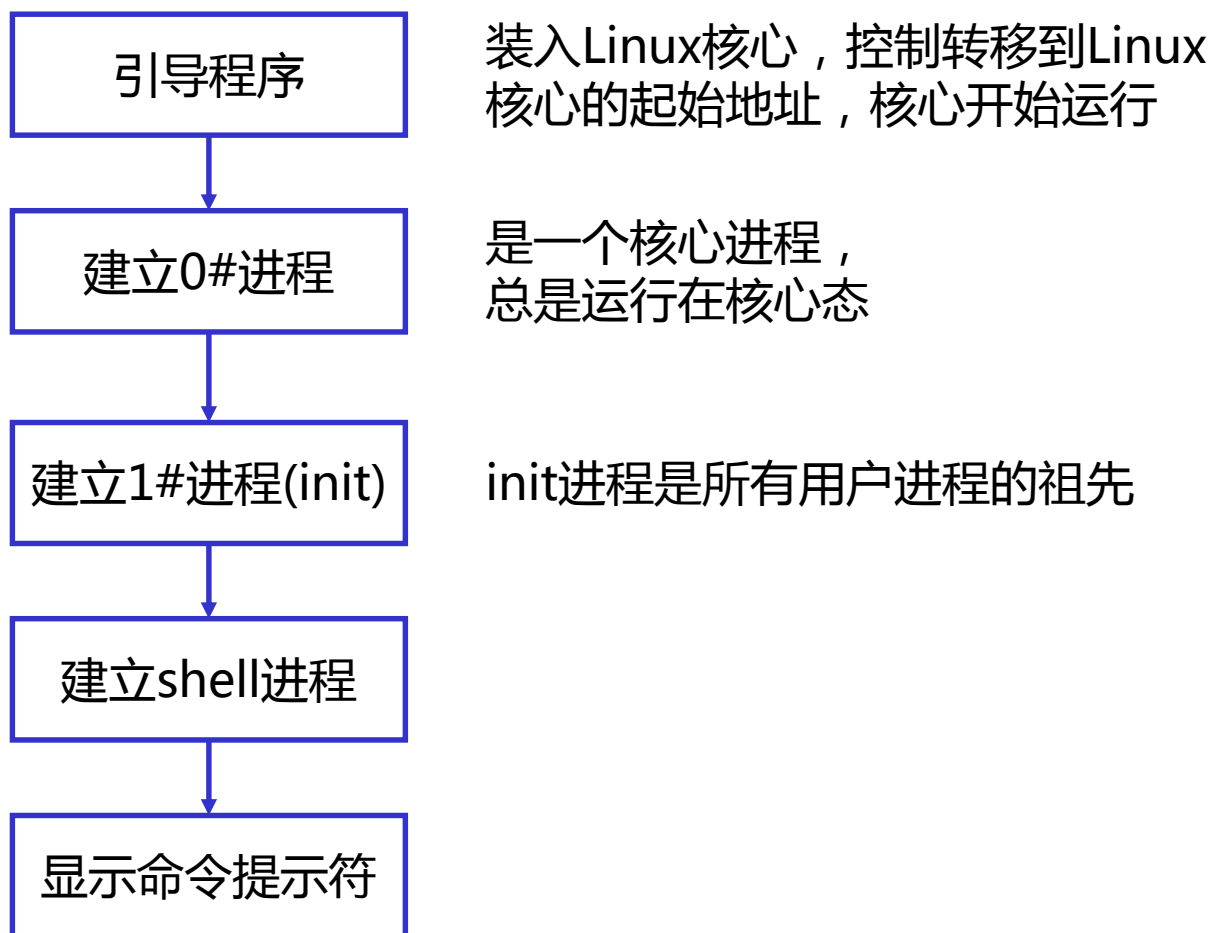
二、Shell

Linux的命令解释器称为Shell

Shell常常被普通用户看作是“Linux”，而实际上它与操作系统本身毫不相干，可以很容易换掉。

Shell

1. Shell进程





Shell

2. Shell解释命令的方式

当用户输入一行命令后，shell抽出第1个词，作为要运行程序的名字，将后面的参数以字符串的形式传递给被调用程序，同时建立一个执行该命令的用户进程。

shell等待直到用户进程执行结束后，shell恢复运行。

3. Shell脚本

用**shell语言**（看似C语言，具有if、for、while等结构），写一个shell脚本文件，可执行完成复杂任务。

一个简单的shell脚本由普通的shell命令组成，就像在shell命令行中交互地输入这些命令一样。用户可以在shell脚本中定义自己的命令。脚本中可以包括注释行。



5.2 Linux 进 程 管 理

一、进程的结构task_struct

每个进程对应1个task_struct结构，称为**进程描述符**相当于PCB。

主要包含下列信息：

- ✓ 进程标识符pid
- ✓ 进程状态
- ✓ 调度信息：优先数，消耗的CPU时间
- ✓ 信号及其处理方式
- ✓ 文件描述符表：记录该进程打开的所有文件
- ✓ 进程在内存的位置（页表指针）



5.2 Linux 进 程 管 理

二、进程的状态及其转换

5种状态：

(1) TASK_RUNNING：运行状态

进程正在运行或就绪。表示进程具备运行的资格，正在运行或等待被调度执行。进程控制块中有一个run_list成员，所有处于TASK_RUNNING状态的进程都通过该成员链在一起，称之为可运行队列。

(2) TASK_INTERRUPTIBLE：可中断睡眠（阻塞）

可被软中断信号唤醒

(3) TASK_UNINTERRUPTIBLE：不可中断睡眠（阻塞）

不可以被软中断信号唤醒

以上两种状态均表示进程处于阻塞状态



Linux进程状态

(4) TASK_STOPPED : 暂停

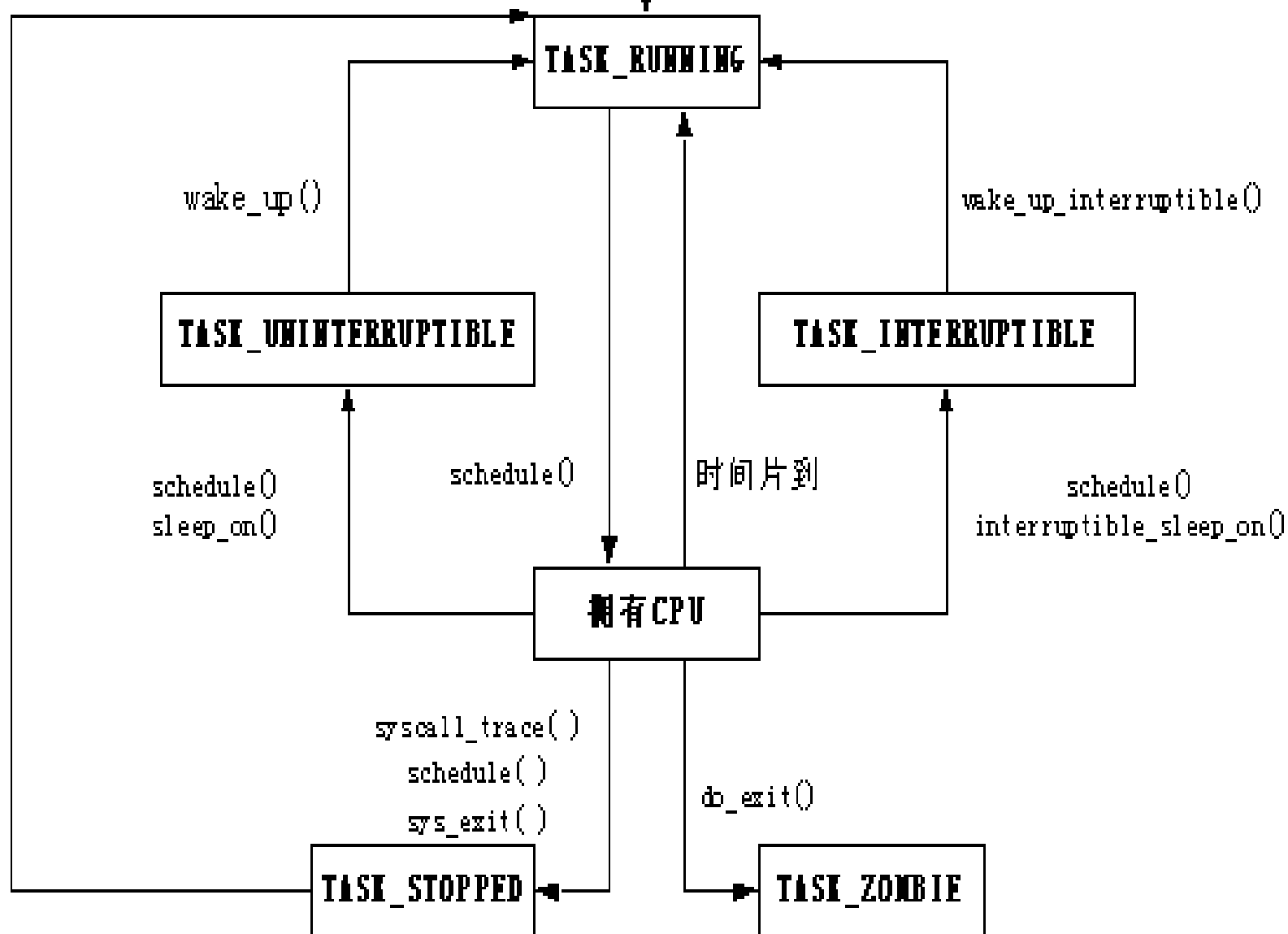
进程处于暂停状态，主要用于调试目的。如正在运行的进程收到信号SIGSTOP、SIGSTP、SIGTTIN、SIGTTOU后进入这个状态。

(5) TASK_ZOMBIE : 僵死

表示进程已经结束运行并释放了大部分占用的资源，但task_struct结构还未被释放。

收到SIG_KILL或
SIG_CONT后, 执行
wake_up()

do_fork()





5.2 Linux 进 程 管 理

三、进程控制

1. 进程创建fork

```
pid = fork()
```

创建子进程。子进程是父进程的一个逻辑副本。

父进程返回子进程id，子进程返回0。



fork

algorithm fork()

in : 无

out : 父进程返回子进程的pid , 子进程返回0

{

为新进程分配一个task_struct结构和唯一的pid ;

复制父进程task_struct的内容到子进程中 , 并重新设置与父进程不同的数据成员 ;

父进程地址空间的逻辑副本复制到子进程 ;

if (执行的是父进程) {

 将子进程的状态置为TASK_RUNNING ;

 return 子进程的pid ;

}

else { //正在执行的子进程 , 被调度后才会执行

 return 0 ;

}

}



fork

说明：

- (1) 进程的pid是一个整数，在0到最大值之间。
- (2) 复制逻辑副本的意思是：Linux使用写时复制（ copy on write ）机制，子进程暂时共享父进程的有关数据结构，复制延迟到子进程作修改时才进行。



进程控制

2. 执行一个文件的调用exec

由指定的可执行文件覆盖进程的地址空间。

具有多种形式：execve，execlp，execvp等。



进程控制

`execve(char *filename, char *argv[], char *envp[])`

执行文件filename，覆盖进程的地址空间

C语言主函数的形式：

`main(int argc, char **argv, char **envp)`

argc：参数个数，包括程序名

argv：或写作char *argv[]

argv[0]~argv[argc-1]指向每个参数串

envp：环境变量串，以空串结束



进程控制

【例】一个简化的shell。

```
while (1) {  
    type_prompt(); //在屏幕上显示提示符  
    read_command(command, params); //读输入的命令  
    pid = fork();  
    if (pid < 0) {  
        printf( "Unable to fork.\n" ); //出错  
        continue;  
    }  
    if (pid != 0) { //子Shell创建完毕  
        wait(0); //父Shell代码，等待子Shell结束  
    }  
    else {  
        execve(command, params, 0); //子Shell代码  
    }  
}
```



5.2 Linux 进 程 管 理

四、进程调度

1. 调度原理

3种调度策略：

① 动态优先级：普通进程

② 先来先服务：实时进程

③ 时间片轮转：实时进程

✓ 进程可以设置调度策略（保存在进程描述符中）

✓ 优先数越大，优先级越低

✓ 类似于多级反馈队列



进程调度

2. 调度的时机

(1) 进程主动放弃CPU —— 直接调用`schedule()`

✓ 隐式主动放弃CPU

如执行系统调用`read`阻塞而转入`schedule()`

✓ 显式主动放弃CPU

如执行系统调用`sched_yield()`、`nanosleep()`等

(2) 进程从核心态返回到用户态 (从中断处理、系统调用、异常处理) 之前 , 核心检查当前进程的调度标志`need_resched` , 若为1 , 则调用`schedule()`。



进 程 调 度

3. 调度标志`need_resched`置1的时机

- ✓ 当前进程的时间片用完，时钟中断处理程序设置该进程的`need_resched`标志。
- ✓ 刚唤醒的进程优先级高于当前进程，将当前进程的`need_resched`标志置1。



5.2 Linux 进 程 管 理

五、进程同步与通信

1. 信号 (signal)

信号是软件层次上对中断的一种模拟，又称为软中断。

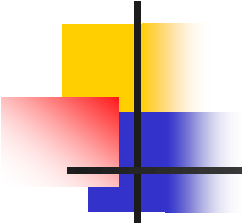
表示进程之间传送预先约定的信息类型，用于通知某进程发生了一个事件。

例如：

2 SIGINT 中断 (用户键入ctrl-C)

9 SIGKILL 杀死 (终止) 进程

11 SIGSEGV 越界访问内存



信号

(1) 信号的发送

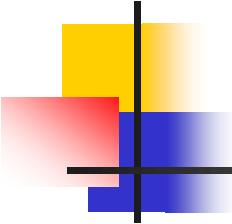
`kill(int pid, int sig)`

`pid` : 发给哪个进程

`= 0` : 信号给同组所有进程

`> 0` : 信号发给`pid`所指的进程

`sig` : 发送的信号



信号

(2) 信号处理方式的预置

`signal(sig, func);`

`sig` : 信号

`func` : 信号处理函数的指针

0 : SIG_DFL , 收到信号后终止

1 : SIG_IGN , 忽略

其他 : 处理函数的指针



信号处理方式预置例

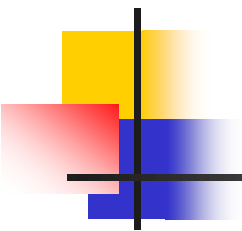
```
void SetupSignalHandlers(void)
{
    signal( SIGHUP, SIG_IGN );           /* 1 */
    signal( SIGINT, SIG_IGN );           /* 2 */

    /* . . . . . */

    signal( SIGSEGV, UnexpectedSignal ); /* 11 */
    signal( SIGSYS, UnexpectedSignal );  /* 12 */
    signal( SIGPIPE, SIG_IGN );          /* 13 broken pipe */
    signal( SIGALRM, SIG_IGN );          /* 14 */
    signal( SIGTERM, Terminate );        /* 15 */
}

void UnexpectedSignal(int sig)
{
    ErrorMsg("Unexpected signal %d caught. Process aborted.", sig);
    ClearUp();
    abort();
}

void Terminate( int sig )
{
    ClearUp();
    exit(0);
}
```



信号

(3) 信号和中断的区别

- ✓ 中断有优先级；信号没有优先级（所有信号是平等的）
- ✓ 信号处理程序在用户态运行；而中断处理程序运行在核心态
- ✓ 中断响应是及时的；信号响应通常有较大的时间延迟，必须等到接收进程执行时才生效



进程同步与通信

2. 管道 (pipe)

连接一个写进程和读进程的pipe文件

- ✓ 无名管道

一个临时文件，用于进程及其子孙进程之间的通信

- ✓ 有名管道

具有路径名的文件，可用于任何进程之间的通信



进程同步与通信

3. 消息队列 (message queue)

- (1) 每个消息队列都有1个key(整数) , 由应用程序指定
- (2) 在进程内 , 创建/打开的消息队列都有1个消息队列描述符Qid , 其作用与文件描述符类似
- (3) 每个进程可以有多个消息队列
- (4) 使用方法

`Qid = msgget(key, msgflag); //创建/打开消息队列`

`msgsnd(Qid, buf, nsize, ...); //发送消息`

`msgrcv(Qid, buf, nsize, ...); //接收消息`

`msgctl(Qid, cmd, buf); //读取消息队列的状态 , 或删除消息队列`



进程同步与通信

4. 共享内存 (shared memory)

与消息队列类似，以key唯一标识。

```
shmid = shmget(key, size, shmflag); //创建/打开共享内存
```

```
char *shmaddr = shmat(shmid, ...); //将共享内存附接到进程的虚地址空间
```

```
shmdt(shmaddr); //把共享内存与进程断开
```

```
shmctl(shmid, cmd, buf); //读取共享内存的状态，或删除共享内存
```




共享内存例

```
/*
** Attach System BB
*/
if ((SystemBBid = shmget(SystemBBKey, sizeof( BBOARD ), 0600 ))<0)
{
    ErrorMsg("%s:Cannot open BB: shmget():%m", ProgName);
    return -1;
}

if ((pBBoard = (BBOARD *)shmat( SystemBBid, (char *)0, 0))== (BBOARD *)0)
{
    ErrorMsg("%s:Cannot attach BB: shmat():%m",ProgName);
    return -1;
}

pServerDesc = &(pBBoard->bb_servertab[ServerIndex]);
```



进程同步与通信

5. 信号量 (semaphore)

以key唯一标识。

`semid = semget(key, nsems, semflag);` //创建/打开信号量集

nsem : 信号量个数

`semop(semid, sops, nsops);` //信号量的P、V操作

sops : 指向信号量集操作结构的数组指针，操作可以是：

正数：相当于V操作

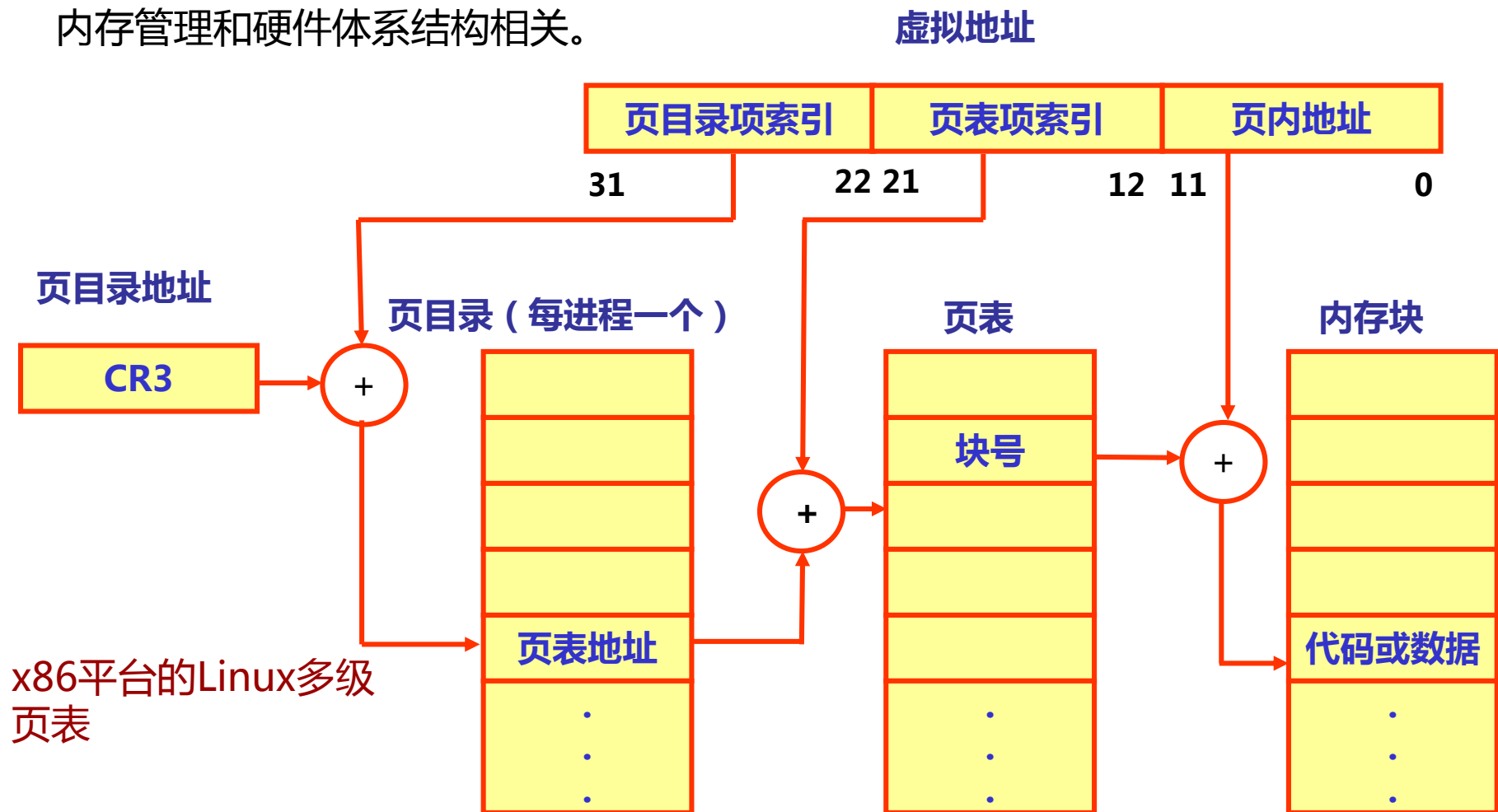
负数：相当于P操作

`semctl(semid, ...);` //可用来设置信号量的初始值

5.3 Linux 内存管理

一、Linux的多级页表

内存管理和硬件体系结构相关。





Linux的多级页表

页表项的主要内容：

- ✓ 有效位：该页是否在内存中
- ✓ 访问位：该页最近是否被访问过
- ✓ 修改位：该页是否被修改过
- ✓ 读写位：只读/可写



5.3 Linux 内存管理

二、Linux的请求调页技术

Linux采用请求调页策略进行内存管理。

当页表有效位 = 0时，产生缺页异常，进行缺页处理。

(1) 交换缓冲区

缺页若在内存的交换缓冲区中，则不必读交换区。



Linux的 请求调页 技术

(2) 页的换出

页的换出时机：

- ① 分配内存时发现空闲内存低于设定的极限值
- ② 使用核心线程kswapd**周期性地换出内存页**

Linux将未使用的物理块作为系统的缓冲区和块设备的缓冲区。如果内存中无足够的物理块存放调入页，则首先减少各种缓冲区的大小来满足进程的需要。如果仍然不够，则淘汰相应的物理块。

Linux的页置换算法是近似的LRU算法，是NRU算法。



5.4 Windows 2000/xp的进程与内存管理

Windows 2000/2003/xp/... :

用C/C++和少量汇编语言实现

基于客户/服务器模型、对象模型、对称多处理模型

网络功能是OS的一部分

可移植、运行于不同CPU和硬件平台的32位OS



5.4 Windows 2000/xp的进程与内存管理

一、进程与线程

进程是作为对象实现的。

进程的组成部分：

- ✓ 唯一的Id
- ✓ 一个可执行程序（代码、数据）
- ✓ 私有的虚拟地址空间
- ✓ 拥有的系统资源
- ✓ 至少1个执行线程

资源分配的对象是进程



Windows 2000/xp的进程与线程

多个线程共享其进程的虚拟地址空间

线程的组成部分：

- ✓ 唯一的Id
- ✓ 描述处理机状态的一组寄存器值
- ✓ 用户栈和核心栈

调度的基本单位是线程



Windows 2000/xp的进程与线程

二、与进程和线程有关的API

Win32子系统的进程（线程）控制系统调用主要有：

CreateProcess()

ExitProcess()

TerminateProcess()

CreateThread()

ExitThread()

SuspendThread()

ResumeThread()

等等。



与进程和线程有关的API

CreateProcess()：用于创建新进程及其主线程，以执行指定的程序。

ExitProcess()或TerminateProcess()：进程包含的线程全部终止。

ExitProcess()终止一个进程和它的所有线程；它的终止操作是完整的，包括关闭所有对象句柄、它的所有线程等。

TerminateProcess()终止指定的进程和它的所有线程；它的终止操作是不完整的（如：不向相关DLL通报关闭情况），通常只用于异常情况下对进程的终止。



与进程和线程有关的API

CreateThread()函数在调用进程的地址空间上创建一个线程，以执行指定的函数；返回值为所创建线程的句柄。

ExitThread()函数用于结束本线程。

SuspendThread()函数用于挂起指定的线程。

ResumeThread()函数递减指定线程的挂起计数，挂起计数为0时，线程恢复执行。



5.4 Windows 2000/xp的进程与内存管理

三、线程调度

1. 线程的7种状态

- (1) 就绪 (Ready) : 具备执行条件, 等待被挑选进入 “备用” 状态
- (2) 备用 (Standby) : 准备在特定处理机上执行。每个处理机上只能有一个线程处于备用状态
- (3) 运行 (Running) : 完成描述符表切换, 线程进入运行状态, 直到内核抢占、时间片用完、线程终止或进入等待状态
- (4) 等待 (Waiting) : 线程等待对象句柄, 以同步它的执行。等待结束时, 根据优先级进入运行或就绪状态。



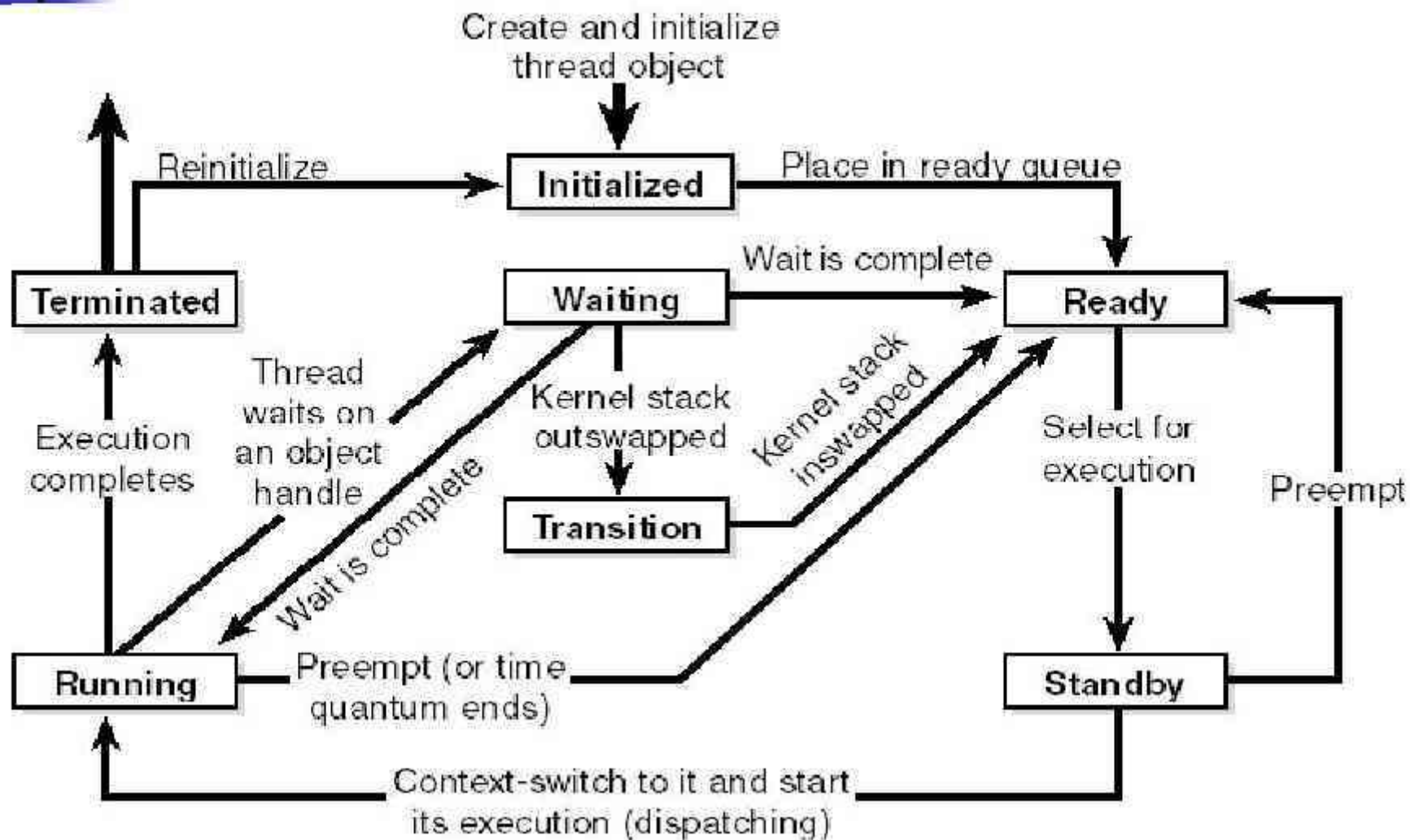
线程状态

(5) 转换 (Transition) : 与就绪状态类似, 但线程的内核栈处于外存。当线程等待的事件出现而它的内核栈处于外存时, 进入转换状态。当其内核栈调入内存, 线程进入就绪状态

(6) 终止 (Terminated) : 线程执行完就进入终止状态。如果执行体有一指向线程对象的指针, 可将线程对象重新初始化, 并再次使用

(7) 初始化 (Initialized) : 线程正在创建过程中

线程状态





线程调度

2. 调度算法

调度单位是线程而不是进程，采用抢占式动态优先级多级队列，依据优先级和分配时间配额来调度。

(1) 基本思想

- ✓ 每个优先级的就绪线程排成一个先进先出队列
- ✓ 线程的基本优先级与其进程的基本优先级有关
- ✓ 当一个线程变成就绪时，它可能立即运行或排到相应优先级队列的尾部
- ✓ 调度程序总是挑选优先级最高的就绪线程运行
- ✓ 在同一优先级的各线程按时间片轮转算法进行调度
- ✓ 线程时间配额用完而被抢占，优先级降低一级，进入下一级队列，直到其基本优先级
- ✓ 在多处理机系统中多个线程可并行运行



线程调度

(2) 线程时间配额

- ✓ 时间配额是一个线程从进入运行状态到Windows 检查是否有其他优先级相同的线程需要开始运行之间的时间总和
- ✓ 一个线程用完了自己的时间配额时，如果没有其它相同优先级线程，Windows将重新给该线程分配一个新的时间配额，并继续运行
- ✓ 时间配额不是一个时间长度值，而是一个称为配额单位(quantum unit)的整数



线程调度

(3) 线程优先级

Windows 2000/XP内部使用32个线程优先级，范围从0到31，它们被分成以下三类：

- ✓ 16个实时线程优先级 (16 - 31)

Windows 并不是通常意义上的实时，并不提供实时操作系统服务

- ✓ 15个可变线程优先级 (1 - 15)
- ✓ 1个系统线程优先级 (0)，仅用于对系统中空闲物理块进行清零的零页线程



线程调度

(4) 线程优先级提升

- ✓ 在下列5种情况下，Windows 会提升线程的当前优先级：
 - I/O操作完成
 - 信号量或事件等待结束
 - 前台进程中的线程完成一个等待操作
 - 由于窗口活动而唤醒图形用户接口线程
 - 线程处于就绪状态超过一定时间，但没能进入运行状态(处理机饥饿)
- ✓ 线程优先级提升的目的是改进系统吞吐量、响应时间等整体特征，解决线程调度策略中潜在的不公正性。但它也不是完美的，它并不会使所有应用都受益
- ✓ Windows 永远不会提升实时优先级范围内(16至31)的线程优先级



线程优先级提升

1) I/O操作完成后的线程优先级提升

- ✓ 在完成I/O操作后，Windows 将临时提升等待该操作线程的优先级，以保证等待I/O操作的线程能有更多的机会立即开始处理得到的结果
- ✓ 为了避免I/O操作导致对某些线程的不公平，在I/O操作完成后唤醒等待线程时将把该线程的时间配额减1
- ✓ 线程优先级的实际提升值是由设备驱动程序决定的
- ✓ 设备驱动程序在完成I/O请求时通过内核函数IoCompleteRequest来指定优先级提升的幅度
- ✓ 线程优先级的提升幅度与I/O请求的响应时间要求是一致的，响应时间要求越高，优先级提升幅度越大



线程优先级提升

2) 等待事件和信号量后的线程优先级提升

- ✓ 当一个等待事件对象或信号量对象的线程完成等待后，将提升一个优先级
- ✓ 阻塞于事件或信号量的线程得到的处理机时间比处理机繁忙型线程要少，这种提升可减少这种不平衡带来的影响



线程优先级提升

3) 前台线程在等待结束后的优先级提升

- ✓ 对于前台线程，一个内核对象上的等待操作完成时，内核函数 KiUnwaitThread 会提升线程的当前优先级 (不是线程的基本优先级)
- ✓ 在前台应用完成它的等待操作时小幅提升其优先级，以使它更有可能马上进入运行状态，有效改善前台应用的响应时间特性



线程优先级提升

4) 图形用户接口线程被唤醒后的优先级提升

- ✓ 拥有窗口的线程在被窗口活动唤醒(如收到窗口消息)时将得到一个幅度为2的额外优先级提升
- ✓ 这种优先级提升的目的是为了改善交互应用的响应时间



线程优先级提升

5) 对处理机饥饿线程的优先级提升

- ✓ 系统线程“平衡集管理器(balance set manager)”会每秒钟检查一次就绪队列，看是否存在一直在就绪队列中排队超过300个时钟中断间隔的线程
- ✓ 如果找到这样的线程，平衡集管理器将把该线程的优先级提升到15，并分配给它一个长度为正常值两倍的时间配额
- ✓ 当被提升线程用完它的时间配额后，该线程的优先级立即衰减到它原来的基本优先级



线程调度

(5) 空闲线程

- ✓ 如果在一个处理机上没有可运行的线程，Windows会调度相应处理机对应的空闲线程
- ✓ 由于在多处理机系统中可能存在多个处理机同时运行空闲线程，所以系统中的每个处理机都有一个对应的空闲线程
- ✓ Windows给空闲线程指定的线程优先级为0，该空闲线程只在没有其他线程运行时才运行

空闲线程的功能就是在一个循环中检测是否有要做的工作，例如处理所有待处理的中断请求，检查是否有就绪线程可进入运行状态。如果有，调度相应线程进入运行状态，调用硬件抽象层的处理机空闲例程，执行相应的电源管理功能等



5.4 Windows 2000/xp的进程与内存管理

四、线程间同步与通信

1. 内核的同步与互斥机制

(1) 提高临界区代码执行的中断优先级，暂时屏蔽那些有可能也要使用该临界资源的中断。

只能用于单处理机。因为一个处理机上提高中断优先级不能阻止其他处理机上的中断。

(2) 转锁 (spin-lock) 机制

用TestAndSet指令实现，可用于多处理机。



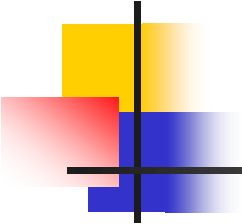
线程的同步与通信

2. 线程间同步

(1) 同步对象

Windows提供了以下几种同步对象的API，用于进程和线程同步：

- ✓ 事件 (Event)
- ✓ 互斥量 (Mutex)
- ✓ 信号量 (Semaphore)
- ✓ 临界区 (Critical Section)



同步对象

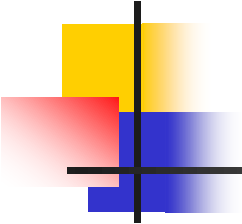
同步对象的2种状态：

- ✓ 可用，或称有信号（signaled）：允许等待函数返回
- ✓ 不可用，或称无信号（nonsignaled）：阻止等待函数返回

这些同步对象有2种使用方式：

- ✓ 无名对象：只能用于同一进程中的线程同步
- ✓ 有名对象：可以用于不同进程中的线程同步。临界区除外。

为同步对象指定一个名称（字符串），不同进程用同样名称打开或创建对象，从而获得该对象在本进程的句柄。



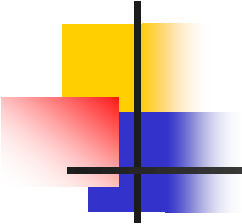
同步对象

1) 事件

相当于“触发器”，用于通知一个或多个等待它的线程“事件已发生”。

相关API有CreateEvent、OpenEvent、SetEvent、ResetEvent。

- ✓ CreateEvent：创建一个事件对象，返回对象句柄。
- ✓ OpenEvent：返回一个已存在的事件对象的句柄，用于后续访问。
- ✓ SetEvent：设置指定事件对象为可用状态
- ✓ ResetEvent：设置指定事件对象为不可用状态



同步对象

2) 互斥量 (Mutex)

就是互斥信号量，某一时刻只能被一个线程所拥有。

相关API有CreateMutex、OpenMutex、ReleaseMutex。

- ✓ CreateMutex：创建一个互斥量对象，返回对象句柄。
- ✓ OpenMutex：返回一个已存在的互斥量对象的句柄，用于后续访问。
- ✓ ReleaseMutex：释放对互斥量对象的占用，使之成为可用。



同步对象

3) 信号量 (Semaphore)

取值为0到指定最大值之间，可用来限制并发访问共享资源的线程个数。

相关API有CreateSemaphore、OpenSemaphore、ReleaseSemaphore等。

- ✓ CreateSemaphore：创建一个信号量对象，在参数中指定最大值和初始值，返回对象句柄。
- ✓ OpenSemaphore：返回一个已存在的信号量对象的句柄，用于后续访问。
- ✓ ReleaseSemaphore：释放信号量对象，使之成为可用。



同步对象

4) 临界区 (Critical Section)

同时只能由一个线程所拥有，但只能用于同一个进程中的线程互斥。

相关API有InitializeCriticalSection、EnterCriticalSection、LeaveCriticalSection、DeleteCriticalSection等。

- ✓ InitializeCriticalSection：对临界区对象进行初始化。
- ✓ EnterCriticalSection：等待拥有临界区对象，得到使用权后返回。
- ✓ LeaveCriticalSection：释放临界区对象的使用权。
- ✓ DeleteCriticalSection：释放与临界区对象相关的所有资源。



线程间同步

(2) 等待函数

对于同步对象，Win32 API提供了**2个等待函数**：

WaitForSingleObject与WaitForMultipleObjects。

等待函数执行时会阻塞，直到超时或者等待条件满足。

WaitForSingleObject：等待指定对象为可用状态

WaitForMultipleObjects：等待多个对象为可用状态



线程间同步

(3) 使用Win32 API实现线程同步的一般方法

以信号量为例，其他同步对象类似。

- ① 给信号量对象约定一个名字；
- ② 一个线程以指定的名字为参数，调用CreateSemaphore()创建一个信号量，返回句柄；
任一线程访问共享资源前，以指定的名字调用OpenSemaphore()打开该信号量，返回句柄；
- ③ 以句柄为参数调用等待函数（P操作）；
- ④ 访问完后，调用ReleaseSemaphore释放该信号量（V操作）。



线程的同步与通信

3. 线程间通信 —— 共享存储区 (shared memory)

- ✓ 共享存储区可用于进程间的大数据量通信
- ✓ 进行通信的各进程可以任意读写共享存储区，也可在共享存储区上使用任意数据结构
- ✓ 在使用共享存储区时，需要进程互斥和同步机制的辅助来确保数据一致性
- ✓ Windows 2000/XP采用文件映射(file mapping)机制来实现共享存储区，用户进程可以将整个文件映射为进程虚拟地址空间的一部分来加以访问



共享存储区

Windows 的文件映射（内存映象文件）：

- ✓ CreateFileMapping为指定文件创建一个文件映射对象，返回对象句柄
- ✓ OpenFileMapping打开一个命名的文件映射对象，返回对象句柄
- ✓ MapViewOfFile把文件视图（文件的一部分或全部）映射到本进程的地址空间，返回映射地址空间的首地址。当完成文件到进程地址空间的映射后，就可利用首地址进行读写
- ✓ FlushViewOfFile可把映射地址空间的内容写到物理文件中
- ✓ UnmapViewOfFile拆除文件映射与本进程地址空间的映射关系
- ✓ CloseHandle可关闭文件映射对象

共享存储区

基于文件映射的共享存储区的用法：

CreateFile()

from system page file:
(HANDLE)0xffffffff

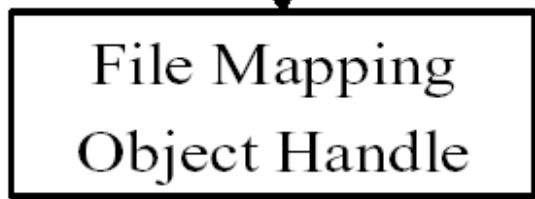
File Handle

CreateFileMapping()

OpenFileMapping()

File Mapping
Object Handle

MapViewOfFile()





线程的同步与通信

除了共享存储区外，Windows还提供以下进程通信机制：

- ✓ 管道：无名管道，有名管道
- ✓ 信号
- ✓ 邮件槽：不定长、不可靠的单向消息通信机制
- ✓ Socket：网络通信机制



5.4 Windows 2000/xp的进程与内存管理

五、内存管理

1. 内存管理器的组成部分

6个关键组件：

- ✓ 工作集管理器(MmWorkingSetManager)：当空闲内存低于某一界限值时，便启动所有的内存管理策略，如：工作集的修整、老化和已修改页的写入等
- ✓ 进程/堆栈交换器(KeSwapProcessOrStack)：完成进程和内核线程堆栈的换入和换出操作
- ✓ 已修改页写入器(MiModifiedPageWriter)：将已修改页链表上的“脏”页写回到页文件

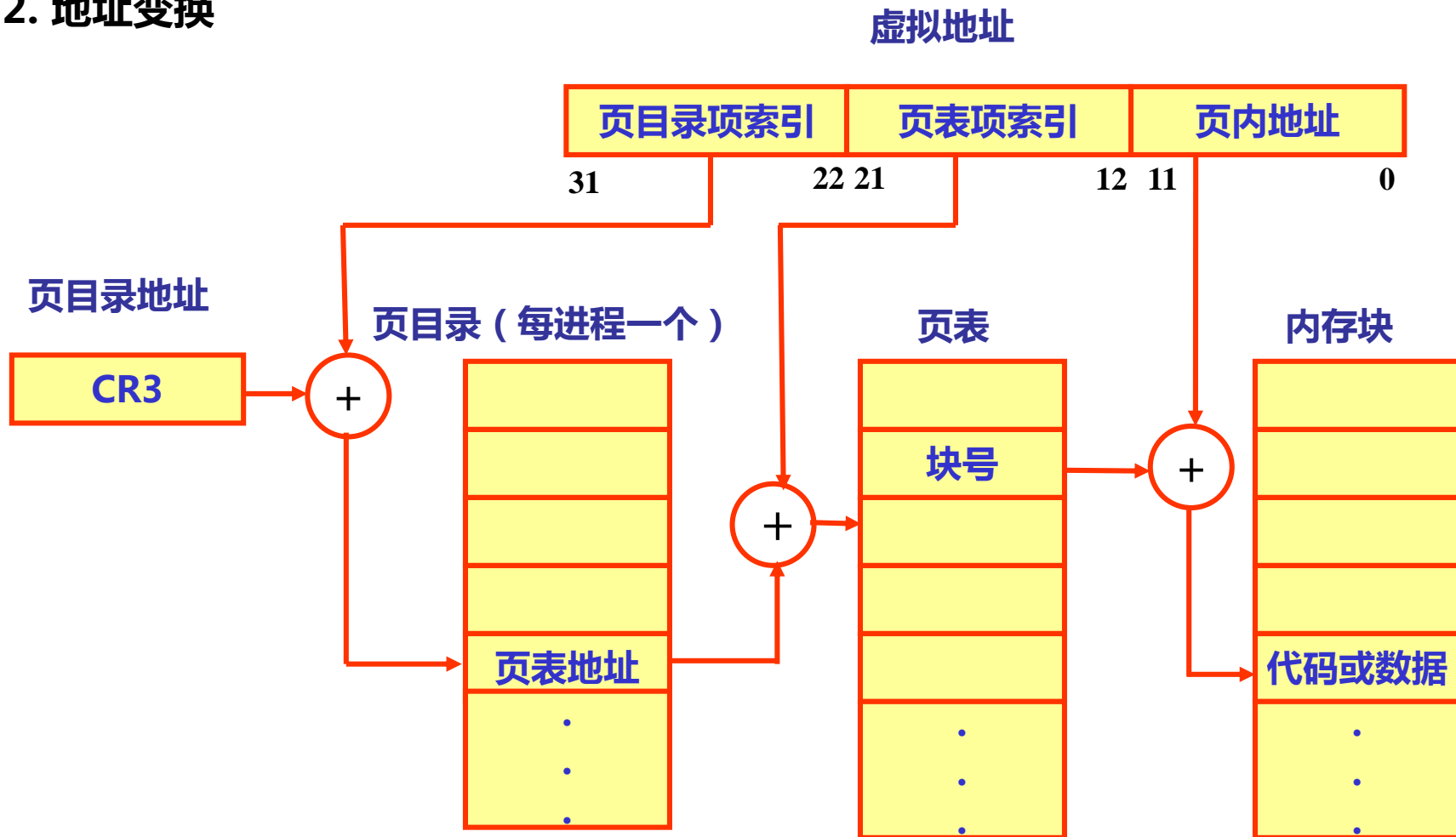


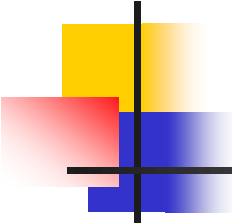
内存管理的组成

- ✓ 映射页写入器(MiMappedPageWriter)：将映射文件中脏页写回磁盘
- ✓ 废弃段线程(MiDereferenceSegmentThread)：负责系统高速缓存和页文件的扩大和缩小
- ✓ 零页线程(MmZeroPageThread)：将空闲链表中的页清零

内存管理

2. 地址变换





内存管理

说明：

- (1) 页内地址占12位，页大小 = 4KB
- (2) 每个进程1个页目录，每个页目录占4KB (1024项，每项4B)
- (3) 页表大小是4KB (1024项，每项4B)
- (4) 页表不常驻内存，可换入/换出
- (5) 运行PAE (物理地址扩展) 内核的系统是三级页表



内存管理

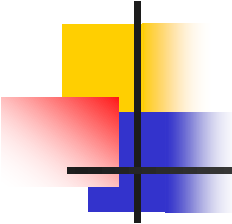
3. 页的调度策略

(1) 取页策略

- ✓ 请求调页
- ✓ 提前取页：采用簇（cluster）方式将页装入内存

(2) 置换策略

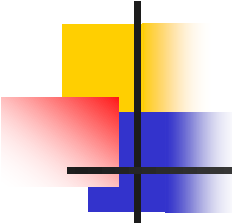
- ✓ 采用局部置换策略
- ✓ 在多处理器系统中，采用了FIFO置换算法。而在单处理器系统中，其实现接近于最近最少使用策略（LRU）算法（称为轮转算法）
- ✓ 使用“自动调整工作集”技术



5.5 小结

一、Linux

1. Linux的3个接口
2. 系统调用
3. 用户态和核心态
4. Shell的功能
5. 进程创建与执行：fork()，exec()
6. 进程调度时机
7. 了解进程同步与通信机制及其适用场合
信号，消息队列，共享内存，信号量
8. 内存管理
页式管理；页的周期性换出；交换缓冲区



5.5 小结

二、Windows 2000/XP

1. 线程优先级提升的思想

2. 进程/线程的创建

Createprocess(); CreateThread()

3. 了解线程同步与通信机制

同步对象，共享内存（内存映射文件）