

## 版权声明

- 本内容版权归北京理工大学计算机学院操作系统课程组马锐所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源
  - 内容来自：北京理工大学计算机学院+马锐+材料名字
- 对于不准守此声明或其他违法使用本内容者，将依法保留追究权

2

## 第2章 进程、线程与调度

- 2.1 进程
- 2.2 线程
- 2.3 操作系统调度

3

## 2.1 进程

- 2.1.1 进程的引入与概念
- 2.1.2 进程的描述
- 2.1.3 进程的控制

4

### 2.1.1 进程的引入与概念

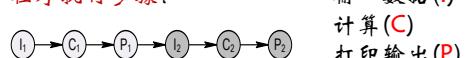
- 2.1.1.1 程序的顺序执行
- 2.1.1.2 程序的并发执行
- 2.1.1.3 进程的基本概念

5

### 2.1.1.1 程序的顺序执行(1)

例1：有一程序对用户输入的数据进行计算并打印计算结果。

程序执行步骤：



输入数据(I)  
计算(C)  
打印输出(P)

例2：有一包含3条语句的程序段：

S1: a=x+y;

S2: b=a-5;

S3: c=b+1;

(S1) → (S2) → (S3)

语句执行顺序为：

6

### 2.1.1.1 程序的顺序执行(2)

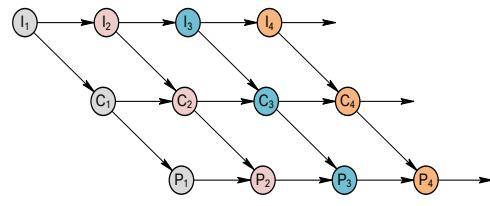
- 程序顺序执行的特点
  - 程序执行的连续性
  - 程序环境的封闭性
  - 程序结果的可再现性
- 优点
  - 顺序程序的封闭性和可再现性为程序员调试程序带来了很大方便
- 缺点
  - 资源的独占性使得系统资源利用率非常低

7

### 2.1.1.2 程序的并发执行(1)

例3：系统中有多道程序对用户输入的数据进行计算并打印计算结果。

程序执行步骤：



8

### 2.1.1.2 程序的并发执行(2)

➤ 程序并发执行的特点

- 增强了计算机系统的处理能力，提高了资源利用率
- 程序执行的间断性：并发执行的程序间产生了相互制约关系
  - 执行—暂停—执行
  - 因共享资源或协调完成同一任务引起
    - 间接制约：共享资源
    - 直接制约：协同完成任务

9

### 2.1.1.2 程序的并发执行(3)

• 失去了程序的封闭性

• 资源共享

• 程序结果的不可再现性

例4：有两个并发执行的程序A和B，共享一个变量count，count的初值为N。

Program A:

.....  
S1 count=count+1;  
.....

Program B:

.....  
Print(count);  
count=0;  
.....  
S2 S3

10

### 2.1.1.2 程序的并发执行(4)

程序A和B执行时count的结果取决于语句执行顺序

- S1,S2,S3, count: N+1,N+1,0
- S2,S3,S1, count: N,0,1
- S2,S1,S3, count: N,N+1,0

• 程序与CPU执行的活动之间不再一一对应

- 程序是完成某一特定功能的指令序列，是静态的
- CPU执行的活动是动态的

11

### 2.1.1.3 进程的基本概念(1)

➤ 进程的定义

- 进程是程序的一次执行
- 进程是可以并发执行的计算
- 进程是一个程序与其使用的数据在处理器上顺序执行时发生的活动
- 进程是程序在一个数据集合上的运行过程，它是系统进行资源分配和调度的一个独立单位
- 进程是可以和其他程序并发执行的程序关于某个数据集合的一次执行

12

### 2.1.1.3 进程的基本概念(2)

➤ 进程的特性

- 动态性：进程是程序的一次执行，具有生命周期
- 独立性：进程是系统进行资源分配和调度的一个独立单位
- 并发性：进程可以并发执行
- 异步性：进程间的相互制约，使进程执行具有间隙
- 结构性：进程具有结构
  - 程序
  - 数据（地址空间、堆栈等）
  - 进程控制块（PCB）

13

## 2.1.2 进程的描述

- 2.1.2.1 进程的状态
- 2.1.2.2 进程控制块
- 2.1.2.3 进程上下文切换
- 2.1.2.4 进程的内存空间布局
- 2.1.2.5 进程的组织

14

### 2.1.2.1 进程的状态(1)

#### 三种基本状态

- **运行态:** 正在计算单元(CPU core)上运行的进程所处状态(Instructions are being executed.)
- **就绪态:** 已经获得了除CPU core之外的全部资源并等待系统分配CPU core, 一旦获得CPU core即可以变为运行态的进程状态
- **阻塞态/等候态:** 一个进程因等待某事件发生而不能运行时所处状态

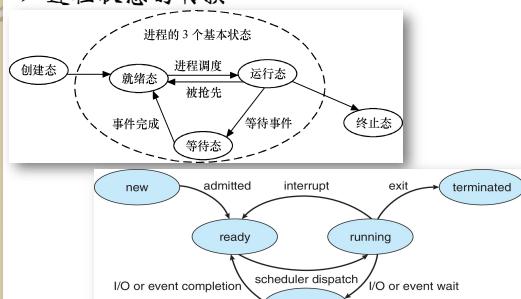
#### 创建态与终止态

- **创建态:** 进程被创建时的状态
- **终止态:** 进程运行完成时的状态

15

### 2.1.2.1 进程的状态(2)

#### 进程状态的转换



### 2.1.2.1 进程的状态(3)

- 运行态 → 就绪态
- 运行态 → 等待态
- 就绪态 → 运行态
- 等待态 → 就绪态

#### 创建态 → 就绪态

- 操作系统准备好再接纳一个进程时，把一个进程从创建态变为就绪态。
- 运行态 → 终止态

  - 进程已结束，但尚未撤消，以便其它进程去收集该进程的有关信息。

17

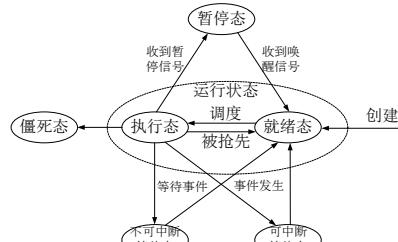
### 2.1.2.1 进程的状态(4)

#### Linux进程状态

- 进程生命期状态
- 可运行状态: 进程正在或准备在CPU上运行的状态
  - 可中断的等待状态: 进程睡眠, 等待系统资源可用或收到一个信号后, 进程被唤醒
  - 不可中断的等待状态: 进程睡眠, 等待一个不可被中断的事件发生 (很少使用)
  - 暂停状态
  - 跟踪状态
  - 僵死状态: 进程已终止, 等待父进程处理
  - 死亡状态: 系统删除该进程 进程退出状态

18

### 2.1.2.1 进程的状态(5)



Linux进程状态转换图

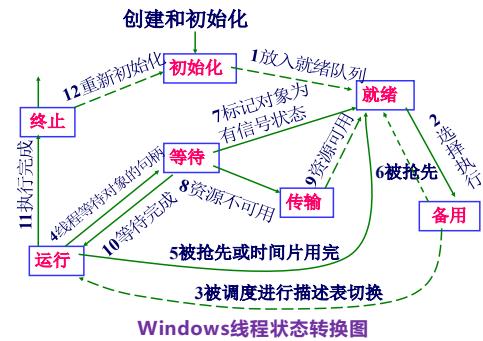
19

### 2.1.2.1 进程的状态(6)

- Windows线程的状态
  - 就绪状态(ready)
  - 备用状态(standby)
    - ◆ 已选好处理器，正等待描述表切换，以便进入运行状态
  - 运行状态(Running)
  - 等待状态(waiting)
  - 传输状态(transition)
    - ◆ 核心栈被调到外存的就绪态
  - 终止状态(terminated)
  - 初始化状态(Initialized)

20

### 2.1.2.1 进程的状态(7)



21

### 2.1.2.2 进程控制块PCB(1)

- 进程存在的唯一标识
- 包含进程的描述信息和管理控制信息
  - 进程标识符：内部/外部
  - 进程的状态（当前状态）
  - 进程调度信息
    - 进程优先级
    - 进程所在各种队列的指针
  - 进程的存储管理信息
  - 进程要执行程序的内外存起始地址及采取的保护信息

22

### 2.1.2.2 进程控制块PCB(2)

- 进程使用的资源信息
  - 分配给进程的I/O设备
  - 正在执行的I/O请求信息
  - 当前进程正打开的文件
- CPU现场保护区
  - 程序计数器
  - 工作寄存器
  - 程序状态字
  - 堆栈指针
- 进程之间的家族关系
  - 父进程
  - 子进程

23

### 2.1.2.2 进程控制块PCB(3)

```
//Linux中的PCB中的部分信息
struct task_struct {
    pid_t pid;           //进程标识符(32767)
    struct thread_info *thread_info; //当前进程基本信息
    long state;          //进程状态
    int prio;            //进程优先级0-139
    int static_prio;     //进程的静态优先级
    unsigned long rt_priority; //进程的实时优先级
    struct list_head tasks; //进程链表
    struct task_struct *parent; //指向父进程
    struct list_head children; //子进程链表
    struct files_struct *files; //打开文件列表
    struct mm_struct *mm, *active_mm; //指向进程拥有和执行
                                    //的虚拟内存描述符
};
```

24

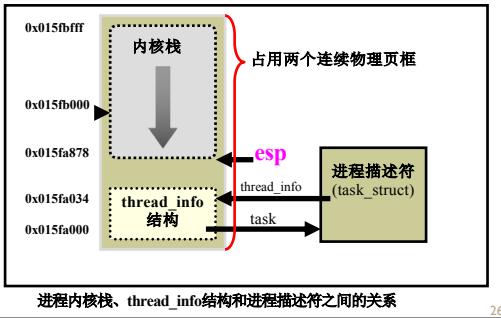
### 2.1.2.2 进程控制块PCB(4)

- Linux进程基本信息

```
struct thread_info {
    struct task_struct* task;      //进程描述符指针
    unsigned long flags;          //重调度标识
    struct exec_domain exec_domain;
    int preempt_count;           //软中断计数器
    __u32 cpu;
    struct restart_block restart_block;
};
```

25

### 2.1.2.2 进程控制块PCB(5)



26

### 2.1.2.2 进程控制块PCB(6)

- Linux把每个进程的内核栈和基本信息 **thread\_info** 存放在两个连续的页框(8KB)中，第1个页框的起始地址是 $2^{13}$ 倍数
- 由于 **esp** 寄存器存放的是内核栈的栈顶指针，内核很容易从 **esp** 寄存器的值获得正在CPU上运行进程的 **thread\_info** 结构的地址，进而获得进程描述符(PCB)的地址
- 进程刚从用户态切换到内核态时，其内核栈为空，只要将栈顶指针减去8k，就能得到 **thread\_info** 结构的地址

27

### 2.1.2.2 进程控制块PCB(7)

- Windows的进程
  - 执行体进程块EPROCESS
    - 表示进程的基本属性
  - 内核进程块KPROCESS
    - **进程控制块**
    - 包含了内核调度线程的必要信息，例如基本优先级、默认时间片、进程状态、进程目录表等
  - 进程环境块PEB
    - 位于进程地址空间（用户模式）
    - 包含了映像加载程序需要的信息、线程使用的堆信息等

28

### 2.1.2.2 进程控制块PCB(8)

```
//Windows
typedef struct _EPROCESS {
    KPROCESS Pcb;           //内核进程块
    PPEB Peb;              //进程环境块
    HANDLE UniqueProcessID; //进程编号，PID
    LARGE_INTEGER CreateTime; //进程创建时间
    LARGE_INTEGER ExitTime; //进程退出时间
    KSPIN_LOCK HyperSpaceLock; //自旋锁
    .....
};
```

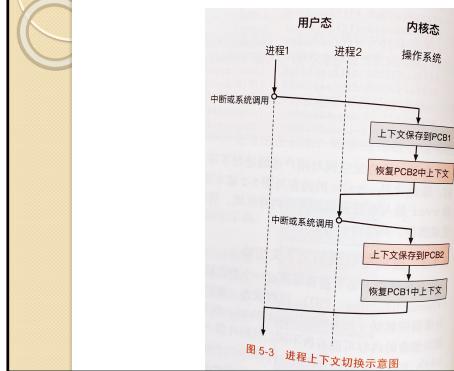
29

### 2.1.2.3 进程上下文切换(1)

- 进程的**上下文(context)**包括进程运行时的**寄存器状态**，能够用于保存和恢复一个进程在处理器上运行的状态
- 当操作系统需要切换当前执行进程时，使用**上下文切换(context switch)**机制
  - 上下文保存在对应的PCB中
  - 进程被调度时，从PCB中取出上下文并恢复
  - 上下文切换时，进程通过中断或系统调用进入内核

30

### 2.1.2.3 进程上下文切换(2)



31

### 2.1.2.3 进程上下文切换(3)

#### ➤ Linux的进程切换

- 上下文
  - 硬件上下文：装入CPU寄存器的一组数据
  - Linux 2.6以后：软件实现进程上下文切换
- 进程切换只发生在内核态
  - 在发生进程切换之前，用户态进程使用的所有寄存器值都已被保存在进程的内核栈中

32

### 2.1.2.3 进程上下文切换(4)

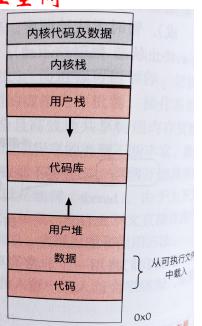
- 之后，大部分寄存器值存放在PCB的类型为`thread_struct`的`thread`字段（进程硬件上下文）里，一小部分仍在内核栈中
- 进程切换步骤
  - 切换页面目录表以安装一个新的地址空间
  - 切换内核栈和硬件上下文，由`schedule()`函数完成切换

33

### 2.1.2.4 进程的内存空间布局(1)

#### ➤ 进程具有独立的虚拟地址空间

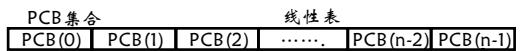
- 用户栈
- 代码库
- 用户堆
- 数据与代码段
- 内核部分
  - 内核栈
  - 内核代码与数据



### 2.1.2.5 进程的组织(1)

#### ➤ 线性表方式

- 将所有进程的PCB组成一个数组，系统通过数组下标访问每一个PCB
- 优点
  - 简单，节省存储空间
- 缺点
  - 系统开销大，查找一个指定的PCB较费时间

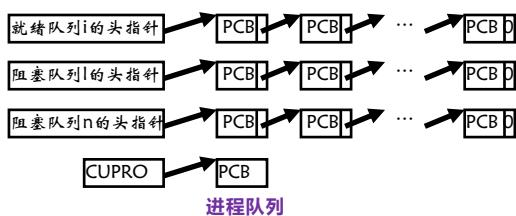


35

### 2.1.2.5 进程的组织(2)

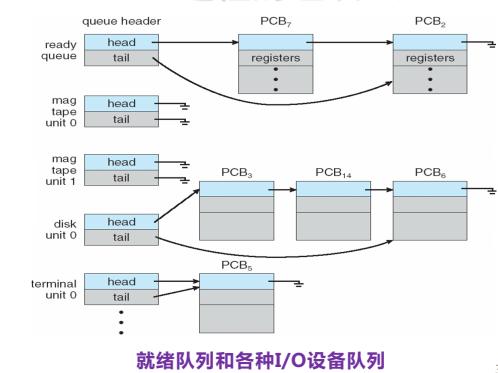
#### ➤ 链接方式

- 将处于同一状态的进程按照一定方式链接成一个队列



36

### 2.1.2.5 进程的组织(3)



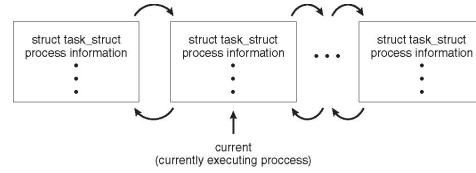
37

## 2.1.2.5 进程的组织(4)

- Linux (传统) 进程链表(list\_head)
  - 所有进程链表(tasks)
    - ◆ 链表头是0号进程(idle进程)
  - 可运行进程链表(run\_list)
    - ◆ 双向链表(140级)
  - 子进程链表(children)
  - 兄弟进程链表(sibling)
  - 等待进程链表
    - ◆ 互斥等待临界资源的进程：一次唤醒一个
    - ◆ 非互斥等待的进程：唤醒所有进程

38

## 2.1.2.5 进程的组织(5)



Linux进程队列

39

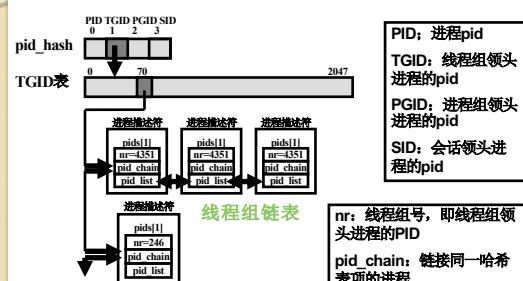
## 2.1.2.5 进程的组织(6)

- 哈希链表
  - 内核定义了4类哈希表
    - ◆ PIDTYPE\_PID链表：进程
    - ◆ PIDTYPE\_TGID链表：线程组
      - 同一线程组中的所有轻量级进程的tgid值相同
    - ◆ PIDTYPE\_PGID链表：进程组
    - ◆ PIDTYPE\_SID链表：会话
  - 哈希表地址存入pid\_hash中

40

## 2.1.2.5 进程的组织(7)

- PIDTYPE\_TGID哈希链表



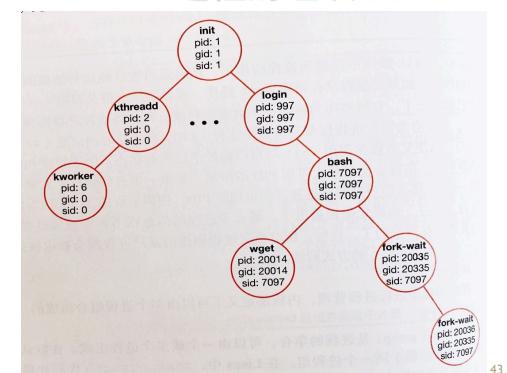
41

## 2.1.2.5 进程的组织(8)

- 进程组与会话
  - 为方便应用程序管理，内核定义的由多个进程组合而成的“小集体”
- 进程组
  - 进程的集合，由一个或多个进程组成
  - 父子进程默认属于同一进程组
- 会话
  - 进程组的集合，由一个或多个进程组构成
  - 根据执行状态分为前台进程组和后台进程组

42

## 2.1.2.5 进程的组织(9)



43

## 2.1.3 进程的控制

- 2.1.3.1 进程控制及原语
- 2.1.3.2 创建原语
- 2.1.3.3 撤销原语
- 2.1.3.4 阻塞原语
- 2.1.3.5 唤醒原语

44

### 2.1.3.1 进程控制及原语

#### ➤ 进程控制

- 系统使用一些具有特定功能的程序段来创建、撤销进程以及完成进程各状态间转换等一系列有效管理
- 一般由操作系统内核完成

#### ➤ 原语(primitive)

- 某些程序段的执行过程是不允许被中断的，或者说其执行过程不可分割。这样的程序段叫原语

45

### 2.1.3.2 创建原语(1)

#### ➤ 创建进程的时机

系统内核在处理系统中为每个作业创建一个进程  
分时系统中为每个用户创建一个进程  
创建提供服务：打印进程 应用程序创建

- 应用请求：已存在的进程创建子进程

#### ➤ 创建原语的功能

- 申请空白PCB
- 为新进程分配资源
- 初始化进程控制块
- 将新进程插入就绪队列

创建  
进程控制块

46

### 2.1.3.2 创建原语(2)

#### ➤ 创建进程时需注意的问题

- 资源共享
  - 父子进程共享资源
  - 子进程共享部分父进程资源
  - 父子进程不共享资源
- 数据
  - 父进程传递给子进程初始数据
- 地址空间
  - 子进程复制父进程的地址空间
  - 子进程新创建自己的地址空间

47

### 2.1.3.2 创建原语(3)

#### ➤ 进程执行

- 父子进程并发执行
- 父进程等待至子进程执行完毕

#### ➤ Linux的进程创建—fork()

- fork()系统调用创建新进程
- 父子进程是2个完全独立的进程，拥有不同的pid与虚拟地址空间，但内存、寄存器、程序计数器等状态都完全一致

48

### 2.1.3.2 创建原语(4)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[]){
    pid_t pid;
    int x=100;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
    } else if (pid == 0) { /* child process */
        printf("Child process: x is: %d\n",
               The value of x is: %d\n", pid, x);
    } else { /* parent process */
        printf("Parent process: x is: %d\n",
               The value of x is: %d\n", pid, x);
    }
}
```

49

### 2.1.3.2 创建原语(5)

#### ➤ 运行结果

```
ting@ubuntu:~/Desktop$ ./test
The value of x is: 100
child process: x is: 100
ting@ubuntu:~/Desktop$ ./test
The value of x is: 100
ting@ubuntu:~/Desktop$ ./test
Parent process: x is: 3543;
child process: x is: 0;
The value of x is: 100
ting@ubuntu:~/Desktop$ ./test
Parent process: x is: 3547;
child process: x is: 0;
The value of x is: 100
ting@ubuntu:~/Desktop$ ./test
Parent process: x is: 3549;
child process: x is: 0;
The value of x is: 100
ting@ubuntu:~/Desktop$ ./test
Parent process: x is: 3551;
child process: x is: 0;
The value of x is: 100
ting@ubuntu:~/Desktop$ ./test
Parent process: x is: 3553;
child process: x is: 0;
The value of x is: 100
ting@ubuntu:~/Desktop$ ./test
Parent process: x is: 3557;
child process: x is: 0;
The value of x is: 100
ting@ubuntu:~/Desktop$ ./test
Parent process: x is: 3559;
child process: x is: 0;
The value of x is: 100
```

50

### 2.1.3.2 创建原语(6)

#### ➤ Linux创建进程函数fork()、clone()、vfork()

- **创建子进程函数fork()**: 创建成功之后，子进程采用写时复制技术读共享父进程的全部地址空间，仅当父或子要写一个页时，才为其复制一个私有的页的副本
- **创建轻量级进程函数clone()**: 实现对多线程应用程序的支持。共享进程在内核的很多数据结构，如页表、打开文件表等等
- **vfork()**: 阻塞父进程直到子进程退出或执行了一个新程序为止

51

### 2.1.3.2 创建原语(7)

#### ➤ Linux创建内核线程函数kernel\_thread()

- **0号进程**就是一个内核线程，0号进程是所有进程的祖先进程，又叫idle进程或叫做swapper进程。每个CPU都有一个0号进程
- **1号进程**是由0号进程创建的内核线程init，负责完成内核的初始化工作。在系统关闭之前，init进程一直存在，它负责创建和监控在操作系统外层执行的所有用户态进程

52

### 2.1.3.2 创建原语(8)

#### ➤ Windows创建进程的主要流程

- 打开可执行文件(.exe)，创建一个区域对象，建立可执行文件与虚拟内存之间的映射关系
- 创建执行体进程对象EPROCESS
- 创建一个主线程
- 通知Win32子系统，对新进程和线程进行一系列初始化
- 完成地址空间的初始化，开始执行程序

53

### 2.1.3.2 创建原语(9)

#### ➤ Win32 API创建进程

```
#include <stdio.h>
#include <windows.h>
int main(){
    STARTUPINFO si;           //for new process
    PROCESS_INFORMATION pi;
    //allocate memory
    ZeroMemory(&si,sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi,sizeof(pi));
    //create child processes
    if(!CreateProcess(NULL,   //use command line
                    "C:\Windows\system32\mspaint.exe",
                    NULL,    //don't inherit process handle
                    NULL,    //don't inherit thread handle
                    FALSE,   //disable handle inheritance
                    0,        //no creation flags
                    NULL,    //use parent's environment block
                    NULL,    //use parent's existing directory
                    &si,
                    &pi)) {
        fprintf(stderr,"create failed.");
        return -1;
    }
}
```

54

### 2.1.3.2 创建原语(10)

```
FALSE, //disable handle inheritance
0,      //no creation flags
NULL,   //use parent's environment block
NULL,   //use parent's existing directory
&si,
&pi)) {
fprintf(stderr,"create failed.");
return -1;
}
//parent will wait for the child to complete
WaitForSingleObject(pi.hProcess,INFINITE);
printf("Child Completed");
//close handles
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
}
```

55

### 2.1.3.3 撤销原语(1)

#### ➤ 撤销进程的时机

- 进程已完成任务，正常结束
- 由于故障不能继续执行，异常结束
  - 越界错误、保护错、非法指令、运行超时、算术运算错、I/O故障
- 外界干预
  - 操作员或操作系统干预：死锁
  - 父进程请求
  - 父进程终止

56

### 2.1.3.3 撤销原语(2)

#### ➤ 撤销原语的功能

- 根据被终止进程标识符查找被撤销进程PCB
- 终止该进程，重新调度
- 终止该进程的全部子进程
- 回收资源
- 释放PCB

#### ➤ 撤销进程时需注意的问题

- 是否撤销该进程的子进程

57

### 2.1.3.3 撤销原语(3)

#### ➤ Linux中的进程撤销

- 进程终止
  - exit()系统调用只终止某一个线程。
  - exit\_group()系统调用能终止整个线程组
- 进程删除
  - 父进程先结束的子进程会成为孤儿进程，系统会强迫所有的孤儿进程成为init进程的子进程
  - init进程在用wait()类系统调用检查并终止子进程时，就会撤销所有僵死的子进程

58

### 2.1.3.3 撤销原语(4)

#### ➤ Windows中的进程撤销

- ExitProcess和TerminateProcess终止进程和进程中的所有线程
  - 正常退出：ExitProcess
  - 异常退出：TerminateProcess，可以终止自己，也可以终止其他进程

59

### 2.1.3.4 阻塞原语

#### ➤ 阻塞进程的时机

- 处于运行状态的进程等待某一事件发生
  - 等待I/O数据传输完成
  - 等待其他进程发送信息

#### ➤ 阻塞原语的功能

- 处于运行态的进程中断CPU，将其运行现场保存在其PCB的CPU现场保护区
- 将其状态置为阻塞态，并插入相应事件的等待队列
- 转进程调度，选择一个就绪进程投入运行

#### ➤ 阻塞原语由进程自己执行

60

### 2.1.3.5 唤醒原语(1)

#### ➤ 唤醒进程的时机

- 进程期待的事件到来时

#### ➤ 唤醒原语的功能

- 期待的事件是等待输入输出完成

- 输入/输出完成后，由硬件提出中断请求，CPU响应中断，暂停当前进程的执行，转去中断处理，检查有无等待该输入/输出完成的进程
- 有则将该进程从等待队列抽出，并将其由阻塞态置为就绪态，插入就绪队列，结束中断处理

61