操作系统
Operating System

北京理工大学计算机学院
马　锐
Email: mary@bit.edu.cn

版权声明

第2章　进程、线程与调度

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

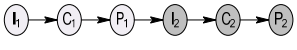
2.1.1.3 进程的基本概念

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。

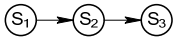
程序执行步骤：



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;



语句执行顺序为：

6

2.1.1.1 程序的顺序执行(2)

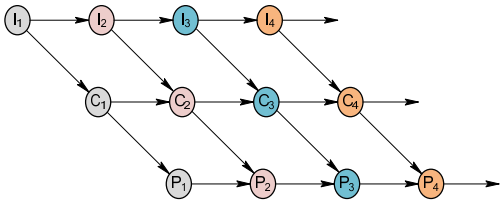
- 程序顺序执行的特点
  - 程序执行的连续性
  - 程序环境的封闭性
  - 程序结果的可再现性
- 优点
  - 顺序程序的封闭性和可再现性为程序员调试程序带来了很大方便
- 缺点
  - 资源的独占性使得系统资源利用率非常低

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:		Program B:	
	……		……	
S1	count=count+1;		Print(count);	S2
	……		count=0;	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机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 进程控制块

2.1.2.2 进程的状态

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

15

2.1.2.1 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

16

2.1.2.1 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
   ......
};
```

17

2.1.2.1 进程控制块PCB(4)

➢ 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、进程页目录表等
- 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

18

2.1.2.1 进程控制块PCB(5)

```
//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

  ......
};
```

19

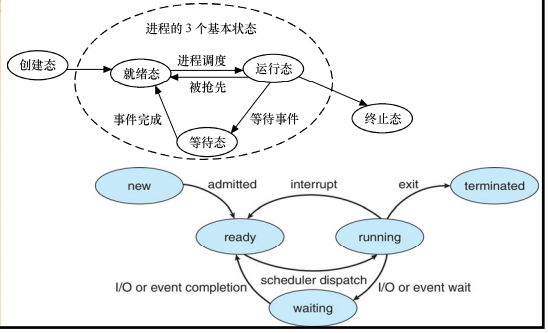
2.1.2.2 进程的状态(1)

- 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

20

2.1.2.2 进程的状态(2)

- 进程状态的转换



2.1.2.2 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。

22

2.1.2.2 进程的状态(4)

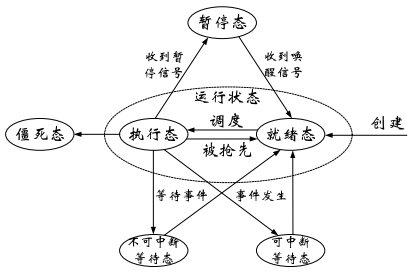
➢ Linux进程状态

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

23

2.1.2.2 进程的状态(5)



Linux进程状态转换图

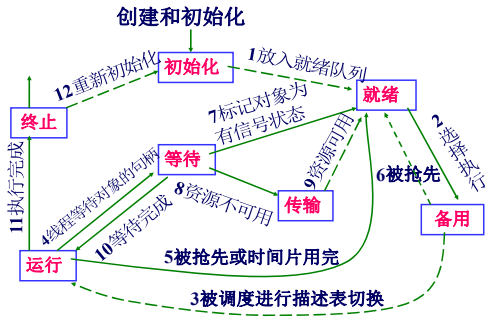
24

2.1.2.2 进程的状态(6)

- Windows线程的状态
  - 就绪状态(ready)
  - 备用状态(standby)
    - 已选好处理机，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)

25

2.1.2.2 进程的状态(7)



Windows线程状态转换图

26

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
- 进程切换只发生在核心态

27

2.1.2.3 进程上下文切换(2)

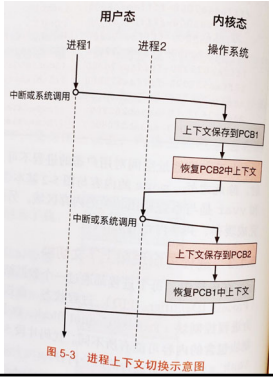


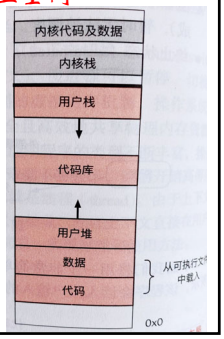
图 5-3 进程上下文切换示意图

2.1.2.3 进程上下文切换(3)

- Linux进程切换
  - 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核心栈中
  - 之后，大部分寄存器值存放在thread_struct的thread字段（进程硬件上下文）里，一小部分仍在核心栈中
  - 进程切换步骤
    - 切换页目录表以安装一个新的地址空间
    - 切换核心栈和硬件上下文，由schedule()函数完成进程切换
- Windows线程的上下文切换
  - 上下文：CPU的寄存器组、线程环境块、核心栈和用户栈

2.1.2.4 进程地址空间布局(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

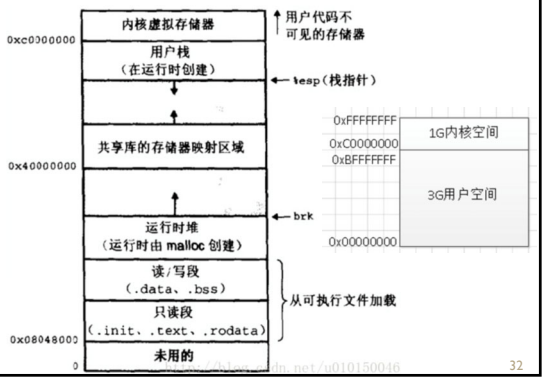


2.1.2.4 进程地址空间布局(2)

- Linux进程使用虚拟地址空间，由低地址到高地址分别为
  - 只读段：包括代码段、rodata 段(C常量字符串和#define定义的常量))
  - 数据段：保存全局变量、静态变量的空间
  - 堆：动态内存（malloc/new分配）
  - 文件映射区域：如动态库、共享内存等映射物理空间的内存，一般是 mmap 函数所分配的虚拟地址空间
  - 栈：用于维护函数调用的上下文空间，一般为 8M
  - 内核虚拟空间：用户代码不可见的内存区域，由内核管理(页表就存放在内核虚拟空间)

31

2.1.2.4 进程地址空间布局(3)



32

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

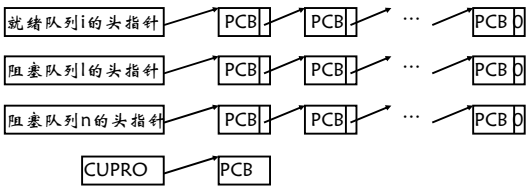
PCB集合　　线性表

PCB(0)	PCB(1)	PCB(2)	…….	PCB(n-2)	PCB(n-1)

33

2.1.2.5 进程的组织(2)

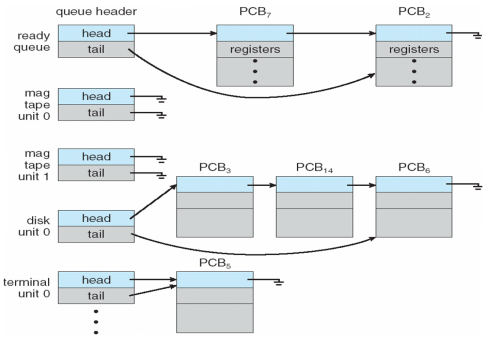
- 链接方式
  - 将处于同一状态的进程按照一定方式链接成一个队列



进程队列

34

2.1.2.5 进程的组织(3)



就绪队列和各种I/O设备队列

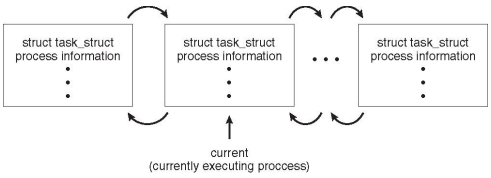
35

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

36

2.1.2.5 进程的组织(5)



Linux进程队列

37

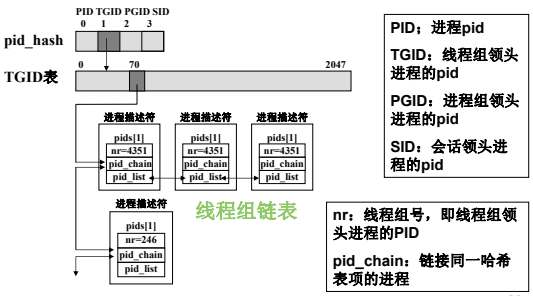
2.1.2.5 进程的组织(6)

- 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

38

2.1.2.5 进程的组织(7)

- PIDTYPE_TGID哈希链表



PID：进程pid

TGID：线程组领头进程的pid

PGID：进程组领头进程的pid

SID：会话领头进程的pid

nr：线程组号，即线程组领头进程的PID

pid_chain：链接同一哈希表项的进程

39

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

40

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、撤消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

41

2.1.3.2 创建原语(1)

- 创建进程的时机
  - 批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
  - 提供服务：打印进程
  - 应用请求：已存在的进程创建子进程

系统内核创建

应用程序创建

- 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

42

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

43

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()系统调用创建子进程
  - 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

44

2.1.3.2 创建原语(4)

- 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
- Linux的进程创建—clone()与vfork()
  - 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

45

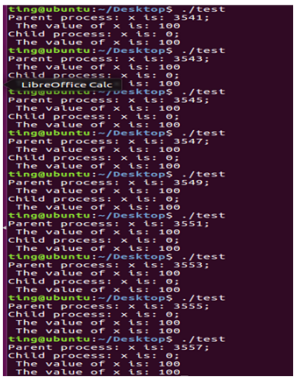
2.1.3.2 创建原语(5)

```
#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);
        }
}
```

46

2.1.3.2 创建原语(6)

- 运行结果



47

2.1.3.2 创建原语(7)

- Windows的进程创建—CreateProcess()
  - 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

48

2.1.3.2 创建原语(8)

- 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
```

49

2.1.3.2 创建原语(9)

```
                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);
}
```

50

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

51

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

52

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

53

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

54

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

55

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/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

56

2.1.3.5 唤醒原语(2)

    - 返回被中断进程继续执行，或者转进程调度，选择一个就绪进程投入运行
  - 期待的事件是等待某进程发送消息
    - 当信息发送给该等待进程时，由发送进程把该等待进程唤醒，并将其由阻塞态置为就绪态，插入就绪队列即可

57

2.2 线程的引入与概念

2.2.1 线程的引入
2.2.2 线程的概念
2.2.3 线程的实现
2.2.4 多线程模型
2.2.5 线程库
2.2.6 线程实例
2.2.7 进程与线程的比较

58

2.2.1 线程的引入(1)

- 原因
  - 在许多应用中同时发生着许多活动
  - 硬件技术的发展，使得计算机拥有了更多的CPU核，程序的可并行度提高
  - 创建进程的开销较大
  - 进程拥有独立的虚拟地址空间，在进程间进行数据共享和同步的开销较大
- 解决方法
  - 将拥有资源的基本单位与调度的基本单位分离

59

2.2.1 线程的引入(2)

- Benefits
  - Responsiveness
    - may allow continued execution if part of process is blocked, especially important for user interfaces
  - Resource Sharing
    - threads share resources of process, easier than shared memory or message passing
  - Economy
    - cheaper than process creation, thread switching lower overhead than context switching
  - Scalability
    - can take advantage of multicore architectures

60

2.2.2 线程的概念(1)

- 线程的概念
  - 进程内的一个执行单元
  - 进程内的一个可调度实体
  - 线程是程序（或进程）中相对独立的一个控制流序列
  - 线程是执行的上下文，其含义是执行的现场数据和其他调度所需的信息
  - 轻质进程(Light weight process)
    - 重质进程(Heavy weight process)
      - 传统的进程是拥有一个线程的进程

61

2.2.2 线程的概念(2)

- 线程的特性
  - 独立调度和分派的基本单位
  - 可并发（并行）执行
  - 动态性
    - 线程的状态
      - 就绪态
      - 运行态
      - 阻塞态/等待态
  - 结构性（线程控制块）
    - 线程的标识

62

2.2.2 线程的概念(3)

- 线程状态
- 程序计数器
- 执行堆栈
- 寄存器集
- 所有线程具有相同的地址空间（进程地址空间），共享进程资源
  - 地址空间
  - 代码段
  - 数据段
  - 其他OS资源

63

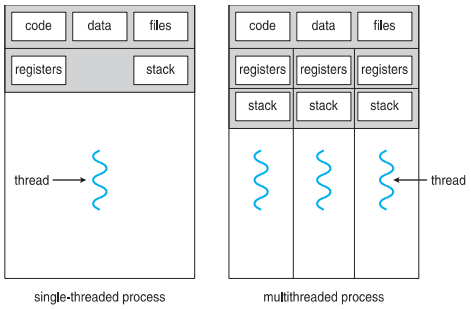
2.2.2 线程的概念(4)

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

进程结构与线程结构

64

2.2.2 线程的概念(5)

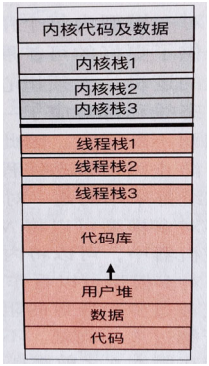


单线程进程与多线程进程

65

2.2.2 线程的概念(6)

- 多线程进程的地址空间布局



66

2.2.2 线程的概念(7)

- 线程的创建
  - 创建进程时，系统同时为进程创建第一个线程，即“初始化线程”
  - 由初始化线程根据需要再去创建若干个线程
- 线程的终止
  - 线程完成工作后自愿退出
  - 线程在运行中出现错误或由于某种原因而被其它线程强行终止

67

2.2.2 线程的概念(8)

- Linux
  - 创建内核线程的函数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

68

2.2.2 线程的概念(9)

- 2号进程是由1号进程init创建的内核线程，所有由内核创建和管理的进程都是由它fork出来的。

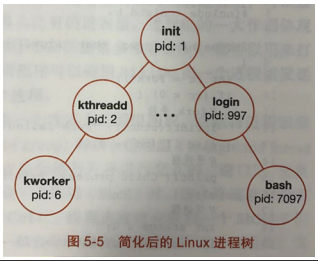


图 5-5 简化后的 Linux 进程树

69

2.2.3 线程的实现(1)

- 内核级线程
  - 所有线程的创建、撤销、调度和管理都由OS依靠内核负责
  - 在内核空间为每一个内核支持线程设置了一个线程控制块，内核根据线程控制块感知线程的存在，并对其加以控制
  - 缺点
    - 创建和管理的开销大

70

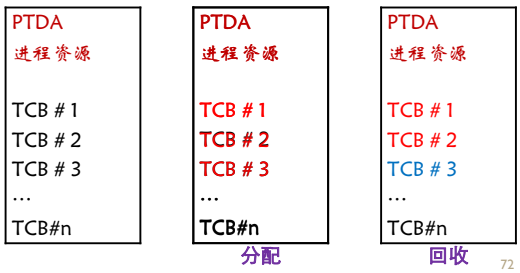
2.2.3 线程的实现(2)

- 优点
  - 可调度一个进程中的多个线程同时在多个计算单元上并行运行，从而提高程序执行速度和效率
  - 当进程中的一个线程被阻塞时，进程中的其他线程仍可以被调度运行
  - 具有很小的数据结构和堆栈，线程切换较快，开销较小

71

2.2.3 线程的实现(3)

- 实现
  - 创建进程时分配任务数据区空间



72

2.2.3 线程的实现(4)

- 用户级线程
  - 由用户应用程序建立的线程，并且由用户应用程序负责所有这些用户级线程的调度执行和管理工作
  - 仅存在于用户空间，操作系统内核完全不知道这些线程的存在
  - 优点
    - 用户应用程序中的线程切换的开销比内核线程切换的开销小
    - 线程库的线程调度算法与操作系统的调度算法无关

73

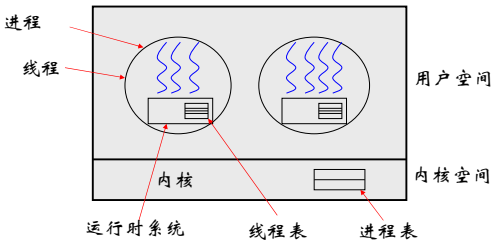
2.2.3 线程的实现(5)

- 用户级线程不要求内核支持，可适用于任何操作系统
- 缺点
  - 任一线程的阻塞将导致进程中所有其他线程被阻塞
  - 只能有一个线程在CPU上运行，不能利用多核的好处

74

2.2.3 线程的实现(6)

- 实现
  - 运行时系统



75

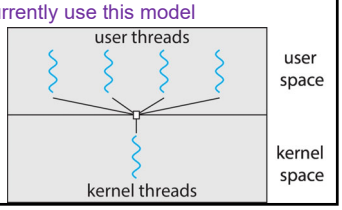
2.2.4 多线程模型(1)

- Many-to-One
- One-to-One
- Many-to-Many

76

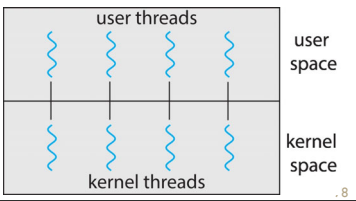
2.2.4 多线程模型(2)

- Many-to-One
  - Many user-level threads mapped to single kernel thread
  - One thread blocking causes all to block
  - Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
  - Few systems currently use this model
  - Examples:
    - Solaris Green Threads
    - GNU Portable Threads



2.2.4 多线程模型(3)

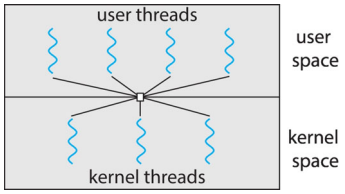
- One-to-One
  - Each user-level thread maps to kernel thread
  - Creating a user-level thread creates a kernel thread
  - More concurrency than many-to-one
  - Number of threads per process sometimes restricted due to overhead
  - Examples:
    - Windows
    - Linux



8

2.2.4 多线程模型(4)

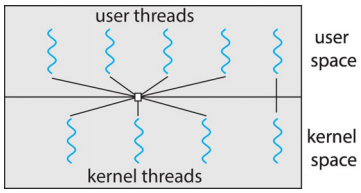
- Many-to-Many
  - Allows many user level threads to be mapped to many kernel threads
  - Allows the operating system to create a sufficient number of kernel threads
  - Windows with the ThreadFiber package
  - Otherwise not very common



79

2.2.4 多线程模型(5)

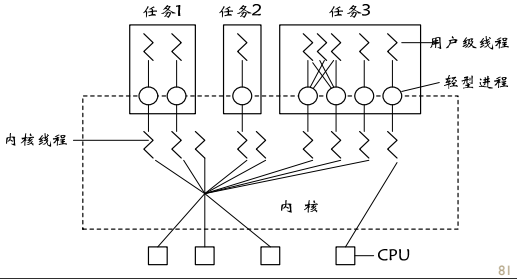
- Two-level Model
  - Similar to M:M, except that it allows a user thread to be bound to kernel thread



80

2.2.4 多线程模型(6)

- 内核控制线程(轻型进程LWP)
  - 用户级线程通过LWP与内核通信



81

2.2.5 线程库(1)

- 线程库为程序员提供创建和管理线程的API
- 实现方式
  - 在用户空间提供没有内核支持的库
    - 所有代码和数据存在于用户空间
    - 用户空间的本地函数调用
  - 由操作系统直接支持的内核级的库
    - 所有代码和数据存在于内核空间
    - 对内核的系统调用

82

2.2.5 线程库(2)

- 两种主要的线程库
  - POSIX Pthread
    - 可以提供用户级或内核级的线程库
  - Win32
    - 适用于Windows系统的内核级线程库
- 利用线程库创建线程
  - 示例：设计一个多线程程序，在独立的线程中完成非负数整数的加法功能

83

2.2.5 线程库(3)

- 使用Pthread API的多线程C程序

```
#include <pthread.h>
#include <stdio.h>
int sum;     //this data is shared by the thread(s)
void* runner(void *param);        //the thread
int main(int argc, char* argv[]) {
    pthread_t tid;
    pthread_attr_t attr;
    if(argc!=2) {
       fprintf(stderr,"usage:a.out <integer value>\n");
       return -1;
    }
    if(atoi(argv[1]<0) {
       fprintf(stderr,"%d must be<=0\n",atoi(argv[1]));
       return -1;   }
```

84

2.2.5 线程库(4)

```
    //get the default attributes
    pthread_attr_init(&attr);
    //create the thread
    pthread_create(&tid,&attr,runner,argv[1]);
    //now wait for the thread to exit
    pthread_join(tid,NULL);
    printf("sum=%d\n",sum);
}
//the thread will begin control in this function
void* runner(void* param) {
    int i,upper=atoi(param);
    sum=0;
    for(i=1;i<=upper;i++)
        sum+=i;
    pthread_exit(0);
}
```

85

2.2.5 线程库(5)

- 使用Win32 API的多线程C程序

```
#include <windows.h>
#include <stdio.h>
DWORD sum;    //this data is shared by the thread(s)
//the thread runs in this separate function
DWORD WINAPI Summation(LPVOID Param) {
    DWORD Upper=*(DWORD*)Param;
    for(DWORD i=0;i<=Upper;i++)
        sum+=i;
    return 0;
}
int main(int argc, char* argv[]) {
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;
```

86

2.2.5 线程库(6)

```
    if(argc!=2) {
        fprintf(stderr,"An integer param is required\n");
        return -1;
    }
    Param=atoi(argv[1]);
    if(Param<0) {
        fprintf(stderr,"An integer>=0 is required\n");
        return -1;
    }
    //create the thread
    ThreadHandle=CreateThread(
        NULL,              //default security attributes
        0,                 //default stack size
        Summation,         //thread function
        &Param,            //parameter to thread function
        0,                 //default creation flags
        &ThreadId);        //returns the thread identifier
```

87

2.2.5 线程库(7)

```
if(ThreadHandle!=NULL)
{
    //now wait for the thread to finish
    WaitForSingleObject(ThreadHandle,INFINITE);

    //close the thread handle
    CloseHandle(ThreadHandle);

    printf("sum=%d\n",sum);
}
}
```

88

2.2.6 线程实例(1)

- Windows线程
  - Windows应用程序以独立进程方式运行，每个进程可包括一个或多个线程
  - Windows实现了一对一的映射，每个用户线程映射到相关的内核线程
  - Windows也提供了对fiber库的支持，该库提供了多对多模型
  - 每个Windows线程包括
    - 一个线程ID，用于唯一标识线程

89

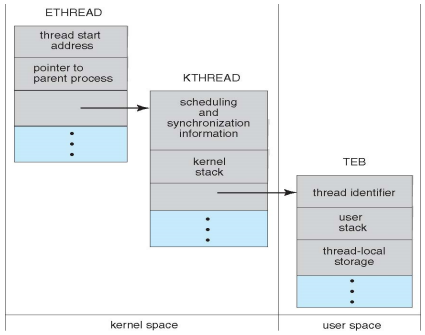
2.2.6 线程实例(2)

线程上下文
- 一组寄存器集合，表示处理器状态
- 一个用户栈和一个内核栈，分别供线程在用户模式和内核模式下运行
- 一个私有存储区域，为各种运行时库和动态链接库使用

- Windows线程的数据结构
  - ETHREAD：执行线程块 } 内核空间
  - KTHREAD：内核线程块 }
  - TEB：线程环境块 用户空间

90

2.2.6 线程实例(3)



Windows 线程的数据结构

91

2.2.6 线程实例(4)

- ETHREAD
  - 包括线程进程的指针和线程开始控制的子程序的地址，以及相应的KTHREAD
- KTHREAD
  - 包括线程的调度和同步信息，以及内核栈（线程在内核模式下运行时使用）和TEB
- TEB
  - 包括线程相关信息、用户模式栈和用于线程特定数据的数组

92

2.2.6 线程实例(5)

- 线程对象的服务
  - 创建线程CreateThread
  - 线程退出ExitThread
  - 终止某个线程TerminateThread
  - 改变线程优先级SetThreadPriority

➢ Linux系统中不区分进程与线程，对程序内的控制流通常称为任务(task)

- Linux的主进程数据结构中不包含进程的整个上下文，进程的文件系统上下文、文件描述表、信号处理表和虚拟内存上下文保存在独立的数据结构中

93

2.2.6 线程实例(6)

- Linux中创建进程与线程
  - fork()系统调用提供复制进程的传统功能
  - clone()系统调用提供创建线程的功能
    - 通过传递一组标志，决定在父任务和子任务之间发生多少共享

标　志	含　义
CLONE_FS	共享文件系统信息
CLONE_VM	共享内存空间
CLONE_SIGHAND	共享信号处理程序
CLONE_FILES	共享打开文件集合

94

2.2.7 进程与线程的比较(1)

- 拥有资源
  - 进程拥有独立的地址空间，若干代码段和数据段，若干打开文件、主存以及至少一个线程
  - 一个进程内的多线程共享该进程的所有资源，线程自己拥有很少资源
- 调度
  - 进程调度需进行进程上下文切换，开销大
  - 同一进程内的线程切换，仅交换线程拥有的一小部分资源，效率高；不同进程的线程切换将引起进程调度

95

2.2.7 进程与线程的比较(2)

- 并发性
  - 引入线程后，系统并发执行程度更高；进程之间、进程内的多线程之间可以并发执行
- 安全性
  - 同一进程的多个线程共享进程的所有资源，一个线程可以改变另一个线程的数据，而多进程实现则不会产生此问题

96

2.3 操作系统调度

2.3.1 基本概念
2.3.2 单核调度
2.3.3 实时系统调度
2.3.4 多核调度

97

2.3.1 基本概念(1)

- 什么是调度
  - Q1：程序员小明想在单核计算机上运行一个需要执行30分钟的机器学习程序，同时还想打开播放器收听音乐
    - 轮流执行
  - Q2：如果仅有8个处理器，如何运行168个任务？
    - 下一个要执行的任务是哪一个？
    - 执行该任务的CPU是哪一个？
    - 每个任务执行多长时间？

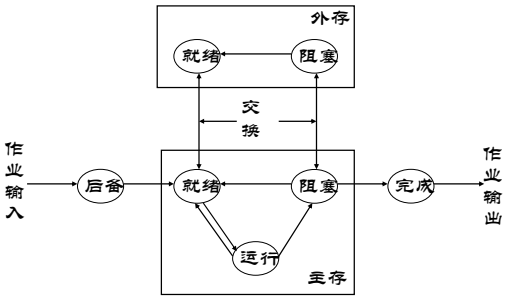
98

2.3.1 基本概念(2)

- 处理机调度级别
  - 高级调度（作业调度）
    - 选择哪个作业进入就绪队列
    - 调度不频繁
  - 低级调度（进程调度）
    - 选择哪个进程可以占有CPU
    - 调度频繁
  - 中级调度（交换调度）
    - 进程换入换出
    - 提高内存利用率和系统吞吐量

99

2.3.1 基本概念(3)



作业状态及其转换

100

2.3.1 基本概念(4)

- 调度方式
  - 非剥夺方式
    - 实现简单，系统开销小，适合于批处理系统
  - 剥夺方式
    - 优先级/时间片，分时系统/实时系统
- 进程调度的功能
  - 记录系统中各进程的执行状况
  - 选择进程真正占有CPU
  - 进行进程上下文的切换

101

2.3.1 基本概念(5)

- 调度时机
  - 现行进程完成执行或由于某种错误而中止运行
  - 正在执行的进程提出I/O请求，等待I/O完成
  - 分时系统中按照时间片轮转调度策略，分配给进程的时间片用完
  - 优先级调度策略中，进程有更高优先级进程变为就绪
  - 进程执行了某种操作原语，如阻塞原语或唤醒原语时，可能引起进程调度

102

2.3.1 基本概念(6)

- 处理机调度准则
  - CPU利用率高
  - 吞吐量高
  - 周转时间/平均周转时间短
  - 等待时间/平均等待时间短
  - 响应时间短
  - 批处理系统：增加系统吞吐量和提高系统资源的利用率
  - 分时系统：保证每个分时用户的响应时间

103

2.3.2 单核调度

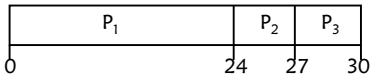
2.3.2.1 FCFS调度算法
2.3.2.2 SJF调度算法
2.3.2.3 高响应比优先调度算法
2.3.2.4 优先级调度算法
2.3.2.5 时间片轮转调度算法
2.3.2.6 多级队列调度算法
2.3.2.7 多级反馈队列调度算法

104

2.3.2.1 FCFS调度算法(1)

- 采用非剥夺调度方式
- 既可用于作业调度，也可用于进程调度
- 示例：3个进程的到达顺序及运行时间为

进程：	P_1	P_2	P_3
运行时间：	24	3	3



等待时间： $P_1 = 0; P_2 = 24; P_3 = 27$

平均等待时间： $(0 + 24 + 27)/3 = 17$

105

2.3.2.1 FCFS调度算法(2)

- 若进程到达顺序为P_2，P_3，P_1，则

等待时间：$P_1 = 6; P_2 = 0; P_3 = 3$

平均等待时间：$(6 + 0 + 3)/3 = 3$

- 简单，但效率不高
- 有利于长作业（进程），不利于短作业（进程），容易被大作业（进程）垄断，使得平均等待时间很长

106

2.3.2.2 SJF调度算法(1)

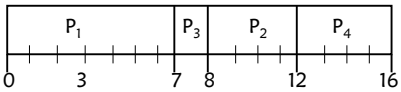
- 考虑作业（进程）运行时间
- 在长期调度中频繁使用
- 既可采用非剥夺调度方式，也可采用剥夺调度方式（Shortest-Remaining-Time-First，SRTF）
- 示例：4个进程的到达时间及运行时间为

进程	到达时间	运行时间
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

107

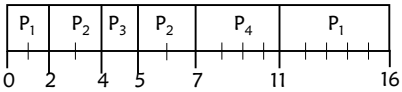
2.3.2.2 SJF调度算法(2)

- 非剥夺方式



平均等待时间 $(0 + 6 + 3 + 7)/4 = 4$

- 剥夺方式



平均等待时间 $(9 + 1 + 0 + 2)/4 = 3$

108

2.3.2.2 SJF调度算法(3)

- 优点
  - SJF是一种最优算法，它可以获得最小平均等待时间，提高系统吞吐量
- 缺点
  - 对长作业不利，容易产生“饥饿”现象
  - 未考虑作业的紧迫程度
- 困难
  - 难以确定进程的执行时间

109

2.3.2.3 高响应比优先调度算法(1)

- 优点
  - 兼顾了运行时间短和等待时间长的作业（结合FCFS和SJF方法），优先运行短作业和等待时间足够长的长作业
- 缺点
  - 较复杂，系统开销大

$$\text{响应比}=\frac{\text{作业等待时间}+\text{作业估计运行时间}}{\text{作业估计运行时间}}$$
$$=1+\frac{\text{作业等待时间}}{\text{作业估计运行时间}}$$

110

2.3.2.3 高响应比优先调度算法(2)

- 响应比计算
  - 若作业等待时间相同，则作业估计运行时间越短，响应比越高，因此有利于短作业
  - 若作业估计运行时间相同，则作业等待时间越长，响应比越高，此时算法即FCFS
  - 对长作业来说，其响应比将随等待时间的增加而提高，当等待时间足够长时，响应比将足够高，从而可以获得处理机，避免产生“饥饿”现象

111

2.3.2.4 优先级调度算法(1)

- 为每个进程设定一个优先级
- 既可采用非剥夺调度方式(批处理系统)
- 也可采用剥夺调度方式（实时系统）
- 优先级设定
  - 进程类型：系统进程/用户进程
  - 进程对资源的需求
    - 申请资源较多的进程，优先级较低
  - 用户要求
- 优先级与优先数

112

2.3.2.4 优先级调度算法(2)

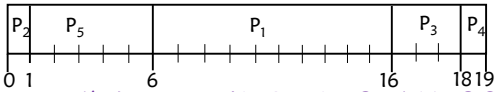
- 优先级类型
  - 静态优先级
    - 进程创建时确定，整个运行期间保持不变
    - 不能反映进程特点，调度性能差
    - 容易导致“饥饿”，即不能调度低优先级进程
  - 动态优先级
    - 随着进程的推进或等待时间的增加而改变

113

2.3.2.4 优先级调度算法(3)

- 示例：5个进程的运行时间及优先级如下

进程	运行时间	优先数
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



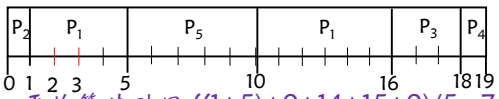
平均等待时间：(6+0+16+18+1)/5=8.2

114

2.3.2.4 优先级调度算法(4)

- 示例：5个进程的运行时间及优先级如下

进程	到达时间	运行时间	优先数
P1	0	10	3
P2	0	1	1
P3	2	2	4
P4	3	1	5
P5	5	5	2



平均等待时间 ((1+5)+0+14+15+0)/5=7

115

2.3.2.5 时间片轮转调度算法(1)

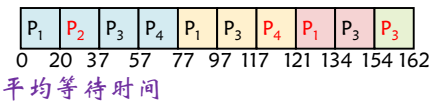
- 用于分时系统
- 采用剥夺调度方式
- 时间片的确定
  - 既要保证系统各个用户进程及时地得到响应，又不要由于时间片太短而增加调度的开销，降低系统的效率

116

2.3.2.5 时间片轮转调度算法(2)

- 示例：时间片为20，4个进程运行时间为

进程	运行时间
P1	53
P2	17
P3	68
P4	24



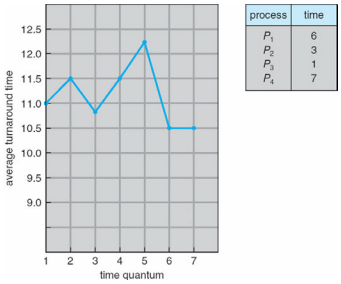
平均等待时间

((57+24)+20+(37+40+17)+(57+40))/4=73

((134-53-0)+(37-17-0)+(162-68-0)+(121-24-0))/4=73

117

2.3.2.5 时间片轮转调度算法(3)



process	time
P_1	6
P_2	3
P_3	1
P_4	7

周转时间随时间片大小变换

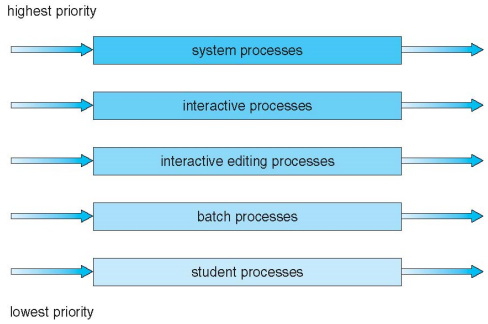
2.3.2.6 多级队列调度算法(1)

- 综合考虑各种类型进程的需要
  - 终端型作业用户
    - 交互性好，响应时间短
  - 短批处理作业用户
    - 周转时间短，在较短时间内完成
  - 长批处理作业用户
    - 不会出现长期得不到处理的“饥饿”现象

2.3.2.6 多级队列调度算法(2)

- 将就绪队列分成多个独立队列
- 根据进程的属性（内存大小、进程优先级、进程类型），一个进程被永久地分配到一个队列
- 每个队列有自己的调度算法
- 队列之间进行调度
  - 固定优先级抢占调度

2.3.2.6 多级队列调度算法(3)



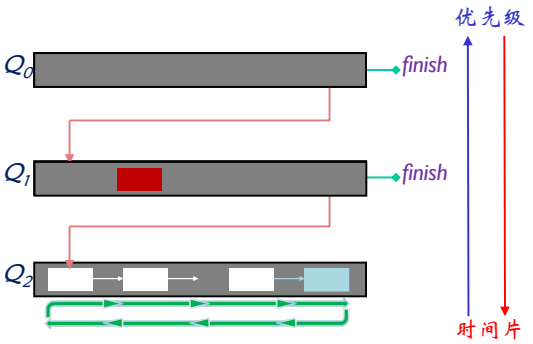
121

2.3.2.7 多级反馈队列调度算法(1)

- 设置多个就绪队列，并为其赋予不同的优先级和时间片，高优先级队列中进程的时间片较小
- 新进程放入第一队列尾，按FCFS算法调度
- 若该进程在指定时间片内未完成，调度程序将该进程转入第二队列尾，按FCFS算法调度，依次类推
- 仅当第一队列空时，调度程序才调度第二队列中的进程运行，依次类推
- 当处理机为某级队列服务时，有高优先级进程进入，则采用剥夺方式为高优先级进程服务

122

2.3.2.7 多级反馈队列调度算法(2)



123

2.3.2.7 多级反馈队列调度算法(3)

- 调度时考虑的问题
  - 队列数目
  - 每个队列采用的调度算法
  - 初始状态时，每个进程处于哪个队列

124

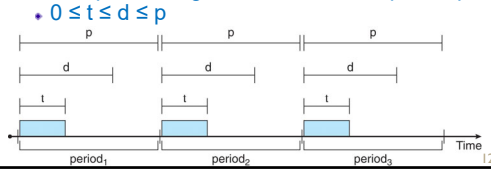
2.3.3 实时系统调度(1)

- 2.3.3.1 速率单调调度
- 2.3.3.2 EDF调度算法
- 2.3.3.3 LLF调度算法

125

2.3.3 实时系统调度(2)

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics:
  - periodic ones require CPU at constant intervals
    - Has processing time t, deadline d, period p
    - $0 \le t \le d \le p$

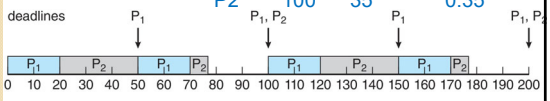


126

2. 3. 3. 1 速率单调调度(1)

- A priority is assigned based on the inverse of its period
  - Rate of periodic task is 1/p
- Example1:

Process	p	t	CPU利用率 t/p
P1	50	20	0.4
P2	100	35	0.35



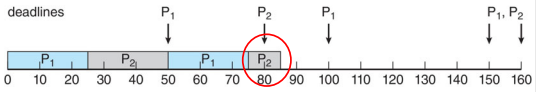
Rate-monotonic scheduling.

127

2. 3. 3. 1 速率单调调度(2)

- Example2:

Process	p	t	CPU利用率t/p
P1	50	25	0.5
P2	80	35	0.4375

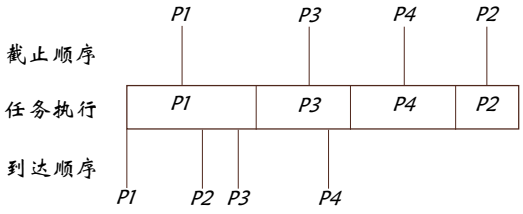


Missing deadlines with rate-monotonic scheduling.

128

2. 3. 3. 2 EDF调度算法(1)

- 最早截止时间优先（Earliest Deadline First）
- 开始截止时间越早，优先级越高

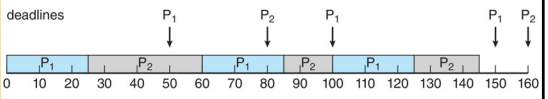


129

2.3.3.2 EDF调度算法(2)

- Example2:

Process	p	t	CPU利用率t/p
P1	50	25	0.5
P2	80	35	0.4375



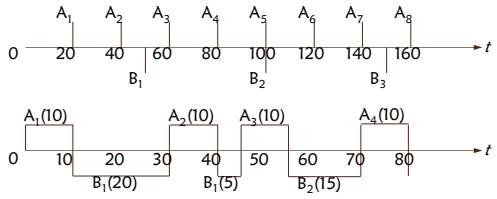
Earliest-deadline-first scheduling.

130

2.3.3.3 LLF调度算法

- 最小松弛度优先（Least Laxity First）
- 松弛度 = 结束时间 - 运行时间 - 当前时间
- 示例：

Process	p	t
A	20	10
B	50	25



131

2.3.4 多核调度(1)

- 多核调度
  - 任务同时在多个CPU核心上并行执行
  - 调度器的考虑
    - 当前应该调度哪个/哪些任务？
    - 每个调度的任务应该在哪个CPU核心上执行？
    - 每个调度的任务应该执行多长时间？

132

2.3.4 多核调度(2)

- 负载分担
  - 沿用单核调度思路，多核共享一个全局运行队列
  - 优点
    - 设计实现简单
    - 每个CPU核心分担系统负载，不会造成CPU的浪费
  - 缺点
    - 共享全局队列的同步开销
    - 任务在多个CPU核心之间切换开销

133

2.3.4 多核调度(3)

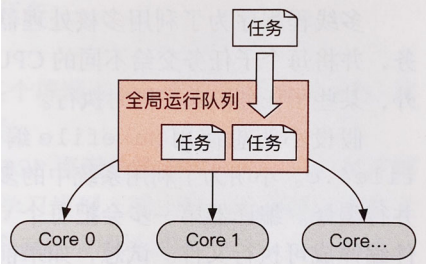


图 6-26 基于负载分担的多核调度示意图

134

2.3.4 多核调度(4)

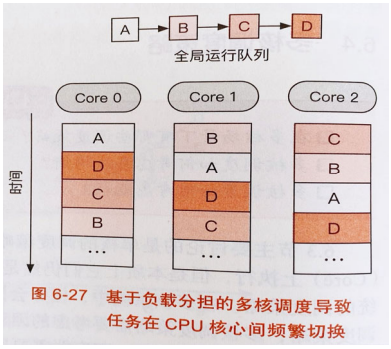


图 6-27 基于负载分担的多核调度导致任务在 CPU 核心间频繁切换

135

2.3.4 多核调度(5)

- 协同调度
  - 适用于关联任务或任务之间有依赖关系的场景/并行计算场景
    - 将一个工作量较大的任务切分成多个子任务，每个子任务由不同CPU核心完成
    - 尽可能让一组任务并行执行，避免调度器同时调度有依赖关系的两组任务
  - 群组调度策略

136

2.3.4 多核调度(6)

  - 将关联任务设置为一组，以组为单位调度任务在多个CPU核心上执行
  - 缺点
    - 无关联任务之间的相互等待可能造成CPU资源的浪费
- 两级调度
  - 全局调度器：决定任务被哪个CPU核心执行
  - 本地调度器：任务一直被该核心调用，不会发生迁移

137

2.3.4 多核调度(7)

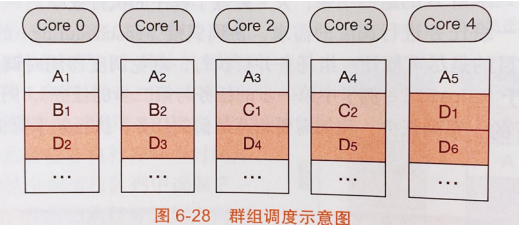


图 6-28 群组调度示意图

138

2.3.4 多核调度(8)

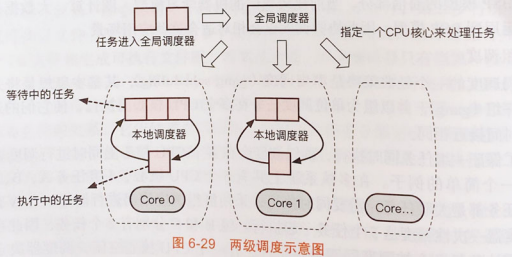


图 6-29 两级调度示意图

本章要点(1)

- 进程的概念与特点
- 进程控制块的作用及内容
- 进程的状态与状态转换
- 进程上下文切换
- 进程的内存空间布局
- 进程的组织
- 进程的控制
- 线程的概念与实现
- 多线程模型

本章要点(2)

- 先来先服务调度算法
- 短作业优先调度算法
- 高响应比优先调度算法
- 时间片轮转调度算法
- 优先级调度算法
- 多级反馈队列调度算法
- 实时系统调度算法
- 多核调度算法