

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

64

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

65

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

66

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

67

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

68

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

69

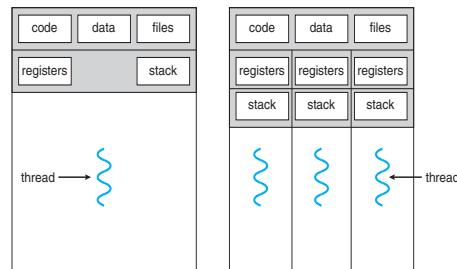
2.2.2 线程的概念(4)

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

70

2.2.2 线程的概念(5)



单线程进程与多线程进程

71

2.2.2 线程的概念(6)

➤ 多线程进程的地址空间布局



72

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

73

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，并对其进行控制
- 缺点
 - 创建和管理的开销大

74

2.2.3 线程的实现(2)

• 优点

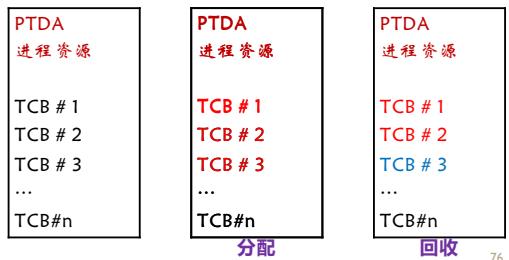
- 可调度一个进程中的多个线程同时在多个计算单元上并行运行，从而提高程序执行速度和效率
- 当进程中的一个线程被阻塞时，进程中的其他线程仍可以被调度运行
- 具有很小的数据结构和堆栈，线程切换较快，开销较小

75

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



76

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

77

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

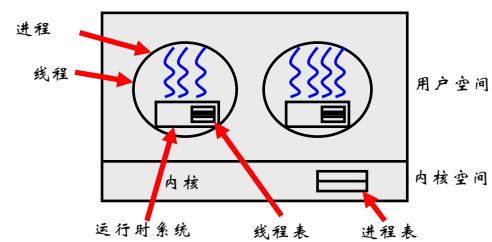
- 任一线程的阻塞将导致进程中所有其他线程被阻塞
- 只能有一个线程在CPU上运行，不能利用多核的好处

78

2.2.3 线程的实现(6)

- 实现

- 运行时系统



79

2.2.4 多线程模型(1)

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

80

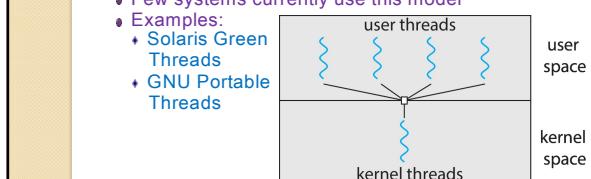
2.2.4 多线程模型(2)

- 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



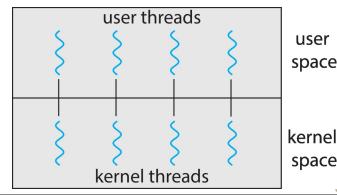
2.2.4 多线程模型(3)

➤ 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

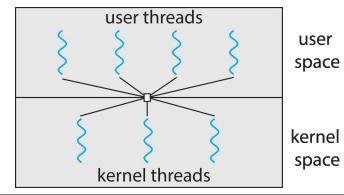


82

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

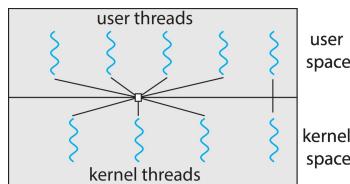


83

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

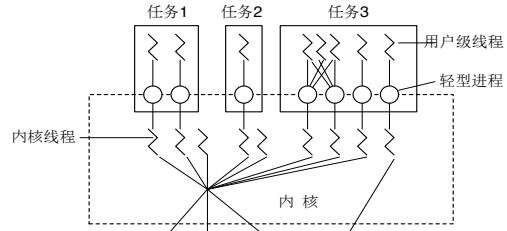


84

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



85

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

86

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

87

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;
    }
```

88

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);
}
```

89

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;
```

90

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
```

91

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);
}
```

92

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

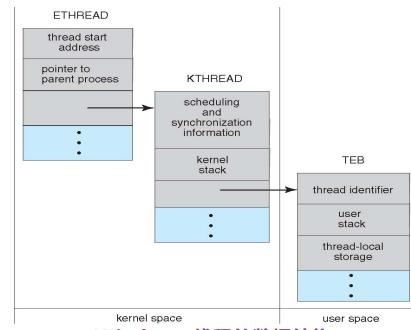
93

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
 - KTHREAD：内核线程块
 - TEB：线程环境块

94

2.2.6 线程实例(3)



95

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

96

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

97

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，仅交换线程拥有的小部分资源，效率高；不同进程的线程切换将引起进程调度

98

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

99