

第4章 线程

孙承杰

E-mail: sunchengjie@hit.edu.cn

哈工大计算学部人工智能教研室

2023年秋季学期

主要内容

4.1 线程的引入

4.2 多线程模型

多对一模型、一对一模型、多对多模型

4.3 用户线程与内核线程的比较

4.4 线程库

POSIX Pthread线程库、Win32线程库、Java线程库

4.5 操作系统进程/线程模型分类

多线程/多任务实例

4.6 协程（用户态轻量级线程）

4.1 线程的引入

回顾并讨论进程的相关问题：

- (1)进程生命周期中，核心任务是竞争CPU
- (2)一个进程只能有一个代码执行序列吗？
- (3)若进程遇到了I/O等待，不得不放弃CPU
- (4)代码序列中是否有可以并行处理的程序段？

代码例: 完全可以并行执行的代码

```
#include <stdio.h>
long m, n;
int main(void)
{
    int i;
    int MMM;
    printf("Input MMM = ");
    scanf("%d", &MMM);

    /*计算1 -- MMM 整数累加和 == > m */
    m = 0;
    for (i = 1; i <= MMM; i++)
    {
        m = m + i;
    }
    printf("Sum from 1 to %d = %d\n", MMM, m);

    /*计算MMM -- MMM*2 整数累加和 == > n */
    n = 0;
    for (i = MMM + 1; i <= MMM * 2; i++)
    {
        n = n + i;
    }
    printf("Sum from %d to %d = %d\n", MMM + 1, 2 * MMM, n);

    printf("\nMain thread m+n = %ld\n\n", m + n);
}
```

4.1 线程的引入

回顾 进程的描述与表达

进程控制块PCB的内容：

不同操作系统中，PCB的内容不尽相同，但通常包括：

- (1) 进程标识符 - 确认进程的唯一标识(PID)
- (2) 进程当前状态 - 进程调度程序分配处理机的依据
- (3) 进程队列指针 - 记录PCB链表中下一个PCB的地址
- (4) 程序地址范围 - 开始地址与结束地址 (地址空间)
- (5) 进程优先级 - 反映进程要求CPU的紧迫程度
- (6) CPU现场保护区 - 记录让出处理机时的CPU现场信息
- (7) 通信信息 - 记录与其他的进程的信息交换情况
- (8) 家族联系 - 记录父进程的PID
- (9) 占有资源清单 - 打开文件列表、所需资源及已分配资源清单

4.1 线程的引入

回顾 进程的描述与表达

表 3.5 进程控制块中的典型元素

进程标识信息	
标识符	<p>存储在进程控制块中的数字标识符，包括：</p> <ul style="list-style-type: none">● 此进程的标识符 (Process ID，简称进程 ID)● 创建这个进程的进程 (父进程) 标识符● 用户标识符 (User ID，简称用户 ID)
处理器状态信息	
用户可见寄存器	<p>用户可见寄存器是处于用户态的处理器执行的机器语言可以访问的寄存器。通常有 8 ~ 32 个此类寄存器，而在一些 RISC 实现中有超过 100 个此类寄存器</p>
控制和状态寄存器	<p>用于控制处理器操作的各种处理器寄存器，包括：</p> <ul style="list-style-type: none">● 程序计数器：包含将要取的下一条指令的地址● 条件码：最近的算术或逻辑运算的结果 (如符号、零、进位、等于、溢出)● 状态信息：包括中断允许/禁用标志，异常模式
栈指针	<p>每个进程有一个或多个与之相关联的后进先出 (LIFO) 系统栈。栈用于保存参数和过程调用或系统调用的地址，栈指针指向栈顶</p>
进程控制信息	
调度和状态信息	<p>这是操作系统执行其调度功能所需要的信息，典型的信息项包括：</p> <ul style="list-style-type: none">● 进程状态：定义将被调度执行的进程的准备情况 (如运行态、就绪态、等待态、停止态)● 优先级：用于描述进程调度优先级的一个或多个域。在某些系统中，需要多个值 (如默认、当前、最高许可)● 调度相关信息：这取决于所使用的调度算法。例如进程等待的时间总量和进程在上一次运行时执行时间总量● 事件：进程在继续执行前等待的事件标识

进程PCB

```
struct task_struct {
    long state;           //任务的运行状态 (-1 不可运行, 0 可运行(就绪), >0 已停止)。
    long counter;         // 任务运行时间计数(递减)(滴答数), 运行时间片。
    long priority;        // 运行优先数。任务开始运行时 counter=priority, 越大运行越长。
    long signal;          // 信号。是位图, 每个比特位代表一种信号, 信号值=位偏移值+1。
    struct sigaction sigaction[32]; // 信号执行属性结构, 对应信号将要执行的操作和标志信息。
    long blocked;         // 进程信号屏蔽码(对应信号位图)。
    int exit_code;        // 任务停止执行后的退出码, 其父进程会来取。
    unsigned long start_code; // 代码段地址。
    unsigned long end_code;   // 代码长度(字节数)。
    unsigned long end_data;   // 代码长度 + 数据长度(字节数)。
    unsigned long brk;        // 总长度(字节数)。
    unsigned long start_stack; // 堆栈段地址。
    long pid;               // 进程标识号(进程号)。
    long father;           // 父进程号。
    long pgrp;             // 进程组号。
    long session;          // 会话号。
    long leader;           // 会话首领。
    unsigned short uid;     // 用户标识号(用户 id)。
    unsigned short euid;    // 有效用户 id。
    unsigned short suid;    // 保存的用户 id。
    unsigned short gid;     // 组标识号(组 id)。
    unsigned short egid;    // 有效组 id。
    unsigned short sgid;    // 保存的组 id。
    long alarm;             // 报警定时值(滴答数)。
    long utime;             // 用户态运行时间(滴答数)。
    long stime;             // 系统态运行时间(滴答数)。
    long ctime;             // 子进程用户态运行时间。
    long cstime;            // 子进程系统态运行时间。
    long start_time;        // 进程开始运行时刻。
    unsigned short used_math; // 标志: 是否使用了协处理器。
    int tty;                // 进程使用 tty 终端的子设备号。-1 表示没有使用。
    unsigned short umask;   // 文件创建属性屏蔽位。
    struct m_inode * pwd;   // 当前工作目录 i 节点结构指针。
    struct m_inode * root;  // 根目录 i 节点结构指针。
    struct m_inode * executable; // 执行文件 i 节点结构指针。
    unsigned long close_on_exec; // 执行时关闭文件句柄位图标志。(参见 include/fcntl.h)
    struct file * filp[NR_OPEN]; // 文件结构指针表, 最多 32 项。表项号即是文件描述符的值。
    struct desc_struct ldt[3]; // 局部描述符表。0-空, 1-代码段 cs, 2-数据和堆栈段 ds&ss。
    struct tss_struct tss;   // 进程的任务状态段信息结构。
}
```

进程PCB

```
struct task_struct {  
    /*-----*/  
    long state;    // 进程运行状态 (-1 不可运行, 0 可运行, >0 以停止)   
    long counter;  // 任务运行时间片, 递减到 0 是说明时间片用完   
    long priority; // 任务运行优先数, 刚开始是 counter=priority   
    long signal;   // 任务的信号位图, 信号值=偏移+1   
    struct sigaction sigaction[32]; //信号执行属性结构, 对应信号将要执行的操作和标志信息   
    long blocked;  // 信号屏蔽码   
    /*----- various fields -----*/  
    int exit_code; // 任务退出码, 当任务结束时其父进程会读取   
    unsigned long start_code, end_code, end_data, brk, start_stack;   
        // start_code 代码段起始的线性地址   
        // end_code 代码段长度   
        // end_data 代码段长度+数据段长度   
        // brk 代码段长度+数据段长度+bss 段长度   
        // start_stack 堆栈段起始线性地址   
    long pid, father, pgrp, session, leader;   
        // pid 进程号, father 父进程号, pgrp 父进程组号, session 会话号, leader 会话首领   
    unsigned short uid, euid, suid;   
        // uid 用户标 id, euid 有效用户 id, suid 保存的用户 id   
    unsigned short gid, egid, sgid;   
        // gid 组 id, egid 有效组 id, sgid 保存组 id   
    long alarm; // 报警定时值
```


进程PCB

```
long alarm;    // 报警定时值↵
long utime, stime, cutime, cstime, start_time;↵
    // utime 用户态运行时间↵
    // stime 内核态运行时间↵
    // cutime 子进程用户态运行时间↵
    // cstime 子进程内核态运行时间↵
    // start_time 进程开始运行时刻↵
unsigned short used_math;    // 标志, 是否使用了 387 协处理器↵
/* ----- file system info ----- */↵
int tty;    // 进程使用 tty 的子设备号, -1 表示没有使用↵
unsigned short umask;    // 文件创建属性屏蔽码↵
struct m_inode * pwd;    // 当前工作目录的 i 节点↵
struct m_inode * root;    // 根目录的 i 节点↵
struct m_inode * executable;    // 可执行文件的 i 节点↵
struct file * filp[NR_OPEN];    // 进程使用的文件↵
unsigned long close_on_exec;    // 执行时关闭文件句柄位图标志↵
↵
/* ----- ldt for this task 0 - zero 1 - cs 2 - ds&ss ----- */↵
struct desc_struct ldt[3];    // 本任务的 ldt 表, 0-空, 1-代码段, 2-数据和堆栈段↵
/* ----- tss for this task ----- */↵
struct tss_struct tss;    // 本任务的 tss 段↵
};↵
```

4.1 线程的引入

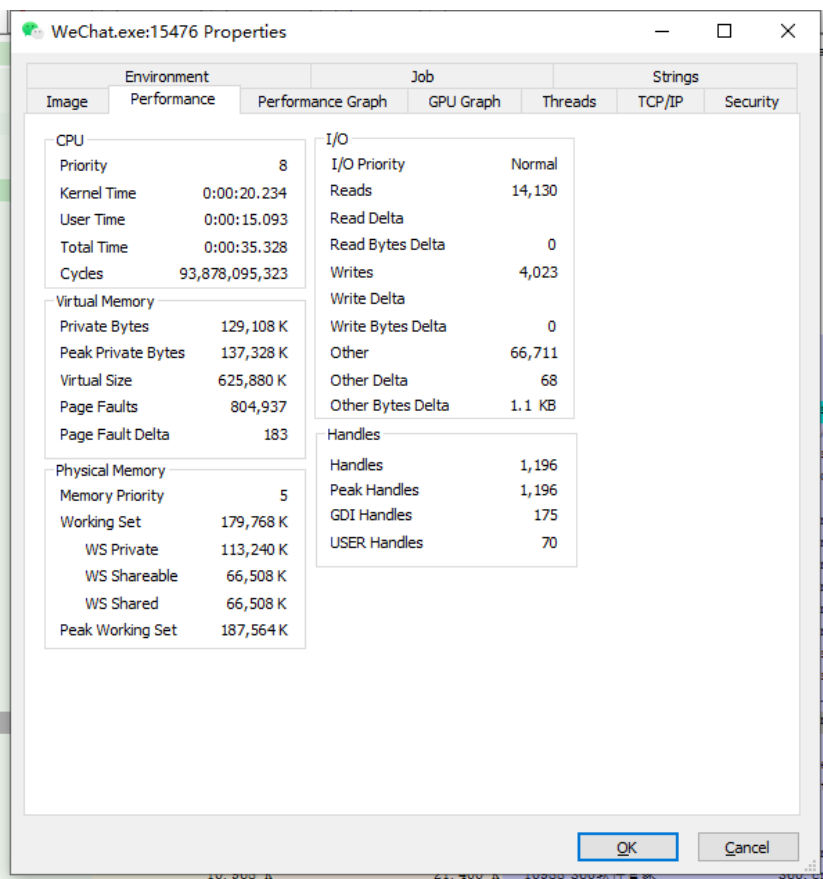
问题：

1) 多个进程协作完成任务，需要进程间进行通信，如果需要频繁通信？会带来什么问题？

——需要进程间通信，本质是系统调用，需要较多的时间和CPU资源开销。

2) 进程切换开销如何？考虑进程切换需要切换什么？

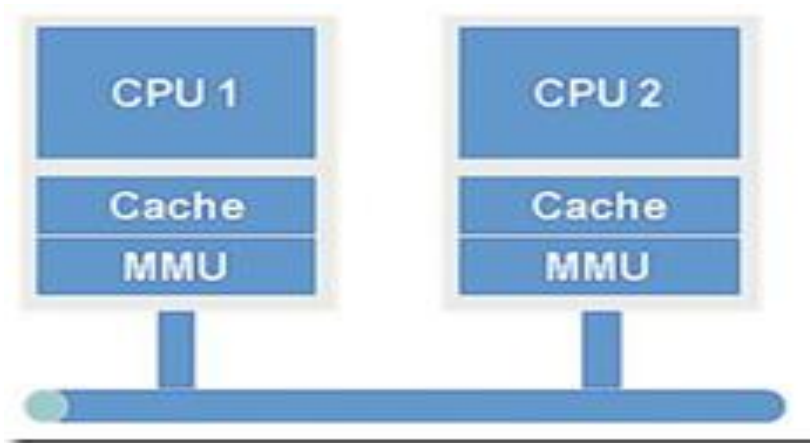
——资源、内存、寄存器



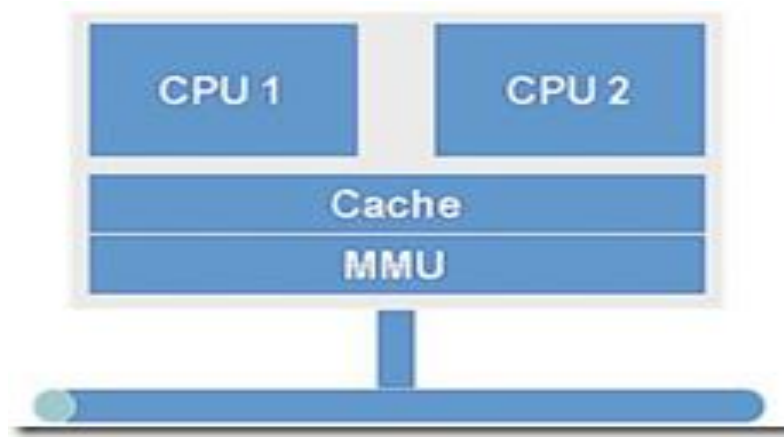
4.1 线程的引入

计算机硬件体系结构

多处理器



多核



4.1 线程的引入

再看看IE浏览器的工作过程

访问一个网页

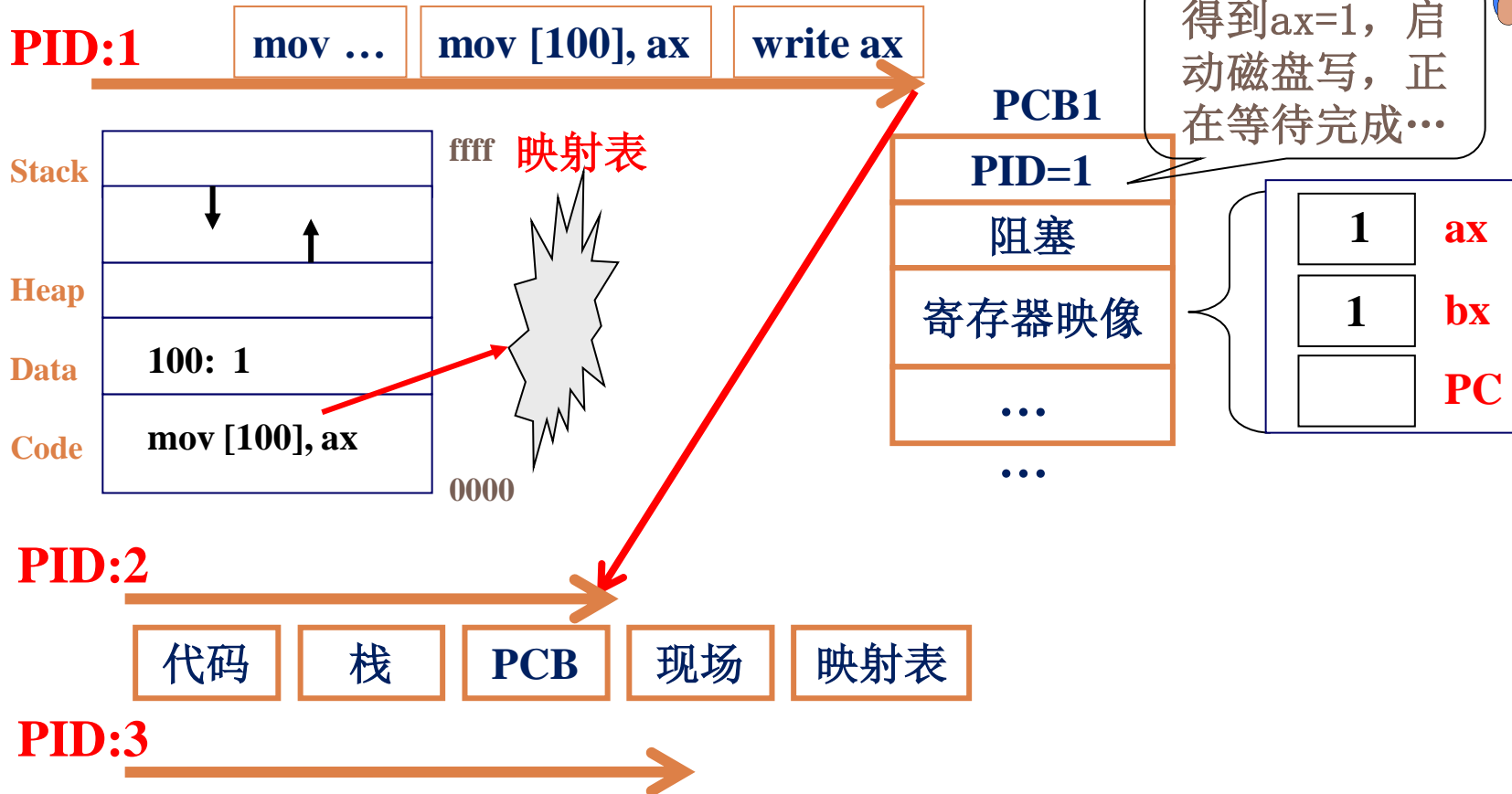
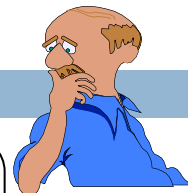
- 从Web服务器接收数据（主要网卡工作）
- 显示文本信息（普通Tag）
- 解压缩图片、音视频
- 显示图片、播放音视频



线程概念 – 更有效的并发思想

4.1 线程的引入

多进程是操作系统的基本特点



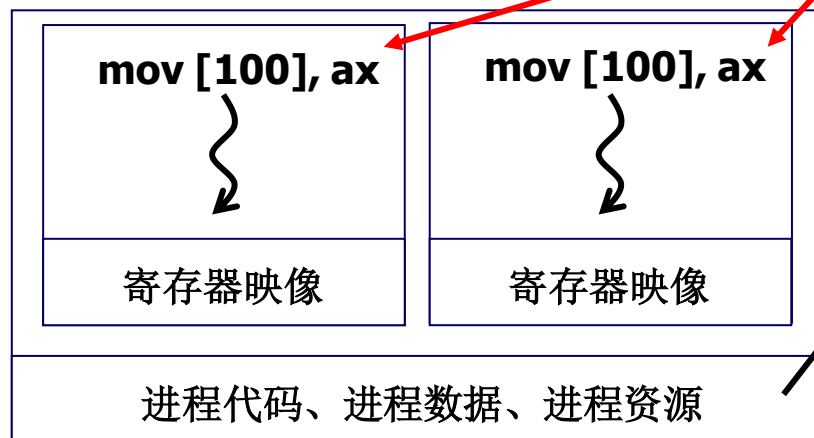
4.1 线程的引入

是否可以资源不动而切换指令序列?

■ 进程 = 资源 + 指令执行序列

- 将资源和指令执行分开
- 一套资源 + 多个指令执行序列

进程



线程

内存映射表

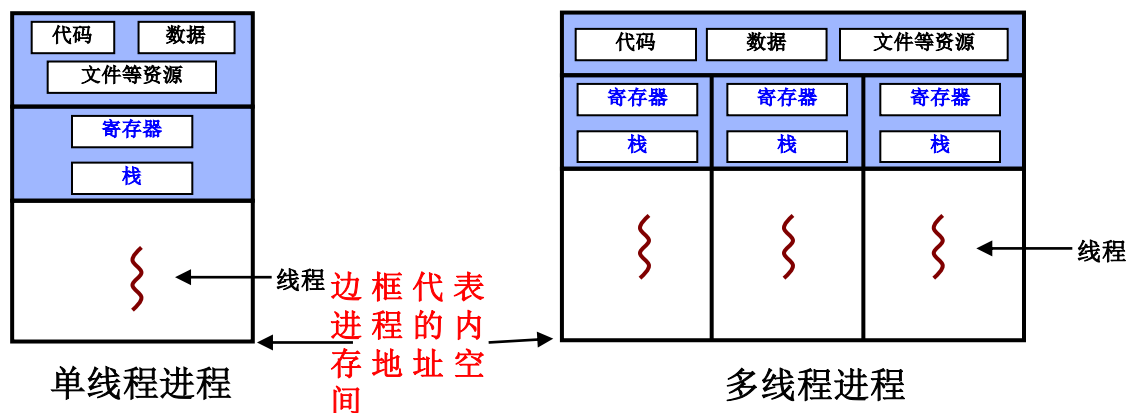
- 线程：保留了并发的优点，避免了进程切换代价
- 实质就是内存映射表与资源不变而PC指针和基本寄存器变

4.1 线程的引入

什么是线程？

- **线程**：进程中代码执行的一个序列
- 显然，一个进程中**至少**存在一个线程
- 从一个程序的逻辑功能角度，可以将其划分成**n个执行序列**
这些序列可以独立，可以有一定的关联关系
- 在一个进程的生命周期中，可以创建n个线程
- 线程是使用CPU的基本单元，由**线程ID**、**程序计数器PC**、**寄存器组**和**栈**构成

实质就是内存映射表与资源不变而PC指针和基本寄存器变



4.1 线程的引入

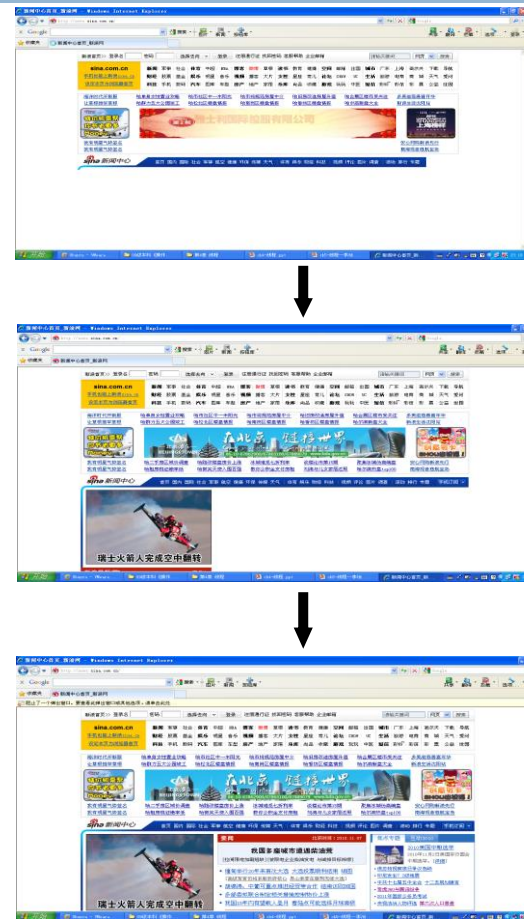
多个执行序列+一个地址空间是否实用?

一个网页浏览器

- 一个线程用来从服务器接收数据
- 一个线程用来显示文本
- 一个线程用来处理图片(如解压缩)
- 一个线程用来显示图片

这些线程共用一个地址空间吗?

- 同一个服务器建立连接并接收数据
- 一次请求所有文本、图片都显示在浏览器一页中



4.1 线程的引入

前面的例子使用子进程+IPC不可以吗？

一个网页浏览器程序（主进程）

- 一个子进程用来从服务器接收数据
- 一个子进程用来显示文本
- 一个子进程用来处理图片(如解压缩)
- 一个子进程用来显示图片

完全可以！但开销较大、控制过程繁琐！

- 每个进程必须有一个完整的映像（内储空间）
- 大量的重复存储：库函数、复用的过程、全局变量等
- 主进程协调子进程工作：只有通过频繁的IPC过程完成
- 子进程之间调度切换代价大：现场、栈、内存（页表）

4.1 线程的引入

□ 多线程编程的4个优点：

▣ 响应度高

- 交互程序采用多线程，即使部分阻塞，其他部分仍能够执行

▣ 资源共享

- 线程默认共享他们所属进程的所有资源

▣ 经济快捷：

- (1) 多进程比多线程消耗资源更多；
- (2) 线程切换只需切换少量现场（寄存器、栈），速度快很多

▣ 方便多CPU处理

- 由用户（程序员）编写，能够充分考虑任务的 有效分离（多序列），同时执行序列隶属于同一个进程的地址空间，仍然可以充分利用多CPU处理

4.2 多线程模型

- 线程有2种实现方法：

- 1) 用户层的用户线程

- 2) 内核层的内核线程

- 用户线程：受内核支持，但不需内核管理

- 内核线程：由操作系统直接支持和管理

- 用户线程机制和内核线程机制存在一种关联关系，具体反映在线程模型的实现上

- 多对一模型

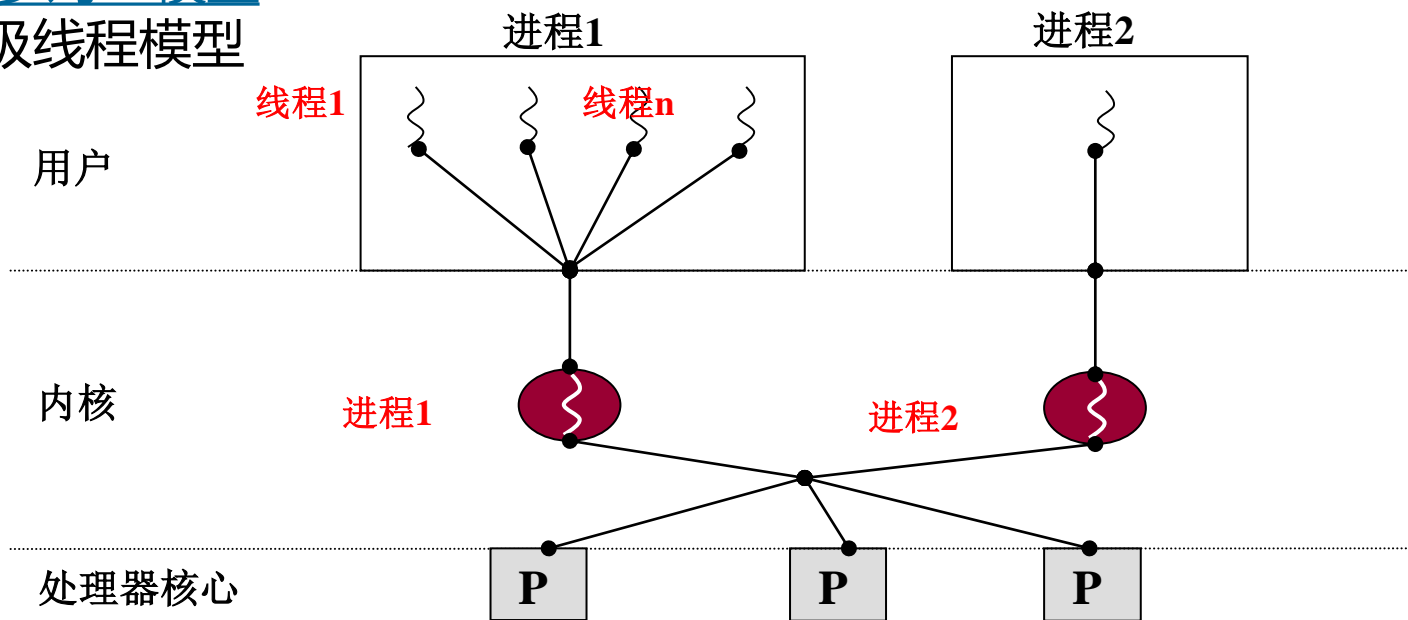
- 一对一模型

- 多对多模型

4.2 多线程模型

4.2.1 多对一模型

用户级线程模型



管理是由线程库在用户空间进行的，**内核并无感知**

任何时刻只有一个线程能访问内核（如果需要的话）

同一进程的线程**不能**运行在不同的CPU上

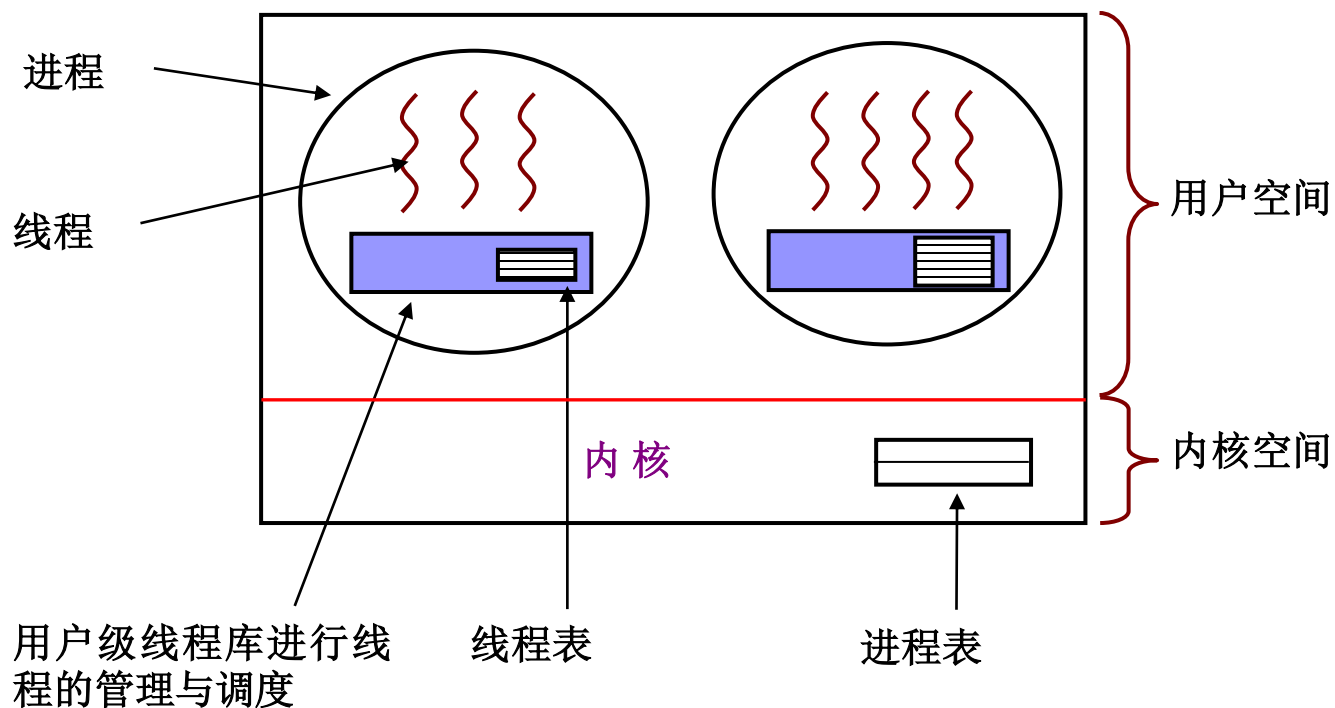
若一个线程执行了阻塞系统调用，则整个进程会被阻塞

线程间占用CPU等协调工作由用户程序自行解决

4.2 多线程模型

4.2.1 多对一模型

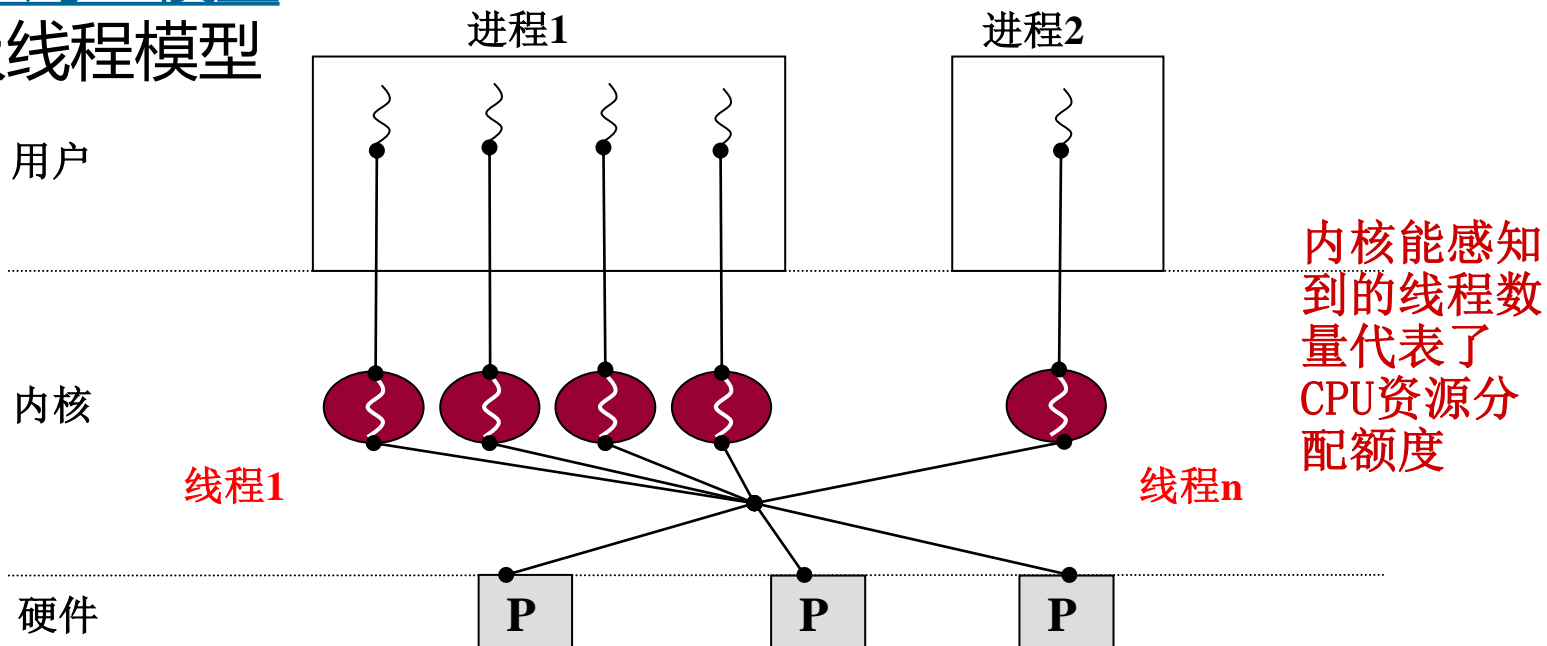
用户级线程模型



用户级线程实现

4.2 多线程模型

4.2.2 一对一模型 内核级线程模型



用户创建的就是内核级线程！（通过系统调用）

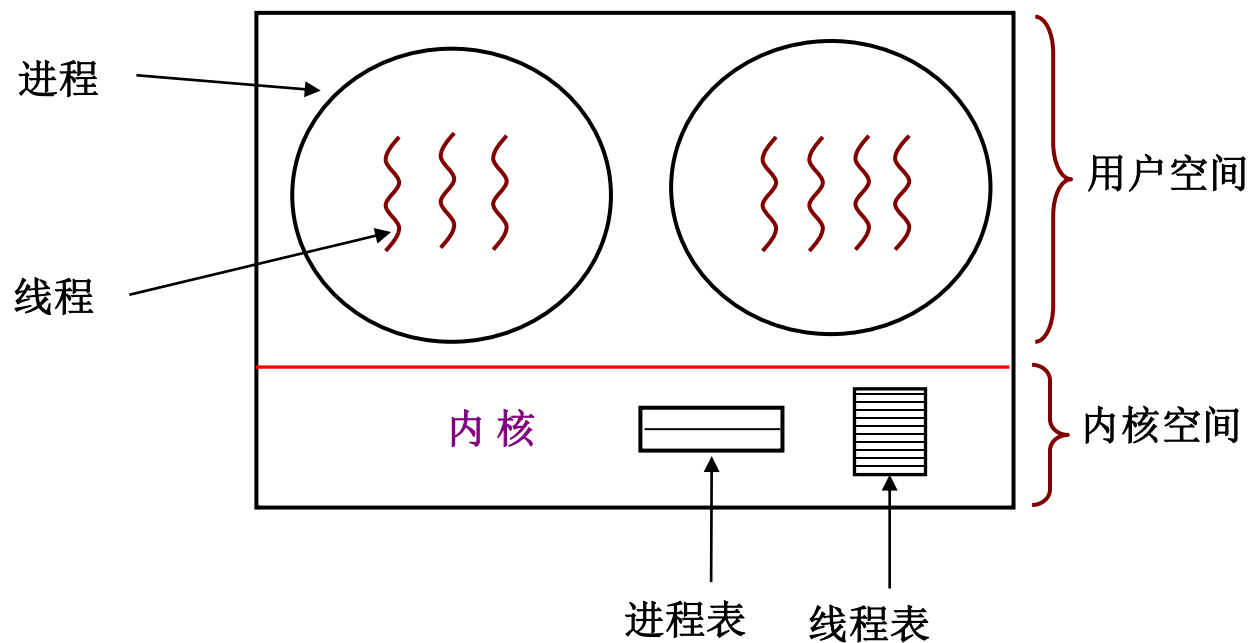
一个线程执行阻塞系统调用时，允许CPU切换给其他线程，因此具有更好的**并发能力**

一个进程的不同线程可运行在不同的CPU上，实现**多CPU的并行处理**

但创建内核线程的数量会**影响系统性能**，OS会**限制线程总量**

4.2 多线程模型

4.2.2 一对一模型 内核级线程模型



内核级线程实现

4.2 多线程模型

4.2.2 一对一模型

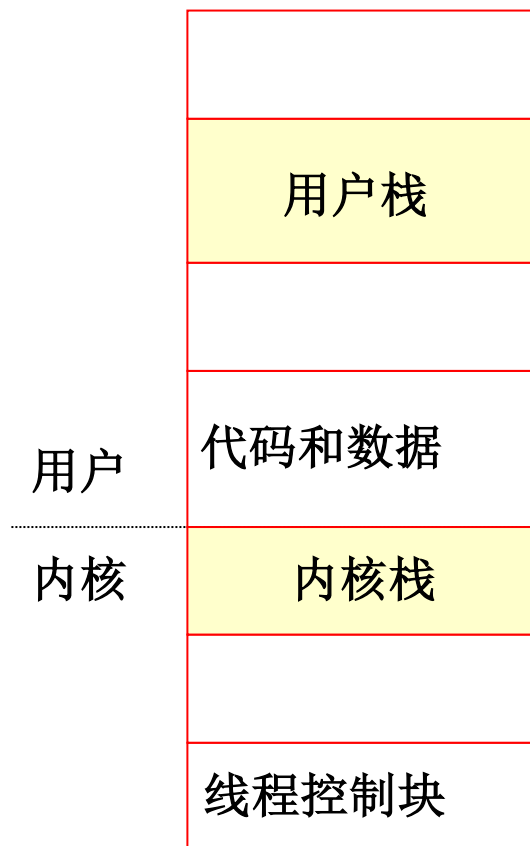
内核级线程模型

和用户级相比，内核级线程有什么不同？
以x86下的linux为例

由内核完成线程的创建、调度等

主要工作仍是保存现场 – 在哪里？ – 内核栈

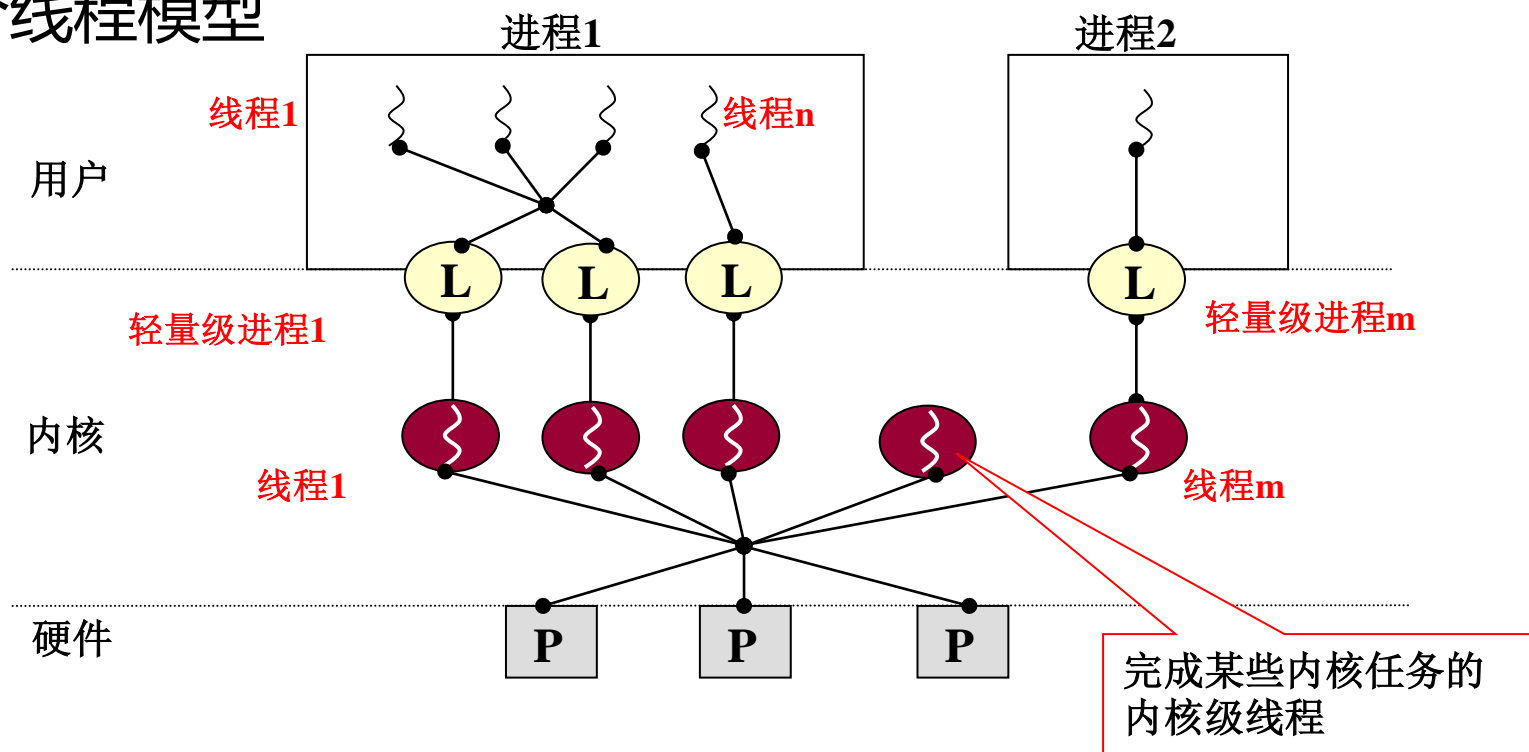
- 每个执行序列需有两个栈：
用户栈 + 内核栈
- 用户栈：普通的函数调用
- 内核栈：系统调用、中断处理



4.2 多线程模型

4.2.3 多对多模型

混合线程模型

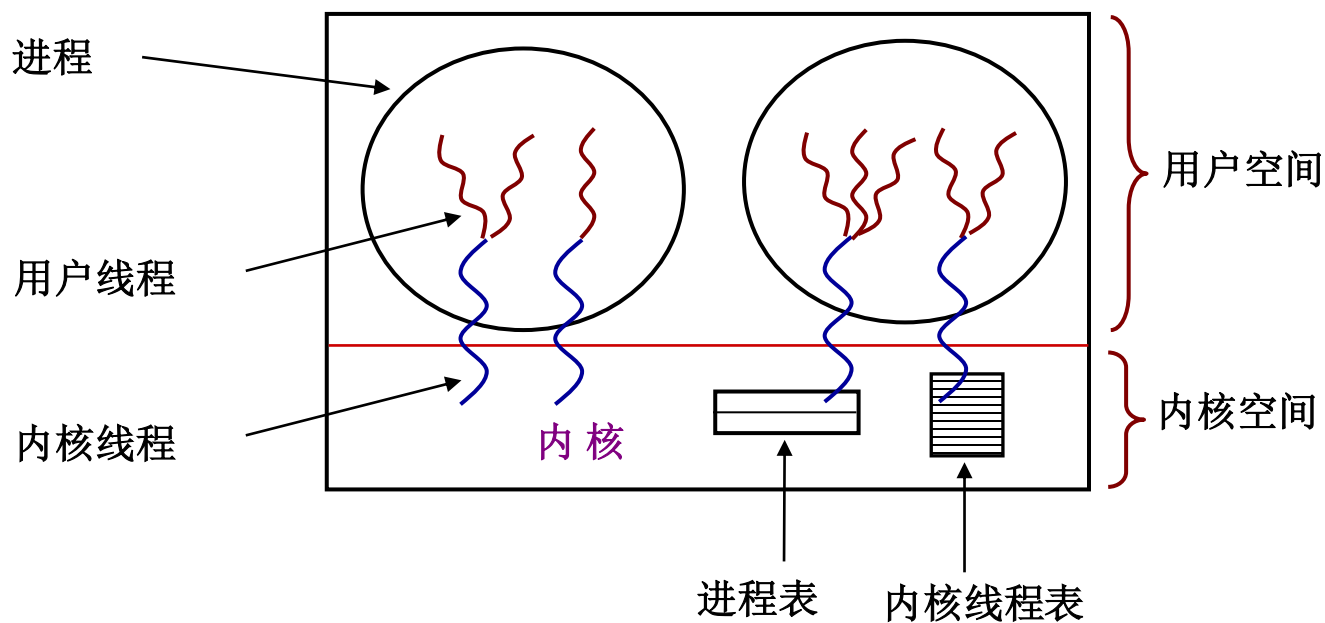


用户线程与内核线程间增加映射管理（轻量级进程）

4.2 多线程模型

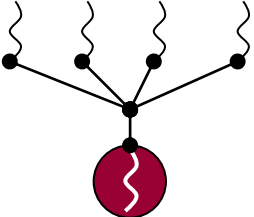
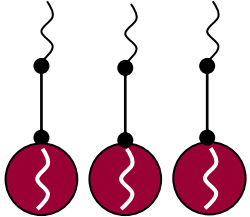
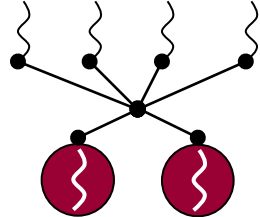
4.2.3 多对多模型

混合线程模型



混合线程实现

4.3 多线程模型比较

	用户级线程	内核级线程	用户+内核级
实现模型			
利用多核	差	好	好
并发度	低	高	高
切换代价	小	大	中
OS代码	无	大	大
用户灵活性	大	小	大
CPU资源分配	少	多	中

4.4 线程库

线程库：为程序员提供的用来创建和管理线程的API

■ 有两种方法实现线程库：

(1) 用户线程库：

- 在用户空间中提供一个没有内核支持的线程库，其所有代码和数据结构都存在于用户空间中。
- 调用库中的一个函数只是导致了用户空间中的一个**本地函数调用**，而不是系统调用。
- 编程语言提供的库函数。

(2) 内核线程库：

- 由操作系统直接提供的线程库。其所有代码和数据结构存在于内核空间中。
- 调用库中的一个**API**函数通常会导致对内核的**系统调用**。
- 操作系统提供的**API**。
- 编程语言的线程库通常是对内核线程库进行更高级别的封装

4.4 线程库

3类线程库：

(1) POSIX Pthread (2) 操作系统（如Win32） (3) 高级语言（如Java）

(1) POSIX Pthread线程库：

- 只是一种规范，不是具体实现
- Pthread是POSIX标准的扩展，可以实现为用户级或内核级的线程库

线程API	功能描述
Pthread_create	创建一个新线程
Pthread_exit	结束调用的线程
Pthread_join	等待一个特定的线程完成/退出
Pthread_yield	让出CPU给其他线程
Pthread_attr_init	创建并初始化一个线程的属性结构
Pthread_attr_destroy	删除一个线程的属性结构

4.4 线程库

3类线程库：

(1) **POSIX Pthread** (2) 操作系统（如**Win32**） (3) 高级语言（如**Java**）

(1) **POSIX Pthread**线程库：

- 只是一种规范，不是具体实现
- **Pthread**是**POSIX**标准的扩展，可以实现为用户级或内核级的线程库

(2) **Win32**线程库：

- **Win32**线程库是应用于**Windows OS**的内核级线程库

(3) **Java**线程库：

- **Java**线程库允许在**Java**程序中直接创建和管理线程
- **Java**线程库跟随**JVM**运行在宿主**OS**中
- **Java**线程库使用宿主**OS**支持的线程库实现的

Pthreads Example

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    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;
    }
}
```

Pthreads Example (Cont.)

```
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid,&attr,runner,argv[1]);
    /* 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);
}
```


Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

Windows Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* 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;

    if (argc != 2) {
        fprintf(stderr, "An integer parameter is required\n");
        return -1;
    }
    Param = atoi(argv[1]);
    if (Param < 0) {
        fprintf(stderr, "An integer >= 0 is required\n");
        return -1;
    }
}
```

Windows Multithreaded C Program (Cont.)

```
/* 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 */

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

    /* close the thread handle */
    CloseHandle(ThreadHandle);

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

4.4 线程库

- **Java线程机制建立在宿主操作系统的线程基础上**
将宿主操作系统的线程机制包装为语言一级的机制提供给程序员使用。

⇒ 对上：为程序员提供简单一致、独立于平台的多线程编程接口。

⇒ 对下：为程序员屏蔽宿主操作系统线程机制的细节。

不同宿主操作系统的线程机制有差别。

□ 由于java程序运行在JVM中，JVM根据运行的操作系统又有不同的实现，因此不同平台上的JVM实现了不同的多线程模型。

□ 比如在Windows和Linux中使用的是一对一模型，而在早期的Solaris中实现的是多对一模型，后来改为多对多模型，Solaris 9后又采用了一对一模型。

4.4 线程库

Java线程库实例

- Step 1: 编写一个线程类

程序员自定义的类:

- ① 要么实现`java.lang.Runnable`接口。

- ⇒ 更好的面向对象设计，提供一致的设计风格。

- ⇒ 可节约Java语言宝贵的单继承指标。

- 例如，一个Applet程序必须选择这一途径实现多线程

-

- ② 要么继承`java.lang.Thread`类。

- ⇒ 该类实现了`Runnable`接口，但`run()`被定义为空的。

- ⇒ 可用更简单的代码完成同样的事情。

线程类中最重要的重定义方法：`run()`。

- ⇒ 通过重定义该方法完成线程本身要执行的任务。

4.4 线程库

Java线程库

- Step 2: 创建线程类的一个对象实例

- ① 如果线程类实现Runnable接口:

- ⇒ 首先, 创建该线程类的一个实例。

- ⇒ 然后, 再以上述实例为参数创建Thread的一个实例。

- ② 如果线程类继承Thread类:

- ⇒ 直接创建线程类的一个实例。

Thread类的构造方法有3个参数, 共7种组合:

- ① 一个Runnable对象实例

- 该对象的run()方法将在线程中执行。

- ② 一个String对象实例

- 用于标识线程。

- ③ 一个ThreadGroup对象实例

- 将线程指派到某一线程组; 线程组将相关线程组织在一起。

4.4 线程库

Java线程库

- Step 3: 调用对象实例的`start()`方法

该方法创建并运行线程。

⇒ 申请线程运行所需系统资源、调度线程、调用线程的

`run()`方法。

当该方法返回时，线程已开始运行并处于`Runnable`状态。

⇒ 对于计算机只有一个处理器，JVM必须调度线程。

Java Multithreaded Program

```
class Sum
{
    private int sum;

    public int getSum() {
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }
}

class Summation implements Runnable
{
    private int upper;
    private Sum sumValue;

    public Summation(int upper, Sum sumValue) {
        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run() {
        int sum = 0;
        for (int i = 0; i <= upper; i++)
            sum += i;
        sumValue.setSum(sum);
    }
}
```


Java Multithreaded Program (Cont.)

```
public class Driver
{
    public static void main(String[] args) {
        if (args.length > 0) {
            if (Integer.parseInt(args[0]) < 0)
                System.err.println(args[0] + " must be >= 0.");
            else {
                Sum sumObject = new Sum();
                int upper = Integer.parseInt(args[0]);
                Thread thrd = new Thread(new Summation(upper, sumObject));
                thrd.start();
                try {
                    thrd.join();
                    System.out.println
                        ("The sum of "+upper+" is "+sumObject.getSum());
                } catch (InterruptedException ie) { }
            }
        }
        else
            System.err.println("Usage: Summation <integer value>"); }
}
```

4.5 操作系统进程/线程模型分类

根据对进程与线程的支持，操作系统大致分为如下类型：

- 1) 单进程、单线程：MS-DOS是这种操作系统；
 - 2) 多进程、多线程（轻量级进程）：多数UNIX、类UNIX的linux是这种操作系统，他们的内核级线程是对进程进行了轻量级的封装；
 - 3) 多进程、多线程：Windows NT（以及基于NT内核的Windows 2000、XP、win7、win10等）、Solaris 2.x和OS/2都是这种操作系统；
 - 4) 单进程、多线程：vxWorks、RTEMS、UCOS等就是这种操作系统。vxWorks只有一个进程（共享内存空间和资源），其任务的概念与线程相当，所有任务间共享内存和资源。
- 多进程、多线程模型的操作系统一般都需要虚拟内存的支持，一个进程有一个虚拟地址空间；
 - 进程间的地址空间通过虚拟内存进行了隔离，进程内的所有线程共用进程地址空间。

4.6 协程（用户态轻量级线程）

协程 (Coroutine)：是一种轻量级的用户态线程，实现的是非抢占式的调度，即由当前协程切换到其他协程由当前协程来控制。

目前的协程框架一般都是设计成 1:N 模式。1:N 就是一个线程作为一个容器里面放置多个协程。

那么谁来适时的切换这些协程？由协程自己主动让出 CPU，有一个调度器，这个调度器是被动调度的。当一个协程需要让出 CPU 时（比如：异步等待网络的数据，当前还没有数据到），通知调度器。

调度器根据设计好的**调度算法**找到一个协程。切换当前协程的 CPU 上下文把 CPU 的运行权交给找到的协程，直到这个协程出现执行不下去需要等待的情况，或者它调用主动让出 CPU 的 API 之类，触发下一次调度。

- 协程(用户级线程) 可以在编译器中基于线程技术封装和实现

4.6 协程（用户态轻量级线程）

- 不同语言对协程的支持：

- C++通过Boost.Coroutine实现对协程的支持
 - Java不支持
 - Python3.5使用async def对原生协程的支持
 - Go (Golang) 天然（内置）支持协程，是一种并行计算语言
-
- 利用多核、GPU对图像、语音、信号处理会有大量的并行运算。
 - 为了更好地编写并发程序，Go语言就注重在编程语言层级上设计一个简洁安全高效的抽象模型。
 - 让程序员专注于分解问题和组合方案，而且不用被线程管理和信号互斥这些烦琐的操作分散精力。

4.6 协程（用户态轻量级线程）

□ GO语言一种并行计算语言

- Go语言是谷歌2009发布的第二款开源编程语言。
- Go语言专门针对多处理器系统的编程进行了优化，使用Go编译的程序可以媲美C或C++代码的速度，而且更加安全、支持并行计算。
- 2017年8月24日，Go语言Go 1.9版发布。
- Go语言设计支持主流的32位和64位的x86平台，同时也支持32位的ARM架构。
- Go语言在Go1版本上支持Windows,苹果MacOSX,Linux和FreeBSD操作系统。
- LiteIDE是一款专门为Go语言开发的跨平台轻量级集成开发环境(IDE)，QT编写。

4.6 协程（用户态轻量级线程）

▣ Go语言:goroutine（创建协程）

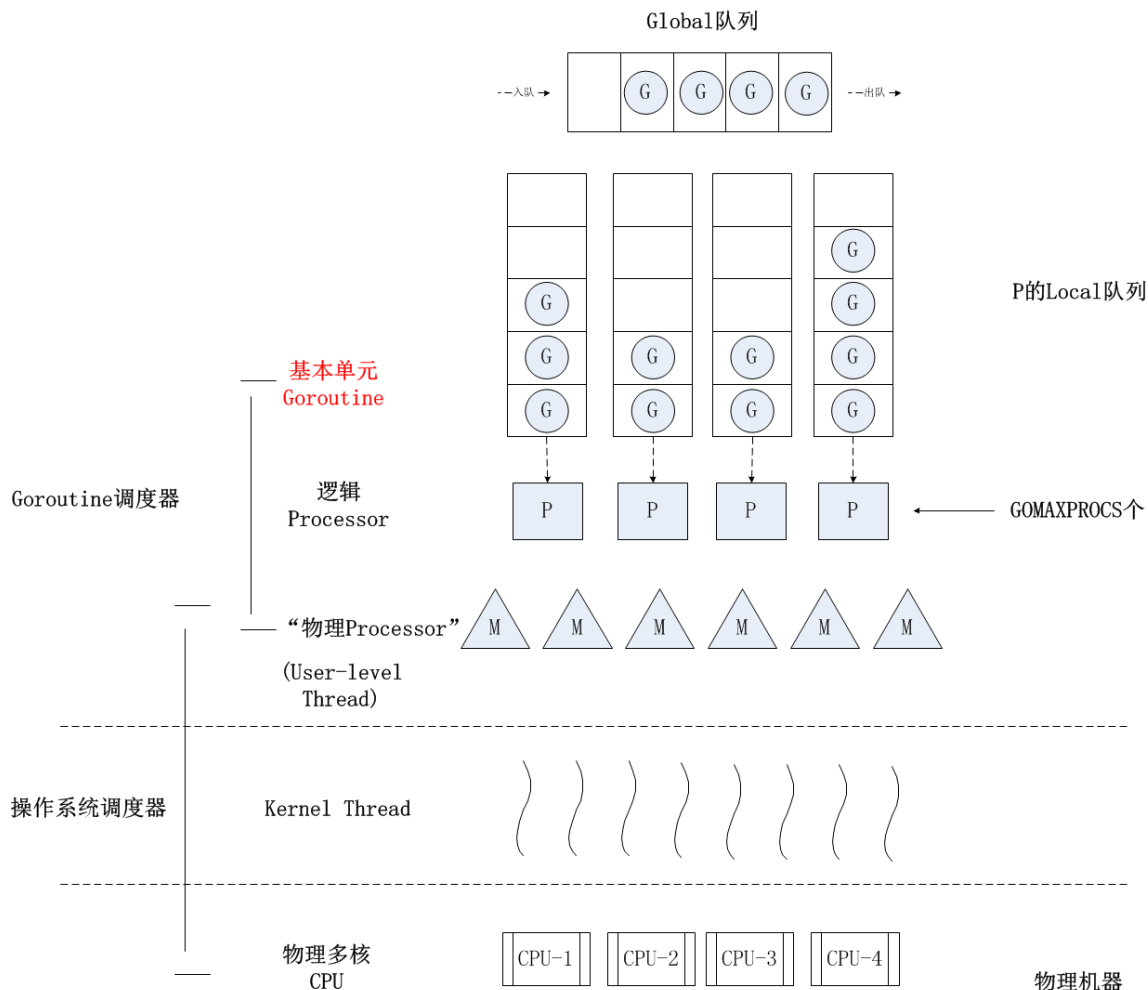
- 在Go语言中，只需要在函数调用前加上**关键字go**即可创建一个**并发任务单元**，新建的任务会被**放入队列**中，等待**调度器安排（编译器内嵌打包）**。
- go语言中并发指的是让某个函数独立于其它函数运行的能力，**N**个goroutine(**G**)是一个独立的工作单元，Go的runtime（运行时）会在逻辑处理器(**P**)上调度goroutine来运行，一个逻辑处理器绑定一个操作系统内核线程(**M**)。这种模式称为：MPG调度。

4.6 协程（用户态轻量级线程）

- Go: MPG调度

为了实现**M: N(多对多)** 线程调度机制, Go引入了3个结构体:

- M: 操作系统的内核级线程
- G: goroutine对象, G结构体包含调度一个goroutine所需要的堆栈和 instruction pointer (IP指令指针), 以及其它一些重要的调度信息。每次go调用的时候, 都会创建一个G对象。
- P: Processor, 调度的上下文, 实现 M: N调度模型的关键, **M必须获得P才能对G进行调度, P限制了调度goroutine的最大并发度。** 每一个运行的M都必须绑定一个P。
- P的个数是**GOMAXPROCS (最大256)**, 启动时固定, 一般不修改; **M的个数和P的个数没有严格对应关系 (会有休眠的M或者不需要太多的M); 每一个P保存着本地G任务队列, 也能使用全局G任务队列。**



Goroutine调度原理图

4.6 协程（用户态轻量级线程）

Go语言:goroutine

- Go适合用来做什么
- 服务器编程:如处理日志、数据打包、虚拟机处理、文件系统等。
- 分布式系统，数据库代理器等
- 网络编程:应用最广，包括Web应用、API应用、下载应用
- 内存数据库:google开发的groupcache，couchbase的部分组建
- 云平台:目前国外很多云平台采用Go开发，docker，CloudFoundry的部分组件，apcera云平台。

```
package main

import (
    "fmt"
    "sync"
)

func main(){
    var wg sync.WaitGroup
    wg.Add(2)
    go func() {
        defer wg.Done()
        for i := 0; i < 10000; i++ {
            fmt.Printf("Hello,Go.This is %d\n", i)
        }
    }()
    go func() {
        defer wg.Done()
        for i := 0; i < 10000; i++ {
            fmt.Printf("Hello,World.This is %d\n", i)
        }
    }()
    wg.Wait()
}
```


4.6 协程（用户态轻量级线程）

Go成功的项目

- nsq: bitly开源的消息队列系统，性能非常高，目前他们每天处理数十亿条的消息
- docker: 基于lxc的一个虚拟打包工具，能够实现PAAS平台的组建。
- packer: 用来生成不同平台的镜像文件，例如VM、vbox、AWS等，作者是vagrant的作者
- skynet: 分布式调度框架
- Doozer: 分布式同步工具，类似ZooKeeper
- Heka: mazila开源的日志处理系统
- cbfs: couchbase开源的分布式文件系统
- tsuru: 开源的PAAS平台，和SAE实现的功能一模一样
- groupcache: memcache作者写的用于Google下载系统的缓存系统
- god: 类似redis的缓存系统，但是支持分布式和扩展性
- gor: 网络流量抓包和重放工具

以下是一些公司，只是一小部分：

- Apcera.com
- Statthat.com
- Juju at Canonical/Ubuntu, presentation
- Beachfront.io at Beachfront Media
- CloudFlare
- Soundcloud
- Mozilla
- Disqus
- Bit.ly
- Heroku
- google
- youtube

利用多处理器的方法？

- 操作系统级别（多核、多处理器）：多进程/多线程
- 语言内嵌/库+编译器（多核、多处理器）：协程
- 编程库+编译器（众核GPU）：CUDA和OpenCL是最主流的两个GPU编程库，CUDA和OpenCL都是原生支持C/C++的

总结

进程的切换代价过大，进程间共享和交互开销较大，寻求轻量级的并行/并发解决办法。

线程：用户级线程和内核级线程，栈的数量和使用上不同。用户级线程和内核级线程各有优缺点。

线程的切换是进程切换的基础，再加上地址空间（内存映射表）的切换就可以构建出进程切换。

总结

● 进程 —— 重量级任务

- ① 每一进程占用独立的地址空间。
 - ⇒ 此处的地址空间包括代码、数据及其他资源。
- ② 进程间的通信开销较大且受到许多限制。
 - ⇒ IPC, 对象 (或函数) 接口、通信协议、...
- ③ 进程间的切换开销也较大。
 - ⇒ 上下文切换 (**context switch**) , 地址空间切换。
 - ⇒ 上下文包括代码、数据、堆栈、处理器状态、资源、...

● 线程 —— 轻量级任务

- ① 多个线程共享进程的地址空间 (代码、数据、其他资源等) 。
 - ⇒ 线程也需要自己的资源, 如程序计数器、寄存器组、调用栈等。
- ② 线程间的通信开销较少且比较简单。
 - ⇒ 因为共享而减少了需要通信的内容。
 - ⇒ 但也因为充分共享, 需要用户对共享资源进行保护。
- ③ 线程间的切换开销也较小。
 - ⇒ 只需保存每一线程的程序计数器、寄存器组、堆栈等空间。
 - ⇒ 不必切换地址空间, 从而成本大为降低。