串行、并发与并行

- 串行：仅当一个程序执行完毕，下一个程序才能开始；
- 特点：CPU每次被一个程序独占，CPU利用率不高；例如：当一个程序正在等待用户输入，CPU就会在相当长的时间无事可做，而又不能去执行别的程序。

串行、并发与并行

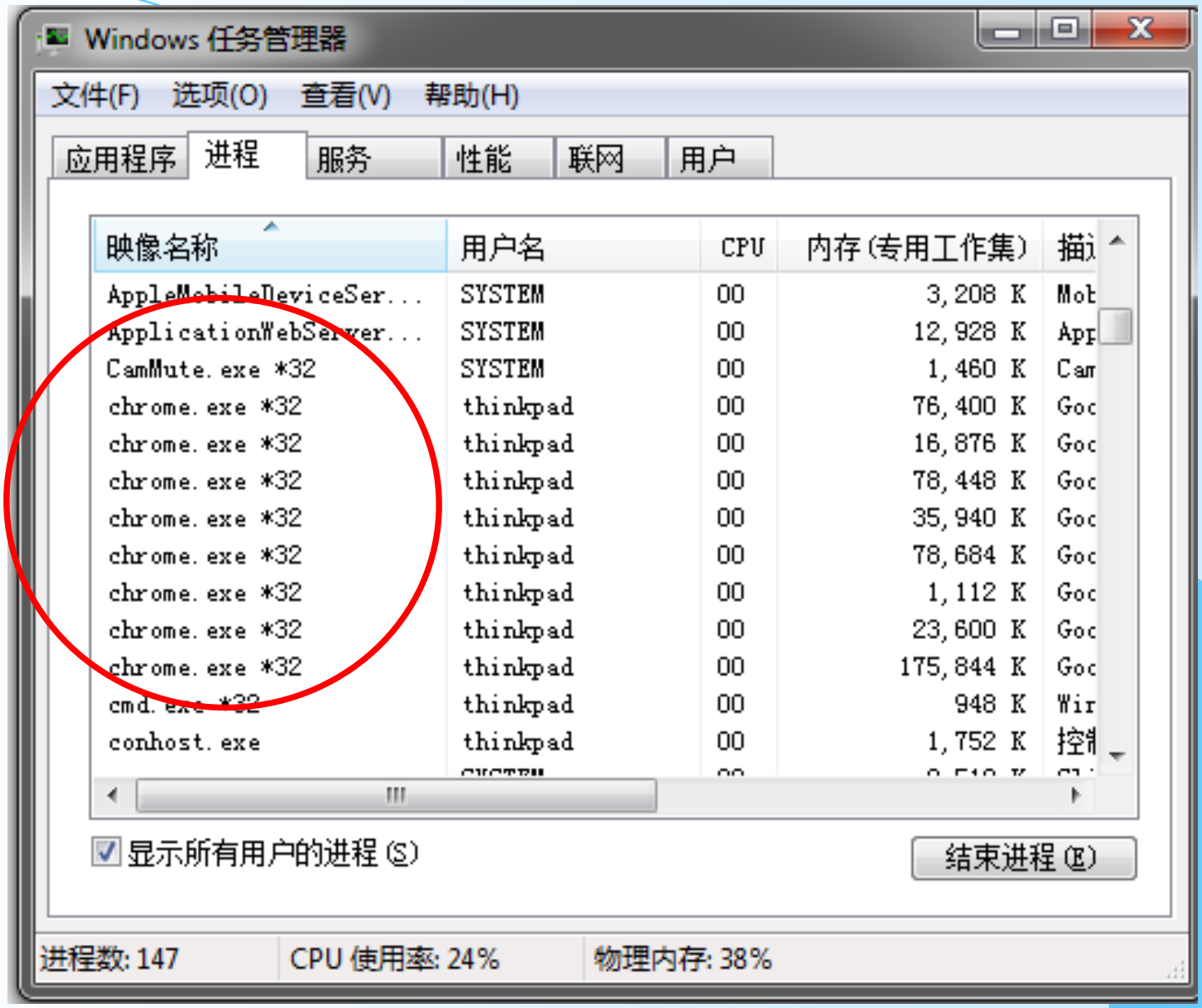
- 并发执行
- 现在操作系统允许多个程序“同时”运行（“多任务”）；事实上，一个CPU在任何一时刻只能执行一条指令；
- 操作系统让多个程序分时使用CPU，CPU不停地在多个程序之间切换；由于CPU运算速度快，用户感觉不到这种切换过程。

串行、并发与并行

- 并行执行：计算机如果有多个处理器（核心是CPU），可以真正做到多个程序同时执行。

程序与进程

- 程序：实现某些功能和服务的计算机编码（静态）
- 进程：程序的一次执行所形成的实体，每当程序开始执行，就会创建一个进程，是一个动态的概念。它可以申请和拥有系统资源，每个进程有程序代码以及一些状态信息（如进程数据的当前值和当前执行点）组成，状态信息也称为进程的上下文。


Windows 任务管理器
文件(F) 选项(O) 查看(V) 帮助(H)
应用程序 进程 服务 性能 联网 用户
映像名称 用户名 CPU 内存(专用工作集) 描述
AppleMobileDeviceSer... SYSTEM 00 3,208 K Mob
ApplicationWebServer... SYSTEM 00 12,928 K App
CamMute.exe *32 SYSTEM 00 1,460 K Cam
chrome.exe *32 thinkpad 00 76,400 K Goo
chrome.exe *32 thinkpad 00 16,876 K Goo
chrome.exe *32 thinkpad 00 78,448 K Goo
chrome.exe *32 thinkpad 00 35,940 K Goo
chrome.exe *32 thinkpad 00 78,684 K Goo
chrome.exe *32 thinkpad 00 1,112 K Goo
chrome.exe *32 thinkpad 00 23,600 K Goo
chrome.exe *32 thinkpad 00 175,844 K Goo
cmd.exe *32 thinkpad 00 948 K Win
conhost.exe thinkpad 00 1,752 K 控制
显示所有用户的进程(S)
结束进程(E)
进程数: 147
CPU 使用率: 24%
物理内存: 38%

调度与进程

- 调度：操作系统控制处理器在多个程序之间切换执行的过程
- 传统的多任务操作系统是以进程为单位进行调度的
- 操作系统通过划分时间片来调度进程

能否利用分配给同一个进程的资源，尽量实现多个任务？？

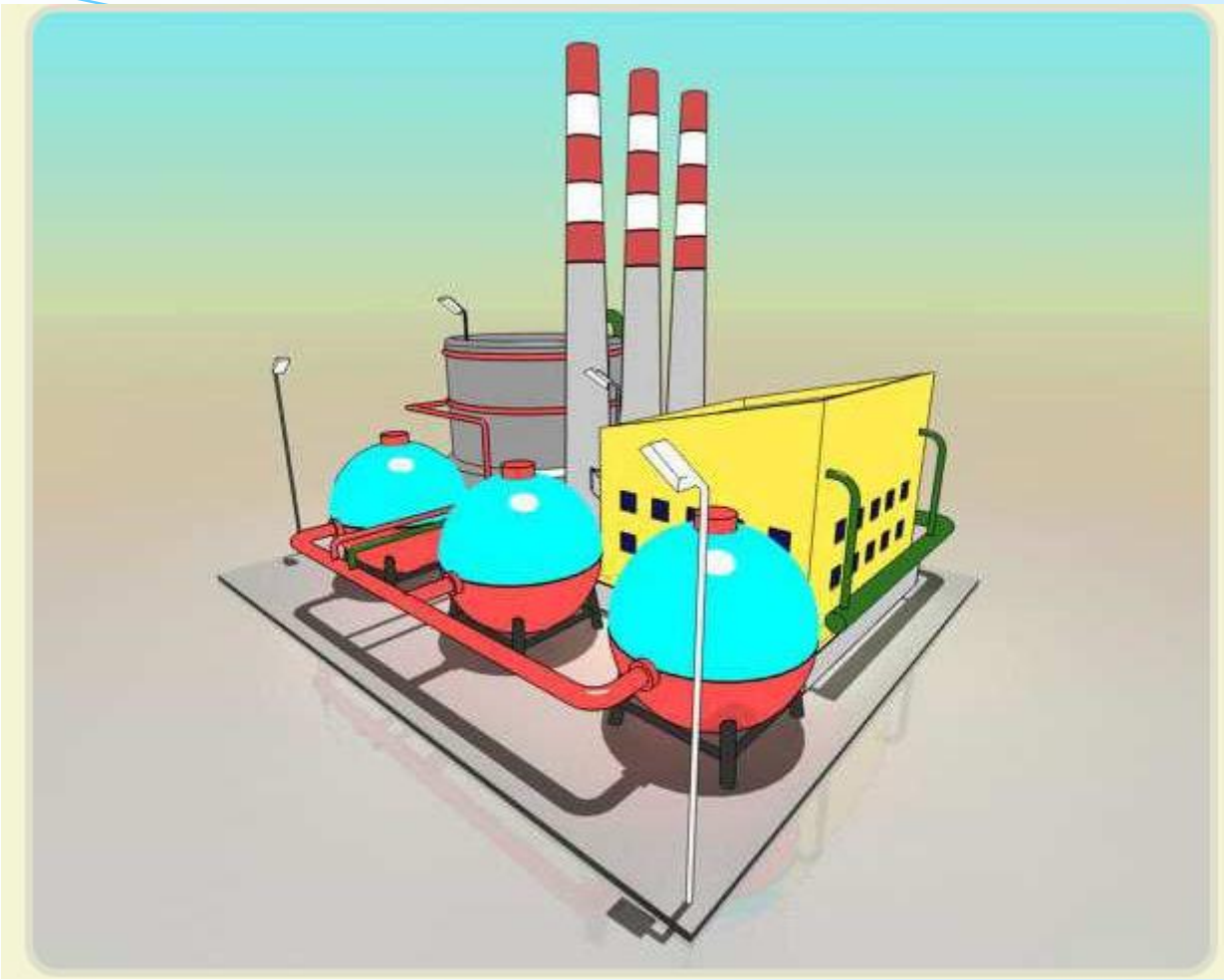
- 问题提出：一个GUI程序，为了有更好的交互性，通常一个任务支持界面，另一个任务支出后台运算。
  - 实现多进程并发需要花费不少系统的开销，每个进程都需要为它分配一些内存，以便存储它的上下文；进程切换时需要保存和恢复进程上下文；
  - 进程与进程是隔离的，进程间通信比较困难。

进程与线程

- 线程：进程中的执行的一段程序片段，进程的执行单元，可调度的实体。
- 通常在一个进程中可以包含若干个线程，它们可以利用进程所拥有的资源。
- 在引入线程的操作系统中，通常都是把进程作为分配资源的基本单位，而把线程作为独立运行和独立调度的基本单位。
- 由于线程比进程更小，基本上不拥有系统资源，故对它的调度所付出的开销就会小得多，能更高效的提高系统内多个程序间并发执行的程度。

进程与线程的区别

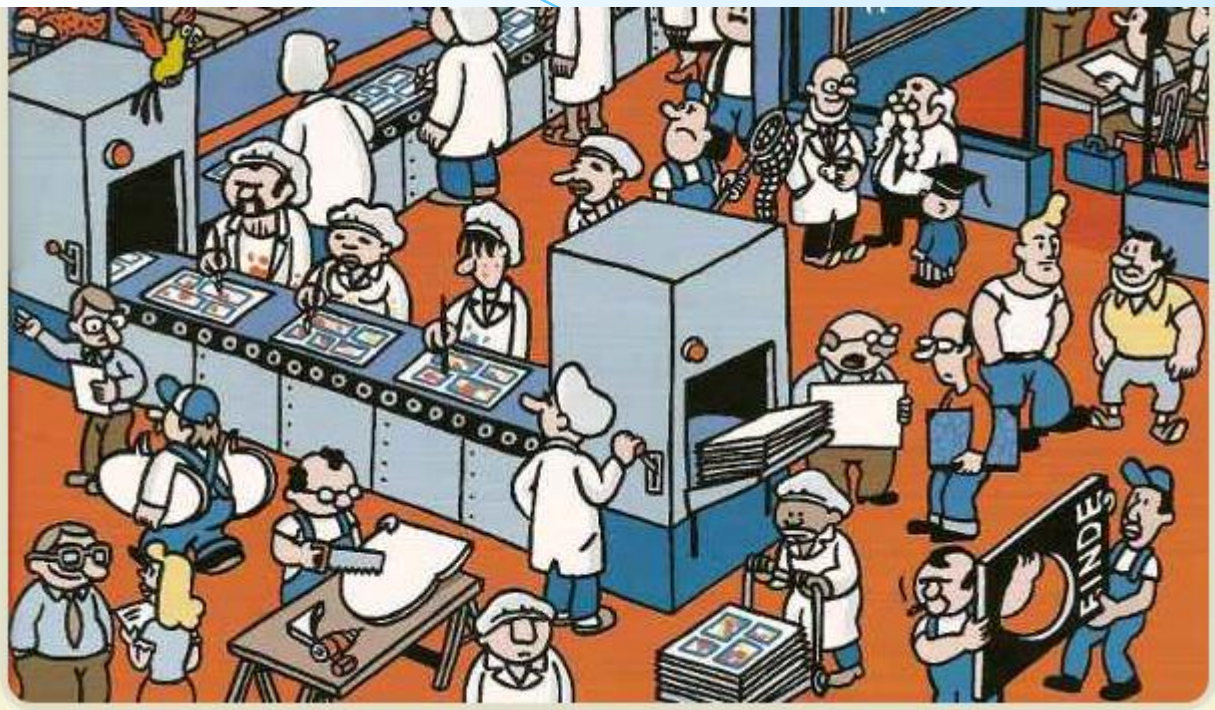
- 1)地址空间：线程是进程内的一个执行单元;进程至少有一个线程;它们共享进程的地址空间;而进程有自己独立的地址空间;
- 2)进程是资源分配和拥有的单位,同一个进程内的线程共享进程的资源;
- 3)线程是处理器调度的基本单位,进程不是;
- 4)二者均可并发执行。



计算机的核心是CPU,它承担了所有的计算任务。它就像一座工厂，时刻在运行。

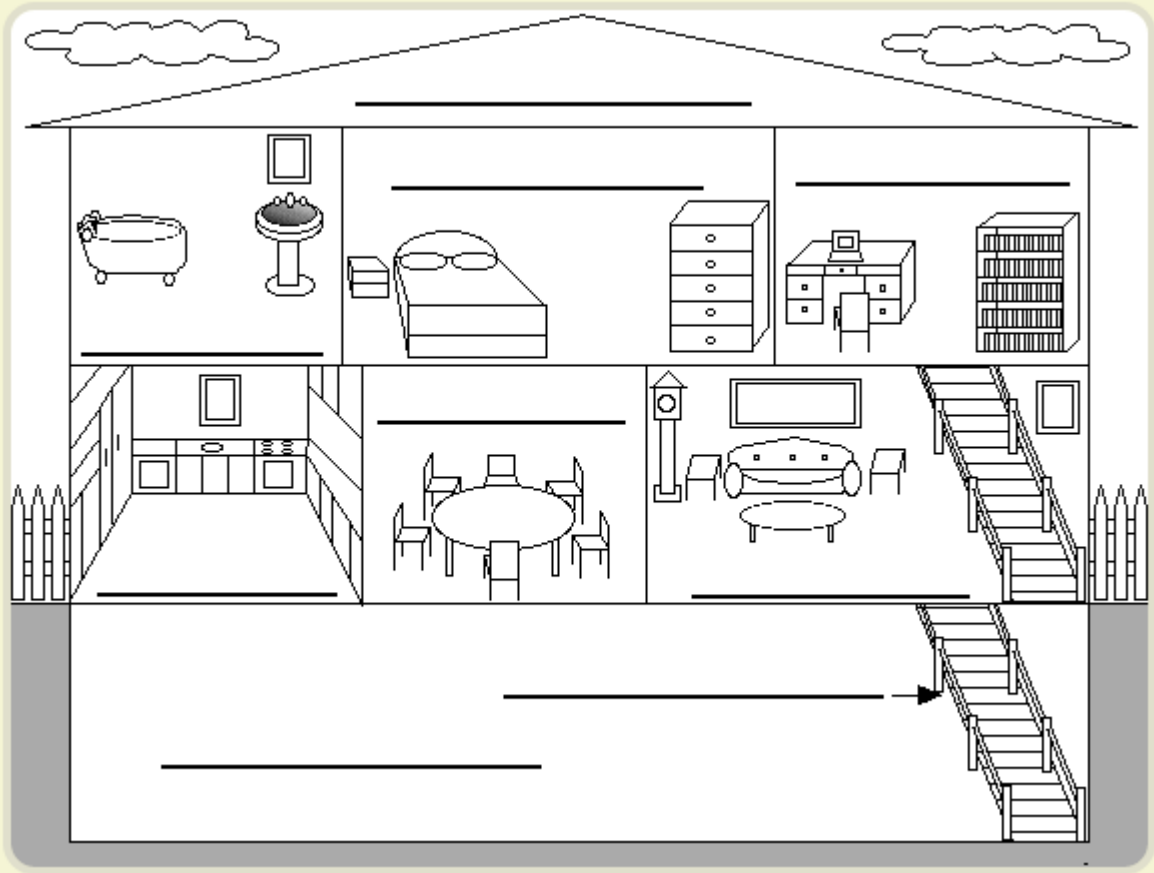
假定工厂的电力有限，一次只能供给一个车间使用。也就是说，一个车间开工的时候，其他车间都必须停工。背后的含义就是，单个CPU一次只能运行一个任务。

进程就好比工厂的车间，它代表CPU所能处理的单个任务。任一时刻，单个CPU总是运行一个进程，其他进程处于非运行状态。



一个车间里，可以有很多工人。他们协同完成一个任务。

线程就好比车间里的工人。一个进程可以包括多个线程。



车间的空间是工人们共享的，比如许多房间是每个工人都可以进出的。

这象征一个进程的内存空间是共享的，每个线程都可以使用这些共享内存。

可是，每间房间的大小不同。

1）有些房间最多只能容纳一个人，比如厕所。里面有人的时候，其他人就不能进去了。

这代表一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存。

一个防止他人进入的简单方法，就是门口加一把锁。先到的人锁上门，后到的人看到上锁，就在门口排队，等锁打开再进去。

这就叫"互斥锁"（Mutual exclusion，缩写Mutex），防止多个线程同时读写某一块内存区域。

2）有些房间，可以同时容纳n个人，比如厨房。也就是说，如果人数大于n，多出来的人只能在外面等着。这好比某些内存区域，只能供给固定数目的线程使用。

这时的解决方法，就是在门口挂n把钥匙。进去的人就取一把钥匙，出来时再把钥匙挂回原处。后到的人发现钥匙架空了，就知道必须在门口排队等着了。这种做法叫做"信号量"（Semaphore），用来保证多个线程不会互相冲突。

操作系统的设计，因此可以归结为三点：

（1）以多进程形式，允许多个任务同时运行；

（2）以多线程形式，允许单个任务分成不同的部分运行；

（3）提供协调机制，一方面防止进程之间和线程之间产生冲突，另一方面允许进程之间和线程之间共享资源。

两大应用领域

- 网络应用
  - 例如：网络数据库
- 嵌入式应用

多线程特性：并发、切换快

多线程的实现

- Ada和VHDL语言,对多线程的支持直接内建在语言中
- C/C++语言，对多线程的支持由具体的操作系统提供的函数接口支持
- Python…

典型操作系统

- UNIX
- Linux　Android　Chrome OS
- Mac OS X　iOS
- Windows　WP

POSIX

- POSIX：可移植操作系统接口（Portable Operating System Interface
- POSIX标准定义了操作系统应该为应用程序提供的接口标准，最初是IEEE为要在各种UNIX操作系统上运行的软件而定义的一系列API标准的总称，其正式称呼为IEEE 1003，而国际标准名称为ISO/IEC 9945。
- POSIX标准意在期望获得源代码级别的软件可移植性，即为一个POSIX兼容的操作系统编写的程序，应该可以在任何其它的POSIX操作系统（即使是来自另一个厂商）上编译执行。
- LINUX/UNIX/MacOSX 系统自带
- WINDOWS，下载PTHREAD的WINDOWS开发包，网站地址是http://sourceware.org/pthreads-win32/

线程环境

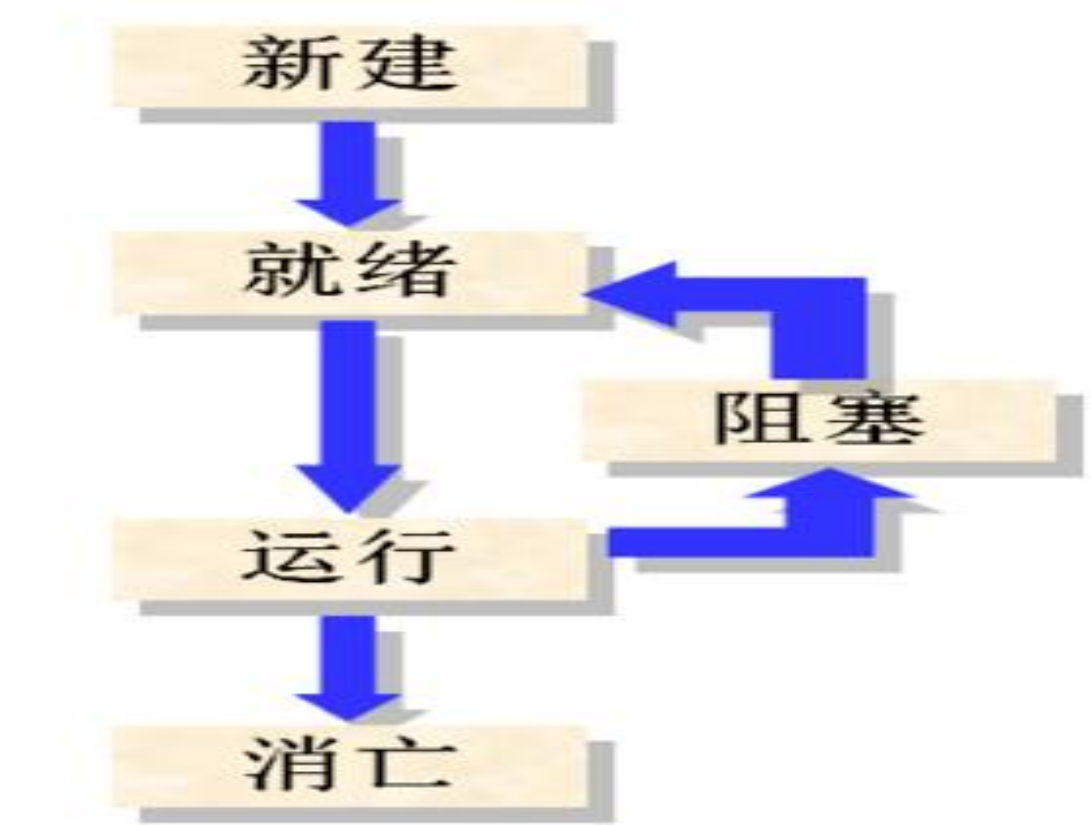
- 进程中所有全局资源对每个线程均可见
  - 代码区：本进程空间内所有可见的函数代码
  - 静态存储区：全局变量，静态变量
  - 动态存储区：堆空间
- 进程中的局部资源
  - 本地栈空间，存放本线程的函数调用栈、函数内部的局部变量
  - 部分寄存器变量：本线程下一步要执行代码的指针偏移量

```
void f1();


void  f2();
int main()
{
    f1();
    f2();
    return 0;
}
```

- 程序开始运行main就产生一个进程，同时产生一个主线程；
- 主程序中运行的“函数过程”可以很好的描述线程的概念。
- 主程序包含了许多函数，操作系统可以调度这些函数，使之同时或者（和）独立的执行。这就描述了“多线程”程序。

多线程的生存周期



Posix Pthreads API

- ***线程管理（Thread management）:*** 用于线程：创建（creating），分离（detaching），连接（joining）等等。
- ***互斥量（Mutexes）:*** 用于线程同步的，称为互斥量（mutexes），是"mutual exclusion"的缩写。Mutex函数提供了创建，销毁，锁定和解锁互斥量的功能。同时还包括了一些用于设定或修改互斥量属性的函数。
- ***条件变量（Condition variables）:*** 处理共享一个互斥量的线程间的通信，基于程序员指定的条件。这类函数包括指定的条件变量的创建，销毁，等待和受信（signal）。设置查询条件变量属性的函数也包含其中。

创建进程

- pthread_create 创建一个新线程并使之运行起来。
- pthread_create参数：
  - thread：返回一个不透明的，唯一的新线程标识符。
  - attr：不透明的线程属性对象。可以指定一个线程属性对象，或者NULL为缺省值。
  - start_routine：线程将会执行一次的C函数。
  - arg: 传递给*start_routine*单个参数，传递时必须转换成指向void的指针类型。没有参数传递时，可设置为NULL。
- 一个线程被创建后，线程何时何地被执行取决于操作系统的实现。强壮的程序应该不依赖于线程执行的顺序。

结束线程

- 结束线程的方法有以下几种：
  - 线程从主线程（main函数的初始线程）返回。
  - 线程调用了pthread_exit函数，显示退出线程。
  - 其它线程使用 pthread_cancel函数结束线程。
  - 调用exec或者exit函数，整个进程结束。

结束线程

- 如果main()在其他线程创建前用pthread_exit()退出了，其他线程将会继续执行。否则，他们会随着main的结束而终止。
- **pthread_exit (status)**

连接线程

- “连接”是一种在线程间完成同步的方法
- pthread_join (threadid, status)
- pthread_join()函数阻塞调用线程，直到threadid所指定的线程终止

例子1

- Tom和Mary同时在操场上开始跑步，每个人都需要跑5圈，Tom一圈跑30秒，Mary一圈跑60秒
- 实际仿真 秒->毫秒

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <time.h>

#define MAXCOUNT  5

void Tomrun()
{
    int i;
    for(i = 1; i <= MAXCOUNT; ++i)
    {
        Sleep(30);  //Sleep()的单位为毫秒
        printf("Tom: the %d rap\n",i);
    }
    return NULL;
}
```

```
void Maryrun()
{
    int i;

    for(i=1; i <= MAXCOUNT; ++i)
    {
        Sleep(60);
        printf("Mary: the %d rap\n",i);
    }
    return NULL;
}
```

串行

```
int main()
{
    clock_t start, finish;
    double  duration;
    start = clock();

    Tomrun();
    Maryrun();

    finish = clock();
    duration = (double)(finish - start) / CLOCKS_PER_SEC;
    printf( "\n\nThe total time is %f seconds\n", duration );

    return 0;
}
```

```
Tom: the 1 rap
Tom: the 2 rap
Tom: the 3 rap
Tom: the 4 rap
Tom: the 5 rap
Mary: the 1 rap
Mary: the 2 rap
Mary: the 3 rap
Mary: the 4 rap
Mary: the 5 rap


The total time is 0.450000 seconds

Process returned 0 (0x0)   execution time : 0.499 s
Press any key to continue.
```

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <time.h>

#define MAXCOUNT  5

void* Tomrun(void* args)
{
    int i;
    for(i = 1; i <= MAXCOUNT; ++i)
    {
        Sleep(30);  //Sleep()的单位为毫秒
        printf("Tom: the %d rap\n",i);
    }
    return NULL;
}
```

```
void* Maryrun(void* args)
{
    int i;

    for(i=1; i <= MAXCOUNT; ++i)
    {
        Sleep(60);
        printf("Mary: the %d rap\n",i);
    }
    return NULL;
}
```

```
int main()
{
    pthread_t t1;  //线程标识符
    pthread_t t2;

    clock_t start, finish;
    double  duration;
    start = clock();

    pthread_create(&t1,NULL,Tom
    pthread_create(&t2,NULL,Mar
    pthread_join(t1,NULL);  //等待
    pthread_join(t2,NULL); //等待t2结束

    finish = clock();
    duration = (double)(finish - start) / CLOCKS_PER_SEC;
    printf( "\n\nThe total time is %f seconds\n", duration );
    return 0;
}
```

```
Tom: the 1 rap
Mary: the 1 rap
Tom: the 2 rap
Tom: the 3 rap
Mary: the 2 rap
Tom: the 4 rap
Tom: the 5 rap
Mary: the 3 rap
Mary: the 4 rap
Mary: the 5 rap


The total time is 0.300000 seconds


Process returned 0 (0x0)   execution time : 0.584 s
Press any key to continue.
```

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <time.h>

#define MAXCOUNT  5

void* Tomrun(void* args)
{
   int i;
   for(i = 1; i <= MAXCOUNT; ++i)
   {
      Sleep(30);  //Sleep()的单位为毫秒
      printf("Tom: the %d rap\n",i);
   }
   return NULL;
}
```

```
void* Maryrun(void* args)
{
   int i;
   unsigned int seconds;
   for(i=1; i <= MAXCOUNT; ++i)
   {
      //Sleep(60);
      seconds = 40 + rand()% 31;  //[40,70]
      Sleep(seconds);
      printf("Mary: the %d rap\n",i);
   }
   return NULL;
}
```

田径自动计时仪器

田径自动计时仪器

田径自动计时仪器

- COMS高感光摄像头。
- 与高性能PC机或笔记本电脑使用USB2.0高速连接，传输图像信息和摄像头设置调整指令。充分支持热拔插，高速特性。
- 图像采集控制器负责接收发令盒传来的启动计时指令，并通过高精度晶振产生1/1000s外部时钟信号，供摄像头采集图像。
- 发令盒和声音传感器将发令信号送达图像采集控制器。
- **终点红外线计时系统辅助终点摄像图像判读系统记录第一名队员成绩并将第一名队员成绩实时发送到终点红外显示屏上。**

例子1分析

- Tomrun与Maryrun相互对立，不影响

共享数据

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

互斥量（Mutex）

- 互斥量是实现线程同步，和保护同时写共享数据的主要方法；
- 互斥量对共享数据的保护就像一把锁。在Pthreads中，任何时候仅有一个线程可以锁定互斥量，因此，当多个线程尝试去锁定该互斥量时仅有一个会成功。直到锁定互斥量的线程解锁互斥量后，其他线程才可以去锁定互斥量。线程必须轮着访问受保护数据。

互斥量（Mutex）

- 使用互斥量的典型顺序如下：
  - 创建和初始一个互斥量
  - 多个线程尝试去锁定该互斥量
  - 仅有一个线程可以成功锁定改互斥量
  - 锁定成功的线程做一些处理
  - 线程解锁该互斥量
  - 另外一个线程获得互斥量，重复上述过程
  - 最后销毁互斥量
- 当多个线程竞争同一个互斥量时，失败的线程会阻塞在lock调用处。可以用“trylock”替换“lock”，则失败时不会阻塞。
- 当保护共享数据时，程序员有责任去确认是否需要使用互斥量。

互斥量

- 创建和销毁
  - pthread_mutex_init (mutex,attr)
  - pthread_mutex_destroy (mutex)
- 锁定和解锁
  - pthread_mutex_lock (mutex)　//若mutex已被锁，该线程被阻塞
  - pthread_mutex_trylock (mutex) //若mutex已被锁，程序会立刻返回，可一定程度上避免死锁
  - pthread_mutex_unlock (mutex)

例子2

两个向量$a = [a_1, a_2,\ldots, a_n]$和$b = [b_1, b_2,\ldots, b_n]$的点积定义为：$a \cdot b=a_1b_1+a_2b_2+\ldots\ldots+a_nb_n$

- 例子2演示了线程使用互斥量处理一个点积计算。
- 主数据通过一个可全局访问的数据结构被所有线程使用，每个线程处理数据的不同部分，主线程等待其他线程完成计算并输出结果。

```
typedef struct
{
   double    *a;   //a向量
   double    *b;   //b向量
   double    sum;   //点积
   int          veclen;   //分段长度
} DOTDATA;

#define NUMTHRDS 4
#define VECLEN 100

DOTDATA dotstr;  //点积变量
```

```
int main (int argc, char *argv[])
{
   int i;   double *a, *b;

   a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
   b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
   for (i=0; i<VECLEN*NUMTHRDS; i++)
   {     a[i] = 1.0;      b[i] = a[i];     }
   dotstr.veclen = VECLEN;
   dotstr.a = a;     dotstr.b = b;     dotstr.sum = 0;

    for( i =0; i<NUMTHRDS*VECLEN; ++i)  //计算点积
            dotstr.sum += dotstr.a[i] * dotstr.b[i];
    printf ("Sum =  %f /n", dotstr.sum);
    free (a);
    free (b);
    return 0;
}
```

传统方法

```
pthread_t callThd[NUMTHRDS]; //存放线程标示符
pthread_mutex_t mutexsum;  //定义互斥量

void *dotprod(void *arg)
{
  int i, start, end, offset, len ;
  double mysum, *x, *y;
  offset = (int)arg;
  len = dotstr.veclen;     start = offset*len;     end   = start + len;
  x = dotstr.a;      y = dotstr.b;     mysum = 0;
  for (i=start; i<end ; i++)        mysum += (x[i] * y[i]);

  pthread_mutex_lock (&mutexsum);  //加锁
  dotstr.sum += mysum;
  pthread_mutex_unlock (&mutexsum); //解锁

  pthread_exit((void*) 0);
  return NULL;
}
```

```
int main (int argc, char *argv[])
{
   ……
   pthread_mutex_init(&mutexsum, NULL);  //互斥量初始化

   for(i=0; i<NUMTHRDS; i++)
        pthread_create( &callThd[i], NULL, dotprod, (void *)i);
    /* Wait on the other threads */
    for(i=0; i<NUMTHRDS; i++)  //等待线程结束
      pthread_join( callThd[i], NULL);

   printf ("Sum =  %f /n", dotstr.sum);
    free (a);    free (b);

   pthread_mutex_destroy(&mutexsum);  //销户互斥量
   pthread_exit(NULL);                //主线程退出
    return 0;
}
```

等待条件发生

- 如果线程正在等待共享数据内某个条件出现，那会发生什么呢？
- 代码可以反复对互斥对象锁定和解锁，以检查值的任何变化。同时，还要快速将互斥对象解锁，以便其它线程能够进行任何必需的更改。这是一种非常可怕的方法，因为线程需要在合理的时间范围内频繁 地循环检测变化。

当线程在等待某些条件时使线程进入睡眠状态。一旦条件满足，能及时唤醒该线程？？

- 未完待续

中断机制

- 计算机打印业务，计算机不可能时刻的去监控着你的打印机状态！在打印机有打印需求时，计算机会产生一个中断信号，发送给CPU请求占用计算机核心态，进行打印操作！这个过程就是个中断过程！

中断机制

- 中断就是让cpu中断当前的正常指令而转去执行另一处特点的代码的一种机制。
- 中断向量表就是对应的中断号所对应的内存内址，某一中断发生后，CPU就去查这个表，从中取出一个地址，然后转去执行该地址处的指令。
- 中断的类型有硬件中断，如计时器中断，DMA中断，串口中断，和软件中断。

中断机制

- 有中断了，cpu一定要会中断当前的执行吗？不一定，有些中断是可以屏蔽的。
- CPU是通过设置时间中断来实现抢占机制的，在进入保护模式之前先初始化中断向量表，在时钟中断入口处放置任务切换代码。然后设置好时钟中断的时间。
- 当某一线程的执行用光了时间片时，时钟中断产生，cpu转去执行中断处的任务切换代码，保存当前线程的状态，得到并灰复下一个线程的状态，然后转去执行那个线程，依此类推...