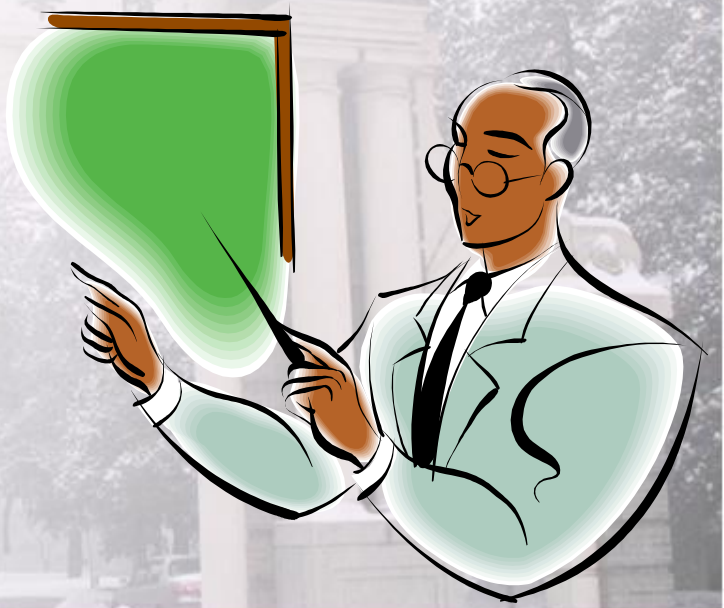
第五讲：多线程编程技术

清华大学计算机系

主要内容

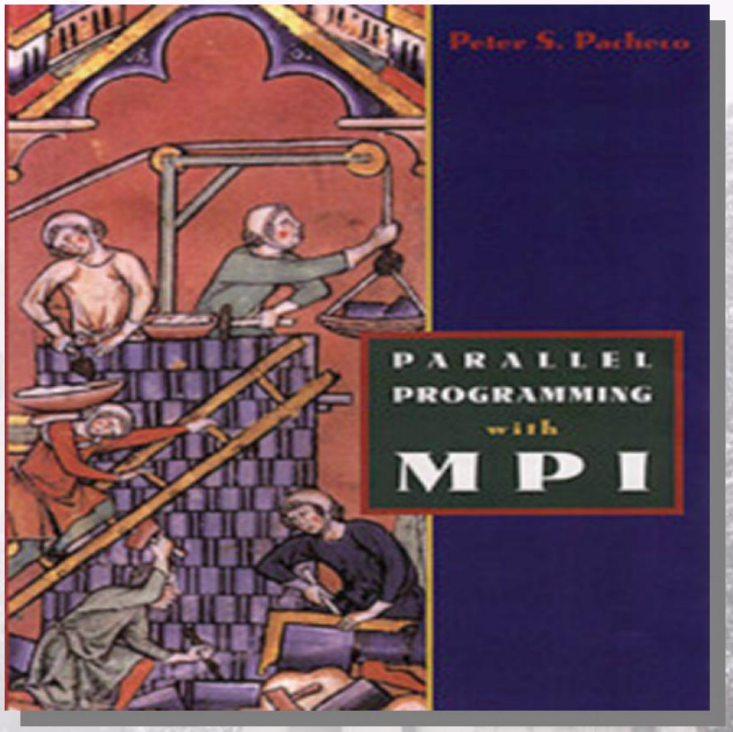
- 多任务介绍
- 进程与线程的概念
- Qt 多进程编程
- Qt 多线程编程
- 多线程的同步机制



并行:古老的思想!

- “... 并行计算并不是什么新的思想,只是将它扩展应用于计算机而已. 作者也不认为这种扩展应用会存在什么无法克服的困难. 但也不要期待有效的并行编程方法与技术能够在一夜之间诞生. 期间还需要有许多的工作和实验要做. 毕竟, 今天的编程技术(串行)是若干年来艰苦的探索才取得的. 现在编程工作似乎成了一种令人单调乏味的工作,事实上,并行编程的出现将会使重新恢复编程工作者们的探索精神 ...”

(Gill, S. (1958), “Parallel Programming,” The Computer Journal, vol. 1, April, pp. 2-10.)



Parallel Programming with MPI by Peter Pacheco(2000)

1、什么是多任务并行

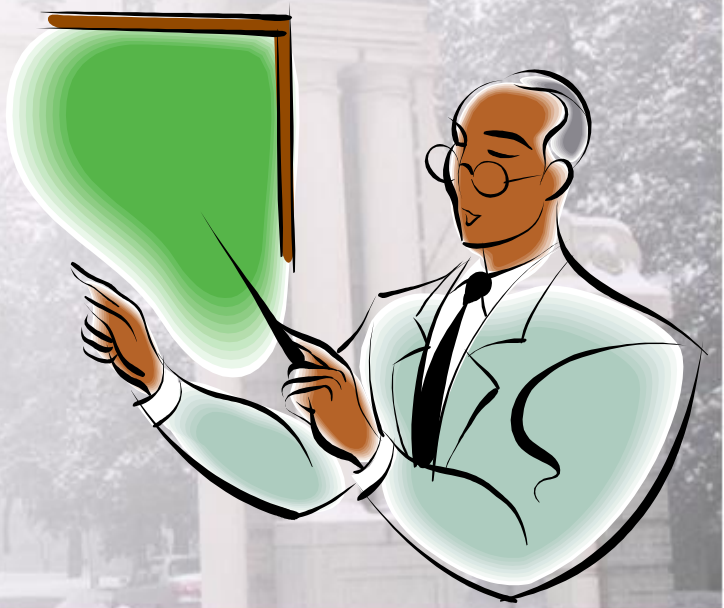
- 什么是多任务，生活中很常见
  - 妈妈：一边织毛衣，一边看电视
  - 售货员：招呼多个顾客看货、购物
  - 同学们：同时应付多门课的作业
- 计算机为什么需要支持多任务?
  - 大型机器：需要同时服务多位用户
  - 个人机器：同时有多个需求：听歌、上网
- 计算机为什么能支持多任务
  - CPU资源大量富余
  - memory/disk等设备速度很慢
  - 运行单个任务，会经常有大量资源闲置

多任务的交互——竞争与协作

- 竞争关系
  - 多个任务共用一套资源，因而出现多个任务竞争资源的情况。
  - 任务的互斥(Mutual Exclusion)是解决进程间竞争关系的手段
- 协作关系
  - 多个子任务为完成同一任务需要分工协作，一个子任务的继续执行依赖于另一个子任务的执行状态
  - 任务的同步(Synchronization)是解决任务间协作关系的手段。

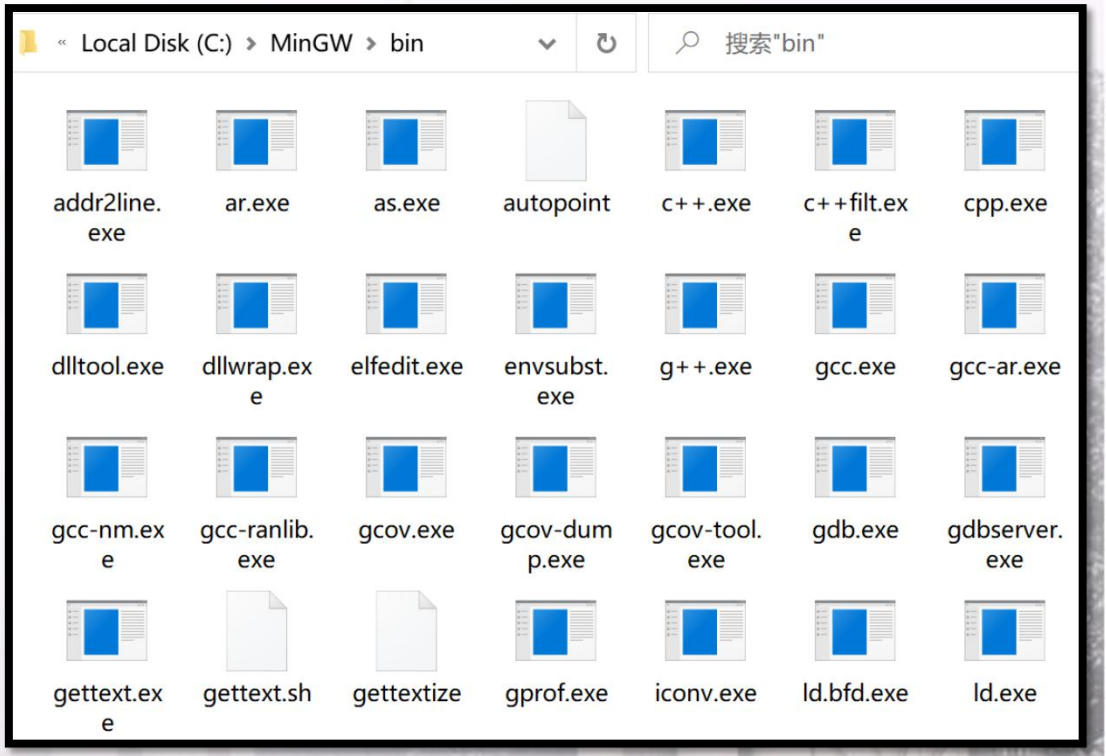
主要内容

- 多任务介绍
- 进程与线程的概念
- Qt 多进程编程
- Qt 多线程编程
- 多线程的同步机制



2、基本概念：程序

- 程序通常是指一个可供计算机执行的文件。
- 最常见的是以 .exe 或 .elf 为扩展名的文件。



2、基本概念：进程

- 程序是静态的概念，进程是程序的动态概念。
- 进程是应用程序的执行实例，描述程序的执行状态。
- 一个以exe作为扩展名的文件，在没有被执行的时候称之为应用程序。当用鼠标双击执行以后，就被操作系统作为进程来管理了。
- 当关机或者在任务栏的图标上单击鼠标右键选“退出”时，进程便消亡，彻底结束了生命。

2、Windows任务管理器—进程

任务管理器

文件(F) 选项(O) 查看(V)

进程 性能 应用历史记录 启动 用户 详细信息 服务

名称	状态	14% CPU	38% 内存	6% 磁盘	0% 网络
服务主机: Windows 推送通知系统服务		0%	1.8 MB	0 MB/秒	0 Mbps
服务主机: Local Session Manager		0.1%	1.8 MB	0 MB/秒	0 Mbps
服务主机: Storage Service		0%	1.7 MB	0 MB/秒	0 Mbps
服务主机: IP Helper		0%	1.7 MB	0 MB/秒	0 Mbps
服务主机: Microsoft Passport Container		0%	1.7 MB	0 MB/秒	0 Mbps
服务主机: Windows 许可证管理器服务		0%	1.7 MB	0 MB/秒	0 Mbps
服务主机: 本地系统		0%	1.7 MB	0 MB/秒	0 Mbps
记事本		0.1%	1.7 MB	0 MB/秒	0 Mbps
Windows Security Health Service		0%	1.7 MB	0 MB/秒	0 Mbps
Microsoft Windows Search Filter Host		0%	1.7 MB	0 MB/秒	0 Mbps
服务主机: Program Compatibility Assistant Service		0%	1.6 MB	0 MB/秒	0 Mbps
Client Server Runtime Process		0%	1.6 MB	0 MB/秒	0 Mbps
Runtime Broker		0%	1.6 MB	0 MB/秒	0 Mbps
服务主机: Windows Audio Endpoint Builder		0%	1.6 MB	0 MB/秒	0 Mbps
HuaweiHiSuiteService		0.1%	1.6 MB	0 MB/秒	0 Mbps
服务主机: 更新 Orchestrator 服务		0%	1.5 MB	0 MB/秒	0 Mbps
服务主机: WinHTTP Web Proxy Auto-Discovery Service		0.1%	1.5 MB	0 MB/秒	0 Mbps
服务主机: 蓝牙音频网关服务		0%	1.5 MB	0 MB/秒	0 Mbps
Windows 驱动程序基础 - 用户模式驱动程序框架主机进程		0%	1.5 MB	0 MB/秒	0 Mbps
服务主机: DHCP Client		0%	1.5 MB	0 MB/秒	0 Mbps
服务主机: cbdhsvc_6a3b3		0%	1.5 MB	0 MB/秒	0 Mbps
服务主机: 功能访问管理器服务		0%	1.5 MB	0 MB/秒	0 Mbps
服务主机: Geolocation Service		0%	1.5 MB	0 MB/秒	0 Mbps
服务主机: Time Broker		0%	1.5 MB	0 MB/秒	0 Mbps
服务主机: Phone Service		0%	1.5 MB	0 MB/秒	0 Mbps

简略信息(D)

进程的并发性

- 宏观上，所有进程都是并发运行的。
- 微观上，除非是多处理器或多核处理器，否则不可能有两个进程在同时运行。具体方法是时间片轮转：一个进程运行一个时间片，就把CPU让出来让另一个进程运行。因为时间片很小，所以用户看起来所有进程都在运行。
- 任何两个不相关的进程其推进速度可能是任意的。

并发带来的好处

- 很明显的好处：可以让多个用户分享CPU。对单用户而言，也可同时运行多个程序，如一边上网一边QQ。

- 更深层次的好处：充分利用CPU资源。
  - 当一个进程在等待数据时（来自网络，外部设备等），其它进程可占用CPU。

并发带来的挑战

- 挑战：并不是所有的事情都可以同时做。
- 两个进程同时写一个文件，对于普通文件，文件某一个位置上的内容是最后一次写入的结果。好像还不太糟。
- 但如果这个文件是一台打印机那将会怎么样？可以想像打印出来的东西将不是任何一个进程想得到的。
- 数据的不一致性

数据的不一致性

- 例：多个进程通过共享内存实现通信，共享一块物理地址。

- 每个进程都通过 int *p 映射到这块物理地址。进程每次获取一个网页，调用*p=*p＋1。最后*p的值就是多个进程获取到的网页总和。

数据的不一致性

- 进程1
  - mov eax, [p]
  - inc eax
  - mov [p], eax
- 进程2
  - mov eax, [p]
  - inc eax
  - mov [p], eax

结果不是我们想要的，*p只被加了1！

数据的不一致性

- 因为*p是共享资源，因此对它写操作应该是互斥的。访问文件也是类似。
- 不加同步控制的多进程程序，运行结果是不可预知的。
- 在编写多进程或多线程程序时应当特别注意。

引入多线程技术的动机

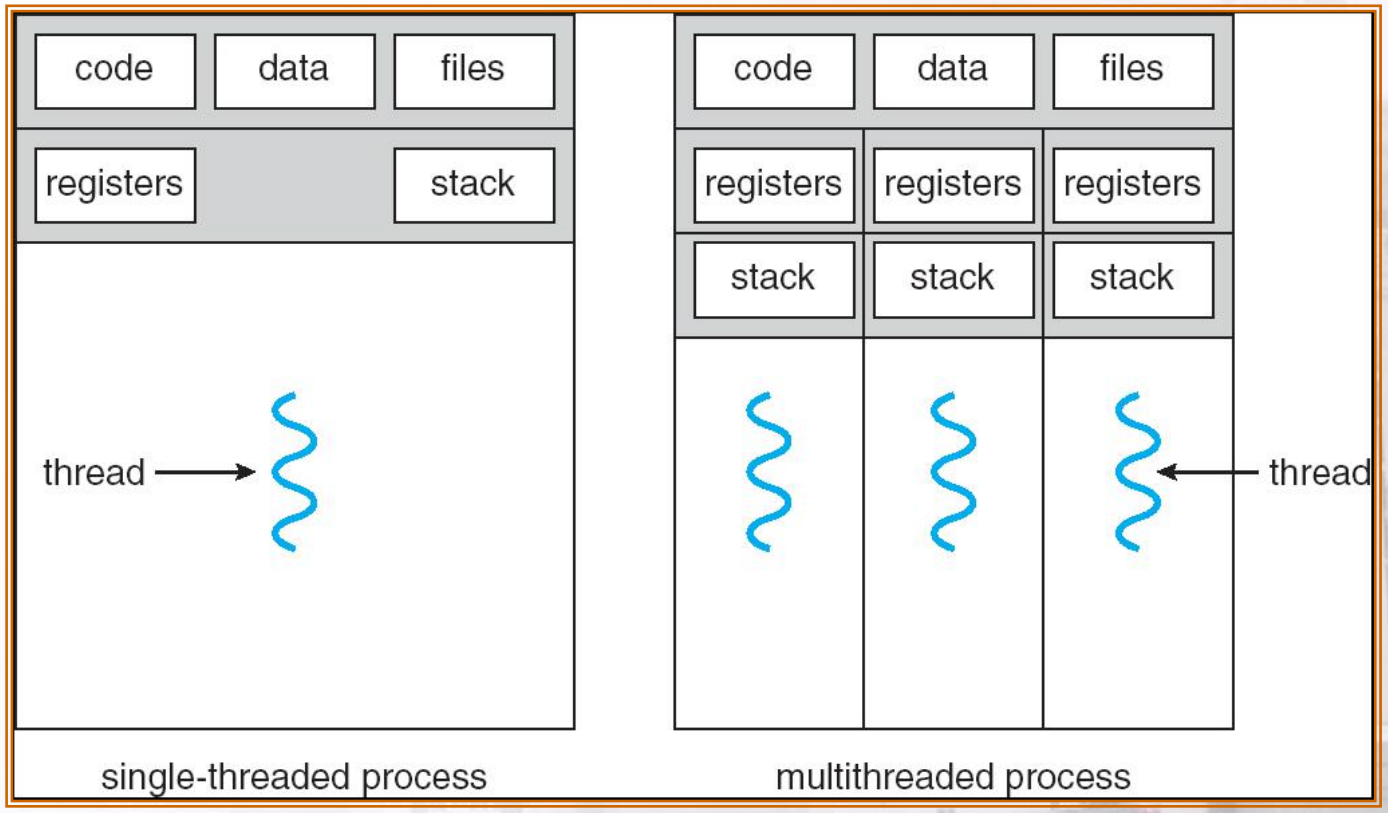
- 进程切换开销大，频繁的进程调度将耗费大量处理器时间
- 进程间通信代价大，每次通信均要涉及通信进程之间以及通信进程与操作系统之间的切换。
- 进程间的并发粒度较粗，并发度不高。过多的进程切换和通信使得细粒度的并发得不偿失。

轻量级进程（**Light Weight Process)**

——**线程（Thread）**

2、基本概念：线程

- 线程是进程的执行单元，一个进程内可以有多个线程
- 进程所具有的动态含义，是通过线程来体现的。

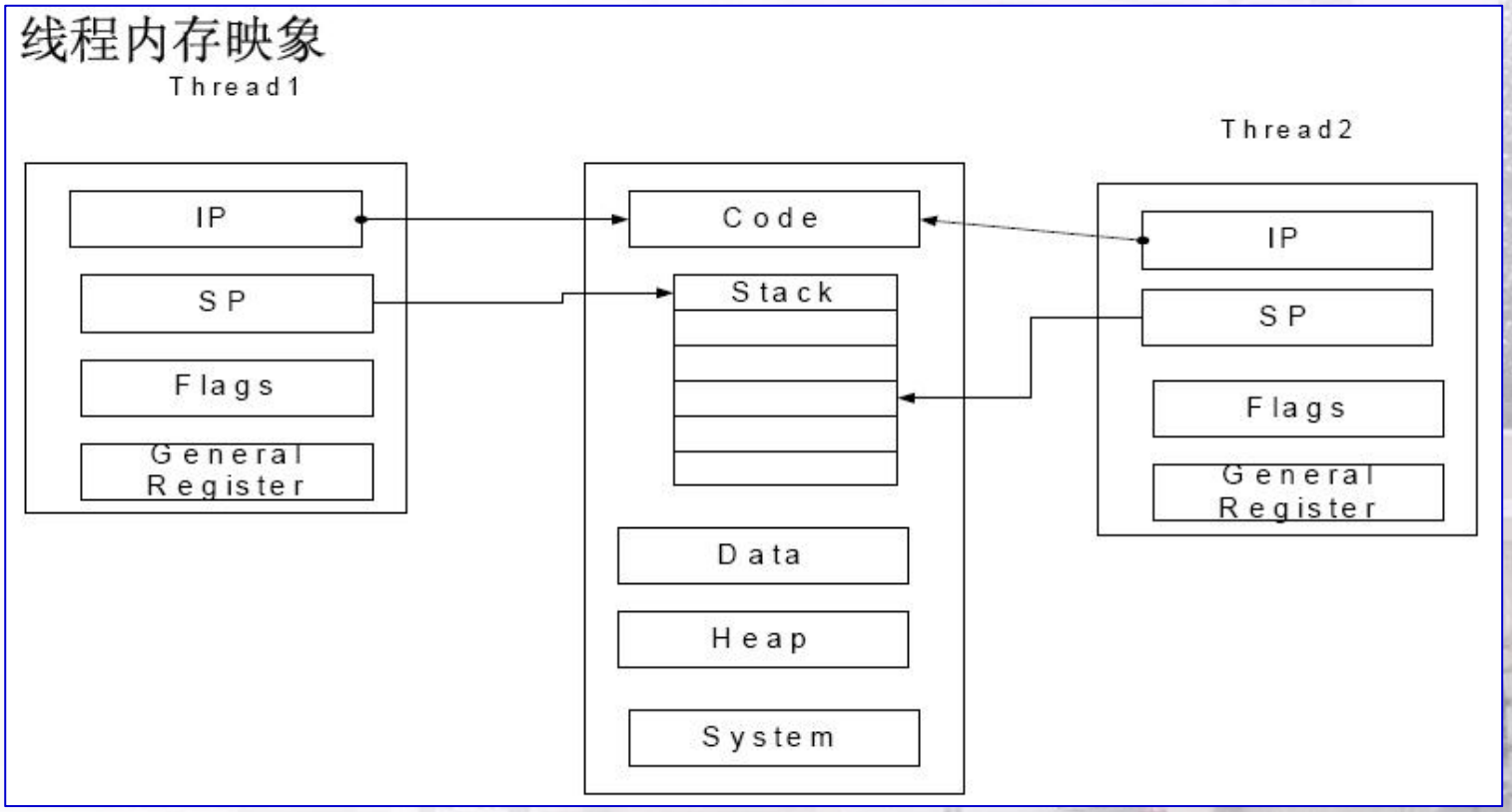


2、进程 vs 线程

- 进程是系统中程序执行和资源分配的基本单位
  - 每个进程有自己的数据段、代码段和堆栈段
- “独立地址空间”是指各个进程都有自己的虚拟地址空间（在Linux下为0x0-0xbfffffff），而且任何进程都只能访问到自己的虚拟地址空间。
- 线程通常叫做轻量级进程。线程是在共享内存空间中并发执行的多道执行路径
  - 多个线程共享一个进程的资源
- 因为线程和进程比起来很小，所以相对来说，线程花费更少的CPU资源。

线程内存映象和内容

- 线程存在于进程之中，除了堆栈和CPU状态外，全部数据是共享的

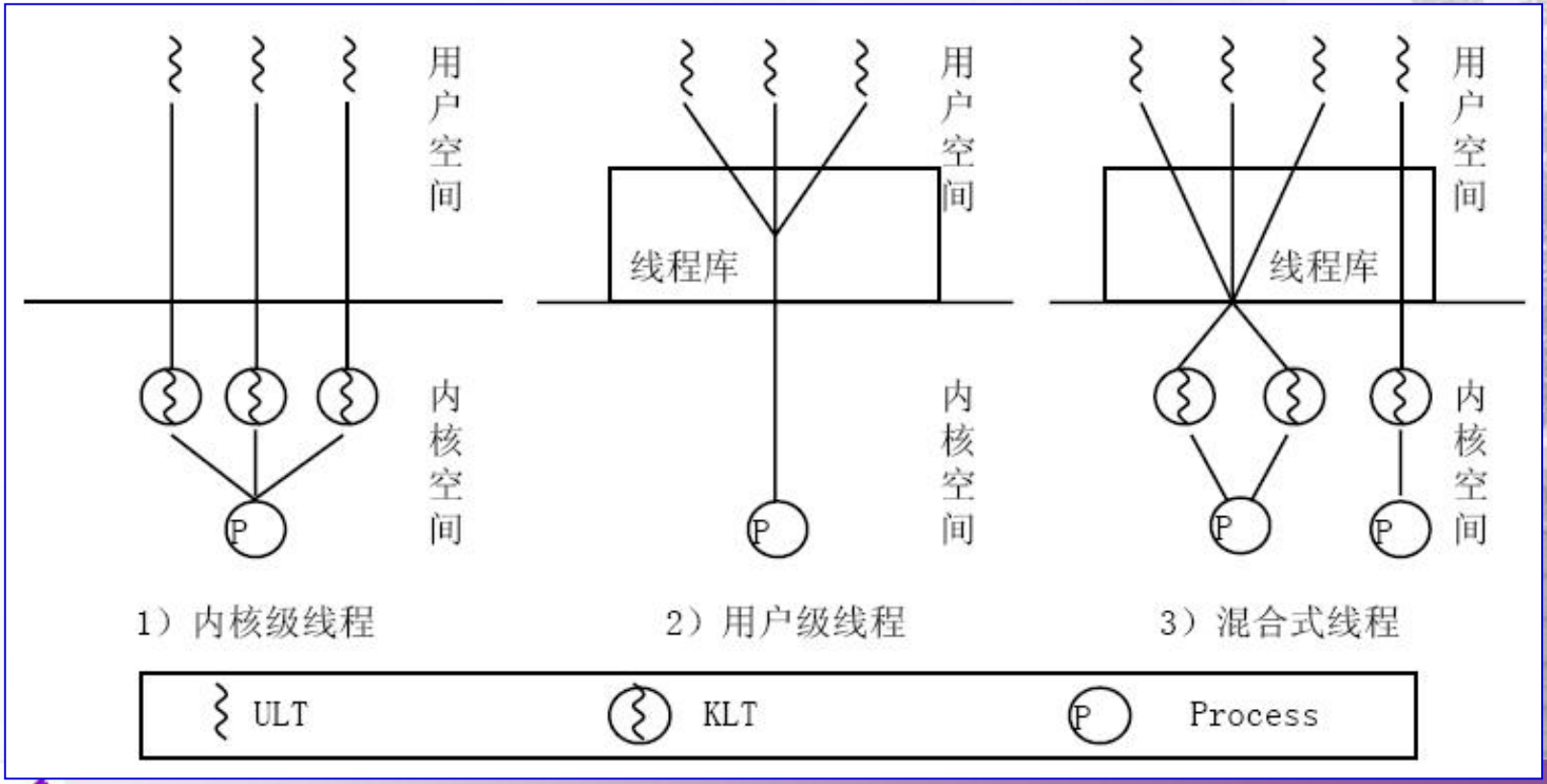


2、用户级线程 vs 内核级线程

- 线程按照其调度者可分为两种：
  （1）用户级线程：主要解决的是上下文切换的问题，其调度算法和调度过程全部由用户决定。
  （2）内核级线程：由内核调度机制实现。
- 现在大多数操作系统都采用用户级线程和内核级线程并存的方法。
- 用户级线程可与内核级线程实现“一对一”、“一对多”的对应关系。

线程的实现方法

- 从实现的角度看，线程可以分成用户级线程ULT(如，Java和Informix)和内核级线程KLT(如OS/2)。也有一些系统(如，Solaris)提供了混合式线程，同时支持两种线程实现



主要内容

- 多任务介绍
- 进程与线程的概念
- Qt 多进程编程
- Qt 多线程编程
- 多线程的同步机制

进程的启动

- QProcess可用于完成启动外部程序，并与之交互通信
- 启动外部程序的两种方式：
- （1）一体式：外部程序启动后，将随主程序的退出而退出。

  void QProcess::start(const QString & program, const QStringList & arguments, OpenMode mode = ReadWrite)
- （2）分离式：外部程序启动后，当主程序退出时并不退出，而是继续运行。

  void QProcess::startDetached(const QString & program, const QStringList & arguments, const QString & workingDirectory = QString(), qint64 * pid = Q_NULLPTR)
- 传递外部程序的路径和执行参数，参数用QStringList来带入。

进程的程序与参数

- 取得该进程上次启动的程序：

  QString QProcess::program() const

- 取得该进程上次启动程序时所带的参数：

  QString QProcess::nativeArguments() const

- 如果该进程正在运行，返回进程id；否则，返回0。

  qint64 QProcess::processId() const

取得和设置进程的运行状态

- 取得一个进程的当前状态：

  QProcess::ProcessState QProcess::state() const

- 设置一个进程的当前状态：

  void QProcess::setProcessState(ProcessState *state*)

Constant	Value	Description
QProcess::NotRunning	0	The process is not running.
QProcess::Starting	1	The process is starting, but the program has not yet been invoked.
QProcess::Running	2	The process is running and is ready for reading and writing.

进程的终止

- 终止一个进程有两种方法： kill( )和Terminate( )。
- 杀死当前进程，导致其立即退出。

  void QProcess::kill()

- 尝试结束当前进程。

  void QProcess::terminate()

  调用该函数当前进程未必退出，比如给机会提示用户文件未保存。

主要内容

- 多任务介绍
- 进程与线程的概念
- Qt 多进程编程
- Qt 多线程编程
- 多线程的同步机制

QThread中 的几个常用函数

函数	操作语义
start	启动执行一个新线程，通过调用run()函数
started	执行start时，在调用run函数之前发射该信号
exit，quit	结束一个线程的执行，停止事件处理循环
terminate	尝试终止一个线程的执行，可能有延迟
priority	得到线程的优先级
SetPriority	设置一个线程的优先级
sleep，msleep，usleep	强迫当前线程睡眠一段时间
yieldCurrentThread	让出CPU资源给其他线程
isFinished	线程运行是否结束
wait	等待一个线程的运行完成

线程的运行启动

- 线程通过调用run()函数开始执行：
  - void QThread::start(Priority priority = InheritPriority)
  - 操作系统将根据priority参数来调度该线程。
  - 如果该线程正在运行，则该函数什么也不做。
- 线程的启动点
  - void QThread::run()
  - 缺省的实现简单的调用exec()。
- 线程执行start时，在调用run函数之前发射信号：
  - void QThread::started()

线程的运行结束

- 告诉线程的事件处理循环，以给定的返回码退出
  - void QThread::exit(int returnCode = 0)
  - 线程离开事件处理循环，exec返回returnCode
  - 返回码为0，表示成功；否则，表示出错。
- void QThread::quit() = QThread::exit(0).

- 尝试终止一个线程的执行，可能有延迟
  - void QThread::terminate()
  - 线程是否立即终止，依赖于操作系统的调度策略
  - 为了确保线程结束，调用 QThread::wait()
  - 当线程真的结束了，所有“等待该线程完成”的线程都会被唤醒

操作线程优先级的函数

- 获得一个线程的优先级：
  - Priority QThread::priority() const
  - 如果该线程正在执行，则返回该线程的优先级；
  - 否则，返回 InheritPriority.

- 设置一个线程的优先级：
  - void QThread::setPriority(Priority priority)
  - 为一个正在运行的线程设置优先级。（如果线程没在运行，则什么都不做）
  - Use start() to start a thread with a specific priority.

线程的优先级

Constant	Value	Description
QThread::IdlePriority	0	scheduled only when no other threads are running.
QThread::LowestPriority	1	scheduled less often than LowPriority.
QThread::LowPriority	2	scheduled less often than NormalPriority.
QThread::NormalPriority	3	the default priority of the operating system.
QThread::HighPriority	4	scheduled more often than NormalPriority.
QThread::HighestPriority	5	scheduled more often than HighPriority.
QThread::TimeCriticalPriority	6	scheduled as often as possible.
QThread::InheritPriority	7	use the same priority as the creating thread. This is the default.

线程的运行状态

- 使线程休眠一定时间，时间到后线程被自动唤醒：
  - void QThread::sleep(unsigned long *secs*)
  - void QThread::msleep(unsigned long *msecs*)
  - void QThread::usleep(unsigned long *usecs*)
  - Sleep(0)可以暂时挂起自身，以运行同优先级线程

- 让出CPU资源给其他线程
  - void QThread::yieldCurrentThread()
  - 如果有可运行的其他线程，则当前线程让出CPU资源给其他线程
  - 具体让给哪个runnable的线程，由操作系统决定

线程运行是否结束

- 线程的运行是否结束
  - bool QThread::isFinished() const
  - 如果运行结束，返回true
- 等待线程运行的结束
  - bool QThread::wait(unsigned long time = ULONG_MAX)
  - 阻塞等待，直到（1）线程运行已经结束（或者未曾启动）；或者（2）超时
  - 若（1），返回true；若（2），返回false

QT图形界面与子线程通讯

- 在QT系统中，始终运行着一个GUI主事件线程
  - 负责从窗口系统中获取事件，并分发给各组件处理
- 如果主线程运行耗时的计算任务，将影响GUI的响应速度
  - 而阻塞的网络等待，将导致GUI僵死
- 解决办法：引入多线程技术，让子线程执行耗时的任务
  - 需要解决“QT图形界面与子线程通讯”的问题
- 在QThread与QWidget之间，使用signal-slot机制
  - 具体实现见示例代码
- 只有在QThread::run() 中的代码是在“子线程”中执行

主要内容

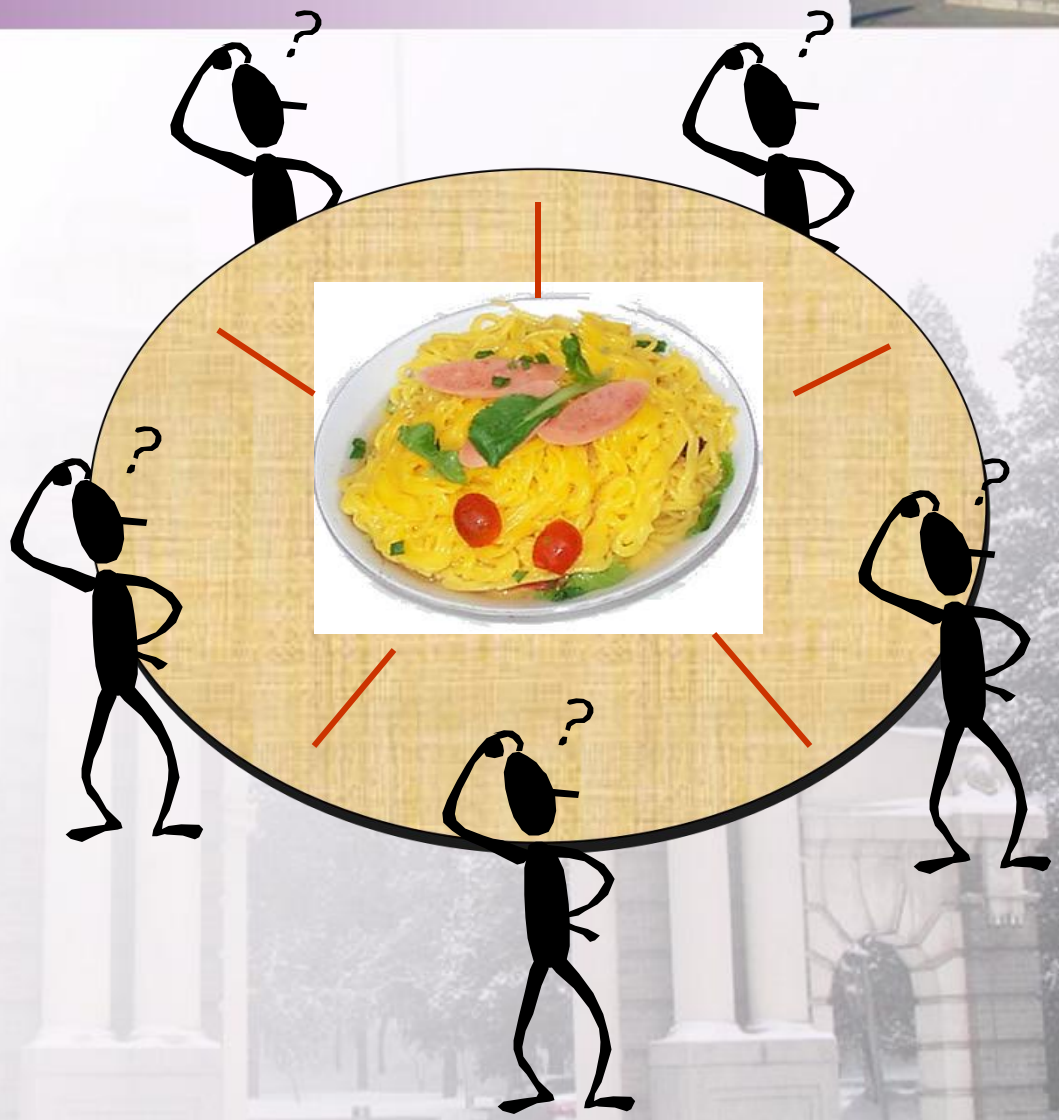
- 多任务介绍
- 进程与线程的概念
- Qt 多进程编程
- Qt多线程编程
- 多线程的同步机制

多任务的交互——竞争与协作

- 竞争关系
  - 多个任务共用一套资源，因而出现多个任务竞争资源的情况。
  - 任务的互斥(Mutual Exclusion)是解决进程间竞争关系的手段
- 协作关系
  - 多个子任务为完成同一任务需要分工协作，一个子任务的继续执行依赖于另一个子任务的执行状态
  - 任务的同步(Synchronization)是解决任务间协作关系的手段。

哲学家就餐问题：永远等待

- 并发的几个任务同时访问了一个不可重入代码段，结果不可预期
- 并发程序在时间上错误的两种表现形式：结果不唯一或者永远等待
- 哲学家就餐问题(永远等待)



订票问题：结果不唯一

- 假设一个飞机订票系统有两个终端，分别运行进程T1和T2。该系统的公共数据区中的一些单元Aj(j=l，2，…)分别存放某月某日某次航班的余票数，而x1和x2表示进程T1和T2执行时所用的工作单元，程序如下：
- 售票进程 Ti (i = 1, 2)

```
Int Xi;
{按旅客定票要求找到票源票数为Aj; Xi=Aj;};
if （Xi >=1）
    {Xi = Xi -1; Aj= Xi; 输出一张票；}
else
    {输出票已售完};
```

- 由于T1和T2是两个可同时运行的并发进程，它们在同一个计算机系统中运行，共享同一批票源数据，因此可能出现如下所示的运行情况：
  - T1: X1 = Aj; X1 = nn(nn> 0)
  - T2: X2 = Aj; X2 = nn
  - T2: X2=X2-1; Aj=X2;输出一张票； Aj= nn-1
  - T1: X1=X1-1; Aj=X1;输出一张票； Aj= nn-1
- 显然此时出现了把同一张票卖给了两个旅客的情况，两个旅客可能各自都买到一张同天同次航班的机票，可是，Aj的值实际上只减去了1，造成余票数的不正确。特别是，当某次航班只有一张余票时，就可能把这一张票同时售给了两位旅客，显然这是不能允许的。

线程的调度和同步

- Qt提供了一组对象用来实现多线程的同步，包括：
  - 信号-槽机制（signal-slot）
  - 互斥锁（QMutex）
  - 读写锁（QReadWrtieLock）

互斥锁

- QMutex是一种简单的加锁的方法来控制对共享资源的互斥访问：
  - 同一时刻只能有一个线程掌握某个互斥资源上的锁，
  - 拥有上锁状态的线程能够对共享资源进行访问。
  - 若其他线程希望对“一个已经被上了互斥锁的资源”上锁，则该线程挂起，直到上锁的线程释放互斥锁为止。
- 程序例子：

```
Thread1::run():
mutex->lock()
….
访问共享资源
….
mutex->unlock()
```

```
Thread2::run():
mutex->lock()
….
访问共享资源
….
mutex->unlock()
```

读写锁

- QReadWriteLock支持多个线程的并发读，而又能保证数据一致性
  - 当reader操作时，其它reader可以读取数据——保证最大并发性
  - 当reader操作时，其它writer无法写入数据——保证一致性
  - 当writer操作时，其他线程（包括reader和writer）无法读取或写入数据

```
reader::run():
rwlock-> lockForRead()
….
读取共享资源
….
rwlock->unlock()
```

```
writer::run():
rwlock->lockForWrite()
….
修改共享资源
….
rwlock->unlock()
```

Thank you!

Questions?