湘潭大学信息工程学院
The College of Information Engineering of Xiangtan University

TCPIP网络编程基础教程

王雷

博士、教授

教学大纲（四个部分）：

1.基于套接字的TCP/IP网络通信原理与模型

套接字的基本概念、基于套接字的网络通信实现流程、BSD UNIX套接字API函数、 Windows 套接字API函数、 TCP/IP网络通信模型及其C语言实现方法

2.循环服务器软件的实现原理与方法

循环服务器软件的实现原理及其C语言实现方法

3.服务器与客户进程中的并发机制与实现方法

服务器与客户进程中的并发机制、多进程并发机制的实现原理与方法、多线程并发TCP服务器软件的实现原理与方法、单线程并发机制的实现原理与方法、基于POOL和EPOLL的并发机制与实现方法

4.客户/服务器系统中的死锁问题与解决方法

死锁的定义、产生死锁的原因、处理死锁的基本方法

第三章：服务器与客户进程中的并发机制

3.1 服务器与客户进程中的并发概念

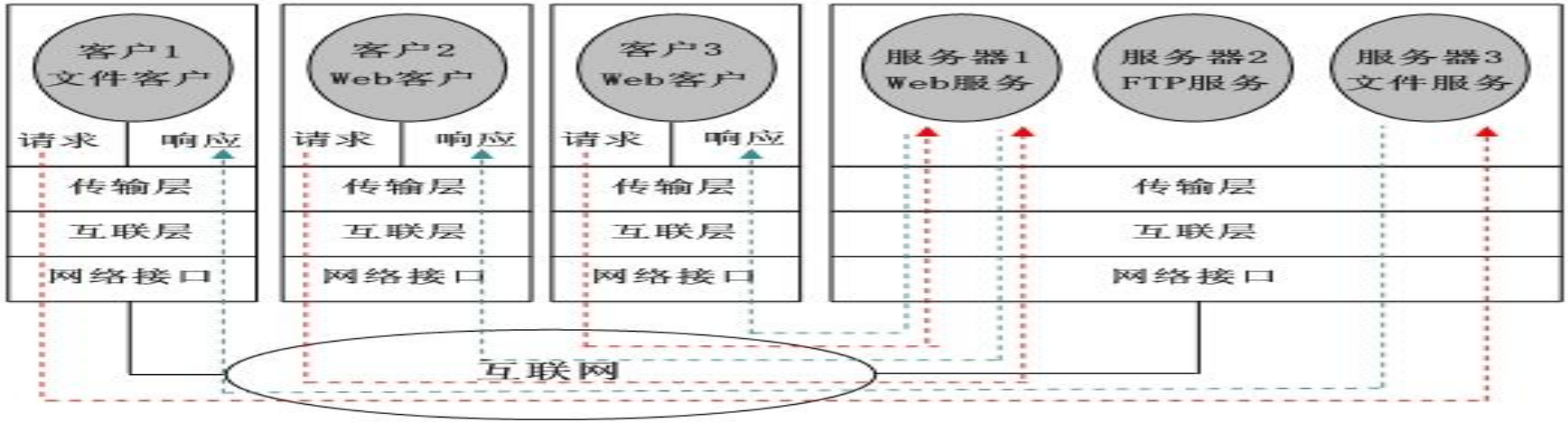
3.1.1服务器进程中的并发问题

并发（Concurrency）是指真正的或表面呈现的同时计算。通常，一个多用户的计算机系统可以通过分时（Time Sharing）或多处理器（Multiprocessing）来获得并发。

其中，分时机制是使得单个处理器在多个计算任务（或多个用户）之间快速地切换，从而使得从表面上来看这些计算（或这些用户所获得的服务）是同时进行的；而多处理器机制则是让多个处理器同时执行多个任务，因此，所实现的是真正的同时计算（即真正的并发）。

第三章：服务器与客户进程中的并发机制

在客户/服务器模型中，很多时候会有多个客户使用服务器的一个熟知协议端口与服务器联系，如图3.1所示，在一台主机上有可能会运行有多个服务器进程，并且每个服务器进程也可能需要及时处理多个客户的请求，并将处理的结果返回给客户，因此，服务器软件还必须在设计中处理好并发请求。



第三章：服务器与客户进程中的并发机制

为了理解服务器中并发的重要性，考虑一下需要大量计算或通信的服务器操作。例如：设想一个远程登录服务器，如果其不能并发运行，而是一次只能处理一个远程登录。此时，一旦有一个客户与该服务器建立了联系，则服务器在第一个用户结束会话之前，必须忽略或拒绝所有其它客户的请求。显然，这样的设计限制了服务器的使用效率，而且还使得多个远程用户不能在同一时间对该服务器进行访问。

3.1.2客户进程中的并发问题

在开发客户/服务器体系结构的系统过程中，由于以下原因，使得设计人员往往重视服务器端的并发设计：

※ 并发可改善观察到的时间，从而改善全部客户机的总吞吐量。

※ 并发可排除潜在的死锁。

第三章：服务器与客户进程中的并发机制

※ 并发实现使得设计人员易于创建多协议的或多服务的服务器。

※ 使用多进程实现并发非常灵活，因为这样就可以在多种硬件平台上很好地运行。当把并发实现移植到具有多个处理器的计算机时，可以得到更高的工作效率，因为可以充分利用额外的处理能力而不需要改变代码。

由于客户端通常在一个时刻只进行一种活动，客户端一旦向服务器发送了一个请求，在收到响应之前一般不需进行其他活动。因而客户端似乎不能从并发中受益。此外，客户端的效率和死锁问题也不如服务器那样严重，因为如果一个客户端延缓或停止执行，它只是自己停止了，而其他客户端将继续运行。然而，尽管表面上如此，由于以下原因，在客户端中采用并发确实有其优点：

第三章：服务器与客户进程中的并发机制

※ 由于功能已被划分为概念上能分开的一些部分，并发实现更容易编程。

※ 由于代码已经模块化了，并发实现可使维护和扩展变得更容易。

※ 并发客户端可在同一时刻联系多个服务器，往往需要比较响应时间或合并服务器返回的结果。

※ 并发允许用户改变参数，查询客户端状态，或动态地控制处理。

※ 在客户端中使用并发的最主要的优点在于异步性。异步性允许客户端同时处理多个请求，且不严格规定其执行顺序。

由以上描述可见，并发执行提供了一个强有力的工具。并发客户端实现不但可提供更快的响应时间，而且还可避免死锁问题，帮助程序员将控制和状态处理从正常的处理中分离出来。

第三章：服务器与客户进程中的并发机制

3.1.3服务器与客户端并发性的实现方法

在UNIX/LINUX与Windows环境下，主要提供了以下两种方法来实现服务器与客户端的并发性：一种是服务器或客户端创建多个进程（Process），每个进程都有一个线程（Thread），使得不同进程中的多个线程并行工作以完成多项任务，从而以提高系统的效率；另一种则是服务器或客户端在一个进程中创建多个线程，使得同一进程中的多个线程并行工作以完成多项任务，从而以提高系统的效率。

其中，进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程是系统进行资源分配和调度的一个独立单位，是具有一定独立功能的程序关于某个数据集合上的一次运行活动。由于每个进程都拥有自己独立的地址空间，因此，当一个进程崩溃后，在保护模式下它不会对其它进程产生影响。

第三章：服务器与客户进程中的并发机制

而线程则是进程的一个实体，是CPU调度和分派的基本单位，它是比进程更小的能独立运行的基本单位。但线程自己基本上不拥有系统资源，但可与同属一个进程内的其他线程共享该进程所拥有的全部资源。

由于线程除了只拥有一点在运行中必不可少的资源（如程序计数器、一组寄存器和栈）之外，没有单独的地址空间，因此一个线程的崩溃也就等于整个进程的崩溃。这也意味着多进程的服务器或客户端要比多线程的服务器或客户端健壮，但线程间彼此切换所需的时间要远远小于进程间切换所需要的时间，因此，多进程服务器或客户端的效率也就要比多线程的服务器或客户端差。

第三章：服务器与客户进程中的并发机制

3.1.4循环服务器与并发服务器

由第2章的介绍可知，循环服务器是指服务器在同一时刻只可以响应一个客户端的请求的服务器；而在网络程序中，通常都是有多个客户端对应同一个服务器，因此，为了使服务器可以同时处理来自多个客户的请求，人们提出了并发服务器的概念。其中，所谓并发服务器就是指在同一个时刻可以处理来自多个客户端的请求的服务器。

由于用户需求、处理速率和通信能力的不同，往往在循环的和并发的服务器设计之中难以选择。

第三章：服务器与客户进程中的并发机制

循环服务器采用客户轮流等待的工作方式，具有设计、编程、调试和修改简单的有点，因此，在其响应时间可以满足需求的条件下（该时间可以在本地或全局网络中进行测试），可以采用循环服务器模式。

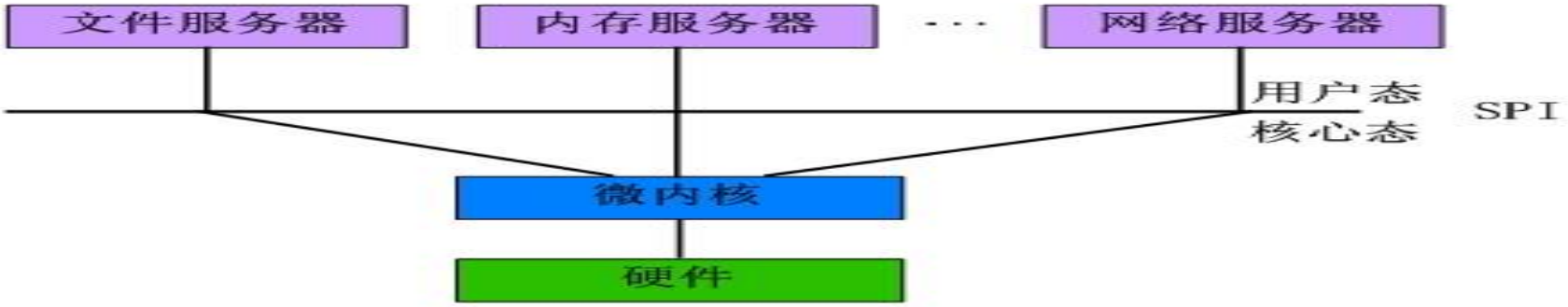
如果构建一个响应需要大量的I/O操作，且各个请求所需要的处理时间差别非常大，或服务器在一台多处理器的计算机上运行，则可采用并发服务器模式来缩短响应时间。

3.1.5多进程与多线程并发概念

当前计算机技术发展的突出特点是要求对广泛的信息与其它各类资源实现共享，从而促使了网络技术的普遍应用和快速发展，进而也要求操作系统必须为用户提供一个符合信息处理要求的分布式计算环境。因此现代操作系统一般均采用了微内核（Microkernel）结构。

第三章：服务器与客户进程中的并发机制

其中，微内核是指操作系统的小核心，它将各种操作系统共同需要的核心功能提炼出来，形成微内核的基本功能。这些操作系统的基本功能包括：IPC（Inter-Process Communication，进程通信），VM（Virtual Machine，虚拟机），Tasks（任务）管理、threads（线程）管理，中断处理及与硬件相关部分等。这样一来，从功能上而言，微内核为各种操作系统打好了一个公共基础，或者说构成了基本操作系统。其中，微内核操作系统的模型如下图3.2所示。



第三章：服务器与客户进程中的并发机制

由图3.2可见，微内核在核心态下工作，负责直接与硬件打交道；而操作系统的其它功能则由各服务器（除内核以外操作系统的其它部分被分成若干相对独立的进程，每个进程完成一组服务，称为服务器进程，简称为服务器）实现，服务器处于微内核之上，在用户态下工作。

各服务器同处一层，通过SPI（Service Provider Interface，服务提供者接口）与微内核联系。各服务器之间相互独立但彼此间可以直接通信。微内核负责对整个操作系统中的各种来往消息进行验证，在各大部分之间进行消息传递，并保证它们对硬件的访问。

第三章：服务器与客户进程中的并发机制

在微内核系统中，进程只是资源分配的单位，而真正可以在处理器（CPU）上独立调度运行的基本单位是线程。在多处理器系统中，每个线程在一个处理器上运行，从而实现应用程序的并发，使每个处理器都得到充分运行。因此，实际实现并发功能的是线程。进程和线程各自的优缺点可简单总结如下：

（1）进程优点：编程、调试简单，可靠性较高。

（2）进程缺点：创建、销毁、切换速度慢，内存、资源占用大。

（3）线程优点：创建、销毁、切换速度快，内存、资源占用小。

（4）线程缺点：编程、调试复杂，可靠性较差。

第三章：服务器与客户进程中的并发机制

采用多进程或多线程的方式均可实现服务器与客户端的并发，但由以上关于进程和线程各自的优缺点的描述可知，进程和线程有着各自的特点，因此，在C/S通讯中并发技术的选型上，到底是应该采用多线程还是该采用多进程并发技术呢？这样的争执由来已久。例如：

在WEB服务器技术中，Apache是采用多进程的（每个客户连接对应一个进程，每个进程中只存在唯一一个执行线程），而Java的Web容器Tomcat、Websphere等则都是采用多线程的（每个客户连接对应一个线程，所有线程都在同一个进程之中）。

第三章：服务器与客户进程中的并发机制

3.1.6并发等级

在并发服务器模式中，由于每一个访问连接都需要耗费一定的系统资源，过多的并发连接会将服务器系统资源消耗殆尽，从而导致服务器无法正常处理每一个客户络连接请求。

为了避免服务器系统无法响应，有必要在服务器系统中对并发连接数量进行适当控制，以保证服务器能够有足够系统资源来处理每一个客户连接请求。其中，在某个给定时刻一个服务器中正在运行着的执行线程总数，我们称之为该服务器的并发等级。

第三章：服务器与客户进程中的并发机制

为了处理一个传入的客户连接请求，并发服务器均需创建一个新的从线程/进程，在处理完该请求之后，该从线程/进程再退出。因此，并发服务器的并发等级是随时间变化的。

显然，服务器在任一时刻的并发等级反映了服务器已经收到、但还未处理完毕的客户请求的数目。

不过，在程序设计中程序员一般无需关心某个服务器在某个给定时刻的并发等级，而只需关心服务器在整个生命周期中所展现出来的并发等级的最大值。

第三章：服务器与客户进程中的并发机制

3.2 UNIX/LINUX环境下基于多进程并发机制

3.2.1创建一个新进程

在UNIX/LINUX环境下，一个现有的进程可以调用fork()函数来创建一个新的进程。从本质上来说，fork函数将运行着的程序分成两个（几乎）完全一样的进程，每个进程都启动一个从代码的同一位置开始执行的线程。其中，由fork()函数所创建的新进程被称为子进程（Child Process），而调用fork()函数的进程则称为父进程（Parent Process）。fork()函数的原型如下：

```
#include <unistd.h>     /* LINUX标准头文件，包含了各种LINUX
                           系统服务函数原型和数据结构的定义*/
int fork();
```

第三章：服务器与客户进程中的并发机制

成功调用fork()函数之后，操作系统会复制出一个与父进程（几乎）完全相同的新进程，不过这两个进程虽说是父子关系，但是在操作系统看来，它们更像兄弟关系，这两个进程共享代码空间，但是数据空间是互相独立的，子进程数据空间中的内容是父进程的完整拷贝，指令指针也完全相同，子进程拥有父进程当前运行到的位置，也就是说子进程是从fork()函数返回处开始执行的，但唯一不同的是，若fork()函数调用成功，则在子进程中其返回值为0，而在父进程中其返回值为子进程的进程号（进程ID）；若调用不成功，则在父进程中其返回值为-1。调用fork()函数创建一个新进程的基本方法主要有以下两种：

第三章：服务器与客户进程中的并发机制

1. 调用 fork()函数创建一个新进程的基本方法之一

```
int pid;
pid = fork();                          //调用 fork()函数创建一个新进程
if(pid == -1 ) {                       //若调用 fork()函数出错
        perror ("fork failed!");
        exit(1);
}
else if( pid = =0 ) {                  //以下是子进程所执行的操作
        printf("This is the child process!");
} else {                               //以下是父进程所执行的操作
        printf("This is the parent process!");
}
```

第三章：服务器与客户进程中的并发机制

2. 调用 fork()函数创建一个新进程的基本方法之二

```
int pid;
pid=fork();
switch (pid) {
        case -1:                                    //若调用 fork()函数出错
             perror("fork failed!");
             exit(1);
        case 0:                                     //以下是子进程所执行的操作
             printf("This is the child process!");
             break;
        default:                                    //以下是父进程所执行的操作
             printf("This is the parent process!");
}
```

第三章：服务器与客户进程中的并发机制

由以上给出的调用fork()函数的代码段可知，在调用fork()函数创建一个子进程之后，父子进程之间的关系可以这样想象：在fork()函数返回之前，这两个进程一直在同时运行，而且步调也保持着一致，但在fork()函数返回之后，它们虽然仍是在同时运行，但从此分别开始做不同的工作，也就是分岔了，这也是fork()为什么叫做fork的原因。

不过在fork()函数返回之后，父子进程在同时运行时，到底是父进程先运行、还是子进程先运行，这就与操作系统的实际运行情况有关了。即：上述代码段的运行结果到底是先输出"This is the child process!"，还是先输出"This is the parent process!"是不确定的，则与操作系统的实际运行情况有关。

第三章：服务器与客户进程中的并发机制

3.2.2终止一个进程

在Linux系统中，进程终止分为了正常终止和异常终止两种，可通过调用exit()函数来实现。exit()函数在头文件stdlib.h中声明，其函数原型如下：

```
#include <stdlib.h>
void exit(int status);
```

上述exit()函数可用来终止当前进程的执行，并把参数status返回给当前进程的父进程，而当前进程所有的缓冲区数据将会被自动写回并关闭所有未关闭的文件。其中，exit(0)表示程序正常终止，而exit(1)/exit(-1)则表示程序出错/异常终止。

第三章：服务器与客户进程中的并发机制

3.2.3获得一个进程的进程标识

进程标识也称为进程号或进程ID，可通过getpid()函数来获得，

getpid()函数的原型如下：

```
#include<unistd.h>
pid_t getpid(void);
```

getpid()函数的返回值即为当前进程的进程号。

3.2.4获得一个进程的父进程的进程标识

一个进程的父进程的进程标识可通过getppid()函数来获得，

getppid()函数的原型如下：

```
#include<unistd.h>
pid_t getppid(void);
```

getppid()函数的返回值即为当前进程的父进程的进程号。

第三章：服务器与客户进程中的并发机制

3.2.5僵尸进程的清除

1. 僵尸进程的定义与清除方法

一个进程在调用exit()函数结束自己的生命的时候，操作系统内核仍然会在进程表中为其保留一定的信息（包括进程号、退出状态、运行时间等）。由于这类进程已经放弃了几乎所有内存空间，没有任何可执行代码，也不能被调度，仅仅继续占用了系统的进程表资源，除此之外不再占有任何的内存空间，因此被称为僵尸进程（Zombie Process）。在Linux中，利用命令ps命令可以看到有标记为Z的进程就是僵尸进程。

第三章：服务器与客户进程中的并发机制

由于僵尸进程需要占用系统的进程表资源，但Linux系统对运行的进程数量有限制，如果产生过多的僵尸进程占用了可用的进程号，将会导致新的进程无法生成。为此，有必要对僵尸进程进行及时清除。

僵尸进程的清除工作一般是由其父进程来负责进行的，当父进程调用fork()函数创建了子进程后，主要可通过如下几种方法来避免产生僵尸进程：

（1）父进程可通过调用wait()或waitpid()等函数来等待子进程结束，从而避免产生僵尸进程，但这会导致父进程被挂起（即父进程被阻塞，处于等待状态）；

第三章：服务器与客户进程中的并发机制

（2）如果父进程很忙而不能被挂起，那么可以通过调用signal()函数为SIGCHLD信号安装handler来避免产生僵尸进程。因为当子进程结束后，内核将会发送SIGCHLD信号给其父进程，而父进程在收到该信号之后，则可以在handler中调用wait()函数来进行回收；

（3）如果父进程不关心子进程何时结束，那么可以通过调用signal(SIGCHLD, SIG_IGN)函数来通知内核，表明自己对子进程的结束不感兴趣，那么子进程结束后将会被内核自动回收，且不会再给父进程发送SIGCHLD信号，由此可以避免产生僵尸进程；

第三章：服务器与客户进程中的并发机制

（4）由于当一个父进程死后，其子进程将成为"孤儿进程"，从而会被过继给1号进程init，init是系统中的一个特殊进程，其进程ID为1，主要负责在系统启动时启动各种系统服务以及子进程的清理，只要有子进程终止，init就会调用wait函数清理它。因此，当一个父进程死后，其产生的僵尸进程也会被过继给1号进程init，再由init进程负责自动清理，这样一来，就使得一个父进程也可通过fork两次来避免产生僵尸进程，具体实现步骤如下：**首先**，父进程fork一个子进程并继续工作，**然后**，该子进程再在fork了一个孙进程之后退出，由于该孙进程将会被init进程接管，因此当该孙进程结束之后，将会被init进程自动回收。当然，子进程的回收工作还得由父进程来负责。

第三章：服务器与客户进程中的并发机制

2. 清除僵尸进程的相关函数定义

（1）wait()函数

wait()函数的原型如下：

```
#include <sys/types.h>                    //提供数据类型定义
#include <sys/wait.h>                     //提供wait()函数的原型定义
pid_t wait(int *status);
```

在上述wait()函数的原型之中，各参数的含义如下：

※ **status**：用来保存被收集进程退出时的一些状态，它是一个指向int类型的指针。可以通过调用以下宏来判别子进程的结束情况：

第三章：服务器与客户进程中的并发机制

① WIFEXITED(status)：子进程正常结束则该宏将返回非0值；

② WEXITSTATUS(status)：若子进程正常结束则利用该宏可获得子进程由exit()返回的结束代码；

③ WIFSIGNALED(status)：子进程因为信号而结束则该宏将返回非0值；

④ WTERMSIG(status)：若子进程因为信号结束则利用该宏可获得子进程的中止信号代码;

⑤ WIFSTOPPEN(status)：子进程处于暂停执行状态则该宏将返回非0值；

⑥ WSTOPSIG(status)：若子进程处于暂停状态则利用该宏可获得引发子进程暂停的信号代码。

第三章：服务器与客户进程中的并发机制

进程一旦调用了wait()函数，就立即阻塞自己，由wait()函数自动分析是否当前进程的某个子进程已经退出，如果让它找到了这样一个已经变成僵尸的子进程，wait()函数就会收集这个子进程的信息，并把它彻底销毁后返回；如果没有找到这样一个子进程，wait()函数就会一直阻塞在这里，直到有这样一个子进程出现为止。如果wait()函数调用成功，将会返回被收集的子进程的进程ID，如果调用失败则返回-1。

（2）**waitpid()函数**

waitpid()函数的原型如下：

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

第三章：服务器与客户进程中的并发机制

在上述waitpid()函数的原型之中，各参数的含义如下：

※ **pid**：是指需要等待的那个子进程的进程号。当pid取不同的值时有不同的意义：

① pid>0时，只等待进程ID等于pid的子进程，不管其它已经有多少子进程运行结束退出了，只要指定的子进程还没有结束，waitpid()就会一直等下去。

② pid=-1时，等待任何一个子进程退出，没有任何限制，此时waitpid()和wait()的作用完全等同。

③ pid=0时，等待同一个进程组中的任何子进程，如果子进程已经加入了别的进程组，waitpid()不会对它做任何理睬。

④ pid=-1时，等待一个指定进程组中的任何子进程，这个进程组的ID等于pid的绝对值。

第三章：服务器与客户进程中的并发机制

※ **status**：用来保存被收集进程退出时的一些状态，它是一个指向int类型的指针。

※ **options**：提供了一些额外的选项来控制waitpid()，主要包括WNOHANG和WUNTRACED等选项，这些选项可以用"|"运算符连接起来使用，若不想使用这些选项，也可将参数options置为0。其中，若将参数options置为WUNTRACED，则当子进程处于暂停状态，waitpid()将马上返回；若将参数options置为WNOHANG，则即使没有子进程退出，waitpid()也将立即返回；而若将参数options置为0，则waitpid()会像wait()那样阻塞父进程，直到所等待的子进程退出。

第三章：服务器与客户进程中的并发机制

waitpid()的返回值比wait稍微复杂一些，一共有3种情况：（1）当正常返回的时候，waitpid返回收集到的子进程的进程ID；（2）如果设置了选项WNOHANG，而调用中waitpid发现没有已退出的子进程可收集，则返回0；（3）如果调用中出错，则返回-1，这时errno会被设置成相应的值以指示错误所在。

为了进一步具体阐述清楚wait()和waitpid()函数的用法，下面分别举一个例子来加以说明：

第三章：服务器与客户进程中的并发机制

例 1（wait()函数的用法）：

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main() {
    int status;
    pid_t pc, pr;
    pc = fork();                    //调用 fork()函数创建一个子进程
    if (pc < 0) {                   //若创建子进程失败
        printf("fork failed");
        exit(-1);                   //退出程序
    }
```

第三章：服务器与客户进程中的并发机制

```
else if(pc == 0) {                          //子进程中执行以下代码段
    int i;
    for (i = 3; i > 0; i--) {
        printf("This is the child\n");
        sleep(5);
    }
    exit(3);                                //终止子进程，并给父进程返回终止代码 3
} else {                                    //父进程中执行以下代码段
    pr = wait(&status);                     //调用 wait()等待子进程终止
    if(WIFEXITED(status)){                  //若子进程正常结束
```

第三章：服务器与客户进程中的并发机制

```
            printf("The child process %d exit normally. \n",pr);
            printf("The WEXITSTATUS return code is %d \n",
                    WEXITSTATUS(status));
            printf("The WIFEXITED return code is %d \n",
                    WIFEXITED(status));
        }else                                   //若子进程非正常结束
            printf("The child process %d exit abnormally.\n", pr);
        printf("Status is %d.\n", status);
    }
    return 0;
}
```

第三章：服务器与客户进程中的并发机制

例 2（waitpid()函数的用法）：

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main(){
    pid_t pid;
    pid = fork();                          //调用 fork()函数创建一个子进程
    if (pid < 0) {                         //若创建子进程失败
        printf("fork failed");
        exit(-1);                          //退出程序
    }
    else if (pid == 0) {                   //子进程中执行以下代码段
        int i;
```

第三章：服务器与客户进程中的并发机制

```
        for (i = 3; i > 0; i--) {
            printf("This is the child\n");
            sleep(5);
        }
        exit(3);                        //终止子进程，并给父进程返回终止代码 3
    } else {     //父进程中执行以下代码段
        int stat_val;
        waitpid(pid, &stat_val, 0);              //调用 waitpid()等待子进程终止
        if (WIFEXITED(stat_val))                 //若子进程正常结束
            printf("Child exited with code   %d\n", WEXITSTATUS(stat_val));
        else if (WIFSIGNALED(stat_val))  //若子进程因为信号而结束
            printf("Child    terminated    abnormally,    signal    %d\n",
WTERMSIG(stat_val));
    }
    return 0;
}
```

第三章：服务器与客户进程中的并发机制

（3）signal()函数

signal()函数的原型如下：

```
#include<signal.h>                    //提供signal函数原型的定义
void (*signal(int signum,void(* handler)(int)))(int);
```

在上述signal()函数的原型之中，各参数的含义如下：

※ **signum**：指明了signal()函数所要处理的信号编号；

※ **handler**：描述了与信号关联的动作，它可取以下三种值：

① 一个参数类型为int返回值类型为void的函数地址：该函数必须在signal()函数被调用之前申明，handler为该函数的名字。当接收到一个信号编号为signum的信号时，进程就执行handler 所指定的函数。

第三章：服务器与客户进程中的并发机制

② SIGIGN：这个符号表示忽略该信号，执行了相应的signal()调用后，进程会忽略信号编号为signum的信号。

③ SIGDFL：这个符号表示恢复系统对信号的默认处理。

由上述signal()函数的原型可知，signal()函数有两个参数，第一个是参数类型为int，第二个是指向参数类型为int返回值类型为void的函数指针，signal()函数的返回值类型是一个函数指针，同样指向一个参数类型为int返回值类型为void的函数。signal()函数会依参数signum 指定的信号编号来设置该信号的处理函数。当指定的信号到达时就会跳转到参数handler所指定的函数执行。

第三章：服务器与客户进程中的并发机制

由以上描述可知，signal()函数的第二个参数类型是一个函数指针，而且signal()函数的返回值类型也是一个函数指针。这里，所谓的函数指针就是指一个指向函数的指针变量，即：函数指针本身首先应该是一个指针变量，只不过该指针变量指向的是一个函数，其实亦与用指针变量指向整型变量、字符型、数组相类似，只不过这里是用指针变量指向函数罢了。

如前所述，C在编译时，每一个函数都有一个入口地址，该入口地址就是函数指针所指向的地址。有了指向函数的指针变量之后，即可用该指针变量调用函数，就如同用指针变量可引用其他类型变量一样，这在概念上都是一致的。函数指针有两个用途：调用函数和做函数的参数。函数指针的声明方法如下：

第三章：服务器与客户进程中的并发机制

数据类型标志符 (指针变量名) (形参列表);

其中，“函数类型”说明了函数的返回类型，由于“( )”的优先级高于“*”，所以指针变量名外的括号必不可少；而后面的“形参列表”则表示指针变量指向的函数所带的参数列表。例如：

```
int func(int x);  //声明一个函数
int (*f) (int x);  //声明一个函数指针
f=func;        /*将func()函数的首地址赋给指针f，赋值时函数func()
    不带括号，也不带参数，由于func代表函数的首地址，因此
    经过赋值以后，指针f就指向函数func(x)的代码的首地址*/
```

第三章：服务器与客户进程中的并发机制

基于以上关于函数指针的定义，signal()函数的原型可理解为是由如下两个步骤所组成的：

步骤1：typedef void(*sig_t) (int);　/*首先，定义一个参数类型为int返回值类型为void的函数指针sig_t */

步骤2：sig_t signal(int signum,sig_t handler);　/*然后，定义一个返回值类型为函数指针sig_t的函数signal()，该函数有两个参数，一个参数类型为int，另一个参数类型为函数指针sig_t */

第三章：服务器与客户进程中的并发机制

由以上步骤2可知，handler为一个类型为sig_t的函数指针，因此再由步骤1可知，handler所指向的那个函数只能有一个int类型的参数；另外，由以上步骤2亦可知，signal()函数的返回值也为一个类型为sig_t的函数指针，因此再由步骤1可知，signal()函数返回的函数指针所指向的那个函数也只能有一个int类型的参数。

基于以上描述，下面给出一个简单的例子来进一步说明signal()函数的用法：

第三章：服务器与客户进程中的并发机制

```
#include<unistd.h>
#include<signal.h>
void handler() {
    printf("hello\n");
}
int main() {
    int i;
    signal(SIGALRM, handler);          /*调用 signal()函数获取 SIGALRM
                                        信号，并交由 handler 所指向的函
                                        数进行处理*/
    alarm(5);          /*调用 alarm()函数设置超时时钟为 5 秒，若时钟超时
                        则内核将给进程发送 SIGALRM 信号*/
```

第三章：服务器与客户进程中的并发机制

```
        for(i=1;i<7;i++){
            printf("sleep %d ...\n", i);
            sleep(1);        //调用 sleep()函数休眠 1 秒
        }
        return 0;
}
```

执行结果如下：

```
sleep 1 ...
sleep 2 ...
sleep 3 ...
sleep 4 ...
sleep 5 ...
hello
sleep 6 ...
```

第三章：服务器与客户进程中的并发机制

3.2.6多进程例程剖析

为了进一步说明上述各进程函数的具体用法，下面给出一个简单的多进程的例程：

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
int main(){
    pid_t child;
    int i;
    child=fork();
    if(child<0){
        printf("创建进程失败!");
        exit(-1);
    }
```

第三章：服务器与客户进程中的并发机制

```
else if(child==0){
        printf("这是子进程， 进程号是:%d\n",getpid());
        for(i=0;i<100;i++)
            printf("这是子进程第%d 此打印！ \n",i+1);
        printf("子进程结束!");
    }
    else{
        printf("这是父进程， 进程号是:%d\n",getppid());
        printf("父进程等待子进程结束...");
        wait(&child);
        printf("父进程结束!");
    }
}
```

第三章：服务器与客户进程中的并发机制

3.3 UNIX/LINUX环境下基于多线程的并发机制

3.3.1 创建一个新线程

在UNIX/LINUX环境下，线程的创建是通过调用pthread_create()函数来实现的。当创建线程成功时，该函数返回0，若不为0则说明创建线程失败。若创建线程成功，则新创建的线程将运行由pthread_create()函数中第三个参数和第四个参数所确定的函数，而原来的线程则继续运行下一行代码。pthread_create()函数的原型如下：

第三章：服务器与客户进程中的并发机制

```
#include <pthread.h>      //提供线程函数原型和数据结构的定义
int pthread_create(
pthread_t *thread,
pthread_attr_t *attr,
void *(*start_routine) (void *),
void * arg);
```

在上述pthread_create()函数的原型之中，各参数的含义如下：

※ **thread**：所创建的线程的标识符；

※ **attr**：是pthread_attr_t结构体指针，所指向的结构中的元素分别用于标示线程的运行属性；

第三章：服务器与客户进程中的并发机制

※ **start_routine**：是一个参数类型为(void *)返回值类型也为(void *)的函数指针，用于指向线程的线程体函数，该线程体函数所执行的操作即为该线程所执行的操作；

※ **arg**：是用于传递给线程体函数的参数。

1. 用单变量向线程体函数传递参数

下面的代码片段演示了如何向一个线程传递一个简单的整数：

第三章：服务器与客户进程中的并发机制

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 3
void *PrintHello(void *threadargs) {
    int pid;
    pid = (int) threadargs;
    printf("Hello! I am thread #%d!\n", pid);
    pthread_exit(NULL);                    //调用 pthread_exit()终止该线程
}
```

第三章：服务器与客户进程中的并发机制

```
int main (int argc, char *argv[]) {
    pthread_t pids[NUM_THREADS];
    int *args[NUM_THREADS];
    int rc, i;
    for(i=0; i<NUM_THREADS; i++) {
        args[i] = (int *) malloc(sizeof(int));
        *args[i] = i;
        printf("Creating thread %d\n", i);
        rc = pthread_create(&threads[i], NULL, PrintHello, (void *) args[i]);
        ...
    }
    ...
}
```

第三章：服务器与客户进程中的并发机制

2. 用结构体变量向线程体函数传递参数

下面的代码片段演示了如何向一个线程传递一个简单的结构体：

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 3
struct thread_data{                          //定义一个结构体 thread_data
    int thread_id;
    int sum;
};
```

第三章：服务器与客户进程中的并发机制

```
void *PrintHello(void *threadarg) {
    struct thread_data *my_data;
    int pid, sum;
    char * hello_msg;
    my_data = (struct thread_data *) threadarg;
    pid = my_data->thread_id;
    sum = my_data->sum;
    printf("Hello! I am thread #%d! The sum is %d!\n", pid, sum);
    pthread_exit(NULL);                    //调用 pthread_exit()终止该线程
}
```

第三章：服务器与客户进程中的并发机制

```
int main (int argc, char *argv[]) {
    pthread_t pids[NUM_THREADS];
    struct thread_data thread_data_array[NUM_THREADS];
    int rc, I,sum;
    sum=0;
    for(i=0; i<NUM_THREADS; i++) {
        sum++;
        thread_data_array[i].thread_id = i;
        thread_data_array[i].sum = sum;
        printf("Creating thread %d\n", i);
        rc = pthread_create(&pids[i], NULL, PrintHello, (void *) &thread_
data_array[i]);
        ...
    }
    ...
}
```

第三章：服务器与客户进程中的并发机制

3.3.2 设置线程的运行属性

线程具有运行属性，用pthread_attr_t结构体表示，在对该结构体进行处理之前必须进行初始化，在使用后需要对其去除初始化，以释放该结构体所占用的资源。用于对pthread_attr_t结构体进行初始化的函数为pthread_attr_init()，对其去除初始化的函数为pthread_attr_destroy()。

其中pthread_attr_init()与pthread_attr_destroy()函数的原型如下：

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

第三章：服务器与客户进程中的并发机制

在上述pthread_attr_init()与pthread_attr_destroy()函数的原型之中，各参数的含义如下：

※ **attr**：是pthread_attr_t结构体指针，所指向的结构中的元素分别用于标示线程的运行属性，其中，pthread_attr_t结构体的定义如下：

```
typedef struct {
        int  detachstate;                  //线程的分离状态
        int  schedpolicy;                  //线程调度策略
        struct sched_param  schedparam;      //线程的调度参数
        int  inheritsched;                 //线程的继承性
        int          scope;                  //线程的作用域
        size_t    guardsize;                //线程堆栈保护区的大小
        int          stackaddr_set;                  //线程堆栈的地址集
        void * stackaddr;                  //线程堆栈的地址
        size_t  stacksize;                 //线程堆栈的大小
} pthread_attr_t;
```

第三章：服务器与客户进程中的并发机制

Linux中可通过以下函数来设置上述线程的运行属性：

1. 设置/获取线程的分离状态

线程的分离状态决定了一个线程以何种方式来终止自己。在默认情况下线程为非分离状态，此时，需要让某个原有的线程调用pthread_join()函数来等待创建的线程结束，只有当pthread_join()函数返回时，创建的线程才真正终止、并释放自己所占用的系统资源。而分离线状态的程则无需被其他线程等待，一旦自己运行结束，该线程也就自动终止，并立即释放自己所占用的系统资源。为此，若在创建线程时就知道无需关注其终止状态，则可通过设置pthread_attr_t结构中的detachstate属性来让线程以分离状态运行。

第三章：服务器与客户进程中的并发机制

设置线程的分离状态可通过调用pthread_attr_setdetachstate()函数来实现，而获取线程的分离状态可通过调用pthread_attr_getdetachstate()函数来实现，这两个函数若调用成功将返回0，失败则返回-1，其函数原型分别如下：

```
#include <pthread.h>
int pthread_attr_getdetachstate(const pthread_attr_t * attr, int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

在上述pthread_attr_setdetachstate()函数与pthread_attr_getdetachstate()函数的原型之中，各参数的含义如下：

第三章：服务器与客户进程中的并发机制

※ **attr**：是pthread_attr_t结构体指针，所指向的结构中的元素分别用于标示线程的运行属性。

※ **detachstate**：线程的分离状态属性。

在pthread_attr_setdetachstate()函数中，若将参数detachstate设置为PTHREAD_CREATE_DETACHED，则该线程将以分离状态运行；若将参数detachstate设置为PTHREAD_CREATE_JOINABLE，则该线程将以默认的非分离状态运行。

以下给出一个创建分离状态线程的例子：

第三章：服务器与客户进程中的并发机制

```
#include <pthread.h>
void *child_thread() {
    printf("this is the child thread!\n");
}
int main() {
    pthread_t pid;
    pthread_attr_t attr;                          //定义线程属性结构体变量 attr
    pthread_attr_init(&attr);                     //对 attr 结构体变量进行初始化
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
                /*设置 attr 结构体变量中的 detachstate 字段值为
                PTHREAD_CREATE_DETACHED（分离状态线程）*/
    pthread_create(&pid,&attr, child_thread,NULL);   /*创建运行属性为 attr
                                                       的新线程*/
    pthread_attr_destroy(&attr);                  /*对 attr 去除初始化，以释放该结
                                                    构体所占用的资源*/
    return 0;
}
```

第三章：服务器与客户进程中的并发机制

2.设置/获取线程的继承性

函数pthread_attr_setinheritsched()和函数pthread_attr_getinheritsched()分别用来设置和获取线程的继承性，这两个函数若调用成功将返回0，若调用失败则返回-1，其函数原型分别如下：

```
#include <pthread.h>
int pthread_attr_getinheritsched(const pthread_attr_t *attr, int *inheritsched);
int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched);
```

第三章：服务器与客户进程中的并发机制

在上述pthread_attr_setinheritsched()函数和pthread_attr_getinheritsched()函数的原型之中，各参数的含义如下：

※ **attr**：是pthread_attr_t结构体指针，所指向的结构中的元素分别用于标示线程的运行属性。

※ **inheritsched**：线程的继承性。

线程的继承性决定了线程是从创建自己的父线程中自动继承调度策略与参数还是使用在pthread_attr_t结构体中的schedpolicy和schedparam字段中显式地设置的调度策略与参数。

第三章：服务器与客户进程中的并发机制

若将pthread_attr_setinheritsched()函数中的参数inheritsched的值设置为PTHREAD_INHERIT_SCHED，则表示新线程将继承创建自己的父线程的调度策略和参数；

若设置为PTHREAD_EXPLICIT_SCHED则表示使用在schedpolicy和schedparam属性中显式设置的调度策略和参数。

第三章：服务器与客户进程中的并发机制

3.设置/获取线程的调度策略

函数pthread_attr_setschedpolicy()和函数pthread_attr_getschedpolicy()分别用来设置和得到线程的调度策略，函数若调用成功将返回0，若失败则返回-1；其函数原型分别如下：

```
#include <pthread.h>
int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

在上述两个函数的原型之中，各参数的含义如下：

第三章：服务器与客户进程中的并发机制

※ **attr**：是pthread_attr_t结构体指针，所指向的结构中的元素分别用于标示线程的运行属性。

※ **policy**：为线程的调度策略，主要包括先进先出（SCHED_FIFO）、轮循（SCHED_RR）或其它（SCHED_OTHER）等。其中，SCHED_FIFO策略允许一个线程运行直到有更高优先级的线程准备好，或者直到它自愿阻塞自己。而SCHED_RR策略则与SCHED_FIFO策略稍有不同：如果有一个SCHED_RR策略的线程执行了超过一个固定的时期（时间片间隔）没有阻塞，而有另外的SCHED_RR或SCHBD_FIFO策略的相同优先级的线程准备好时，运行的线程将被抢占以便准备好的线程可以执行。

第三章：服务器与客户进程中的并发机制

4.设置/获取线程的调度参数

函数pthread_attr_getschedparam()和函数pthread_attr_setschedparam()分别用来设置和得到线程的调度参数，函数若调用成功将返回0，若失败则返回-1；其函数原型分别如下：

```
#include <pthread.h>
int pthread_attr_getschedparam(const pthread_attr_t *attr,
struct sched_param *param);
int pthread_attr_setschedparam(pthread_attr_t *attr,
const struct sched_param *param);
```

在上述两个函数的原型之中，各参数的含义如下：

第三章：服务器与客户进程中的并发机制

※ **attr**：是pthread_attr_t结构体指针，所指向的结构中的元素分别用于标示线程的运行属性。

※ **param**：是sched_param结构体指针，所指向的结构中的元素sched_priority用于标示线程运行的优先权值。结构sched_param的定义如下：

```
struct sched_param{
        int sched_priority;                    //线程运行的优先权值
};
```

系统支持的最大和最小优线程先权值可以通过调用sched_get_priority_max()函数和sched_get_priority_min()函数来分别得到，其中，大的优先权值对应高的优先权。

第三章：服务器与客户进程中的并发机制

sched_get_priority_max()函数和sched_get_priority_min()函数的原型分别如下：

```
#include <pthread.h>
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

上述两个函数若调用失败将返回-1；若调用成功则返回0、同时将得到的系统支持的最大/最小优线程先权值保存在policy变量之中。

第三章：服务器与客户进程中的并发机制

5.设置/获取线程的作用域

函数pthread_attr_setscope()和函数pthread_attr_getscope()分别用来设置和得到线程的作用域。这两个函数若调用失败将返回-1，若调用成功则返回0，其函数原型如下：

```
#include <pthread.h>
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(const pthread_attr_t *attr, int *scope);
```

在上述两个函数的原型之中，各参数的含义如下：

※ **attr**：是pthread_attr_t结构体指针，所指向的结构中的元素分别用于标示线程的运行属性。

第三章：服务器与客户进程中的并发机制

※ **scope**：是作用域（在pthread_attr_setscope()函数中）或指向作用域的指针（在pthread_attr_getscope()函数中）。

其中，作用域用于控制线程是否在进程内或系统级上竞争资源，可能的值有PTHREAD_SCOPE_PROCESS（进程内竞争资源），PTHREAD_SCOPE_SYSTEM （系统级上竞争资源）。

例如：假设有两个进程A和B，其中，A进程包含4个线程，而B进程仅仅只有1个主线程，如果假设作用域设置为PTHREAD_SCOPE_ SYSTEM，则A进程中的4个线程将和B进程中的那1个线程一起竞争CPU资源，但若作用域设置为PTHREAD_SCOPE_PROCESS，则A进程中的4个线程只能竞争所属进程A的CPU时间。不过，目前Linux只支持PTHREAD_SCOPE_ SYSTEM方式。

第三章：服务器与客户进程中的并发机制

6.设置/获取线程堆栈保护区的大小

函数pthread_attr_getguardsize()和函数pthread_attr_setguardsize()分别用来设置和得到线程的堆栈保护区（警戒堆栈）的大小，这两个函数若调用失败将返回-1，若调用成功则返回0，其函数原型如下：

```
#include <pthread.h>
int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
size_t *restrict guardsize);
int pthread_attr_setguardsize(pthread_attr_t *attr, size_t
*guardsize);
```

在上述两个函数的原型之中，各参数的含义如下：

第三章：服务器与客户进程中的并发机制

※ **attr**：是pthread_attr_t结构体指针，所指向的结构中的元素分别用于标示线程的运行属性。

※ **guardsize**：控制着线程堆栈末尾之后以避免堆栈溢出的堆栈保护区（扩展内存）的大小。

其中，堆栈保护区被用来在堆栈指针越界的情况下提供保护。如果一个线程具有堆栈保护的特性，那么系统在创建线程堆栈时会在堆栈的末尾多分配一块内存，以用来防止指针访问堆栈时溢出堆栈的边界。如果一个应用程序访问堆栈时溢出到堆栈保护区时将会引发一个错误（此时，当前线程将收到一个SIGSEGV信号）。

第三章：服务器与客户进程中的并发机制

提供堆栈保护区属性设置函数的主要有以下两个原因：（1）首先，堆栈保护会引起系统资源浪费，因此，如果一个应用程序创建了大量线程，而且确保这些线程均不会越界访问堆栈时，则应用程序可通过调用堆栈保护区属性设置函数来取消堆栈保护区以节省系统资源。（2）当线程在堆栈中存放大的数据结构时，有可能需要一个大的堆栈保护区，此时则可通过调用堆栈保护区属性设置函数来增加堆栈保护区的大小。

如果线程属性对象的guardsize参数值为0，则创建线程时将不会创建堆栈保护区，若线程属性对象的guardsize参数值大于0，则使用该线程属性对象创建的线程将起码有一个guardsize大小的堆栈保护区。

第三章：服务器与客户进程中的并发机制

7. 设置/获取线程堆栈的地址

函数pthread_attr_setstackaddr()和函数pthread_attr_getstackaddr()分别用来设置和得到线程堆栈的地址，这两个函数若调用失败将返回-1，若调用成功则返回0，其函数原型如下：

```
#include <pthread.h>
int pthread_attr_getstackaddr(const pthread_attr_t *attr, void **stackaddr);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
```

在上述两个函数的原型之中，各参数的含义如下：

第三章：服务器与客户进程中的并发机制

※ **attr**：是pthread_attr_t结构体指针，所指向的结构中的元素分别用于标示线程的运行属性。

※ **stackaddr**：线程的堆栈地址。

注：设置堆栈地址将降低可移植性，建议最好不要自己设置堆栈地址。

8. 设置/获取线程堆栈的大小

函数pthread_attr_setstacksize()和函数pthread_attr_getstacksize()分别用来设置和得到线程堆栈的大小，这两个函数若调用失败将返回-1，若调用成功则返回0，其函数原型如下：

第三章：服务器与客户进程中的并发机制

```
#include <pthread.h>
int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
size_t *restrict stacksize);
int pthread_attr_setstacksize(pthread_attr_t *attr , size_t
*stacksize);
```

在上述两个函数的原型之中，各参数的含义如下：

※ **attr**：是pthread_attr_t结构体指针，所指向的结构中的元素分别用于标示线程的运行属性。

※ **stacksize**：线程堆栈的大小。

第三章：服务器与客户进程中的并发机制

3.3.3 终止一个线程

在Linux系统中，一个线程既可以通过自身调用pthread_exit()函数来实现显式地终止，也可以通过在另一个线程中调用pthread_join()函数来实现隐式地终止。其中：

1. pthread_exit()函数：该函数在头文件pthread.h中声明，线程在调用该函数后将自行终止并释放其所占资源。pthread_exit()函数的原型如下：

```
#include <pthread.h>
int pthread_exit(void * value_ptr);
```

在上述pthread_exit()函数的原型之中，各参数的含义如下：

第三章：服务器与客户进程中的并发机制

※ **value_ptr**：线程返回值指针，该返回值将被传给另一个线程，另一个线程则可通过调用pthread_join()函数来获得该值。

2. pthread_join()函数：该函数在头文件pthread.h中声明，其作用是用于等待一个指定的线程结束，该函数的原型如下：

```
#include <pthread.h>
int pthread_join(pthread_t thread,void **value_ptr);
```

在上述pthread_join()函数的原型之中，各参数的含义如下：

※ **thread**：等待终止的线程的线程标识符

第三章：服务器与客户进程中的并发机制

※ **value_ptr**：如果value_ptr不为NULL，那么线程thread的返回值存储在该指针指向的位置。该返回值可以是由pthread_exit()给出的值，或者该线程被取消而返回PTHREAD_CANCELED。pthread_join()函数是一个线程阻塞函数，调用它的函数将一直等到被等待的线程结束为止。调用pthread_join()函数的线程将被挂起，直到参数thread所代表的线程终止时为止。

当一个非分离的线程终止后，该线程的内存资源（线程描述符和栈）并不会被释放，直到有线程对它调用了pthread_join()时才会被释放。因此，必须对每个创建的非分离的线程调用一次pthread_join()以避免内存泄漏。另外，至多只能有一个线程调用pthread_join()等待给定的线程终止，如果已有一个线程调用了pthread_join()以等待某个线程终止，那么其它再调用pthread_join()以等待同一线程终止的线程将返回一个错误。

第三章：服务器与客户进程中的并发机制

3.3.4 获得一个线程的线程标识

函数pthread_self()可以用来使得调用线程获取自己的线程ID，其函数原型如下：

```
#include <pthread.h>

pthread_t pthread_self();
```

3.3.5多线程例程剖析

为了进一步说明上述各线程函数的具体用法，下面给出一个简单的多线程的例程：

第三章：服务器与客户进程中的并发机制

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void thread() {                                          //线程体函数
    int i;
    for(i=0;i<3;i++)
        printf("This is a pthread.\n");
}
int main() {
```

第三章：服务器与客户进程中的并发机制

```
pthread_t pid;                                  //声明线程标识符变量 pid
pthread_attr_t attr;                            //声明线程运行属性变量 attr
int i, ret;
pthread_attr_init(&attr);                       //初始化线程运行属性变量 attr
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED);
ret=pthread_create(&pid, &attr, (void *) thread, NULL);     /*创建一个新
                                                线程执行线程体函数 thread()，该线
                                                程的标识符为 pid*/
if(ret!=0){                     //若创建新线程失败，则输出出错提示并退出程序
        printf ("Create pthread error!\n");
        exit (1);
}
```

第三章：服务器与客户进程中的并发机制

```
    pthread_attr_destroy(&attr);        //去初始化线程运行属性变量 attr
    for(i=0;i<3;i++)                    //在主线程中输出 3 次主线程提示信息
        printf("This is the main process.\n");
    pthread_join(pid,NULL);             /*在主线程中调用 pthread_join()函数等
                                        待新线程结束并回收新线程所占资源*/
    return 0;
}
```

第三章：服务器与客户进程中的并发机制

3.4 Windows环境下基于多进程的并发机制

3.4.1创建一个新进程

在Windows环境下，一个现有的进程可以调用CreateProcess()函数来创建一个新的进程。如果函数执行成功，返回非零值，如果函数执行失败，返回零，可以使用GetLastError()函数获得错误的附加信息。CreateProcess()函数的原型如下：

```
BOOL CreateProcess(
    LPCTSTR lpApplicationName,
    LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
```

第三章：服务器与客户进程中的并发机制

BOOL bInheritHandles,

DWORD dwCreationFlags,

LPVOID lpEnvironment,

LPCTSTR lpCurrentDirectory,

LPSTARTUPINFO lpStartupInfo,

LPPROCESS_INFORMATION lpProcessInformation);

在上述CreateProcess()函数的原型之中，各参数的含义如下：

※**lpApplicationName**：指向一个NULL结尾的、用来指定可执行文件的字符串。该参数可以被设为NULL，在这种情况下，可执行模块的名字必须处于 lpCommandLine参数最前面并由空格符与后面的命令行字符串分开。

第三章：服务器与客户进程中的并发机制

※**lpCommandLine**：指向一个以NULL结尾的字符串，该字符串指定要执行的命令行。该参数可以为空，那么函数将使用lpApplicationName参数指定的字符串当做要运行的程序的命令行。若lpApplicationName和lpCommandLine参数都不为空，则lpApplicationName参数指定将要被运行的文件，lpCommandLine参数指定将被运行的文件的命令行。新运行的进程可以使用GetCommandLine()函数获得整个命令行，C语言程序则可以使用argc和argv参数。

※**lpProcessAttributes**：指向一个SECURITY_ATTRIBUTES结构体，该结构体决定是否返回的句柄可以被子进程继承。如果lpProcessAttributes参数为空（NULL）那么句柄不能被继承。

第三章：服务器与客户进程中的并发机制

typedef struct _SECURITY_ATTRIBUTES {
DWORD nLength;　　//结构体的大小，可用SIZEOF取得
LPVOID lpSecurityDescriptor;　//安全描述符
BOOL bInheritHandle ;　//安全描述的对象能否被新创建的进程继承
} SECURITY_ATTRIBUTES，* PSECURITY_ATTRIBUTES;

※**lpThreadAttributes**：同lpProcessAttribute，不过该参数决定的是线程是否被继承，通常置为NULL。

※**bInheritHandles**：指示新进程是否从调用进程处继承了句柄。如果参数的值为真，则调用进程中的每一个可继承的打开句柄都将被子进程继承，且被继承的句柄与原进程拥有完全相同的值和访问权限。

第三章：服务器与客户进程中的并发机制

※dwCreationFlags：指定附加的用来控制优先类和进程的创建的标志，以下的创建标志可以以除下面列出的方式外的任何方式组合后指定：

（1）CREATE_DEFAULT_ERROR_MODE：新的进程不继承调用进程的错误模式。CreateProcess()函数赋予新进程当前的默认错误模式作为替代。应用程序可以调用SetErrorMode()函数设置当前的默认错误模式。对于CreateProcess()函数，默认的行为是为新进程继承调用者的错误模式。设置这个标志以改变默认的处理方式。

（2）CREATE_NEW_CONSOLE：新的进程将使用一个新的控制台，而不是继承父进程的控制台。该标志不能与DETACHED_PROCESS标志一起使用。

第三章：服务器与客户进程中的并发机制

（3）CREATE_NEW_PROCESS_GROUP：新创建的进程将是一个进程树的根进程。进程树中的全部进程都是根进程的子进程。新进程树的用户标识符与这个进程的标识符是相同的，由lpProcessInformation参数返回。进程树经常使用GenerateConsoleCtrlEvent()函数允许发送CTRL+C或CTRL+BREAK信号到一组控制台进程。

（4）CREATE_SEPARATE_WOW_VDM：如果被设置，新进程将会在一个私有的虚拟DOS机（VDM）中运行。另外，默认情况下所有的16位Windows应用程序都会在同一个共享的VDM中以线程的方式运行。单独运行一个16位程序的优点是一个应用程序的崩溃只会结束这一个VDM的运行；其他那些在不同VDM中运行的程序会继续正常的运行。

第三章：服务器与客户进程中的并发机制

（5）CREATE_SHARED_WOW_VDM：如果WIN.INI中的Windows段的DefaultSeparateVDM选项被设置为真，该标识使得CreateProcess()函数越过该选项并在共享的虚拟DOS机中运行新进程。

（6）CREATE_SUSPENDED：新进程的主线程会以暂停的状态被创建，直到调用ResumeThread()函数被调用时才运行。

（7）CREATE_UNICODE_ENVIRONMENT：如果被设置，由lpEnvironment参数指定的环境块使用Unicode字符，如果为空，环境块使用ANSI字符。

（8）DEBUG_PROCESS：如果这个标志被设置，调用进程将被当做一个调试程序，并且新进程会被当做被调试的进程。如果使用了该标志创建进程，则只有调用进程（调用CreateProcess函数的进程）可调用WaitForDebugEvent()函数。

第三章：服务器与客户进程中的并发机制

（9）DEBUG_ONLY_THIS_PROCESS：如果此标志没有被设置且调用进程正在被调试，新进程将成为调试调用进程的调试器的另一个调试对象。如果调用进程没有被调试，有关调试的行为就不会产生。

（10）DETACHED_PROCESS：对于控制台进程，新进程没有访问父进程控制台的权限。新进程可以通过AllocConsole()函数自己创建一个新的控制台。该标志不可以与CREATE_NEW_CONSOLE标志一起使用。

（11）CREATE_NO_WINDOW：系统不为新进程创建CUI（Command User Interface，命令行用户交互）窗口，使用该标志可以创建不含窗口的CUI程序。

第三章：服务器与客户进程中的并发机制

此外，dwCreationFlags参数还可用来控制新进程的优先类，优先类用来决定该进程的线程调度的优先级，其中，可以被指定的优先级类标志包括（如果下面的优先级类标志都没有被指定，那么默认的优先类是NORMAL_PRIORITY_CLASS，除非被创建的进程是IDLE_PRIORITY_CLASS。在该情况下子进程的默认优先类是IDLE_PRIORITY_ CLASS）：

（1）HIGH_PRIORITY_CLASS：指示这个进程将执行时间临界的任务，所以它必须被立即运行以保证正确。这个优先级的程序优先于正常或空闲优先级的程序。一个例子是Windows任务列表，为保证当用户调用时可立刻响应，放弃了对系统负荷的考虑。确保在使用高优先级时应该足够谨慎，因为一个高优先级的CPU关联应用程序可占用几乎全部的CPU可用时间。

第三章：服务器与客户进程中的并发机制

（2）IDLE_PRIORITY_CLASS：指示这个进程的线程只有在系统空闲时才会运行并且可以被任何高优先级的任务打断。例如屏幕保护程序。空闲优先级会被子进程继承。

（3）NORMAL_PRIORITY_CLASS：指示这个进程没有特殊的任务调度要求。

（4）REALTIME_PRIORITY_CLASS：指示这个进程拥有可用的最高优先级。一个拥有实时优先级的进程的线程可以打断所有其他进程线程的执行，包括正在执行重要任务的系统进程。例如，一个执行时间稍长一点的实时进程可能导致磁盘缓存不足或鼠标反映迟钝。

第三章：服务器与客户进程中的并发机制

※lpEnvironment：指向一个新进程的环境块。如果此参数为空，新进程使用调用进程的环境。一个环境块存在于一个由以NULL结尾的字符串组成的块中，这个块也是以NULL结尾的。每个字符串都是name=value的形式。因为相等标志被当做分隔符，所以它不能被环境变量当做变量名。与其使用应用程序提供的环境块，不如直接把这个参数设为空（NULL）。

※lpCurrentDirectory：指向一个以NULL结尾的字符串，该字符串用来指定子进程的工作路径。该字符串必须是一个包含驱动器名的绝对路径。若该参数为空，则新进程将使用与调用进程相同的驱动器和目录。

第三章：服务器与客户进程中的并发机制

※**lpStartupInfo**：指向一个用于决定新进程的主窗体如何显示的STARTUPINFO结构体，该结构体用于指定新进程的主窗口特性。

```
typedef struct _STARTUPINFO {
    DWORD cb;       /*包含 STARTUPINFO 结构中的字节数，如果 Microsoft
                      将来扩展该结构，它可用作版本控制手段，应用程序
                      必须将 cb 初始化为 sizeof(STARTUPINFO)*/
    LPTSTR lpReserved;      //保留，必须初始化为 NULL
    LPTSTR  lpDesktop;     /*用于标识启动应用程序所在的桌面的名字。如
                             果该桌面存在，新进程便与指定的桌面相关
                             联，如果桌面不存在，便创建一个带有默认属
                             性的桌面，并使用为新进程指定的名字。如果
                             lpDesktop 是 NULL，那么该进程将与当前桌
                             面相关联*/
```

第三章：服务器与客户进程中的并发机制

```
LPTSTR lpTitle;      /*用于设定控制台窗口的名称。如果 lpTitle 是
                       NULL，则可执行文件的名字将用作窗口名*/
DWORD dwX;           /*用于设定应用程序窗口在屏幕上应该放置的位
                       置的 x 和 y 坐标（以像素为单位），只有当子
                       进程用 CW_USEDEFAULT 作为 CreateWindow
                       的 x 参数来创建它的第一个重叠窗口时，才使
                       用这两个坐标。若是创建控制台窗口的应用程
                       序，这些成员用于指明控制台窗口的左上角*/
DWORD dwY;
```

第三章：服务器与客户进程中的并发机制

```
DWORD  dwXSize;     /*用于设定应用程序窗口的宽度和长度（以像素
                    为单位），只有当子进程将 CW_USEDEFAULT
                    用作 CreateWindow()函数的 nWidth 参数来创建
                    它的第一个重叠窗口时，才使用这些值*/
DWORD dwYSize;
DWORD  dwXCountChars;      /*用于设定子应用程序的控制台窗口的
                           宽度和高度（以字符为单位）*/
DWORD dwYCountChars;
DWORD  dwFillAttribute;    /*用于设定子应用程序的控制台窗口使
                           用的文本和背景颜色*/
DWORD dwFlags;             //创建窗口标志。
```

第三章：服务器与客户进程中的并发机制

```
    WORD wShowWindow;     /*如果子应用程序初次调用的 ShowWindow()
                            函数将 SW_ SHOWDEFAULT 作为
                            nCmdShow 参数传递时，用于设定该应用程
                            序的第一个重叠窗口应该如何出现，本成员
                            可以是通常用于 ShowWindow()函数的任何
                            一个 SW_*标识符*/
    WORD cbReserved2;     //保留，必须被初始化为 0
    LPBYTE lpReserved2;   //保留，必须被初始化为 NULL
    HANDLE hStdInput;     /*用于设定供控制台输入和输出用的缓存的句
                            柄。按照默认设置，hStdInput 用于标识键盘
                            缓存，hStdOutput 和 hStdError 用于标识控制
                            台窗口的缓存*/
    HANDLE hStdOutput;
    HANDLE hStdError;
} STARTUPINFO, *LPSTARTUPINFO;
```

第三章：服务器与客户进程中的并发机制

当Windows创建新进程时，它将使用该结构的有关成员。大多数应用程序将要求生成的应用程序仅仅使用默认值。至少应该将该结构中的所有成员初始化为零，然后将cb 成员设置为该结构的大小：

```
STARTUPINFO si = { sizeof(si) };
CreateProcess(...,&si,...);
```

※**lpProcessInformation**：指向一个用来接收新进程的识别信息的PROCESS_INFORMATION结构体。

第三章：服务器与客户进程中的并发机制

```
typedef struct _PROCESS_INFORMATION{
    HANDLE hProcess;              //返回新进程的句柄。
    HANDLE hThread;               //返回主线程的句柄。
    DWORD  dwProcessId;           /*返回一个全局进程标识符。该标识
                                    符用于标识一个进程。从进程被创
                                    建到终止，该值始终有效*/
    DWORD  dwThreadId;            /*返回一个全局线程标识符。该标识
                                    符用于标识一个线程。从线程被创
                                    建到终止，该值始终有效*/
}PROCESS_INFORMATION;
```

第三章：服务器与客户进程中的并发机制

3.4.2打开一个进程

OpenProcess()函数用来打开一个已存在的进程对象，并返回进程的句柄，该函数若调用成功，返回值为指定进程的句柄，若失败，则返回值为NULL，并可调用GetLastError()来获得错误代码。OpenProcess()函数的原型如下：

```
HANDLE OpenProcess(
   DWORD dwDesiredAccess,    //渴望得到的访问权限（标志）
   BOOL bInheritHandle,      //是否继承句柄
   DWORD dwProcessId         //进程标示符
);
```

第三章：服务器与客户进程中的并发机制

3.4.3终止/关闭一个进程

1. ExitProcess()函数用于进程自己**强制终止**（非正常终止）自己，其函数原型如下：

```
VOID ExitProcess(
UINT fuExitCode          //退出代码
);
```

2. TerminateProcess()函数用于进程**强制终止**（非正常终止）其它进程，其函数原型如下：

第三章：服务器与客户进程中的并发机制

```
BOOL TerminateProcess(
    HANDLE hProcess,              //希望终止的进程的句柄
    UINT fuExitCode               //退出代码
);
```

3. CloseHandle()函数用于关闭一个进程/线程的句柄。在CreateProcess()函数或CreateThread()函数调用成功之后，会返回一个句柄（handle），且内核对象的引用计数将增加1，而在调用CloseHandle()函数之后该引用计数会减少1，当减为0时，系统将删除该内核对象。CloseHandle()函数的原型如下：

```
BOOL CloseHandle(HANDLE hObject);
```

在上述CloseHandle()函数的原型之中，各参数的含义如下：

※**hObject**：是指将要关闭的线程的句柄。

第三章：服务器与客户进程中的并发机制

3.4.4获得进程的可执行文件或DLL对应的句柄

GetModuleHandle()函数用于获取一个指定的应用程序或动态链接库的模块句柄，若调用成功则返回模块句柄，失败则返回0，GetModuleHandle()函数的原型如下：

```
HMODULE GetModuleHandle(
  PCTSTR pszModule  //进程的可执行文件名或DLL对应的句柄
);
```

注：当参数传NULL时获取的是进程的地址空间中可执行文件的基地址。

第三章：服务器与客户进程中的并发机制

3.4.5获取与指定窗口关联在一起的一个进程和线程标识符

GetWindowThreadProcessId()函数用于获取一个指定窗口的创建者（线程或进程）的标志符，若调用成功则返回该指定窗口的创建者（线程或进程）的标志符，失败则返回0，

GetWindowThreadProcessId()函数的原型如下：

```
HANDLE GetWindowThreadProcessId(
    HWND hWnd,                  //窗口句柄
    LPDWORD lpdwProcessId       //与该窗口相关的进程ID
);
```

第三章：服务器与客户进程中的并发机制

3.4.6获取进程的运行时间

GetProcessTimes()函数用于获取当取进程的经过时间，若调用成功则返回一个大于0的值，失败则返回0，GetProcessTimes()函数的原型如下：

```
Bool GetProcessTimes(
    HANDLE hProcess,                    //进程句柄
     PFILETIME pftCreationTime,         //创建时间
    PFILETIME pftExitTime,              //退出时间
    PFILETIME pftKernelTime,            //内核时间
    PFILETIME pftUserTime               //用户时间
);
```

注：返回的时间适用于某个进程中的所有线程（甚至已经终止运行的线程）。

第三章：服务器与客户进程中的并发机制

3.4.7获取当前进程ID

GetCurrentProcessId()函数用于获取当前进程唯一的标识符，若调用成功则返回当前进程唯一的标识符，失败则返回0，

GetCurrentProcessId()函数的原型如下：

DWORD GetCurrentProcessId();

3.4.8等待子进程/子线程的结束

1. WaitForSingleObject()函数：主要用来检测hHandle事件的信号状态，在某一进程/线程中调用该函数时，该进程/线程将被暂时挂起，如果在挂起的dwMilliseconds毫秒内，进程/线程所等待的对象变为有信号状态，则该函数立即返回；如果超时时间已经到达dwMilliseconds毫秒，但hHandle所指向的对象还没有变成有信号状态，该函数照样返回。

第三章：服务器与客户进程中的并发机制

参数dwMilliseconds有两个具有特殊意义的值：0和INFINITE。若为0，则该函数立即返回；若为INFINITE，则进程/线程将一直被挂起，直到hHandle所指向的对象变为有信号状态时为止。

WaitForSingleObject()函数的原型如下：

DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);

在上述WaitForSingleObject()函数的原型之中，各参数含义如下：

※**hHandle**：对象句柄，可以指定一系列的对象，如：Event、Job、Memory resource notification、Mutex、Process、Semaphore、Thread、Waitable timer等。

第三章：服务器与客户进程中的并发机制

※**dwMilliseconds**：定时时间间隔，单位为毫秒。若指定一个非零值，则WaitForSingleObject()函数处于等待状态，直到hHandle标记的对象被触发或时间到了。若dwMilliseconds为0，对象没有被触发信号，WaitForSingleObject()函数不会进入一个等待状态，而会立即返回。若dwMilliseconds为INFINITE，则只有当对象被触发信号后，WaitForSingleObject()函数才会返回。WaitForSingleObject()函数若执行成功，则其返回值指示出了引发函数返回的事件。WaitForSingleObject()函数的返回值可能为以下值：

第三章：服务器与客户进程中的并发机制

※WAIT_ABANDONED 0x00000080：当hHandle为mutex时，若拥有mutex的线程在结束时没有释放核心对象会引发此返回值。

※WAIT_OBJECT_0 0x00000000：核心对象已被激活。

※WAIT_TIMEOUT 0x00000102：等待超时。

※WAIT_FAILED 0xFFFFFFFF：出现错误，可通过GetLastError()函数得到错误代码。

2. **WaitForMultipleObjects()函数**：该函数的功能与WaitForSingleObject()函数类似，主要区别在于WaitForMultipleObjects()函数允许调用进程/线程同时查看若干个内核对象的已通知状态。WaitForMultipleObjects()函数的原型如下：

第三章：服务器与客户进程中的并发机制

DWORD WaitForMultipleObjects(DWORD nCount, const HANDLE* lpHandles, BOOL bWaitAll, DWORD dwMilliseconds);

在上述WaitForMultipleObjects()函数的原型之中，各参数的含义如下：

※**nCount**：句柄的数量 最大值为MAXIMUM_WAIT_OBJECTS（64）。

※**lpHandles**：句柄数组的指针。HANDLE 类型可以为Event，Mutex，Process，Thread，Semaphore数组

※**bWaitAll**：等待的类型，如果为TRUE ，则进程/线程将等待所有信号量均有效后才会再往下执行，如果为FALSE，则进程当其中的一个信号量有效时就会向下执行。

※**dwMilliseconds**：含义同WaitForSingleObject()函数中的该参数。

第三章：服务器与客户进程中的并发机制

3.4.9多进程例程剖析

为了进一步说明上述各进程函数的具体用法，下面给出一个简单的多进程的例程，在该例程中，每次从文件中读取一条命令行命令，然后启动一个新的进程来执行该命令行命令：

```
#include <windows.h>
# include <process.h>
#include <stdlib.h>
#include <stdio.h>
using namespace std;
#define MAX_LINE_LEN 80
```

第三章：服务器与客户进程中的并发机制

```
int main(int argc,char* argv) {
    FILE* fid;
    char cmdLine[MAX_LINE_LEN];      //每个新进程执行一条命令行命令
    LPSECURITY_ATTRIBUTES processA=NULL;
    LPSECURITY_ATTRIBUTES threadA=NULL;
    BOOL shareRights=TRUE;
    DWORD  creationMask=CREATE_NEW_CONSOLE;     /*每个进程将使
                                                  用一个新的控
                                                  制台*/
    LPVOID enviroment=NULL;
    LPSTR curDir=NULL;
    STARTUPINFO startInfo;
    PROCESS_INFORMATION procInfo;
```

第三章：服务器与客户进程中的并发机制

```
if(argc!=2) {
    printf("Input error, Usage:launch!\n");          //命令行参数输入出错
    exit(0);
}
fid=fopen(argv[1],"r");                    //打开包含有一组命令的文件 launch
/*依次读取文件 launch 中包含的每一条命令*/
while(fgets(cmdLine,MAX_LINE_LEN,fid)!=NULL) {
    if(cmdLine[strlen(cmdLine)-1]= ='\n')       //若最后一个字符为换行符
        cmdLine[strlen(cmdLine)-1]='\0';        //用行结束符替换该换行符
    ZeroMemory(&startInfo,sizeof(startInfo)); /*调用  ZeroMemory()函数
```

第三章：服务器与客户进程中的并发机制

```
                                        初始化 startInfo 变量为 0，ZeroMemory()
                                        函数的功能与 memset()函数类似*/
        startInfo.cb=sizeof(startInfo);          /*应用程序必须将 cb 初始化为
                                                  sizeof(STARTUPINFO) */
        ZeroMemory(&procInfo,sizeof(procInfo));
        /*针对读取的每一条命令，生成一个新的进程来执行该条命令*/
        if(!CreateProcess(NULL, cmdLine, processA, threadA, shareRights,
creationMask, enviroment, curDir, &startInfo, &procInfo)) {
            printf("CreatProcess failed on error %d\n",GetLastError);
            ExitProcess(0);
        }
```

第三章：服务器与客户进程中的并发机制

```
            WaitForSingleObject(procInfo.hProcess, INFINITE);
            CloseHandle(procInfo.hProcess);
            CloseHandle(procInfo.hThread);
        }
        return 0;
    }
```

注：Windows 中的进程之间的父子关系很弱，没有僵尸进程的概念。

为了更进一步说明 Windows 环境下基于多进程的服务器并发机制，下面再给出一个例子，将利用 CreateProcess 开启一个新进程，启动 IE 浏览器，打开百度的主页，5s 后再将其关闭。

第三章：服务器与客户进程中的并发机制

```
#include <Windows.h>
#include <tchar.h>
#include <iostream>
using namespace std;
#define IE L"C:\\Program Files\\Internet Explorer\\iexplore.exe"  //IE 浏览器
#define CMD L"open http://www.baidu.com/"                         //百度主页

int _tmain(int argc, _TCHAR* argv[]){
    STARTUPINFO startup_info;
```

第三章：服务器与客户进程中的并发机制

```
ZeroMemory(&startup_info,sizeof(startup_info));
startup_info.dwFlags=STARTF_USESHOWWINDOW;
startup_info.wShowWindow=SW_HIDE;
startup_info.cb=sizeof(startup_info);
PROCESS_INFORMATION process_info;
ZeroMemory(&process_info,sizeof(process_info));
/*创建一个新进程，启动 IE 浏览器，打开百度的主页*/
if(!CreateProcess(IE,CMD,NULL,NULL,FALSE,CREATE_NO_WINDOW,
NULL, NULL,&startup_info,&process_info)) {
    printf("CreatProcess failed on error %d\n",GetLastError);
    return 0;
}
```

第三章：服务器与客户进程中的并发机制

```
        Sleep(5000);
        WaitForSingleObject(process_info.hProcess, INFINITE);
        CloseHandle(process_info.hProcess);
        CloseHandle(process_info.hThread);
        return 0;
    }
```

- **3.5 Windows 环境下基于多线程的并发机制**
- **3.5.1 在本地进程中创建一个新线程**

在 Windows 环境下，线程创建是通过调用 CreateThread()函数实现的。当创建线程成功时，该函数返回线程句柄，失败则返回 False。CreateThread()函数的原型如下：

第三章：服务器与客户进程中的并发机制

```
#include <windows.h>
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
);
```

在上述 CreateThread()函数的原型之中，各参数的含义如下：

第三章：服务器与客户进程中的并发机制

※**lpThreadAttributes**：指向SECURITY_ATTRIBUTES型态的结构的指针。

※**dwStackSize**：设置初始栈的大小，以字节为单位，如果为0，那么默认将使用与调用该函数的线程相同的栈空间大小。

※**lpStartAddress**：指向线程函数的指针，形式：@函数名，函数名称没有限制，但是必须以下列形式声明：

DWORD WINAPI ThreadProc (LPVOID lpParam) ;

格式不正确将无法调用成功。

也可以直接调用void类型，但lpStartAddress要按照以下形式来通过LPTHREAD_START_ROUTINE转换：

(LPTHREAD_START_ROUTINE) MyVoid

第三章：服务器与客户进程中的并发机制

然后，再声明MyVoid()函数为：

```
void MyVoid(){
    ……
    return;
}
```

※**lpParameter**：向线程函数传递的参数，是一个指向结构的指针，不需传递参数时，为NULL。

※**dwCreationFlags**：线程标志，可取值如下：

（1）CREATE_SUSPENDED(0x00000004)：创建一个挂起的线程，表示线程创建后暂停运行，直到调用ResumeThread()函数后才可调度执行。

（2）0：表示线程创建之后立即就可以进行调度。

※**lpThreadId**：保存新线程的id。

第三章：服务器与客户进程中的并发机制

3.5.2 在远程进程中创建一个新线程

在Windows环境下，CreateRemoteThread()函数提供了一个在远程进程中执行代码的方法，就像代码长出翅膀飞到别处运行。当创建线程成功时，该函数返回新线程句柄，失败则返回NULL，并且可通过调用GetLastError()函数来获得错误代码值。

CreateRemoteThread()函数的原型如下：

第三章：服务器与客户进程中的并发机制

```
#include <windows.h>
HANDLE CreateRemoteThread(
    HANDLE hProcess,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
    );
```

在上述 CreateRemoteThread()函数的原型之中，各参数的含义如下：

第三章：服务器与客户进程中的并发机制

※**hProcess**：线程所属进程的进程句柄。

※**lpThreadAttributes**：一个指向 SECURITY_ATTRIBUTES 结构的指针，该结指定了线程的安全属性。

※**dwStackSize**：线程初始大小，以字节为单位，如果该值设为 0，那么使用系统默认大小。

※**lpStartAddress**：在远程进程的地址空间中该线程的线程函数的起始地址。

※**lpParameter**：传给线程函数的参数。

※**dwCreationFlags**：线程的创建标志，其取值如下表所示：

第三章：服务器与客户进程中的并发机制

表 3.1 线程的创建标志取值及含义

值	含义
0	线程创建后立即运行
CREATE_SUSPENDED 0x0000004	线程创建后先将线程挂起，直到 ResumeThread 被调用。
STACK_SIZE_PARAM_IS_A_RESERVATION 0x00010000	dwStackSize 参数指定为线程栈预订大小，如果 STACK_SIZE_PARAM_IS_A_RESERVATION 没有被指定，dwStackSize 参数指定为线程栈分配大小。

※**lpThreadId**：指向所创建线程句柄的指针，若创建失败则该参数为 NULL。

第三章：服务器与客户进程中的并发机制

3.5.3 获取/设置线程的优先级

Windows是抢先式执行任务的操作系统，无论进程还是线程都具有优先级的选择执行方式，这样就可以让用户更加方便处理多任务。

例如：当你一边听着音乐，一边上网时，这时就可以把音乐的任务执行级别高一点，这样不让音乐听起来断断续续。在Windows环境下，可通过调用GetThreadPriority()函数来获取线程的优先级，或调用SetThreadPriority()函数来设置线程的优先级。

第三章：服务器与客户进程中的并发机制

1. SetThreadPriority()函数的原型如下：

BOOL SetThreadPriority(HANDLE hThread, int priority);

在上述SetThreadPriority()函数的原型之中，各参数含义如下：

※**hThread**：是指要设置优先级的线程的句柄。

※**priority**：是指要设置的优先级。

在调用SetThreadPriority()函数时，如果发生错误，则返回值为0；否则，返回非0值。

第三章：服务器与客户进程中的并发机制

2. GetThreadPriority()函数的原型如下：

int GetThreadPriority(HANDLE hThread);

在上述GetThreadPriority()函数的原型之中，各参数含义如下：

※**hThread**：是指要获取优先级的线程的句柄。

在调用GetThreadPriority()函数时，如果发生错误，则返回值为0；否则，返回非0值。

3.5.4 终止一个线程

1. ExitThread()函数用于正常结束一个线程的执行，其原型如下：

VOID ExitThread(DWORD status);

在上述ExitThread()函数的原型之中，各参数的含义如下：

※**status**：是指线程的终止状态。

第三章：服务器与客户进程中的并发机制

2. TerminateThread()函数用于立即中止线程的执行（理论上最好不去使用），其原型如下：

BOOL TerminateThread(HANDLE thread, DWORD status);

在上述TerminateThread()函数的原型之中，各参数的含义如下：

※**thread**：是指将要终止的线程的句柄。

※**status**：是指线程的终止状态。

3. GetExitCodeThread()函数用于获取线程结束时的退出码，其原型如下：

BOOL GetExitCodeThread (HANDLE hThread, LPDWORD lpExitCode);

第三章：服务器与客户进程中的并发机制

在上述GetExitCodeThread()函数的原型之中，各参数含义如下：

※**hThread**：是指将要退出的线程的句柄，也就是CreateThread()的返回值。

※**lpExitCode**：用于存储线程结束代码，也就是线程的返回值。

3.5.5 挂起/启动一个线程

在Windows环境下，每个执行的线程都有与其相关的挂起计数。如果这个计数为0，那么不会挂起线程。如果为非0值，则线程就会处于挂起状态。其中，用于挂起一个线程的Windows API函数为SuspendThread()函数，每次调用SuspendThread()函数之后都会增加挂起计数。

第三章：服务器与客户进程中的并发机制

而用于重新启动一个线程的Windows API函数为ResumeThread()函数，每次调用ResumeThread()函数之后都会减小挂起计数。挂起的线程只有在它的挂起计数达到0时才会恢复。因此，为了恢复一个挂起的线程，对ResumeThread()函数的调用次数必须与对SuspendThread()函数的调用次数相等。SuspendThread()函数和ResumeThread()函数均返回线程先前的挂起计数，如果发生错误，则返回值为–1。

SuspendThread()函数的原型如下所示：

```
#include <windows.h>
DWORD SuspendThread(HANDLE hThread);
```

在上述SuspendThread()函数的原型之中，各参数的含义如下：

※**thread**：是指将要挂起的线程的句柄。

第三章：服务器与客户进程中的并发机制

2. ResumeThread()函数的原型如下所示：

```
#include <windows.h>
DWORD ResumeThread(HANDLE hThread);
```

在上述ResumeThread()函数的原型之中，各参数的含义如下：

※**thread**：是指将要重新启动的线程的句柄。

3.5.6 获得一个线程的线程标识

GetCurrentThreadId()函数用于获取当前线程的标识，该函数返回当前线程的伪句柄（Pseudohandle）。之所以称之为伪句柄，是因为它是一个预定义的值，总是引用当前的线程，而不是引用指定的调用线程。然而，它能够用在任何可以使用普通线程处理的地方。GetCurrentThreadId()函数的原型如下：

```
HANDLE GetCurrentThread();
```

第三章：服务器与客户进程中的并发机制

3.5.7多线程例程剖析

为了进一步说明上述各线程函数的具体用法，下面给出一个简单的多线程的例程：

```
#include <stdio.h>
#include <windows.h>
//子线程函数
DWORD WINAPI ThreadFun(LPVOID pM) {
    printf("子线程ID：%d, 输出Hello World\n",
    GetCurrentThreadId());
    return 0;
}
```

第三章：服务器与客户进程中的并发机制

```
//主函数，所谓主函数其实就是主线程执行的函数。
int main(){
    printf("最简单的创建多线程实例!\n");
    HANDLE handle = CreateThread(NULL, 0, ThreadFun,
NULL, 0, NULL);
    WaitForSingleObject(handle, INFINITE);
    return 0;
}
```

第三章：服务器与客户进程中的并发机制

3.6从线程/进程分配技术

3.6.1从线程/进程预分配技术

在并发服务器中，若针对每一个到达的客户连接请求，均创建一个新的从线程/进程来进行处理，在处理完该请求之后，该从线程/进程再退出。这样在服务器负载很重的时候，将导致过多的线程/进程创建开销。为此，为了降低操作系统创建线程/进程所需的额外开销、提高服务器的吞吐率，人们提出了预分配技术。

在采用预分配技术的服务器设计之中，设计人员一般按以下方法来编写服务器程序：服务器在启动时就创建若干个并发的从线程/进程，每个从线程/进程都使用操作系统中提供的设施以等待客户请求的到达，当客户请求到达后，其中一个空闲的从线程/进程就开始执行并处理该客户请求，当完成客户请求的处理后，从线程/进程不退出，而是重新返回到等待客户请求到达的状态。

第三章：服务器与客户进程中的并发机制

显然，在基于预分配技术的并发服务器中，由于服务器不需要在客户请求到达时创建从线程/进程，避免了在每次客户请求到达时创建从线程/进程的开销，因此可更快地处理客户请求，减低了服务器的时延。特别地，当客户请求的处理涉及到的I/O多于计算时，由于预分配技术允许服务器系统在等待与前一个请求相关的I/O活动时，切换到另一个从线程/进程，并开始处理下一个请求，因此，此时采用预分配技术就显得尤为重要。

另外，在多处理器上采用预分配技术还可允许设计人员使服务器的并发等级与服务器的硬件性能相关联。如果服务器有K个处理器，则设计人员可预分配K个从线程/进程，由于多处理器操作系统给每个从线程/进程一个单独的处理器，为此，采用预分配技术可保证服务器的并发等级与硬件之间的匹配，从而服务器可获得尽可能高的处理速率。

第三章：服务器与客户进程中的并发机制

3.6.2延迟的从线程/进程分配技术

虽然预分配技术可提高服务器的效率，但它不能解决所有问题。

例如：由前述分析可知，预先创建从线程/进程不但需要消耗时间和服务器资源，也会为操作系统管理从线程/进程带来额外的开销，另外，预分配多个试图接收传入客户请求的从线程/进程还可能会给网络代码增添额外的开销，因此，只有当预先创建额外的从线程/进程能提高系统的吞吐率或降低系统的时延时，采用预分配技术才是合理的。

第三章：服务器与客户进程中的并发机制

但是，由于处理客户请求的时间是与客户请求直接相关的，因此，设计人员有时候可能会无法预先明确知道采用预分配技术是否合理。此时，可采用一种称为延迟的从线程/进程分配（Delayed Slave Allocation）技术。该技术的主要思想如下：服务器在启动时先循环地处理每个客户请求，只有当处理需要花费很长时间时，服务器才创建一个并发的从线程/进程来处理该请求。在LINUX中，采用延迟的从线程/进程分配技术并不难，只需要在主线程中设置一个计时器，并设计在计时器到期时调用fork()函数创建一个新的从进程，由于该从进程将从父进程处继承打开的套接字以及执行程序和数据的副本，因此，该从进程将恰好可从父进程超时所执行代码处继续进行处理。

第三章：服务器与客户进程中的并发机制

3.6.3两种从线程/进程分配技术的结合

由前述分析可知，预分配技术提高了在客户请求到达前的服务器的并发等级，而延迟的从线程/进程分配技术则提高了在客户请求到达服务器后的并发等级。这两种技术均是通过把服务器的并发等级从当前活跃的客户请求数目中分离出来，从而使得设计人员可获得灵活性并由此提高服务器的效率。显然，这两种技术可按以下方法结合使用：

服务器在启动时不采用预分配技术而是采用延迟分配技术，此时，服务器没有预先分配的从线程/进程；当有客户请求到达时，若处理需要花费很长时间时，服务器创建一个并发的从线程/进程来处理该请求；但一旦创建了从线程/进程后，该从线程/进程在处理完客户请求之后不必立即退出，它可以认为自己是永久分配的，并继续运行，在处理完一个客户请求之后，继续等待下一个客户请求的到达。

第三章：服务器与客户进程中的并发机制

在上述结合了两类并发技术的系统之中，需要对服务器的并发等级进行控制。常用的两种方法如下：第一种方法是设法让主线程/进程在创建一个从线程/进程时，指明其最大增长值M，从而限定了系统最终可达到的并发等级的最大值。另一种方法就是设法让一段时期内不活跃的从线程/进程退出，从线程/进程在等待下一个客户请求前先启动一个计时器，若在下一个客户请求到达之前计时器到期，则从线程/进程就退出。

3.7 基于多进程与基于多线程的并发机制的性能比较

3.7.1 多进程与多线程的任务执行效率比较

第三章：服务器与客户进程中的并发机制

究竟何时该采用多进程的并发模式，何时该采用多线程的并发模式？为了具体地说明该问题，下面将通过一个UNIX/Linux环境下的简单例子来进行分析比较。在该例子中包含有两个例程，其中，一个是多进程的例程，而另一个是多线程的例程。这两个例程所实现的功能完全相同，都是首先创建“若干”个新的进程/线程，然后再由每个新创建出的进程/线程分别负责打印出“若干”条“Hello Linux”字符串到控制台与日志文件，其中，这里的两个“若干”是分别由两个宏 P_NUMBER与COUNT来定义的。这两个例程的具体代码实现如下：

第三章：服务器与客户进程中的并发机制

1. 多进程并发例程的 C 语言代码（process.c）

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#define P_NUMBER 255                    //定义并发进程的数量
#define COUNT 100                       //定义每个进程打印字符串的次数
#define TEST_LOGFILE "logFile.log"
FILE *logFile = NULL;
char *s = "hello linux\0";
int main() {
    int i = 0, j = 0;
    logFile = fopen(TEST_LOGFILE, "a+");    //打开日志文件"logFile.log"
```

第三章：服务器与客户进程中的并发机制

```
for(i=0; i<P_NUMBER; i++){                    /*若并发线程的个数没有超过给
                                                定的最大值 P_NUMBER*/
    if(fork() == 0) {                  //创建子进程并在子进程中执行以下操作
        for(j=0; j<COUNT; j++) {
            printf("[%d]%s\n", j, s);     //打印输出 COUNT 次字符串 s
            fprintf(logFile,"[%d]%s\n",j,s);   /*向日志文件输出 COUNT
                                                次字符串 s*/
        }
        exit(0);                                          //结束子进程
    }
}
for(i=0; i<P_NUMBER; i++){          //调用 wait()函数回收所有的子进程
    wait(0);
}
```

第三章：服务器与客户进程中的并发机制

```
    printf("OK\n");
    return 0;
}
```

2. 多线程并发例程的 C 语言代码（thread.c）

```
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#define P_NUMBER 255                    //并发线程的最大个数
#define COUNT 100                       //每个线程打印字符串的次数
#define Test_Log "logFIle.log"
FILE *logFile = NULL;
char *s = "hello linux\0";
```

第三章：服务器与客户进程中的并发机制

```
print_hello_linux(){                                    //线程执行的线程体函数
    int i = 0;
    for(i = 0; i < COUNT; i++) {
        printf("[%d]%s\n", i, s);                       //向控制台输出信息
        fprintf(logFile, "[%d]%s\n", i, s);             //向日志文件输出信息
    }
    pthread_exit(0);                                    //结束线程
}
int main() {
    int i = 0;
```

第三章：服务器与客户进程中的并发机制

```
pthread_t pid[P_NUMBER];          //用于记录线程 ID 的数组 pid[]
logFile = fopen(Test_Log, "a+");  //打开日志文件
for(i=0; i<P_NUMBER; i++)         /*若并发线程的个数没有超过给定
                                    的最大值 P_NUMBER*/
    /*以下语句用于调用 pthread_create()函数创建新的子线程*/
    pthread_create(&pid[i], NULL, (void *)print_hello_linux, NULL);
for(i = 0; i < P_NUMBER; i++)
    pthread_join(pid[i],NULL);    /*调用 pthread_join()函数隐式地终止
                                    所有子线程
printf("OK\n");
return 0;
}
```

第三章：服务器与客户进程中的并发机制

3. 任务执行效率比较结果

基于上述两个例程，通过在每批次的实验中修改宏 P_NUMBER 和 COUNT 的值来调整进程/线程的数量与打印次数，每批次测试五轮后计算平均值，在 Linux2.6、单核单 CPU 的 i386 处理器环境下，所得到的结果如下表 4.1 至表 4.4 所示：

表 4.1 第 1 次实验结果（进程/线程数 255 / 打印次数：100）

	第 1 次	第 2 次	第 3 次	第 4 次	第 5 次	平均
多进程	0m1.277s	0m1.175s	0m1.227s	0m1.245s	0m1.228s	0m1.230s
多线程	0m1.150s	0m1.192s	0m1.095s	0m1.128s	0m1.177s	0m1.148s

第三章：服务器与客户进程中的并发机制

表 4.2 第 2 次实验结果（进程/线程数 255 / 打印次数：500）

	第 1 次	第 2 次	第 3 次	第 4 次	第 5 次	平均
多进程	0m6.341s	0m6.121s	0m5.966s	0m6.005s	0m6.143s	0m6.115s
多线程	0m6.082s	0m6.144s	0m6.026s	0m5.979s	0m6.012s	0m6.048s

表 4.3 第 3 次实验结果（进程/线程数 255 / 打印次数：1000）

	第 1 次	第 2 次	第 3 次	第 4 次	第 5 次	平均
多进程	0m12.155s	0m12.057s	0m12.433s	0m12.327s	0m11.986s	0m12.184s
多线程	0m12.241s	0m11.956s	0m11.829s	0m12.103s	0m11.928s	0m12.011s

第三章：服务器与客户进程中的并发机制

表 4.4 第 4 次实验结果（进程/线程数 255 / 打印次数：5000）

	第 1 次	第 2 次	第 3 次	第 4 次	第 5 次	平均
多进程	1m2.182s	1m2.635s	1m2.683s	1m2.751s	1m2.694s	1m2.589s
多线程	1m2.622s	1m2.384s	1m2.442s	1m2.458s	1m3.263s	1m2.614s

从以上实验数据可得出以下结果：当任务量较大时（打印次数大于等于 5000 次时），多进程比多线程的效率要高；而当任务量较小时（打印次数小于 5000 次时），则多线程要比多进程快。但整体上来看，多线程比较多进程在效率上没有太大的优势。

第三章：服务器与客户进程中的并发机制

3.7.2 多进程与多线程的创建与销毁效率比较

预先创建进程或线程可以节省进程或线程的创建、销毁时间，在实际的应用中很多程序使用了这样的策略，例如：Apapche采用了预先创建进程、Tomcat采用了预先创建线程的策略，预先创建的进程或线程通常被称为进程池或线程池。为了实际比较进程或线程的创建与销毁效率，下面将通过一个简单的例子来进行分析比较。在该例子中包含有两个例程，其中，一个是多进程的例程，而另一个是多线程的例程。这两个例程所实现的功能完全相同，都是计算创建与销毁10万个进程/线程所需的绝对用时。这两个例程的具体代码实现如下：

第三章：服务器与客户进程中的并发机制

1. 多进程并发例程的 C 语言代码（process_time.c）

```
#include <stdlib.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/types.h>
#include <signal.h>
#include <sys/wait.h>
int count;                    //声明记录子进程创建成功个数的全局变量 count
int fcount;                   //声明记录子进程创建失败个数的全局变量 fcount
int scount;                   //声明子进程回收数量
```

第三章：服务器与客户进程中的并发机制

```
void sig_chld(int signo) {        //信号处理函数–子进程关闭收集
    pid_t chldpid;                //子进程 id
    int stat;                     //子进程的终止状态
    while ((chldpid = wait(&stat)) > 0) {   //子进程回收，避免出现僵尸进程
        scount++;
    }
}
int main() {
    signal(SIGCHLD, sig_chld);          //注册子进程回收信号处理函数
    int i;
```

第三章：服务器与客户进程中的并发机制

```
for (i = 0; i < 100000; i++) {          //循环创建 100000 个子进程
    pid_t pid = fork();                 //调用 fork()函数创建子进程
    if (pid == -1) {                    //若子进程创建失败
        fcount++;                       //则将创建失败的子进程的个数加 1
    }
    else if (pid > 0) {                 //若子进程创建成功
        count++;                        //则将创建成功的子进程的个数加 1
    }
    else if (pid == 0) {                //子进程的执行过程
        exit(0);                        //退出子进程
    }
}
printf("count: %d fcount: %d scount: %d\n", count, fcount, scount);
                                        //输出创建成功与失败的子进程的个数
}
```

第三章：服务器与客户进程中的并发机制

2. 多线程并发例程的 C 语言代码（thread_time.c）

```
#include <stdio.h>
#include <pthread.h>
int count = 0;                          //成功创建线程数量
void thread(void) {
                                        //子线程啥也不做
}
int main(void) {
    pthread_t id;                       //声明用于记录线程 ID 的局部变量 id
    int i, ret;                         /*声明用于计数记录的局部变量 i 和用于记录
                                          pthread_create()返回值的局部变量 ret*/
```

第三章：服务器与客户进程中的并发机制

```
for (i = 0; i < 100000; i++) {          //创建 10 万个子线程
        ret = pthread_create(&id, NULL, (void *)thread, NULL);
        if(ret != 0) {                  //若创建子线程出错则提示出错信息
            printf ("Create pthread error!\n");
            return (1);
        }
        count++;                        //将子线程的个数加 1
        pthread_join(id, NULL);    /*在主线程中调用 pthread_join 函数来
                                     实现对子线程的隐式地终止*/
    }
    printf("count: %d\n", count);       //输出子线程的个数信息
}
```

第三章：服务器与客户进程中的并发机制

3. 创建与销毁效率比较结果

基于上述两个例程，采用测试五轮后计算平均值，在 Linux2.6、赛扬 1.5G 的 CPU 环境下，所得到的结果如下表 4.5 所示：

表 4.5 多进程与多线程的创建与销毁效率比较

创建销毁 10 万个进程	创建销毁 10 万个线程
0m18.201s	0m3.159s

第三章：服务器与客户进程中的并发机制

从表 4.5 中的数据结果可以看出，多线程比多进程在创建与销毁效率上有 5-6 倍的优势。但由于平均创建销毁一个进程仅约需要 0.18 毫秒（= 0m18.201s/100000），且预先派生子进程/线程需要对池中进程/线程数量进行动态管理，比现场创建子进程/线程要复杂很多，因此对于当前服务器几百或几千的并发量，预先派生线程也不见得比现场创建线程快。

- **3.8 本章小结**

本章主要对基于多进程或多线程的服务器并发机制及其 C 语言实现方法进行了详细介绍，并在 UNIX/LINUX 环境下针对多进程与多线程并发机制的性能进行了深入比较。通过本章学习，需要了解基于多进程或多线程的服务器并发概念；需要熟习基于多进程与多线程的并发机制、从线程/进程预分配技术以及延迟的从线程/进程分配技术；需要掌握 UNIX/LINUX 和 Windows 环境下基于多进程与多线程并发软件的 C 语言实现方法。

第三章：服务器与客户进程中的并发机制

本章作业

第1-7题

谢谢！