套接字程序设计II
(Socket Programming II)

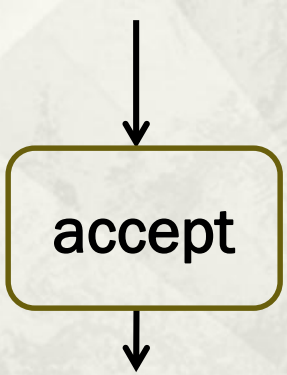
套接字并发编程

参考　　参考

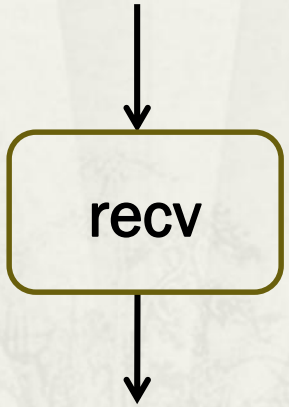
中山大学 计算机系 张永民
2020年4月30日

套接字编程的阻塞问题

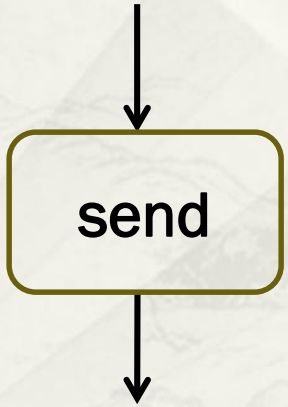
- 套接字函数**accept**、**recv**等函数会因缺乏资源而发生阻塞。发生阻塞时，程序将不再往前执行了，所占用的**CPU(核)**会被操作系统切换给其它线程。



当连接请求队列为空时执行该函数会发生阻塞



当接收缓冲区为空时执行该函数会发生阻塞

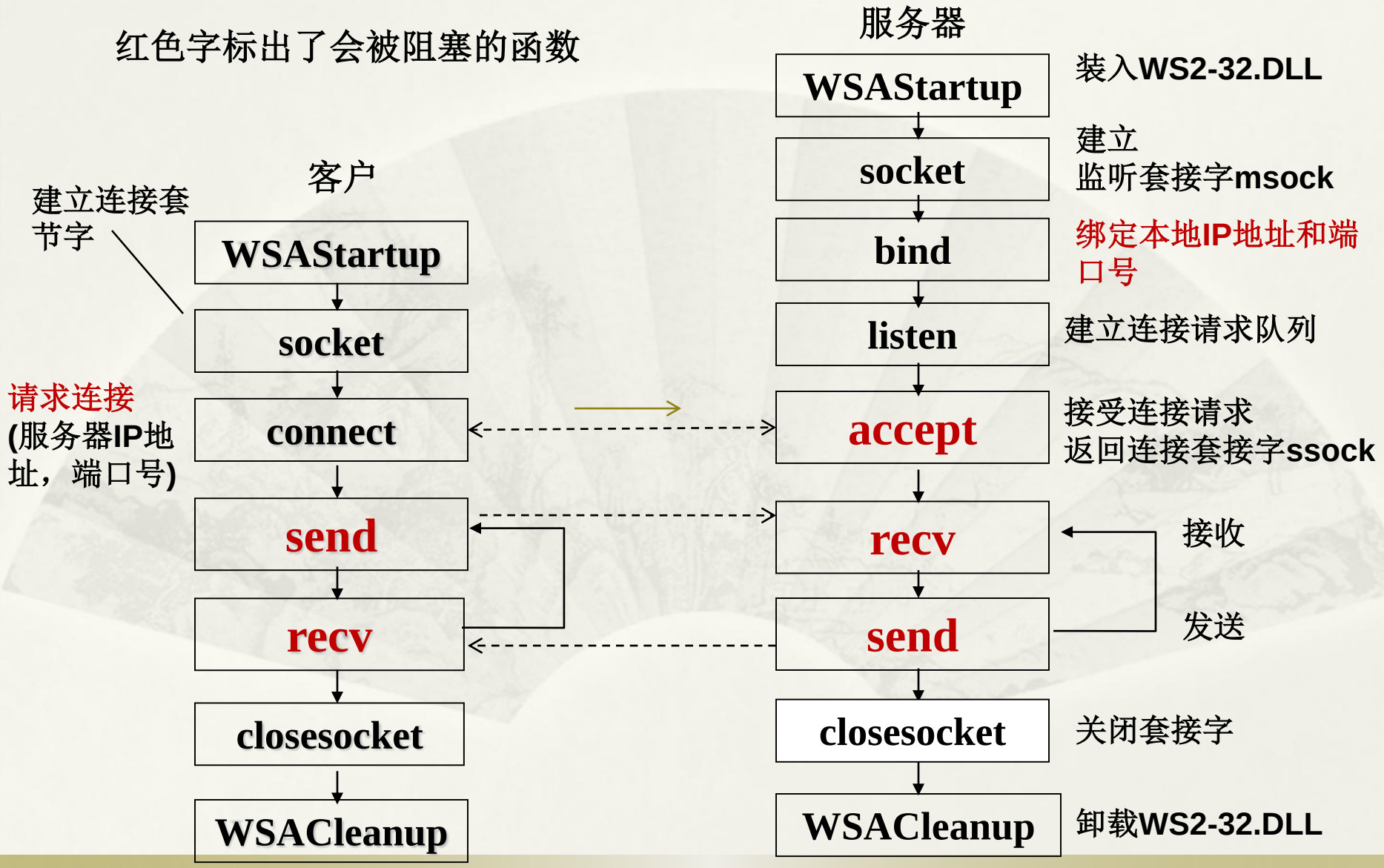


当发送缓冲区满时执行该函数会发生阻塞

解决方法：（1）线程 （2）非阻塞套接字

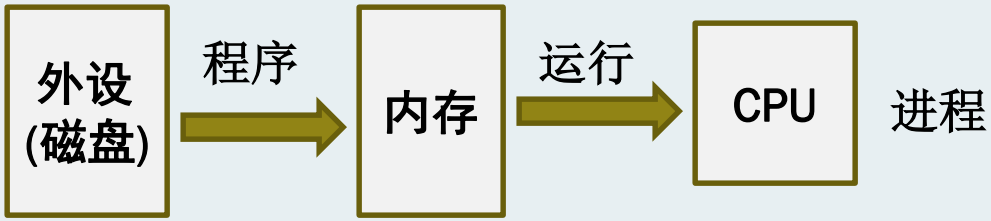
循环模式下的套接字编程

红色字标出了会被阻塞的函数



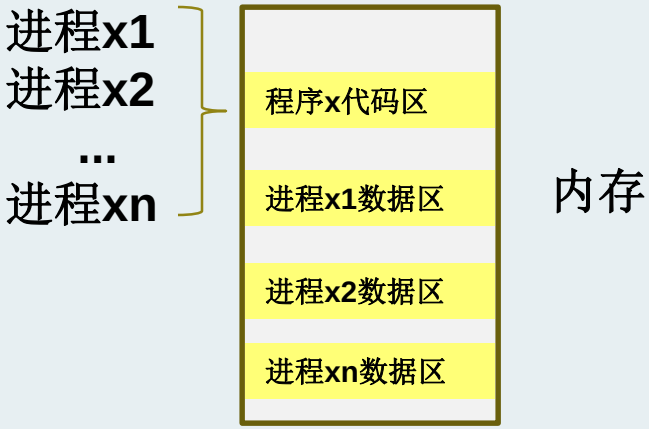
进程

- 进程(process)是装入内存执行或准备执行的程序。程序被执行一次就产生一个进程。

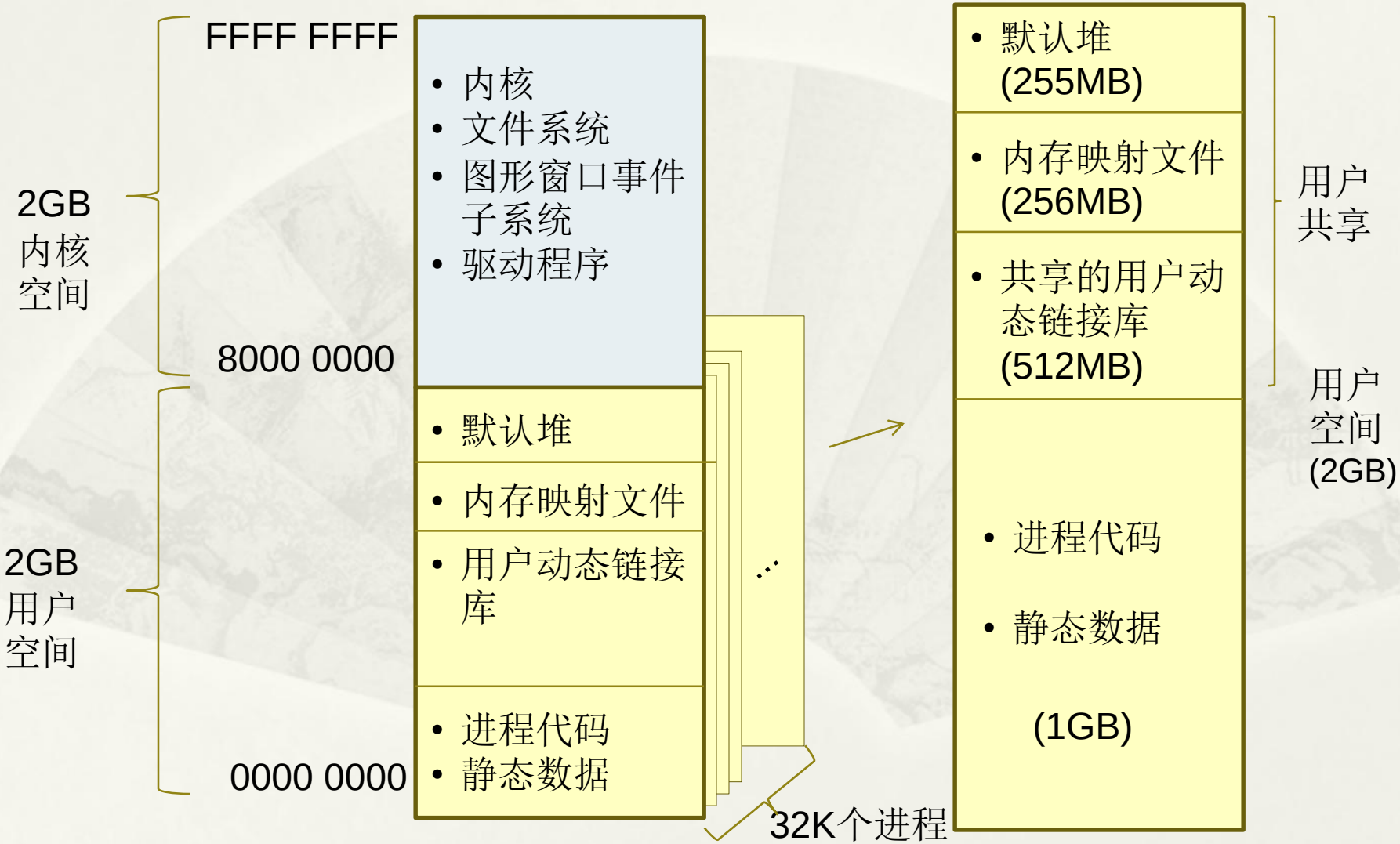


执行n次程序x得到：进程x1，进程x2，...，进程xn

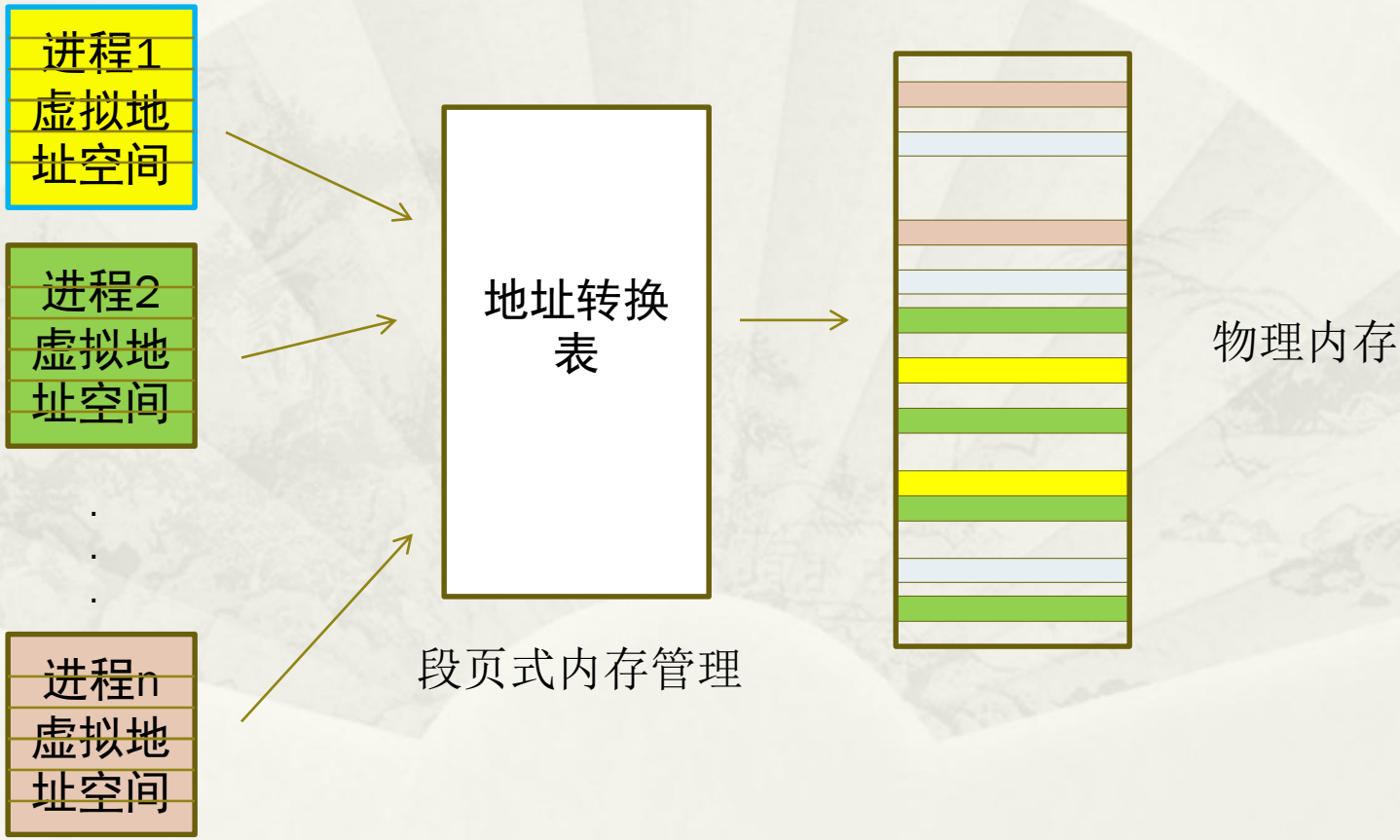
- 一个应用程序的所有进程共享程序代码，但是每个进程都具有自己的数据区。



- 每个32位Windows的进程使用独立的4GB的虚拟地址空间。
- 每个进程的用户空间有它自己的映射，包括栈和堆的建立。



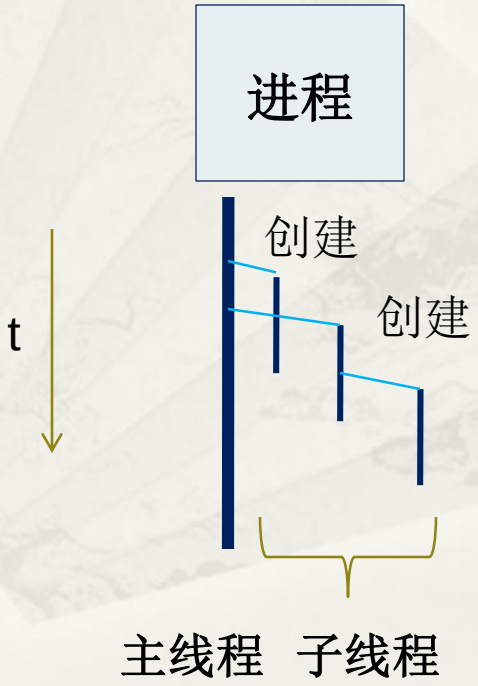
- Windows所有进程都在自己的虚拟地址空间执行指令。
- 物理内存由所有进程所共享。按需调入内存。当内存不够时，长久不使用或使用较少的页被调出内存。调入的内核程序页由所有进程共享。
- 进程之间的通信不能直接通过共享变量来访问，需要使用其它方法，例如：命名管道、邮槽、套接字，效率较低。



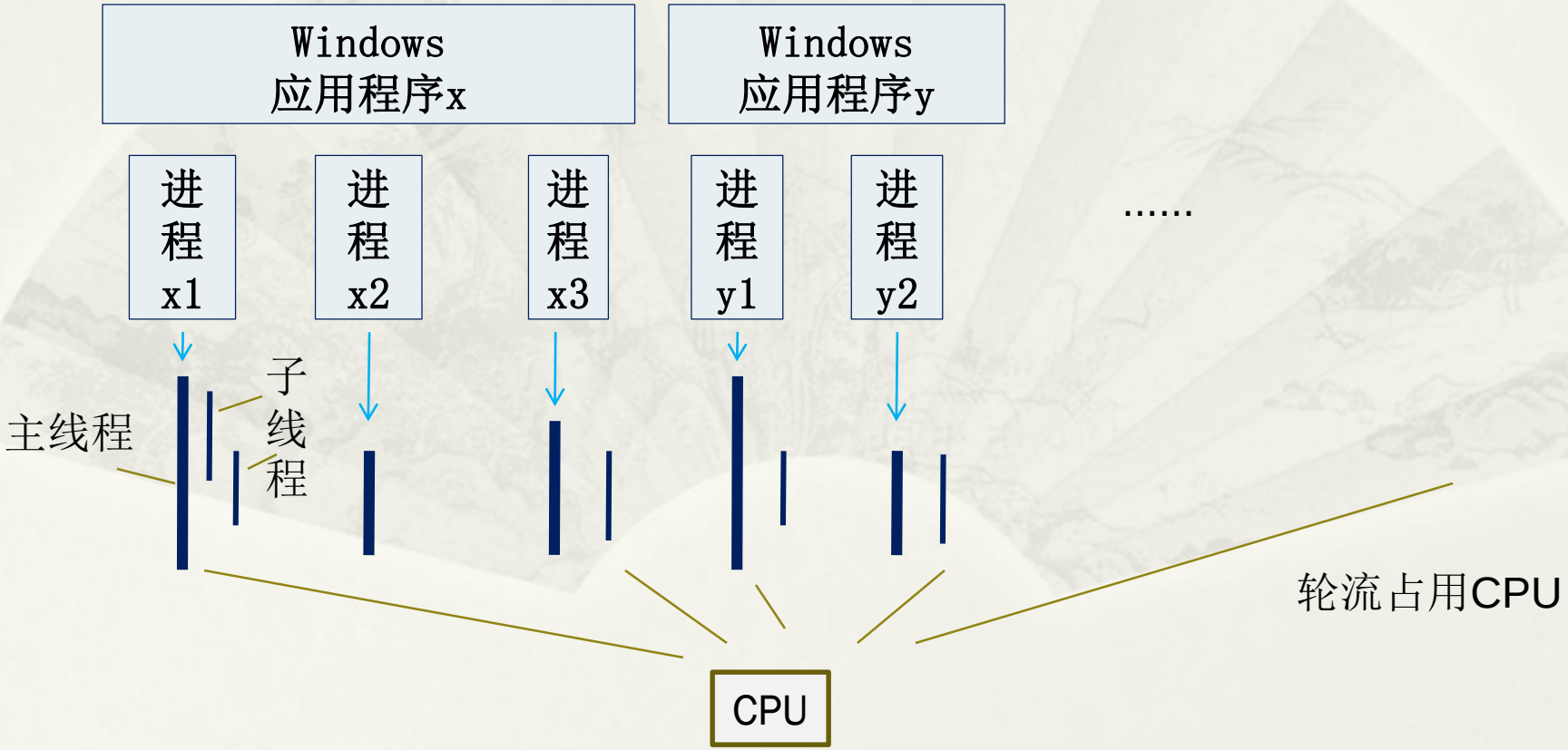
段页式内存管理

线程

- Windows进程是惰性的，实际执行由线程(thread)完成。
- 每个Windows进程在创建时自动建立一个**主线程**。一旦主线程停止了，整个程序的执行就结束了。
- 主线程可以建立更多的子线程。
  子线程还可以继续创建线程。
- **一个进程创建的所有线程共享进程的虚拟地址空间**。所以，一个进程所创建的线程可以直接访问该进程具有的所有资源，例如：该进程(主线程)的全局变量和内核句柄。



- 线程是Windows操作系统中可以占用CPU执行的基本对象。系统的所有线程轮流占用CPU执行。
- 正在占用CPU执行的线程处于执行态(running)。如果缺少资源，线程将会停止执行并让出CPU，即处于阻塞态(blocked)，当获得所缺资源时线程将变为就绪态(runnable)，排队等待获得CPU执行。
- 对于多核CPU，可以同时执行多个线程。例如，4核8线程的CPU可以同时执行8个线程。



- Windows采用抢占式调度方式管理线程的执行。高优先权的线程一旦就绪将立即剥夺正在执行的低优先权线程，并占用CPU。相同级别的线程采用时间片(约20 ms)轮转的方法占用CPU。新线程占用CPU时需要进行上下文切换，主要是保存CPU内的环境（寄存器、程序计数器等）以便下次执行被移出的线程前再恢复环境，这需要比较大的开销。
- Windows线程的优先级：0~31，其中，0最低，31最高。
- 为了防止冲突和死锁，线程对共享资源的访问需要采用互斥和同步技术进行控制。
- 进程结束时系统将释放其未关闭的线程资源。
- 线程结束时将执行操作系统调度进程以选择新的线程执行。

线程创建函数

在Windows中，每次用_beginthreadex去调用一个函数就可以产生一个新线程。

```
sumAll(void *pthrno){
  ......
}

void main(){
    HANDLE h1,h2;
    int  p1=1, p2=2;
    h1 = _beginthreadex(NULL, 0,&sumAll, (void *)&p1, 0, NULL);
    h2 = _beginthreadex(NULL, 0,&sumAll, (void *)&p2, 0, NULL);
}
```

返回线程标识符

默认安全性

默认堆大小

函数名

参数

立即运行(也可以先挂起)

线程创建示例程序

```
/* 创建独立子线程求和(1+2+...+n), 并用全局变量sumall保存所有线程的总和 */
#include <windows.h>
#include <stdio.h>
#include <process.h>
#include <math.h>
double sums[]={0,0,0};                  // 用于返回每个线程的和值
double allsum=0;                        // 共享变量，用于记录所有线程总和。

unsigned __stdcall sumAll(void *pthrno) {
    int thrno = *((int *)pthrno);
    double sum=0;
    for(int i=0;i<40000;i++){
        sum+=i;
        allsum+=i;
        printf("%d'th thread: the sum from 1 to %d is %0.0f. total is %0.0f\n",
              thrno, i, sum, allsum);
        fflush(stdout);
     }
     sums[thrno]=sum;     // 用全局变量返回和值
     return 0;            // 返回时自动结束线程。_endthreadex(0)也可以用于结束线程
}
```

```
void main(){
    HANDLE hThread1,hThread2;
    int  p1=1, p2=2;      //自定义线程编号，用于访问数组sums

    hThread1 = (HANDLE)_beginthreadex(NULL, 0,&sumAll, (void *)&p1, 0, NULL);
    hThread2 = (HANDLE)_beginthreadex(NULL, 0,&sumAll, (void *)&p2, 0, NULL);
    WaitForSingleObject(hThread1, INFINITE );   //等待线程hThread1结束
    WaitForSingleObject(hThread2, INFINITE );   //等待线程hThread2结束

    printf("Finished! The sums are %0.0f and %0.0f.the total is %0.0f.\n",
           sums[1],sums[2],sums[1]+sums[2]);
    printf("Finished! The shared sum is %0.0f.\n",allsum);
    CloseHandle(hThread1);              // 关闭线程句柄，释放线程资源
    CloseHandle(hThread2);
    getchar();                          // 按任意键退出
}
```

- 共享变量要使用临界区才能访问，否则可能会出错。
- 要带入多个参数到线程中需要使用结构类型(struct)的变量。

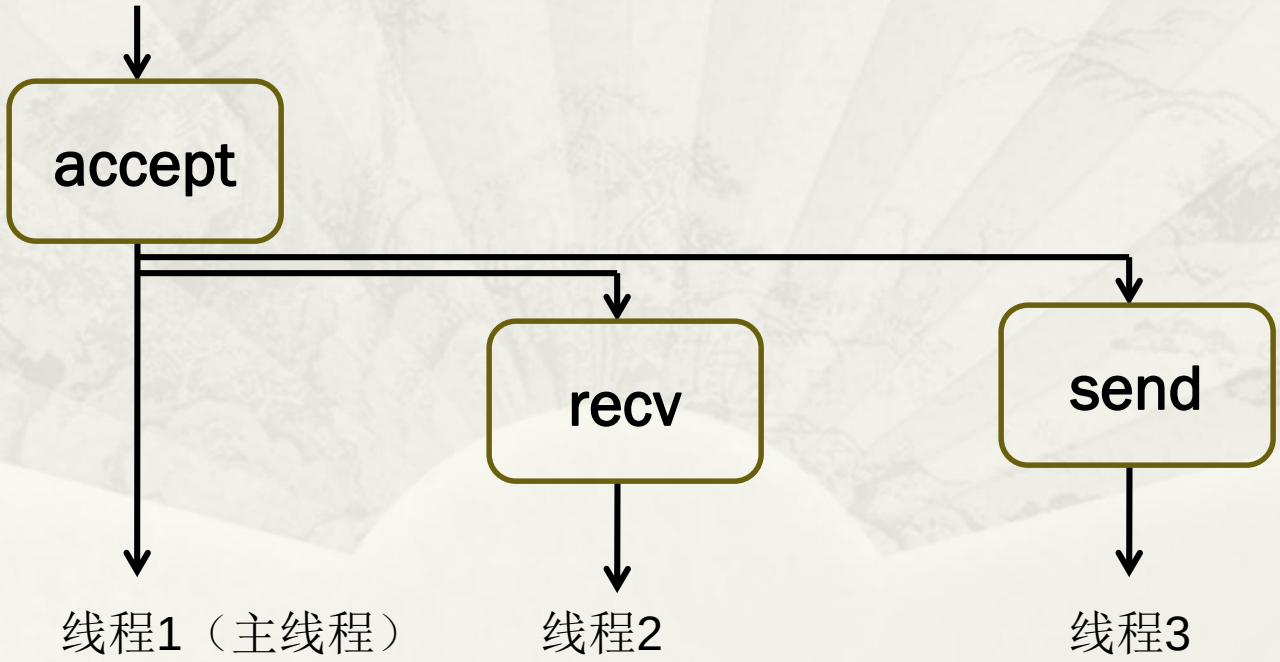
临界区

通过使用临界区(Critical Section)控制对共享变量的访问。

```
CRITICAL_SECTION cs;                    // 临界区
double sumall=0;                        // 记录所有线程总和值。
unsigned __stdcall SumAll(void * p){
     EnterCriticalSection(&cs);         // 等待进入临界区
     sumall+=i;                         // 每个时刻只能有一个线程进入临界区。
     LeaveCriticalSection(&cs);         // 离开临界区
}
void main(){
     ...
     InitializeCriticalSection(&cs);            // 临界区初始化
     hThread1 = (HANDLE)_beginthreadex(NULL, 0,&SumAll,
                                       (void *)ptr1, 0, &threadID);
     hThread2 = (HANDLE)_beginthreadex(NULL, 0,&SumAll,
                                       (void *)ptr2, 0, &threadID1);
     WaitForSingleObject(hThread1, INFINITE);
     WaitForSingleObject(hThread2, INFINITE);
     DeleteCriticalSection (&cs) ;              // 删除临界区
     ...
}
```

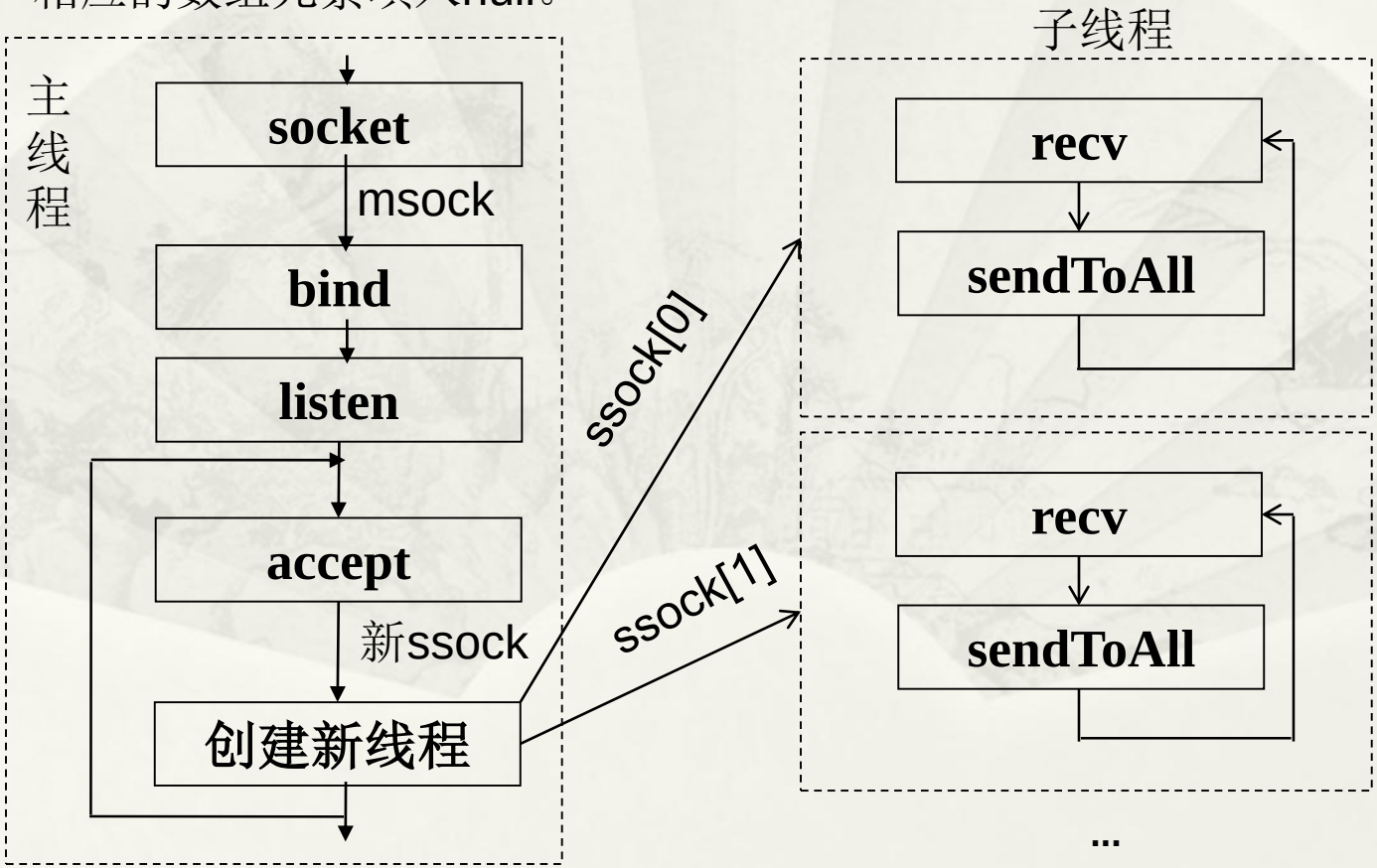
并发模式下的套接字编程

- 执行套接字函数accept、recv和send会产生阻塞。我们可以利用线程解决这个问题。
- 我们为每个阻塞点建立一个线程，这样当执行这些函数被阻塞时就不会影响其它任务的执行。



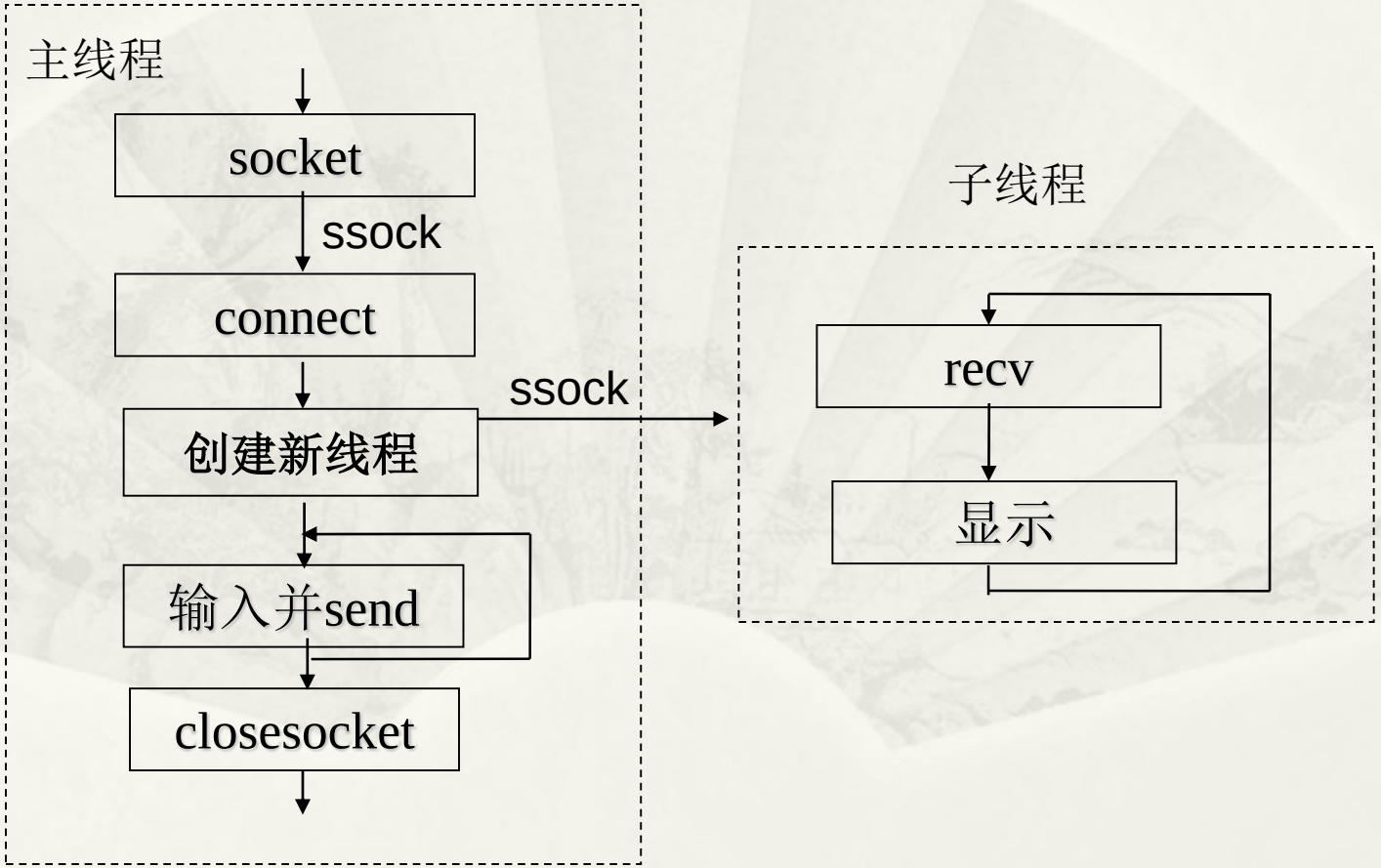
Chat并发编程(服务器)

- 主线程接收来自客户端的连接请求。每个子线程接收一个客户端发来的数据并转发给所有其它客户端。
- 为了每个线程都可以访问到，所有连接客户端的套接字可以采用一个**共享**数组保存。对于新ssock，在数组中找一个值为null的数组元素填入；当关闭了一个套节字，相应的数组元素填入null。



Chat并发编程(客户端)

主线程负责输入聊天文字并发送给服务器。子线程负责接收服务器传来的文字并显示出来。



_beginthreadex函数

```
uintptr_t _beginthreadex(                    // 建立线程
   void *security,                           // 安全属性，NULL使用默认安全性
   unsigned stack_size,                      // 新线程栈的大小或0(系统默认大小)
   unsigned (*start_address )(void *),       // 函数名参数
   void *arglist,                            // 传递给新线程的参数列表或NULL
   unsigned initflag,                        // 新线程的初始状态：0-运行
   unsigned *thrdaddr                        // 返回新线程标识或NULL(失败)
);
```

- 新线程的初始状态：　0 - 运行，　CREATE_SUSPENDED - 挂起
- 如果线程初始被挂起，以后需要用函数ResumeThread(线程句柄)恢复执行。
- 返回值：线程句柄。
- thrdaddr为NULL时表示不要返回值。

WaitForSingleObject函数

```
DWORD WaitForSingleObject(     // 等待对象发生事件
        HANDLE hHandle,        // 对象句柄
        DWORD dwMilliseconds); // 等待时间(ms)：INFINITE 一直等待
```

- **功能：**
  等待hHandle事件发生。如果hHandle是某个线程的句柄，则其功能是等待该线程结束。
- **参数：**
  hHandle --对象句柄，如Event、Mutex、Process、 Semaphore、Thread、Waitable timer等的句柄。
  dwMilliseconds -- 等待时间(ms)，在这个时间内一直等待对象句柄的某个事件发生。如果有事件发生则立即返回，超过这个时间没有事件发生也返回。0 - 立即返回， INFINITE - 一直等待。
- **返回：**执行成功，返回值指示出引发函数返回的事件。

recv函数

```
int recv(
    SOCKET s,      // 一个标识的已连接套接字的描述符（in）
    char* buf,     // 指向要接收数据的用户缓冲区（out）
    int len,       //上述缓冲区的长度(字节数)  (in)
    int flags);    //影响接收方式的标志  (in)
```

功能： 从套节字缓冲区读取数据，并放进用户开辟的缓冲区buf。
如果缓冲区为空，则会被阻塞.

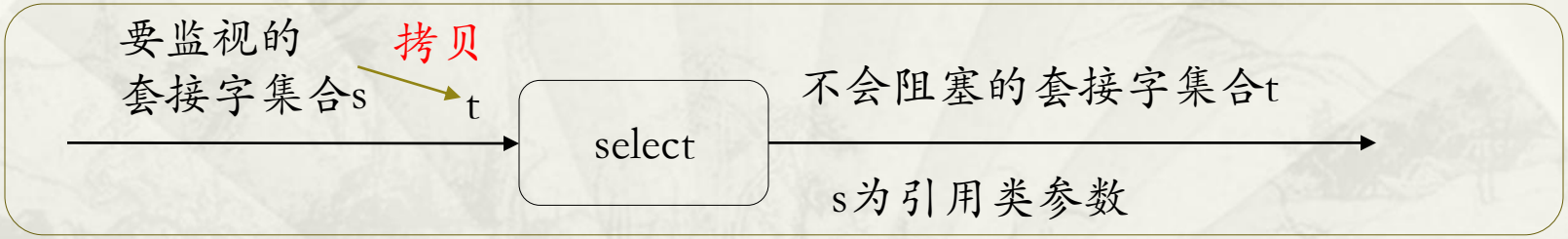
返回：

无错时，返回实际接收的字节数(不大于len)，缓冲区中包含所接收的数据。如果对方关闭了连接，将返回0。

出错时返回SOCKET_ERROR,可以调用函数WSAGetLastError取得错误代码。

用select函数实现套接字编程*

- 采用多线程的并发模式可以克服阻塞问题，但是太多的线程会造成很大的上下文切换开销，而且操作系统创建和销毁线程开销也很大。因此，这种模型能接受的最大连接数都不会高，一般在几百个左右。
- select函数采用单线程实现套接字编程。Select函数通过集合类型的参数带入要监视的（会被阻塞）套接字，并带出其中可以进行操作而不会阻塞的套接字，然后对这些套接字进行操作就不会发生阻塞。



- 每次带入的套接字前都要复制它们的句柄到集合中，连接数量很大时会复制产生巨大的开销。内核实现select是采用轮询(poll)的方式扫描文件描述符，当数量很多时，这个操作的开销很大。套接字集合的加入和删除操作在十分频繁时也要消耗大量的时间。
- 因此，select的连接数一般限制在1千个左右。
- linux和windows都支持select函数。

用epoll函数实现套接字编程*

- 在Linux下的另一种套接字模型是poll模型，它使用链表保存文件描述符，加快了从集合插入和删除套接字的速度，但select模型的其它缺点依然存在。
- epoll是Linux下的另一个单线程套接字模型，它会在操作系统内核中申请一个简易的文件系统，并通过这个文件系统监控所有的套接字。进程只需要往内核加入新的套接字句柄，而不用每次都拷贝所有要监视的套接字句柄。
- epoll采用**红黑树**和**双向链表**实现快速查找、插入和删除套接字句柄。
- epoll内核每次只把所有发生了事件的套接字句柄返回给进程，进程只需要对它们循环处理一遍。
- 采用epoll函数没有最大并发连接的限制，上限是最大可以打开文件的数目，可以达到百万级的并发连接数。**一般来说这个数目和系统内存关系很大**，具体数目可以在Linux的/proc/sys/fs/file-max中找到。

用IOCP函数实现套接字编程*

- **输入输出完成端口**（Input/Output Completion Port，IOCP）是Windows支持多个同时发生的异步I/O操作的应用程序编程接口（API）。IOCP可以让每一个socket有一个线程负责同步（阻塞）数据处理，特别适合C/S模式的网络服务器端模型。
- 为了防止线程过多又可以利用多核的优势，服务器端在采用**线程池**控制线程的数量的同时让工作线程的数量与CPU内核数量相同，以此来最小化线程切换代价。
- 一个IOCP对象，在操作系统中可以关联着多个Socket和（或）文件控制端。 IOCP对象内部有一个**先进先出（FIFO）队列**，用于存放IOCP所关联的输入输出端的服务请求完成消息。请求输入输出服务的进程不接收IO服务完成通知，而是检查IOCP的消息队列以确定IO请求的状态。
- 线程池中的线程负责从IOCP消息队列中取走完成通知并执行数据处理；如果队列中没有消息，那么线程阻塞挂起在该队列。这些线程从而实现了负载均衡。

- Windows中利用CreateIoCompletionPort命令创建完成端口对象时， 操作系统内部为该对象自动创建了5个数据结构，分别是：
  - **设备列表**（Device List）： 每当调用CreateIoCompletionPort函数时，操作系统会将该设备句柄添加到设备列表中；每当调用CloseHandle关闭了某个设备句柄时，系统会将该设句柄从设备列表中删除。
  - **IO完成请求队列**（I/O Completion Queue-FIFO）：当I/O请求操作完成时，或者调用了PostQueuedCompeltionStatus函数时，操作系统会将I/O请求完成包添加到I/O完成队列中。当操作系统从完成端口对象的等待线程队列中取出一个工作线程时，操作系统会同时从I/O完成队列中取出一个元素（I/O请求完成包。
  - **等待线程队列**（WaitingThread List-LIFO）：当线程中调用GetQueuedCompletionStatus函数时，操作系统会将该线程压入到等待线程队列中。为了减少线程切换，该队列是LIFO。当I/O完成队列非空，且工作线程并未超出总的并发数时，系统从等待线程队列中取出线程，该线程从自身代码的GetQueuedCompletoinStatus函数调用处返回并继续运行。
  - **释放线程队列**（Released Thread List）：当操作系统从等待线程队列中激活了一个工作线程时，或者挂起的线程重新被激活时，该线程被压入释放线程队列中，也即这个队列的线程处于运行状态。这个队列中的线程有两个出队列的机会：一是当线程重新调用GetQueuedCompeltionStatus函数时，线程被添加到等待线程队列中；二是当线程调用其他函数使得线程挂起时，该线程被添加到“暂停线程队列”中。
  - **暂停线程队列**（Paused Thread List）：释放线程队列中的线程被挂起的时候，线程被压入到“暂停线程队列”中；当挂起的线程重新被唤醒时，从“暂停线程队列”中取出放入到释放线程队列。