网络编程案例库实现各种服务器范式

目录

[一．服务器类型 1](#_Toc481431597)

[二． 运行环境与测试环境 1](#_Toc481431598)

[运行环境 1](#_Toc481431599)

[测试环境 2](#_Toc481431600)

[三．运行说明 2](#_Toc481431601)

[四．服务器代码编写简要说明及测试 3](#_Toc481431602)

[1.TCP服务器 3](#_Toc481431603)

[2. UDP服务器 6](#_Toc481431605)

[3. 多进程技术TCP并发服务器，每个客户一个子进程 7](#_Toc481431607)

[4. 多线程技术TCP并发服务器程序，每个客户一个线程 9](#_Toc481431610)

[5.TCP预先派生子进程服务器，accept无上锁保护。 11](#_Toc481431611)

[6. TCP预先派生子进程服务器程序，accept使用文件上锁保护 13](#_Toc481431612)

[7. TCP预先派生子进程服务器程序，accept使用线程锁上锁保护 16](#_Toc481431614)

[8. TCP预先创建线程服务器，每个线程各自accept 18](#_Toc481431615)

[9. TCP预先创建线程服务器程序，主线程统一accept 20](#_Toc481431616)

[10．I/O复用技术TCP服务器——select模型 22](#_Toc481431617)

[11．I/O复用技术TCP服务器——poll模型 26](#_Toc481431618)

[12．I/O复用技术TCP服务器——epoll模型 28](#_Toc481431619)

# 一．服务器类型

1. TCP服务器
2. UDP服务器
3. 多进程技术TCP并发服务器，每个客户一个子进程
4. 多线程技术TCP并发服务器程序，每个客户一个线程
5. 进程池技术TCP并发服务器，accept无上锁保护
6. 进程池技术TCP并发服务器，accept使用文件锁保护
7. 进程池技术TCP并发服务器，accept使用线程锁保护
8. 线程池技术TCP预先创建线程服务器程序，每个线程各自accept
9. 线程池技术TCP预先创建线程服务器程序，主线程统一accept
10. I/O复用技术TCP服务器——select模型
11. I/O复用技术TCP服务器——poll模型
12. I/O复用技术TCP服务器——epoll模型

二． 运行环境与测试环境

运行环境

|  |  |  |
| --- | --- | --- |
|  | 最低配置 | 建议配置 |
| 硬件 | 800MHZ及以上能力的CPU  128M及以上内存 | 1GHZ及以上处理能力的CPU  512及以上内存 |
| 软件 | 内核版本：linux2.6.32-2-g内核  GCC版本：4.4.3 | 内核版本：linux2.6.32-2-g内核及以上  GCC版本：4.4.3及以上 |

测试环境

|  |  |
| --- | --- |
| 硬件 | 奔腾酷睿 2.1HZ 内存4G |
| 操作系统 | Centos6.5 |
| 编译器 | Gcc 4.4.7 |
| 内核版本 | Linux version 2.6.32-431.el6.i686 |

# 三．运行说明

服务器端：

首先在linux系统下建立终端。先启动服务器，其中第1、2、3、4、11、12服务器端需要输入<IP> <PORT>。其4、5、7、8、9、10类型服务器端需要输入<IP> <PORT> <CHILDNUM>.

客户端：

在启动服务器之后客户端需要输入：<IP><PORT><CHIDNUM><CONNECTNUM><DATASIZE>

参数说明

|  |  |
| --- | --- |
| IP | 服务端IP地址 |
| PORT | 服务端端口号 |
| CHIDNUM | 生成的子进程（线程数） |
| CONNECTNUM | 进程各自发起的连接次数 |
| DATASIZE | 发送数据的大小（字节） |

# 四．服务器代码编写简要说明及测试

## 1.TCP服务器，

该服务器没有进程控制开销。服务器编写中主要用到了网络案例库libnet中的tcp准备函数，下面是服务器部分主要代码，代码的第14行为案例库tcp\_pre()函数的使用，第一个参数是绑定的IP地址，第二参数为绑定的端口号。

-------------------------------------------------------------- -----server/serv01/main.c

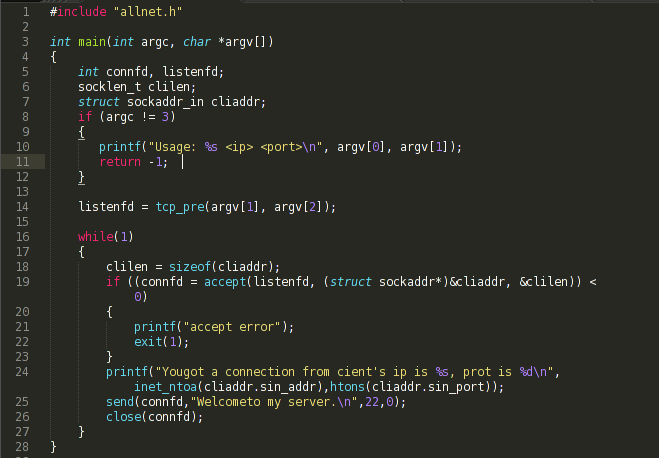


图1.1 TCP服务器main函数

-------------------------------------------------------------- ---server/serv01/tcp\_pre.c

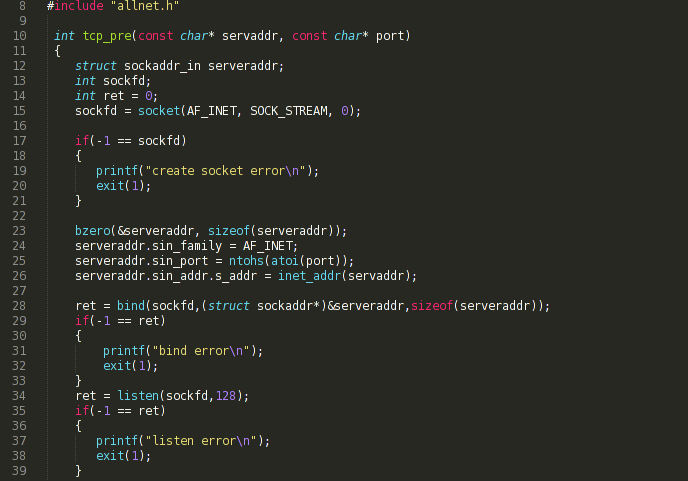


图1.2 tcp\_pre.c 代码

在服务器运行后，客户端要去连接服务器，我们的网络案例库编写封装了一个TCP客户端连接服务器函数,main.c的第8行引用了这个函数，该函数第一个参数为键入的参数个数，第二个参数为要连接的IP地址，第三个参数为要连接的端口。

客户端主要代码如下：

-------------------------------------------------------------- -----server/cli01/main.c

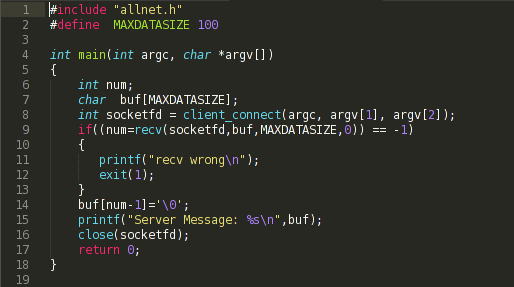


图1.3 客户端main函数

------------------------------------------------------------server/cli01/client\_connect.c



图1.4 客户端连接函数

运行截图如下：

服务器运行

服务器运行1.png

客户端连接成功

客户端链接.png

## 2. UDP服务器

UDP套接口是无连接的，不可靠的数据协议。我们的网络案例库主要实现了UDP服务器准备函数（包含服务器地址发现、套接字创建、地址绑定、服务器监听等）和UDP客户端连接服务器函数（包含服务器地址解析、套接字创建、服务器连接等）；

在编写UDP服务器过程调用网络编程案例库中的UDP服务器准备函数udp\_pre()进行套接字创建和初始化，接下来的udp\_work()函数服务器主要是服务器的业务逻辑（根据实际应用自己编写），该文档的UDP服务器实现的业务是接收客户端的连接和客户端发来的消息，然后打印出IP地址和端口号并且给客户端传送一句消息。

UDP客户读主要代码如下：

----------------------------------------------------------------------server/ucli/main.c

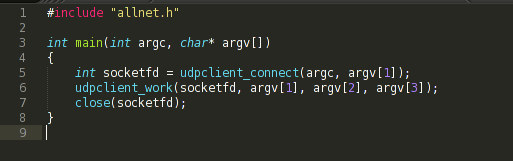


图2.1 udp客户端主函数

--------------------------------------------------------------------server/udpserv/main.c

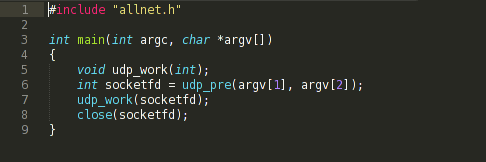


图2.2 udp服务器主函数

运行截图如下：

服务器运行

serv.png

客户端连接成功

client.png

3. 多进程技术TCP并发服务器，每个客户一个子进程

该服务器调用fork派生一个子进程来处理每个客户。在编写服务器时我们大概依据一下流程，首先调用案例库的tcp\_pre()来创建套接字并初始化，然后服务器accept客户端的连接请求，并派生子进程来处理客户端请求。对于该服务器模型我们的网络编程案例库封装了resource\_getback()处理僵死进程回收资源，serv\_accept()用于接收客户端请求、process\_create()函数用于创建子进程;process\_handle\_data()函数是子进程处理的业务逻辑处理，该函数不属于案例库，需要实际需求自己编写。

下面是服务器编写的主要代码：

-------------------------------------------------------------- ------server/serv03/main.c

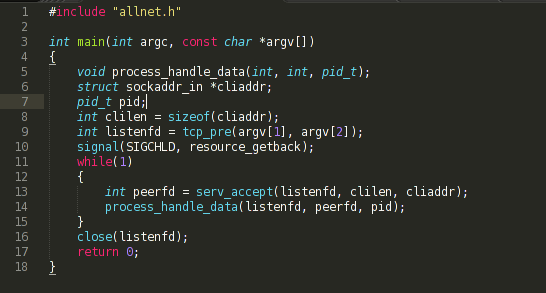


图3.1 多进程服务器main函数

-----------------------------------------------------------------server/serv03/process\_create.c

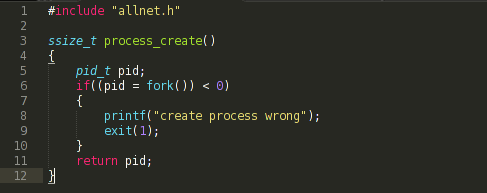


图3.2多进程服务器子进程构造函数

------------------------------------------------------server/serv03/resource\_getback.c

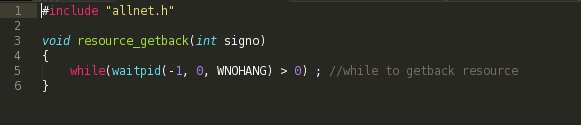


图3.3多进程服务器资源回收

------------------------------------------------------server/serv03/process\_work.c

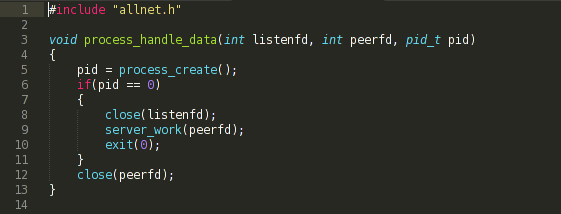


图3.4子进程工作函数

------------------------------------------------------server/serv03/process\_accept.c

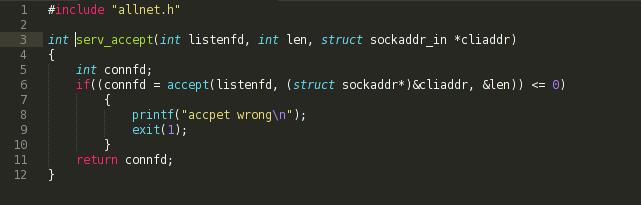


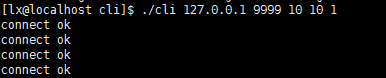
图3.5 serv\_accept函数

运行截图如下：

服务器运行

serv.png

客户端运行



## 4. 多线程技术TCP并发服务器程序，每个客户一个线程

为每个客户申请分配一个线程以取代为每个客户申请分配一个进程，每次申请时候有线程开销。案例库封装了派生子线程的函数preads\_create()函数，用于派生子线程。在主函数中 \*thread\_func()是线程负责的业务。

-------------------------------------------------------------- -----server/serv04/main.c

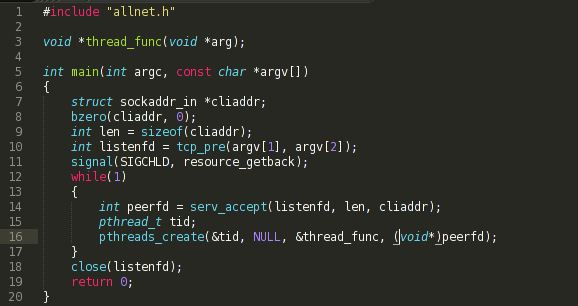


图4.1 main函数代码

-------------------------------------------------------server/serv04/pthreads\_create.c

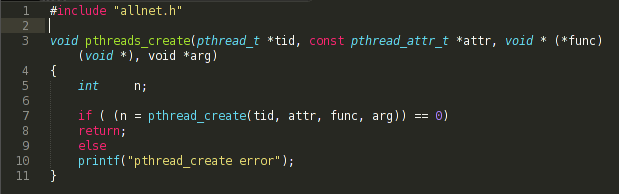


图4.2 pthread\_create()函数

-------------------------------------------------------server/serv04/pthread\_work.c

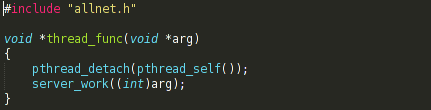
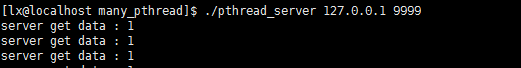


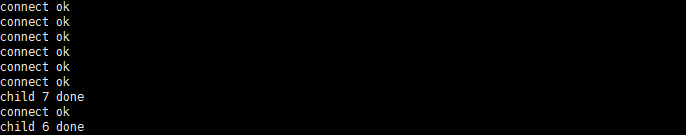
图4.3 pthread\_create()函数

运行截图如下：

服务器运行



客户读运行：



## 5.TCP预先派生子进程服务器，accept无上锁保护。

基本设计是预先创建一个进程池，创建一定数量的子进程，当客户端连接到来时这些子进程就立马为其服务。图5.1 给出该服务器的基本结构，对于该服务器模型，网络案例库libnet.so封装了poll\_resource\_back()函数用于处理僵死进程，19行代码是自己写的一个进程池构造函数（不包含在案例库中）。

-----------------------------------------------------------------server/serv05/main.c

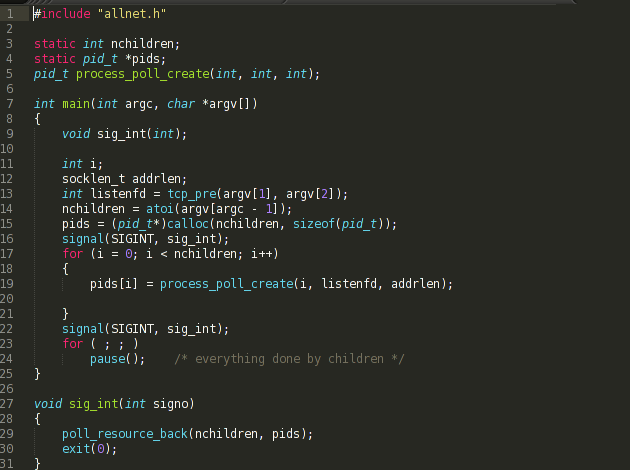


图5.1 服务器main函数

------------------------------------------------------server/serv05/processpoll\_work.c

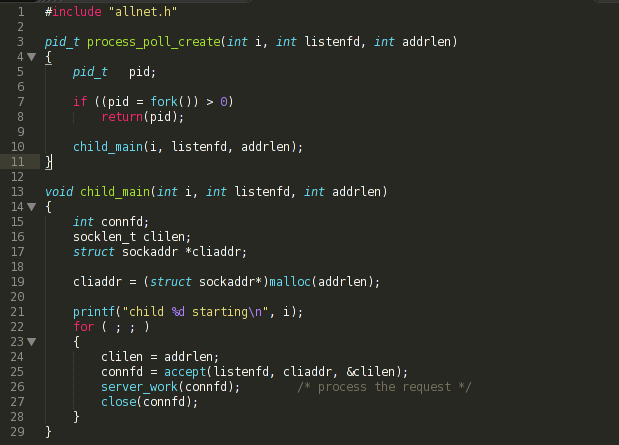
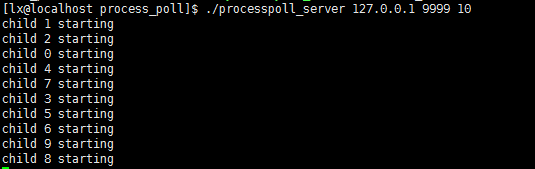


图5.2 进程池工作函数代码

运行截图如下：

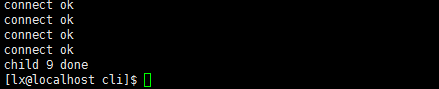
预派生10个子进程

服务器运行



客户读运行

微信截图_20170426154157.png



## 6. TCP预先派生子进程服务器程序，accept使用文件上锁保护

服务器进程在程序启动阶段派生N个子进程，当没有客户连接到来时候子进程处于睡眠状态。当某一时刻有一个客户连接到达时，所以子进程都将被唤醒。这些子进程是的都监听同一个socket结构，这就是经群现象。让应用进程在调用accept前后安置锁的形式，任意时刻只有一个子进程阻塞在accept调用中，其它子进程试图获取用于保护的accept的锁上。在该服务器模型中，我们案例库libnet.so中封装了meter()函数，上锁功能的my\_filelock\_init()函数，以及上锁和解锁文件的my\_lock\_wait()函数、my\_lock\_release()函数，用于查看全体客户连接在阻塞于accept调用中的可用子进程池上的分布。比起图5.1的main函数，在共享内存中分配一个长整数计数器数组，每个子进程一个计数器；在每个子进程accept返回后递增各自计数器，并在accept之前获取文件锁，在accept之后释放文件锁。增加代码如下

Long \*cptr, \*meter(int);

cptr = meter(nchildren);

my\_filelock\_init("/tmp/lock.XXXXXX");

下面是meter()函数代码

--------------------------------------------------------------------server/serv06/meter.c

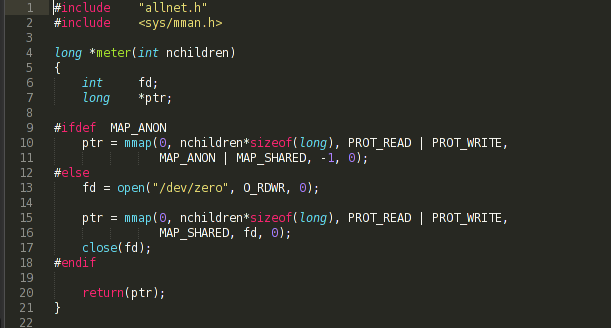


图6.1 meter（）函数代码

下图给出了上锁功能的my\_filelock\_init ()函数

----------------------------------------------------------------server/serv06/lock\_fcntl.c

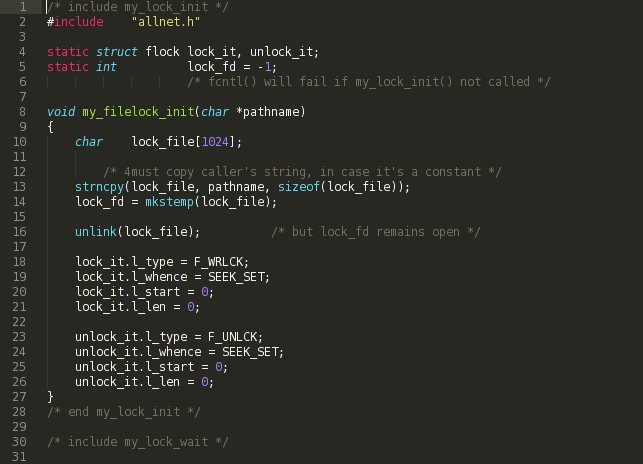


图6.2 my\_filelock\_init ()函数代码

下图给出了上锁功能的my\_lock\_wait()和my\_lock\_release()函数

----------------------------------------------------------------server/serv06/lock\_fcntl.c

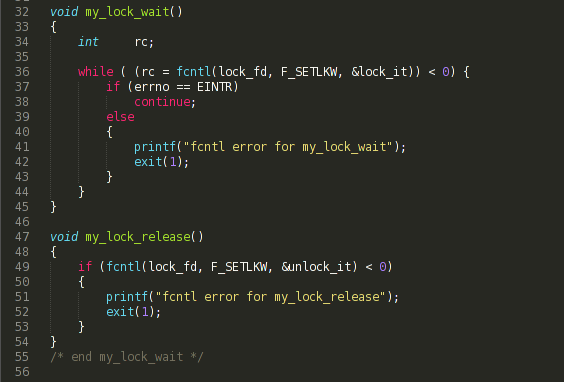
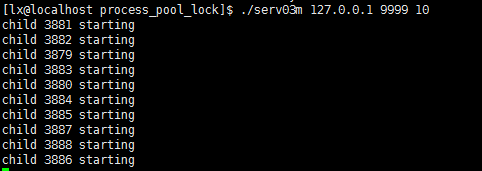


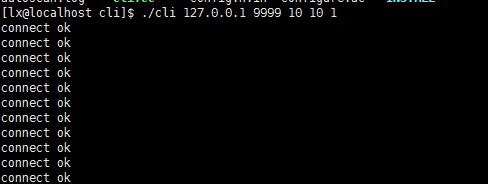
图6.3 文件上锁解锁函数代码

运行截图如下：

预派生10个子进程



服务器运行



## 7. TCP预先派生子进程服务器程序，accept使用线程锁上锁保护

用线程锁上锁保护accept比起用文件锁保护可以节约时间成本，不仅适用于同一进程内各线程之间上锁，而且适用于不同进程之间的上锁。我们的案例库libnet.so中封装了my\_pthreadlock\_init()函数初始化线程锁，以及pthread\_lock\_accep()函数，对accept用线程锁上锁，函数内部分别调用了线程锁上锁和解锁功能置于accept前后。

对于该服务器模型比较于accept使用文件上锁，在main函数中只需要将

my\_filelock\_init("/tmp/lock.XXXXXX");

改为

my\_pthreadlock\_init(NULL, mptr);

下图给出了上锁功能的my\_pthreadlock\_init ()函数

----------------------------------------------------------------server/serv07/lock\_fcntl.c

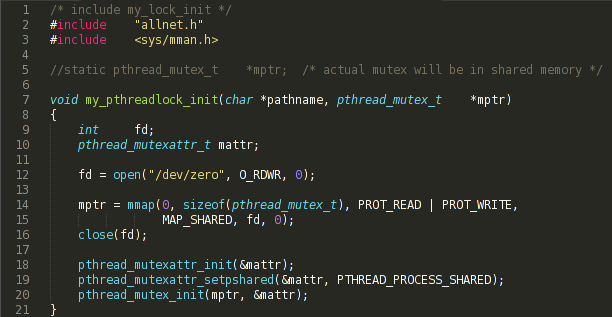


图7.1 my\_pthreadlock\_init ()函数

下图给出了线程锁上锁accept的代码，其中引入了头文件pthread\_polle.h，图7.3的第7行与第9行代码也可以引用网络案例库封装好的上锁函数。

int lock(pthread\_mutex\_t \*clifd\_mutex)

{

return pthread\_mutex\_lock(clifd\_mutex);

}

int unlock(pthread\_mutex\_t \*clifd\_mutex)

{

return pthread\_mutex\_unlock(clifd\_mutex);

}

----------------------------------------------------------server/serv07/pthread\_polle.h

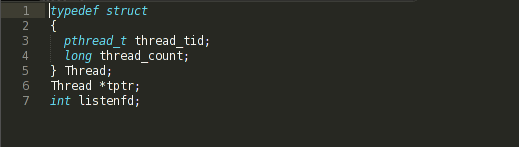


图7.2 pthread\_polle.h

-------------------------------------------------------server/serv07/pthreadpoll\_lock.c

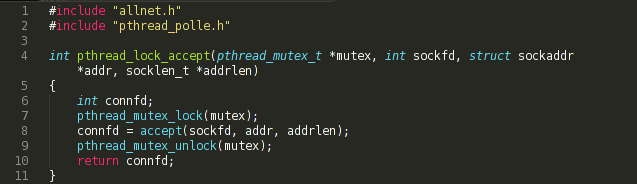
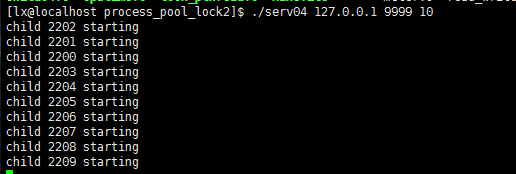


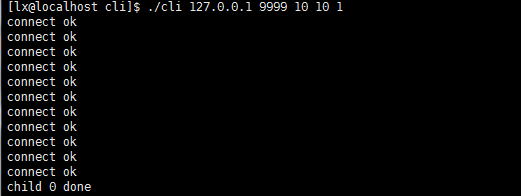
图7.3 pthread\_lock\_accept()

运行截图如下：

预派生10个子进程



客户运行：



## 8. TCP预先创建线程服务器，每个线程各自accept

本服务器的基本设计是预先创建一个线程池，并让每个线程各自调用accept。取代让每个线程都阻塞在accept调用之中的做法，我们改用互斥锁以保证任何时刻都有一个线程在 accept。对于该服务器模型我们的网络案例库libnet.so主要封装了pthread\_poll\_create()函数，该函数功能是创建线程池，函数的第二个参数就是线程池中派送子线程的数量。

图8.1给出了pthread\_polle.h头文件，定义了用于维护关于每个线程若干的信息的Thread结构。

-------------------------------------------------------server/serv08/pthread\_poll2.h

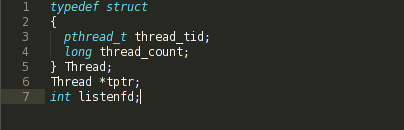


图8.1 pthread\_poll2.h

------------------------------------------------------- ------------server/serv08/main.c

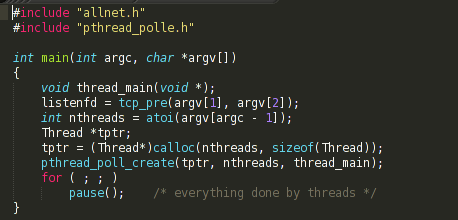
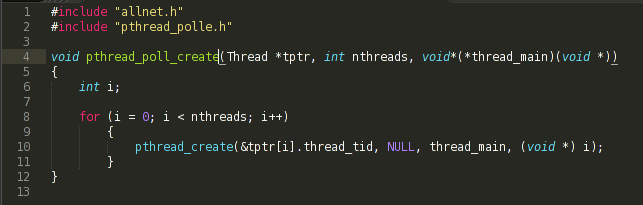


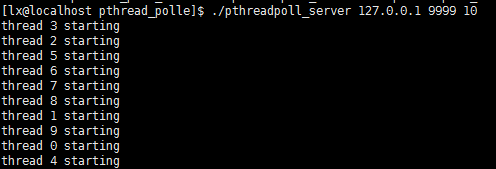
图8.1给出该服务器模型的main()函数

---------------------------------------------------server/serv08/pthread\_poll\_create.c

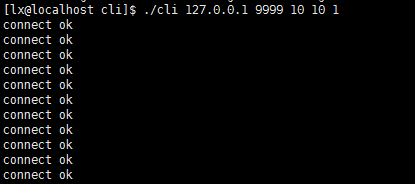


运行截图如下：

预派生10个子进程



客户端运行结果



## 9. TCP预先创建线程服务器程序，主线程统一accept

该服务器模型首先定义了一个头文件pthread\_pollm.h。我们还定义一个chifd数组，iput是往数组里存入的下一个元素下标，iget是从数组里取出的下一个元素的下标。主线程往数组中存入已经接收的已经连接套接字描述符，线程池中空闲线程从中取一个描述符服务客户端。主线程大部分时间阻塞再accept调用中，客户端连接到来时候要先获得互斥锁，将套接字描述符存入数组，并发送信号量，最后释放互斥锁。

---------------------------------------------------------------server/serv09/pthreadm.h

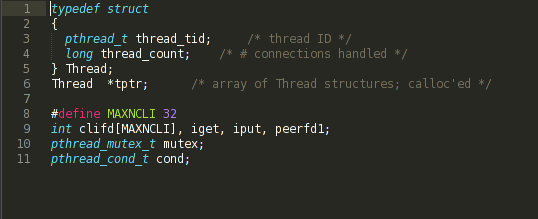


图9.1 pthreadm.h

--------------------------------------------------------------------server/serv09/main.c

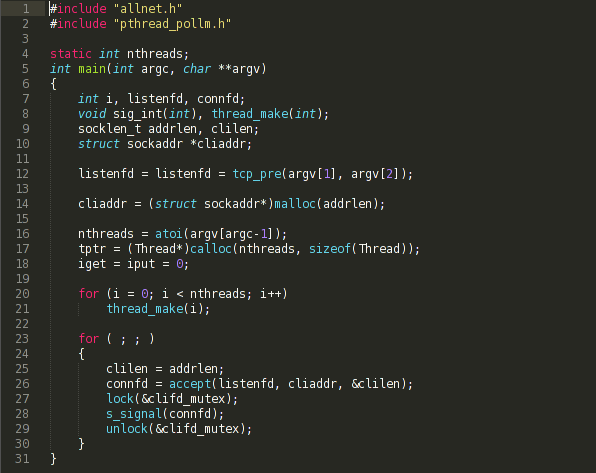


图9.2 该服务器模型的main()函数

线程池中每个线程都争夺chilfd数组得互斥锁。每一个线程都调用Pthread\_cond\_wait睡眠在clifd\_cond条件变量上。若iput与iget不等则从chilfd数组中取出一个元素以获得一个连接。主线程调用pthread\_cond\_signal在所有可能线程中轮询唤醒其中一个。

--------------------------------------------------------------server/serv09/lock\_signal.c

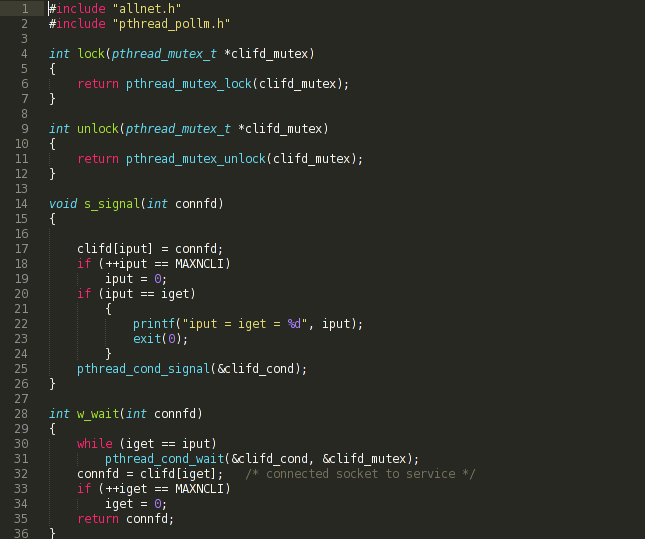


图9.3 lock\_singal.c

## 10．I/O复用技术TCP服务器——select模型

select模型原理利用select函数，判断套接字上是否存在数据，或者能否向一个套接字写入数据；select函数允许向内核注册感兴趣的描述符（读、写或异常）以及等待时间。

当select被调用时，进程就会受阻于select调用，不是阻塞在真正的I/O系统调用上，当有准备好的描述符就绪，就开始接收数据。该函数允许进程指示内核等待多个事件中的任何一个发生，并只在有一个或多个事件发生或经历制定的时间后才唤醒它。对于该服务器模型，我们的网路案例库libnet.so封装了如下一些函数，select\_pre()、select\_do\_wait()、select\_handle\_accept()、select\_add\_fd()、select\_del\_fd()、select\_close()函数。

下图是该服务器模型实例的main()函数，代码第10行是对一些变量的初始化，12到16行则是调用select函数，接受客户端数据，执行业务逻辑。具体的讲解在本文下面

---------------------------------------------------------------------server/serv10/main.c



图10.1 select模型main()函数

下图是之前提到的网络案例库libnet.so中封装的函数代码，代码5到8行是描述符的初始化，紧接着把client[]数字填充-1，表示没有装载描述符。select\_do\_wait()函数则是对select函数的调用，轮询select的描述符集，如果返回值为-1则说明没有描述符就绪，就阻塞在这个地方。

select\_handle\_accept()是接受客户端连接，监听到有客户端请求就接受连接。select\_add\_fd()函数则是添加描述字，将接受到的客户端描述符添加打client[]数组中，并添加到描述符集all\_set中。select\_del\_fd()则是从select的描述符集中清除内核感兴趣的描述符。

------------------------------------------------------------server/serv10/select\_handle.c

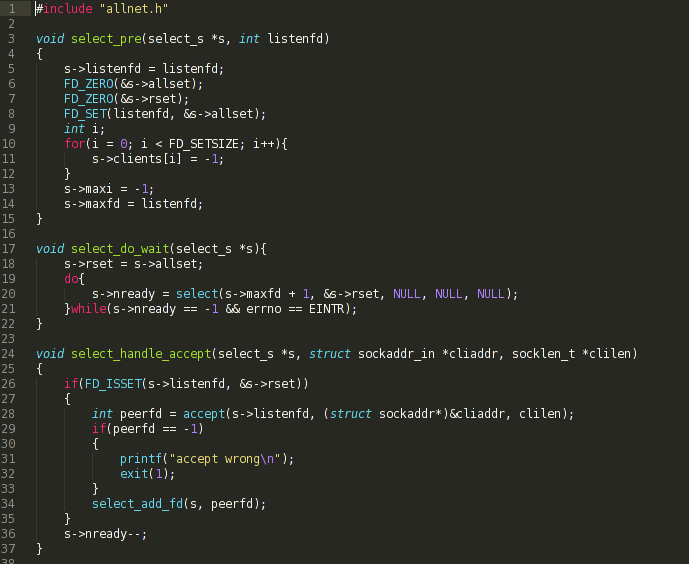


图10.2 select\_handle.c中的封装的库函数（1）

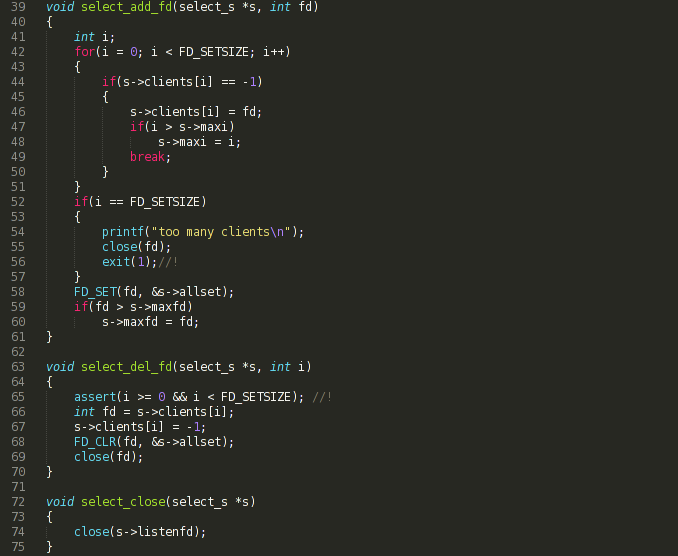
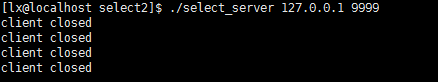
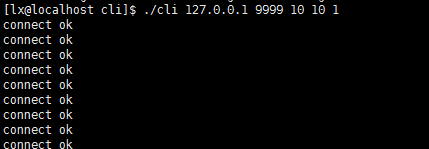


图10.2 select\_handle.c中的封装的库函数（2）

运行截图如下：



客户端运行：



## 11．I/O复用技术TCP服务器——poll模型

poll的机制与select类似，与select在本质上没有多大差别，管理多个描述符也是进行轮询，根据描述符的状态进行处理，但是poll没有最大文件描述符数量的限制。poll和select同样存在一个缺点就是，包含大量文件描述符的数组被整体复制于用户态和内核的地址空间之间，而不论这些文件描述符是否就绪，它的开销随着文件描述符数量的增加而线性增大。我们的网路案例库libnet.so封装了如下一些函数，poll\_pre()、poll\_do\_wait()、poll\_handle\_accept()、poll\_add\_fd()、poll\_del\_fd()、poll\_close()函数。该服务器模型实例的执行与select模型服务器相似，代码如下图

---------------------------------------------------------------------server/serv11/main.c

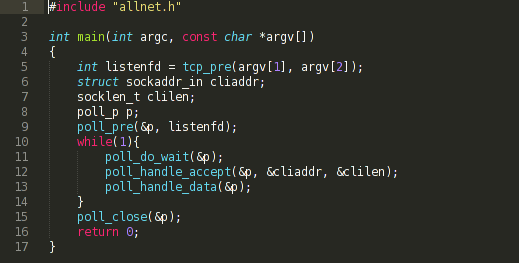
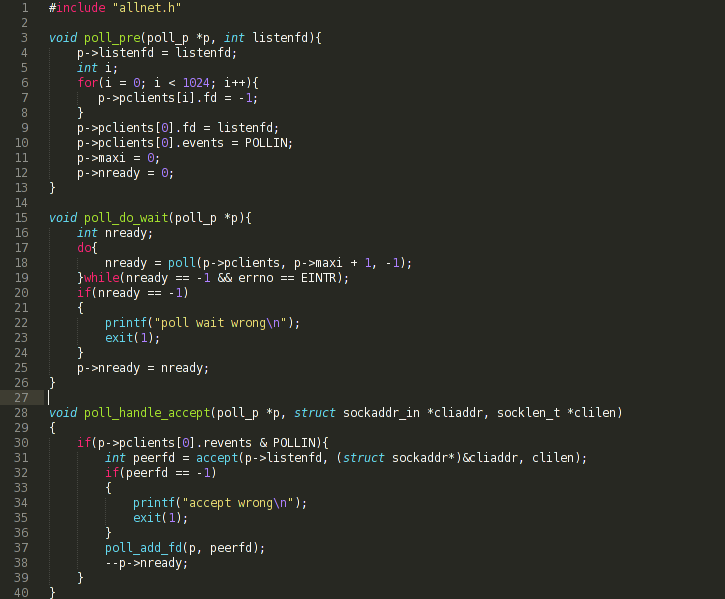


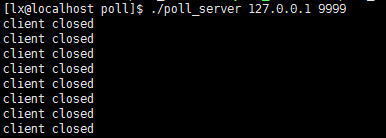
图 11.1 poll模型main()函数

poll\_handle.c文件中是网络案例库libnet.so封装的poll服务器模型的操作，比起之前的select模型在变量初始化阶段有一些不同需要注意。在使用select中我们必须分配一个client数组意见以及一个描述符集合。在poll中我们只需要分配一个pcclients结构的数组来维护客户信息。代码第9行我们把数组第一项用于监听套接字，其余成员都是-1.我们还给第一项设置POLLIN事件。

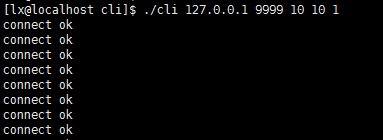
-------------------------------------------------------------server/serv11/poll\_handle.c

 图11.2 poll\_handle.c中的封装的库函数（1）

运行截图如下：



客户读运行：



## 12．I/O复用技术TCP服务器——epoll模型

epoll的实现机制与select和poll有所不同，使用select/poll机制，服务器进程每次都把客户读连接告诉操作系统(从用户态复制句柄数据结构到内核态)，让操作系统内核去查询这些套接字上是否有事件发生，轮询完后，再将句柄数据复制到用户态，让服务器应用程序轮询处理已发生的网络事件，如果客户端连接有很多得话，这一过程资源消耗非常打，因此，select/poll一般只能处理几千的并发连接。epoll通过在Linux内核中申请一个简易的文件系统，在epoll服务器中首先要建立一个epoll对象，然后向epoll对象中添加套接字，最后收集epoll\_wait收集发生的事件连接。这样就不需要轮询所有套接字，这样epoll只对活跃的套接字做操作，也就是所谓的触发机制。对于epoll模型我们的网络案例库libnet.so 封装了如下函数：epoll\_pre()、epoll\_do\_wait()、epoll\_handle\_accept()、epoll\_add\_fd()、epoll\_del\_fd()、epoll\_close()。该模型服务器实例主函数与之前的select、poll相似，代码如下图

---------------------------------------------------------------------server/serv12/main.c

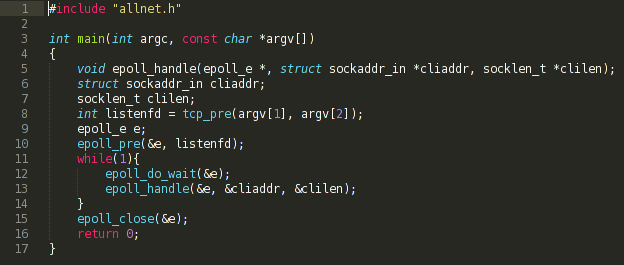


图12.1 epoll服务器main函数

在epoll\_handle.c中，代码的第5行创建了一个epoll句柄，在代码的11行添加了感兴趣的事件”listenfd”， epoll\_add\_fd()、epoll\_del\_fd()则是添加删除感兴趣事件的函数，都是将epoll\_ctl()系统调用封装在函数中。epoll\_do\_wait()函数功能是收集在epoll监控中已经发生的事件，封装了epoll\_wait()系统调用。以上操作的封装对于快速构造一个简单的回射服务器足够使用。

------------------------------------------------------------server/serv12/epoll\_handle.c



图12.2 poll\_handle.c中的封装的库函数（1）

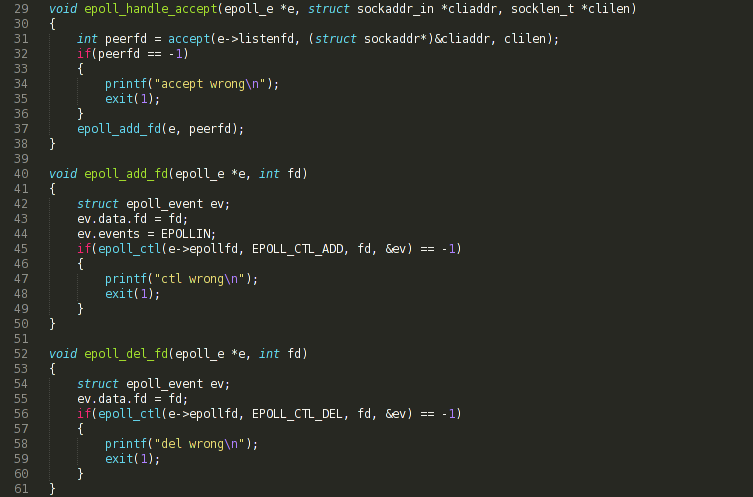
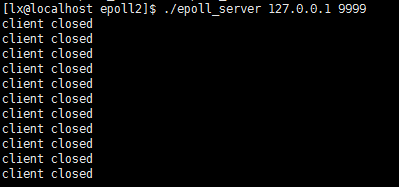


图12.3 poll\_handle.c中的封装的库函数（2）

运行截图如下：



客户端运行：

