TCP和UDP

整理自群内鹏佬的实验笔记

UDP

- UDP存在数据边界
  与TCP不一样，UDP协议中，调用IO函数的次数很重要，**输入函数的调用次数和输出函数的调用次数完全一致**，才能保证接收完全。
  其实，UDP套接字传输的数据包又称为数据报，它本身可以成为一个完整数据。（一次性发出）
- 可靠性方面来说
  TCP的确比UDP好，但是UDP的结构比TCP简洁，不会发送类似ACK应答消息，也不会有SEQ序号，性能有时比TCP高出很多。同时区分TCP和UDP的重要标志是流控制：TCP的生命在于流控制。
- 在UDP中，使用sendto数据传输过程分三阶段：
  a. 向UDP套接字注册目标IP和端口号；
  b. 传输数据；
  c. 删除UDP套接字中注册的目标地址信息。
- unconnected UDP套接字：
  每次调用sendto重复1的三个步骤，每次都变更目标地址，因此可以重复利用同一UDP向不同目标传递数据。

> 注册了目标地址的UDP套接字是connected UDP，默认情况下是unconnected的。需要与同一主机进行长时间通信时，将套接字变成connected会提高效率。

> Connected UDP套接字不仅可以使用sendto和recvfrom函数，甚至还可以使用write和read函数。

- connected套接字

```
connect(sock,(struct sockaddr*)&serv_adr,sizeof(serv_adr));
  //后面可以直接使用read和write函数进行操作
```

TCP

- time-wait
  先断开连接FIN的主机有个时间间隙，在该间隙中端口号被占用，bind时会出错。
  在time-wait状态可以继续接收主机B的终止信息，假如没有这个机制，A发送ACK时终止，但是ACK丢失了，则B永远无法接收A的ACK。
  系统SOL_SOCKET中的SO_REUSEADDR，默认是FALSE，即time-wait内端口不能重复使用。将值改成TRUE，即可重新使用。

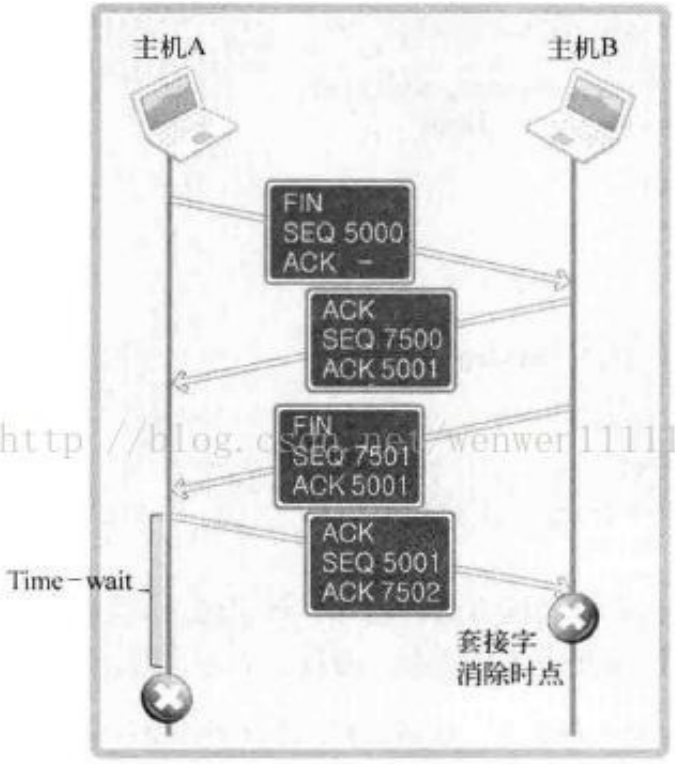


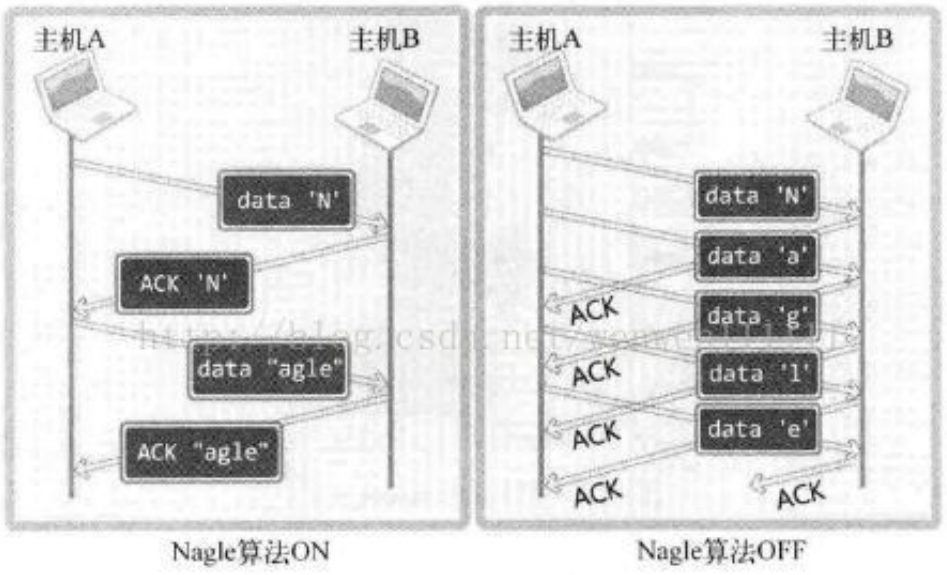
图9-1　Time-wait状态下的套接字

```
optlen = sizeof(option);
option = TRUE;
```

```
setsockopt(serv_sock,SOL_SOCKET,SO_RESUREADDR,(void
*)&option,optlen);
```

- Nagle算法
  1984年诞生，应用于TCP层，只有当受到上一个数据包的ACK后，Nagle算法才会发送下一数据包。
  a. TCP套接字默认使用Nagle算法，最大限度缓冲数据，直到收到ACK。
  b. 对应的，不使用Nagle算法将对网络流量Traffic产生负面影响，影响效率。
  c. 并不是任何情况都适用Nagle算法，在网络流量未受到太大影响时，不使用Nagle算法传输速度会更快，比如传输大文件数据时。

  > nagle相当于阻塞



Nagle算法ON　　Nagle算法OFF

图9-3　Nagle算法

```
int opt_val = 1;//禁用Nagle算法
setsockopt(sock,IPPROTO_TCP,TCP_NODELAY,(void
*)&opt_val,sizeof(opt_val));
```

I/O复用和多进程

- IO复用模型与多进程模型对比：
  a. 多进程模型：需要大量的运算和内存空间，进程间数据交换复杂IPC。
  b. IO复用模型可以不创建多进程同时又向多个客户端提供服务，但是并不适用于所有情况。
- select函数，将多个文件描述符集中在一起统一监视。

监视项目有：
是否存在套接字接收数据；
无需阻塞传输数据的套接字有哪些；
哪些套接字发生异常；
三个集合：{可读、可写和异常}

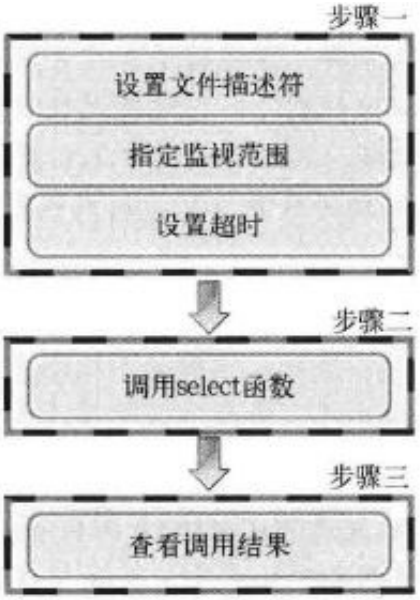


图12-5　select函数调用过程

- IO流分离方式（两种）：
  a. 第十章TCP IO routine分离，通过fork文件描述符区分输入输出，虽然文件描述符不会根据输入、输出进行区分，但是分开了两个文件描述符的用途（父进程负责读，子进程负责写）；
  b. 第十五章 调用fdopen创建FILE指针，分离输入工具和输出工具。
- 分离流的好处：
  //第十章
  a. 通过分开输入过程和输出过程降低实现难度；
  b. 与输入无关的输出操作可以提高速度；
  //第十五章
  a. 将FILE指针按读模式、写模式加以区分；
  b. 可以通过区分读写模式降低难度；
  c. 通过区分IO缓冲提高缓冲性能；
  d. 将文件描述符转成文件指针后，可以使用标准IO函数

> 原来可以把文件指针和socket连接起来，通过文件的读写指针可以实现发送和接收缓存数据

边缘触发和条件触发

- 两者区别：在于发生事件的时间点
    - 条件触发：只要输入缓冲有数据就一直通知该事件；
    - 边缘触发：输入缓冲受到数据时仅注册一次事件。
      epoll默认以条件触发方式工作，select也是以条件触发模式工作的。
- 条件触发与边缘触发比较：应该从服务器端实现模型角度考虑。
  边缘触发能够做到接收数据与处理数据的时间点分离。

epoll

实现IO复用的传统方法select和poll，但是性能不满意，因此有Linux的epoll，BSD的kqueue，Solaris的/dev/poll,Windows的IOCP。

- select不适合以web服务器端开发为主流的现代开发环境。
  a. 调用select后常见的针对所有文件描述符的循环语句；
  b. 每次调用select函数时都需要向函数传递监视对象信息。 —— 致命弱点
- select的优点：
  a. 程序具有兼容性；
  b. 服务器端接入者少。
- epoll可以克服select的缺点。
  epoll服务器端用到的三个函数
  epoll_create：创建保存epoll文件描述符的空间 //对应 fd_set
  epoll_ctl：向空间（位数组）注册并注销文件描述符 //对应FD_SET，FD_CLR
  epoll_wait：等待文件描述符发生变化 //对应select

```
 //epoll将发生事件的文件描述符集中在一起，放在epoll_event结构体中
struct epoll_event{
 __uint32_t events;
 epoll_data_t data;
}

typedef union epoll_data{
 void * ptr;
 int fd;
 __uint32_t u32;
 __uint64_t u64;
}epoll_data_t;
```

1. epoll_create函数

   调用epoll_create函数时创建的文件描述符保存空间称为“epoll例程”，size只是建议epoll例程大小，实际大小由操作系统决定。

   Linux2.6.8之后，内核忽略size参数，会根据情况自动调整。

   epoll_create函数创建的资源与套接字相同，由操作系统管理，终止时要close。

```
#include <sys/epoll.h>
int epoll_create(int size)
//成功时返回epoll文件描述符，失败返回-1
//size epoll实例的大小
```

1. epoll_ctl函数

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)
//成功时返回0，失败-1
epfd：用于注册监视对象的epoll例程（事件发生的监视范围）的文件描述符
op：用于指定监视对象的添加、删除或更
改//EPOLL_CTL_ADD,EPOLL_CTL_DEL,EPOLL_CTL_MOD
fd：需要注册的监视对象文件描述符
event：监视对象事件类型

struct epoll_event event;
event.events = EPOLLIN; //
event.data.fd = sockfd;
epoll_ctl(epfd,EPOLL_CTL_ADD,sockfd,&event);

//events 成员：
EPOLLIN：需要读取数据的情况
EPOLLOUT：输出缓冲为空，可以立即发送数据的情况
EPOLLPRI：受到OOB数据情况
EPOLLRDHUP：断开连接或半关闭的情况
EPOLLERR：发生错误的情况；
EPOLLET：以边缘触发的方式得到事件通知
EPOLLONESHOT：发生一次事件后，相应的文件描述符不再受到事件通知
```

1. epoll_wait函数

```
```

```
int epoll_wait(int epfd,struct epoll_event * events,int maxevents,int timeout)
 //成功时返回事件的文件描述符数，失败-1

epfd：事件发生监视范围epoll例程的文件描述符
events：保存发生事件的文件描述符集合的结构体地址值
maxevents：第二个参数中可以保存的最大事件数
timeout：以1/1000秒为单位等待时间
```

多线程

- 多进程模型的缺点：
  a. 创建进程的过程需要大的开销；
  b. 进程间通信需要用到IPC技术；
  c. 经常发生“上下文切换Context Switching”，时间很长，很致命：进程A切换到进程B时，要将进程A信息移出内存，并读入进程B信息。
- 多线程（轻量级进程）的优点：
  a. 线程的创建和上下文切换比进程更快；
  b. 线程间通信无需特殊技术。
- 进程VS线程
  a. 每个进程都有自己的数据区、heap、stack，进程间相互独立。
  线程间共享数据区和heap（线程拥有自己的栈）。

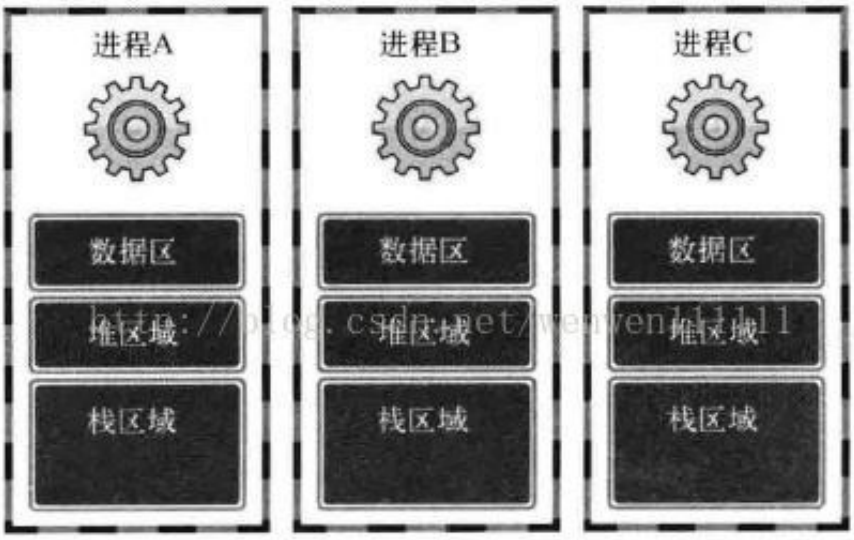


图18-1　进程间独立的内存

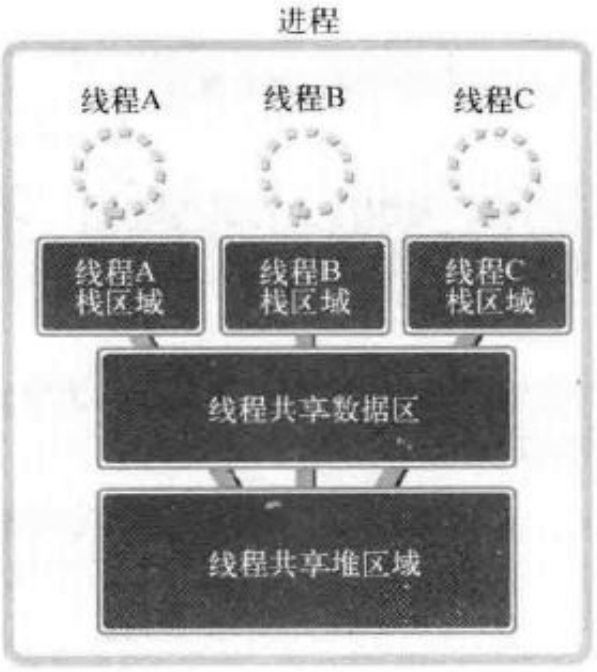


图18-2　线程的内存结构

b. 进程：在操作系统构成单独执行流的单位；
线程：在进程构成单独执行流的单位；

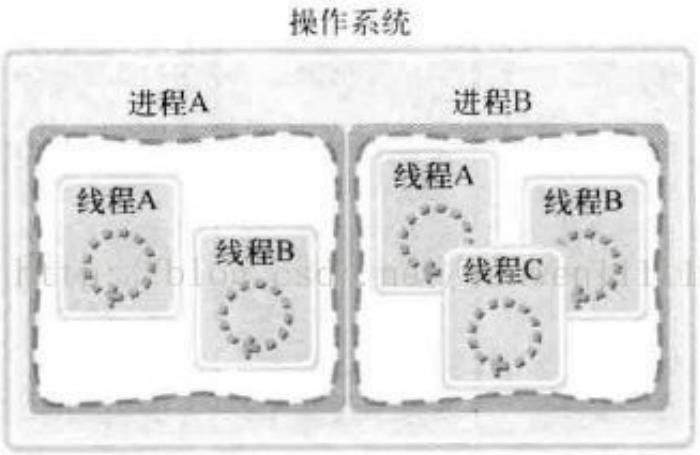


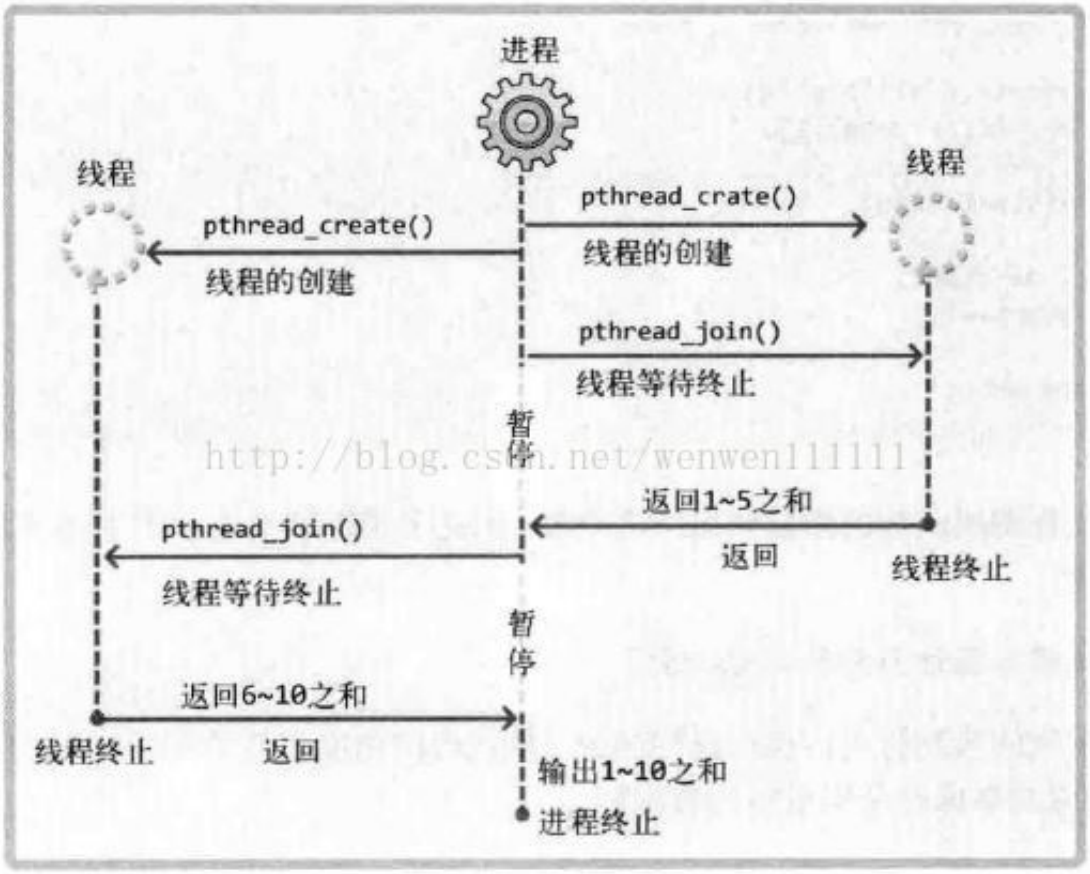
图18-3　操作系统、进程、线程之间的关系

- 可在临界区critical section内调用的函数
  临界区：多个线程同时调用函数时可能产生问题，这类函数内部存在临界区（共同访问的那块代码）。临界区中至少存在一条这类代码。
  线程安全函数：被多个线程同时调用时不会引发问题；线程安全函数的名称以_r后缀 //与临界区无关，安全函数也可能有临界区
  非线程安全函数：被同时调用时会引发问题。
  编译使用_r的函数方法：声明头文件前定义_REENTRANT，也可以

```
# gcc -D_REENTRANT mythread.c -o mthread -lpthread
```

- 工作worker线程模型

  代码只说模型应用，里面存在临界区问题。



- 线程同步

  a. 需要线程同步的情况：

  1）同时访问同一内存空间；

  2）需要指定访问同一内存空间的线程执行顺序。

  b. 同步技术：互斥量Mutex和信号量Semaphore

> 将临界区比喻成洗手间，线程同步理解成一把锁。为了保护个人隐私，进洗手间时锁上门，出来再打开；如果有人使用洗手间，其他人需要在外面等待；
> 等待的人数可能很多，这些人需要排队进入洗手间。

1. 互斥量Mutual Exclusion，创建和销毁函数

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t * mutex,const
pthread_mutexattr_t * attr);
int pthread_mutex_destroy(pthread_mutex_t * mutex);
//成功返回0，失败返回其他值
```

```
mutex：创建/销毁互斥量时传递保存互斥量的变量地址值
attr：创建的互斥量属性
```

创建互斥量时，如果第二个参数为NULL，也可以通过宏PTHREAD_MUTEX_INITIALIZER创建。
但是最好还是使用函数，宏比较难检查错误。

1. 互斥量锁住、释放临界区（lock和unlock是对临界区进行操作的）

```
int pthread_mutex_lock(pthread_mutex_t * mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
//成功返回0，失败返回其他值
```

注意：临界区锁住后，忘记解锁，则其他尝试进入临界区的线程将会“死锁”。

1. 信号量的创建、销毁

```
int sem_init(sem_t *sem,int pshared,unsigned int value);
int sem_destroy(sem_t * sem);
 //成功返回0，失败返回其他值
sem：创建信号量时传递保存信号量的变量地址值
pshared：传递其他值时，可创建多个进程共享的信号量。0时，只允许一个进程内部使用该信号量
value：指定信号量初始值。
```

1. post/wait

```
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
//成功返回0，失败时返回其他值
sem：传递保存信号量读取值得变量地址值
```

调用sem_init时，操作系统创建信号量对象，并赋初始值。
调用sem_post函数值，对象+1，sem_wait函数时-1.
信号量的值不能小于0，在信号量为0的情况，sem_wait函数会进入阻塞直到值大于0。