第七讲
守护进程与其他网络服务器编程技术

任立勇

电子科技大学计算机学院

目　录

- 守护进程和inetd超级服务器
  - 概述
  - syslogd守护进程和syslog函数
  - 创建守护进程
  - inetd守护进程
- 几种服务器技术的比较；
  - tcp并发服务器－每个客户一个子进程；
  - tcp预先派生子进程服务器程序，accept无上锁保护；
  - tcp预先派生子进程服务器程序，accept使用文件锁保护；
  - tcp预先派生子进程服务器程序，accept使用线程互斥锁保护；
  - tcp预先派生子进程服务器程序，传递描述字；
  - tcp并发服务器程序，每个客户一个线程；
  - tcp预先创建线程服务器程序，每个线程各自accept;
  - tcp预先创建线程服务器程序，主线程统一accept;

守护进程概述

- 守护进程是在后台运行不受终端控制的进程（如输入、输出等），一般的网络服务都是以守护进程的方式运行。
- 守护进程脱离终端的主要原因有两点：
  - 用来启动守护进程的终端在启动守护进程之后，需要执行其他任务。（如其他用户登录该终端后，以前的守护进程的错误信息不应出现）
  - 由终端上的一些键所产生的信号（如中断信号），不应对以前从该终端上启动的任何守护进程造成影响。
- 要注意守护进程与后台运行程序（即加&启动的程序）的区别。

启动守护进程的方法

- 在系统启动是由系统初始化脚本启动，这些脚本一般在/etc或/etc/rc开头的目录。如inet超级服务器，web服务器等；
- 许多网络服务器是由inet超级服务器启动的，如Telnetd、FTPd等；
- cron守护进程按一定的规则执行一些程序，由它启动的程序也以守护进程的方式运行。
- 守护进程可以在用户终端上启动，这是测试守护进程或重新启动守护进程常用的方法。

用户守护进程登记出错信息

- 创建一个Unix域数据报套接口，并向syslogd守护进程绑定的路径名发送我们的消息，我们就能从自己的守护进程向syslogd发送登记信息。
- 可以创建一个UDP套接口，将日志消息发到回馈地址及端口号514；
- 更简单的方法是利用syslog函数；

syslogd守护进程

- syslogd是一个系统守护进程，它主要负责接收系统或用户守护进程的输出消息，并根据配置信息作出相应处理。
- syslogd在启动时执行以下操作：
  - 读入配置文件，通常是/etc/syslogd.conf，它设定守护进程对接收的各种登记消息如何处理。这些消息可能被写入一个文件（一种特殊文件是/dev/console，这将把消息写到控制台上）。或发给指定的用户，或转发给另一台主机上的syslogd进程。
  - 创建一个Unix域套接字，给它捆绑路径名/var/run/log
  - 创建一个udp套接字，给它捆绑端口514
  - 打开路径名/dev/klog，内核中的所有出错消息作为这个设备的输入出现；
- 在此之后syslogd进程运行一个无限循环，循环中调用select,等待三个描述字（以上2、3、4创建）之一变为可读，并按配置文件对消息进行处理

openlog函数

```
void openlog(const char *ident, int option, int facility);
void closelog(void);
```

- openlog函数在第一次调用syslog函数之前调用，当不再需要发生登记消息时可调用closelog函数；
- ident是一个字符串，它将被加到每条登记消息前面；option参数由下页图中的值组合而成。facility参数为后面没有设置设施的syslog调用设置一个缺省值。

openlog的选项

选项（options）	描述
LOG_CONS	如果不能发往syslogd守护进程，则登记到控制台上
LOG_NDELAY	不延迟打开，立即创建套接口
LOG_PERROR	既发往syslogd守护进程，又登记到标准错误输出
LOG_PID	登记每条消息的进程ID

syslog函数

#include <syslog.h>

void syslog(int priority, const char * message,...);

- 参数message与printf所用的格式化字符串类似，同时增加了%m，它将由对应的当前errno值的出错消息所取代；
- 参数priority是级别（level）和设施（facility）的组合。设施和级别的目的是,允许在/etc/syslog.conf文件中进行配置，使得对相同设施的消息得到同样的处理，或使相同级别的消息得到同样的处理。

登记消息的级别

级别（level）	值	描述
LOG_EMERG	0	系统不可用（优先级最高）
LOG_ALERT	1	必须立即进行处理
LOG_CRIT	2	危险情况
LOG_ERR	3	出错情况
LOG_WARNING	4	警告性情况
LOG_NOTICE	5	常见但值得注意的情况(缺省)
LOG_INFO	6	通告消息
LOG_DEBUG	7	调试消息（优先级最低）

登记消息的设施

设施（facility）	描述	设施（facility）	描述
LOG_AUTH	安全／授权消息	LOG_LOCAL6	本地使用
LOG_AUTHPRIV	安全／授权消息（私有）	LOG_LOCAL7	本地使用
LOG_CRON	cron守护进程	LOG_LPR	行式打印机系统
LOG_DAEMON	系统守护进程	LOG_MAIL	邮件系统
LOG_FTP	FTP守护进程	LOG_NEWS	网络新闻系统
LOG_KERN	内核消息	LOG_SYSLOG	由syslogd内部消息
LOG_LOCAL0	本地使用	LOG_USER	任意的用户消息(缺省
LOG_LOCAL1	本地使用	LOG_UUCP	UUCP消息
LOG_LOCAL2	本地使用		
LOG_LOCAL3	本地使用		
LOG_LOCAL4	本地使用		
LOG_LOCAL5	本地使用		

syslog函数的例子

- 当调用rename函数失败时，守护进程可能会调用：

```
if (rename(file1, file2) == -1)
    syslog(LOG_INFO|LOG_LOCAL2,"rename(%s,%s):%m",file1,file2);
```

- 如果配置文件中有以下两行：

```
kern *                  /dev/console
local2.debug            /var/log/cisco.log
local2.info             /var/log/info.log
```

则指定内核的所有消息登记到控制台上，所有设施为local2的调试消息将添加到/var/log/cisco文件的末尾；

setsid()函数

```
#include <sys/types.h>
#include <unistd.h>
pid_t setsid(void);
```

返回值：若成功则为进程组ID,出错则为-1

- 该函数是实现守护进程必须调用和非常重要的函数。
- 如果调用进程不是一个进程组的组长，则此函数创建一个新的会话：
  - 此进程变成该会话的首进程，同时是该会话的唯一进程；
  - 此进程成为一个新进程组的组长进程。新进程组ID是此调用进程的进程ID；
  - 此进程没有控制终端，如果在调用setsid之前此进程有一个控制终端，那么这种联系也被解除。
- 如果调用进程已经是一个进程组长，则函数出错。为了保证不处于这种情况，通常首先调用fork，然后使父进程终止。

创建守护进程

```
#include      <syslog.h>
#define       MAXFD       64

extern int    daemon_proc;    /* defined in error.c */

void daemon_init(const char *pname, int facility)
{
    int       i;
    pid_t     pid;

    if ( (pid = Fork()) != 0)
        exit(0);                    /* parent terminates */

    /* 1st child continues */
    setsid();                       /* become session leader */
```

调试目的

创建守护进程（续）

```
    Signal(SIGHUP, SIG_IGN);
    if ( (pid = Fork()) != 0)
            exit(0);                        /* 1st child terminates */

    /* 2nd child continues */
    daemon_proc = 1;                        /* for our err_XXX() functions */

    chdir("/");                             /* change working directory */

    umask(0);                               /* clear our file mode creation mask */

    for (i = 0; i < MAXFD; i++)
            close(i);

    openlog(pname, LOG_PID, facility);
}
```

程序说明

- 第一次调用fork的目的是保证调用setsid的调用进程不是进程组长。（而setsid函数是实现与控制终端脱离的唯一方法）；
- setsid函数使进程成为新会话的会话头和进程组长，并与控制终端断开连接；
- 第二次调用fork的目的是：即使守护进程将来打开一个终端设备，也不会自动获得控制终端。（因为在SVR4中，当没有控制终端的会话头进程打开终端设备时，如果这个终端不是其他会话的控制终端，该终端将自动成为这个会话的控制终端），这样可以保证这次生成的进程不再是一个会话头。
- 忽略SIGHUP信号的原因是，当第一次生成的子进程（会话头）终止时，该会话中的所有进程（第二次生成的子进程）都会收到该信号；

守护进程方式运行的时间服务器

```
#include        <time.h>

int main(int argc, char **argv)
{
    int                 listenfd, connfd;
    socklen_t           addrlen, len;
    struct sockaddr     *cliaddr;
    char                buff[MAXLINE];
    time_t              ticks;

    daemon_init(argv[0], 0);

    if (argc == 2)
        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
    else if (argc == 3)
        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
```

时间服务器

```
    else
        err_quit("usage: daytimetcpsrv2 [ <host> ] <service or port>");
    cliaddr = Malloc(addrlen);

    for ( ; ; ) {
        len = addrlen;
        connfd = Accept(listenfd, cliaddr, &len);
        err_msg("connection from %s", Sock_ntop(cliaddr, len));

        ticks = time(NULL);
        snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
        Write(connfd, buff, strlen(buff));

        Close(connfd);
    }
}
```

编写守护进程的注意事项

- 当程序开始时，尽快调用daemon_init，使之变成守护进程，否则容易受控制终端影响；
- 守护进程必须避免调用printf和fprintf函数，而调用syslog函数。

配置守护进程

- 由于守护进程没有控制终端，因而无法通过用户输入来进行相应配置，在编程时通常采用以下几种方法来配置守护进程：
  - 配置文件：将所有的配置参数存入一个配置文件，当守护进程启动时可以自动读取配置信息进行配置；
  - 环境变量：守护进程通过读取环境变量而获得配置信息
  - 同样，因为守护进程运行时没有控制终端，所以它不会收到来自内核的SIGHUP信号，因此很多守护进程将该信号作为管理员通知其配置文件已修改之用。守护进程收到该信号后应重新读入配置文件。另外两个守护进程不应收到的信号是SIGINT和SIGWINCH，这些信号也可以作为通知用。

inetd：超级网络服务器

- 4.3BSD以前的版本中的每个网络服务有一个与之对应的进程，它们的启动几乎完全一样（如创建套接字，绑定地址，监听端口...），这种模型存在以下问题：
  - 这些守护进程有几乎相同的代码，首先是创建套接字，还要考虑变成守护进程；
  - 每个守护进程在进程表中要占一项，但它们在大多数时间处于睡眠状态；

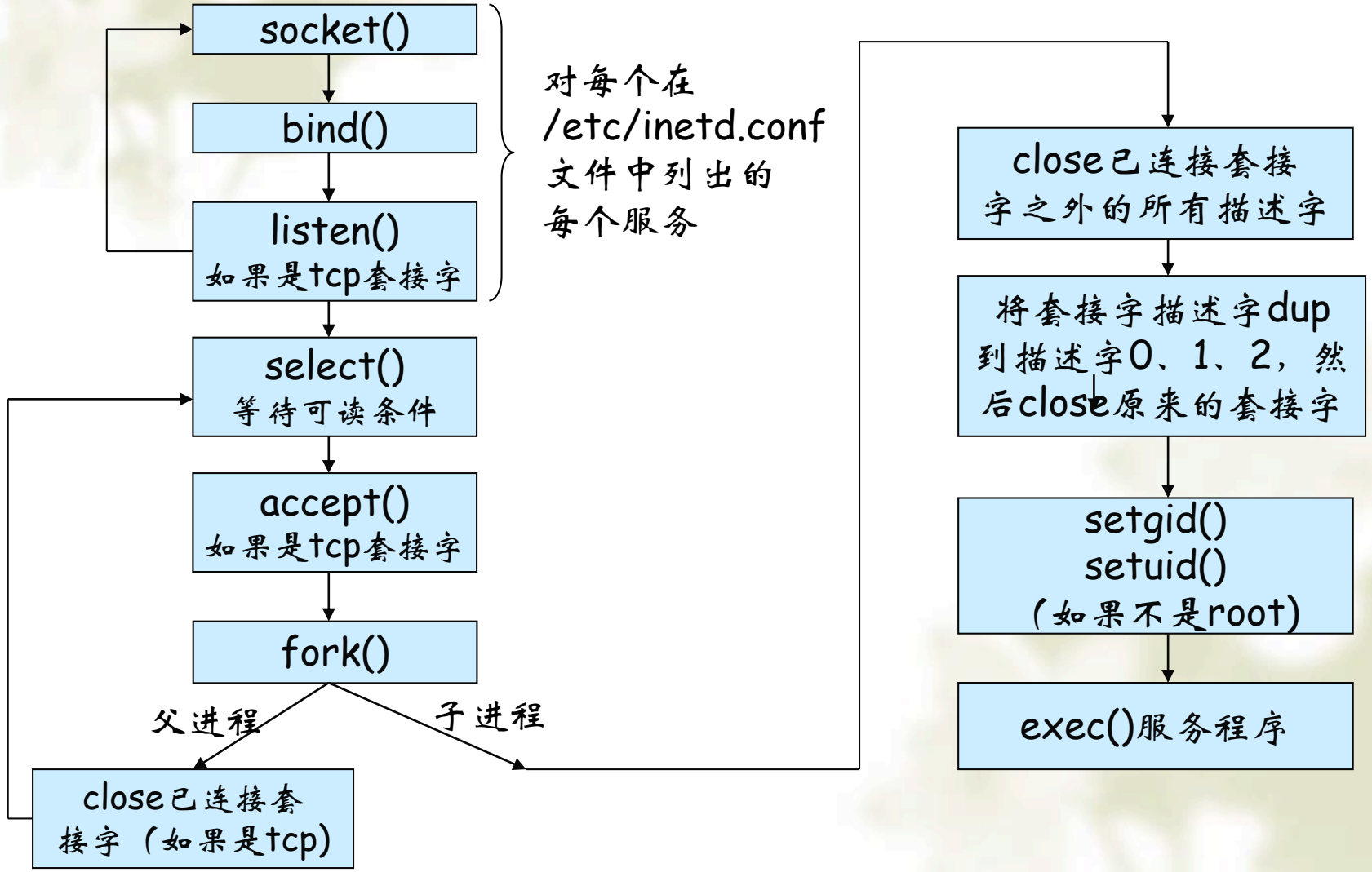
inetd：超级网络服务器（续）

- 4.3BSD通过一个网络超级服务器inet守护进程简化了上述问题：
  - 大部分启动时要做的工作由inetd处理，所有守护进程的编写得到简化。这避免了每个服务器程序都要调用daemon_init函数；
  - 单个进程（inetd）能为多个服务等待客户的请求，取代了每个服务一个进程的方式，这样减少了系统中的进程数；

inetd守护进程的工作流程

1. 启动时读取/etc/inetd.conf文件并给文件中指定的所有服务创建一个相应类型的套接字，inetd能处理的服务器数目依赖于它最多能创建的描述字的数目。每个创建的套接字都被加入到select调用的描述字集中；
2. 为每个套接字调用bind，给它们捆绑服务器的众所周知端口和通配地址。
3. 对tcp套接字调用listen,以接受外来的连接请求；
4. 所有套接字建立后，调用select等待这些套接字变为可读；
5. Select返回一个可读的套接字后，如果是一个tcp套接字，就调用accept接受这个新的连接；
6. inetd守护进程fork，由子进程处理服务请求。子进程关闭除连接套接字以外的所有描述字；
7. 如果是tcp套接字，父进程必须关闭连接套接字。

inetd守护进程的工作流程



inetd的配置文件

```
ftp     stream    tcp    nowait   root    /usr/bin/ftpd       ftpd -l
telnet  stream    tcp    nowait   root    /usr/bin/telnetd    telnetd
login   stream    tcp    nowait   root    /usr/bin/rlogind    rlogind -s
tftp    stream    udp    wait     root    /usr/bin/tftpd      tftpd -s
    /tftpboot
```

inetd的等待方式

- 对tcp服务器设置nowait方式，意味着inetd在接受请求同一服务的其他连接之前不需要等待该服务的子进程终止。
- 给数据报服务设置wait标志，需要对父进程的操作步骤作一定的修改。即inetd在该UDP套接字上再次选择之前，必须等待在该套接字上服务的子进程终止：
  - 父进程中的fork返回时，记录子进程的进程号，以便父进程可以用waitpid等待它终止；
  - 父进程用FD_CLR宏关闭select使用的描述字集中与该这个套接字相关的位。
  - 当子进程终止时，父进程收到一个SIGCHLD信号，父进程的信号处理程序得到终止子进程的进程号，父进程通过打开描述字集中相应的位恢复对该套接字的select。
- 父进程需要等待数据报服务的子进程终止是因为：一个数据报服务只有一个套接字，如果不关闭相应的位，则随后到来的数据将会使select返回可读条件，从而fork一个错误的子进程。

常见的网络服务器模型

- 最简单的迭代服务器模型
- 并发服务器（服务器为每个客户创建一个新进程）
- 并发服务器（服务器为每个客户创建一个新线程）
- 在一个进程内select多个客户

本讲将介绍两种新的并发服务器程序设计方法

- 预先派生子进程
- 预先派生线程

存在的问题

- 对于预先创建进程或线程的服务器技术，客户程序一般不作特殊处理，因为很少有进程控制问题。
- 但对服务器而言，则存在许多新的问题，如：如果池中的进程或线程不够怎么办？过多又怎么办？父子进程、父子线程，以及进程之间、线程之间如何同步？

试验说明

- 我们针对每个服务器运行同一客户程序的多个实例，测量服务固定数目的客户请求所需的CPU时间，而迭代服务器是我们的基准，我们把它从实际的CPU时间中减去就得到用于进程控制那部分CPU时间，因为迭代服务器没有进程控制开销。
- 所有数据都是在与服务器主机处于同一子网的两台不同主机上运行同一客户程序。每个客户派生5个子进程，对服务器开5个连接，因此服务器在任意时刻最多有10个连接。每个客户请求从服务器返回4000个字节的数据量。预先派生子进程或线程个数为15个。

各种类型的服务器的耗时比较

编号	服务器描述	进程控制CPU时间（与基准之差）		
		solaris	DUinx	BSD/OS
0	迭代服务器（测量基准，无进程控制）	0.0	0.0	0.0
1	简单并发服务器，为每个客户请求fork一个进程	504.2	168.9	29.6
2	预先派生子进程，每个子进程调用accept		6.2	1.8
3	预先派生子进程，用文件上锁方式保护accept	25.2	10.0	2.7
4	预先派生子进程，用线程互斥锁保护accept	21,5		
5	预先派生子进程，由父进程向子进程传递套接字描述字	36.7	10.9	6.1
6	并发服务器，为每个客户请求创建一个线程	18.7	4.7	
7	预先派生子线程，用互斥锁上锁方式保护accept	8.6	3.5	
8	预先派生子线程，由主线程调用accept	14.5	5.0	

TCP测试用客户程序

```
#define MAXN   16384            /* max #bytes to request from server */
int main(int argc, char **argv)
{
    int             i, j, fd, nchildren, nloops, nbytes;
    pid_t           pid;
    ssize_t         n;
    char            request[MAXLINE], reply[MAXN];
    if (argc != 6)
        err_quit("usage: client <hostname or IPaddr> <port> <#children> "
                 "<#loops/child> <#bytes/request>");
    nchildren = atoi(argv[3]);
    nloops = atoi(argv[4]);
    nbytes = atoi(argv[5]);
    snprintf(request, sizeof(request), "%d\n", nbytes); /* newline at end */
    for (i = 0; i < nchildren; i++) {
        if ( (pid = Fork()) == 0) {             /* child */
```

TCP测试用客户程序（续）

```
                for (j = 0; j < nloops; j++) {
                        fd = Tcp_connect(argv[1], argv[2]);
                        Write(fd, request, strlen(request));
                        if ( (n = Readn(fd, reply, nbytes)) != nbytes)
                                err_quit("server returned %d bytes", n);
                        Close(fd);/* TIME_WAIT on client, not server */
                }
                printf("child %d done\n", i);
                exit(0);
        }
        /* parent loops around to fork() again */
    }
    while (wait(NULL) > 0) /* now parent waits for all children */       ;
    if (errno != ECHILD)
        err_sys("wait error");
    exit(0);
}
```

TCP并发服务器：每个客户一个子进程

```
int main(int argc, char **argv) {
    int                 listenfd, connfd;
    pid_t               childpid;
    void                sig_chld(int), sig_int(int), web_child(int);
    socklen_t           clilen, addrlen;
    struct sockaddr     *cliaddr;
    if (argc == 2)
        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
    else if (argc == 3)
        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
    else
        err_quit("usage: serv01 [ <host> ] <port#>");
    cliaddr = Malloc(addrlen);
    Signal(SIGCHLD, sig_chld);
    Signal(SIGINT, sig_int);
```

TCP并发服务器：每个客户一个子进程（续）

```
    for ( ; ; ) {
        clilen = addrlen;
        if ( (connfd = accept(listenfd, cliaddr, &clilen)) < 0) {
                if (errno == EINTR)
                        continue;                       /* back to for() */
                else
                        err_sys("accept error");
        }
        if ( (childpid = Fork()) == 0) {        /* child process */
                Close(listenfd);                /* close listening socket */
                web_child(connfd);              /* process the request */
                exit(0);
        }
        Close(connfd);                  /* parent closes connected socket */
    }
}
```

TCP并发服务器：每个客户一个子进程（续）

```
#define MAXN   16384            /* max #bytes that a client can request */
void web_child(int sockfd)
{
    int             ntowrite;
    ssize_t         nread;
    char            line[MAXLINE], result[MAXN];
    for ( ; ; ) {
        if ( (nread = Readline(sockfd, line, MAXLINE)) == 0)
                return;         /* connection closed by other end */
        /* 4line from client specifies #bytes to write back */
        ntowrite = atol(line);
        if ((ntowrite <= 0) || (ntowrite > MAXN))
                err_quit("client request for %d bytes", ntowrite);
        Writen(sockfd, result, ntowrite);
    }
}
```

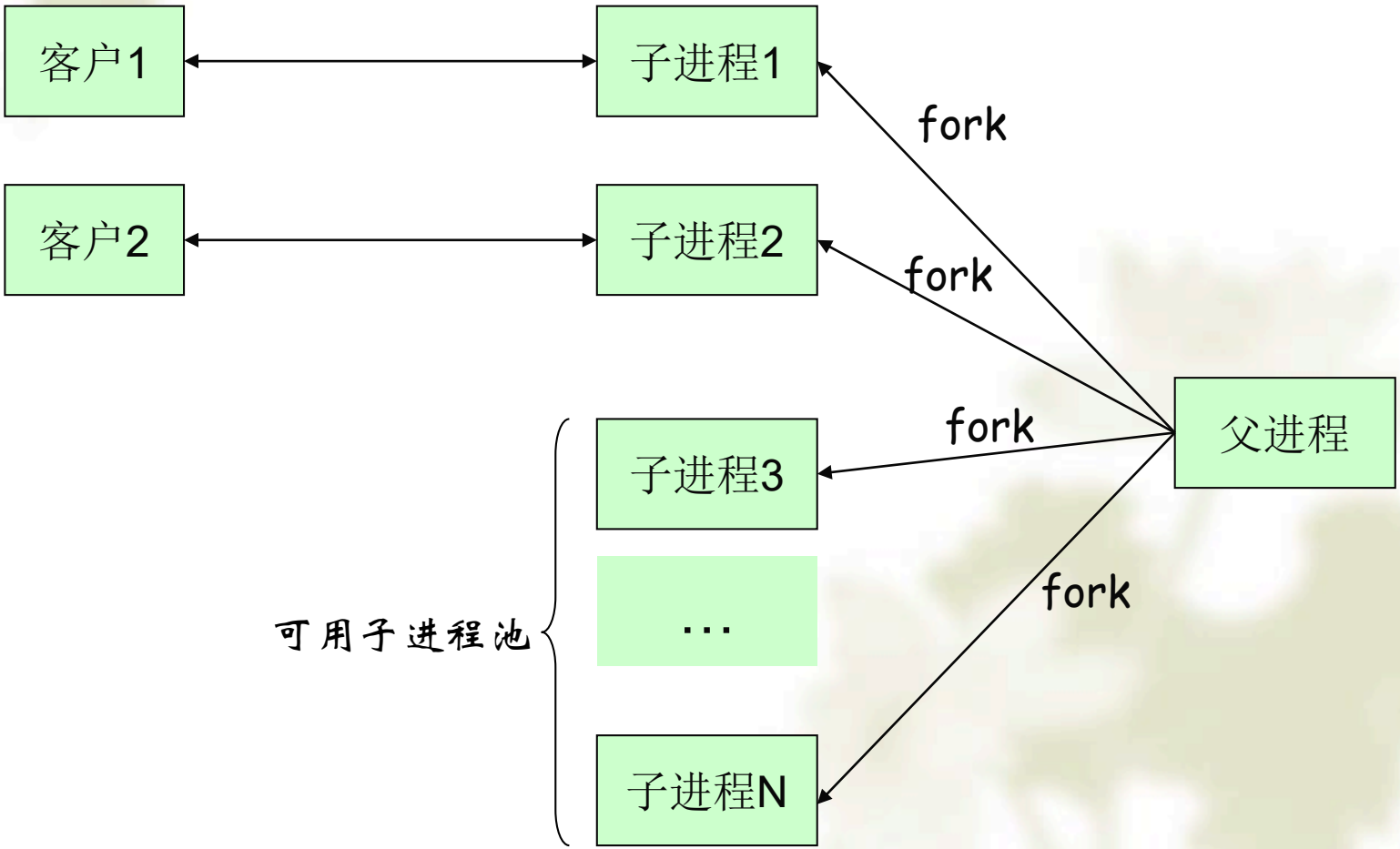
TCP并发服务器：每个客户一个子进程（续）

```
void sig_int(int signo)
{
    void  pr_cpu_time(void);
    pr_cpu_time();
    exit(0);
}

#include        <sys/resource.h>
#ifndef HAVE_GETRUSAGE_PROTO
int     getrusage(int, struct rusage *);
#endif
void pr_cpu_time(void)
{
    double              user, sys;
    struct rusage       myusage, childusage;
```

TCP并发服务器：每个客户一个子进程（续）

```
    if (getrusage(RUSAGE_SELF, &myusage) < 0)
        err_sys("getrusage error");
    if (getrusage(RUSAGE_CHILDREN, &childusage) < 0)
        err_sys("getrusage error");

    user = (double) myusage.ru_utime.tv_sec +
                    myusage.ru_utime.tv_usec/1000000.0;
    user += (double) childusage.ru_utime.tv_sec +
                     childusage.ru_utime.tv_usec/1000000.0;
    sys = (double) myusage.ru_stime.tv_sec +
                     myusage.ru_stime.tv_usec/1000000.0;
    sys += (double) childusage.ru_stime.tv_sec +
                    childusage.ru_stime.tv_usec/1000000.0;

    printf("\nuser time = %g, sys time = %g\n", user, sys);
}
```

TCP预先派生子进程服务器程序



TCP预先派生子进程服务器程序

- 这种技术的优点在于：不需要引入父进程执行fork的开销，新客户就可以得到处理。而缺点在于，每次启动服务器时，父进程必须猜测到底需要预先派生多少个子进程。除此以外，如不考虑再派生子进程，一旦所有子进程都为客户请求所占用，此时新的请求将被暂时忽略，直到有一个新的进程可用。客户的感觉就是服务器的响应变慢。
- 解决上述问题的办法是：由父进程监视可用子进程个数，一旦低于某个阈值，再派生额外的子进程，反之，则终止部分新派生的进程。

TCP预先派生子进程服务器程序：accept无上锁保护

```
static int          nchildren;
static pid_t        *pids;

int main(int argc, char **argv)
{
    int             listenfd, i;
    socklen_t       addrlen;
    void            sig_int(int);
    pid_t           child_make(int, int, int);

    if (argc == 3)
        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
    else if (argc == 4)
        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
    else
        err_quit("usage: serv02 [ <host> ] <port#> <#children>");
```

TCP预先派生子进程服务器程序：accept无上锁保护（续）

```
    nchildren = atoi(argv[argc-1]);
    pids = Calloc(nchildren, sizeof(pid_t));
    for (i = 0; i < nchildren; i++)
        pids[i] = child_make(i, listenfd, addrlen);        /* parent
    returns */

    Signal(SIGINT, sig_int);

    for ( ; ; )
        pause();        /* everything done by children */
}
```

TCP预先派生子进程服务器程序：accept无上锁保护（续）

```
pid_t
child_make(int i, int listenfd, int addrlen)
{
    pid_t           pid;
    void  child_main(int, int, int);

    if ( (pid = Fork()) > 0)
        return(pid);                    /* parent */

    child_main(i, listenfd, addrlen);           /* never returns */
}
/* end child_make */
```

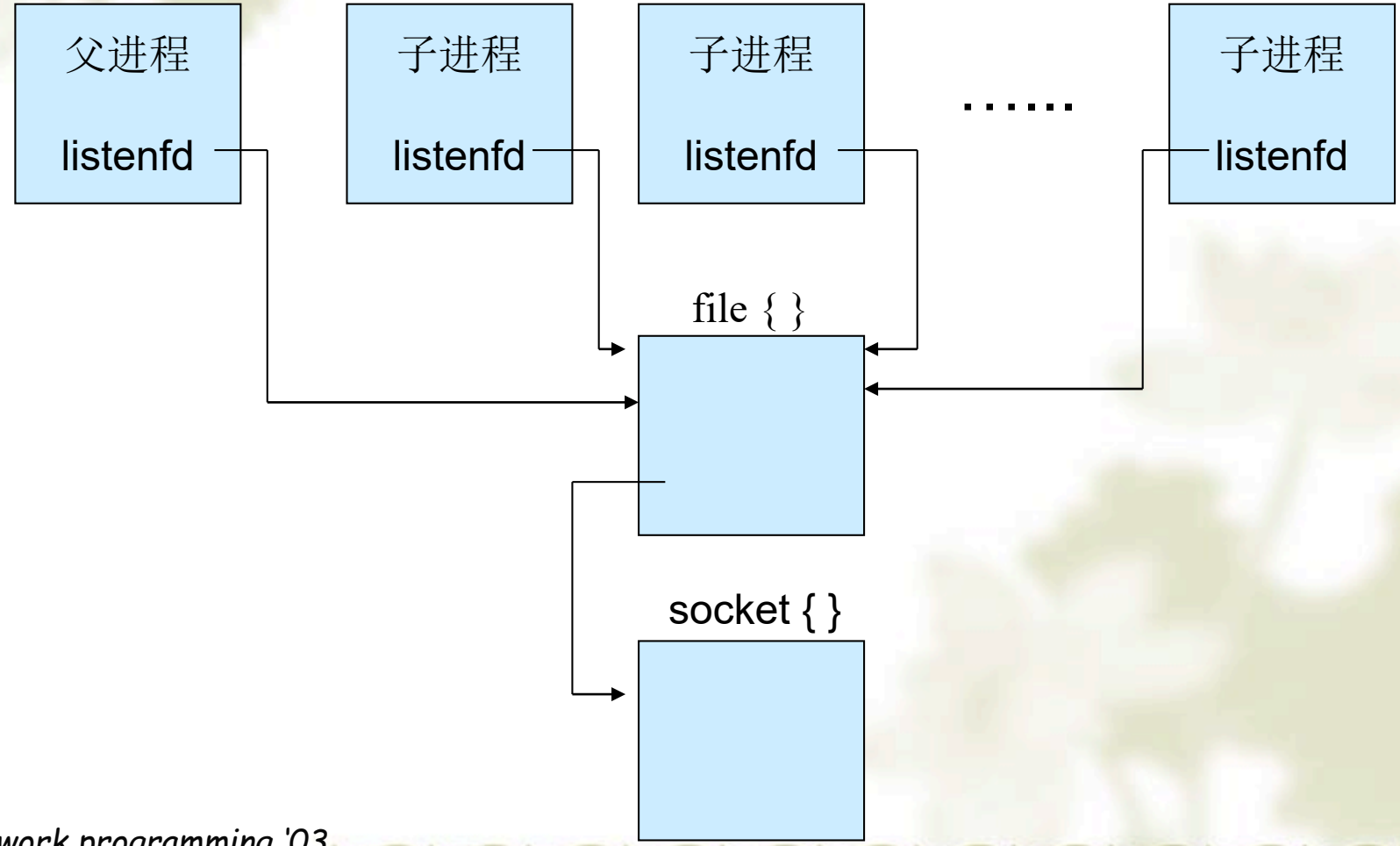
TCP预先派生子进程服务器程序：accept无上锁保护（续）

```
void child_main(int i, int listenfd, int addrlen)
{
    int                 connfd;
    void                web_child(int);
    socklen_t           clilen;
    struct sockaddr     *cliaddr;

    cliaddr = Malloc(addrlen);
    printf("child %ld starting\n", (long) getpid());
    for ( ; ; ) {
        clilen = addrlen;
        connfd = Accept(listenfd, cliaddr, &clilen);
        web_child(connfd);              /* process the request */
        Close(connfd);
    }
}
```

TCP预先派生子进程服务器程序：accept无上锁保护（续）

```
void sig_int(int signo)
{
    int             i;
    void  pr_cpu_time(void);
    /* 4terminate all children */
    for (i = 0; i < nchildren; i++)
        kill(pids[i], SIGTERM);
    while (wait(NULL) > 0)          /* wait for all children */
        ;
    if (errno != ECHILD)
        err_sys("wait error");

    pr_cpu_time();
    exit(0);
}
```

TCP预先派生子进程服务器程序：accept无上锁保护（续）



TCP预先派生子进程服务器程序：accept无上锁保护（续）

- 当程序启动后，N个子进程被派生，它们分别调用accept并由内核置入睡眠状态。当一个客户连接请求到达时，N个睡眠进程均被唤醒，但只有最先被调度执行的进程才能获得客户连接，而其他N-1个进程再次睡眠。这种情况称之为“惊群”问题，它将会导致系统服务性能下降。（测试“惊群”现象）
- 为了解决上述问题，可以增加一个监控进程，监视空闲子进程的数量。

TCP预先派生子进程服务器程序：accept使用文件锁保护

- 源自BSD的内核允许多个进程在同一监听描述字上调用accept，但对于基于SVR4（如Solaris2.5）则会在客户连接到该服务器后不久，某个子进程的accept就会返回EPROTO错误，表示协议错。
- 解决上述问题的方法就是让应用进程在调用accept前后设置某种形式的锁，以保证只有一个子进程阻塞在accept调用，而其他子进程则阻塞在试图获取提供调用accept权力的锁上。实现这种锁的方式有很多，如文件锁，信号量，互斥锁等等。
- 本节主要介绍文件锁保护accept的方式

TCP预先派生子进程服务器程序：accept使用文件锁保护（续）

- 在main函数中的唯一改动是在子进程的循环前增加一个函数调用:my_lock_init:

```
my_lock_init("/tmp/lock.XXXX");
for(i = 0; i < nchildren; i++)
        pids[i] = child_make( i, listenfd, addrlen);
```

- child_make函数不变，child_main函数的唯一改动是在调用accept前获取文件锁，在accept返回后释放文件锁。

```
for ( ; ;) {
        clilen = addrlen;
        my_lock_wait();
        connfd = accept(listenfd, cliaddr, &clilen);
        my_lock_release();
        close(connfd);
}
```

TCP预先派生子进程服务器程序：accept使用文件锁保护（续）

```
static struct flock        lock_it, unlock_it;
static int                 lock_fd = -1;
        /* fcntl() will fail if my_lock_init() not called */

void
my_lock_init(char *pathname)
{
    char   lock_file[1024];

        /* 4must copy caller's string, in case it's a constant */
    strncpy(lock_file, pathname, sizeof(lock_file));
    Mktemp(lock_file);

    lock_fd = Open(lock_file, O_CREAT | O_WRONLY, FILE_MODE);
    Unlink(lock_file);                  /* but lock_fd remains open */
```

TCP预先派生子进程服务器程序：accept使用文件锁保护（续）

```
    lock_it.l_type = F_WRLCK;
    lock_it.l_whence = SEEK_SET;
    lock_it.l_start = 0;
    lock_it.l_len = 0;

    unlock_it.l_type = F_UNLCK;
    unlock_it.l_whence = SEEK_SET;
    unlock_it.l_start = 0;
    unlock_it.l_len = 0;
}
/* end my_lock_init */
```

TCP预先派生子进程服务器程序：accept使用文件锁保护（续）

```
void my_lock_wait()
{
    int                 rc;
    while ( (rc = fcntl(lock_fd, F_SETLKW, &lock_it)) < 0) {
            if (errno == EINTR)
                    continue;
     else
                    err_sys("fcntl error for my_lock_wait");
     }
}

void my_lock_release()
{
    if (fcntl(lock_fd, F_SETLKW, &unlock_it) < 0)
            err_sys("fcntl error for my_lock_release");
}
```

TCP预先派生子进程服务器程序：accept使用文件锁保护（续）

- 从本节的第1章表中可以看出，文件上锁增加了服务器的进程控制CPU时间。同时，文件上锁方式涉及到文件系统操作，这是很耗时的。但加锁机制却是SVR4系统中多子进程accept同一监听套接字的唯一方法。
- Apache Web服务器程序1.1版利用了以上两节的技术。预先派生子进程后，如果实现允许所有子进程都阻塞在accept调用上，那么使用不加锁机制，否则就使用本节介绍的文件锁技术。

TCP预先派生子进程服务器程序：accept使用线程互斥锁保护

- 本节介绍的线程互斥锁不仅适用于同一进程内各线程间的上锁，同时也适用于不同进程间的上锁。
- 为使用线程锁，main, child_make, child_main函数都不用改动，只需要修改3个上锁函数。
- 为了在多个进程之间使用线程锁，应该作到：（1）互斥锁变量必须存储在为所有进程共享的内存中；（2）必须通知线程函数库互斥锁是在不同进程间共享的。（这同样要求线程库支持PTHREAD_PROCESS_SHARED属性）

TCP预先派生子进程服务器程序：accept使用线程互斥锁保护（续）

```
#include                    "unpthread.h"
#include          <sys/mman.h>
static pthread_mutex_t  *mptr;   /* actual mutex will be in shared memory */
void my_lock_init(char *pathname)
{
    int             fd;
    pthread_mutexattr_t mattr;
    fd = Open("/dev/zero", O_RDWR, 0);
    mptr = Mmap(0, sizeof(pthread_mutex_t), PROT_READ | PROT_WRITE,
                MAP_SHARED, fd, 0);
    Close(fd);

    Pthread_mutexattr_init(&mattr);
    Pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);
    Pthread_mutex_init(mptr, &mattr);
}
```

TCP预先派生子进程服务器程序：accept使用线程互斥锁保护（续）

```
void my_lock_wait()
{
    Pthread_mutex_lock(mptr);
}

void my_lock_release()
{
    Pthread_mutex_unlock(mptr);
}
```

TCP预先派生子进程服务器程序：传递描述字

- 对预先派生子进程服务器程序的最后一种改动就是由父进程调用accept，然后再将所接收的连接描述字传递给子进程。这样就绕过了子进程中调用accept可能需要上锁保护的问题。但为了实现给子进程传递描述字，父进程必须跟踪所有子进程的忙闲状态，以便给空闲子进程传递新的描述字。
- 为此，需要为每个子进程维护一个信息结构，用来管理子进程。

TCP预先派生子进程服务器程序：传递描述字（续）

```
typedef struct {
    pid_t       child_pid;              /* process ID */
    int         child_pipefd;/* parent's stream pipe to/from child */
    int         child_status;   /* 0 = ready */
    long        child_count;    /* #connections handled */
} Child;

Child   *cptr;          /* array of Child structures; calloc'ed */
```

TCP预先派生子进程服务器程序：传递描述字（续）

```
pid_t child_make(int i, int listenfd, int addrlen)
{
    int             sockfd[2];
    pid_t           pid;
    void  child_main(int, int, int);

    Socketpair(AF_LOCAL, SOCK_STREAM, 0, sockfd);

    if ( (pid = Fork()) > 0) {
        Close(sockfd[1]);
        cptr[i].child_pid = pid;
        cptr[i].child_pipefd = sockfd[0];
        cptr[i].child_status = 0;
        return(pid);                    /* parent */
    }
```

TCP预先派生子进程服务器程序：传递描述字（续）

```
    Dup2(sockfd[1], STDERR_FILENO);      /* child's stream pipe to
                                            parent */

    Close(sockfd[0]);
    Close(sockfd[1]);
    Close(listenfd);                                  /* child does
    not need this open */
    child_main(i, listenfd, addrlen);      /* never returns */
}
```

TCP预先派生子进程服务器程序：传递描述字（续）

```
void child_main(int i, int listenfd, int addrlen)  {
    char                    c;
    int                     connfd;
    ssize_t                 n;
    void                    web_child(int);
    printf("child %ld starting\n", (long) getpid());
    for ( ; ; ) {
        if ( (n = Read_fd(STDERR_FILENO, &c, 1, &connfd)) == 0)
                err_quit("read_fd returned 0");
        if (connfd < 0)
                err_quit("no descriptor from read_fd");
        web_child(connfd);                  /* process the request */
        Close(connfd);
        Write(STDERR_FILENO, "", 1);    /* tell parent we're ready again */
    }
}
```

TCP预先派生子进程服务器程序：传递描述字（续）

```
static int                  nchildren;
int main(int argc, char **argv)
{
    int                     listenfd, i, navail, maxfd, nsel, connfd, rc;
    void                    sig_int(int);
    pid_t                   child_make(int, int, int);
    ssize_t                 n;
    fd_set                  rset, masterset;
    socklen_t               addrlen, clilen;
    struct sockaddr         *cliaddr;
    if (argc == 3)
        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
    else if (argc == 4)
        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
    else
        err_quit("usage: serv05 [ <host> ] <port#> <#children>");
```

TCP预先派生子进程服务器程序：传递描述字（续）

```
FD_ZERO(&masterset);
FD_SET(listenfd, &masterset);
maxfd = listenfd;
cliaddr = Malloc(addrlen);

nchildren = atoi(argv[argc-1]);
navail = nchildren;
cptr = Calloc(nchildren, sizeof(Child));
/* prefork all the children */
for (i = 0; i < nchildren; i++) {
        child_make(i, listenfd, addrlen);    /* parent returns */
        FD_SET(cptr[i].child_pipefd, &masterset);
        maxfd = max(maxfd, cptr[i].child_pipefd);
}
Signal(SIGINT, sig_int);
```

TCP预先派生子进程服务器程序：传递描述字（续）

```
for ( ; ; ) {
        rset = masterset;
        if (navail <= 0)
                FD_CLR(listenfd, &rset); /* turn off if no available children */
        nsel = Select(maxfd, &rset, NULL, NULL, NULL);
                /* 4check for new connections */
        if (FD_ISSET(listenfd, &rset)) {
                clilen = addrlen;
                connfd = Accept(listenfd, cliaddr, &clilen);
                for (i = 0; i < nchildren; i++)
                        if (cptr[i].child_status == 0)
                                break;                  /* available */
                if (i == nchildren)             err_quit("no available children");
                cptr[i].child_status = 1;   /* mark child as busy */
                cptr[i].child_count++;
                navail--;
```

TCP预先派生子进程服务器程序：传递描述字（续）

```
                n = Write_fd(cptr[i].child_pipefd, "", 1, connfd);
                Close(connfd);
                if (--nsel == 0)
                        continue; /* all done with select() results */
        }
        /* 4find any newly-available children */
        for (i = 0; i < nchildren; i++) {
                if (FD_ISSET(cptr[i].child_pipefd, &rset)) {
                        if ( (n = Read(cptr[i].child_pipefd, &rc, 1)) == 0)
                                err_quit("child %d terminated unexpectedly", i);
                        cptr[i].child_status = 0;
                        navail++;
                        if (--nsel == 0)
                                break;     /* all done with select() results */
                }
        }
    }
}
```

TCP预先派生子进程服务器程序：传递描述字（续）

```
void sig_int(int signo)
{
    int             i;
    void  pr_cpu_time(void);
          /* 4terminate all children */
    for (i = 0; i < nchildren; i++)
          kill(cptr[i].child_pid, SIGTERM);
    while (wait(NULL) > 0)          /* wait for all children */
          ;
    if (errno != ECHILD)
          err_sys("wait error");
    pr_cpu_time();
    for (i = 0; i < nchildren; i++)
          printf("child %d, %ld connections\n", i, cptr[i].child_count);
    exit(0);
}
```

TCP预先派生子进程服务器程序：传递描述字（续）

- 通过字节流管道传递描述字给每个子进程，由子进程通过字节流管道写回1字节的数据以表示可用，要比无论是共享内存中的互斥锁还是文件锁的上锁和解锁更费时；
- 通过统计各个子进程的连接数，发现越是排在前头的子进程所处理的客户请求就越多，出现不公平现象。这与以前几种技术靠内核来调度子进程接收连接是公平的情况不一样。

TCP并发服务器程序：每个客户一个线程

```
int main(int argc, char **argv)
{
    int                 listenfd, connfd;
    void                sig_int(int);
    void                *doit(void *);
    pthread_t           tid;
    socklen_t           clilen, addrlen;
    struct sockaddr     *cliaddr;

    if (argc == 2)
        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
    else if (argc == 3)
        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
    else
        err_quit("usage: serv06 [ <host> ] <port#>");
    cliaddr = Malloc(addrlen);
```

TCP并发服务器程序：每个客户一个线程（续）

```
    Signal(SIGINT, sig_int);
    for ( ; ; ) {
        clilen = addrlen;
        connfd = Accept(listenfd, cliaddr, &clilen);
        Pthread_create(&tid, NULL, &doit, (void *) connfd);
    }
}

void *  doit(void *arg)
{
    void  web_child(int);
    Pthread_detach(pthread_self());
    web_child((int) arg);
    Close((int) arg);
    return(NULL);
}
```

TCP并发服务器程序：每个客户一个线程（续）

```
void
sig_int(int signo)
{
    void  pr_cpu_time(void);

    pr_cpu_time();
    exit(0);
}
```

tcp预先创建线程服务器程序：每个线程各自accept

- 我们不是让每个线程都阻塞在accept调用上，而是直接使用互斥锁来保证线程间互斥地调用accept，此处我们直接使用线程锁（因为它们是属于同一进程）。
- 为维护线程的信息，定义如下结构：

```
typedef struct {
    pthread_t       thread_tid;             /* thread ID */
    long            thread_count;    /* #connections handled */
} Thread;
Thread  *tptr;          /* array of Thread structures; calloc'ed */
int                     listenfd, nthreads;
socklen_t               addrlen;
pthread_mutex_t         mlock;
```

tcp预先创建线程服务器程序：每个线程各自accept（续）

```
#include        "unpthread.h"
#include        "pthread07.h"

pthread_mutex_t         mlock = PTHREAD_MUTEX_INITIALIZER;
int main(int argc, char **argv)
{
    int             i;
    void  sig_int(int), thread_make(int);

    if (argc == 3)
        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
    else if (argc == 4)
        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
    else
        err_quit("usage: serv07 [ <host> ] <port#> <#threads>");
```

tcp预先创建线程服务器程序：每个线程各自accept（续）

```
    nthreads = atoi(argv[argc-1]);
    tptr = Calloc(nthreads, sizeof(Thread));

    for (i = 0; i < nthreads; i++)
        thread_make(i);                    /* only main thread returns */

    Signal(SIGINT, sig_int);

    for ( ; ; )
        pause(); /* everything done by threads */
}
```

tcp预先创建线程服务器程序：每个线程各自accept（续）

```
void
sig_int(int signo)
{
    int                 i;
    void  pr_cpu_time(void);

    pr_cpu_time();

    for (i = 0; i < nthreads; i++)
        printf("thread %d, %ld connections\n", i, tptr[i].thread_count);

    exit(0);
}
```

tcp预先创建线程服务器程序：每个线程各自accept（续）

```
void
thread_make(int i)
{
    void  *thread_main(void *);

    Pthread_create(&tptr[i].thread_tid, NULL, &thread_main, (void *) i);
    return;                     /* main thread returns */
}
```

tcp预先创建线程服务器程序：每个线程各自accept（续）

```
void * thread_main(void *arg) {
    int                 connfd;
    void                web_child(int);
    socklen_t           clilen;
    struct sockaddr     *cliaddr;
    cliaddr = Malloc(addrlen);
    printf("thread %d starting\n", (int) arg);
    for ( ; ; ) {
        clilen = addrlen;
        Pthread_mutex_lock(&mlock);
        connfd = Accept(listenfd, cliaddr, &clilen);
        Pthread_mutex_unlock(&mlock);
        tptr[(int) arg].thread_count++;
        web_child(connfd);              /* process the request */
        Close(connfd);
    }
}
```

tcp预先创建线程服务器程序：主线程统一accept

- 主要的设计问题是如何将已连接的描述字传递给线程池中的空闲线程。我们可以用以前的方法（传递描述字）。不过由于几个线程都在同一各进程内，所以无需传递描述字，所需知道的只是描述字号。
- 需要如下结构：

```
typedef struct {
    pthread_t       thread_tid;             /* thread ID */
    long            thread_count;    /* #connections handled */
} Thread;
Thread  *tptr;          /* array of Thread structures; calloc'ed */
#define MAXNCLI     32
int                 clifd[MAXNCLI], iget, iput;
pthread_mutex_t     clifd_mutex;
pthread_cond_t      clifd_cond;
```

存储已连接套接口描述字。

tcp预先创建线程服务器程序：主线程统一accept（续）

```
#include    "unpthread.h"
#include    "pthread08.h"

static int                  nthreads;
pthread_mutex_t             clifd_mutex =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t          clifd_cond = PTHREAD_COND_INITIALIZER;

int
main(int argc, char **argv)
{
    int                 i, listenfd, connfd;
    void            sig_int(int), thread_make(int);
    socklen_t       addrlen, clilen;
    struct sockaddr         *cliaddr;
```

tcp预先创建线程服务器程序：主线程统一accept（续）

```
if (argc == 3)
        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
else if (argc == 4)
        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
else
        err_quit("usage: serv08 [ <host> ] <port#> <#threads>");
cliaddr = Malloc(addrlen);
nthreads = atoi(argv[argc-1]);
tptr = Calloc(nthreads, sizeof(Thread));
iget = iput = 0;

/* create all the threads */
for (i = 0; i < nthreads; i++)
        thread_make(i);             /* only main thread returns */

Signal(SIGINT, sig_int);
```

tcp预先创建线程服务器程序：主线程统一accept（续）

```
    for ( ; ; ) {
        clilen = addrlen;
        connfd = Accept(listenfd, cliaddr, &clilen);

        Pthread_mutex_lock(&clifd_mutex);
        clifd[iput] = connfd;
        if (++iput == MAXNCLI)
            iput = 0;
        if (iput == iget)
            err_quit("iput = iget = %d", iput);
        Pthread_cond_signal(&clifd_cond);
        Pthread_mutex_unlock(&clifd_mutex);
    }
}
```

tcp预先创建线程服务器程序：主线程统一accept（续）

```
void
sig_int(int signo)
{
    int             i;
    void pr_cpu_time(void);

    pr_cpu_time();

    for (i = 0; i < nthreads; i++)
        printf("thread %d, %ld connections\n", i,
    tptr[i].thread_count);

    exit(0);
}
```

tcp预先创建线程服务器程序：主线程统一accept（续）

```
#include        "unpthread.h"
#include        "pthread08.h"

void
thread_make(int i)
{
    void  *thread_main(void *);

    Pthread_create(&tptr[i].thread_tid, NULL, &thread_main, (void *) i);
    return;                     /* main thread returns */
}
```

tcp预先创建线程服务器程序：主线程统一accept（续）

```
void * thread_main(void *arg)  {
    int             connfd;
    void  web_child(int);
    printf("thread %d starting\n", (int) arg);
    for ( ; ; ) {
        Pthread_mutex_lock(&clifd_mutex);
        while (iget == iput)
            Pthread_cond_wait(&clifd_cond, &clifd_mutex);
        connfd = clifd[iget];       /* connected socket to service */
        if (++iget == MAXNCLI)              iget = 0;
        Pthread_mutex_unlock(&clifd_mutex);
        tptr[(int) arg].thread_count++;
        web_child(connfd);                  /* process the request */
        Close(connfd);
    }
}
```

tcp预先创建线程服务器程序：主线程统一accept（续）

- 这个版本的程序比上一节的要慢一些，原因可能在于本节的例子同时需要互斥锁和条件变量。而上一节的程序只用到了互斥锁。