Network Programming:
Ch.5 TCP Client-Server Example

Li-Hsing Yen

NYCU

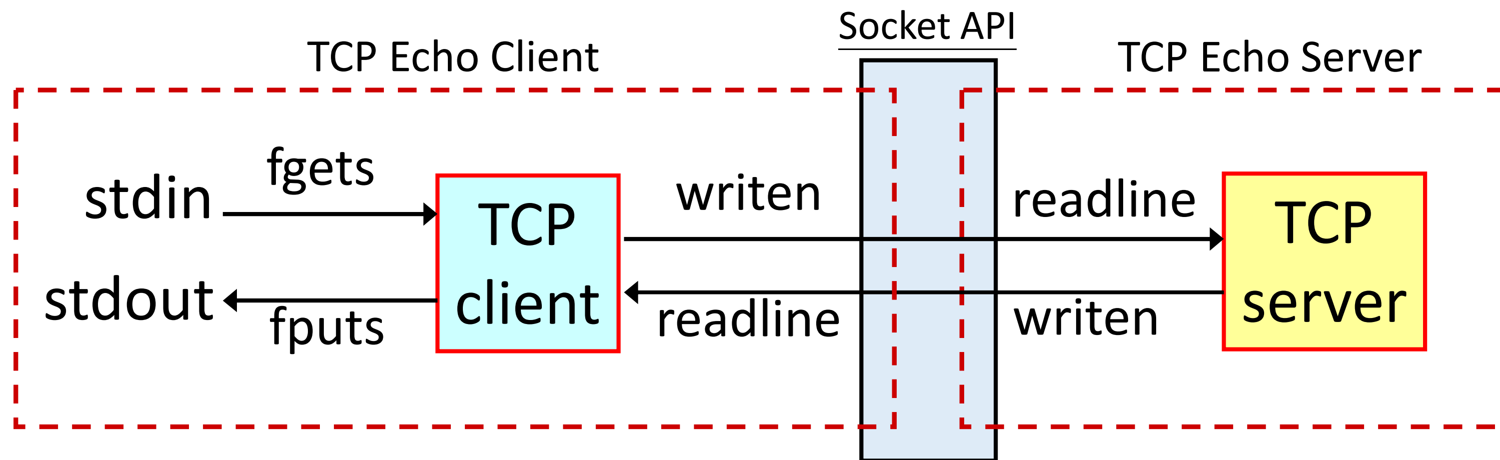
Ver. 1.0.0

TCP Client-Server Example

- TCP echo server: *main* and *str_echo*
- TCP echo client: *main* and *str_cli*
- Normal startup and termination
- POSIX signal handling
- Handling SIGCHILD, interrupted system calls, and preventing zombies
- Connection abort before *accept* returns
- Crash of server process

- SIGPIPE signal
- Crash, reboot, shutdown of server host
- Summary of TCP example
- Data format: passing string or binary

TCP Echo Server and Client



- To expand this example to other applications, just change what the server does with the client input.
- Many boundary conditions to handle: signal, interrupted system call, server crash, etc. The first version does not handle them.

TCP Echo Server: main function

tcpcliserv/tcpserv01.c

非最終版本

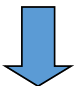
```
#include      "unp.h"
int
main(int argc, char **argv)
{
      int                        listenfd, connfd;
      pid_t                      childpid;
      socklen_t                  clilen;
      struct sockaddr_in         cliaddr, servaddr;

      listenfd = Socket(AF_INET, SOCK_STREAM, 0);

      bzero(&servaddr, sizeof(servaddr));
      servaddr.sin_family      = AF_INET;
      servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
      servaddr.sin_port        = htons(SERV_PORT);

      Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
```

9877

```
struct sockaddr_in {
  uint8_t            sin_len;
  sa_family_t        sin_family;
  in_port_t          sin_port;
  struct in_addr     sin_addr;
  char               sin_zero[8];
};
```

```
struct in_addr {
  in_addr_t       s_addr;
};
```

接
下
頁



非最終版本

```
    Listen(listenfd, LISTENQ);

    for ( ; ; ) {
        clilen = sizeof(cliaddr);
        connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);

        if ( (childpid = Fork()) == 0) {      /* child process */
            Close(listenfd);      /* close listening socket */
            str_echo(connfd);     /* process the request */
            exit(0);
        }
        Close(connfd);              /* parent closes connected socket */
    }
}
```

定義在下頁

應用程式結束(exit)時，所開啟的資源會自動被系統釋放掉

TCP Echo Server: `str_echo` function

lib/str_echo.c

```
#include      "unp.h"

void
str_echo(int sockfd)
{
        ssize_t       n;
        char          buf[MAXLINE];

again:
        while ( (n = read(sockfd, buf, MAXLINE)) > 0)
                Writen(sockfd, buf, n);

        if (n < 0 && errno == EINTR)
                goto again;
        else if (n < 0)
                err_sys("str_echo: read error");
}
```

傳回0表對方close connection

interrupted by a signal

此程式與第二版不同

TCP Echo Client: main function

tcpcliserv/tcpcli01.c

```
#include       "unp.h"
int
main(int argc, char **argv)
{
        int                     sockfd;
        struct sockaddr_in  servaddr;

        if (argc != 2)
                err_quit("usage: tcpcli <IPaddress>");
        sockfd = Socket(AF_INET, SOCK_STREAM, 0);

        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_port = htons(SERV_PORT);
        Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

        Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
        str_cli(stdin, sockfd);        /* do it all */
        exit(0);
}
```

```
struct sockaddr_in {
   uint8_t          sin_len;
   sa_family_t      sin_family;
   in_port_t        sin_port;
   struct in_addr   sin_addr;
   char             sin_zero[8];
};
```

```
struct in_addr {
   in_addr_t       s_addr;
};
```

定義在下頁

指向標準輸入裝置的檔案指標

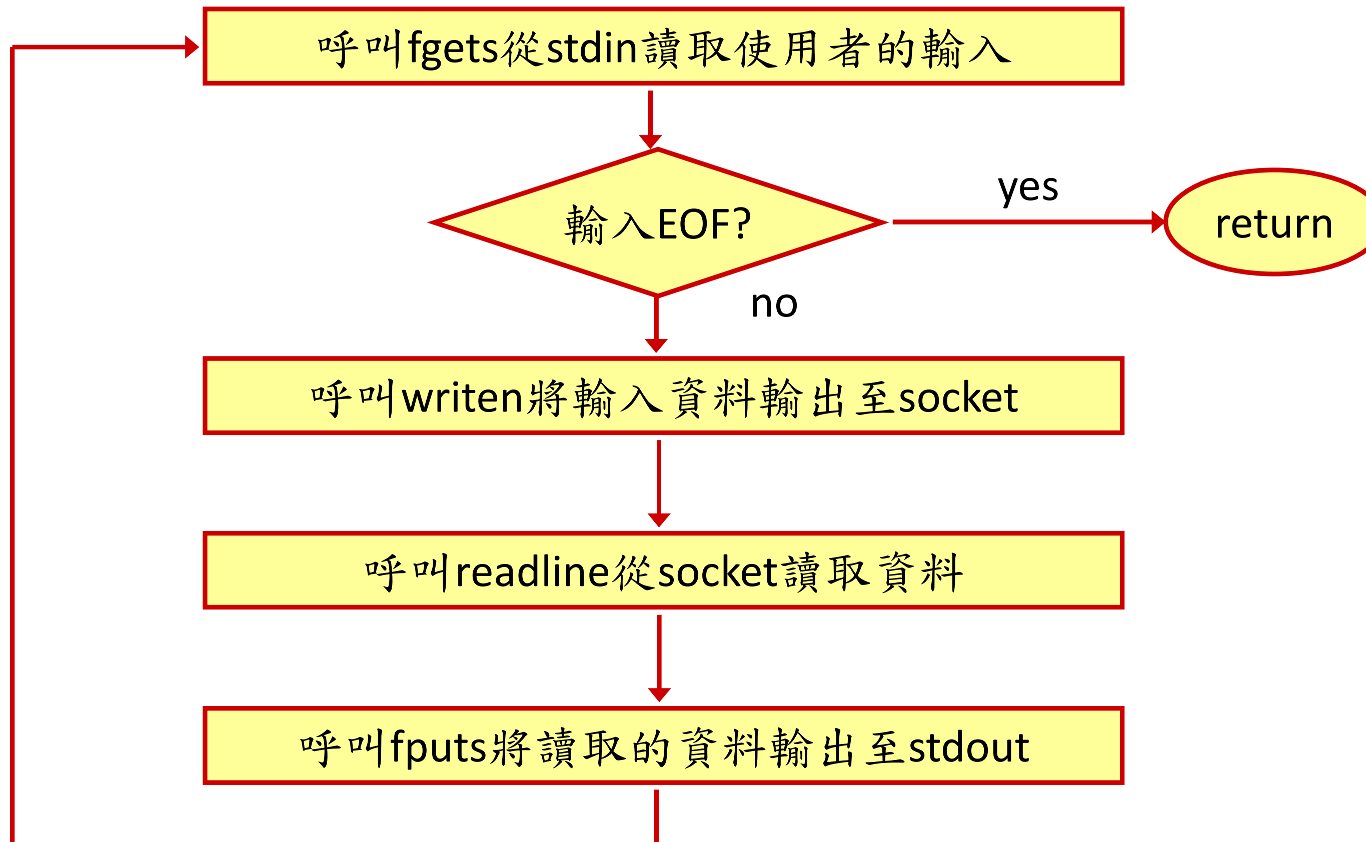
若編譯出現stdin未定義，請在檔案最前面加入#include <stdio.h>

TCP Echo Client: str_cli function

lib/str_cli.c

```
#include      "unp.h"

void
str_cli(FILE *fp, int sockfd)
{
        char    sendline[MAXLINE], recvline[MAXLINE];

        while (Fgets(sendline, MAXLINE, fp) != NULL) {

                Writen(sockfd, sendline, strlen(sendline));

                if (Readline(sockfd, recvline, MAXLINE) == 0)
                        err_quit("str_cli: server terminated prematurely");

                Fputs(recvline, stdout);
        }
}
```

Flow of str_cli (Version 1)



Normal Startup (1/3)

背景執行

- Start the server in the background

```
ws1 [unpv13e/tcpcliserv]% ./tcpserv01 &
[1] 10163
ws1 [unpv13e/tcpcliserv]%
```

Process ID

- Verify the state of the server's listening socket

```
ws1 [unpv12e/tcpcliserv]% netstat -a | grep 9877

TCP: IPv4
   Local Address        Remote Address    Swind Send-Q Rwind Recv-Q  State
-------------------- -------------------- ----- ------ ----- ------ -------
      *.9877               *.*                0      0 24576      0 LISTEN
```

(相關指令與輸出會因系統而異，此輸出與課本範例不同)

Normal Startup (2/3)

- Start the client in the same host

```
ws1 [unpv12e/tcpcliserv]% ./tcpcli01 127.0.0.1
```

- Verify the connection

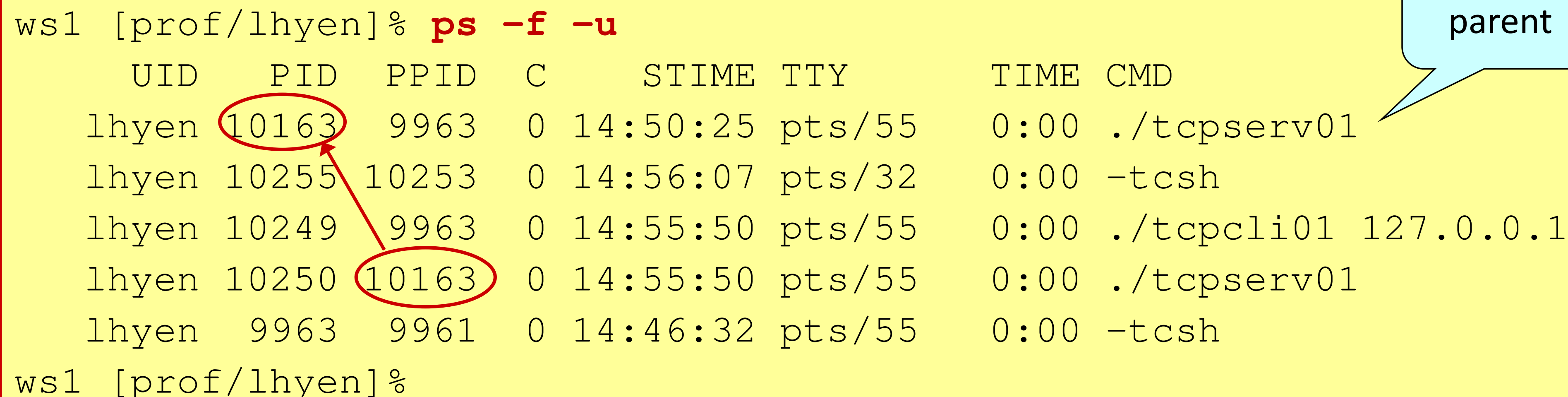
```
ws1 [prof/lhyen]% netstat | grep '9877'
localhost.46570      localhost.9877       32768      0 32768      0 ESTABLISHED
localhost.9877       localhost.46570      32768      0 32768      0 ESTABLISHED
```

(在此系統上沒有出現server parent，但課本範例有)

Proto			Local Address	Foreign Address	(state)
tcp	0	0	localhost.9877	localhost.1052	ESTABLISHED (server child)
tcp	0	0	localhost.1052	localhost.9877	ESTABLISHED (client)
tcp	0	0	*:9877	*:*	LISTEN (server parent)

Normal Setup (3/3)

- Using `ps` to show all the processes



(相關指令與輸出會因系統而異，此輸出與課本範例不同)

課本的ps輸出

server parent

To check the process status: ps -l

pid	ppid	WCHAN	STAT	TT	TIME	COMMAND
19130	19129	wait	Is	p1	0:04.00	-ksh (ksh)
21130	19130	netcon	I	p1	0:00.06	tcpserv
21131	19130	ttyin	I+	p1	0:00.09	tcpcli 127.0.0.1
21132	21130	netio	I	p1	0:00.01	tcpserv
21134	21133	wait	Ss	p2	0:03.50	-ksh (ksh)
21149	21134	-	R+	p2	0:00.05	ps -l

server child

Normal Termination (1/2)

```
ws1 [unpv12e/tcpcliserv]% ./tcpcli01 127.0.0.1
this is a test
this is a test
^D
ws1 [unpv12e/tcpcliserv]%
```

將使得Fgets傳回NULL

- Client離開str_cli ⇒ 呼叫exit離開main
- Client結束時，所有已開啟的socket會被kernel自動關閉。Client的TCP會送FIN至server，server的TCP會回ACK.
- client⇒server child這一半的connection已關閉。Client socket (TCP) 進入TIME_WAIT state

Normal Termination (2/2)

- 當server TCP收到FIN時，server child正等在read，故read會傳回0，導致server child離開str_echo回到main。
- Server child呼叫exit會使server 端的TCP關閉另一半(server child⇒client)的連結(傳送FIN，等待ACK)。
- Server child程序結束後成為zombie

Zombie Process

```
pid    TT  STAT   TIME      COMMAND
19130 p1  Ss     0:05.08  -ksh (ksh)
21130 p1  I      0:00.06  ./tcpserv
21132 p1  Z      0:00.00  (tcpserv)
21167 p1  R+     0:00.10  ps
```

zombie server child process

(另一個系統的輸出)

```
ws1 [prof/lhyen]% ps
     UID   PID  PPID  C    STIME TTY       TIME CMD
   lhyen 10163  9963  0 14:50:25 pts/55    0:00 ./tcpserv01
   lhyen 10255 10253  0 14:56:07 pts/32    0:00 -tcsh
   lhyen 10250 10163  0                    0:00 <defunct>
   lhyen  9963  9961  0 14:46:32 pts/55    0:00 -tcsh
ws1 [prof/lhyen]%
```

Cleaning Up Zombie Processes

- When a child process terminates, it becomes a zombie
- A zombie process 保留child結束時的狀態以備parent取得(使用**wait**)，因此會佔用kernel空間
- Therefore, a parent process should **wait** for child processes to kill possible zombie processes

`wait` and `waitpid` System Calls

- A process that calls `wait` or `waitpid` can
  - Block, if all of its children are still running
  - Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched (the zombie can rest in peace after that)
  - Return immediately with an error, if it doesn't have any child processes

The **wait()** System Call

```
#include <sys/wait.h>
pid_t wait (int *statloc);
        returns: process ID if OK, 0, or -1 on error
```

suspends the calling process until one of its child processes ends

- *statloc*: pointer to an integer that keeps child status information (NULL: no status information returns)
- child status information includes
  - normal/abnormal termination
  - termination cause
  - exit status

Analyzing Status Information

- status information stored at the location pointed to by *statloc* can be evaluated with the following macros:

Macro	Meaning
WIFEXITED(**statloc*)	evaluates to a nonzero (true) value if the child process terminates normally.
WEXITSTATUS(**statloc*)	if the child process terminates normally, this macro evaluates to the lower 8 bits of the value passed to the exit or _exit function or returned from main .
WIFSIGNALED(**statloc*)	evaluates to a nonzero (true) value if the child process terminates because of an unhandled signal.
WTERMSIG(**statloc*)	if the child process ends by a signal that was not caught, this macro evaluates to the number of that signal.

To Make A Parent Wait For Its Child

- An example to make the parent wait for the termination of the child process

```
if (rc < 0) { // fork failed; exit
  fprintf(stderr, "fork failed\n");
  exit(1);
} else if (rc == 0) { // child (new process)
  printf("hello, I am child (pid:%d)\n", (int) getpid());
} else { // parent goes down this path (main)
  int rc_wait = wait(NULL);
  printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n", rc, rc_wait,
          (int) getpid());
}
```

Avoid Blocking in `wait()`

- A parent process will block if all of its children are still running
- If the parent has to call **`wait()`** to kill zombie processes but doesn't want to block waiting for the termination of its child processes, how can it do?
- Use signal (software interrupt).

POSIX Signal Handling

- Signal (software interrupt): a notification to a process that an event has occurred
- sent by one process to another process (or to itself) or by the kernel to a process (signal是程序間的溝通方式)
- Kernel對每個產生的signal有預設的處理方式。有些signal會被kernel忽略，有些signal會導致kernel中止程序的執行。
- 對於任何信號，程序可以事先設定要讓系統用預設的方式處理、忽略此信號、或是由程序自行處理。

Signal Handler

- 如果程序選擇自行處理signal，由於signal是kernel或別的程序主動送過來的，程序沒有辦法預料何時會收到signal 。
- 我們可以設計一個函數，稱為signal handler，負責補捉(catch)並處理收到的signal。此函數不由我們的程式主動呼叫，而是被動由signal觸發執行。
- kernel並不知道程序的signal handler是哪一個，所以程序必須事先呼叫`sigaction`向kernel註冊程序的signal handler函數。

Signal Handler的觸發執行

- Signal handler被觸發執行時，程序正在執行的動作會被暫停，signal handler結束後才能繼續

```
for ( ; ; ) {
      connfd = Accept (listenfd, ...);
            …
      Close (connfd);
}
…
```

signal handler

```
void sig_chld(int signo)
{
      pid_t   pid;
      int      stat;

      pid = wait(&stat);
      printf("child %d terminated\n", pid);
      return;
}
```

Restarting an interrupted system call

- A process may block in a slow system call (e.g., `accept()`) when it catches a signal
- When the signal handler returns, *some* kernels may automatically restart *some* interrupted system calls
- If an interrupted system call cannot be restarted, it will return an error of EINTR

Signal Terminology and Semantics (1/2)

- A signal is generated when the event that causes the signal occurs
- A signal is delivered to a process when the action for the signal is taken
- A process can block the delivery of a signal
  - A blocked signal can still be generated but not be delivered to the process (pending)
  - Signal mask defines the set of signals currently blocked from delivery to a process

Signal Terminology and Semantics (2/2)

- What happens if a blocked signal is generated more than once?
  - The signals are queued if the system delivers the signal more than once
  - Most Unix systems do not queue signals
- When happens if more than one signal is ready to be delivered to a process?
  - POSIX.1 does not specify the order in which the signals are delivered to the process

Disposition of a Signal

- Disposition of a signal
  - the action associated with the signal
- We set the disposition of a signal by calling `sigaction` with 3 choices
  - catch the signal by a specified signal handler
  - **SIG_IGN**: ignore it (一律忽略)
  - **SIG_DFL**: default: terminate or ignore (使用系統預設)

Function signal

- sigaction使用上較為複雜，故作者定義signal函數來包裝sigaction
- 呼叫signal函數需傳入
  - signal no (每個signal都有唯一的號碼)
  - 指向signal hander的函數指標(或**SIG_IGN**, **SIG_DFL**)

struct sigaction

- definition

指向signal handler的指標

SIG_IGN: ignore
SIG_DFL: default

```
struct sigaction {
    void        (*sa_handler)(int);
    void        (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t    sa_mask;
    int         sa_flags;
    void        (*sa_restorer)(void);
};
```

Signal mask

signal Function That Enables System Call Restart

lib/signal.c

```
#include	"unp.h"
Sigfunc *signal(int signo, Sigfunc *func)
{
	struct sigaction	act, oact;

	act.sa_handler = func;
	sigemptyset(&act.sa_mask);
	act.sa_flags = 0;
	if (signo == SIGALRM) {
#ifdef	SA_INTERRUPT
		act.sa_flags |= SA_INTERRUPT;	/* SunOS 4.x */
#endif
	} else {
#ifdef	SA_RESTART
		act.sa_flags |= SA_RESTART;		/* SVR4, 44BSD */
#endif
	}
	if (sigaction(signo, &act, &oact) < 0)
		return(SIG_ERR);
	return(oact.sa_handler);
}
```

Pointer to new signal handler

執行此signal hander時並不額外封阻其它signal的產生

不見得在所有版本的Unix都會成功

讓kernel自動restart被此signal中斷的system call

傳回原先的signal handler

Signal Function That Enables System Call Restart (cont.)

```
Sigfunc *Signal(int signo, Sigfunc *func)  {
        Sigfunc *sigfunc;

        if ( (sigfunc = signal(signo, func)) == SIG_ERR)
                err_sys("signal error");
        return(sigfunc);
}
```

Signal是將signal包裝起來的函數

POSIX signal semantics:

1. Once a signal handler is installed, it remains installed.
2. The signal being delivered is blocked while a signal handler is executing.
3. By default, signals are not queued.

Handling Zombies in Signal Handler

- **SIGCHLD**: a signal sent by the kernel to the parent when a child process is terminated
- First, establish a signal handler to cache **SIGCHLD**

name of the signal handler

```
Signal(SIGCHLD, sig_chld)
```

函數名稱就是指向此函數的指標

- Second, within the handler we call **wait**. (see next page)

Signal Handler for **SIGCHLD**

tcpcliserv/sigchldwait.c

```
#include       "unp.h"

void
sig_chld(int signo)
{
        pid_t   pid;
        int       stat;

        pid = wait(&stat);
        printf("child %d terminated\n", pid);
        return;
}
```

kernel會將signal no傳入

呼叫**wait**以免讓child變成zombie

child的 process id

在signal handler中呼叫printf是不建議的

Interrupted System Call (in the TCP Server)

如果Signal handler被觸發執行時，程序正等在(block)某個system call，則此system call會被暫時中斷(interrupted)。

Server Parent

```
for ( ; ; ) {
        connfd = Accept (listenfd, ...);
        if ( (pid = Fork ( ) ) == 0) {
                Close (listenfd);
                doit (connfd);
                Close (connfd);
                exit (0);
        }
        Close (connfd);
}
…
```

parent blocks here

server child

```
for ( ; ; ) {
        connfd = Accept (listenfd, ...);
        if ( (pid = Fork ( ) ) == 0) {
                Close (listenfd);
                doit (connfd);
                Close (connfd);
                exit (0);
        }
        Close (connfd);
}
```

child executes

terminates

SIGCHLD

Interrupted System Calls

- 當kernel發出**SIGCHLD** signal而被parent捕捉時，parent是block在**accept**這個system call中。此system call被interrupt去執行handler
- 當signal handler return後，如果此interrupted system call沒有被kernel restart，此system call會傳回錯誤(**EINTR**)
  - 我們定義的signal函數有設定要restart system call ← 但不一定work
- EINTR的錯誤在作者定義的**Accept**函數中沒被特別處理；該函數會中止程式的執行

Handling Interrupted System Calls

like accept

- *Slow system call*: system call that can block forever
- Some kernels automatically restart some interrupted system calls, while some don't
- We must prepare for slow system calls to yield EINTR

```
for ( ; ; ) {
    clilen = sizeof (cliaddr);
    if ( (connfd = accept (listenfd, (SA) &cliaddr, &clilen)) < 0) {
        if (errno == EINTR)
            continue;  /* back to for ( ) */
        else
            err_sys ("accept error");
    }
```

修改後的版本

呼叫**accept**而不要呼叫Accept

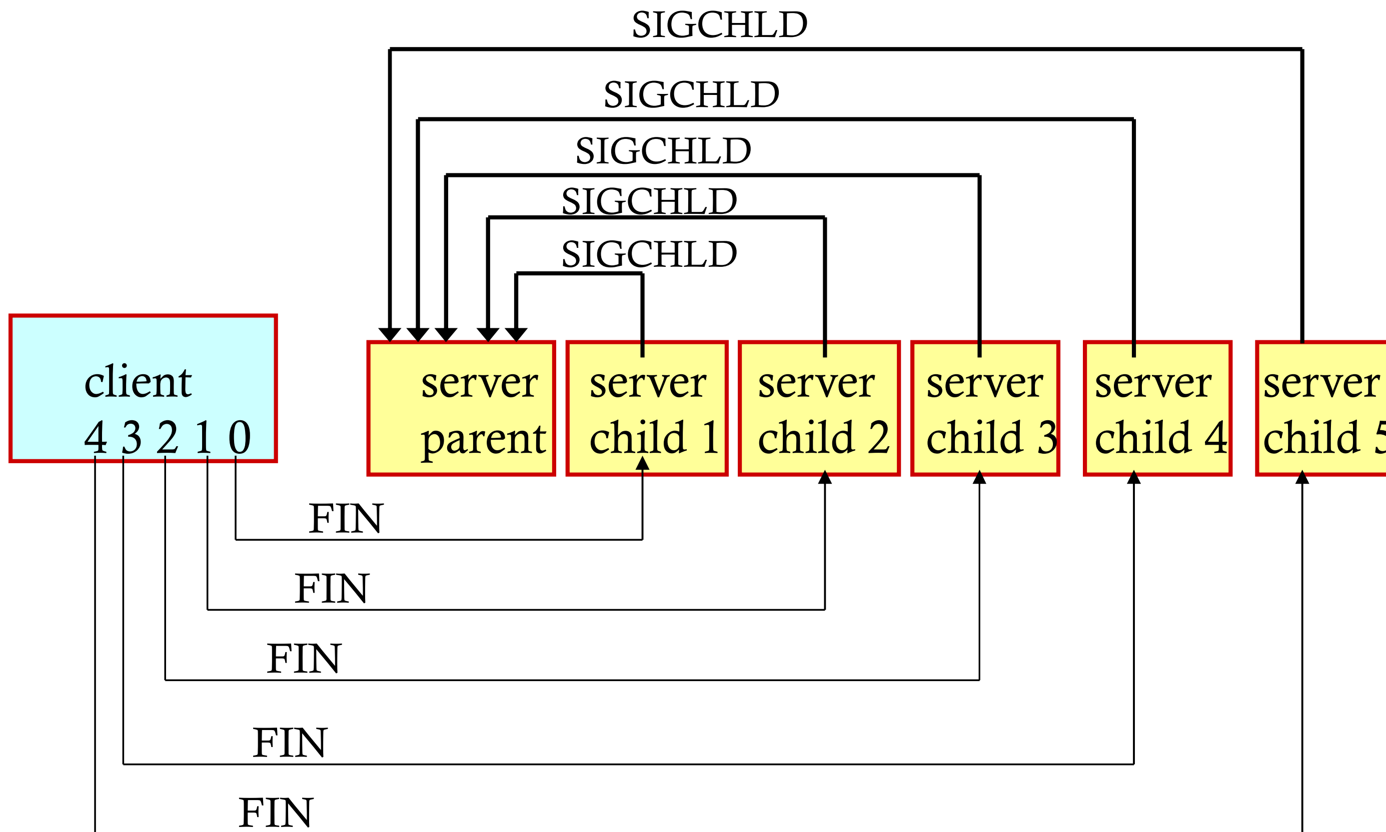
我們自行restart **accept**這個system call

Weakness of `Wait`

- Unix signals normally are not queued
- multiple occurrences of the same signal only cause the handler to be called once
- It's a problem when multiple children terminate at the same time
- Solution: use `waitpid` instead of `wait` in the handler to kill all zombies

Client terminates all five connections

catching all SIGCHLD signals in server parent



Difference Between **`wait`** and **`waitpid`**

- **`wait`** can block the caller until a child process terminates, whereas **`waitpid`** has an option (**`WNOHANG`**) that prevents it from blocking
- **`waitpid`** has a number of options that control which process it waits for (not necessarily the one that terminates first)

SIGCHLD Handler Using Waitpid

```
#include <sys/wait.h>
pid_t waitpid (pid_t pid, int *statloc, int options);
        returns: process ID if OK, o, or -1 on error
```

```
#include  "unp.h"

void
sig_chld(int signo)
{
        pid_t   pid;
        int     stat;

        while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
                printf("child %d terminated\n", pid);
        return;
}
```

Final (correct) Version of TCP Echo Server

handling SIGCHLD, EINTR from accept, zombies

tcpcliserv/tcpserv04.c

```
#include        "unp.h"
int
main(int argc, char **argv)
{
        int                     listenfd, connfd;
        pid_t                   childpid;
        socklen_t               clilen;
        struct sockaddr_in      cliaddr, servaddr;
        void                    sig_chld(int);

        listenfd = Socket(AF_INET, SOCK_STREAM, 0);

        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family      = AF_INET;
        servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
        servaddr.sin_port        = htons(SERV_PORT);

        Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
        Listen(listenfd, LISTENQ);
```

Final (correct) Version of TCP Echo Server (cont.)

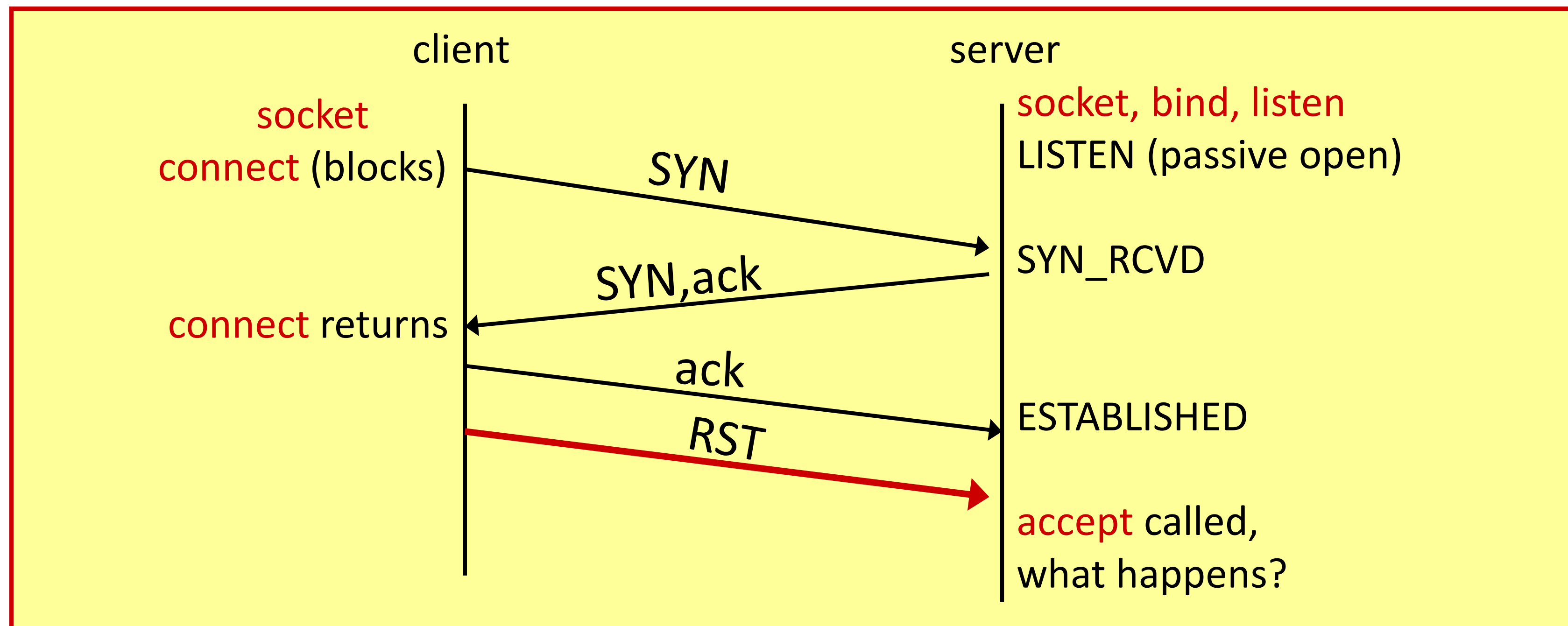
tcpcliserv/tcpserv04.c

```
    Signal(SIGCHLD, sig_chld);    /* must call waitpid() */
    for ( ; ; ) {
        clilen = sizeof(cliaddr);
        if ( (connfd = accept(listenfd, (SA *) &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 */
            str_echo(connfd);   /* process the request */
            exit(0);
        }
        Close(connfd);              /* parent closes connected socket */
    }
}
```

sig_chld程式碼已顯示在p.31

Connection Abort Before *accept* Returns

implementation dependent !



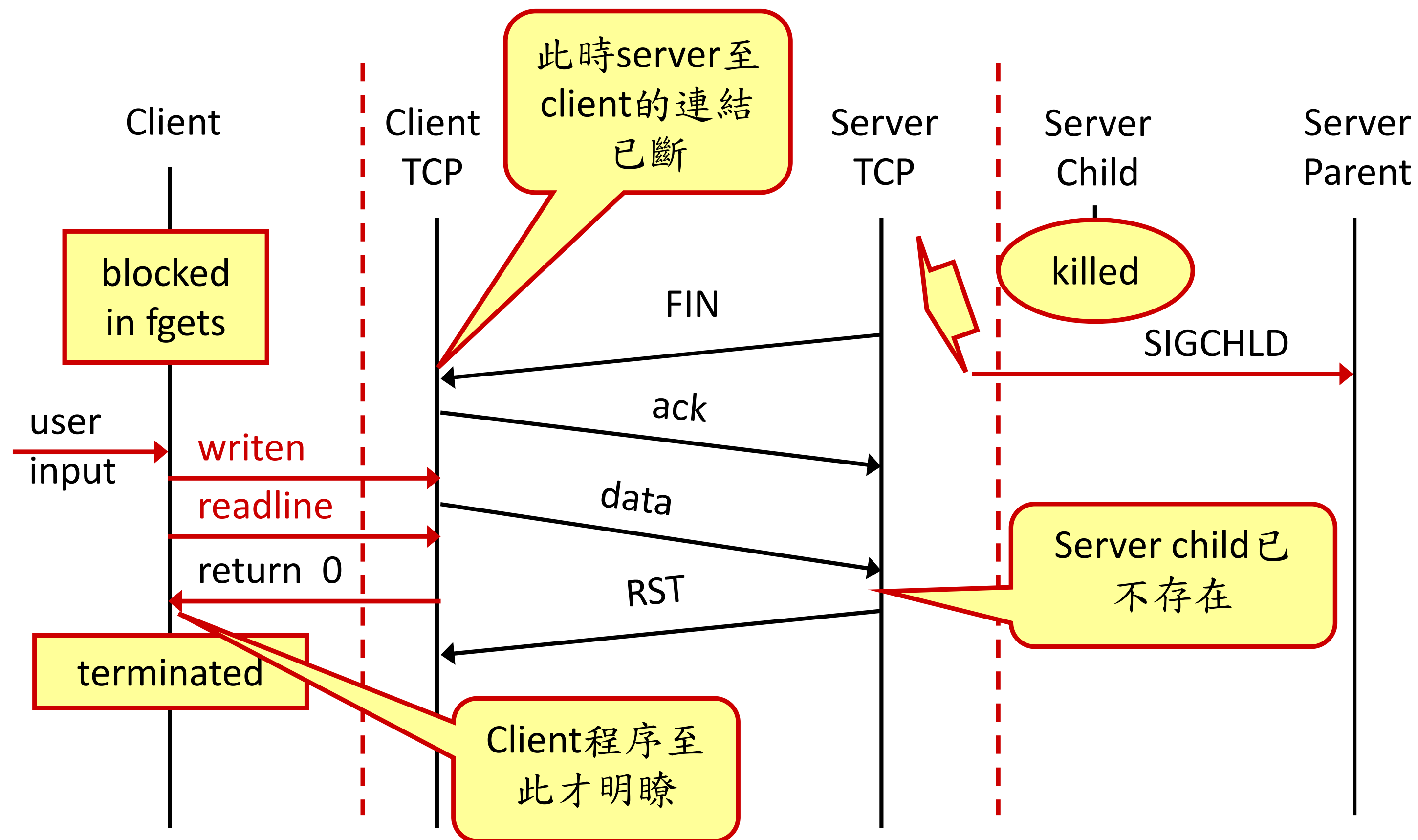
In BSD, kernel handles this. *accept* does not return.

In SVR4, *accept* is returned with EPROTO.

In POSIX.1g, *accept* is returned with ECONNABORTED

The server can ignore the error and just call accept again

Termination of Server Process



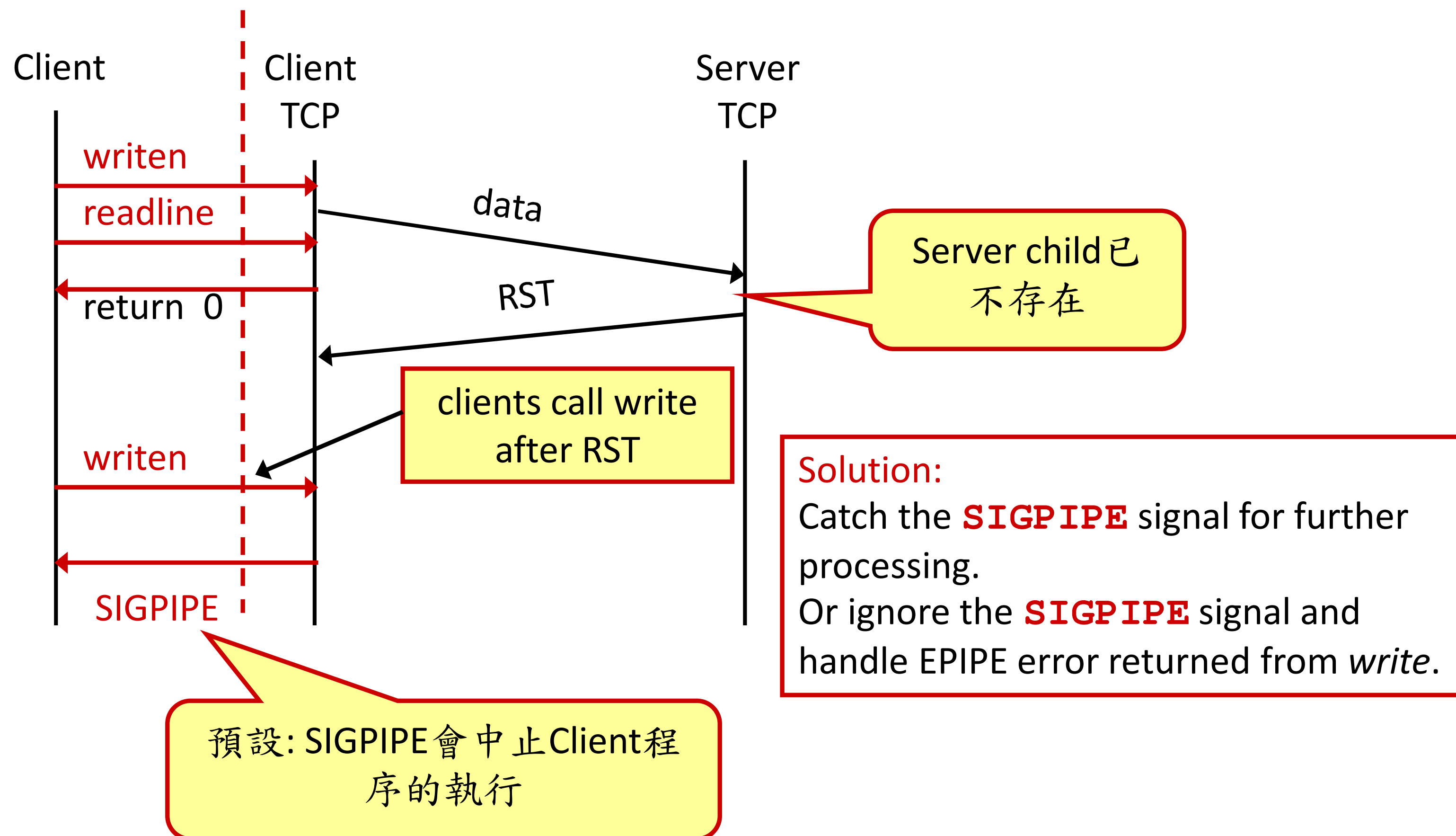
前例中Client設計的問題

- Client用兩個system calls分別取得input
  - **fgets**用於stdin的user input
  - **readline**用於socket input
- Client沒有設計成同時等待兩個input。因此當client等在**fgets**時，無法同時取得socket input進來的資料
- 解決方法: 用**select**或**poll**同時等待兩個inputs (以後會介紹)

`SIGPIPE` Signal

- When a process writes to a socket that has received an RST, the **`SIGPIPE`** signal is sent to the process
  - The default action of this signal is to terminate the process
- If the process either catches the signal and returns from the signal handler, or (uses **`sigaction`** to) ignore the signal, the **`write`** operation returns negative; **`errno = EPIPE`**

SIGPIPE Signal Example



Crash of Server Host (主機)

- Server host shuts down, powers off, or disconnects
- Not acknowledged by the server host, client TCP continuously retransmits data and timeouts around 9 min
- *readline* returns negative; `errno` = ETIMEDOUT or EHOSTUNREACH
- To quickly detect: timeout on *readline*, SO_KEEPALIVE socket option, heartbeat functions

Reboot of Server Host

- The client does not see the server host shutting down
- Client sends data to server after the server reboots
- server TCP responds to client with an RST because it loses all connection information
- **`readline`** returns negative; **`errno`** = **`ECONNRESET`**

Shutdown of Server Host (by Operator)

- **`init`** process sends **`SIGTERM`** to all processes
  - We can catch this signal and close all open descriptors by ourselves (gracefully)
- **`init`** waits 5-20 sec and sends SIGKILL to all processes
  - all open descriptors are closed by kernel

Summary of TCP Example

- From client's perspective:
  - **`socket`** and **`connect`** specifies server's port and IP address
  - Client's port and IP address are chosen by TCP and IP respectively

- From server's perspective:
  - **`socket`** and **`bind`** specifies server's local port and IP address
  - **`listen`** and **`accept`** return client's port and IP address

Data Format: Text Strings

- server process gets two numbers (in a line of text) from client and returns their sum
- In **`str_echo`**: **`sscanf`** converts string to long integer, **`snprintf`** converts long back to string

Data Format: Binary Structure

- Passing binary structure between client and server does not work
  - when the client and server are run on hosts with different byte orders (Intel vs. IBM) or sizes of long integer (32-bit vs. 64-bit)
- Suggestions:
  - pass in string only (in this course)
  - explicitly define the format of data types (e.g. RPC's XDR -- external data representation)