Concurrent Server: TCP Service

Concurrent Server: Processes

- Consider concurrent server using multiple processes: the simplest way
- Spawn new process for each new client to handle the client request

fork

```
#include <unistd.h>

pid_t fork(void);
```

- Called *once* but returns *twice*
- OS creates a copy of parent process which becomes the child
- Returns child PID to parent and 0 to child
  - Child only has one parent
  - Child calls getppid() to get parent PID

fork (2)

- Open descriptors in parent before fork are shared with child
- Two typical uses:
  - The new process does other tasks: e.g., network concurrent servers
  - A process wants to execute another program: used together with exec system call

exec

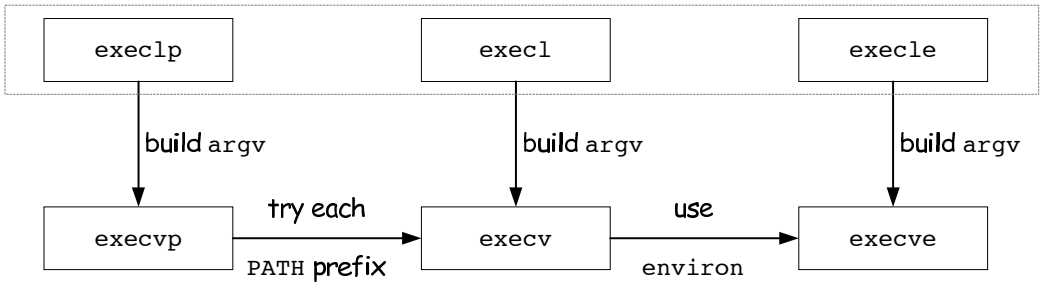
- Only way an executable file on disk can be run by Unix
- Replace the current process image with the new program file; **process ID stays the same**
- Has 6 different flavors
- Differences are:
  - Whether the program file is given by filename or pathname
  - Whether the program arguments listed one by one or referenced through an array of pointers
  - Whether the environment of the calling process is passed to the new program or a new one is used

exec (2)

```
#include <unistd.h>
extern char **environ;
int execl (const char *pathname, const char *argo, ... /* (char *) o */ );
int execle (const char *pathname, const char *argo, ...
            /* (char *) o, char *const envp[] */ );
int execlp (const char *filename, const char *argo, ... /* (char *) o */ );
int execv (const char *pathname, char *const argv[]);
int execve (const char *pathname, char *const argv[], char *const envp[]);
int execvp (const char *filename, char *const argv[]);
```

- List of parameters or an array (l vs. v)
- Use of PATH variable with filename (p)
- Use of environment variable (extern char**) or parameter (e)

exec (3)



- Top row calls specify each argument separately
- Two calls on left specify filename
  - Use PATH variable unless filename has '/'
- Left two column calls do not specify envp, use external variable (extern char **environ)

exec (4)

- The new process inherits many attributes from the calling process
- E.g., from the man page on Linux 4.19, all process POSIX.1 attributes are preserved *except* the following:
  - The **dispositions of any signals** that are being caught are reset to the default (signal(7)).
  - Any alternate signal stack is not preserved (sigaltstack(2)).
  - **Memory mappings** are not preserved (mmap(2)).
  - Attached System V shared memory segments are detached (shmat(2)).
  - POSIX **shared memory** regions are unmapped (shm_open(3)).
  - Open POSIX message queue descriptors are closed (mq_overview(7)).
  - Any open POSIX **named semaphores** are closed (sem_overview(7)).
  - POSIX timers are not preserved (timer_create(2)).
  - Any open directory streams are closed (opendir(3)).
  - Memory locks are not preserved (mlock(2), mlockall(2)).
  - Exit handlers are not preserved (atexit(3), on_exit(3)).
  - …

Fork/Exec (e.g.)

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
   int p;

   p = fork();
   if (fork()==0) {
      if (execl("/bin/echo", "/bin/echo", "foo", 0) == -1) {
         fork();
      }
   } else {
      execl("/bin/echo", "/bin/echo", "bar", 0);
      printf("baz\n");
   }
}
```

Concurrent Servers

- Iterative server: tie up a single server with one client
- Using fork, a child process can handle each client
- For example, establish connection with accept and then call fork

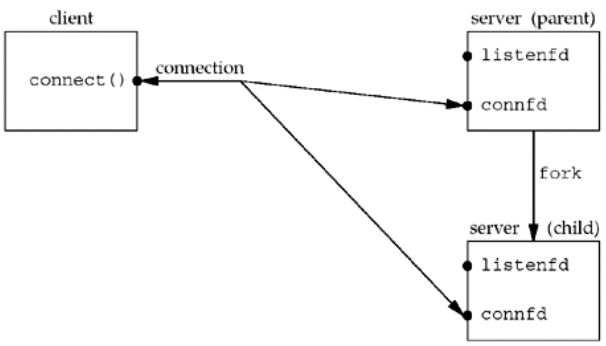


Fig. 4.13

```
pid_t pid;
int   listenfd,  connfd;

listenfd = Socket( ... );

   /* fill in sockaddr_in{} with server's well-known port */
Bind(listenfd, ... );
Listen(listenfd, LISTENQ);

for ( ; ; ) {
   connfd = Accept (listenfd, ... );    /* probably blocks */

   if( (pid = Fork()) == 0) {
      Close(listenfd);    /* child closes listening socket */
      doit(connfd);       /* process the request */
      Close(connfd);      /* done with this client */
      exit(0);            /* child terminates */
   }

   Close(connfd);       /* parent closes connected socket */
}
```

close? => FIN?

***

Signals

- Also known as software interrupts
- Notification to a process that an event has occurred
- Asynchronous in nature

Posix Signal Handling

- A *signal* is a notification to a process that an event has occurred
  - AKA software interrupt
  - Occurs *asynchronously* w/o prior knowledge
- Signals can be sent
  - By one process to another process or to itself
  - By the kernel to a process
- E.g., SIGCHLD signal
  - Sent to the parent of the terminating process
  - Sent by the kernel when a child terminates

Disposition AKA Action

- Every signal has a *disposition*
  - I.e., action associated with the signal
- Set disposition by calling the sigaction()

Disposition

Three choices for the disposition:

1. Call a function (*signal handler*) whenever a specific signal occurs
   - Function prototype: void handler(int signo)
   - This action is called *catching* the signal
2. Ignore a signal by setting its disposition to SIG_IGN
   - SIGKILL and SIGSTOP can't be ignored or caught
3. Set the *default* disposition for a signal by setting its disposition to SIG_DFL
   - Default is to terminate process on the receipt of a signal

Calling sigaction()

```
/* include signal */
#include        "unp.h"
/* typedef  void Sigfunc(int); */

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);
} /* end signal */
```

```
struct  sigaction {
        union __sigaction_u __sigaction_u;  /* signal handler */
        sigset_t sa_mask;       /* signal mask to apply */
        int     sa_flags;       /* see signal options below */
};

union __sigaction_u {
        void    (*__sa_handler)(int);
        void    (*__sa_sigaction)(int, siginfo_t *, void *);
};

#define sa_handler      __sigaction_u.__sa_handler
#define sa_sigaction    __sigaction_u.__sa_sigaction
```

Fig. 5.6

Posix Signal Semantics

- Once a signal handler is installed, it remains installed
- While a signal handler is executing, the signal being delivered is blocked
- If a signal is generated one or more times while it is blocked, it is normally delivered only one time after the signal is unblocked
- By default, Unix signals are not *queued*
- It is possible to selectively block and unblock a set of signals using the sigprocmask() function

Signal Mask Functions

- Posix allows to specify a set of signals that will be *blocked* when our signal handler is called
- Any blocked signal *cannot be delivered* to the process
- Set signal mask for handler

```
#include <signal.h>
int sigaddset(sigset_t *set, int signo);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigdelset(sigset_t *set, int signo);

int sigismember(sigset_t *set, int signo);
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
      /* how: SIG_BLOCK, SIG_SETMASK, SIG_UNBLOCK */
```

Interrupted System Calls

- "Slow system call" : accept, read/write IO
- Return an error code EINTR if all occur:
  - Blocked in slow system call
  - Process catches a signal
  - Signal handler returns
- Portability issues: support of POSIX flag SA_RESTART flag is optional
  - Some OS's use non-portable flags
  - Not all interrupted systems calls may automatically be restarted by the kernel

Handling SIGCHLD Signals

- Zombie state
  - Maintain information about the child for the parent to fetch at some later time
  - PID of the child, termination status, resource utilization of the child (CPU time, memory etc.)
  - <defunct> shown in COMMAND column in some Unix systems
- Handling zombies
  - Clean them up! To clear space in kernel; otherwise, can run out of processes
  - After fork(), parent must wait for the children to prevent them from becoming zombies

Set Signal Handler (e.g.)

```
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));
        Listen(listenfd, LISTENQ);
        Signal(SIGCHLD, sig_chld);

        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 */
        }
}
```

Fig 5.2 tcpserv01.c

Handling SIGCHLD Signals

```
#include   "unp.h"
void sig_chld(int signo)
{
    pid_t  pid;
    int    stat;
    pid = wait(&stat);
    printf("child %d terminated\n", pid);
    return;
}
```

- We establish the signal handler by adding it after the calling listen (before accept):
  - signal(SIGCHLD, sig_chld)

```
hi, there                          // we type this
hi, there                          //this is echoed
^D                                 //EOF
child 16942 terminated             //output  in signal handler
accept error: Interrupted system call   // function aborts
```

- E.g., running tcpserv01 (with signal handling) and tcpcli01 locally
- SIGCHLD interrupts slow system call (accept)
  - Call returns with an error, setting errno EINTER
  - Should handle this return

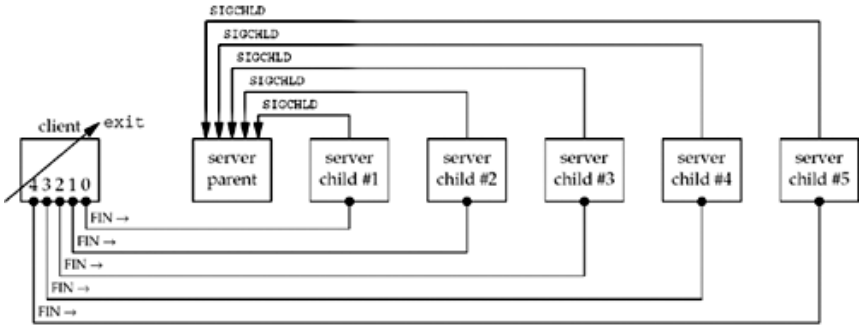
wait() and waitpid()

```
#include <sys/wait.h>

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int option);
```

- *pid_t* : the process ID of the terminated child
- *statloc* : the termination status of the child (an integer) is returned through the *statloc* pointer
- *pid* : specify the process ID that we want to wait for
  - -1 says to wait for the first of our children to terminate
- *option* : specify additional option
  - The most common option is WNOHANG, which tells kernel not to block if there are no terminated children
- waitpid() gives more control over which process to wait for and whether or not to block
  - *pid* or -1 for the first of our children to terminate

Multiple Signals Example

- Client establishes multiple connections with concurrent server
- Client terminates, all open descriptors closed automatically
  - Client only calls `exit` and not `close`
- All five connections terminate at about same time
  - Five FINs sent on each connection
  - This causes five SIGCHLD signals to be delivered to parent
- Delivery of multiple occurrences of same signal causes problems!



Zombie Processes (e.g.)

>**tcpserv03** &
>**tcpcli03 206.62.226.35**
hello
hello //*echoed*
^D
child 21288 terminated

Execute `ps`:

```
PID  TTY  TIME  CMD
20419  pts/6   00:00:00 tcpserv03
20421  pts/6   00:00:00 tcpserv03  <defunct>
20422  pts/6   00:00:00 tcpserv03  <defunct>
20423  pts/6   00:00:00 tcpserv03  <defunct>
```

`wait()` and `waitpid()`

- Establishing signal handler and calling wait() is insufficient to prevent zombies!
- If all five signals generated before signal handler executed, signal handler only executed once, since signals are not queued
- Furthermore, this problem is nondeterministic!
  - In this example, same host so handler is executed once, leaving 4 zombies
  - On different hosts, handler executed 2 times…. 3 zombies
- Solution to problem is to call `waitpid()` instead of `wait()`

waitpid()

- Difference between wait() and waitpid()
  - Since Unix signals are normally not queued
- Must specify the `WNOHANG` option to tell it not to block if there exist running children that have not yet terminated

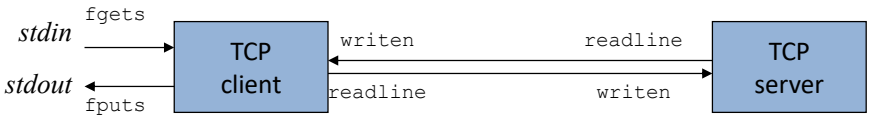
```
void sig_chld(int signo)
{
    pid_t   pid;
    int stat;

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

SIGCHLD Summary

- We must catch the `SIGCHLD` signal when forking child processes!
  - I.e., using concurrent servers
- We must handle interrupted system calls when we catch signals
- `SIGCHLD` handler must be coded correctly using waitpid() to prevent any zombies from being left!

***

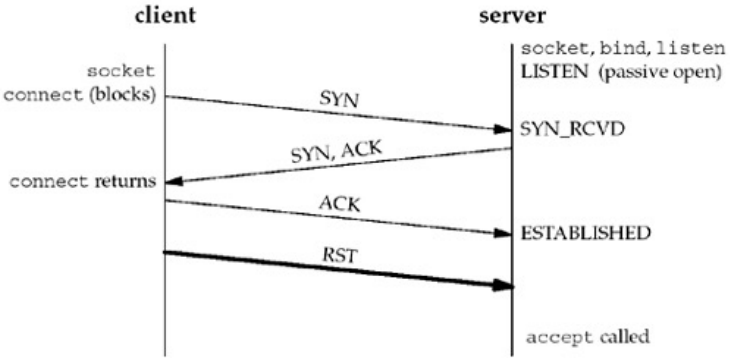
Client-Server Example



- Client reads a line of text from its standard input and writes the line to the server
- Server reads the line from its network input and echoes the line back to the client
- Client reads the echoed line from the server and prints it on standard output

Connection Abort before accept Returns

- The three-way handshake completes, the connection is established, and then the client TCP sends an RST (reset)
- On the server side the connection is queued by its TCP, waiting for the server process to call accept when the RST arrives
- Some time later the server process calls accept()



accept() and Receiving RST

Handling is implementation-dependent:

- BSD: hidden from the application in the kernel
- SVR4: return an errno of EPROTO
- Posix.1g : return an errno of **ECONNABORTED**
- EPROTO error: returned when some fatal protocol-related events occur on the streams subsystem
- ECONNABORTED error: the server can ignore the error and just call accept again ☺

Termination of Server Process

```
solaris % tcpcli01 206.62.226.35
hello
hello

another line
str_cli: server terminated prematurely
```

- Kill server child after hello
- The client is not expecting to receive an EOF at this point, so it quits with the error message "server terminated prematurely"

Read/Write (e.g.)

```
 1 #include    "unp.h"

 2 void
 3 str_cli(FILE *fp, int sockfd)
 4 {
 5     char    sendline[MAXLINE], recvline[MAXLINE];
 6     while (Fgets(sendline, MAXLINE, fp) != NULL) {  // blocked on stdin; gets FIN
 7         Writen(sockfd, sendline, strlen (sendline));  //
 8         if (Readline(sockfd, recvline, MAXLINE) == 0)
 9             err_quit("str_cli: server terminated prematurely");
10         Fputs(recvline, stdout);
11     }
12 }
```

RST arrives?

RST could arrive before or after Readline()

Example shows Readline() called before RST received, so read gets EOF (i.e., FIN from server termination)

Fig 5.5 str_cli.c

SIGPIPE Signal

- What happens if write to a socket whose TCP peer is gone
- First write elicits the RST, additional writes get SIGPIPE signal to the process
- When a process writes to a socket that has received an RST, the SIGPIPE signal is sent to the process
  - Default action of SIGPIPE → terminate the process
- If catching signal or ignoring it, write operation returns EPIPE
  - With multiple sockets, SIGPIPE will not inform about which socket caused the error
  - Useful if we need to know which write cause the error

SIGPIPE (e.g.)

```
>tcpcli11 206.62.226.34
hi there
hi there
                        // kill server child process
Bye
Broken pipe     // printed by our shell
                // client process died with SIGPIPE
```

Nothing is echoed for bye data

Reason:

The default action of SIGPIPE is terminate the process.

Read/Write (e.g.)

```
 1 #include     "unp.h"

 2 void
 3 str_cli(FILE *fp, int sockfd)
 4 {
 5     char    sendline [MAXLINE], recvline [MAXLINE];

 6     while (Fgets(sendline, MAXLINE, fp) != NULL) {
 7         Writen(sockfd, sendline, 1);  // 1st write gets RST
 8         sleep(1);
 9         Writen(sockfd, sendline + 1, strlen(sendline) - 1);  // 2nd write gets sig

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

12         Fputs(recvline, stdout);
13     }
14 }
```

Example elicits a RST from the server, before SIGPIPE.

Fig 5.14 str_cli11.c

***