

# LEC 05

# CONCURRENT TCP SERVER

Bui Trong Tung, SoICT, HUST

1

1

## Content

- I/O Models
- Concurrent TCP server: one child per client
- Concurrent TCP server: one thread per client

2

2

1

# I/O MODELS

---

3

3

## Review TCP Echo Server

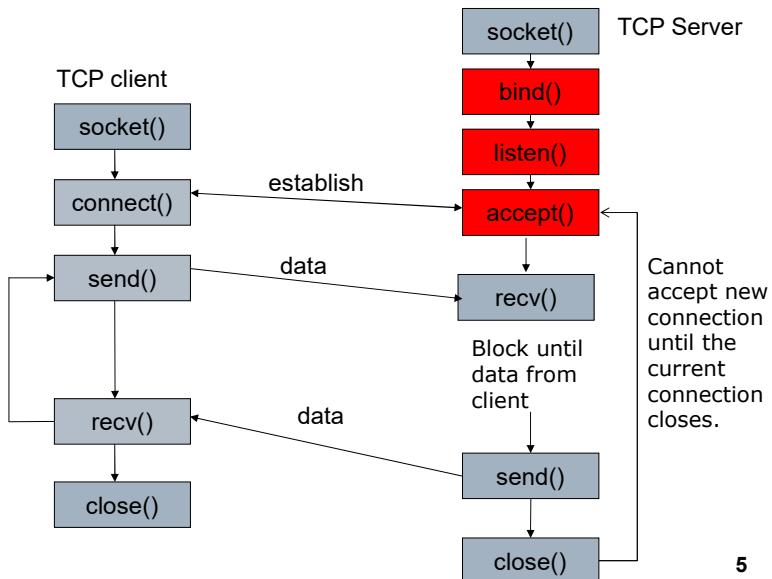
```
while(1) {
    //accept request
    connfd = accept(listenfd, (sockaddr *) & clientAddr,
                    &clientAddrLen);
    //receive message from client
    rcvBytes = recv(connfd, buff, BUFF_SIZE, 0);
    if(rcvBytes < 0){
        perror("Error :");
    }
    else{
        buff[rcvBytes] = '\0';
        printf("Receive from client: %s\n",buff);
        //Echo to client
        sendBytes = send(connfd, buff, strlen(buff), 0);
        if(sendBytes < 0)
            perror("Error: ");
    }
    closesocket(connfd);
} //end while
```

4

4

2

## Blocking I/O Model and TCP server



5

## I/O Models

- blocking I/O
- nonblocking I/O
- I/O multiplexing (select and poll)
- signal driven I/O (SIGIO)
- asynchronous I/O (the POSIX aio\_functions)

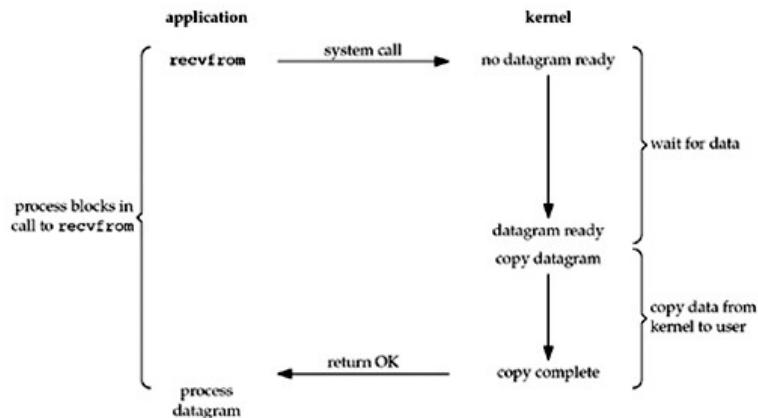
6

6

3

## Blocking I/O Model

- Blocking I/O model: I/O function block process/thread until returning.
- `accept()`, `connect()`, `send()`, `recv()`, ...

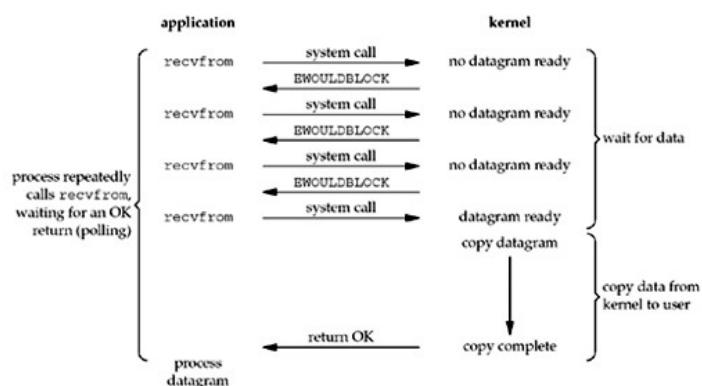


7

7

## Non-blocking I/O Model

- Non-blocking I/O model: I/O function returns immediately
- If there is no data to return, so the kernel immediately returns an error of `EWOULDBLOCK` instead

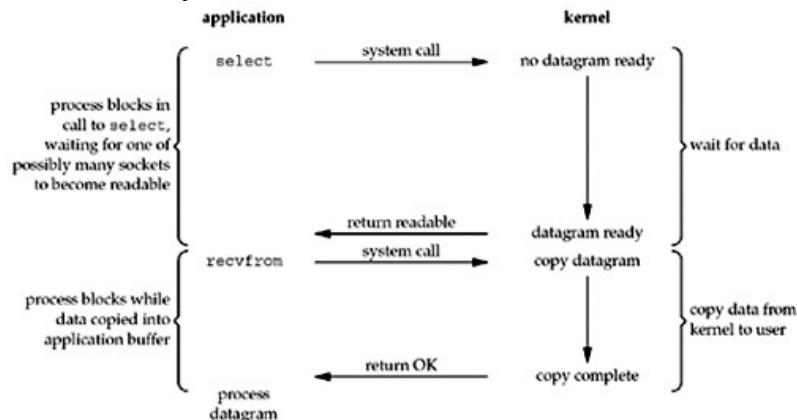


8

8

## I/O Multiplexing Model

- With I/O multiplexing, we call select or poll and block in one of these two system calls, instead of blocking in the actual I/O system call

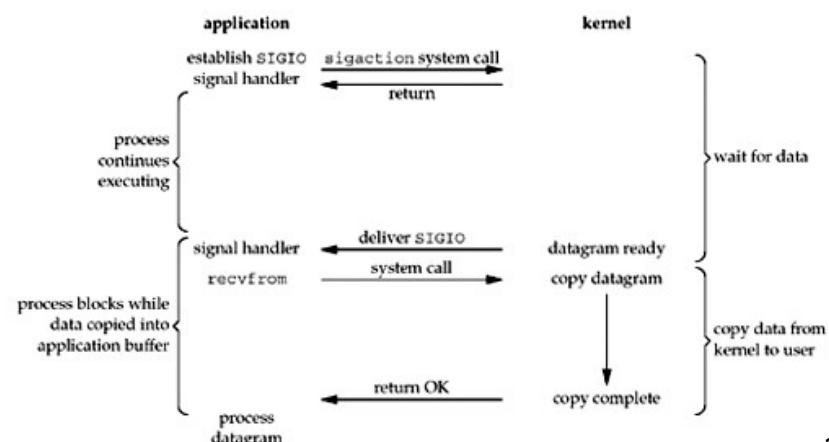


9

9

## Signal-Driven I/O Model

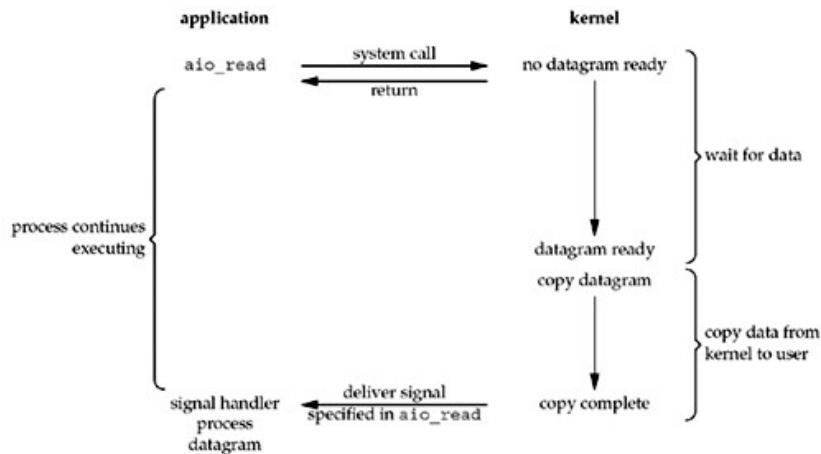
- Use signals, telling the kernel to notify app with the SIGIO signal when the descriptor is ready



10

10

## Asynchronous I/O Model



11

11

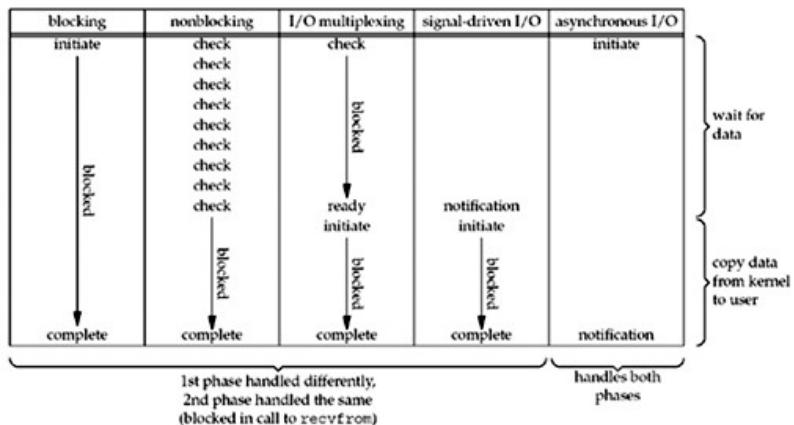
## Asynchronous I/O Model (2)

- App calls `aio_read` (the POSIX asynchronous I/O functions begin with `aio_`)
- Pass the kernel
  - the descriptor
  - buffer pointer
  - buffer size (the same three arguments for `read`)
  - buffer offset (similar to `lseek`)
  - how to notify us when the entire operation is complete
- This system call returns immediately and our process is not blocked while waiting for the I/O to complete.

12

12

## Comparison of the I/O Models



13

13

## Iterating server

- Simple server
  - But when a client request can take longer to service, we can't handle other clients
- Use a *concurrent server*
- One child per client: `fork()` function spawns one child process to handle each client
  - One thread per client: `pthread_create()` creates one thread to handle each client

14

14

# MULTI-PROCESS SERVER

---

15

15

## fork()

```
#include <unistd.h>
pid_t fork(void);
```

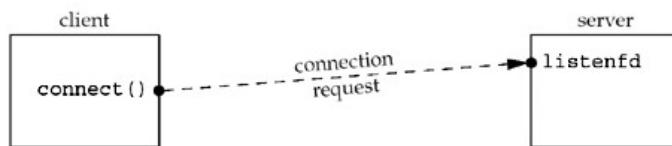
- Create a new process by copying itself.
- Returns twice:
  - Once in the calling process (called the parent) with a return value that is the process ID of the newly created process (the child).
  - Once in the child, with a return value of 0
- All descriptors open in the parent before the call to fork are shared with the child after fork returns

16

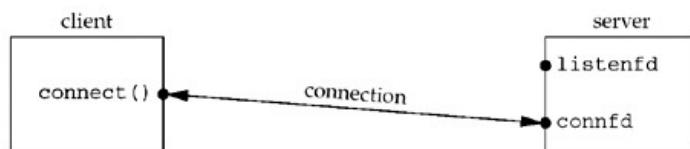
16

## One child per client

(1) LISTENING



(2) accept()

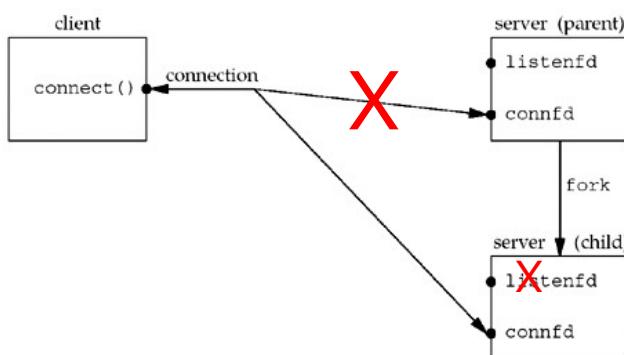


17

17

## One child per client

(3) fork()



18

18

## Use fork()

```
pid_t pid;
int listenfd, connfd;
//Step 1: Construct socket
//Step 2: Bind address to socket
//Step 3: Listen request from client

//Step 4: Communicate with client
while (1) {
    connfd = accept (listenfd, ... );
    if( (pid = fork()) == 0) { // process in child
        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
}
```

19

19

## Handling SIGCHLD Signals

- When a child process ends, it sends the **SIGCHLD** signal to the parent
  - Information about the child process is still maintained in “process table” in order to allow its parent to read the child exit status afterward.
  - If we ignore the **SIGCHLD**, the child process will enter the zombie state
  - We need to wait and handle **SIGCHLD** signal

20

20

## Signaling

- A signal is a notification to a process that an event has occurred.
- Signals are sometimes called software interrupts.
- Signals usually occur asynchronously. By this we mean that a process doesn't know ahead of time exactly when a signal will occur.
- Signals can be sent
  - By one process to another process (or to itself)
  - By the kernel to a process

21

21

## Signal (cont.)

- Typing certain key combinations at the controlling terminal of a running process causes the system to send it certain signals:
  - Ctrl-C sends an INT signal ("interrupt", SIGINT)
  - Ctrl-Z sends a TSTP signal ("terminal stop", SIGTSTP)
  - Ctrl-\ sends a QUIT signal (SIGQUIT)
- SIGHUP is sent to a process when its controlling terminal is closed (a hangup)
- SIGTERM is sent to a process to request its termination.
  - Unlike the SIGKILL signal, it can be caught and interpreted or ignored by the process.

22

22

## Handling SIGCHLD Signals

- The purpose of the zombie state is to maintain information about the child for the parent to fetch at some later time.
- They take up space in the kernel and eventually we can run out of processes
  - Whenever we *fork* children, we must *wait* for them to prevent them from becoming zombies → establish a signal handler to catch *SIGCHLD*, and within the handler, we call *wait*
  - Establish the signal handler by adding the function call :  
*signal (SIGCHLD, handler);*

23

23

## *wait()* and *waitpid()*

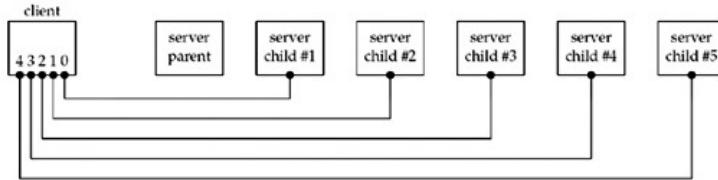
```
#include <sys/wait.h>
pid_t wait (int *statloc);
pid_t waitpid (pid_t pid, int *statloc, int options);
```

- Wait for the status change of a process.
- Use to handle the terminated child
- Both return two values:
  - The return value of the function:
    - the process ID of the terminated child
    - 0 or -1 if error
  - The termination status of the child (an integer) is returned through the *statloc* pointer.

24

24

## wait()

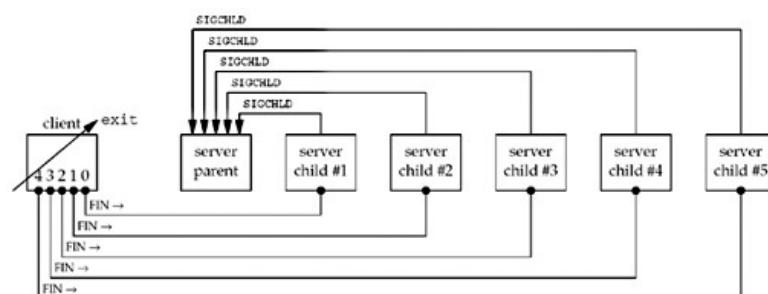


- Create 5 connections from a client to a forking server
- When the client terminates, all open descriptors are closed automatically by the kernel → five connections ended simultaneous

25

25

## wait()



- Client terminates, closing all five connections, terminating all five children → four children are zombies
- It can happen when many users connect to a server
- → we have to use **waitpid()**

26

26

## waitpid()

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

- pid < 0: wait for any child process whose process group ID is equal to the absolute value of *pid*.
- **pid = -1**: wait for any child process.
- pid = 0: wait for any child process whose process group ID is equal to that of the calling process
- pid > 0: wait for the child whose process ID is equal to the value of *pid*
- Without option WNOHANG, waitpid blocks until the status change
- With option WNOHANG, waitpid returns immediately
- Return
  - pid of the child whose state has changed
  - with option WNOHANG, return 0 if the specified process has not changed status.

27

27

## void sig\_chld(int signo)

```
void sig_chld(int signo)
{
    pid_t pid;
    int stat;
    while((pid = waitpid(-1, &stat, WNOHANG))>0)
        printf("child %d terminated\\", pid);
}
```

- WNOHANG: waitpid() does not block
- while loop: waitpid() repeatedly until there is no child process change status, i.e until waitpid returns 0.

28

28

## Forking server

```
pid_t pid;
int listenfd, connfd;
//Step 1: Construct socket
//Step 2: Bind address to socket
//Step 3: Listen request from client

// wait for a child process to stop
signal(SIGCHLD, sig_chld);
//Step 4: Communicate with client
while (1) {
    connfd = accept (listenfd, ... );
    if( (pid = fork()) == 0) {// process in child
        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
}
```

29

29

## Handling EINTR Errors

- When a process is blocked in a **slow system call** and the process catches a signal and the signal handler returns, the system call can return an error of EINTR.
- Slow system call: connect, accept, send, recv...
- Not all kernels automatically restart some interrupted system calls
- We must rewrite function to handle EINTR error

```
while(1)
    if ((connfd = accept (listenfd...)) < 0) {
        if (errno == EINTR)
            continue; /* back to for () */
        else
            perror ("Error: ");
    }
```

30

30

## Other problems

- Connection abort before `accept` return
- Termination of server process
- Crashing of sever host
- Crashing and Reboot of server host

31

31

## MULTI-THREAD SERVER

---

32

32

## pthread\_create()

```
#include <pthread.h>
int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                  void *(*routine) (void *), void *arg);
```

- Create a new thread
- Parameters:
  - [OUT] tid: points to ID of the new thread
  - [IN] attr : points to structure whose contents are used to determine attributes for the new thread
  - [IN] routine: the new thread starts execution by invoking routine()
  - [IN] arg: points to the argument is passed as the sole argument of routine()
- Return:
  - On success, returns 0
  - On error, returns an error number
- **Compile and link with -pthread**

33

33

## pthread\_create()

- By default, the new thread is joinable:
  - Not automatically cleaned up by GNU/Linux when it terminates
  - the thread's exit state hangs around in the system until another thread calls pthread\_join() to obtain its return value
- Detached thread is cleaned up automatically when it terminates
  - Another thread may not obtain its return value
- **Detach a thread:** `int pthread_detach(pthread_t tid)`
  - On success, returns 0
  - On error, returns an error number

34

34

```
main(){
    pthread_create(tid,...);
    pthread_join(tid);
    return 0;
}
```

35

35

## Multi-thread TCP Echo Server

```
pthread_t tid;
int listenfd, *connfd;
//Step 1: Construct socket
//Step 2: Bind address to socket
//Step 3: Listen request from client

//Step 4: Communicate with client
while (1) {
    connfd = malloc(sizeof(int));
    *connfd = accept (listenfd, ... );
    pthread_create(&tid, NULL, &client_handler, connfd);
}

close(listenfd);
return 0;
```

36

36

## Multi-thread TCP Echo Server(cont.)

```
void *client_handler(void *arg) {
    int connfd;
    int sendBytes, rcvBytes;
    char buff[BUFF_SIZE + 1];

    pthread_detach(pthread_self());
    connfd = *((int *) arg);
    while(1) {
        rcvBytes = recv(connfd, buff, BUFF_SIZE, 0);
        if (rcvBytes <= 0)
            break;

        sendBytes = send(connfd, buff, rcvBytes, 0);
        if (sendBytes <= 0)
            break;
    }
    close(connfd);
}
```

37

37

## Synchronize threads

- Since multiple threads can be running concurrently, accessing the shared variables:
  - The order of the accessing shared memory is unpredictable, so
  - The processing flow of the thread may be uncontrollable, and/or
  - The process crash
- Synchronize threads so that only one thread can access shared memory:
  - Mutex
  - Condition variable
  - Inter-lock

38

38

## Mutex

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t * mptr);
int pthread_mutex_unlock(pthread_mutex_t * mptr);
```

- The thread can access the shared variable only when it hold the mutex
  - `pthread_mutex_lock()`: lock a mutex
  - `pthread_mutex_unlock()` : unlock a mutex
  - If the thread try to lock a mutex that is already locked by some other thread, it is blocked until the mutex is unlocked.

```
void *routine(void *arg) {
    //...
    pthread_mutex_lock(mptr);
    // access shared memory
    pthread_mutex_unlock(mptr);
//...
}
```

39

39

## Condition variable

- Mutexes are for **locking** and cannot be used for **waiting**
- Condition variable can be used to **waiting**

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond,
                     pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

- `pthread_cond_wait()` : wait on a condition
- `pthread_cond_signal()` : signal a condition

40

40

## Mutex + Condition variables template

```
struct {
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    whatever variables maintain the condition
} var = (PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER);
```

```
/* on signal thread */
pthread_mutex_lock(&var.mutex);
set condition true
pthread_cond_signal(&var.cond);
thread_mutex_unlock(&var.mutex)
```

```
/* on waiting thread */
pthread_mutex_lock(&var.mutex);
while (condition is false)
    pthread_cond_wait(&var.cond, &var.mutex);
modify condition
pthread_mutex_unlock(&var.mutex)
```

41

41

## fork() VS pthread\_create()

### fork()

- Heavy-weight
- Passing information from the parent to the child before the fork is easy
- Returning information from the child to the parent takes more work
- Needn't synchronize processes
- Greater isolation between the parent and the child

### pthread\_create()

- Light-weight
- Passing information from a thread to the others is easy
- Don't need signal-driven processing when the threads ends.
- May synchronize threads
- If a thread crashes, process may crash

42

42