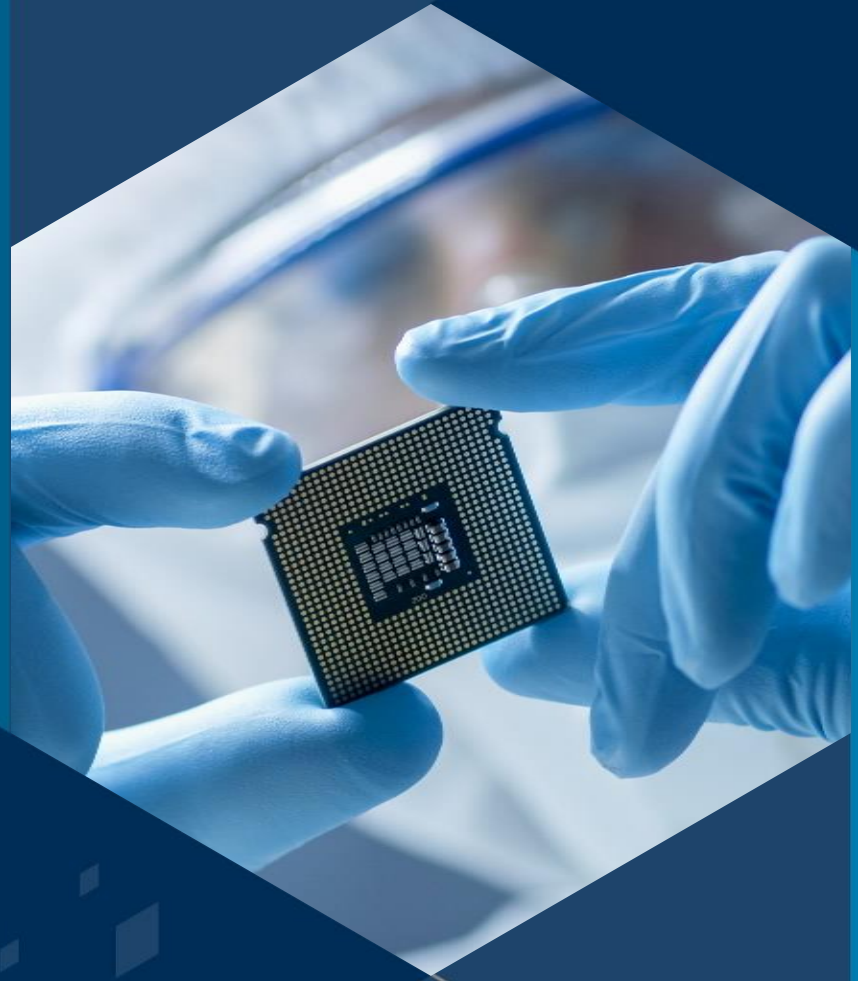
Tutorial 3
COMP4621

Concurrent Client Server Using fork



Spring 2024

香港科技大學
THE HONG KONG
UNIVERSITY OF SCIENCE
AND TECHNOLOGY

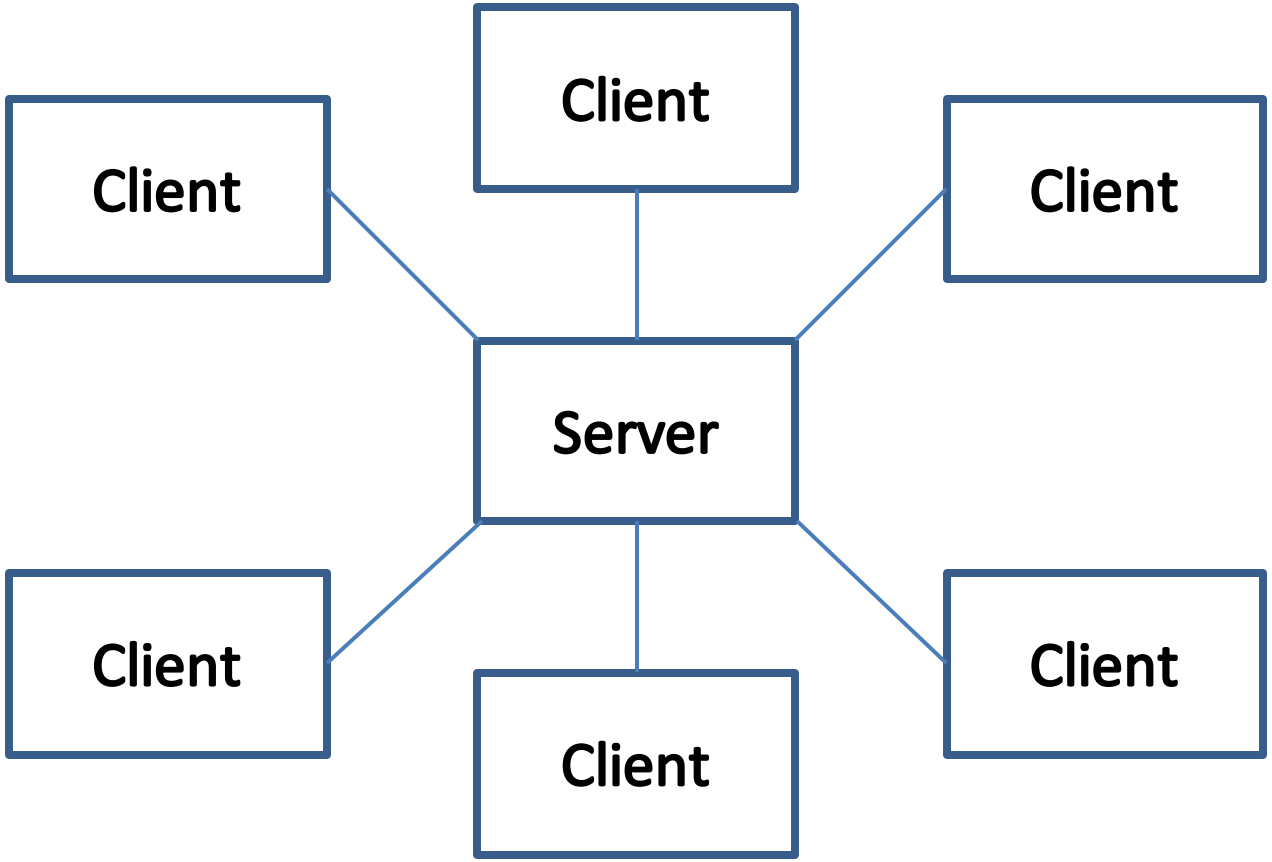
TA: Xinyu YANG
Email: xinyu.yang@connect.ust.hk

Outline

- Understanding the multiple TCP client server problem
- Review fork() function
- Example of concurrent TCP client server problem
- Set timeout using setsockopt() function

Understanding the multiple client server problem

- The figure shows several clients, which have contacted one server. The client connections conceptually form spokes around the central server.



Ways to achieve concurrent client server

- Forked server processes (multi-process method)
- Threaded server processes (multi-thread method)
- One process and a `select(2)` call
- One process and a `poll(2)` call

Comparison of the four ways

Method	Advantages	Disadvantages
Forked server process	• Simplest	• Sharing information becomes more complex. • Requires more CPU to start and manage a new process for each request
Threaded server process	• Lightweight advantages of the multi-process method	• Difficult to debug

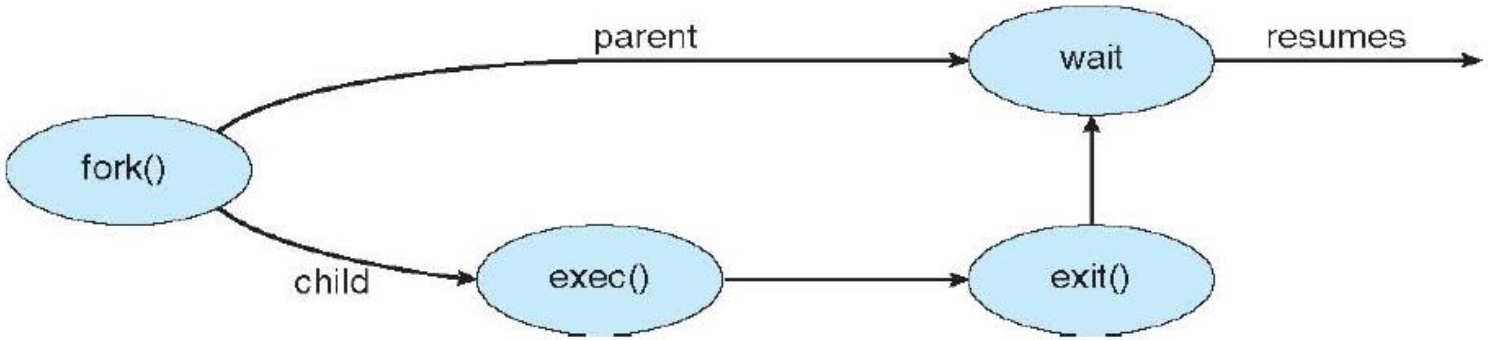
- Select(2) and poll(2) system calls offer a different way to block execution of the server until an event occurs (usually receiving data on a socket)

Review of fork()

- fork – create a child process
- SYNOPSIS

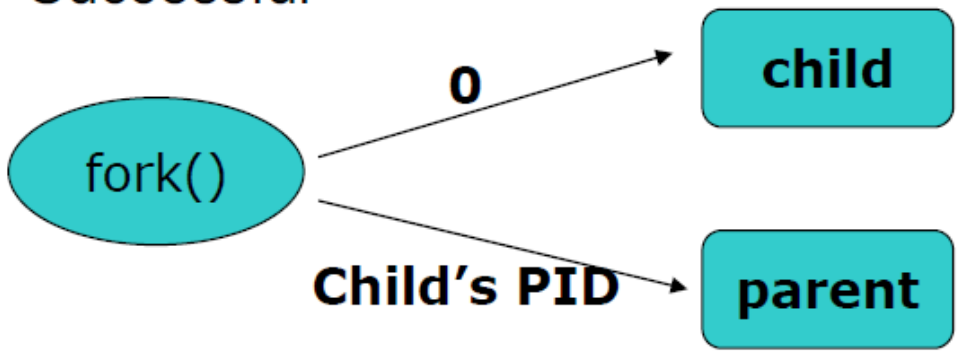
```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void)
```

- DESCRIPTION:
  - fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.
  - The child process and the parent process run in separate address spaces (i.e, they do not share their data).

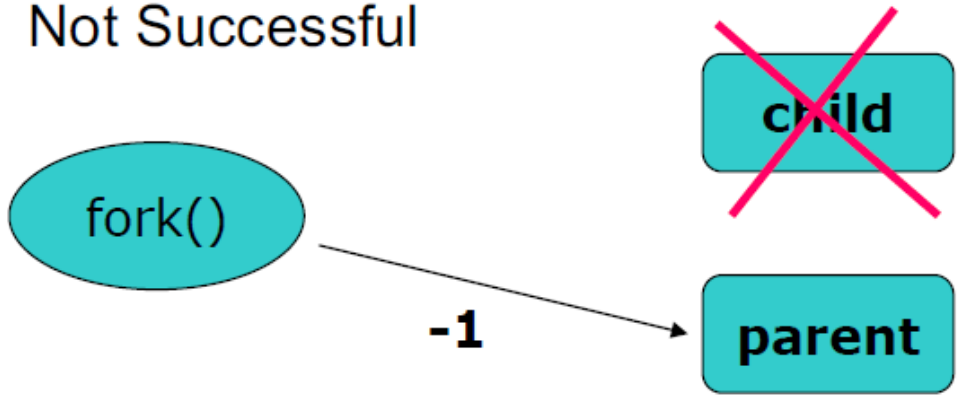


Return values of fork

- Successful



- Not Successful



errno is set to indicate error

fork example

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

#define BUFSIZE 1024
int main(int argc, char *argv[])
{
  char buf[BUFSIZE];
  size_t readlen, writelen, slen;
  pid_t cpid, mypid;
  pid_t pid = getpid();          /* get current processes PID */
  printf("Parent pid: %d\n", pid);
  cpid = fork();
  if (cpid > 0) {                    /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
  }  else if (cpid == 0) {          /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
  } else {
    perror("Fork failed");
    exit(1);
  }
  exit(0);
}
```

fork example

```
cpid = fork();
if (cpid > 0) {        /* Parent Process */
   mypid = getpid();
   printf("[%d] parent of [%d]\n", mypid, cpid);
} else if (cpid == 0) {  /* Child Process */
   mypid = getpid();
   printf("[%d] child\n", mypid);
} else {
   perror("Fork failed");
   exit(1);
}
```

Parent Process (cpid=childpid)

```
cpid = fork();
if (cpid > 0) {        /* Parent Process */
   mypid = getpid();
   printf("[%d] parent of [%d]\n", mypid, cpid);
} else if (cpid == 0) {  /* Child Process */
   mypid = getpid();
   printf("[%d] child\n", mypid);
} else {
   perror("Fork failed");
   exit(1);
}
```

Child Process (cpid=0)

```
cpid = fork();
if (cpid > 0) {        /* Parent Process */
   mypid = getpid();
   printf("[%d] parent of [%d]\n", mypid, cpid);
} else if (cpid == 0) {  /* Child Process */
   mypid = getpid();
   printf("[%d] child\n", mypid);
} else {
   perror("Fork failed");
   exit(1);
}
```

wait() and waitpid()

- **pid_t wait(int *_status_);**
- **pid_t waitpid(pid_t** *pid*, **int *_status_, int** *options*);

- Description:
  - The **wait**() system call suspends execution of the calling process until one of its children terminates
  - The **waitpid**() system call suspends execution of the calling process until a child specified by *pid* argument has changed state.
- Arguments:
  - PID can be <-1 meaning wait for any child process whose process group ID is equal to the absolute value of *pid*.; -1 meaning wait for any child process;, 0 meaning wait for any child process whose process group ID is equal to that of the calling process; >0 meaning wait for the child whose process ID is equal to the value of *pid*.
  - OPTIONS can be 0, WNOHANG, … refer to https://linux.die.net/man/2/waitpid.
- Return value:
  - **wait**(): on success, returns the process ID of the terminated child; on error, -1 is returned.
    waitpid(): on success, returns the process ID of the child whose state has changed;
    if WNOHANG was specified and one or more child(ren) specified by *pid* exist, but have not yet changed state, then 0 is returned. On error, -1 is returned.

Example of multiple client server

- Clients:
  - User types line of text
  - Client program sends line to server
- Server:
  - Server receives line of text
  - Capitalizes all the letters
  - Sends modified line to client
- Clients:
  - Receives line of text
  - Displays

Skeleton of multiple client server using fork()

```
// Create a TCP/IP socket
listener = socket(PF_INET, SOCK_STREAM, 0);
// Bind the server address:
res = bind(listener, (struct sockaddr *)&server_addr, sizeof(server_addr));
// Make a listening socket:
res = listen(s, 10);
```

Skeleton of multiple client server using fork()

```
// Start the server loop
for (;;) {
        // Wait for a connect
        client_socket = accept(listener, (struct sockaddr *)&client_addr, sizeof(client_addr));
         // Fork a new server process to service this client
       if ((PID = fork()) == -1) {
            // Failed to fork: Give up
            close(client_socket);
            continue;
        } else if (PID > 0) { /* The parent process */
            // Parent process … so does not need the client socket anymore
            close(client_socket);
            continue;
        }
        // Child process we can close the server socket s
        close(listener); /* The child process */
        /*Exchange data with the Client*/
        // TODO
        // Close the client socket
        close(client_socket);
        // Child process must exit:
        exit(0);
    }
```

Skeleton of multiple client server using fork()

```
static void
sigchld_handler(int signo){
    pid_t PID;
    int status;

    do {
        PID = waitpid(-1, &status, WNOHANG);
    } while (PID != -1);

    /* Re-instate handler*/
    signal(SIGCHLD, sigchld_handler);
}
```

- Understanding Process Termination Processing
- 1. The signal `SIGCHLD` is sent by the kernel to the parent upon termination of its child process
- Normally SIGCHLD signal is ignored by default
- 2. But we can register a signal handler to process it: `sigchld_handler()` is called, because the function was registered for the `SIGCHLD` signal (“signal(SIGCHLD, sigchld_handler)”)
- 3. The `sigchld_handler()` executes a loop calling `waitpid(2)` until no more exit status
- information is available.
- 4. The `SIGCHLD` handler is re-instated

There’s a problem herein. Find and modify it in your code.

Test the result

```
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$ ./cc_server
Start communicating with client, IP address is: 127.0.0.1, port is: 35708
Start communicating with client, IP address is: 127.0.0.1, port is: 48742
From client: asdf
To client: ASDF
From client: lkjh
To client: LKJH
```

```
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$ ./cc_client
Socket successfully created...
Connected to the server...
Enter the string: asdf
From Server : ASDF
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
```

```
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$ ./cc_client
Socket successfully created...
Connected to the server...
Enter the string: lkjh
From Server : LKJH
[xyangcp@csl2wk31 ~/comp4621/tutorial3]$
```

Non-blocking sockets with timed I/O

```
#include <sys/types.h>
#include <sys/socket.h>
int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);
```

- All the socket information is in the man page for socket in section 7. (Type: “man 7 socket” to get all the information)
- Arguments:
  - s is the socket you’re talking about
  - level should be set to SOL_SOCKET
  - optname is the name of the option you’re interested in changing
  - optval is usually a pointer to an int indicating the value in question. For booleans, zero is false, and non-zero is true
  - optlen should be set to the length of optval, probably sizeof(int), but varies depending on the option.
- Return values
  - Returns zero on success, or -1 on error (and errno will be set accordingly)

Non-blocking sockets with timed I/O

- Socket options
  - SO_RCVTIMEO and SO_SNDTIMEO
    - specify the receiving or sending timeouts until reporting an error.
    - The argument is a struct timeval.
    - If an input or output function blocks for this period of time, and data has been sent or received, the return value of that function will be the amount of data transferred; if no data has been transferred and the timeout has been reached, then -1 is returned with errno set to EAGAIN or EWOULDBLOCK, or EINPROGRESS just as if the socket was specified to be nonblocking. If the timeout is set to zero (the default), then the operation will never timeout.

```
struct timeval {
    int tv_sec;     // seconds
    int tv_usec;    // microseconds
};
```

Non-blocking sockets with timed I/O

What's the problem in non-blocking I/O?

- Example

```
tv.tv_sec = 5;  /* 5 Secs Timeout */
tv.tv_usec = 0;
ret = setsockopt(client_socket, SOL_SOCKET, SO_RCVTIMEO, (struct timeval *)&tv, sizeof(struct timeval));

if (ret == SO_ERROR)
{
printf("setsockopt() failed...\n");
return -1;
}
else
printf("setsockopt() is OK!\n");

// read the message from client and copy it in buffer
int numBytes = recv(client_socket, buffer, sizeof(buffer), 0);
if (numBytes <= 0)
{
// nothing received from client in last 5 seconds
close(client_socket);
printf("nothing received in the last %ld seconds, close client socket\n", tv.tv_sec);
break;
}
```

Summary & Tasks

Summary

- We talked about how to implement a concurrent server using fork().
- We learned how to set timeout for a socket using setsockopt().

Task

The code for concurrent server and client are available in Canvas. Run the code and answer the following questions:

1. Can the UDP server in Tutorial2 serves multiple UDP clients concurrently? Why?
2. What will happen if we do not close the server_socket/client_socket in client/server process?
3. What's the problem in non-blocking I/O?
4. Find the problem of `sigchld_handler` function and give the right one.

Submit a pdf document on Canvas before 11:59pm tonight!

Thank You!