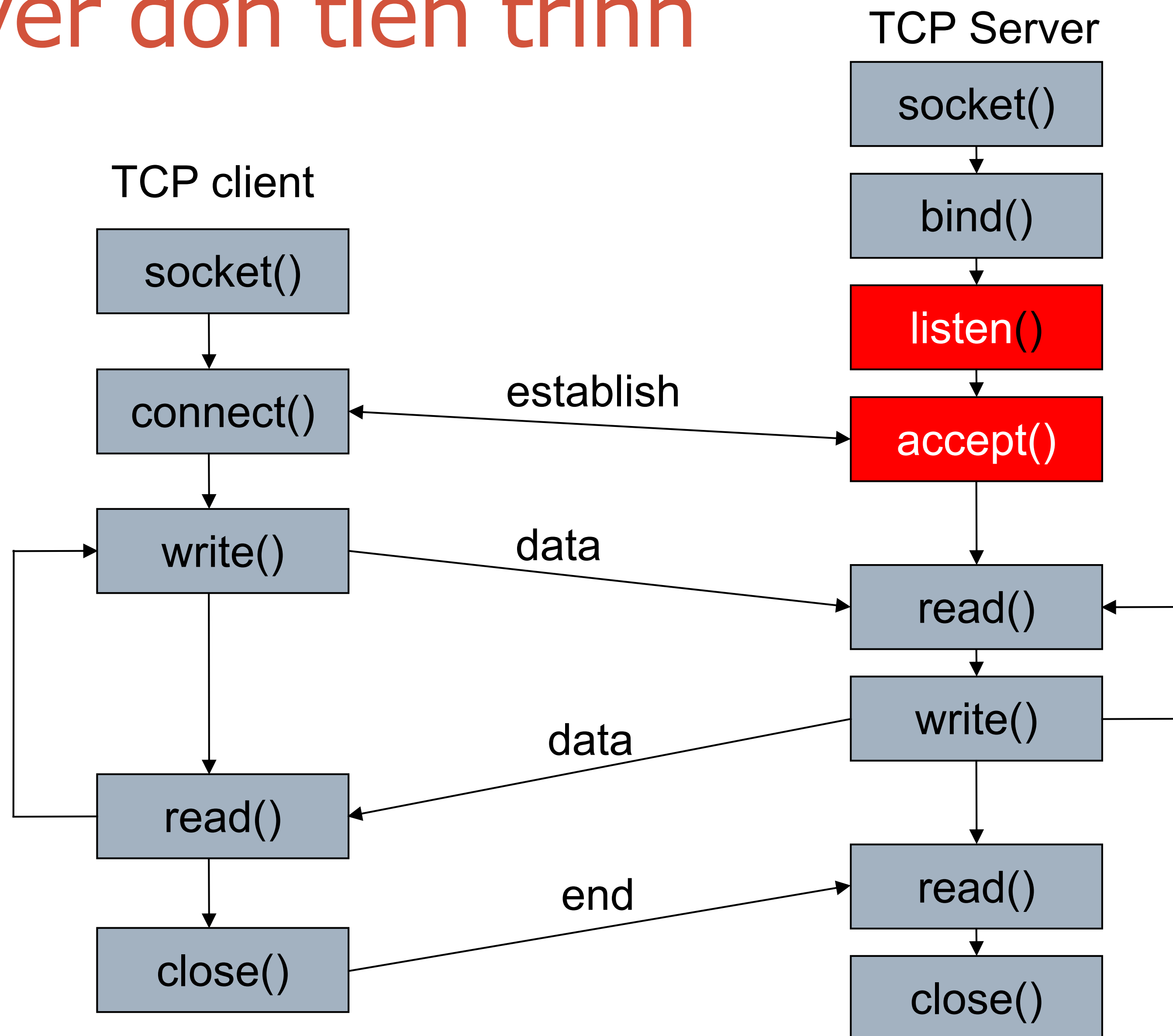
MULTI-PROCESS TCP SERVER

Content

- Forking Server
- Socket I/O Models
- select()

Quy trình làm việc của một TCP server đơn tiến trình



Câu hỏi đặt ra

- Làm thế nào một server có thể phục vụ nhiều client? Liệu có phải chờ một client kết thúc phiên làm việc rồi mới phục vụ một client khác?
- Nếu có 2 client cùng muốn được phục vụ một lúc thì phải làm thế nào?
  - Ví dụ: SV đăng ký học tập trên eHUST, thường có nhiều SV đăng ký đồng thời.

Các loại server

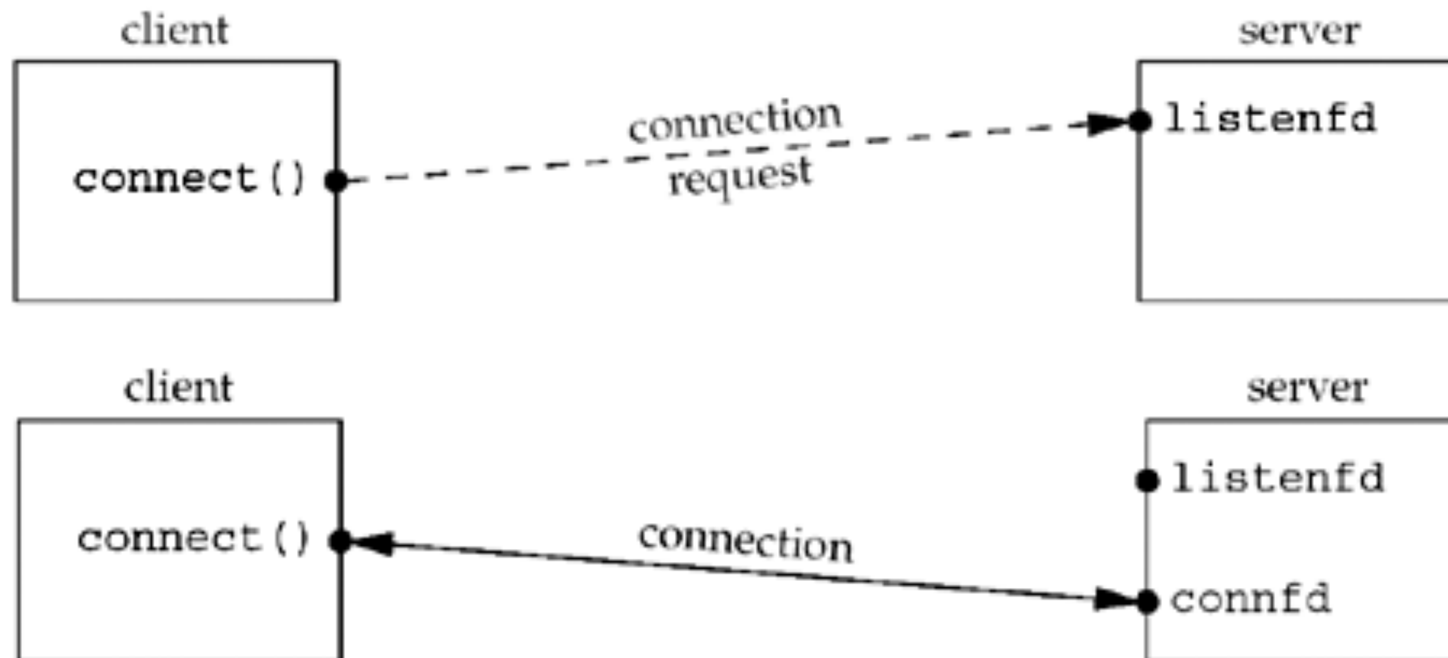
- *Iterating server*: Chỉ mở một socket và chỉ nhận một kết nối tại 1 thời điểm.
- *Forking server*: Mở 1 socket để nghe kết nối. Mỗi khi chấp nhận (accept) 1 kết nối thì tự nhân bản (fork) ra một tiến trình con để làm việc với kết nối đó.
- *Concurrent single server*: Server chỉ có 1 tiến trình theo dõi đồng thời tất cả các socket đang mở cho kết nối với nhiều client bằng cách sử dụng "select". Kết nối với client nào có dữ liệu cần trao đổi thì mới được phục vụ.

Iterating server

- Đơn giản, là loại đang được triển khai cho đến thời điểm này của môn học
- Chỉ có 1 tiến trình nên về cơ bản khi một client sử dung dịch vụ dài thời gian, client khác phải chờ mà không được phục vụ.
  - → Cần một server đa tiến trình
- Cách đơn giản nhất để viết một server đa tiến trình trên Linux là sử dung *fork* để tạo ra một tiến trình con là bản sao phục vụ mỗi client.

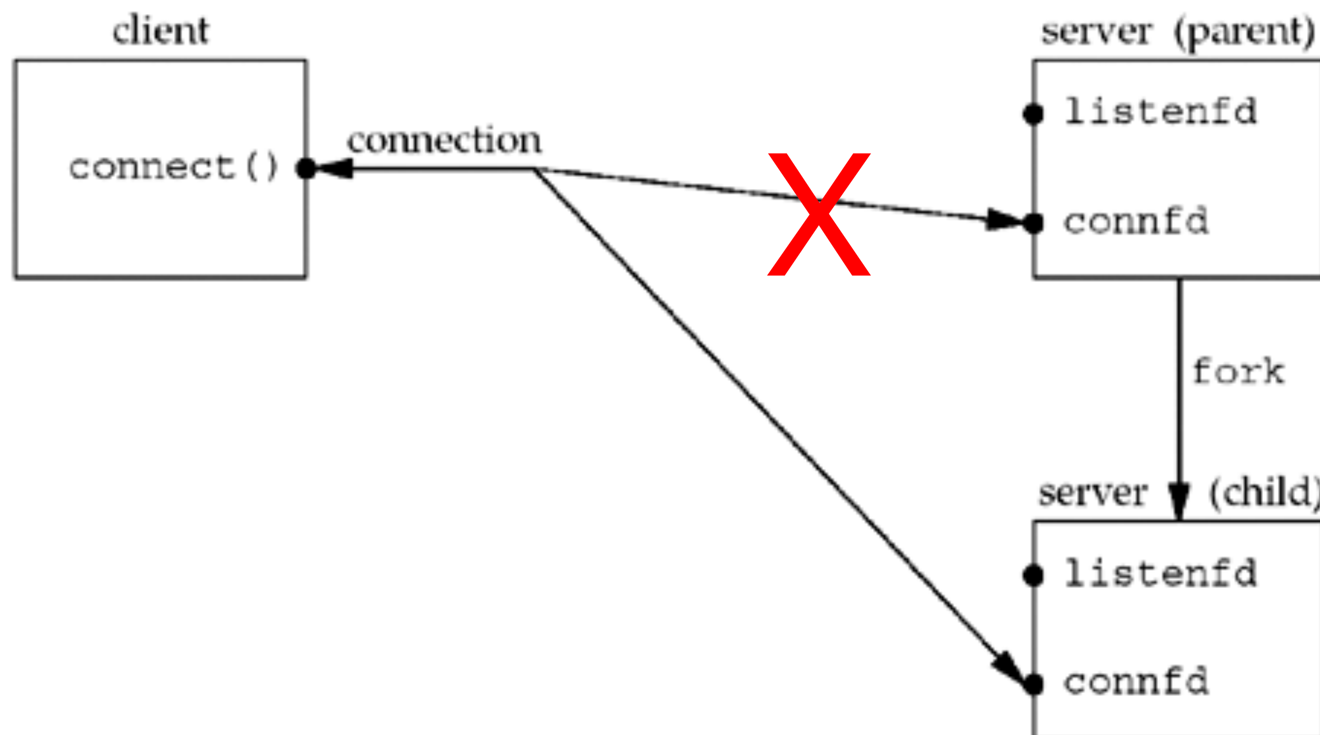
Iterative server vs forking server

- Iterative server
  - Khi kết nối, tạo một socket mới (connfd) trên cổng được kết nối để làm việc với client.
  - Vẫn tiến trình hiện tại đọc trao đổi dữ liệu trên kết nối vừa được thành lập (qua connfd).



Iterative server vs forking server

- forking server
  - Khi kết nối, tạo một socket mới (connfd) trên cổng được kết nối để làm việc với client.
  - Nhân bản tiến trình hiện tại (paren’t) thành tiến trình con (child) trao đổi dữ liệu trên kết nối vừa thành lập



Fork một tiến trình

- **fork** là hoạt động một tiến trình tự nhân bản chính nó trong hệ điều hành đa nhiệm.
- Tiến trình được nhân bản là tiến trình cha
- Tiến trình được tạo ra là tiến trình con
- Tiến trình cha nhân bản bộ nhớ và chuyển cho tiến trình con
  - 2 tiến trình có cùng chương trình, cùng các tham số với cùng giá trị tại thời điểm nhân bản.
- Fork được giới thiệu đầu tiên trong UNIX.
- Trong linux, tiến trình đầu tiên được tạo ra là "init", các tiến trình tiếp theo là con, cháu của init.

fork

```
#include <unistd.h>

pid_t fork(void);
```

- Hàm này và các biến thể của nó là cách duy nhất để các hệ điều hành kiểu Unix tạo ra các tiến trình mới.
- Hàm trả về:
  - Trong tiến trình gọi hàm (tiến trình cha) sẽ trả về process ID của tiến trình được tạo ra (tiến trình con).
  - **Trong tiến trình con, trả về 0** → *giá trị trả về được sử dung làm dấu hiệu phân biệt tiến trình cha và con*

Ví dụ sử dung fork

```
pid_t pid;
int listenfd, connfd;
listenfd = socket( ... );
/* fill in sockaddr_in{} with server's well-known port */
bind(listenfd, ... );
listen(listenfd, 5);
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 */
}
```

Fork ra một tiến trình con

Xử lý riêng của tiến trình con

Kết thúc tiến trình con theo đúng chuẩn

- Khi một tiến trình con kết thúc
  - Thông tin về tiến trình con vẫn được duy trì trong bảng danh mục tiến trình để tiến trình cha có thể đọc trạng thái trả về của tiến trình con.
  - Chừng nào tiến trình cha chưa đọc trạng thái trả về, tiến trình con ở trạng thái Zombie → Z: trạng thái không còn chạy nhưng vẫn chiếm chỗ trong bảng tiến trình
  - Bảng danh mục tiến trình có thể tra được bằng $ps aux
- Tiến trình cha cần đọc trạng thái trả về của tiến trình con
  - Xác định khi nào tiến trình con kết thúc:
    - Khi một tiến trình con kết thúc, nó gửi tín hiệu *SIGCHLD* về tiến trình cha.
  - Đọc trạng thái trả về của tiến trình con:
    - Tiến trình cha có thể theo dõi tín hiệu *SIGCHLD* và kích hoạt việc đọc trạng thái trả về bằng lời gọi hàm hệ thống "wait"
  - Tiến trình zombie sẽ bị xóa khỏi bang tiến trình sau đó

Tín hiệu (signal)- ngắt (interrupt)

- Một tín hiệu (signal) là cách để một tiến trình thông báo về một sự kiện xẩy ra.
  - Tín hiệu còn gọi là ngắt mềm (software interrupts).
- Tín hiệu thường xẩy ra bất chợt, không biết được khi nào tín hiệu sẽ đến.
- Tín hiệu có thể được gửi ...
  - từ một tiến trình đến một tiến trình khác (hoặc đến chính nó)
  - từ hạt nhân đến một tiến trình

Tín hiệu

- Gõ một tổ hợp phím trong cửa sổ terminal đang chạy một chương trình có thể làm hệ thống sinh ra một số tín hiệu và gửi đến tiến trình của chương trình đang chạy. Ví dụ:
  - Ctrl-C gửi tín hiệu INT, nghĩa là bảo tiến trình dừng lại ("interrupt", SIGINT)
  - Ctrl-Z gửi tín hiệu TSTP ("terminal stop", SIGTSTP)
  - Ctrl-\ gửi tín hiệu QUIT (SIGQUIT)
    - Mặc định sẽ làm tiến trình dung lại và gây dump core.
- SIGTERM gửi tín hiệu ra lệnh một tiến trình kết thúc
  - Không giống tín hiệu SIGKILL, tín hiệu này có thể được bắt và xử lý hoặc bỏ qua bởi tiến trình nhận.

Bắt và xử lý tín hiệu bằng **sigaction**

- Sigaction cho phép quy định/thay đổi hàm sẽ được kích hoạt mỗi khi chương trình nhận được một tín hiệu.
- Hàm được định nghĩa trong cấu trúc sigaction
  - **int sigaction(int** *signum*, **const struct sigaction** **act*, **struct sigaction** **oldact*);
  - struct sigaction {

```
        void      (*sa_handler)(int); // hàm được kích hoạt
        void      (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t  sa_mask;
        int       sa_flags;
        void      (*sa_restorer)(void);
    };
```

Bắt và xử lý tín hiệu bằng **sigaction**

- Có 3 cách đăng ký:
  - Cung cấp trong cấu trúc **sigaction** một hàm sẽ được gọi mỗi khi một tín hiệu cụ thể xuất hiện (sa_handler)

    Prototype: void handler (int signo);
  - Bỏ không xử lý một tín hiệu bằng cách thiết lập sa_handler =*SIG_IGN*
  - Xử dung hàm xử lý mặc định bằng cách đặt sa_handler = *SIG_DFL*.

Example

```c
#include <signal.h>
#include <stdio.h>

void
termination_handler (int signum)
{
  struct temp_file *p;
  FILE* f=fopen("log.txt","a");
  fprintf(f, "SIGINT:%d\t SIGTSTP:%d\t SIGTERM:%d\n", SIGINT, SIGTSTP,SIGTERM);
  fprintf(f,"In termination handler Signal:%d\n", signum);
  fclose(f);
}

int
main (void)
{
  printf("Begin of program\n");
  struct sigaction new_action, old_action;

  /* Set up the structure to specify the new action. */
  new_action.sa_handler = termination_handler;
  sigemptyset (&new_action.sa_mask);
  new_action.sa_flags = 0;

  sigaction (SIGINT, &new_action, &old_action);  // Ctrl+C for generating the signal
  sigaction (SIGTSTP, &new_action, &old_action); // Ctrl+Z for generating the signal
  sigaction (SIGTERM, &new_action, &old_action); // kill -15 pid for generating the signal
 while (1) {}
}
```

Xử lý một tín hiệu bang hàm **signal**

- Giải pháp thay thế cho sigaction.
- Mục tiêu: đăng ký một hàm sẽ xử lý một tín hiệu.

- <signal.h>
- void (*signal(int sig, void (*func)(int)))(int)
  - sig: Tín hiệu sẽ được bắt và xử lý
  - **func:** con trỏ đến hàm sẽ được kích hoạt khi tín hiệu sig xảy ra

Example

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

void sighandler(int);

int main()
{
   signal(SIGINT, sighandler);

   while(1)
   {
      printf("Going to sleep for a second...\n");
      sleep(1);
   }
   return(0);
}

void sighandler(int signum)
{
   printf("Caught signal %d, coming out...\n", signum);
   exit(1);
}
```

Xử lý tín hiệu SIGCHLD

- Một tiến trình con khi kết thúc sẽ rơi vào trạng thái zombie và thông tin của nó vẫn ở trong bản danh mục tiến trình cho đến khi tiến trình cha đọc thông tin này sau đó
- Thông tin này bao gồm process ID của tiến trình con, trạng thái kết thúc, thông tin về các tài nguyên tiến trình con đang dùng (CPU time, memory, etc.).
- Việc các thông tin này chiếm chỗ trong hạt nhân dẫn đến hạn chế số lượng tiến trình có thể chạy

→Vì vậy, mỗi khi *fork* một tiến trình con, tiến trình cha phải *wait* (đọc trạng thái kết thúc của con) để tiến trình con có thể thoát khỏi trạng thái zombie

- →Thiết lập một một hàm handler sẽ xử lý *SIGCHLD*, bên trong handler gọi *wait*
- →Đăng ký hàm handler bắt *SIGCHLD* : *signal (SIGCHLD, handler);*

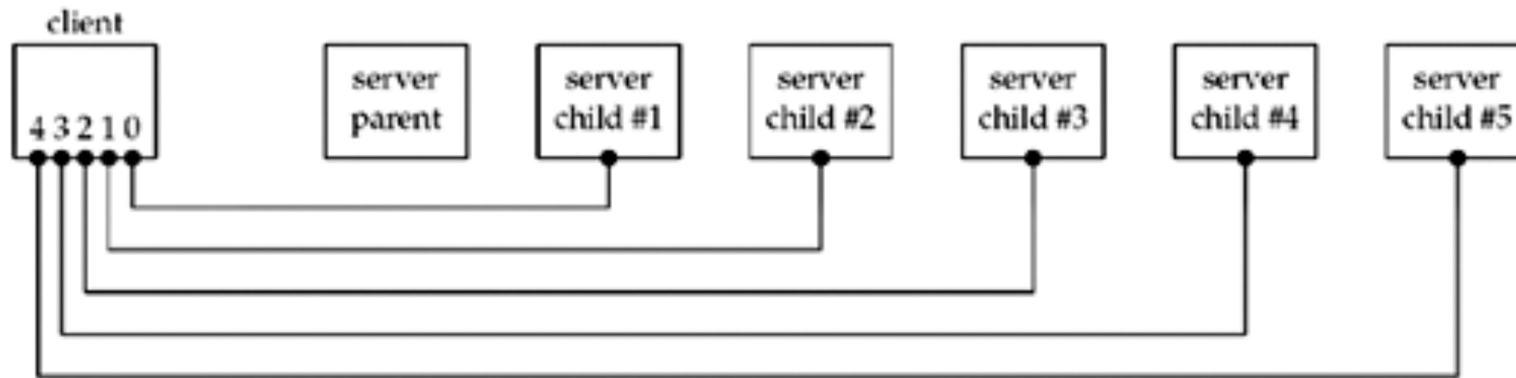
wait() and waitpid()

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

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

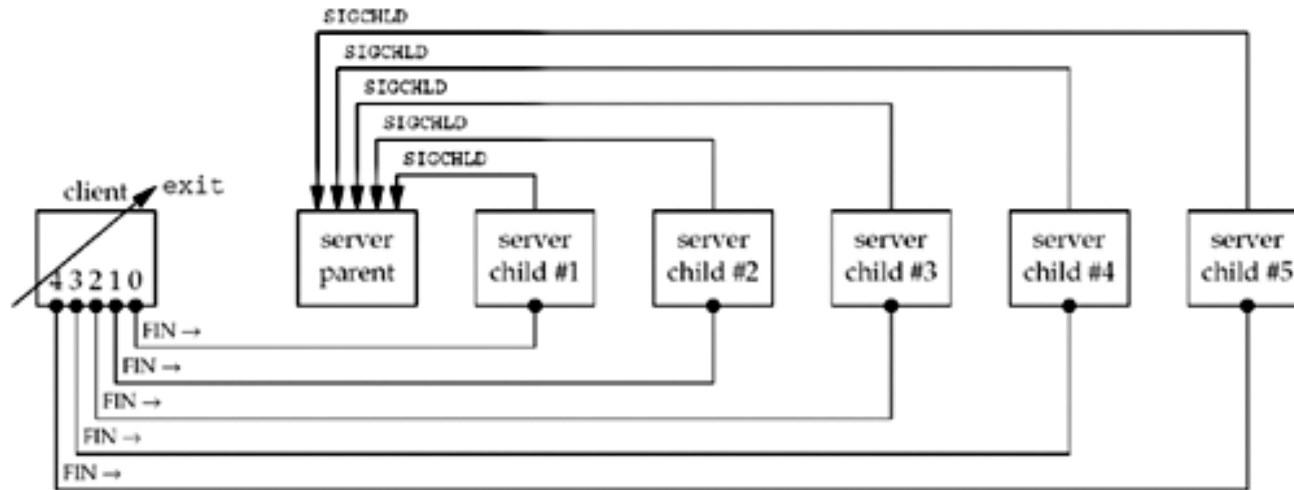
- Chờ một tiến trình con thay đổi trạng thái. Thay đổi trạng thái có thể diễn ra khi:
  - Tiến trình con kết thúc;
  - Tiến trình con kết thúc vì nó nhận được một tín hiệu; hoặc
  - Tiến trình con tiếp tục khi nhận được một tín hiệu.
- Sử dung để đọc trạng thái kết thúc của một tiến trình con
- Hàm trả về các giá trị:
  - Giá trị trả về:
    - process ID của tiến trình con
    - 0 hoặc -1 nếu lỗi
  - Trạng thái kết thúc của tiến trình con (một số nguyên) được trả lại trong con trỏ *statloc*.

Difference between wait() and waitpid()



- 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

Difference between wait() and waitpid() (2)



- → five SIGCHILD is sent
- The signal is handled once with wait () in the server
  - → four children are zombies
- It can happen when many users connect to a server
- → we have to use *waitpid()*

waitpid()

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

- pid < 0: wait for status change of any child process whose process group ID is equal to the absolute value of *pid*.
- **pid = -1**: wait for for status change of 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.

Forking server

```
pid_t pid;
int listenfd, connfd;
listenfd = socket( ... );
bind(listenfd, ... );
listen(listenfd, 5);

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 */
   }
   signal(SIGCHLD,sig_chld);
   close(connfd); /* parent closes connected socket */
}
```

sig_chld: SIGCHLD handler

```
void sig_chld(int signo)
{
  pid_t pid;
  int stat;
  while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
          printf("child %d terminated\n", pid);
  return;
}
/*
WNOHANG: waitpid does not block
while loop: waitpid repeatedly until there is no child
  process change status, i.e until waitpid returns 0.
*/
```

Exercise

1) Make your TCP EchoServer be able to work with multiple client in the same time

- Use forking process
- Handle the SIGCHLD signal in server.

2) Turn the Student schedule management to a TCP server that can serves multiple clients simultaneously (able to handle requests for schedule from different users)

4) Write file transfer application for multiple users using TCP

Server multi-threads

- Trong trường hợp server cần sử lý nhiều kết nối với nhiều client, mỗi kết nối được xử lý độc lập, việc sử dụng mô hình multi-process với fork là ổn.
  - Ví dụ, server đăng ký học tập, server quản lý upload file.
- Tuy nhiên, nếu như các kết nối có liên quan đến nhau, dẫn đến nhu cầu chia sẻ bộ nhớ giữa các tiến trình xử lý thì sử dụng mô hình multi-process có hạn chế
  - Ví dụ: server chat, nếu mỗi process quản lý một client và cần forward các bản tin từ client nọ sang client kia thì mỗi process cần được cung cấp danh sách các file descriptor của process khác ➔ cần chia sẻ nhau file descriptor
  - Nhưng các process có không gian bộ nhớ tách biệt và quản lý các file descriptor riêng.
- ➔ Giải pháp, sử dụng mô hình multi-thread

Server multi-threads

- Một thread là một quá trình thực thi một nhiệm vụ bên trong một process (tiến trình).
  - Trong một tiến trình có thể kích hoạt nhiều thread để thực hiện nhiều nhiệm vụ độc lập/song song với nhau.
- Các thread của cùng một process chia sẻ không gian bộ nhớ và các file descriptor.
- Thay vì sinh 1 process để phục vụ mỗi client, có thể sinh một thread để thực hiện nhiệm vụ phục vụ mỗi client
- Có thể chia sẻ bộ nhớ giữa các thread bằng cách truyền cho các thread các tham số/biến số chung.
- Thư viện: <pthread.h>

Sử dụng thread

- Các thread thuộc cùng một process được gán một định danh phân biệt kiểu *pthread_t*
  - Không có giá trị định danh trong một process không phải là cha của nó.
- `int pthread_equal(pthread_t tid1, pthread_t tid2);` // so sánh 2 định danh
- `pthread_t pthread_self(void);` // trả lại định danh của thread đang gọi hàm

Tạo mới một thread

```
include <pthread.h>
int pthread_create(pthread_t * tidp, const pthread_attr_t
*attr, void *(*start_rtn)(void *), void * arg)
```

- Hàm tạo một thead, thực hiện chương trình trong hàm được truyền vào ở `start_rtn`
- `Tidp:` con trỏ trả lại số hiệu thread được tạo.
- `start_rtn:` tên hàm được thread thực hiện, tham số truyền vào cho hàm là tham số ở biến `arg`
- `Attr:` thuộc tính của thread

Các hàm tương ứng giữa thread và process

Process	Thread	Meaning
`fork`	`pthread_create`	create a new flow of control
`exit`	`pthread_exit`	exit from an existing flow of control
`waitpid`	`pthread_join`	get exit status from flow of control
`getpid`	`pthread_self`	get ID for flow of control
`abort`	`pthread_cancel`	request abnormal termination of flow of control

Đọc thêm về pthread:
https://notes.shichao.io/apue/ch11/#chapter-11-threads

Ví dụ

```
Staring thread tid 6133985280, current nb_thread=2
Staring thread tid 6135705600, current nb_thread=5
Staring thread tid 6134558720, current nb_thread=3
Staring thread tid 6136279040, current nb_thread=7
Staring thread tid 6136852480, current nb_thread=7
Staring thread tid 6135132160, current nb_thread=4
Staring thread tid 6137425920, current nb_thread=8
Staring thread tid 6137999360, current nb_thread=9
Staring thread tid 6138572800, current nb_thread=10
Staring thread tid 6139146240, current nb_thread=10
End of thread tid 6134558720, current nb_thread=10
End of thread tid 6139146240, current nb_thread=10
End of thread tid 6133985280, current nb_thread=10
End of thread tid 6136852480, current nb_thread=10
End of thread tid 6135705600, current nb_thread=10
End of thread tid 6136279040, current nb_thread=10
End of thread tid 6137425920, current nb_thread=10
End of thread tid 6135132160, current nb_thread=10
End of thread tid 6138572800, current nb_thread=10
End of thread tid 6137999360, current nb_thread=10
```

Hàm được thread thực hiện

Tạo một thread

```c
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <pthread.h>
5
6  #define MAX_THREAD 10
7
8  typedef struct
9  {
10     int* nb_threads;
11 } paramStruct; // data structure will be passed to a thread
12
13 // function that a thread executes.
14 void* thread_handler(void* param)
15 {
16     paramStruct * p = (paramStruct *) param; //get parameters
17     pthread_t tID = pthread_self();
18
19     printf("Staring thread tid %lu, current nb_thread=%d\n", (unsigned long int) tID, *p->nb_threads);
20     //Do something here
21     sleep(5);//sleep for 50 seconds
22     printf("End of thread tid %lu, current nb_thread=%d\n", (unsigned long int) tID,  *p->nb_threads);
23     return NULL;
24 }
25
26 int main(int argc, char **argv)
27 {
28     int nb_threads=0;
29     pthread_t threadList [MAX_THREAD]; //list of threads, for later cleaning them.
30
31     for (int i=0; i<10; i++)
32     {
33         nb_threads++; // count nb of thread created
34         // Generate a thread to execute a task in thread_handler with paramet in client_info
35         pthread_t* t_child =malloc(sizeof(pthread_t));
36         paramStruct* param = malloc(sizeof(paramStruct)); // must do it to make sur each thread receive different memory for param
37         param->nb_threads =&nb_threads;
38         pthread_create(t_child, NULL, thread_handler, param );
39
40         threadList[i]=*t_child;
41     }
42     // cleaning the stacks of all threads.
43     for (int i=0; i<10; i++)
44     {
45         pthread_join(threadList[i], NULL);
46     }
47 }
```

Bài tập

- Viết chương trình chat sử dụng TCP, thay vì UDP
- Hướng dẫn:
  - Với mỗi kết nối từ client, hãy tạo một thread phục vụ client thay vì một process
  - Chuyển cho thread phục vụ C1 thông tin về C2 để có thể forward dữ liệu cho C2
  - Chuyển cho thread phục vụ C2 thông tin về C1