CS4233
Network Programming
--Thread

陳柏諭
MNET, CS, NTHU

2017/05/08

Interactive VS. Concurrent

- Interactive server
  - Serve one client at one time
  - Ex: daytime server
- Concurrent server
  - Serve several clients simultaneously
  - Ex: web server

Sample code --Concurrent Servers: Outline

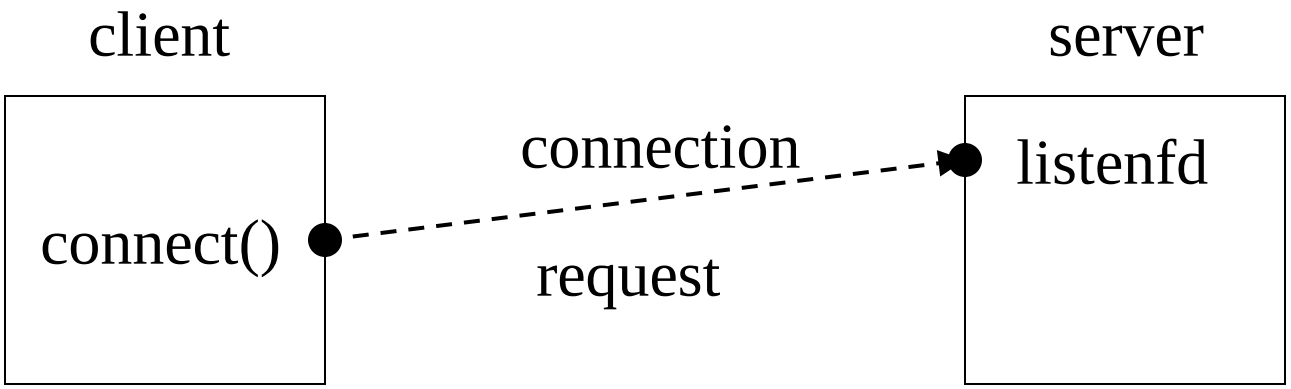
```
pid_t       pid;
int         listenfd, connfd;

listenfd = socket (...);
           /* fill in socket_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 */
}
```

Review
--socket、thread 和繼承性

- 新的 thread 被建立時，應從產生它的 parent 身上繼承一份所有已開啟的 socket。
- Socket 使用 reference count機制，當一個 socket 被建立時，系統就把 socket 的 reference count 設為 1，當這個 reference count大於零的時候，socket都會存在。
  - 當程式產生新的 process/thread 時，會把 parent 和 child process 的 reference count 都加 1。

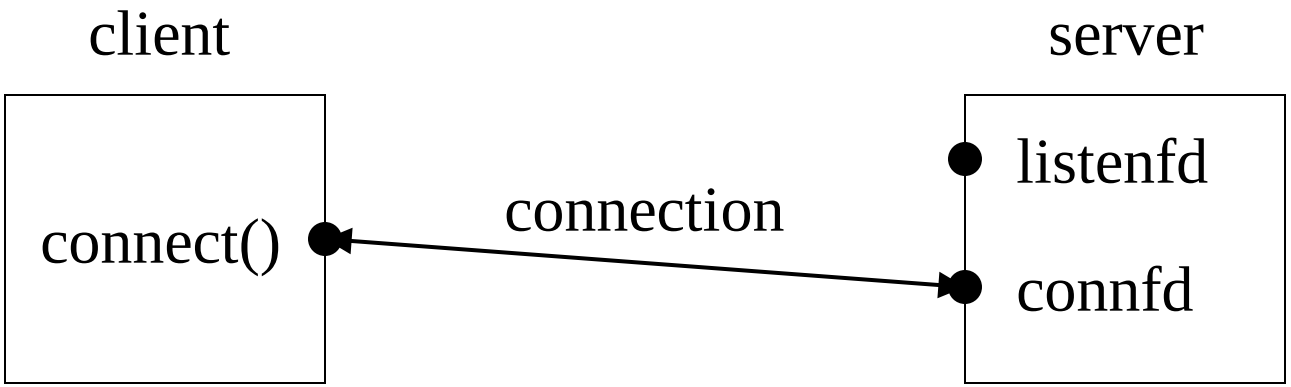
Review
-- socket、thread 和繼承性 (Cont.)

- Status of client-server before call to accept returns



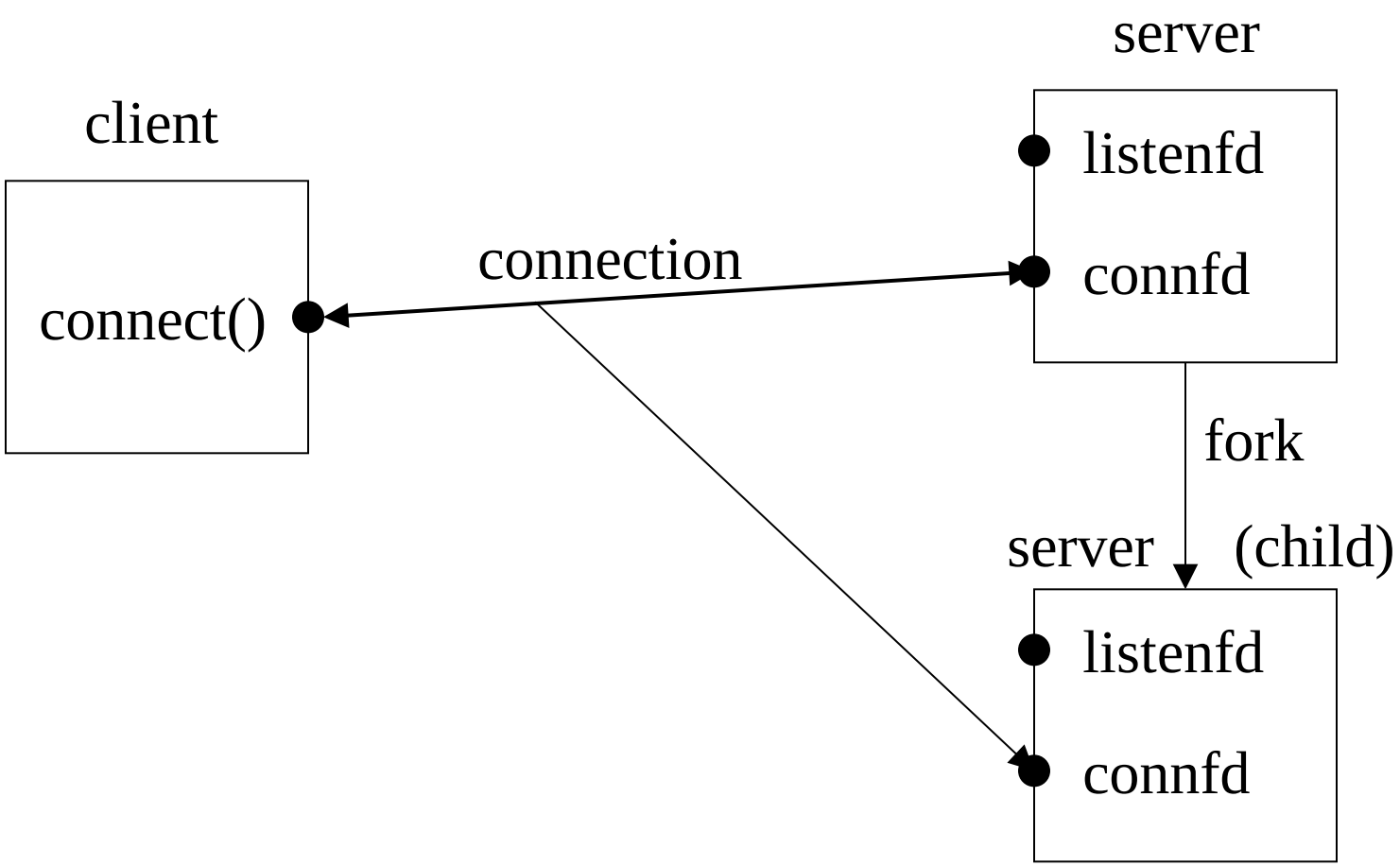
Review
-- socket、thread 和繼承性 (Cont.)

- Status of client-server after return from accept



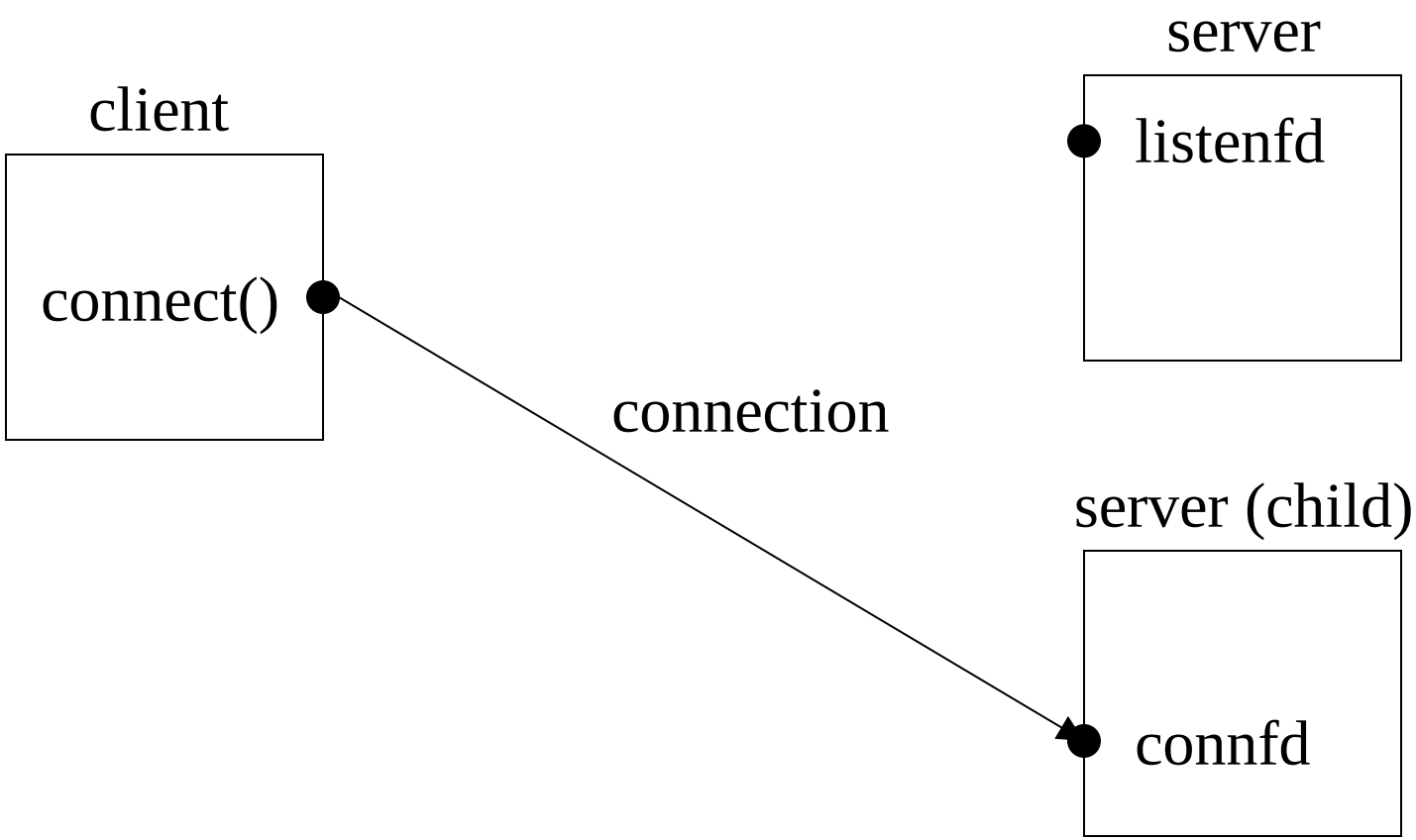
Review
-- socket、thread 和繼承性 (Cont.)

- Status of client-server after **fork** returns



Review
-- socket、thread 和繼承性 (Cont.)

- Status of client-server after parent and child close appropriate sockets



Review
-- socket、thread 和繼承性 (Cont.)

- 當某一 process/thread 呼叫 socket 的 close 程序時，系統將其 reference count 值減一，並且從 thread 清單去除。
- 若 thread 在關閉 socket 前結束，系統會把 thread 開啟的所有 socket，執行 close 程序。

Shortages of fork()

- High cost
  - A duplicate memory concept, descriptors…
- Communication between parent and child
  - Inter Process Communication (IPC)
  - Parent → Child
  - Child → Parent

Thread

- Lightweight Process
  - Creation: 10~100 times faster than creating process
- Share the same global memory
  - Shares
    - Process instruction
    - Most data
    - Open files (e.g., descriptors)
    - Signal handlers and signal descriptors
    - Synchronization problem
    - Current working directory
    - User and group IDs
  - Synchronization problem

Creation time comparison

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

Fig. source: https://computing.llnl.gov/tutorials/pthreads/

Unit: second

Sample code:
https://computing.llnl.gov/tutorials/pthreads/fork_vs_thread.txt

Thread

- Each thread has
  - Thread ID
  - Set of registers, including program counter and stack pointer
  - Stack (for local variables and return address)
  - errno
  - Signal mask
  - Priority

Copy-on-write需要MMU（Memory Management Unit）支援

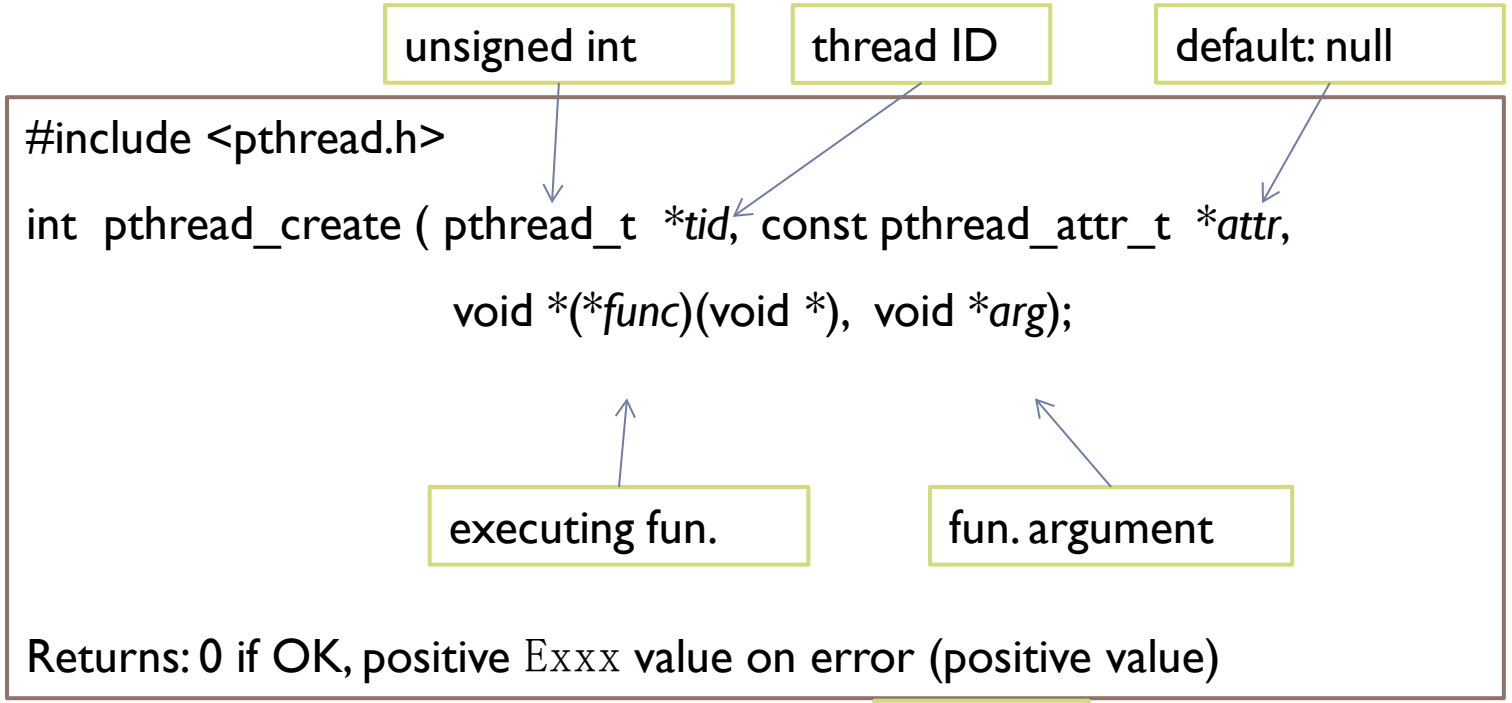
	Advantages	Disadvantages	Enhancement
Fork	● No synchronization problem	● Expensive creation ● IPC required	● Copy-on-write
Threads	● Fast creation ● Information sharing	● Data synchronization	● Mutex and Condition variables ● Thread-specific data

Basic Thread Functions --create

A program is started by exec, a single thread is created, called *initial thread* or *main thread*.

Additional threads are created by ptherad_create.

Pthreads: POSIX threads



Basic Thread Functions
--join: wait for thread termination
--self: get thread ID

```
#include <pthread.h>
int  pthread_join ( pthread_t  tid,  void  ** status);
                          Returns: 0 if OK, positive Exxx value on error
```

```
#include <pthread.h>
pthread_t  pthread_self (void);
                          Returns: thread ID of calling thread
```

therad	process
ptherad_create	fork
pthread_join	waitpid
pthread_self	getpid

Basic Thread Functions --detach: detached thread

```
#include <pthread.h>
int  pthread_detach (pthread_  tid);

        Returns: 0 if OK, positive Exxx value on error
```

- A thread is
  - jointable(default): thread ID and exit status are until calling `pthread_join`
  - detached: like a daemon process. When it terminates, all its resources are released and we cannot wait for it to terminate.
- This function is usually called by itself (child thread)
  - pthread_detach(pthread_self())
- Main thread also can call it for its child thread
  - pthread_detach(thread_id)

Basic Thread Functions --exit: terminate a thread

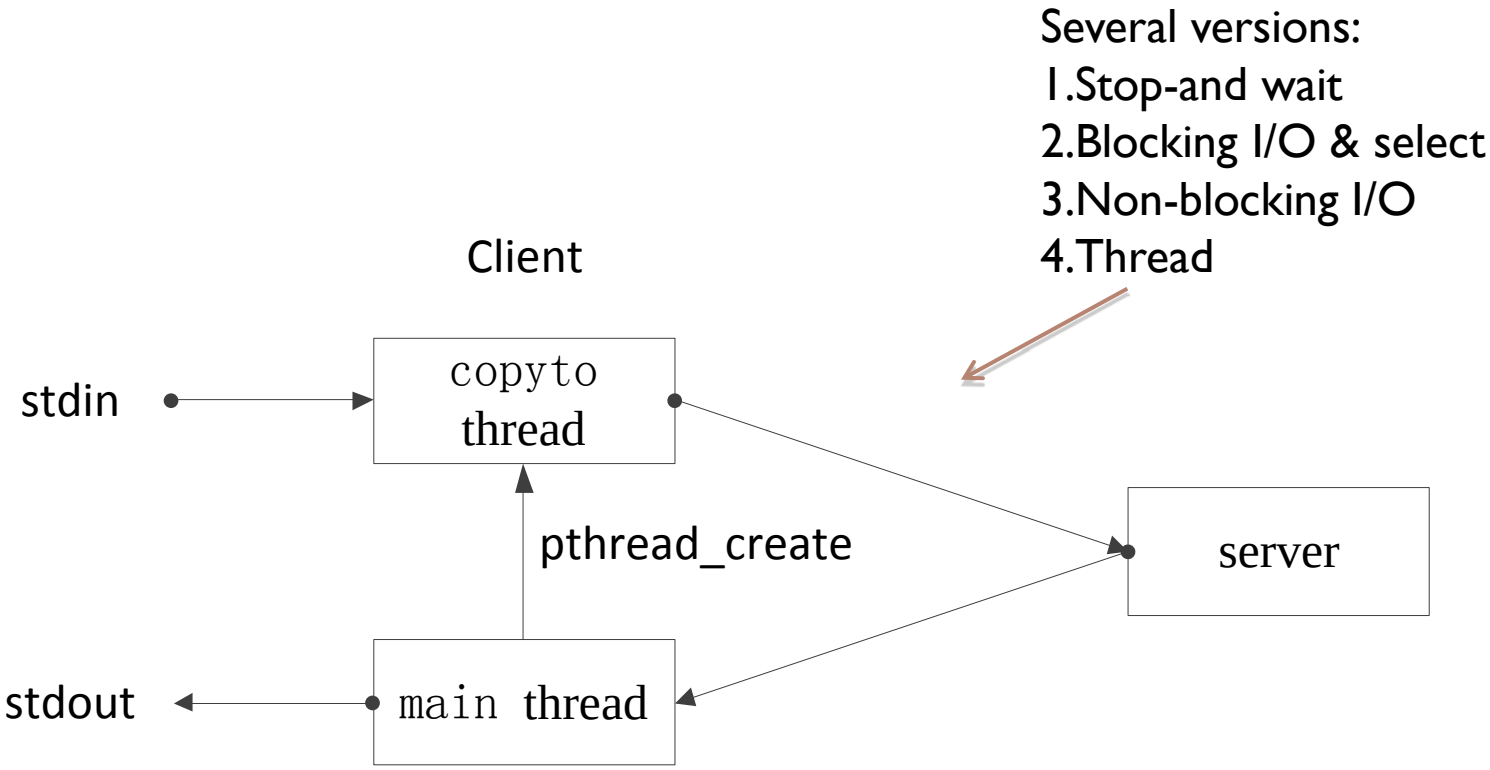
```
void  pthread_exit (void  *status);
```

Does not return to caller

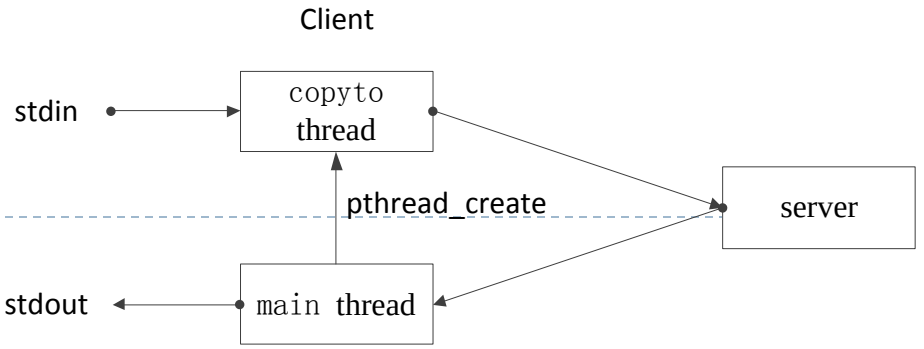
p_join() is called by main
p_exit() is called by child

- If this thread is not detached, its thread ID and exit status are retained for a later `pthread_join`.
- Two ways to terminate a thread
  - The function(3rd arg to `pthread_create`) that starts the thread can return.
  - The `main` function of the process returns or any thread calls `exit`. The process terminates, including all threads.

TCP Echo Client
`str_cli` using thread



TCP Echo Client
str_cli using thread




```c
#include "xxx.h"
#include "pthread.h"

void    *copyto(void *); /*no input argument*/

static int  sockfd;     /* global for both threads to access */
static FILE *fp;

void
str_cli(FILE *fp_arg, int sockfd_arg)
{
    char        recvline[MAXLINE];
    pthread_t   tid;

    sockfd = sockfd_arg;    /* copy arguments to externals */
    fp = fp_arg;

    pthread_create(&tid, NULL, copyto, NULL);

    while (readline(sockfd, recvline, MAXLINE) > 0)
        fputs(recvline, stdout);
}
```

TCP Echo Client
`str_cli` using thread

coypto thread

```
24  void *
25  copyto(void *arg)
26  {
27      char    sendline[MAXLINE];
28
29      while (fgets(sendline, MAXLINE, fp) != NULL)
30          writen(sockfd, sendline, strlen(sendline));
31
32      shutdown(sockfd, SHUT_WR);  /* EOF on stdin, send FIN */
33
34      return(NULL);
35          /* return (i.e., thread terminates) when EOF on stdin */
36  }
```

/*leave str_cli() and go back to main()*/

exit(0);

}

close all threads

TCP Echo Server using thread

```c
#include    "pthread.h"
static void *doit(void *);      /* each thread executes this function */

int
main(int argc, char **argv)
{
    int             listenfd, connfd;
    pthread_t       tid;
    socklen_t       addrlen, len;
    struct sockaddr *cliaddr;

    if (argc == 2)
        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
    else if (argc == 3)
        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
    else
        err_quit("usage: tcpserv01 [ <host> ] <service or port>");

    cliaddr = malloc(addrlen); /*get client address*/

    for ( ; ; ) {
        len = addrlen;
        connfd = accept(listenfd, cliaddr, &len);
        pthread_create(&tid, NULL, &doit, (void *) connfd);
    }
}
```

```c
static void *
doit(void *arg)
{
    pthread_detach(pthread_self());
    str_echo((int) arg);    /* same function as before */
    close((int) arg);       /* done with connected socket */
    return(NULL);
}
```

Thread close socket by itself 因為main thread和其他thread 共享所有的 descriptor

TCP Echo Server using thread --problem: passed by pointer

```
1   ...
2
3   int
4   main(int argc, char **argv)
5   {
6       int             listenfd, connfd;
7       ...
8
9       for ( ; ; ) {
10          len = addrlen;
11          connfd = accept(listenfd, cliaddr, &len);
12          pthread_create(&tid, NULL, &doit, &connfd); /*connfd passed by pointer*/
13      }
14  }
15
16  static void *
17  doit(void *arg)
18  {
19      int connfd;
20
21      connfd = *((int *) arg); /*type transform*/
22      pthread_detach(pthread_self());
23      str_echo(cinnfd);    /* same function as before */
24      close(connfd);       /* done with connected socket */
25      return(NULL);
26  }
```

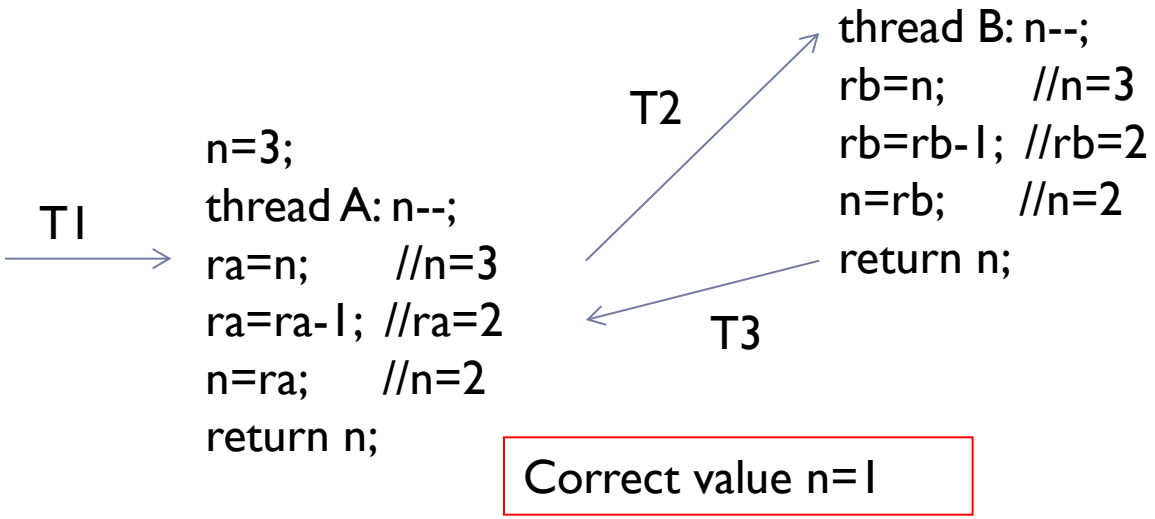
Problem: threads share the same connfd

TCP Echo Server using thread --better solution

```
3   static void *doit(void *);      /* each thread executes this function */
4   int
5   main(int argc, char **argv)
6   {
7       int             listenfd, *iptr;
8       ...
9
10      for ( ; ; ) {
11          len = addrlen;
12          iptr = malloc(sizeof(int)); /*for each thread*/
13          *iptr = accept(listenfd, cliaddr, &len);
14          pthread_create(&tid, NULL, &doit, iptr); /*passed by value*/
15      }
16  }
17
18  static void *
19  doit(void *arg)
20  {
21      int     connfd;
22      connfd = *((int *) arg);
23      free(arg); /*free memory*/
24
25      pthread_detach(pthread_self());
26      str_echo(connfd);       /* same function as before */
27      close(connfd);          /* done with connected socket */
28      return(NULL);
29  }
```

mutex : Mutual Exclusion

- Big problem
  - Randomly occurs
  - Especially in concurrent (parallel) programming
    - Sharing/accessing the same variables
  - *Fork* only shares descriptors but still encounters this problem
    - Sharing memory



Simple example
--counting program using 2 threads

```
1   #include    "pthread.h"
2
3   #define NLOOP 5000
4
5   int     counter;        /* incremented by threads */
6
7   void    *doit(void *);
8
9   int
10  main(int argc, char **argv)
11  {
12      pthread_t   tidA, tidB;
13
14      pthread_create(&tidA, NULL, &doit, NULL);
15      pthread_create(&tidB, NULL, &doit, NULL);
16
17          /* wait for both threads to terminate */
18      pthread_join(tidA, NULL);
19      pthread_join(tidB, NULL);
20
21      exit(0);
22  }
```

Simple example
--counting program using 2 threads

```
24  void *
25  doit(void *vptr)
26  {
27      int     i, val;
28
29      /*
30       * Each thread fetches, prints, and increments the counter NLOOP times.
31       * The value of the counter should increase monotonically.
32       */
33
34      for (i = 0; i < NLOOP; i++) {
35          val = counter;
36          printf("%d: %d\n", pthread_self(), val + 1);
37          counter = val + 1;
38      }
39
40      return(NULL);
41  }
```

Running results

…

4: 517

4: 518 } error

5: 518

5: 519

…

mutex : Mutual Exclusion

```
#include <pthread.h>
pthread_mutex_t work_mutex;

int pthread_mutex_lock (pthread_mutex_t  *mptr);
int pthread_mutex_unlock (pthread_mutex_  *mptr);
                                  Returns: 0 if OK, positive Exxx value on error
```

如果mutex為靜態的方式配置，就必須將他的初值設為PTHREAD_MUTEX_INITIALIZER
如果是在共享記憶體中配置一個mutex就必須呼叫pthread_mutex_init()對他初始化

Simple example
--counting program using 2 threads with mutex

```
24  pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;
25  ...
26  void *
27  doit(void *vptr)
28  {
29      int     i, val;
30
31      /*
32       * Each thread fetches, prints, and increments the counter NLOOP times.
33       * The value of the counter should increase monotonically.
34       */
35
36      for (i = 0; i < NLOOP; i++) {
37          pthread_mutex_lock(&counter_mutex); //LOCK
38
39          val = counter;
40          printf("%d: %d\n", pthread_self(), val + 1);
41          counter = val + 1;
42
43          pthread_mutex_unlock(&counter_mutex); //UNLOCK
44      }
45
46      return(NULL);
47  }
```