套接字编程：并发服务器（I）

Gaoyang Shan
Dept. of Software and Computer Engineering
Ajou University

변화와 융합의 중심
MMcN
이동 멀티미디어 융합 네트워크 연구실
Mobile Multimedia convergecne Network Lab.
http://mmcn.ajou.ac.kr

Contents

- 并发服务器(CS)的概念
- 使用 folk()的CS
- 使用Thread的CS

- References:

[1] W. Stevens, B. Fenner, A. M. Rudoff, UNIX Network Programming Volume 1-The Sockets Networking, Addison Wesley, 3rd ed., 2003

https://github.com/shihyu/Linux_Programming/blob/master/books/UNIX%20Network%20Programming(Volume1%2C3rd).pdf

[2] M. Kerrisk, The Linux Programming Interface, A Linux and UNIX System Programming Handbook, Oreilly & Associates Inc., 2010.

https://github.com/shihyu/Linux_Programming/blob/master/books/The%20Linux%20Programming%20Interface%20-%20A%20Linux%20and%20UNIX%20System%20Programming%20Handbook.pdf

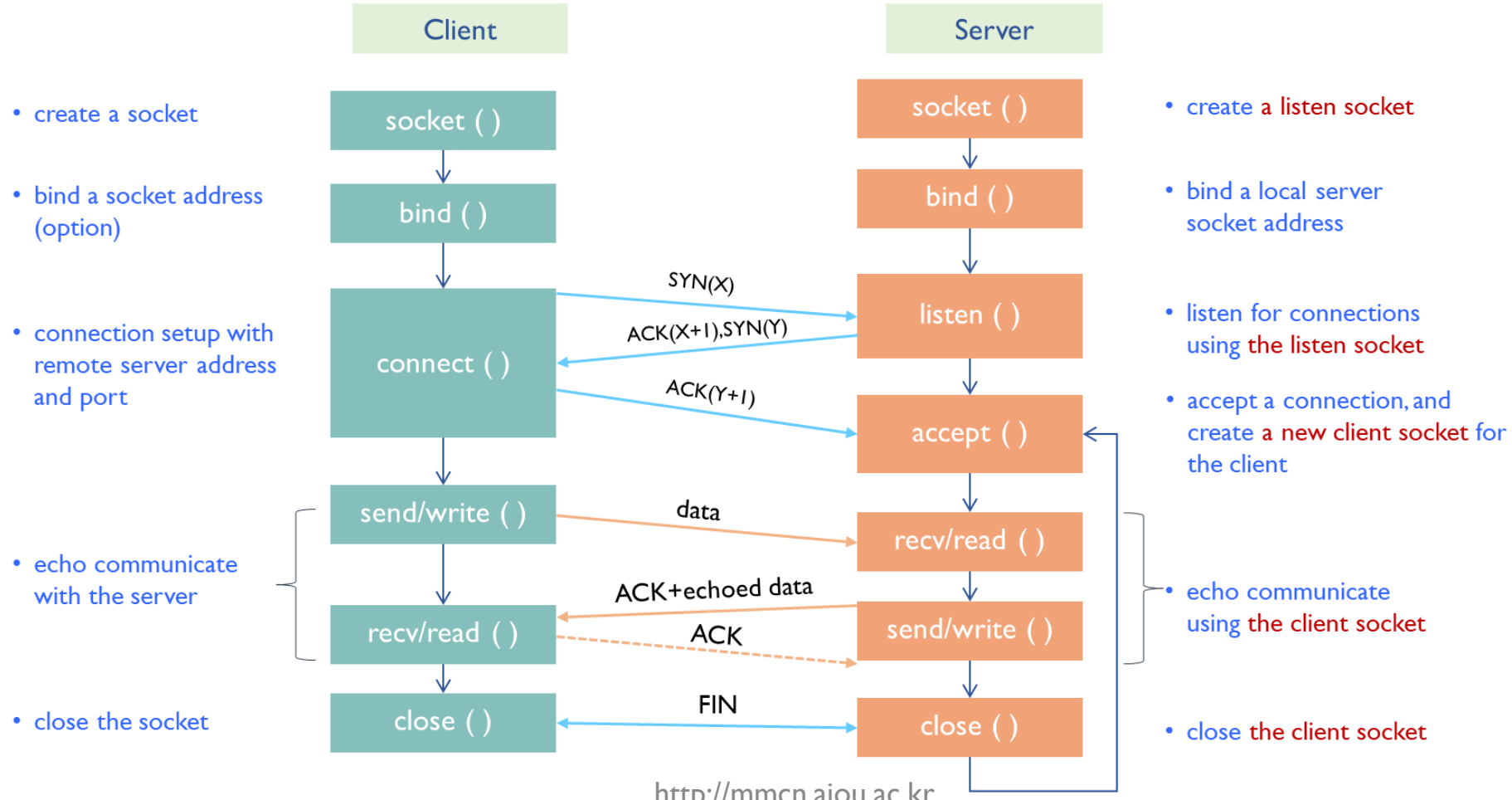
[3] Oracle, Multithreaded Programming Guide, Mar. 2019

https://docs.oracle.com/cd/E53394_01/pdf/E54803.pdf

并发服务器的概念

讨论 – 重新审视CO程序

❖ 示例代码实现可以为多少个客户端提供服务?

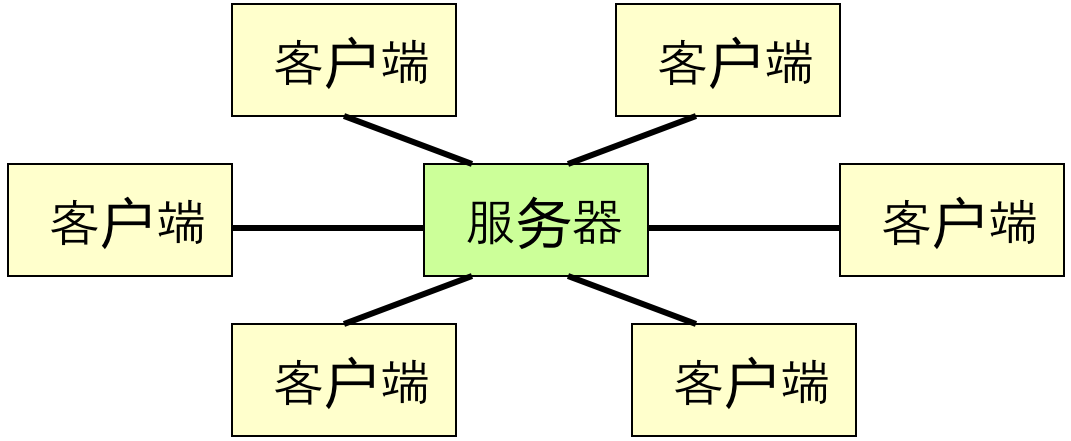


讨论 – 重新审视CO程序

- 迭代(或顺序)服务器
  - 每次处理一个请求（一个客户端）
- 为什么不使用迭代（或顺序）服务器?
  - 延迟增加
    - 客户端必须等待客户端 I 完成才能获得服务
  - 利用率低
    - 服务器处于空闲状态，等待客户端的请求.
    - 在那些空闲时间它本可以为其他客户提供服务!
- 解决方案
  - *并发服务器*
    - 同时为多个客户提供服务

并发服务器

- 并发服务器
  - 为多个客户端提供服务的并发服务器



  - 服务器基本注意事项
    - 在多个连接的客户端之间平衡资源
    - 像拥有专用服务器访问权限一样与客户打交道

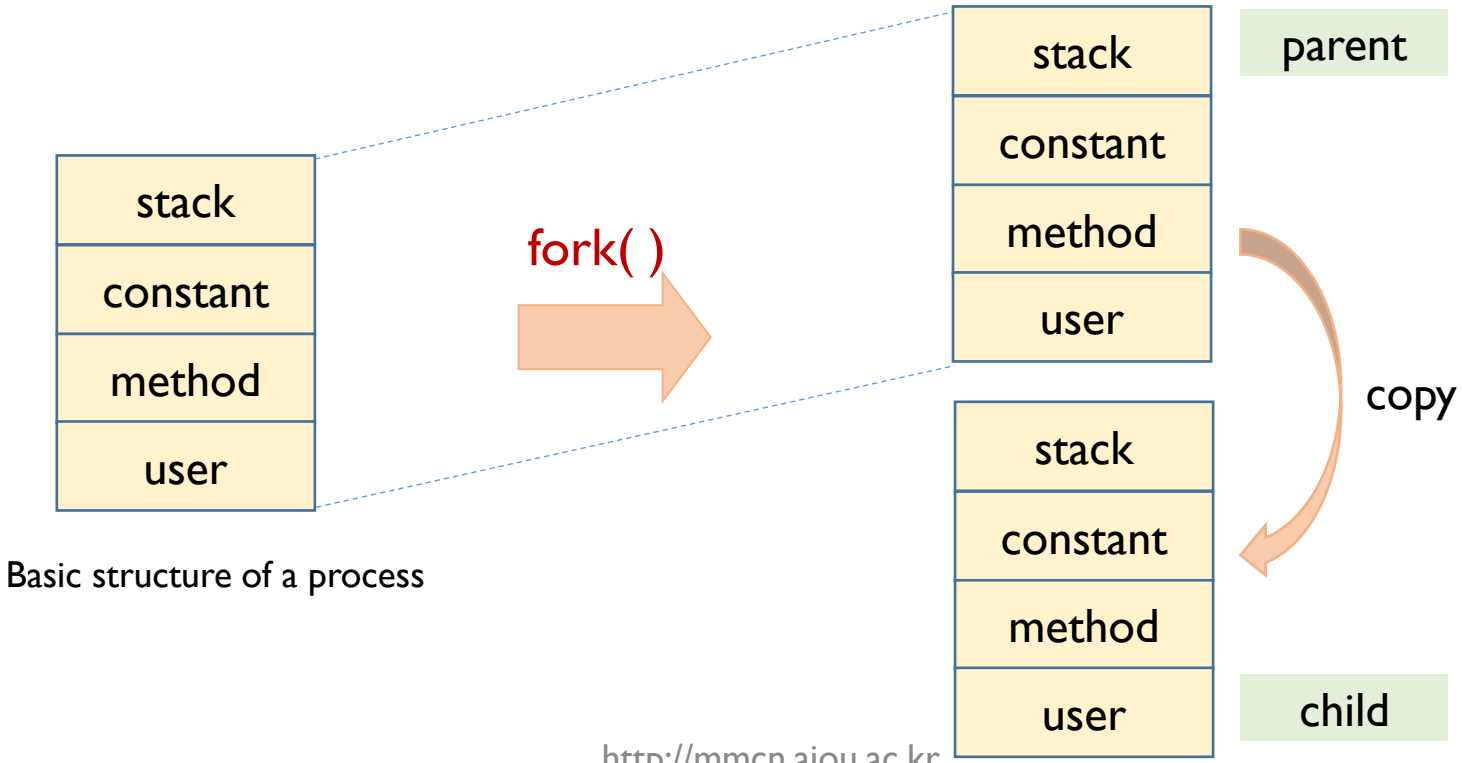
并发服务器的设计

- 并发服务器的可能解决方案
  - 多个进程
    - 分叉服务器进程：多进程
    - 生成一个服务器进程来处理每个客户端连接
    - 内核自动交错多个服务器进程
    - 每个服务器进程都有自己的私有地址空间
  - 多线程
    - 线程服务器进程：多线程方法
    - 创建一个服务器线程来处理每个客户端连接
    - 内核自动交错多个服务器线程
    - 所有线程共享相同的地址空间
  - 使用 select() 调用的一个进程：I/O 多路复用
  - 一个进程使用 epoll() 调用：I/O 多路复用 + 事件
  - 信号驱动I/O（实时信号）

使用 FOLK()函数的多个进程

fork( )

- fork( )函数
  - UNIX 中创建新进程的唯一方法
    - 它可以实现进程级并行的方式
    - 新创建的子进程同时拥有其父进程环境的副本，并共享所有打开的文件描述符.



fork( )

❖ fork( )函数

```
#include <unistd.h>

pid_t   fork ( void );
```

- return values
  - ➢ in the calling (parent) process
    - process ID of the newly created (the child)
  - ➢ in the child process
    - 0

fork( )示例代码(1/2)

```
pid_t        pid;
int          listenfd, connfd, retval;

// create socket
listenfd = socket ( PF_INET, SOCK_STREAM, 0 );

// server address/port : omitted
…

// bind socket
retval = bind ( listenfd, … );

// listening
retval = listen ( listenfd, BACKLOG);
```

fork( )示例代码(2/2)

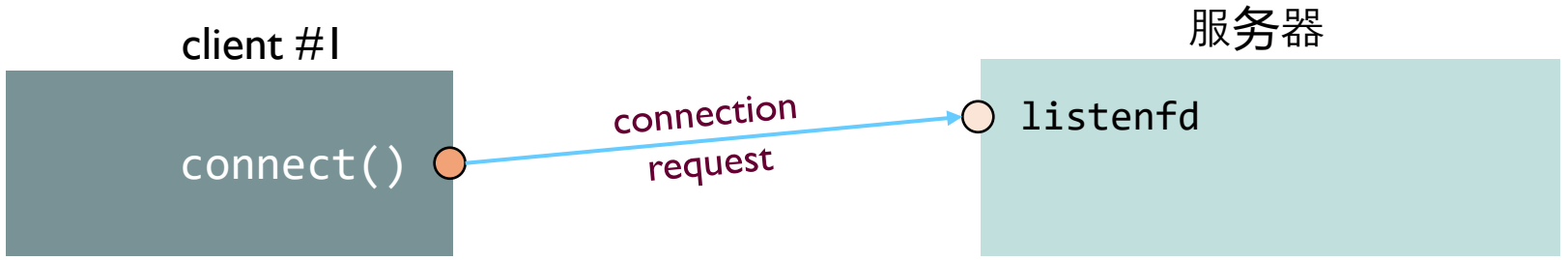
```
for ( ; ; ) {
    connfd = accept ( listenfd,  … );

    if ( ( pid = fork( ) ) == 0 ) {
        close ( listenfd );              // child closes listening socket
        do_child_process ( connfd ) ; // process the request
        close ( connfd ) ;               // close the child socket
        exit ( 0 );                      // terminate the child process
    }

    close( connfd );                     // parent closes connected socket
    …                                    // write a code to clean up all zombies here
}
```

return pid=0: the child process

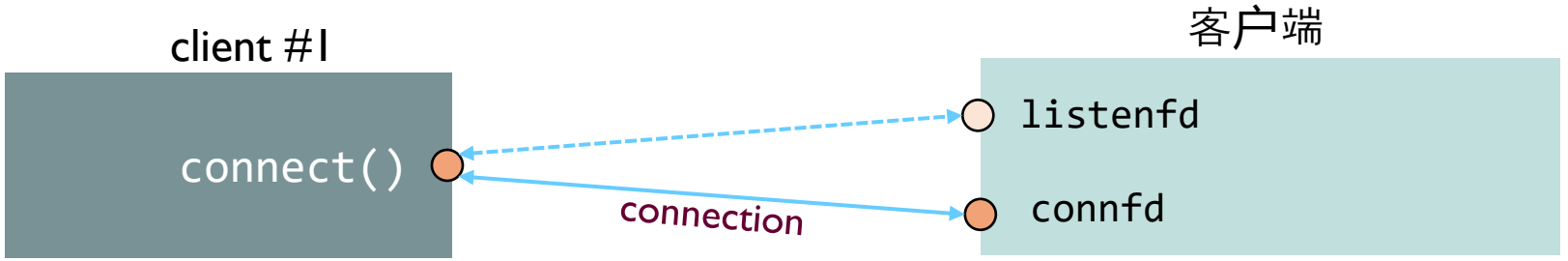
routine for the child process

fork( )图示

- 客户端和并发服务器的状态(1/3)
  - 在 accept() 调用返回之前



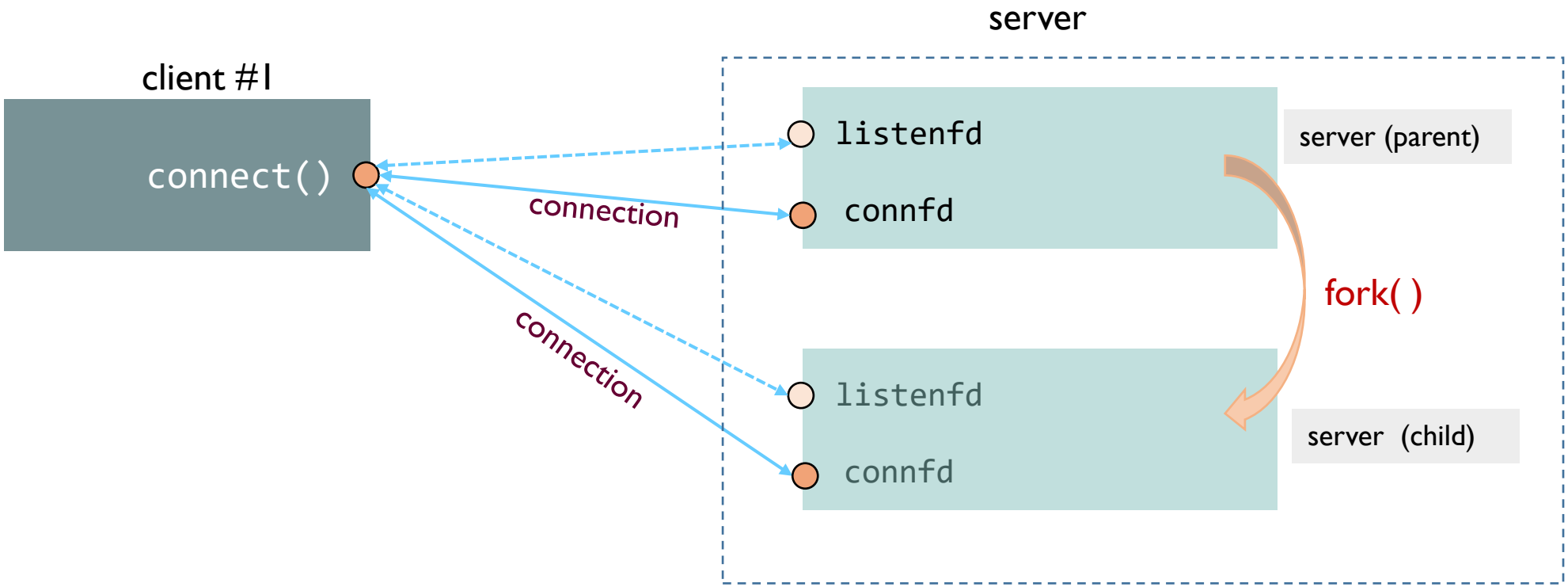
  - 从 accept() 返回后



ref: W. Stevens, UNIX Networking Programming, Vol. 1

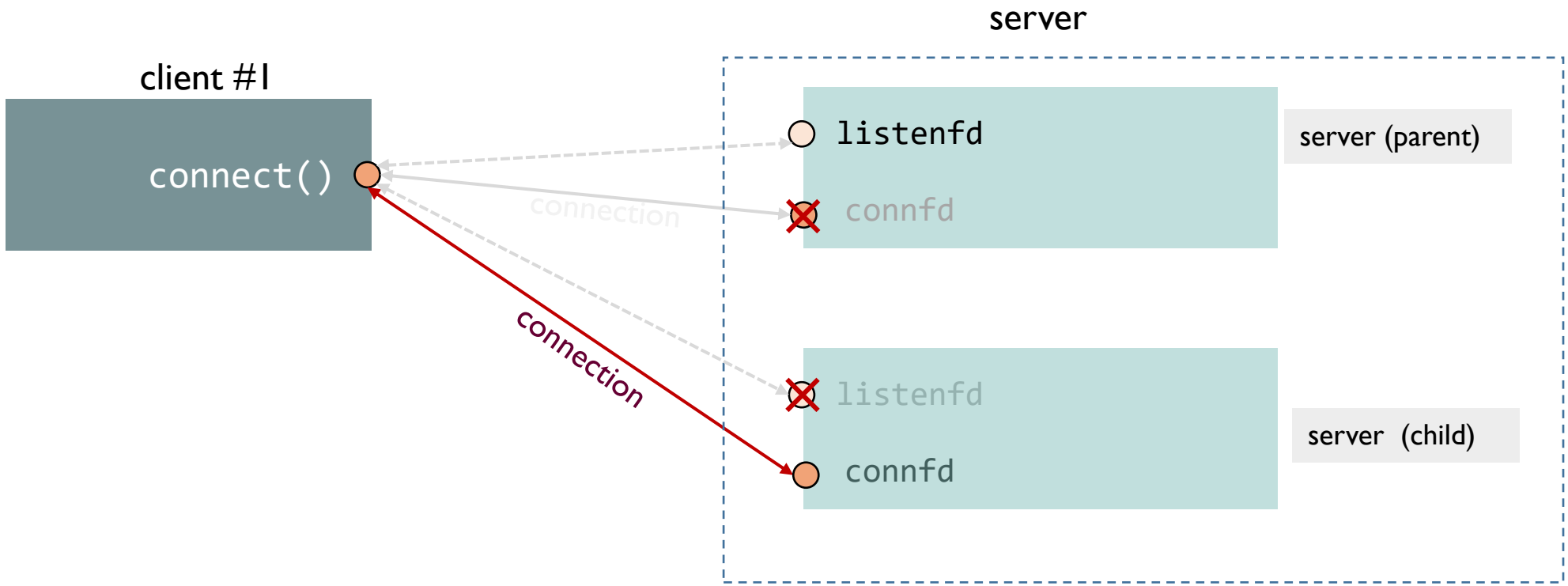
fork( )图示

- 客户端和并发服务器的状态(2/3)
  - fork() 返回后



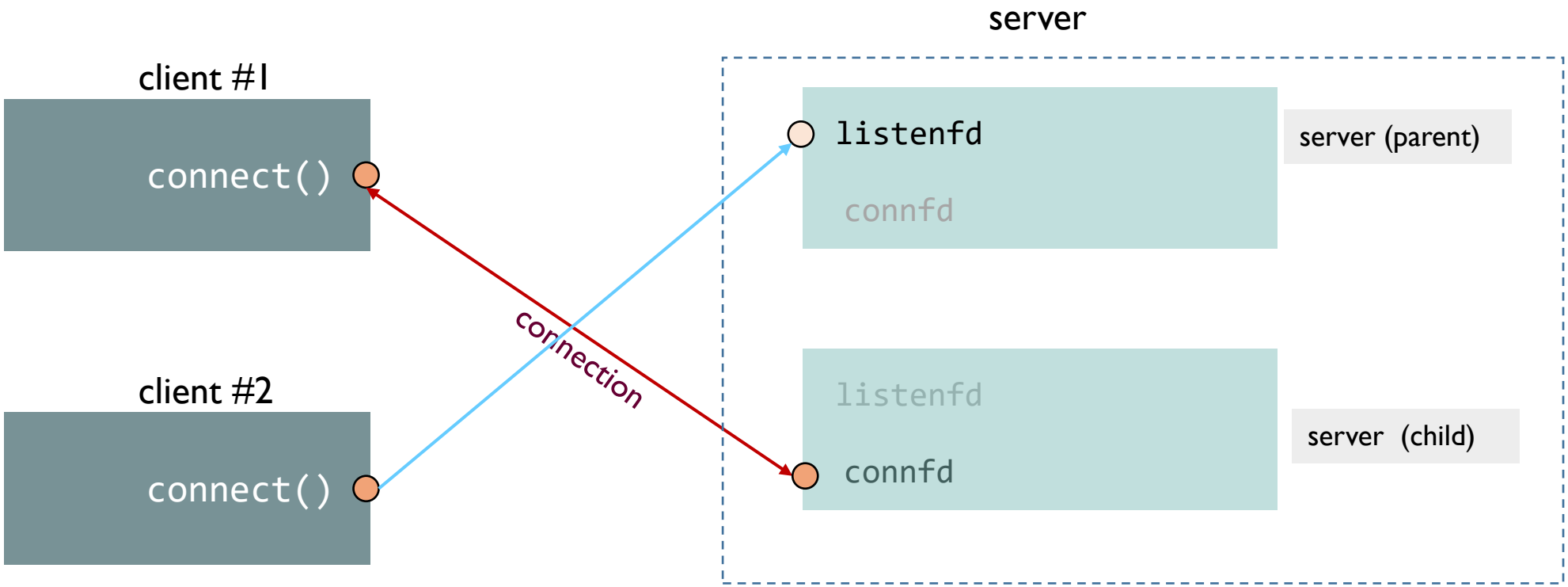
fork( )图示

- 客户端和并发服务器的状态(3/3)
  - 父子进程都关闭相应的套接字后



fork( )图示

- 客户端和并发服务器的状态(3/3)
  - 父子进程都关闭相应的套接字后



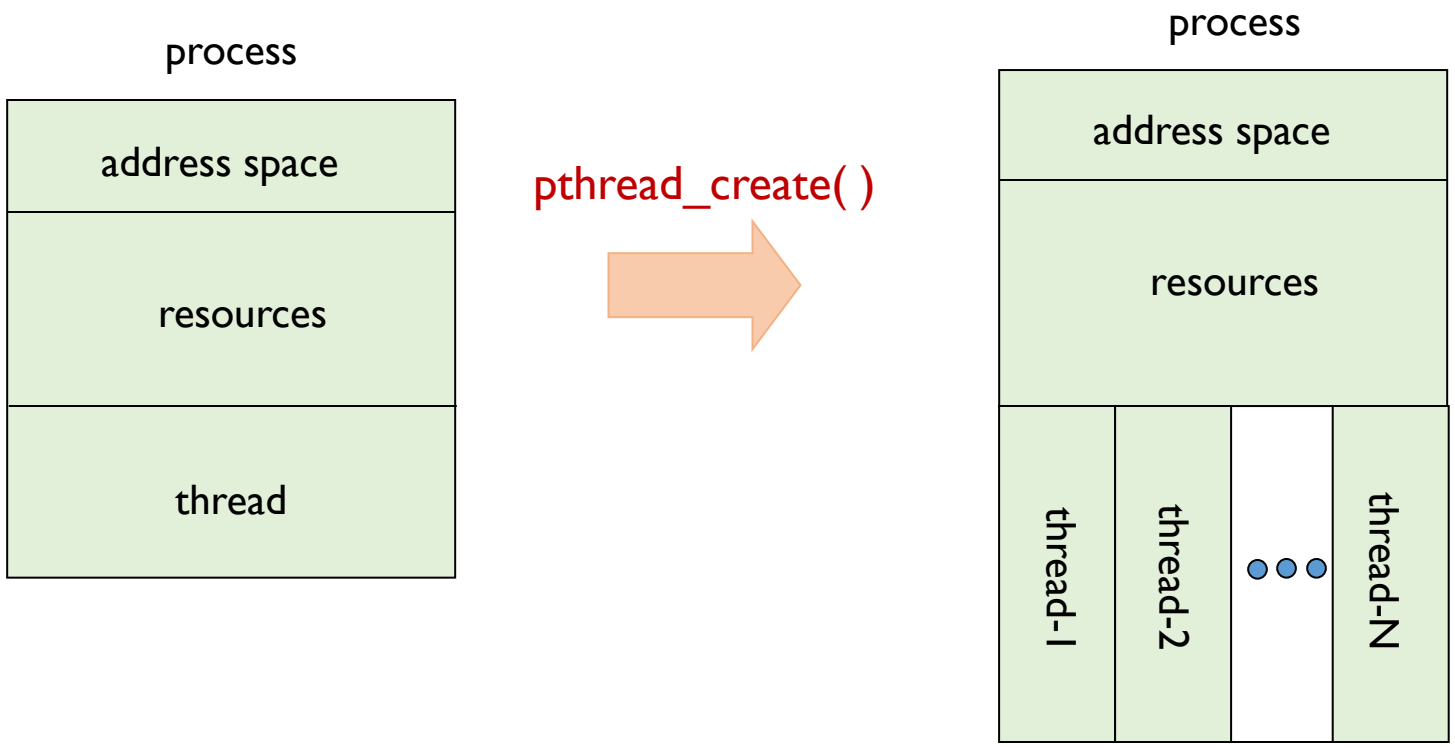
使用fork( )的CS

- 使用folk()为多个客户提供服务
  - 为多客户端进程提供服务的最简单方法
  - 缺点
    - 复制进程的开销更大
    - 流程间信息共享困难
    - 需要更多CPU资源
  - 在 Windows 操作系统中，不支持 folk( )

使用 THREAD 的多线程

Thread

- Thread
  - 同一进程中的线程共享地址空间、打开文件等



Thread vs. Process

- 使用多个进程的并发服务器的局限性
  - 流程间信息共享困难
    - 必须使用IPC (Inter Process Communication)
  - 使用 fork() 创建进程
    - 相对昂贵
    - 耗时
- Thread的优点
  - 轻松在线程间共享信息
    - 将数据复制到共享（全局或堆）变量中
    - 必须采用同步技术
  - 比进程创建更快
    - typically, ten times faster or better

Thread vs. Process

- Thread的缺点
  - 竞争条件 - 数据保护和调度政策
    - 必须使用互斥和连接
  - 必须考虑线程安全（或者线程安全的方式）
  - 一个线程中的错误可能会损坏进程中的所有线程
  - 竞争使用宿主进程的有限虚拟地址空间
    - 大量线程或需要大量内存的线程??
- Check points
  - *多线程应用程序中的信号需要精心设计*
    - 作为一般原则，通常最好避免在多线程程序中使用信号
  - 所有线程必须运行同一个程序，不同的进程可以运行不同的程序
  - 除了数据之外，线程还共享某些其他信息

Thread函数

- Pthreads
  - 用于多线程编程的一组标准 C 库函数（60 多个函数）
    - IEEE 可移植操作系统接口 (POSIX)，第 1003.1 节标准，1995 年
  - 常见于 UNIX 操作系统（ Solaris、 Linux、 Mac OS X ）
- 基础thread函数
  - pthread_create( ) – 创建一个新线程
  - pthread_self( ) – 返回调用线程的ID
  - pthread_exit( ) – 终止调用线程
  - pthread_join( ) – 让当前调用的线程等待另一个线程终止
  - pthread_detatch( ) – 将线程标记为分离.
  - pthread_kill( ) – 终止指定线程

线程创建

❖ pthread_create( )

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

- 参数
  - thread - 用于存储线程 ID 的缓冲区
  - attr - 新线程的各种属性（默认为 NULL）
  - start - 线程中运行的主要函数
  - arg - 传递给start函数的参数. (i.e., *start(arg)*)
- return value
  - 0 : on success
  - negative values : on error

可连接和可分离线程

❖ 线程状态

- 创建线程时的选项
  - ➢ 可连接的（Joinable）
    - 默认情况下线程是可连接的(joinable)
  - ➢ 可分离的（Detached）
- 可连接的线程(by default)
  - ➢ 调用线程等待另一个线程终止
  - ➢ 连接可连接线程失败会产生“僵尸线程”
    - 每个僵尸线程都会消耗一些系统资源，当积累了足够多的僵尸线程时，将不再可能创建新的线程（或进程）.
- 可分离的线程
  - ➢ 没有线程可以调用其上的连接(join)
  - ➢ 线程终止时所有资源都会被释放

可连接和可分离线程

- pthread_detach ( ) – 将线程标记为分离

```
int pthread_detach ( pthread_t tid);
```

  - 参数
    - tid – 要终止的线程

- pthread_join ( ) - 获取调用线程的ID

```
int pthread_join ( pthread_t tid, void **retval );
```

  - 参数
    - tid – 等待终止的线程 id
    - retval – 输出参数，赋予 pthread_exit() 的值
  - return value
    - 0 on success; an error number on error

可连接和可分离线程

❖ pthread 创建和加入(1/3)

- thread function

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

// thread function
void* thread_fn (void *arg)
{
        int id = (int)arg;
        printf ("thread runs (%d)\n", id);
        sleep (id);                                    // wait for id second
        id *= 1000;
        printf("terminate thread (%d)\n", id);
        return (void *)(id);
}
```

可连接和可分离线程

❖ pthread 创建和加入(2/3)

- main function

```
int main()
{
        pthread_t       t1, t2;
        int             retval;

        pthread_create ( &t1, NULL, thread_fn, (void*)1 );
        pthread_create ( &t2, NULL, thread_fn, (void*)2 );

        pthread_join (t1,(void *)&retval);
        printf("thread join: %d\n",retval);

        pthread_join (t2, (void *)&retval);
        printf("thread join: %d\n",retval);

        return 0;
}
```

- a new thread creates to run thread_fn with arg 1 (or 2)
- thread's id is t1 (or t2)
- no customized attributes

Wait for thread t1 termination, and retrieve return value in value_ptr

可连接和可分离线程

- pthread 创建和加入(3/3)
  - result

```
thread runs (1)

thread runs (2)

terminate thread (1000)

thread join: 1000

terminate thread (2000)

thread join: 2000
```

线程终止

❖ pthread_exit ( ) – 终止调用线程

```
void       pthread_exit (void *retval);
```

- 参数
  - 通过 retval 返回一个值（如果线程可连接）

❖ pthread_self ( ) - 获取调用线程的ID

```
pthread_t     pthread_self ( void );
```

- return value
  - This function always succeeds
  - returns a value via retval that (if the thread is joinable)

Linux上基于线程的Echo CS示例(1/3)

```
#include <pthread.h>

void *ThreadMain(void *arg);

struct ThreadArgs {
    int clntSock;
}
```

Linux上基于线程的Echo CS示例(2/3)

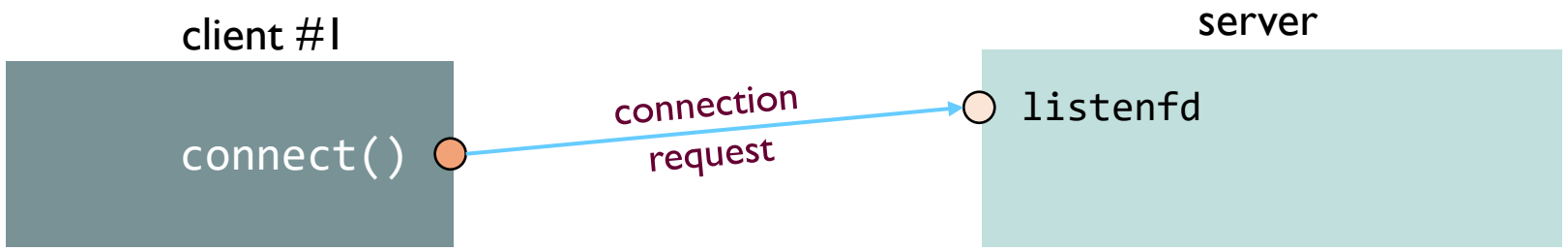
```
int main( )
{
    listenfd = socket ( PF_INET, SOCK_STREAM, 0 );     // create a listen socket
    …                                                  // server address/port : omitted
    bind   ( listenfd, … );                            // bind socket
    listen ( listenfd, BACKLOG);                       // listening

    for ( ; ; ) {
        clntSock = accept ( listenfd, … );

        threadArgs              = (struct ThreadArgs *)malloc( … ); // argument for thread
        threadArgs->clntSock    = clntSock;                         // function

        // create a thread for the accepted client
        pthread_create ( &threadID, NULL, ThreadMain, (void *)threadArgs);
    }
    …
}
```

Linux上基于线程的Echo CS示例(3/3)

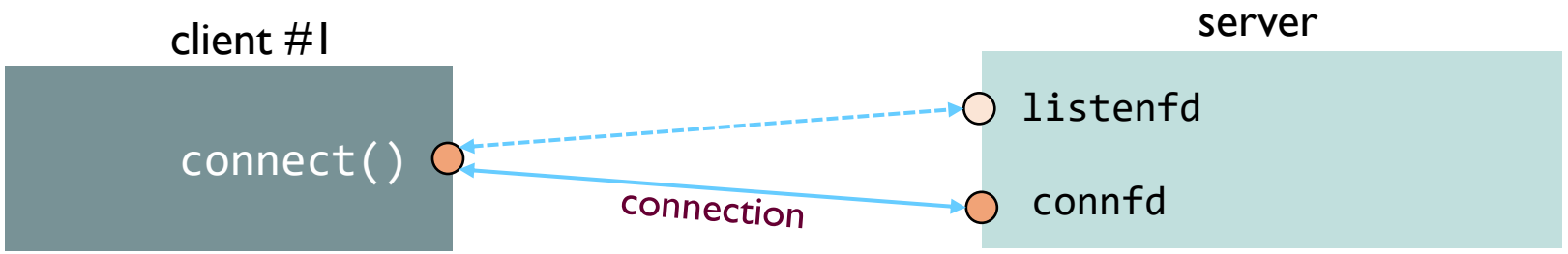
```
void *ThreadMain (void *threadArgs )
{
    int clntSock;

    // make the thread as detached - its resources are deallocated upon return
    pthread_detach( pthread_self( ) );

    // Extract socket file descriptor from argument
    clntSock = ((struct ThreadArgs *)threadArgs)->clntSock;

    free (threadArgs);                  // free dynamically allocate memory for argument

    HandleTCPClient ( clntSock );

    return(NULL);
}
```

使用线程的CS：图示(1/4)

- 客户端和并发服务器的状态
  - 在accept()调用返回之前

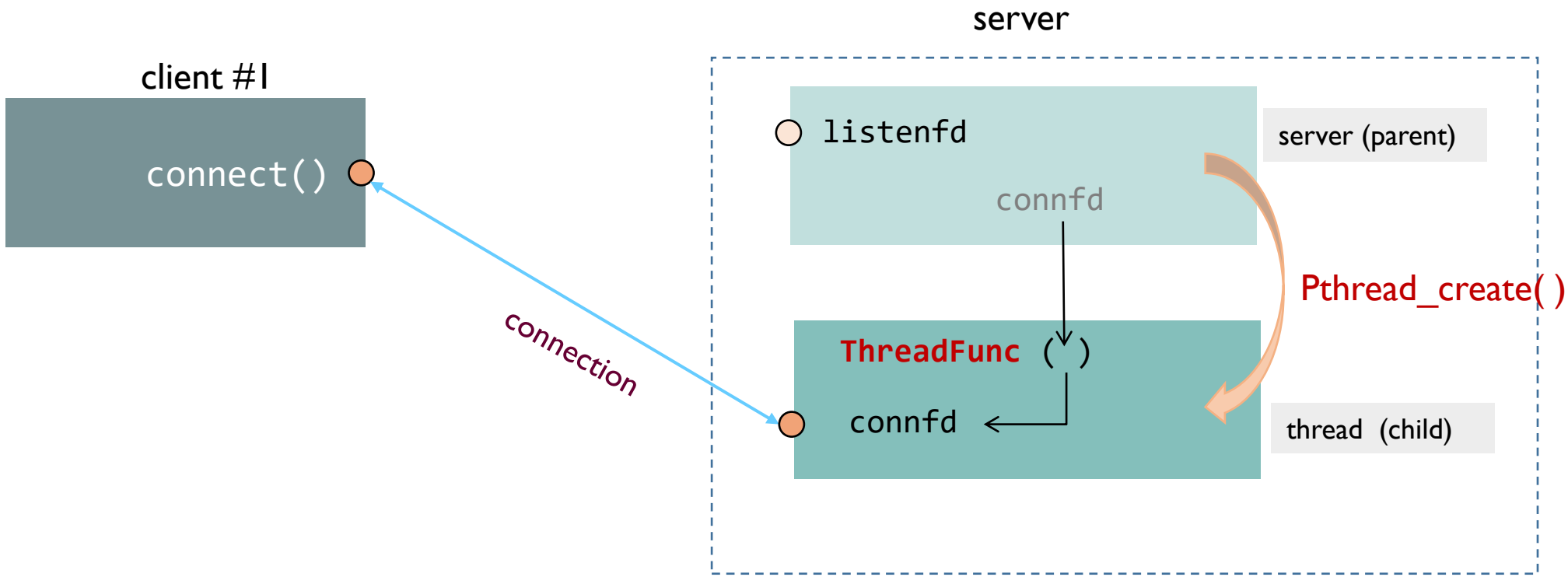


  - 从accept()返回后



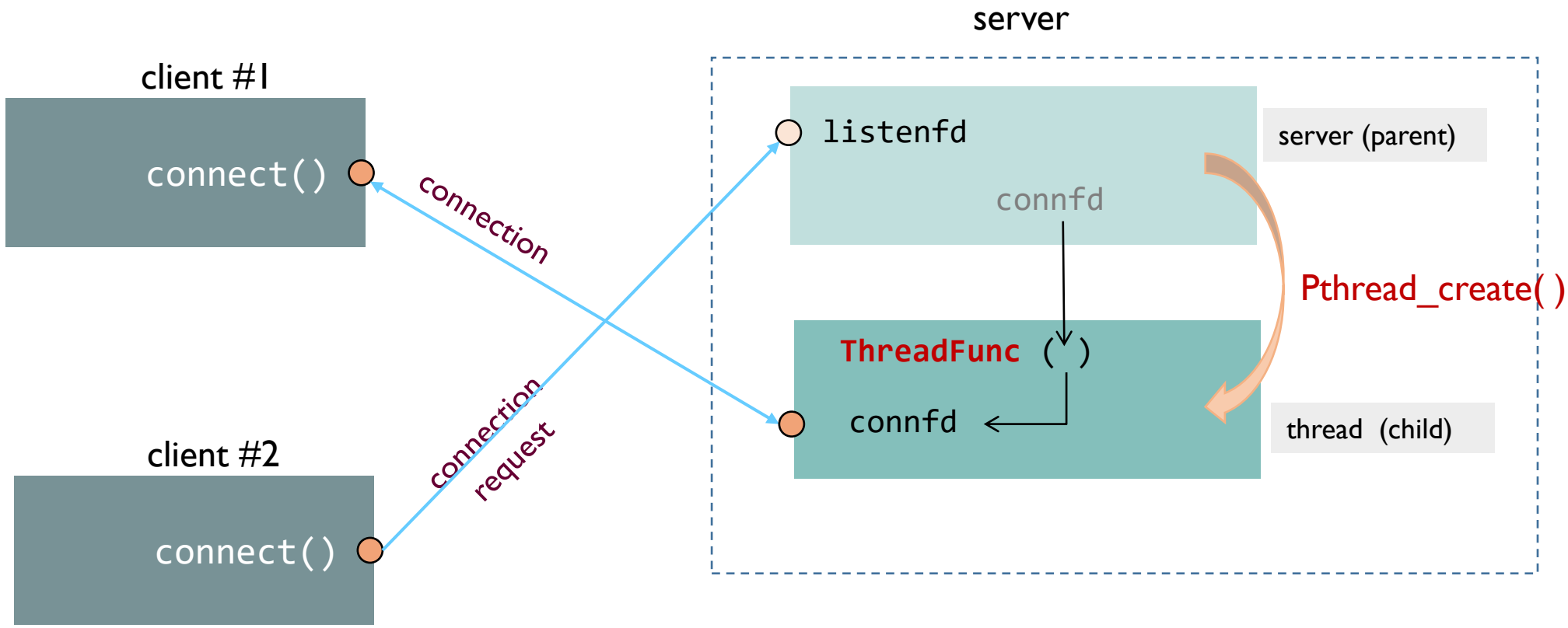
使用线程的CS：图示(2/4)

- 在pthread_create( )之后



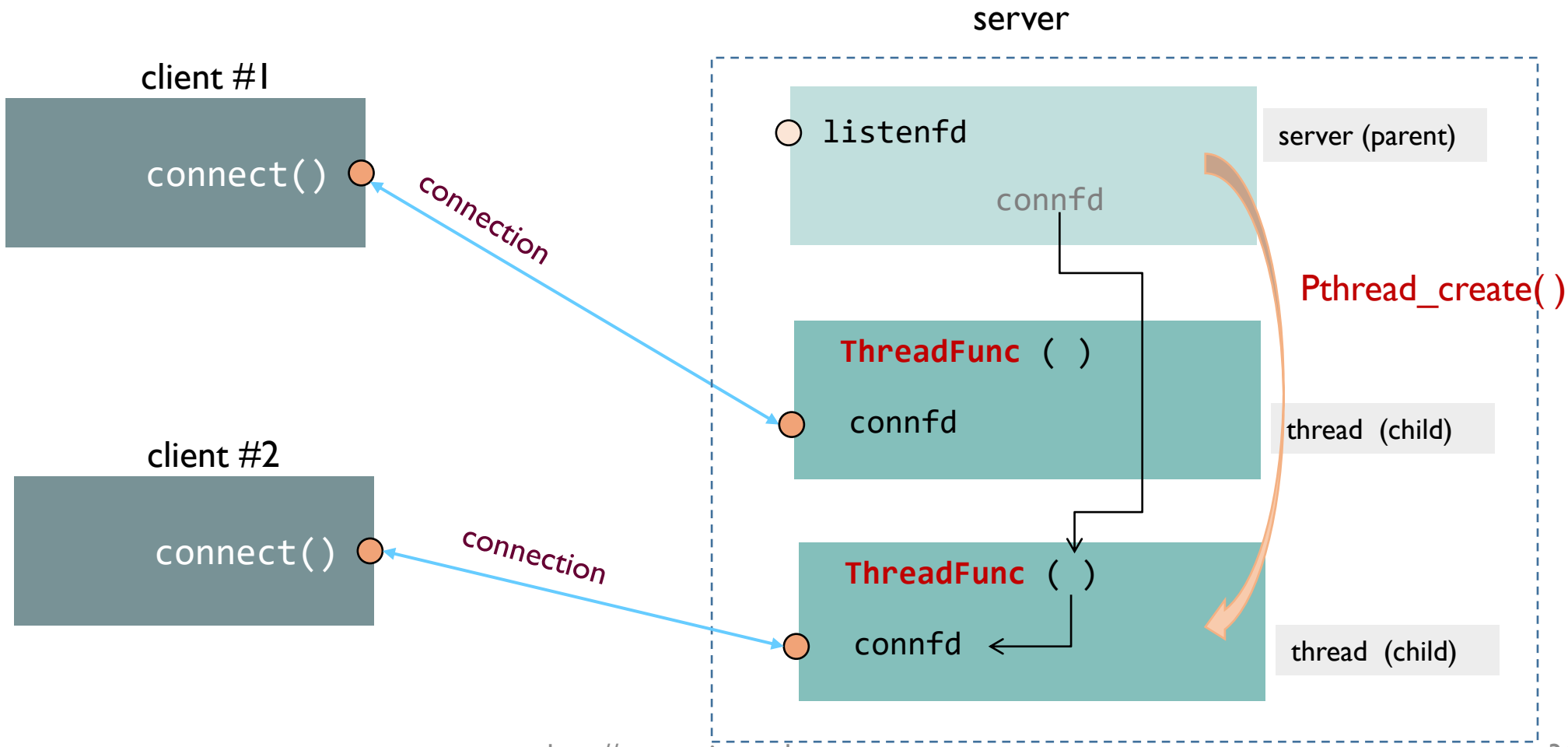
使用线程的CS：图示(3/4)

- 父子线程关闭相应的套接字后



使用线程的CS：图示(4/4)

- 父子线程关闭相应的套接字后



Windows 操作系统中线程的使用

❖ Windows OS中创建线程

```
HANDLE CreateThread (  LPSECURITY_ATTRIBUTES    lpThreadAttributes,
                       SIZE_T                   dwStackSize,
                       LPTHREAD_START_ROUTINE   lpStartAddress,
                       LPVOID                   lpParameter,
                       DWORD                    dwCreationFlags,
                       LPDWORD                  lpThreadId) ;
```

- 参数
  - lpThreadAttributes - 可选指针(NULL by default)
  - dwStackSize - 堆栈的初始大小(0 by default)
  - lpStartAddress - 线程的主要函数
  - lpParameter - 传递给线程函数的参数
  - dwCreationFlags - 控制线程创建的flag
  - lpThreadId - 指向接收线程 ID 的变量的指针.
- return - thread handle on success, NULL on fail

Windows 操作系统中线程的使用

- 线程回调函数
  - CreateThread ()中的lpStartAddress
    - 表示应用程序定义的回调函数的起始地址，该函数用作线程的起始地址
  - ThreadProc
    - 是应用程序定义函数名称的占位符.

```
DWORD WINAPI ThreadProc ( LPVOID lpParameter);
```

```
in main:
    HANDLE  hThread;
    …
    hThread = CreateThread (NULL, 0, MyThreadFunction, pData, 0, &dwThreadId);

DWORD WINAPI MyThreadFunction ( LPVOID lpParam )
{ ... }
```

Windows 操作系统中线程的使用

❖ 关闭一个线程

▪ TerminateThread () 函数

```
BOOL WINAPI TerminateThread (HANDLE hThread, DWORD  dwExitCode);
```

- ➢ 参数
  - hThread：要终止的线程的句柄
  - dwExitCode：线程的退出代码
- ➢ retrun
  - nonzero on success; zero on success

▪ ExitThread( ) 函数

```
VOID WINAPI ExitThread ( DWORD dwExitCode  );
```

- ➢ 参数
  - dwExitCode：线程的退出代码
- ➢ no return value

Thread CS by Windows OS (1/3)

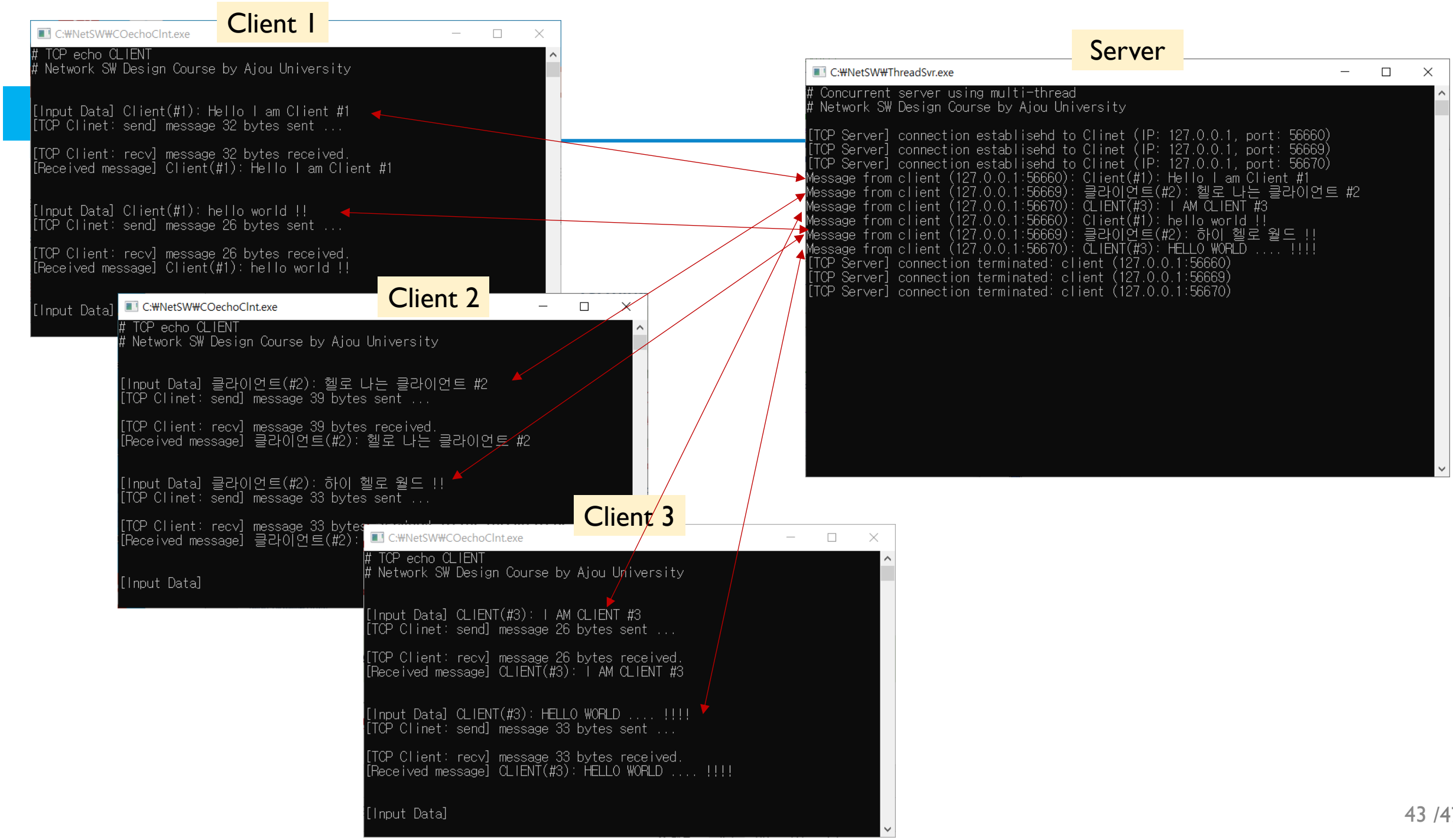
```
int main(int argc, char* argv[])
{
    int                 retval, addrlen;
    SOCKET              listen_sock, client_sock;
    SOCKADDR_IN serveraddr, clientaddr;
    HANDLE              hThread;
    DWORD               ThreadId;

    WSADATA wsa;
    iWSAStartup(MAKEWORD(2,2), &wsa) ;   // initializing winsock

    listen_sock = socket (AF_INET, SOCK_STREAM, 0);

    ZeroMemory(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family       = AF_INET;
    serveraddr.sin_port         = htons (9000);
    serveraddr.sin_addr.s_addr  = htonl (INADDR_ANY);

    retval = bind (listen_sock, (SOCKADDR *)&serveraddr, sizeof(serveraddr));

    retval = listen (listen_sock, SOMAXCONN);
```

Thread CS by Windows OS (2/3)

```
    while(1) {
        addrlen = sizeof(clientaddr);
        client_sock = accept (listen_sock, (SOCKADDR *)&clientaddr, &addrlen);

        printf("[TCP Server] connection established to Client (IP: %s, port: %d)\n",
                inet_ntoa(clientaddr.sin_addr), ntohs(clientaddr.sin_port));

        // Creating Thread
        hThread = CreateThread ( NULL,  0,  ProcessClient,
                                 (LPVOID)client_sock, 0, &ThreadId);
        CloseHandle (hThread);
    }

    // closesocket()
    closesocket (listen_sock);

    // 윈속 종료
    WSACleanup();
    return 0;
}
```

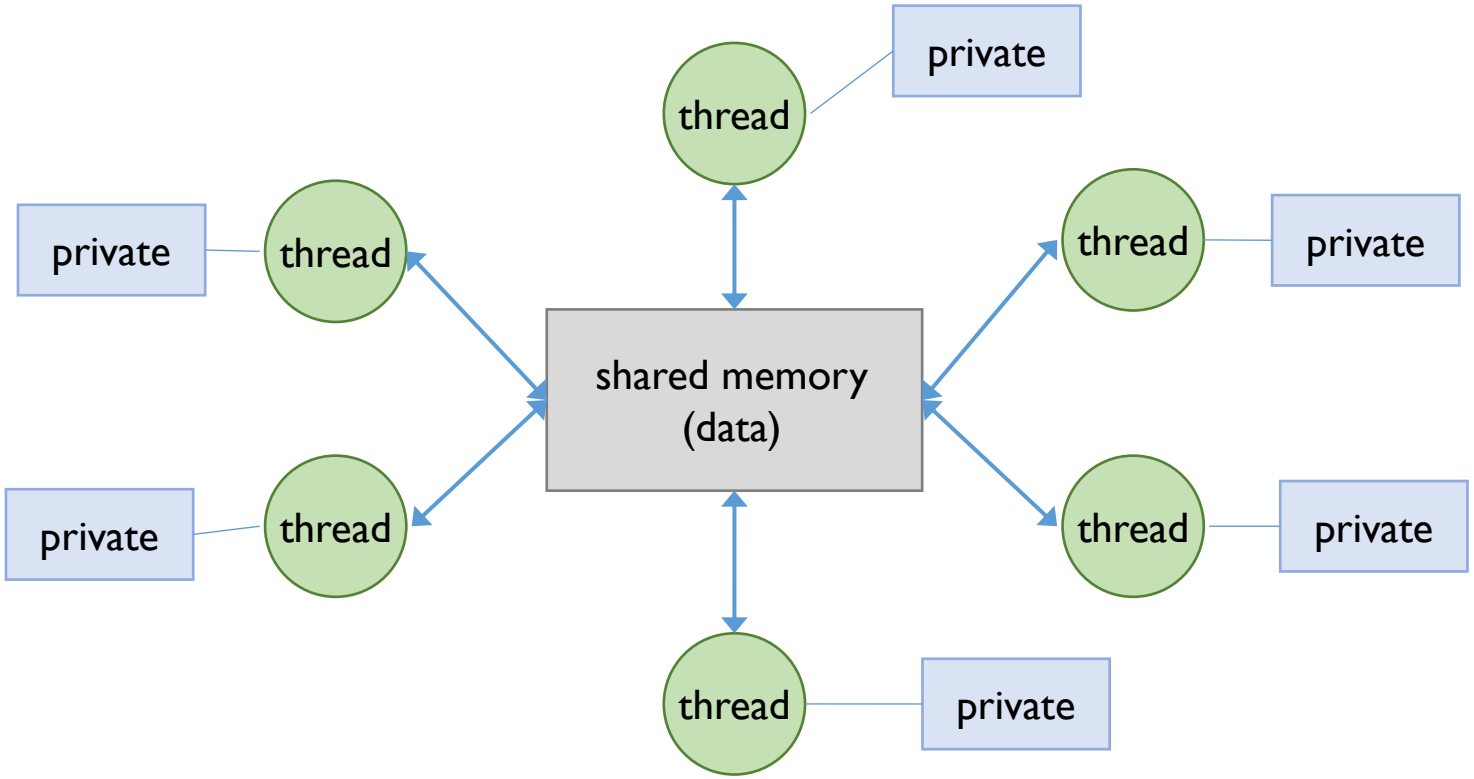
Thread CS by Windows OS (3/3)

```
DWORD WINAPI    ProcessClient (LPVOID arg )  {
    SOCKADDR_IN clientaddr;
    char        buf[BUFSIZE+1];
    int         addrlen, retval;
    SOCKET      client_sock = (SOCKET)arg;

    addrlen = sizeof(clientaddr);                          // obtain client information
    getpeername (client_sock, (SOCKADDR *)&clientaddr, &addrlen);

    while(1) {
        retval = recv (client_sock, buf, BUFSIZE, 0);      // receive data
        buf[retval] = '\0';                                // display the received data
        printf("Message from client (%s:%d): %s \n", inet_ntoa (clientaddr.sin_addr),
                        ntohs (clientaddr.sin_port), buf);
        retval = send (client_sock, buf, retval, 0);       // echo the data
    }

    closesocket (client_sock);
    …                                                      // displaying some information
    return 0;
}
```


Client 1
C:₩NetSW₩COechoClnt.exe
TCP echo CLIENT
Network SW Design Course by Ajou University
[Input Data] Client(#1): Hello I am Client #1
[TCP Clinet: send] message 32 bytes sent ...
[TCP Client: recv] message 32 bytes received.
[Received message] Client(#1): Hello I am Client #1
[Input Data] Client(#1): hello world !!
[TCP Clinet: send] message 26 bytes sent ...
[TCP Client: recv] message 26 bytes received.
[Received message] Client(#1): hello world !!
[Input Data]
Client 2
C:₩NetSW₩COechoClnt.exe
TCP echo CLIENT
Network SW Design Course by Ajou University
[Input Data] 클라이언트(#2): 헬로 나는 클라이언트 #2
[TCP Clinet: send] message 39 bytes sent ...
[TCP Client: recv] message 39 bytes received.
[Received message] 클라이언트(#2): 헬로 나는 클라이언트 #2
[Input Data] 클라이언트(#2): 하이 헬로 월드 !!
[TCP Clinet: send] message 33 bytes sent ...
[Input Data]
Client 3
C:₩NetSW₩COechoClnt.exe
TCP echo CLIENT
Network SW Design Course by Ajou University
[Input Data] CLIENT(#3): I AM CLIENT #3
[TCP Clinet: send] message 26 bytes sent ...
[TCP Client: recv] message 26 bytes received.
[Received message] CLIENT(#3): I AM CLIENT #3
[Input Data] CLIENT(#3): HELLO WORLD .... !!!!
[TCP Clinet: send] message 33 bytes sent ...
[TCP Client: recv] message 33 bytes received.
[Received message] CLIENT(#3): HELLO WORLD .... !!!!
[Input Data]
Server
C:₩NetSW₩ThreadSvr.exe
Concurrent server using multi-thread
Network SW Design Course by Ajou University
[TCP Server] connection establisehd to Clinet (IP: 127.0.0.1, port: 56660)
[TCP Server] connection establisehd to Clinet (IP: 127.0.0.1, port: 56669)
[TCP Server] connection establisehd to Clinet (IP: 127.0.0.1, port: 56670)
Message from client (127.0.0.1:56660): Client(#1): Hello I am Client #1
Message from client (127.0.0.1:56669): 클라이언트(#2): 헬로 나는 클라이언트 #2
Message from client (127.0.0.1:56670): CLIENT(#3): I AM CLIENT #3
Message from client (127.0.0.1:56660): Client(#1): hello world !!
Message from client (127.0.0.1:56669): 클라이언트(#2): 하이 헬로 월드 !!
Message from client (127.0.0.1:56670): CLIENT(#3): HELLO WORLD .... !!!!
[TCP Server] connection terminated: client (127.0.0.1:56660)
[TCP Server] connection terminated: client (127.0.0.1:56669)
[TCP Server] connection terminated: client (127.0.0.1:56670)

线程同步

- 线程同步问题
  - 线程可以读取/修改/写入的共享数据
    - 多个线程不会尝试同时修改数据
    - 当一个线程正在修改数据时，另一个线程不会尝试读取数据



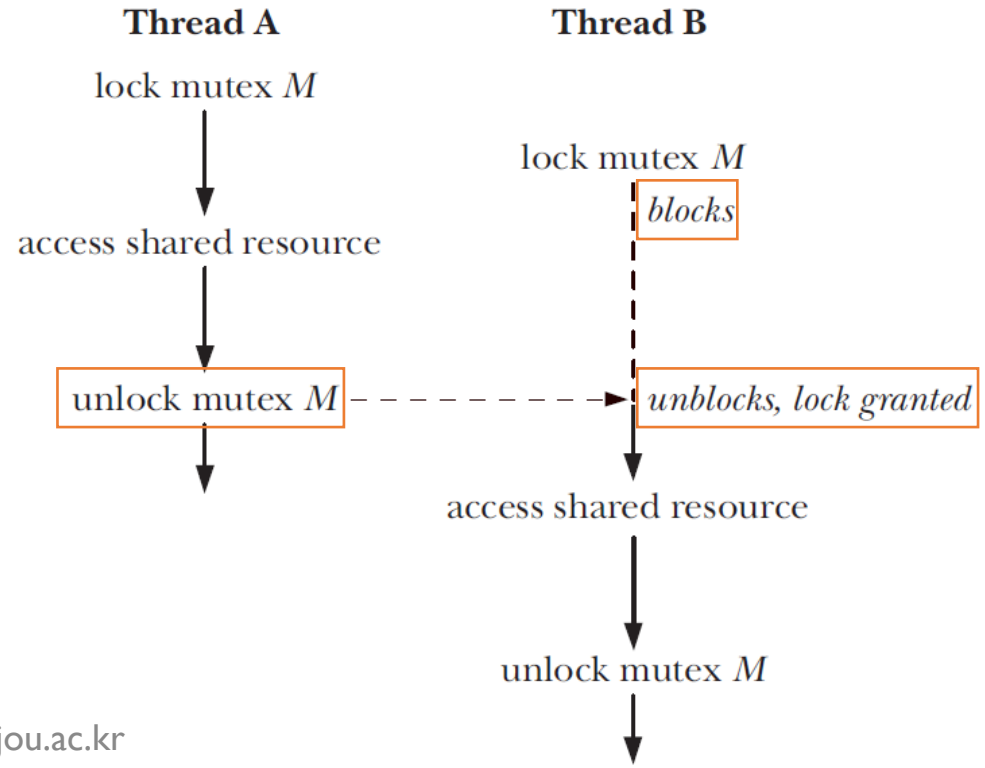
线程同步

- 高级同步原语
  - Mutual Exclusion （互斥）
  - Conditional Variable （条件变量）
  - Semaphore （信号量）
  - Read-Write Lock （读写锁）

线程同步

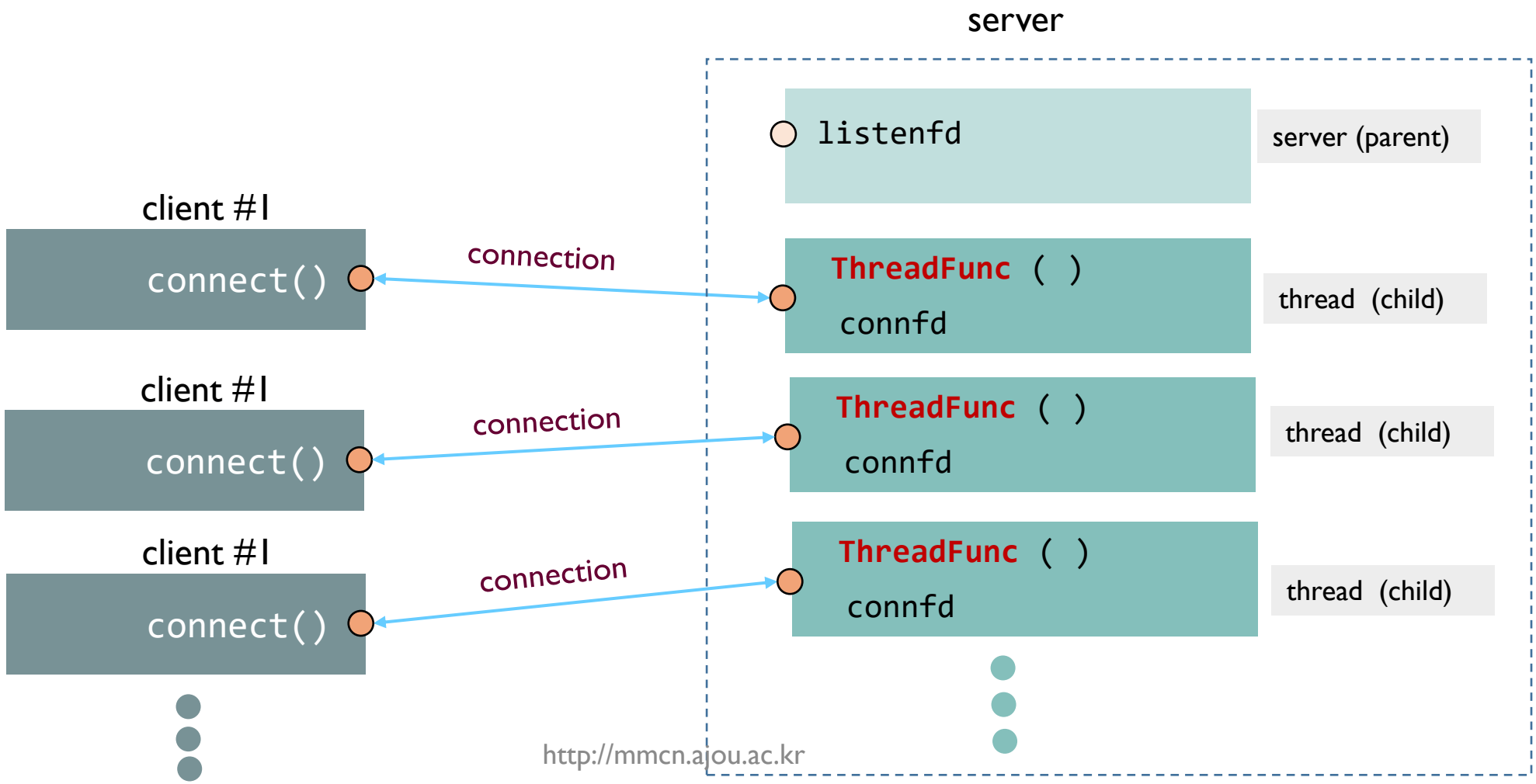
- Mutual Exclusion (Mutex) 互斥
  - Ensure that only one
- 互斥锁的两种状态——锁定和解锁（获取和释放）
  - 任何时候，最多只有一个线程可以持有互斥锁

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```



讨论

❖ 在我们使用线程的并发服务器中，是否会发生同步问题?



Questions ?

IoT & MR convergence

Future Internet

MMcN

Intelligent Networking

Mobile **Multimedia Convergence** Network Lab.

Homepage: http://mmcn.ajou.ac.kr

facebook: http://www.facebook.com/#!/groups/190702010947314