C Developer

Advanced Concepts

SUPINFO

Course Objectives

- ✓ Make socket connection
- ✓ Make a TCP/UDP server and client
- ✓ Manipulate threads
- ✓ Approach the basics of the concurrent programming

Course Plan

1. Sockets
2. Threads

1. Sockets

1. Sockets

Introduction

- Sockets are data flows allowing local or remote machines to communicate over the network

- They use TCP or UDP, and implement many protocols such as IMAP, HTTP, FTP, SSH, *etc*.

- They are available on all platforms and are cross language

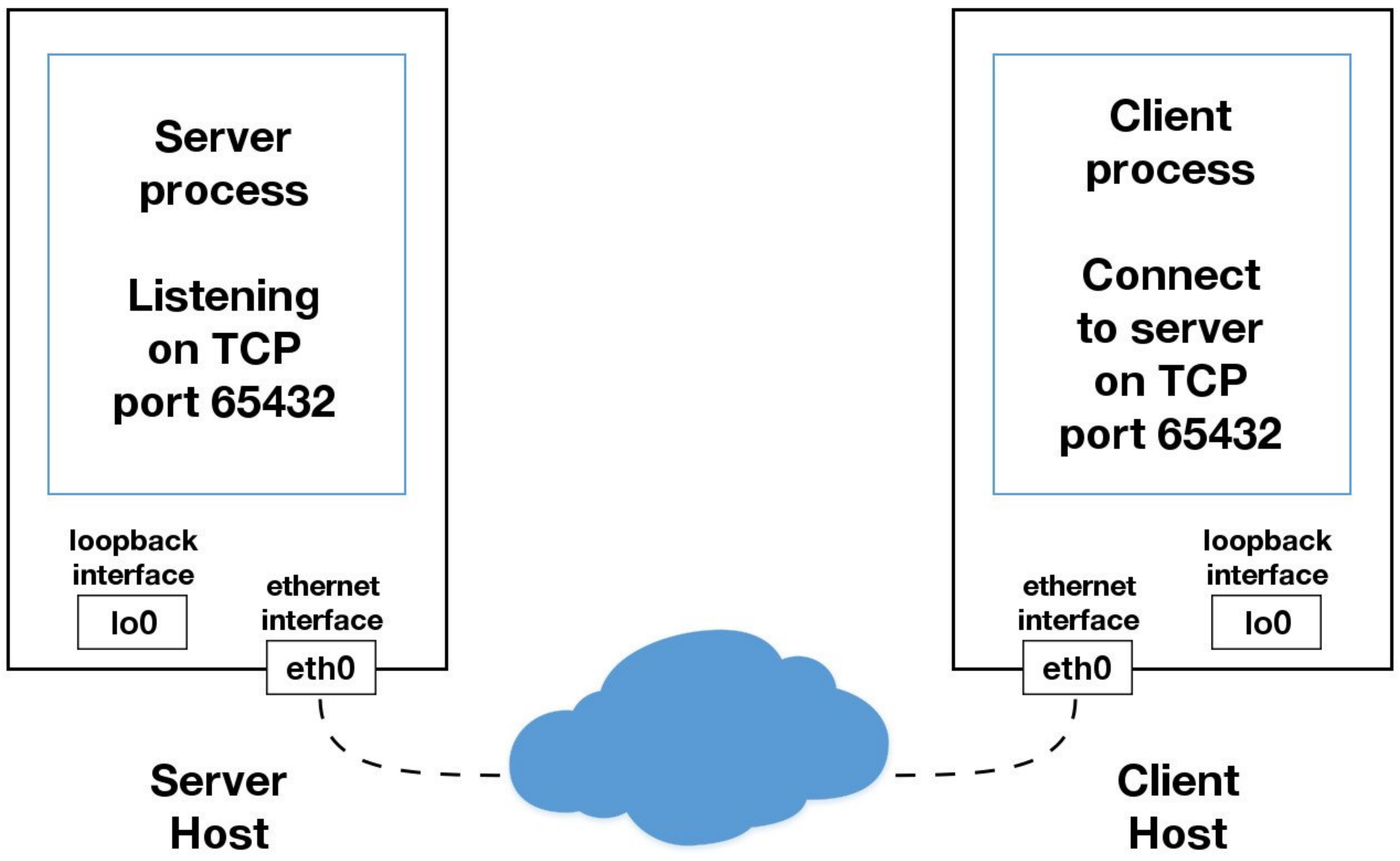
1. Sockets

Introduction

1. Each machine creates a socket
2. Each socket will be associated to a port
3. The two sockets will be explicitly connected
4. Each machine reads/writes to its socket
5. The data goes from one socket to another through the network
6. Once completed, each machine closes its socket

1. Sockets

Introduction



1. Sockets

Introduction

- Each operating system implements them
- The same functions are used for the overall operation
- Some functions and libraries can be added to meet the specificities
- To use, it is necessary to add the libraries, define the structures then the parameters

1. Sockets

Libraries

```
#include <winsock2.h> //Windows library


//Linux libraries
#include <sys/types.h> //Data types of system calls

#include <sys/socket.h> //Structures and parameters

#include <netinet/in.h> //Structures to use Internet domain name

#include <netdb.h> //To resolve domain name

#include <arpa/inet.h> //Address manipulation functions

#include <unistd.h> //To close
```

1. Sockets

Structures

- Basic information:

```
struct sockaddr {
    sa_family_t sa_family; //Address family
    char sa_data[14]; //Protocol address
};
```

- Address families:

```
#define AF_UNSPEC     0        /* unspecified */
#define AF_LOCAL      1        /* local to host (pipes, portals) */
#define AF_UNIX      AF_LOCAL    /* backward compatibility */
#define AF_INET       2        /* internetwork: UDP, TCP, etc. */
#define AF_IMPLINK    3        /* arpanet imp addresses */
#define AF_PUP        4        /* pup protocols: e.g. BSP */
#define AF_CHAOS      5        /* mit CHAOS protocols */
#define AF_NS         6        /* XEROX NS protocols */
#define AF_ISO        7        /* ISO protocols */
#define AF_OSI       AF_ISO
#define AF_ECMA       8        /* european computer manufacturers */
#define AF_DATAKIT    9        /* datakit protocols */
#define AF_CCITT      10       /* CCITT protocols, X.25 etc */
#define AF_SNA        11       /* IBM SNA */
#define AF_DECnet     12       /* DECnet */
#define AF_DLI        13       /* DEC Direct data link interface */
#define AF_LAT        14       /* LAT */
#define AF_HYLINK     15       /* NSC Hyperchannel */
#define AF_APPLETALK    16       /* Apple Talk */
#define AF_ROUTE      17       /* Internal Routing Protocol */
```

```
#define AF_LINK      18       /* Link layer interface */
#define pseudo_AF_XTP    19       /* eXpress Transfer Protocol (no AF) */
#define AF_COIP      20       /* connection-oriented IP, aka ST II */
#define AF_CNT       21       /* Computer Network Technology */
#define pseudo_AF_RTIP   22       /* Help Identify RTIP packets */
#define AF_IPX       23       /* Novell Internet Protocol */
#define AF_INET6     24       /* IP version 6 */
#define pseudo_AF_PIP    25       /* Help Identify PIP packets */
#define AF_ISDN      26       /* Integrated Services Digital Network*/
#define AF_E164      AF_ISDN     /* CCITT E.164 recommendation */
#define AF_NATM      27       /* native ATM access */
#define AF_ARP       28       /* (rev.) addr. res. prot. (RFC 826) */
#define pseudo_AF_KEY    29       /* Internal key management protocol  */
#define pseudo_AF_HDRCMPLT 30        /* Used by BPF to not rewrite hdrs
                                        in interface output routine */

#define AF_BLUETOOTH    31
#define AF_IEEE80211    32       /* IEEE80211 */
#define AF_QNET      33       /* Used for Qnet interface detection */
#define AF_MAX       34
```

1. Sockets

Structures

- IPv4 address format:

```
struct sockaddr_in {
    uint8_t sin_len; //Length
    sa_family_t sin_family; //AF_INET to use TCP/UDP
    in_port_t sin_port; //Port number
    struct in_addr sin_addr; //IPv4 address
    unsigned char sin_zero[8]; //Filled with 0
};

struct in_addr {
    in_addr_t s_addr; //IPv4 address
};
```

1. Sockets

Structures

- IPv6 address format:

```
struct sockaddr_in6 {
    sa_family_t sin6_family; //AF_INET6 to use TCP/UDP
    in_port_t sin6_port; //Port number
    uint32_t sin6_flowinfo; //IPv6 flow information
    struct in6_addr sin6_addr; //IPv6 address
    uint32_t sin6_scope_id; //Scope ID
};

struct in6_addr {
    unsigned char s6_addr[16]; //IPv6 address
};
```

1. Sockets

Structures

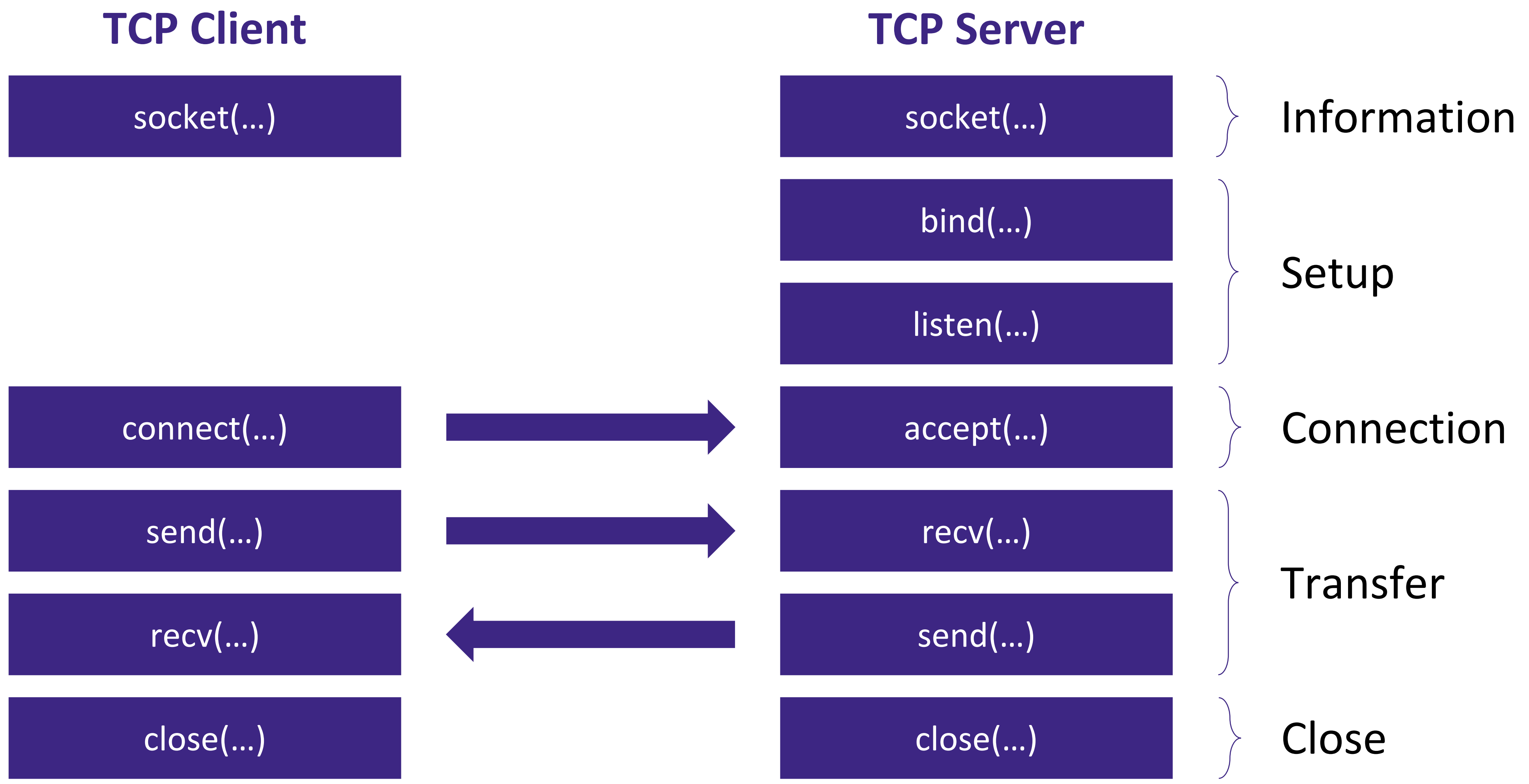
- Domain name:

```
#define h_addr h_addr_list[0] //Compatibility

struct hostent {
    char *h_name; //Hostname
    char **h_aliases; //Alias list
    int h_addrtype; // Address type (IPv4/IPv6)
    int h_length; //Address length
    char **h_addr_list; //Address list
};
```

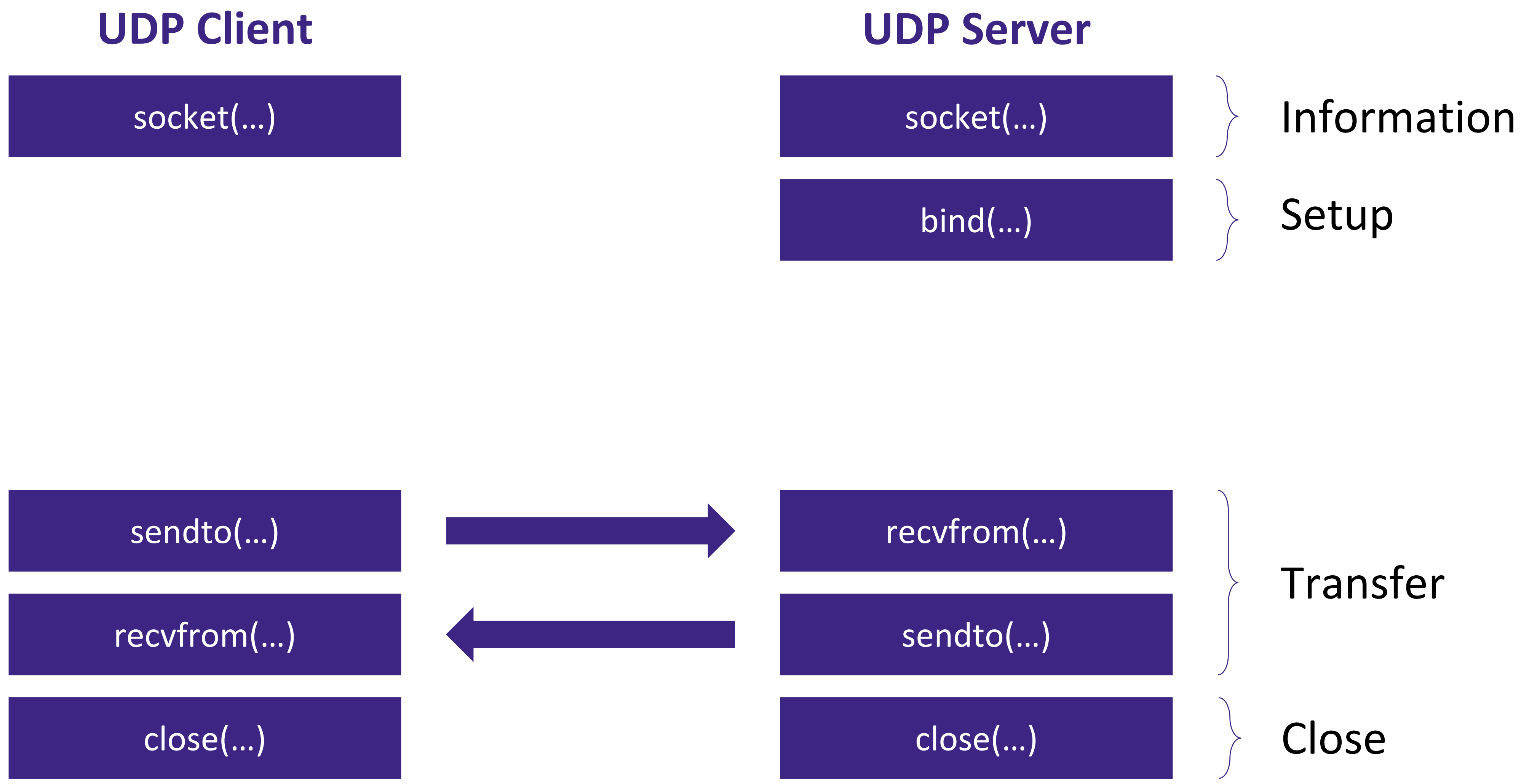
1. Sockets

Functions



1. Sockets

Functions



1. Sockets

Functions

- To define a socket, use the **socket** function

- Function signature:

```
int socket(int domain, int type, int protocol);
```

  - Define IPv4 (**AF_INET**) or IPv6 (**PF_INET6**)
  - Define TCP (**SOCK_STREAM**) or UDP (**SOCK_DGRAM**)
  - It returns the socket descriptor or **-1** in case of failure

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
```

1. Sockets

Functions

- To close the socket descriptor, use the **close** function

- Function signature:

```
int close(int fd);
```

  – It returns **0** or **-1** in case of failure

```
close(sock);
```

1. Sockets

Functions

- To send using connected TCP mode, use the **send** function

- Function signature:

```
int send(int fd, const void *msg, size_t len, int flags);
```

  - It returns the number of characters sent or **-1** in case of failure

```
char buff[1024];
send(sock, buff, strlen(buff), 0);
```

1. Sockets

Functions

- To send using unconnected UDP mode, use the **sendto** function

- Function signature:

```
int sendto(int fd, const void *msg, size_t len, int flags,
           const struct sockaddr *to, socklen_t tolen);
```

  – It returns the number of characters sent or **-1** in case of failure

```
char buff[1024];
sendto(sock, buff, strlen(buff), 0, (struct sockaddr *) &servAddr,
            sizeof(servAddr));
```

1. Sockets

Functions

- To receive using connected TCP mode, use the **recv** function

- Function signature:

```
int recv(int fd, void *buf, int len, unsigned int flags);
```

  – It returns the number of characters received or **-1** in case of failure

```
char buff[1024];
recv(sock, buff, sizeof(buff)-1, 0);
```

1. Sockets

Functions

- To receive using unconnected UDP mode, use the **recvfrom** function

- Function signature:

```
int recvfrom(int fd, void *buf, int len, unsigned int flags,
             struct sockaddr *from, socklen_t *fromlen);
```

  - It returns the number of characters received or **-1** in case of failure

```
char buff[1024];
recvfrom(sock, buff, sizeof(buff)-1, 0, (struct sockaddr *) &servAddr,
         sizeof(servAddr));
```

1. Sockets

Functions

- To attach a socket directly to a port and an address, use the **bind** function

- Function signature:

```
int bind(int fd, struct sockaddr *addr, socklen_t addrlen);
```

  - It returns **0** or **-1** in case of failure

```
bind(sock, (struct sockaddr *) &servAddr, sizeof(servAddr));
```

1. Sockets

Functions

- To connect the socket to a server address, use the **connect** function

- Function signature:

```
int connect(int fd, struct sockaddr *addr, socklen_t addrlen);
```

  – It returns **0** or **-1** in case of failure

```
connect(sock, (struct sockaddr *) &servAddr, sizeof(servAddr));
```

1. Sockets

Functions

- To mark the socket as passive mode and set the size of the connection queue, use the **listen** function

- Function signature:

```c
int listen(int fd, int backlog);
```

  - It returns **0** or **-1** in case of failure

```c
listen(sock, 5); //Max 5 clients
```

1. Sockets

Functions

- To accept a new connection, use the **accept** function

- Function signature:

```
int accept(int fd, struct sockaddr *addr, socklen_t *addrlen);
```

  – It returns the socket descriptor or **-1** in case of failure

```
socklen_t cliAddrSize = sizeof(cliAddr); //Can be an int
accept(sock, (struct sockaddr *) &cliAddr, &cliAddrSize);
```

1. Sockets

Definitions

- Define a **sockaddr_in** to create an IPv4 server:

```c
struct sockaddr_in servAddr;

//Use TCP/IPv4
servAddr.sin_family = AF_INET;

//Use 1337 port
servAddr.sin_port = htons(1337);

//Use all IP addresses
servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

1. Sockets

Definitions

- Define a **sockaddr_in6** to create an IPv6 server:

```
struct sockaddr_in6 servAddr;

//Use TCP/IPv6
servAddr.sin6_family = AF_INET6;

//Use 1337 port
servAddr.sin6_port = htons(1337);

//Use all IP addresses
servAddr.sin6_addr = in6addr_any;
```

1. Sockets

Definitions

- Define a **sockaddr_in** to create an IPv4 client:

```c
struct sockaddr_in cliAddr;

//Use TCP/IPv4
cliAddr.sin_family = AF_INET;

//Use 1337 port
cliAddr.sin_port = htons(1337);

//To connect 127.0.0.1
cliAddr.sin_addr = inet_addr("127.0.0.1");
```

1. Sockets

Definitions

- Define a **sockaddr_in6** to create an IPv6 client:

```c
struct sockaddr_in6 cliAddr;

//Use TCP/IPv6
cliAddr.sin6_family = AF_INET6;

//Use 1337 port
cliAddr.sin6_port = htons(1337);

//To connect 2001:720:1500:1::a100
cliAddr.sin6_addr = inet_addr("2001:720:1500:1::a100");
```

1. Sockets

Definitions

- Retrieve an address using the hostname:

```
struct sockaddr_in cliAddr;
struct hostent *server;
server = gethostbyname("supinfo.com");
cliAddr.sin_addr = (struct in_addr *) server->h_addr;
```

1. Sockets

Definitions

- Convert IPv4 or IPv6 address representation:

```
//Presentation to representation
in_addr_t inet_addr(const char *cp);

//Representation to presentation
char *inet_ntoa(struct in_addr in);

//Presentation to representation Linux
int inet_pton(int type, const char *src, void *addr);

//Representation to presentation Linux
const *inet_ntop(int type, const void *, char *addr, socklen_t size);
```

1. Sockets

Definitions

```c
//IPv4
inet_addr("63.161.169.137");
//IPv6
inet_addr("2001:720:1500:1::a100");
//IPv4
inet_aton("63.161.169.137", &servAddr.sin_addr.s_addr);
//IPv6
inet_pton(AF_INET6, "2001:720:1500:1::a100", &servAddr.sin6_addr);
//IPv4
inet_ntop(AF_INET, &servAddr.sin_addr.s_addr, strServAddr,
          sizeof(strServAddr));
```

1. Sockets

Windows specificities

- Start by loading the DLL:

```
WSADATA wsa;
WSAStartup(MAKEWORD(2, 2), &wsa);
```

- Finish by unloading the DLL:

```
WSACleanup();
```

1. Sockets

Windows specificities

- Many structures are redefined:
  - **sockaddr_in** becomes **SOCKADDR_IN**
  - **sockaddr** becomes **SOCKADDR**
  - **in_addr** becomes **IN_ADDR**

- Socket descriptor is not an **int** but a **SOCKET**
  - Define for UNIX:

```
typedef int SOCKET;
```

1. Sockets

Windows specificities

- To close SOCKET, use:

```
closesocket(SOCKET fd);
```

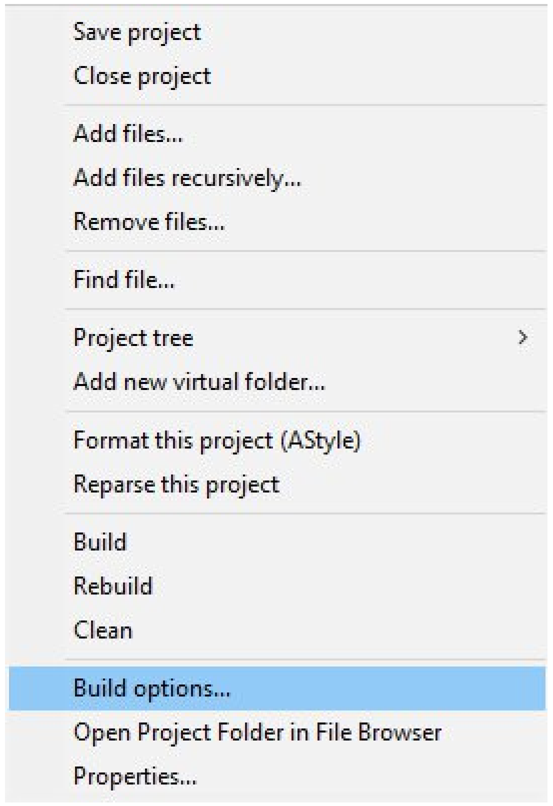
  - Define for UNIX:

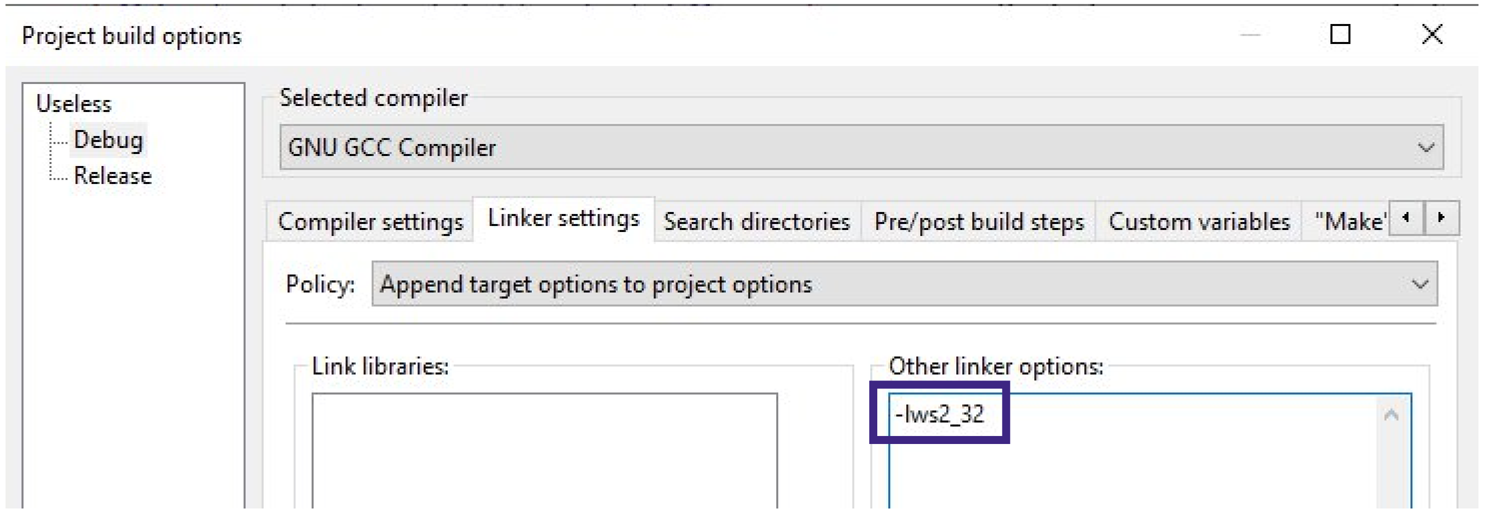
```
#define closesocket(s) close(s)
```

1. Sockets

Windows specificities

- Add the “**-lws2_32**” linker option in your build settings:





Selected compiler: GNU GCC Compiler

Policy: Append target options to project options

Other linker options: -lws2_32

1. Sockets

To go further

- To transfer a quantity of data of unknown size, it must be possible to transfer a buffer several times:
  - JSON
  - Files
  - Conversations
  - *etc.*

1. Sockets

To go further

- To check the state change of a socket, use the **select** function

- Function signature:

```
int select(int fd, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

  - To make a non-blocking server/client
  - To manage many clients
  - It returns the number of socket descriptors that have changed, **0** in case of timeout, or **-1** in case of no change
  - It removes from **fd_set**, sockets that have not been changed

```
int set = select(sock+1, *fs, NULL, NULL, NULL);
```

1. Sockets

To go further

- To monitor a set of socket descriptors, use **fd_set**
  - Add a socket descriptor to a set:

```
FD_SET(int fd, fd_set *fs);
```

  - Remove a socket descriptor from a set:

```
FD_CLR(int fd, fd_set *fs);
```

  - Clear a set:

```
FD_ZERO(fd_set *fs);
```

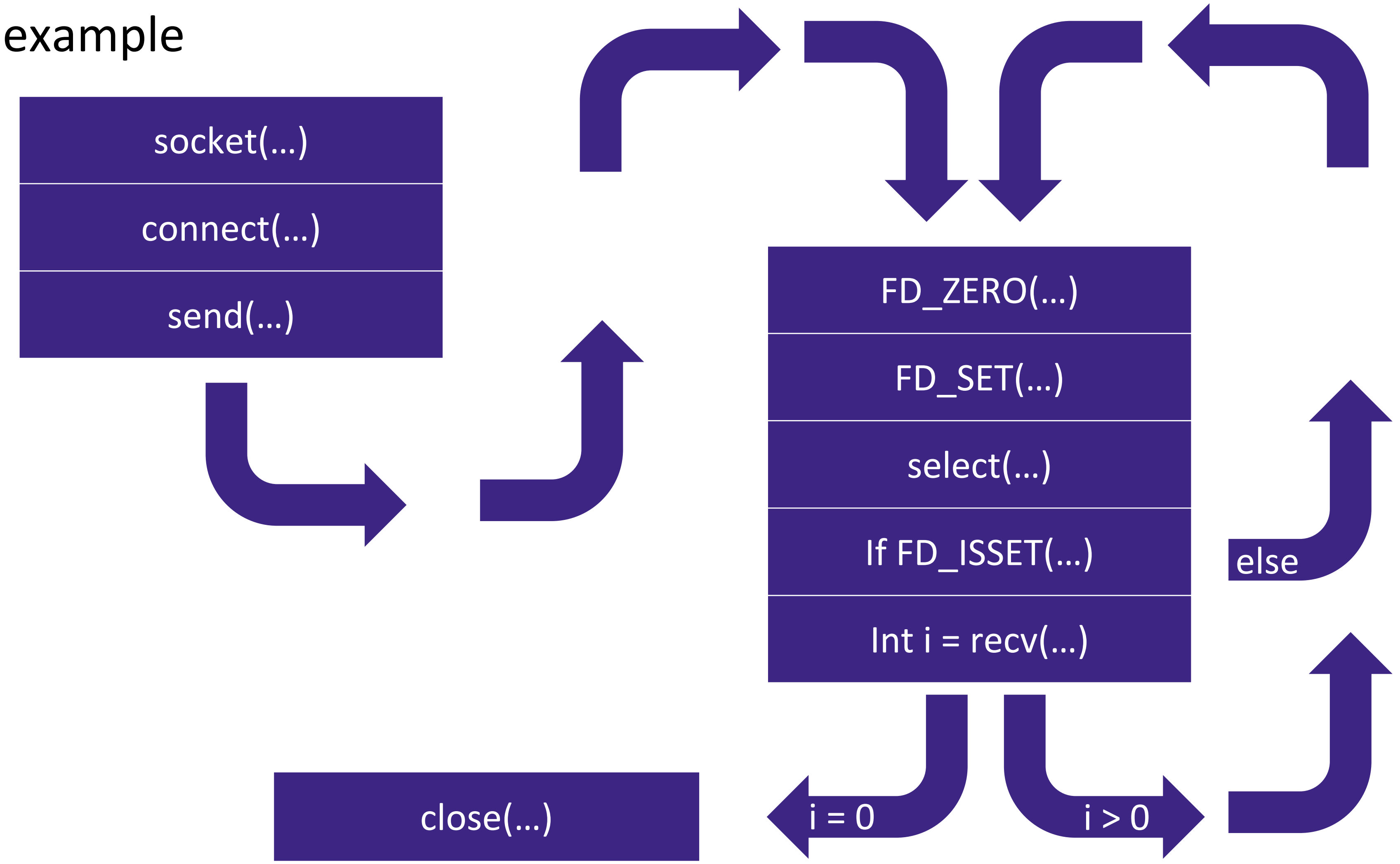
  - Check if the socket descriptor is in the set after **select**:

```
FD_ISSET(int fd, fd_set *fs);
```

1. Sockets

To go further

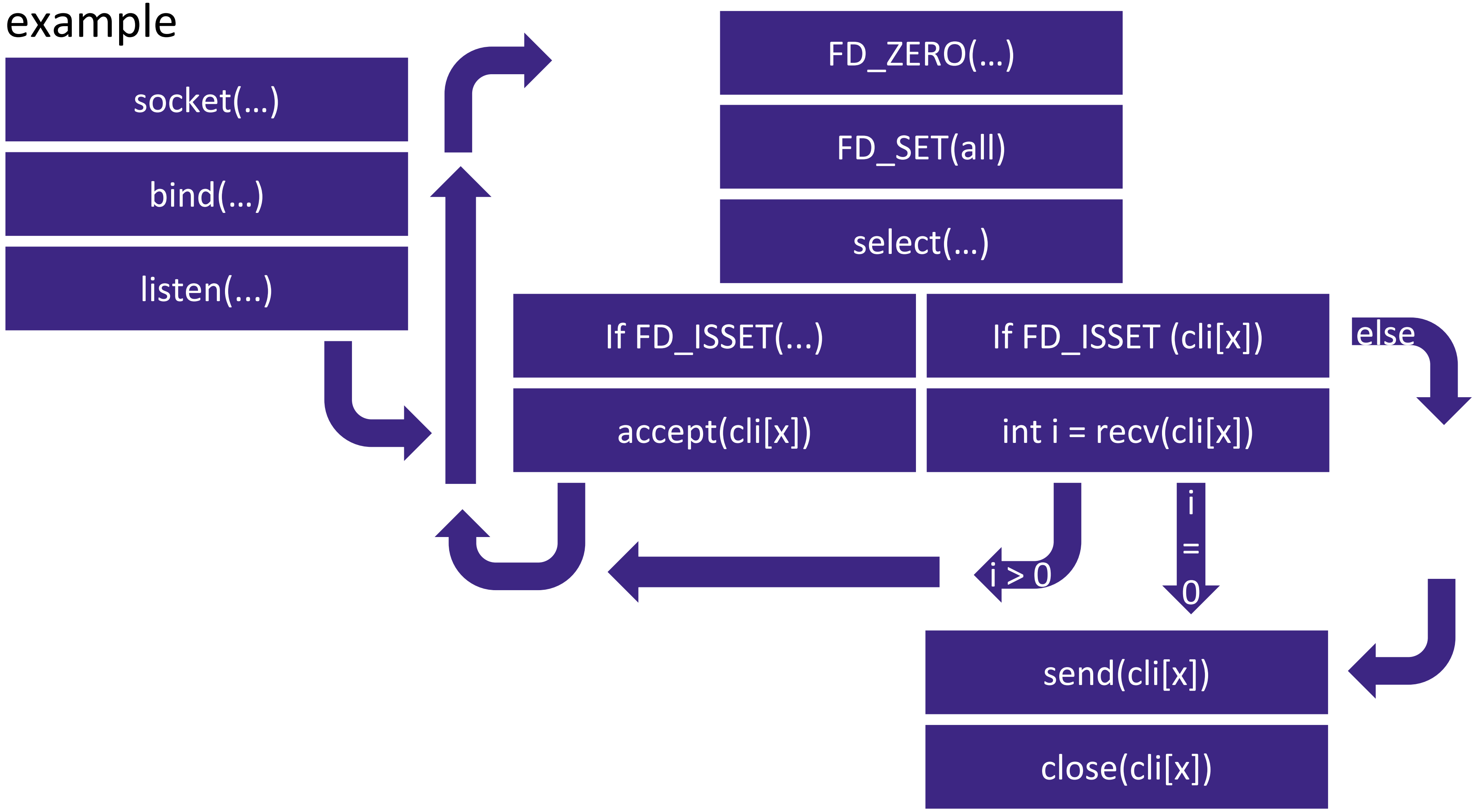
- HTTP client example



1. Sockets

To go further

- HTTP server example



1. Sockets

Exercise

- Ask the user for a domain name
- Use only port 80 and GET requests
- Save the result of the query to a file

1. Sockets

Questions

2. Threads

2. Threads

Overview

- Each application running on a computer is associated with a process representing its activity
- This process is associated with a set of custom resources such as memory space, CPU, *etc*.
- These resources will be dedicated to the execution of the program instructions associated with the application
- Processes are expensive to launch (calculating memory space, setting local variables, *etc*.)

2. Threads

Overview

- Multitasking is the ability to run several programs at the same time without conflict
- A multitasking program can run several parts of its code at the same time
- Each part of the code will be associated with a sub-process to allow parallel execution

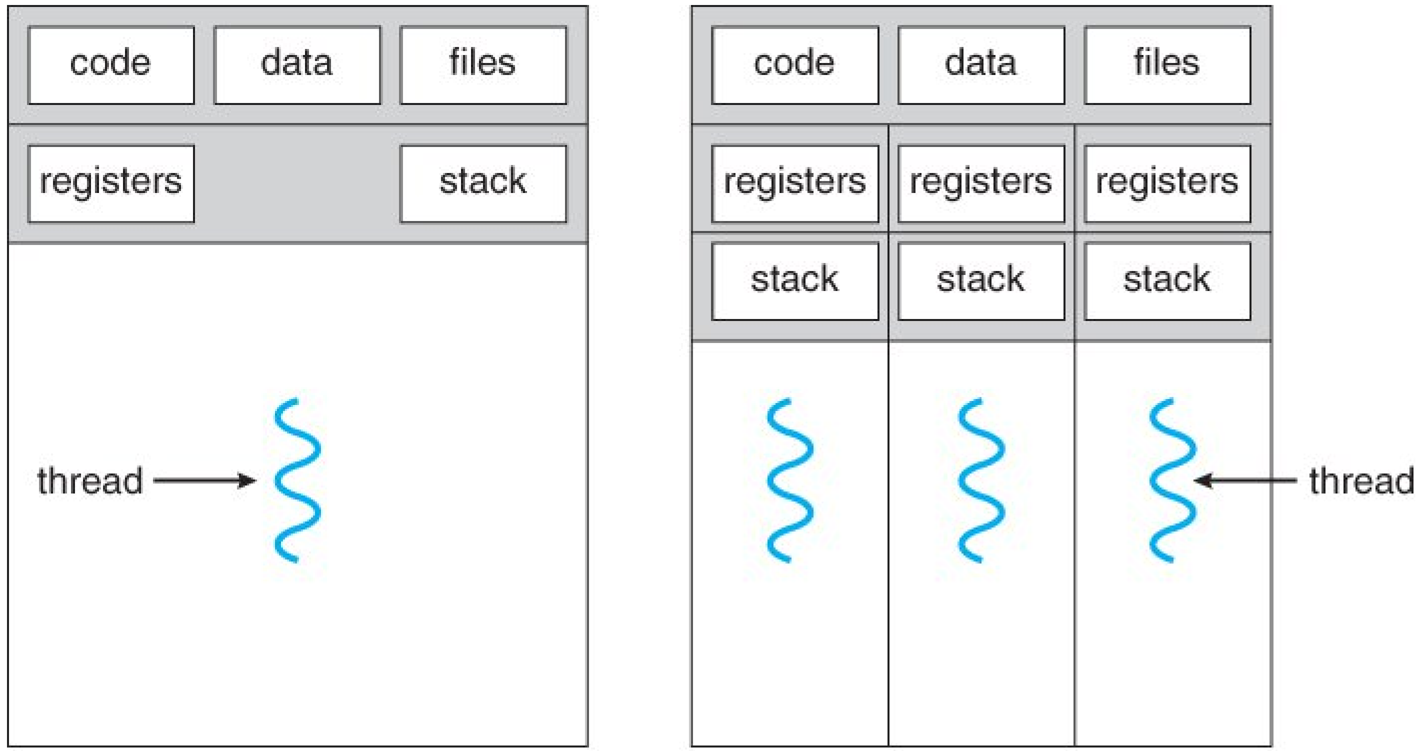
2. Threads

Overview

- A thread is a subset of a process, sharing its memory space and variables
- The costs associated with its launch are therefore reduced, so it is faster
- In addition, each thread has its own units associated with it: such as its stack (to manage the instructions to be executed by the thread), the signal mask (the signals that the thread must respond to), its execution priority (in the queue), private information, *etc*.
- Threads can be **executed in parallel** by a multitasking system, however sharing memory and process variables leads to several problems when there is shared access to a resource; we must protect the access to this resource as soon as a thread is writing

2. Threads

Overview



single-threaded process

multithreaded process

2. Threads

#include <pthread.h>
Linker option: **-lpthread**

Linux

- To create a new thread, use the **pthread_create** function

- Function signature:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

  – It takes as parameter a pointer to a thread ID, a pointer to a thread attributes structure, a pointer to a subroutine, and a pointer to the argument of the function (or to the structure in case of multiple arguments)

2. Threads

Linux

- To stop the current thread, use the **pthread_exit** function

- Function signature:

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

  - It takes as parameter a pointer to the return status of the terminated thread

2. Threads

Linux

- To wait for the termination of a thread, use the **pthread_join** function

- Function signature:

```
int pthread_join(pthread_t th, void **thread_return);
```

  - It takes as parameter a thread ID, and a pointer to the thread exit state

2. Threads

Linux

- To detach a thread (do not require a join and release resources automatically after termination) , use the **pthread_detach** function

- Function signature:

```
int pthread_detach(pthread_t thread);
```

  – It takes as parameter a thread ID

2. Threads

Linux

```
#include <stdio.h>
#include <unistd.h>  //sleep()
#include <pthread.h>

void *func(void *vargp)
{
    sleep(1);
    printf("Hi!\n");
    return NULL;
}

int main()
{
    pthread_t threadId;
    printf("Start!\n");
    pthread_create(&threadId, NULL, func, NULL);
    pthread_join(threadId, NULL);
    printf("Stop!\n");
    return 0;
}
```

2. Threads

Linux

- Mutexes are a **lock** system that guarantees the viability of the data manipulated by the threads
- It often happens that several threads need to read/write the same variables
- If a thread has the lock, only this one can read and write on the variables being in the protected portion of code (**critical section**)
- When the thread has finished, it releases the lock, and another thread can pick it up

2. Threads

Linux

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

// ...

pthread_mutex_lock(&mutex1);
//Critical zone
pthread_mutex_unlock(&mutex1);
```

2. Threads

Linux

- Semaphore is an integer variable initialized with the number of resources present in the system and used for process synchronization

- Where mutex uses a locking mechanism, semaphore uses a **signaling** mechanism

- A mutex object allows multiple process threads to access a single shared resource, but only one at a time; on the other hand, a semaphore allows multiple processes to access a finite instance of the resource until it becomes available

- In mutex, the lock can be acquired and released by the same process at a time; the value of the semaphore variable can be modified by any process that needs some resource but only one process can change the value at a time

2. Threads

#include <semaphore.h>

Linux

```
sem_t sem1;
// ...
sem_init(&sem1, 0, 1);
// ...
sem_wait(&sem1);
//Critical zone
sem_post(&sem1);
// ...
sem_destroy(&sem1);
```

2. Threads

#include <windows.h>

Windows

- To create a new thread, use the **CreateThread** function

- Function signature:

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

  – It takes as parameter a pointer to the inheritance structure, a stack size, a pointer to a subroutine, a pointer to a thread attribute, an initial state, and a thread ID

2. Threads

Windows

- To stop the current thread, use the **ExitThread** function
- Function signature:

```
void ExitThread(DWORD dwExitCode);
```

  – It takes as parameter an exit code value

2. Threads

Windows

- To wait for the termination of a thread, use the **WaitForSingleObject** function using **INFINITE** timeout

- Function signature:

```
DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);
```

  - It takes as parameter a thread handle, and a timeout interval

2. Threads

Windows

- To close a thread handle or to detach a thread, use the **CloseHandle** function

- Function signature:

```
BOOL CloseHandle(HANDLE hObject);
```

  - It takes as parameter a thread handle

2. Threads

Windows

```
#include <stdio.h>
#include <unistd.h>
#include <windows.h>

DWORD WINAPI func(LPVOID lpParameter)
{
    sleep(1);
    printf("Hi!\n");
    return (DWORD) NULL;
}

int main()
{
    printf("Start!\n");
    HANDLE hThread = CreateThread(NULL, 0, func, NULL, 0, NULL);
    WaitForSingleObject(hThread, INFINITE);
    CloseHandle(hThread);
    printf("Stop!\n");
    return 0;
}
```

2. Threads

Windows

- Mutexes:

```
HANDLE mutex1 = CreateMutex(NULL, FALSE, NULL);
// ...
WaitForSingleObject(mutex1, INFINITE);
//Critical zone
ReleaseMutex(mutex1);
// ...
CloseHandle(mutex1);
```

2. Threads

Windows

- Semaphores:

```
HANDLE sem1 = CreateSemaphore(NULL, 10, 10, NULL);
// ...
WaitForSingleObject(sem1, 0L);
//Critical zone
ReleaseSemaphore(sem1, 1, NULL);
// ...
CloseHandle(sem1);
```

2. Threads

Exercise

- Start 10 threads simultaneously
- Threads will have a certain duration in order to finish one every 5 seconds
- The main thread will display the time as it goes along and then an end message

2. Threads

Questions

Advanced Concepts

Thank you for your attention