Network Programming(III)

Lenuta Alboaie
adria@info.uaic.ro

Content

- I/O Primitives - discussions
- UDP Concurrent Server
- TCP or UDP – aspects
- Instruments
- Design and implementation alternatives for TCP/IP client/server paradigm

I/O Primitives

- Reading Data
  - read() / recv() / **readv()** / recvfrom()/ **recvmsg()**
- Sending Data
  - write() / send() / **writev()** / sendto()/ **sendmsg()**

I/O Primitives

```
#include <sys/uio.h>
ssize_t readv (int filedes, const struct iovec *iov, int iovcnt);
ssize_t writev (int filedes, const struct iovec *iov, int iovcnt);
    struct iovec
    {
        void *iov_base; /* buffer start adress */
        size_t iov_len;   /* buffer size */
    };
```

- Wider than read()/write(), it provides the ability to work with data in non-contiguous memory areas
- Both calls return, in normal execution, the transfer length in bytes

I/O Primitives

```
#include <sys/socket.h>
ssize_t recvmsg (int sockfd, struct msghdr *msg, int flags);
ssize_t sendmsg (int sockfd, struct msghdr *msg, int flags);
```

Both functions have options included in *msghrd* structure

The most general I/O functions; read/readv/recv/recvfrom calls can be replaced by *recvmsg*

Both calls return, in normal execution, the transfer length in bytes; -1 in error case

I/O Primitives

Comparison among I/O primitives:

Function	Any descriptor	Just for Socket descriptor	One read/write buffer	Scatter/ gather read/write	Optional flags	*Peer Address*
read, write	●		●			
readv, writev	●			●		
recv, send		●	●		●	
recvfrom, sendto		●	●		●	●
recvmsg, sendmsg		●		●	●	●

UDP Server| Discussions

- Most UDP servers are iterative
  - A UDP Server reads the client`s request, processes the request and sends the response
  - What happened in the situations when multiple datagrams should be exchanged with the client?
- **Concurrent UDP Server**
  - if processing the answer takes time, the server can create (*fork*()) a child process that will resolve the client request

UDP Server| Discussions

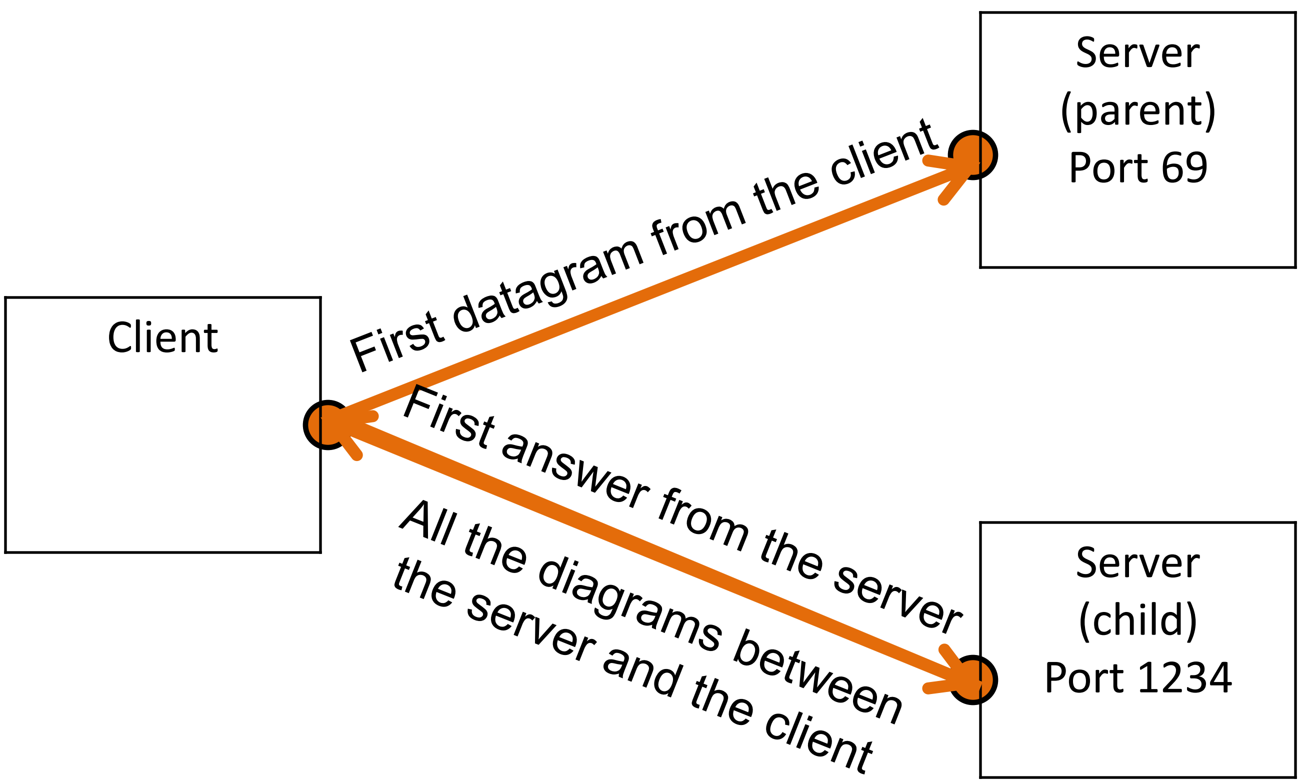
UDP Concurrent Server:

- UDP Server that exchanges multiple datagrams with clients
  - Problem: Just a port is known by the client as "*well-known*"
  - Solution: the server creates a new socket for each client, it attaches it to an "ephemeral" port and uses this socket for all answers
    - Mandatory: the client must take the port number from the server`s first response; subsequently, the next requests will use that port

    Example: TFTP - Trivial File Transfer Protocol

UDP Concurrent Server

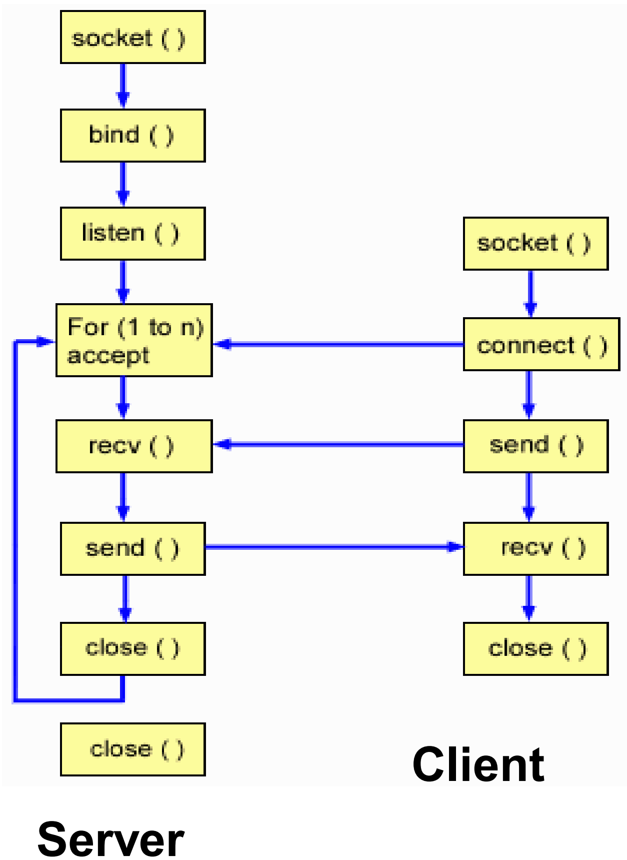
– TFTP use UDP and 69 port



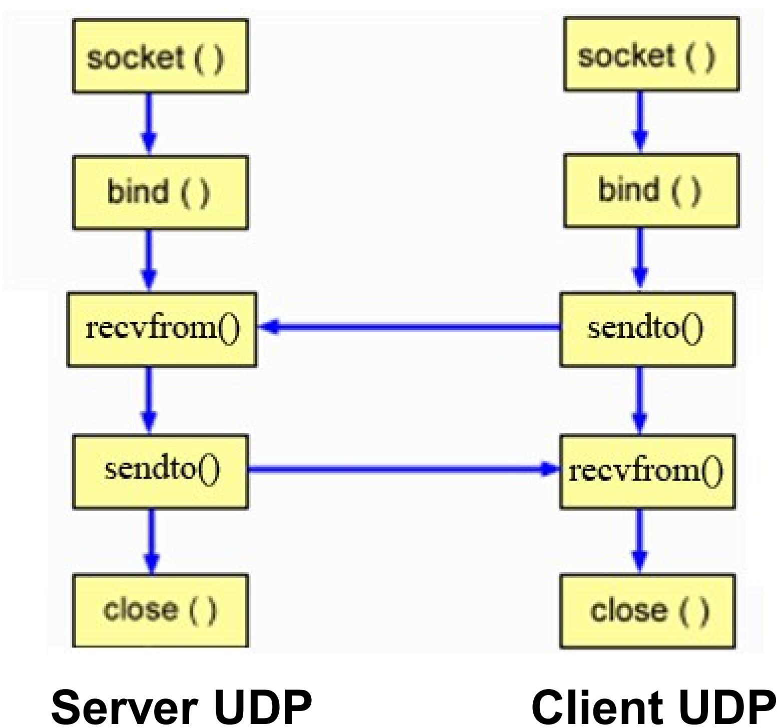
Steps: socket(), bind() to *well-known 69, recvfrom(), fork(), alt recvfrom(),…*

Steps: socket(), bind() to *1234 port,* the client`s request is processed, exchanges of the datagrams between client-server using the new port

TCP or UDP - discussions



TCP server/client Model



UDP server/client Model

TCP or UDP – Discussions

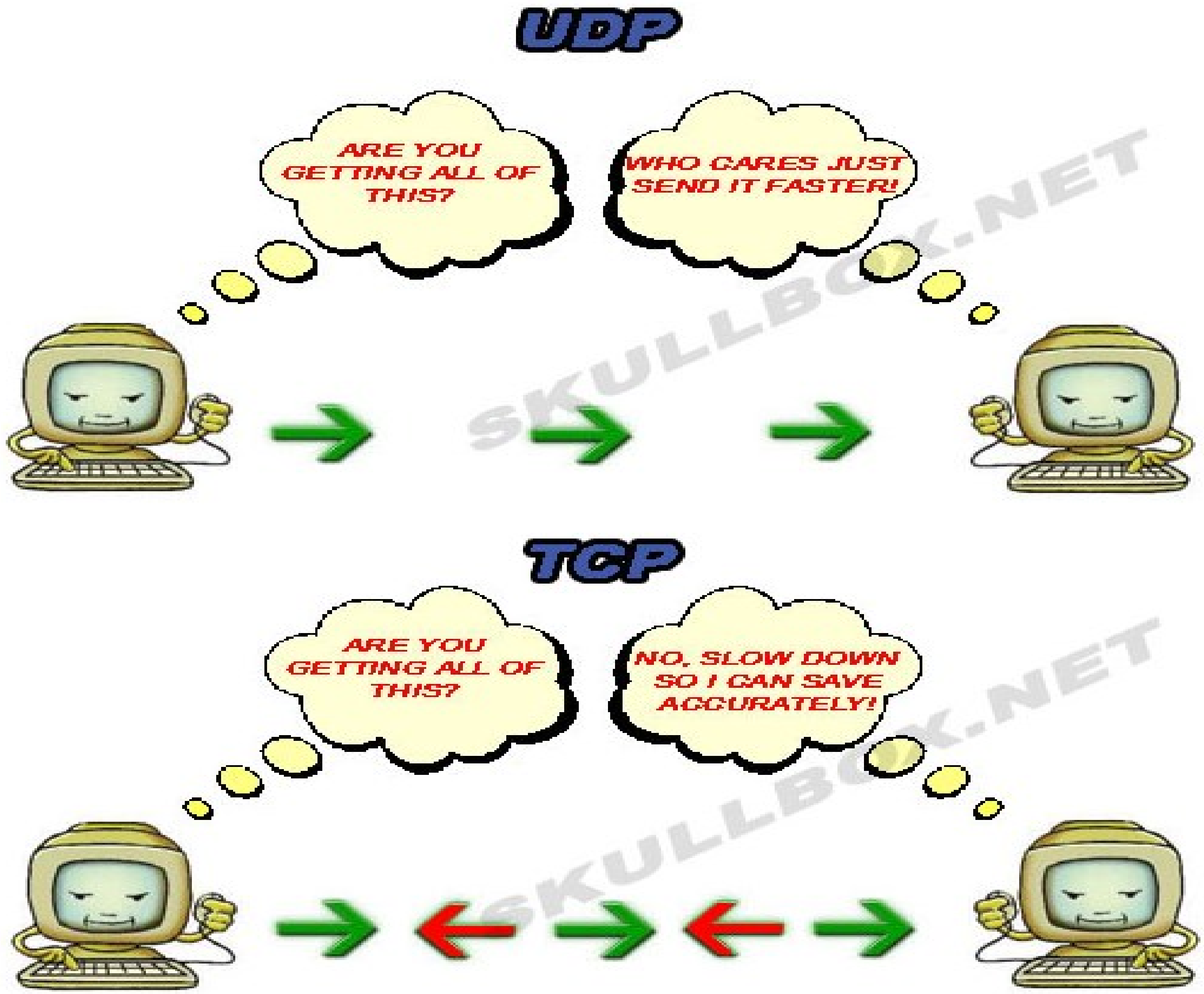
Aspects regarding UDP uses:

- UDP supports broadcasting and multicasting
- UDP does not require a mechanism to establish a connection
- The minimum time for a UDP transaction (request-response) is: RRT(*Round Trip* Time) + SPT (*server processing time*)

Aspects regarding TCP uses:

- TCP supports point-to-point
- TCP is connection-oriented
- Offers safety and sorted data transmission;
- It provides mechanisms for flow control and congestion control
- The minimum time for a TCP transaction is 2 *RRT + SPT

TCP or UDP – Discussions



[http://www.skullbox.net]

TCP or UDP – Discussions

UDP , TCP uses – recommendations

- UDP should be used for multicast or broadcast applications
- The error control must (eventually) be added to the server or the client`s level
- UDP can be used for simple request-response operations; errors should be addressed at application level

Examples: streaming media, teleconferencing, DNS

TCP or UDP – Discussions

UDP , TCP uses – recommendations

- TCP *should be used for bulk data transfer* (e.g. file transfer)
  - Is it still possible to employ UDP? → We reinvent TCP at the application level!

Examples: HTTP (Web), FTP (File Transfer Protocol), Telnet, SMTP

Instruments

- Multiple UNIX systems offer "***system call tracing***" facility

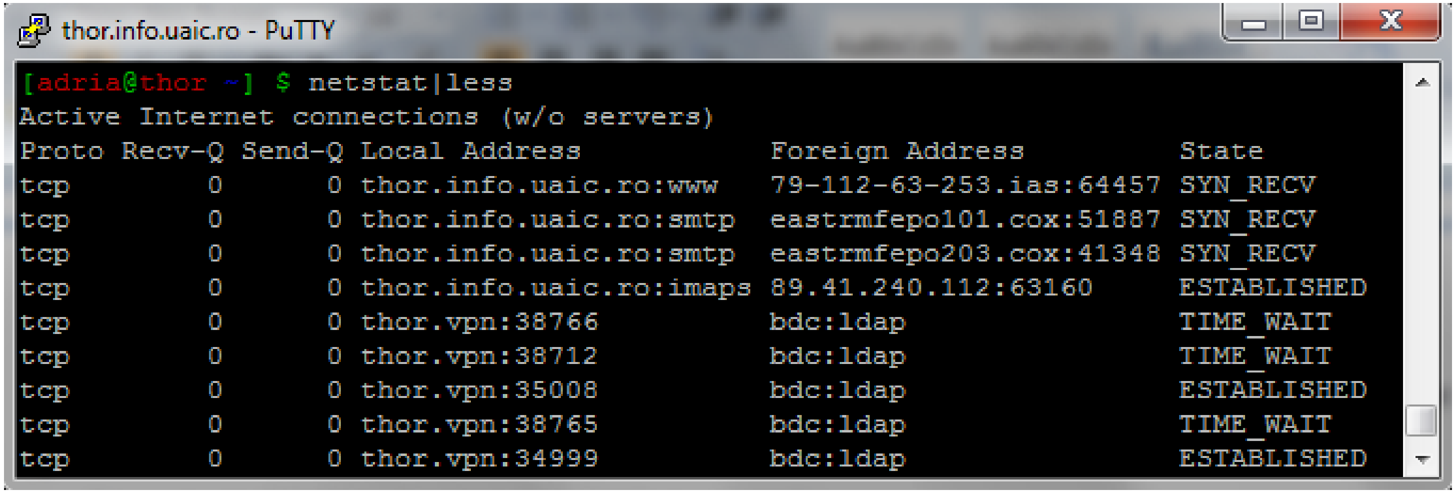
```
adria@ubu: ~/S6
   I A  test.c (Modifi  Row 8     Col 28    8:15
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{

  char *sir=NULL;
  printf("program de debugs: ");
  //sir = (char *) malloc(100*sizeof(char));
  fgets(sir, 1024, stdin);
  printf(sir);
  return 1;
}
```

strace

```
write(1, "program de debug\n"..., 17program de debug
)    = 17
fstat64(0, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 3), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7
fdb000
read(0, 0xb7fdb000, 1024)                  = ? ERESTARTSYS (To be restarted)
--- SIGWINCH (Window changed) @ 0 (0) ---
read(0, Test in saptamina 6
"Test in saptamina 6\n"..., 1024) = 20
--- SIGSEGV (Segmentation fault) @ 0 (0) ---
+++ killed by SIGSEGV +++
```

Instruments

- Small test programs
- Instruments:
  - **tcpdump** – most versions of Unix
    - It provides information on packets from network
    - http://www.tcpdump.org/
  - **snoop** – Solaris 2.x
  - **lsof**
    - Identify what processes have an open socket to a specified IP address or port
  - **netstat**



Instruments

- Instruments:
  - **tcptrack**

```
Client                 Server              State        Idle A Speed
172.23.195.11:48328    67.39.222.44:22     ESTABLISHED  0s     38 KB/s
172.23.195.11:48646    196.30.80.10:80     ESTABLISHED  1s     30 KB/s
172.23.195.11:48661    64.37.246.17:80     ESTABLISHED  0s     387 B/s
172.23.195.11:48620    216.239.39.99:80    RESET        2s     0 B/s
128.230.225.95:3531    172.23.195.10:1220  ESTABLISHED  5s     0 B/s
172.23.195.11:48621    216.239.39.99:80    ESTABLISHED  7s     0 B/s
172.23.195.11:48606    64.233.167.99:80    ESTABLISHED  10s    0 B/s
172.23.195.11:48014    67.39.222.44:22     ESTABLISHED  16s    0 B/s
172.23.195.11:47988    67.39.222.44:22     ESTABLISHED  18s    0 B/s

TOTAL                                                      69 KB/s
Connections 1-9 of 9                             Unpaused  Sorted
```

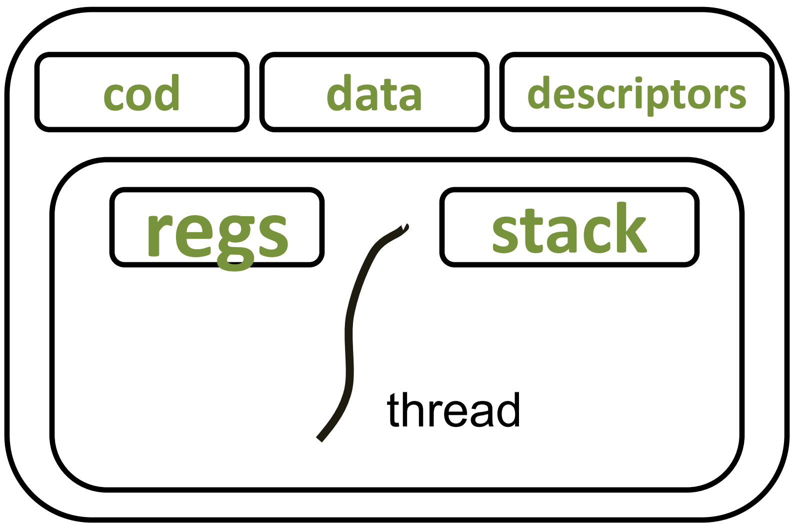
Design and implementation alternatives for TCP/IP client/server paradigm

Threads| Need

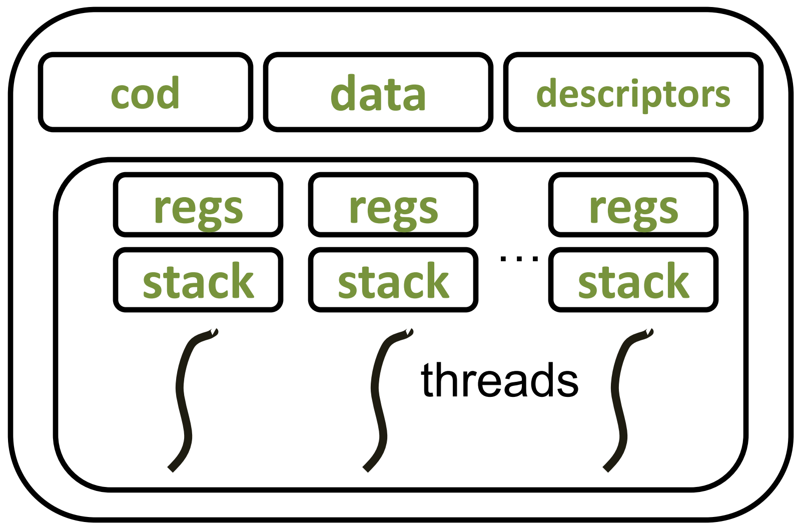
- fork() can be a costly mechanism
  – Current implementations use the *copy-on-write* mechanism
- IPC (Inter-Process Communication) requires sending information between parent and child after fork()

Threads| Characteristics

- *Threads* – are also called *lightweight processes (LWP)*
- They can be seen as a running program without their own address space



Processes with a thread



Multi-threaded processes

Processes, Threads| Comparisons

- Example: The costs associated with creating and managing processes (50,000) is higher than threads (50,000)

Platform	fork()		pthread_create()	
	user	sys	user	sys
AMD 2.4 GHz Opteron (8cpus/node)	2.2	15.7	0.3	1.3
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	30.7	27.6	0.6	1.1
IBM 1.5 GHz POWER4 (8cpus/node)	48.6	47.2	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	1.5	20.8	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	1.1	22.2	1.2	0.6

[https://computing.llnl.gov/tutorials/pthreads/]

Threads| Implementation

- **Pthreads** (POSIX Threads) standard defines an API for creating and manipulating threads
  – FreeBSD
  – NetBSD
  – GNU/Linux
  – Mac OS X
  – Solaris
- Pthread API for Windows – pthreads-w32

Threads| Basic Primitives

pthread_t (-> often an unsigned int) (*thread* identifier)

The structure specifies the attributes of the new created thread(e.g. stack size, priority, NULL=default behavior)

Reference to the function to be executed by the *thread*

Argument to *thread* that is passed to the function

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

Returns: 0 in case of success

an Exxx positive value in case of error

Threads| Basic Primitives

Thread identifier

```
#include <pthread.h>
int pthread_join(
        pthread_t *tid,
        void **status );
```

… will store the return value of the *thread* (a pointer to an object)

- Enables waiting for a thread to finish

Returns: 0 in case of success

an Exxx positive value in case of error

Threads| Basic Primitives

```
#include <pthread.h>
pthread_t pthread_self();
```

Thread identifier

Returns: The thread ID that called the primitive

Threads| Basic Primitives

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

Thread identifier

Threads can be:

- *joinable*: when the thread ends, the ID and return code are kept until pthread_join() is called <- default behavior
- *detached:* when the thread ends, all resources are released

Returns: 0 in case of success

an Exxx positive value in case of error

Example: pthread_detach(pthread_self());

Threads| Basic Primitives

#include <pthread.h>

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

\- Finishing a *thread*

Threads may end if:

- The function that is executed by the *thread* ends (Note: Return value is void * and will represent the output code of the thread)
- The *main* function of the process returns or any of the threads called *exit(),* the process ends

Threads| Example

Example: TCP concurrent server that uses threads instead of fork()

Note. Compiling: **gcc –lpthread server.c or**

gcc server.c –lpthread

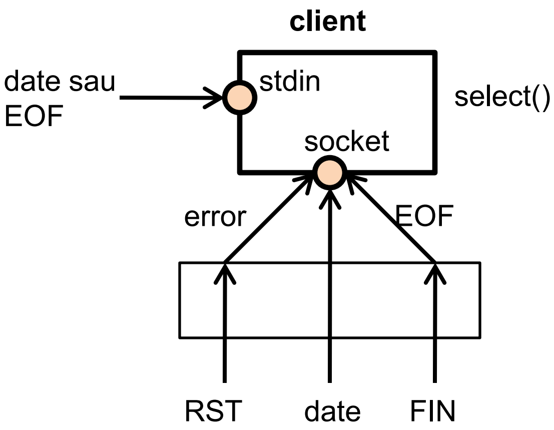
DEMO

Design and implementation alternatives for TCP/IP client/server paradigm

- **Client TCP – the usual model**
  - Aspects:
    - As long as it is blocked by waiting for user data, it does not notice network events(e.g. *peer close()*)
    - Works in "*stop and wait*" mode
    - "*batch processing*"

Design and implementation alternatives for TCP/IP client/server paradigm

- **TCP Client – using select()**
  - The client is notified by network events while waiting for user input



If the *peer* sends data, *read*() returns a value >0;

If the TCP *peer* send FIN, the *socket* becomes "readable" and *read*() returns 0;

If the *peer* send RST (the *peer* has fallen or rebooted), the *socket* becomes readable and *read*() returns -1;

Aspects:

- The *write*() call can be blocked if the *buffer* of the sending *socket* is full

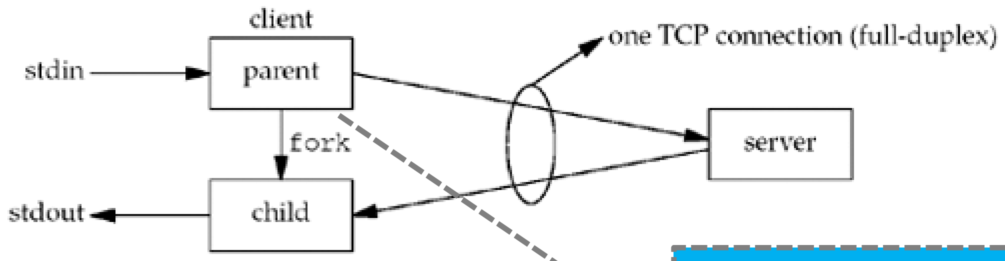
Design and implementation alternatives for TCP/IP client/server paradigm

- **TCP Client** – **using select() and non blocking I/O operations**
  - Aspects:
    - Complex implementation => when non-blocking I/O operations are required, use of fork() or threads is recommended (see next *slides*)

Design and implementation alternatives for TCP/IP client/server paradigm

TCP Client – using fork()

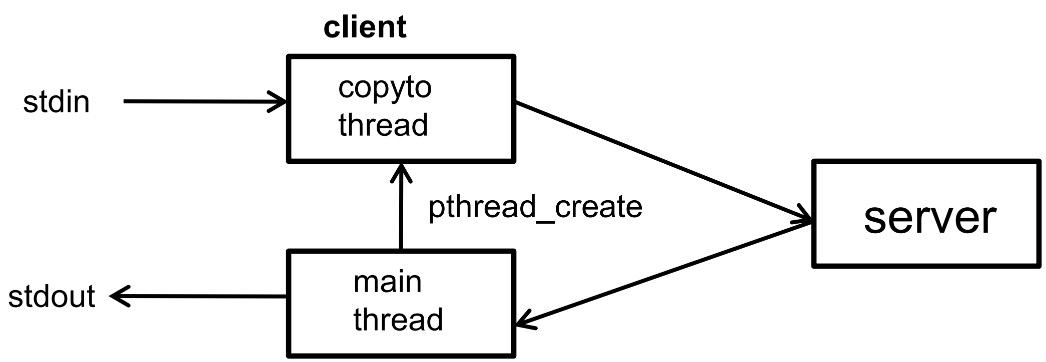
- Operating mechanism:
  - there are two processes
    - A process addresses the client-server data management
    - A process addresses server-to-client data management



The parent and child share the same socket

Design and implementation alternatives for TCP/IP client/server paradigm

- **TCP Client – using pthread()**
  - Operating mechanism:
    - there are two threads
      - A thread addresses the client-server data management
      - A thread addresses server-to-client data management



Design and implementation alternatives for TCP/IP client/server paradigm

- Time comparison of TCP clients execution; the clients have the discussed architectures

TCP client	Execution time (seconds)
Usual pattern (stop-and-wait)	...
Using select and blocking I/O operations	12.3
Using select and unblocking I/O operations	6.9
Using fork()	8.7
Using threads	8.5

- Note. The measurement was conducted using the time command **client**/server *echo* implementations

[Unix Network Programming, R. Stevens B. Fenner, A. Rudoff - 2003

Design and implementation alternatives for TCP/IP client/server paradigm

- **TCP Server – iterative**
  - The clients request are processed sequentially

  Aspects:
  - They are rarely encountered in real implementations
  - Such a server serves a client very quickly

Design and implementation alternatives for TCP/IP client/server paradigm

- **TCP Server** – a child process for each client
  - The server serves clients simultaneously
  - It is widely used in practice

– Example of a mechanism used to distribute requests: *DNS round robin*

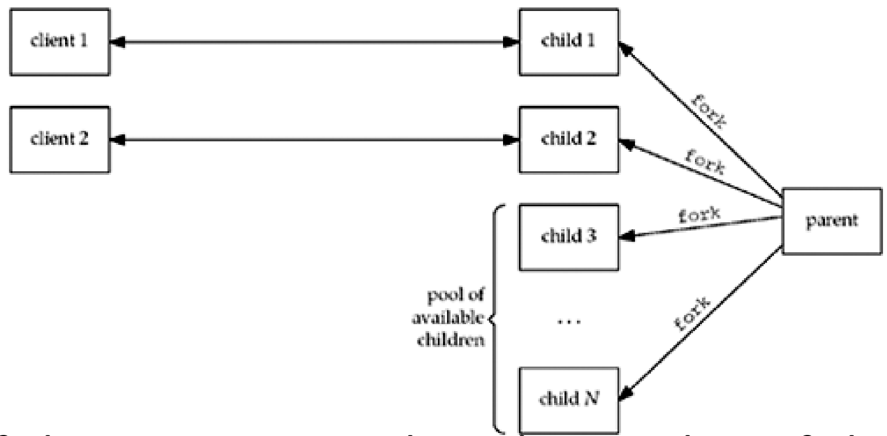
Aspects:

– Creating each child (fork ()) for each client consumes a lot of CPU time

Design and implementation alternatives for TCP/IP client/server paradigm

- **TCP Server** – ***preforking;*** **without protection on accept ()**

The server creates a number of child processes when it is turned on, and then they are ready to serve their clients



Aspects

- If the number of clients is greater than the number of child processes available, the client will experience a "degradation" of the response in relation to the time factor
- This implementations work in systems that have *accept*() as system primitive

Design and implementation alternatives for TCP/IP client/server paradigm

- **TCP Server** – ***preforking;* with lock for *accept*() protection**

  Implementation:

  – The server creates a number of child processes when it is turned on, and then they are ready to serve their clients

  – A blocking mechanism (e.g., *fcntl* ()) of the *accept* () primitive call is used, and only one process at a time can call *accept* (); the remaining processes will be blocked until they can get access

- Example: Apache (http://www.apache.org) uses the *preforking* technique

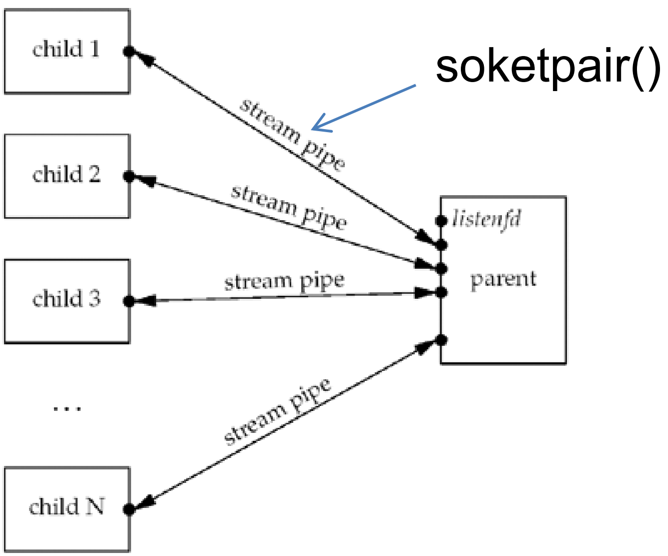
Design and implementation alternatives for TCP/IP client/server paradigm

- **TCP Server** – ***preforking;* with "forwarding" the connected socket**

Implementation:

- The server creates a number of child processes when it is started, and then they are ready to serve customers

- The parent process call *accept*() and "forward" the connected *socket* to a child process



Aspects:

The parent process must have evidence of the actions of all children => a greater complexity of the implementation

Design and implementation alternatives for TCP/IP client/server paradigm

- **TCP Server** – **one *thread* for each client**

  Implementation:

  The main *Thread* is locked in the accept() call and whenever a client is accepted, it creates (*pthread_create*()) a *thread* that will serve it

DEMO (Slide 28)

Aspects:

This implementation is generally faster than the fastest TCP preforked server version

Design and implementation alternatives for TCP/IP client/server paradigm

- **TCP Server** – ***prethreaded;* with lock for *accept*() protection**

  Implementation:

  – The server creates a number of threads when it is turned on, and then they are ready to serve clients

  – A locking mechanism is used (e.g. *mutex lock*) for *accept*() call, and only one thread at a time will call *accept*();

  Note. *Threads* will not be locked in *accept*()

DEMO

Design and implementation alternatives for TCP/IP client/server paradigm

- TCP Server – ***prethreaded;*** **with "forwarding" the connected socket**

  Implementation:

  The server creates a number of threads when it is turned on, and then they are ready to serve clients

  The parent process is the one that calls accept () and "forwards" the connected socket to an available thread

  Note. Because the threads and descriptors are within the same process, the "forwarding" of the connected socket actually means that the target thread knows the descriptor`s number

Design and implementation alternatives for TCP/IP client/server paradigm

- If the server is not highly requested, the traditional concurent server variant (a fork () per client) is usable
- Creating a pool of children or pool of threads is more effective in terms of time factor; care must be taken to monitor the number of free processes, increase or decrease of this number a.i. customers are served dynamically
- The mechanism by which child processes or threads can call accept() is simpler and faster than the one in which the main thread calls accept() and then “forwards" the descriptor to the thread or child process
- Thread-based applications are generally faster than those employing processes, but the choice depends on the SO or the specificity of the task

Summary

- I/O Primitives - discussions
- UDP Concurrent Server
- TCP or UDP – aspects
- Instruments
- Design and implementation alternatives for TCP/IP client/server paradigm

Questions?