# Εργαστήριο 8 – I/O Multiplexing

Στο εργαστήριο θα μελετηθούν:

- *Server High Level View*
- *I/O Multiplexing*
- *Solutions for Concurrency*
  - *nonblocking I/O*
  - *Use alarm and signal handler to interrupt slow system calls.*
  - *Use functions that support checking of multiple input sources at the same time.*

Διδάσκων: Γιώργος Χατζηπολλάς

# Server - High Level View

Create a socket

Bind the socket

Listen for connections

Accept new client connections

Read/write to client connections

Close connection

# Servicing Two Ports

```
int s1;                                  /* socket descriptor 1 */
int s2;                                  /* socket descriptor 2 */

/* 1) create socket s1 */
/* 2) create socket s2 */
/* 3) bind s1 to port 2000 */
/* 4) bind s2 to port 3000 */

while(1) {
    recvfrom(s1, buf, sizeof(buf), ...);
    /* process buf */

    recvfrom(s2, buf, sizeof(buf), ...);
    /* process buf */
}
```

**What problems does this code have?**

# What's the problem?

- Some applications need to process input from more than one peripheral device
- Input is normally obtained using a 'read()' operation on an open file-descriptor
- But 'read()' (also accept(), connect(), recv(), …) may cause a process to sleep if no data is ready from a particular device
- While asleep the process is unable to read new data that is ready on any other device

# Solutions for Concurrency

- Use nonblocking I/O
  - use fcntl() to set O_NONBLOCK
- Use alarm and signal handler to interrupt slow system calls.
- Use functions that support checking of multiple input sources at the same time.
  - use select()
- Use multiple processes/threads.
  - This will not be covered today

# Non blocking I/O

- UNIX applications can open device-files in a 'non-blocking' mode
- Instead of sleeping when no data is ready, a process can proceed immediately to try reading from one of the other device-files
- use fcntl() to set O_NONBLOCK.
- fcntl(int fd, int cmd, long arg);

```
#include <unistd.h>
#include <fcntl.h>
sockfd = socket(PF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
```

socket command (set flag) flag to be set

- If you try to read from a nonblocking socket and there's no data there, it's not allowed to block–it will return -1 and errno will be set to EWOULDBLOCK.

# The problem with nonblocking I/O

- Using blocking I/O allows the Operating System to put the process to sleep when nothing is happening (no input). Once input arrives the OS will wake up the process and read() (or write() … ) will return.

- Using nonblocking I/O a program that can 'starve' other system applications (by constantly calling 'read()' for no data)

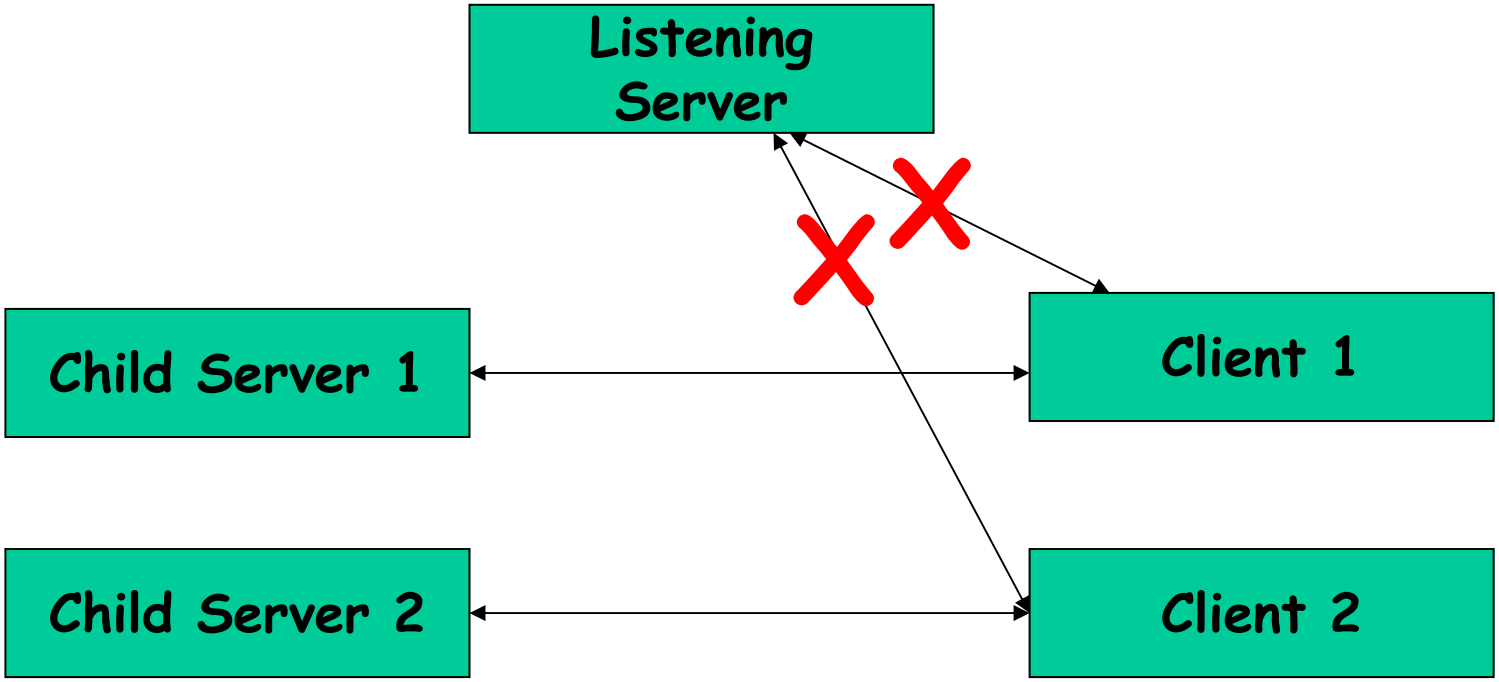
# Using alarms

```
signal(SIGALRM, sig_alrm);
alarm(MAX_TIME);
read(tcpsock,…);
...
```

**A function that you should write**

- Alarming Issue
  - How to choose the value for MAX_TIME?
  - What is the impact of time?

# Forking Concurrent Server



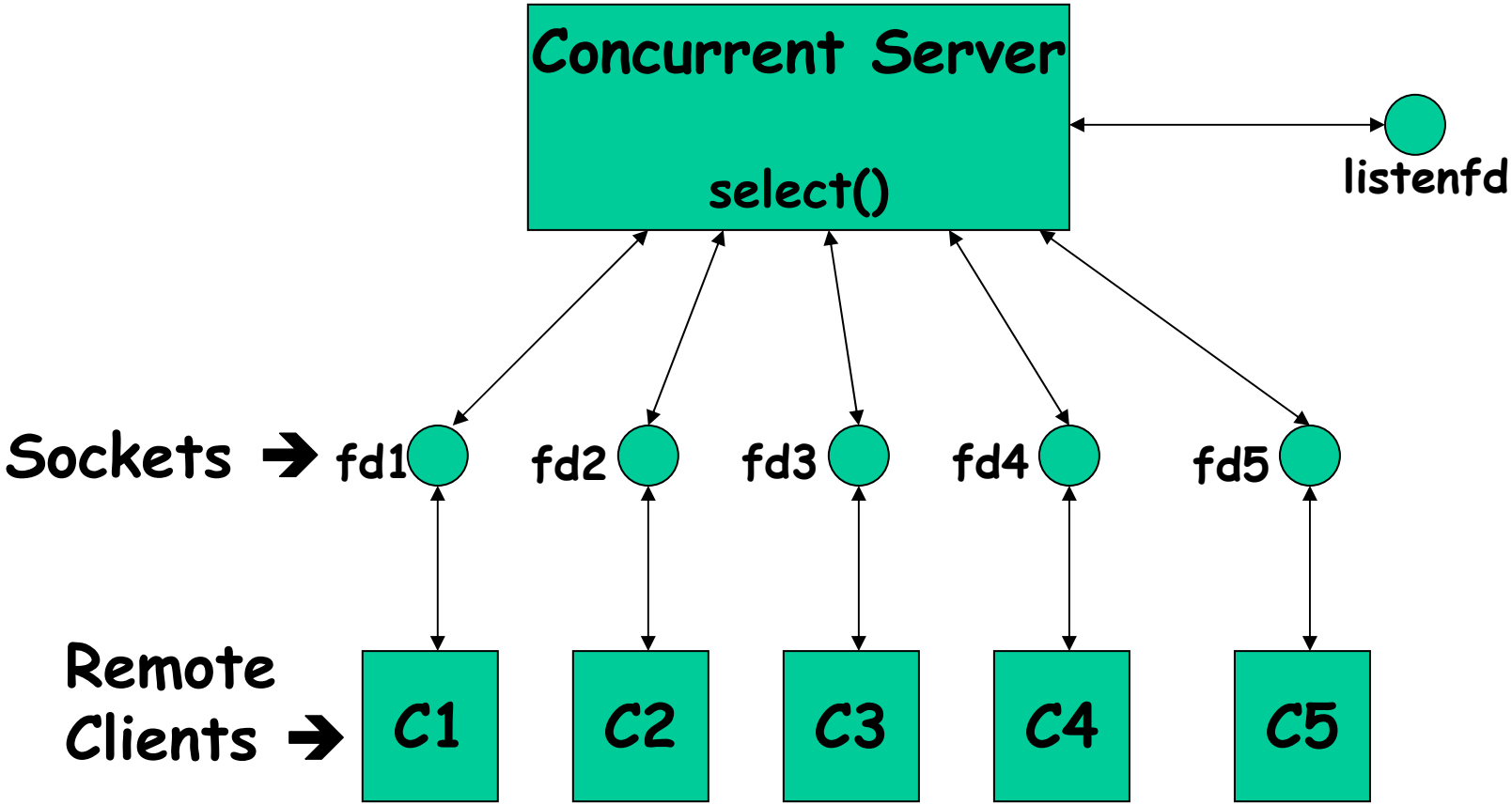
# Forking Server Example

```
listenfd = Socket( … )
Initialize server address
Bind( listenfd, … )
Listen(listenfd,…);
for ( ;; ) {
   /* wait for client connection */
   connfd = Accept(listenfd,…);
   if( (pid = Fork() ) = = 0) {
       /* Child Server */
       Close(listenfd);
       service_client(connfd);
       Close(connfd);
       exit(0);
   } else
       /* Parent */
       Close(connfd);
}
```

# Socket I/O: select()

- With select(), instead of having a process for each request, there is usually only one process that "multiplexes" all requests.
- One main advantage of using select() is that your server will only require a single process to handle all requests.
  - Your server will not need shared memory or synchronization primitives for different 'tasks' to communicate.
- One major disadvantage of using select(), is that your server cannot act like there's only one client, like with a fork()'ing solution.
  - For example, with a fork()'ing solution, after the server fork()s, the child process works with the client as if there was only one client in the universe -- the child does not have to worry about new incoming connections or the existence of other sockets. With select(), the programming isn't as transparent

# Non-Forking Concurrent Server



# Using select()

- select() works by blocking until something happens on a file descriptor (aka a socket).
  - What's 'something'?
    - Data coming in
    - Being able to write to a file descriptor
- Most select()-based servers look pretty much the same:
  - Fill up a fd_set structure with the file descriptors you want to know when data comes in on.
  - Fill up a fd_set structure with the file descriptors you want to know when you can write on.
  - Call select() and block until something happens.
  - Once select() returns, check to see if any of your file descriptors was the reason you woke up. If so, 'service' that file descriptor in whatever particular way your server needs to (i.e. read in a request for a Web page).
  - Repeat this process forever.

# Socket I/O: select()

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

FD_ZERO(fd_set *set)          // clears a file descriptor set
FD_SET(int fd, fd_set *set)   // adds fd to the set
FD_CLR(int fd, fd_set *set)   // removes fd from the set
FD_ISSET(int fd, fd_set *set) // tests to see if fd is in the set
```

- ***maxfds***: number of descriptors to be tested
  - descriptors (0, 1, ... maxfds-1) will be tested
- ***readfds***: a set of *fds* we want to check if data is available
  - returns a set of *fds* ready to read
  - if input argument is *NULL*, not interested in that condition
- ***writefds***: returns a set of *fds* ready to write
- ***exceptfds***: returns a set of *fds* with exception conditions

# Socket I/O: select()

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

struct timeval {
      long tv_sec;       /* seconds /
      long tv_usec;      /* microseconds */
}
```

- ***timeout***
  - if NULL, wait forever and return only when one of the descriptors is ready for I/O
  - otherwise, wait up to a fixed amount of time specified by *timeout*
    - if we don't want to wait at all, create a timeout structure with timer value equal to 0
- Refer to the man page for more information

# Servicing Two Ports using select

```
int s1, s2;                          /* socket descriptors */
fd_set readfds;            /* used by select() */

/* create and bind s1 and s2 */
while(1) {
   FD_ZERO(&readfds);                /* initialize the fd set */
   FD_SET(s1, &readfds);  /* add s1 to the fd set */
   FD_SET(s2, &readfds);  /* add s2 to the fd set */

   if(select(s2+1, &readfds, 0, 0, 0) < 0) {
         perror("select");
         exit(1);
   }
   if(FD_ISSET(s1, &readfds)) {
         recvfrom(s1, buf, sizeof(buf), ...);
         /* process buf */
   }
   if(FD_ISSET(s2, &readfds)) {
         recvfrom(s2, buf, sizeof(buf), ...);
         /* process buf */
   }
}
```

# select() – Example 1

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#define STDIN 0  // file descriptor for standard input
int main(void)
{
   struct timeval tv;
   fd_set readfds;

   tv.tv_sec = 2;
   tv.tv_usec = 500000;

   FD_ZERO(&readfds);
   FD_SET(STDIN, &readfds);

   // don't care about writefds and exceptfds:
   select(STDIN+1, &readfds, NULL, NULL, &tv);

   if (FD_ISSET(STDIN, &readfds))
        printf("A key was pressed!\n");
   else
        printf("Timed out.\n");
   return 0;
}
```

# select() – Example 2

```
#include …
#define PORT 9034   // port we're listening on

int main(void)
{
    fd_set master;   // master file descriptor list
    fd_set read_fds; // temp file descriptor list for select()
    struct sockaddr_in myaddr;     // server address
    struct sockaddr_in remoteaddr; // client address
    int fdmax;        // maximum file descriptor number
    int listener;     // listening socket descriptor
    int newfd;        // newly accept()ed socket descriptor
    char buf[256];    // buffer for client data
    int nbytes;
    int yes=1;        // for setsockopt() SO_REUSEADDR, below
    socklen_t addrlen;
    int i, j;

    FD_ZERO(&master);    // clear the master and temp sets
    FD_ZERO(&read_fds);

    // get the listener
    if ((listener = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }
```

# select() - Example 2

```
 // lose the pesky "address already in use" error message
 if (setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1){
     perror("setsockopt");
     exit(1);
 }
// bind
 myaddr.sin_family = AF_INET;
 myaddr.sin_addr.s_addr = INADDR_ANY;
 myaddr.sin_port = htons(PORT);

 if (bind(listener, (struct sockaddr *)&myaddr, sizeof(myaddr)) == -1){
     perror("bind");
     exit(1);
 }
 // listen
 if (listen(listener, 10) == -1) {
     perror("listen");
     exit(1);
 }

 // add the listener to the master set
 FD_SET(listener, &master);

 // keep track of the biggest file descriptor
 fdmax = listener; // so far, it's this one
```

# select() - Example 2

```
// main loop
   for(;;) {
       read_fds = master; // copy it
       if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
           perror("select");
           exit(1);
       }

       // run through the existing connections looking for data to read
       for(i = 0; i <= fdmax; i++) {
           if (FD_ISSET(i, &read_fds)) { // we got one!!
               if (i == listener) {
                   // handle new connections
                   addrlen = sizeof(remoteaddr);
  if ((newfd = accept(listener, (struct sockaddr *)&remoteaddr, &addrlen)) == -1)
                       perror("accept");
                   else {
                       FD_SET(newfd, &master); // add to master set
                       if (newfd > fdmax) {    // keep track of the maximum
                           fdmax = newfd;
                       }
                       printf("selectserver: new connection from %s on "
                           "socket %d\n", inet_ntoa(remoteaddr.sin_addr), newfd);
                   }
```

# select() - Example 2

```
 } else {
     // handle data from a client
     if ((nbytes = recv(i, buf, sizeof(buf), 0)) <= 0) {
         // got error or connection closed by client
         if (nbytes == 0) {
             // connection closed
             printf("selectserver: socket %d hung up\n", i);
         } else
             perror("recv");
         close(i); // bye!
         FD_CLR(i, &master); // remove from master set
     } else {
         // we got some data from a client
         for(j = 0; j <= fdmax; j++) {
             // send to everyone!
             if (FD_ISSET(j, &master)) {
                 // except the listener and ourselves
                 if (j != listener && j != i) {
                     if (send(j, buf, nbytes, 0) == -1)
                         perror("send");
    }}}}}}}} // Close all brackets
    return 0;
}
```

hpollas@cs4118:~/epl428/lab9

```
[hpollas@cs4118 lab9]$ sjas
-bash: sjas: command not found
[hpollas@cs4118 lab9]$ sel
e
A key was pressed!
[hpollas@cs4118 lab9]$ e
-bash: e: command not found
[hpollas@cs4118 lab9]$ gcc sel
sel             select.c        selectserver.c  selser
[hpollas@cs4118 lab9]$ gcc selectserver.c -o selserver
[hpollas@cs4118 lab9]$ rm selser
rm: remove regular file `selser'? y
[hpollas@cs4118 lab9]$ ls
sel  select.c  selectserver.c  selserver
[hpollas@cs4118 lab9]$ selserver
selectserver: new connection from 127.0.0.1 on socket 4
selectserver: new connection from 127.0.0.1 on socket 5
selectserver: socket 5 hung up
selectserver: new connection from 127.0.0.1 on socket 5
selectserver: new connection from 127.0.0.1 on socket 6
selectserver: socket 4 hung up
selectserver: socket 5 hung up
selectserver: socket 6 hung up
```

hpollas@cs4118:~

```
login as: hpollas
hpollas@cs4118.in.cs.ucy.ac.cy's password:
Last login: Wed Apr  2 14:41:10 2008 from cs5396.cs.ucy.ac.cy
[hpollas@cs4118 ~]$ telnet localhost 9034
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
this is a test
Another test
^]
telnet> quit
Connection closed.
[hpollas@cs4118 ~]$
```

hpollas@cs4118:~

```
login as: hpollas
hpollas@cs4118.in.cs.ucy.ac.cy's password:
Last login: Wed Apr  2 14:41:00 2008 from cs5396.cs.ucy.ac.cy
[hpollas@cs4118 ~]$ telnet localhost 9034
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
^]
telnet> exit
?Invalid command
telnet> quit
Connection closed.
[hpollas@cs4118 ~]$ telnet localhost 9034
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
this is a test
Another test
Last test
^]
telnet> quit
Connection closed.
[hpollas@cs4118 ~]$
```

hpollas@cs4118:~

```
login as: hpollas
hpollas@cs4118.in.cs.ucy.ac.cy's password:
Last login: Wed Apr  2 14:41:23 2008 from cs5396.cs.ucy.ac.cy
[hpollas@cs4118 ~]$ telnet localhost 9034
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
Another test
Last test
Only me
^]
telnet> quit
Connection closed.
[hpollas@cs4118 ~]$
```