Internet Programming

Programming with Sockets
Brief Introduction to Networks
Introduction

- There is **one** way computers can communicate
  - By sending network messages to each other
  - All kinds of communication are built from messages

- There is **one** way programs can send/receive network messages
  - Through sockets
  - All other communication paradigms are built from sockets
Two Different Kinds of Networks

- **Circuit switching**
  - One electrical circuit assigned per communication
  - Example: the (analog) phone network
  - Guaranteed *constant quality* of service
  - *Waste of resources* (periods of silence), fault tolerance

- **Packet switching**
  - Messages are split into *packets*, which are transmitted independently
    - Packets can take different routes
    - Network infrastructures are shared among users
  - Example: the Internet, and most computer networks
  - *Good resource usage, fault tolerance*
  - *Variable QOS*, packets may be delivered in the wrong order
Internet Protocol

- Most computer networks use the Internet Protocol
  - Even if they are not connected to the Internet

- The base protocol: IP (Internet Protocol)
  - Send packets of limited size
    - Up to 65,536 bytes
    - But if the MTU (Maximum Transmission Unit) of some link on the path is lower, the packet will be fragmented (IPv4) or dropped (IPv6)
    - Minimum allowed MTU is 576 bytes; in practice nowadays higher
  - Each packet is sent to an IP address
    - Example: 130.37.193.13
  - IP offers no guarantee:
    - Packets may get lost
    - Packets may be delivered twice
    - Packets may be delivered in wrong order
    - Packets may be corrupted during transfer

- Usually, programs do not use IP directly
  - All other Internet Protocols are built over IP
UDP: User Datagram Protocol

- UDP is very similar to IP
  - Send/receive packets
  - No guarantee

- In UDP, packets are called **datagrams**
  - Each datagram is sent to an **IP address** and a **port number**
  - Example: 130.37.193.13   port=1234

- Ports allow to distinguish between several programs running simultaneously on the same machine
  - Program A uses port 1234
  - Program B uses port 1235
  - When a datagram is received, the OS knows which program it should be delivered to.
TCP: Transmission Control Protocol

- TCP establishes connections between pairs of machines
  - To communicate with a remote host, we must first connect to it

- TCP provides the illusion of a reliable data flow to the users
  - Flows are split into packets, but the users don’t see them
  - Provides flow control and congestion control
    - QUESTION: What is the difference?!?

- TCP guarantees that the data sent will not be lost, unordered or corrupted
  - The sender gives numbers to packets so that the receiver can reorder them
  - The receiver acknowledges received packets so that the sender can retransmit lost packets

- Communication is bidirectional
  - The same connection can be used to send data in both directions
  - E.g., a request and its response
A Very Simplified View

- Establish connections
- Split data streams into packets
- ACK received packets
- Retransmit lost packets
- Reorder packets

- Find how to reach a given host
- If on the same network → easy
- Otherwise → send to a router

- Transform packets into electrical signals
- Make sure signals do not interfere with each other
IP Addressing and Name Resolution
IP Address Conversion

- **IP Addresses**
  - 32-bit integers: **2183468070** (good for computers!)
  - Dotted strings: **130.37.20.38** (good for humans!)
  - DNS name: **www.cs.vu.nl** (even better for humans!)

- You can convert between **integer** and **dotted string**:

  ```c
  #include <arpa/inet.h>
  int inet_aton(const char *dotted, struct in_addr *in); /* Dotted to Network */
  char *inet_ntoa(struct in_addr network); /* Network to Dotted */
  /* number to alphanumeric */
  
  struct in_addr_t: unsigned 32-bit integer
  struct in_addr: structure containing an in_addr_t:
  struct in_addr {
    in_addr_t s_addr;
  };
  
  // there are historic reasons why this is done that way...
  ```
Big/Little-endian, Network Ordering

- Computers represent numbers in different orderings:
  - Is significant byte first or last?
    - Big-endian: 0x12345678 → 0x12 0x34 0x56 0x78
      - examples: PowerPC, Sparc, UltraSparc
    - Little-endian: 0x12345678 → 0x78 0x56 0x34 0x12
      - examples: Alpha, i386, AMD64

- Network byte ordering
  - A standard representation (defined as Big-Endian)

- To convert numbers: host <--- network ordering

```c
#include <netinet/in.h>

uint16_t htons(uint16_t value); /* Host to Network, 16 bits */
uint32_t htonl(uint32_t value); /* Host to Network, 32 bits */
uint16_t ntohs(uint16_t value); /* Network to Host, 16 bits */
uint32_t ntohl(uint32_t value); /* Network to Host, 32 bits */
```
sockaddr_in: Unix Network Addresses

- Unix represents network addresses with a `struct sockaddr`
  - This structure is generic for all kinds of networks
  - For Internet addresses, we use `sockaddr_in`

```c
struct sockaddr_in {
    sa_family_t sin_family; /* set to AF_INET */
    in_port_t sin_port;    /* Port number */
    struct in_addr sin_addr; /* Contains the IP address */
};

struct in_addr {
    in_addr_t s_addr;     /* IP address in network ordering */
};
```

- `sin_family`: indicates which type of address. Always set to AF_INET.
- `sin_port`: port number, in network byte order
- `sin_addr.s_addr`: IP address, in network byte order. To represent an unspecified IP address, set it to `htonl(INADDR_ANY)`.

- QUESTION: When is this useful?
Domain Names

- Internet Protocols are all based on IP addresses
  - But IP addresses are hard for humans to remember
  - Our web server: http://130.37.20.20
  - Better: http://www.cs.vu.nl

- Using Domain Names
  - Domain names cannot be used directly by network protocols
  - Network protocols only use IP addresses
  - But you can convert domain names into IP addresses thanks to DNS

- Domain Name Service (DNS): handles Domain Name resolution
  - Hundreds of thousands of servers around the world that cooperate to resolve addresses
  - To learn more on how this works, go to the Distributed Systems course!
Converting Domain Names to IP

- **Old API:** `gethostbyname()`

```c
#include <netdb.h>
struct hostent *gethostbyname(const char *name);
```

- **...where** `struct hostent` **is as follows**

```c
struct hostent {
    char   *h_name;          /* official name of host */
    char   **h_aliases;      /* alias list */
    int    h_addrtype;       /* host address type */
    int    h_length;         /* length of address */
    char   **h_addr_list;     /* list of addresses */
};
```

- **h_addr_list:** A null-terminated array of network addresses for the host
```c
#include <netdb.h>

int print_resolv(const char *name) {
    struct hostent *resolv;
    struct in_addr *addr;

    resolv = gethostbyname(name);
    if (resolv==NULL) {
        printf("Address not found for %s\n", name);
        return -1;
    }
    else {
        addr = (struct in_addr*) resolv->h_addr_list[0];
        printf("The IP address of %s is %s\n", name, inet_ntoa(*addr));
        return 0;
    }
}
```
Converting Domain Names to IP

- **New API:** getaddrinfo()
  - Thread-safe, Supports IPv6

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *nodename, const char *servname,
    const struct addrinfo *hints, struct addrinfo **res);
```

- ...where **struct addrinfo** is as follows

```c
struct addrinfo {
    int      ai_flags;       // AI_PASSIVE, AI_CANONNAME, ...
    int      ai_family;      // AF_xxx
    int      ai_socktype;    // SOCK_xxx
    int      ai_protocol;    // 0 (auto) or IPPROTO_TCP, IPPROTO_UDP

    socklen_t ai_addrlen;    // length of ai_addr
    char    *ai_canonname;   // canonical name for nodename
    struct sockaddr *ai_addr; // binary address
    struct addrinfo *ai_next; // next structure in linked list
};
```
Example:

```c
int sockfd;
struct addrinfo hints, *servinfo, *p;
int rv;

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC; // use AF_INET6 to force IPv6
hints.ai_socktype = SOCK_STREAM;

ai = getaddrinfo("www.example.com", "http", &hints, &servinfo);
if (ai != 0) {...} // error

for (p = servinfo; p != NULL; p = p->ai_next) {
    ...
    // try to connect to that socket
    ...
}

freeaddrinfo(servinfo); // all done with this structure
```
UDP Sockets
UDP

- Defined in RFC 768

- Popular UDP-based protocols
  - DNS – Domain Name System
  - NFS – Network File System
  - SNMP – Simple Network Management Protocol
  - DHCP – Dynamic Host Configuration Protocol
  - RIP – Routing Information Protocol
UDP Socket Functions

UDP client

- `socket()`
- `sendto()`
- `recvfrom()`
- `close()`

UDP server

- `socket()`
- `bind()`
- `recvfrom()` (blocks until datagram received)
- `sendto()`
- `close()`
Creating a Socket

- socket() creates a new socket (for either UDP or TCP)

```c
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

- To create an Internet socket, use:
  - `domain = AF_INET`
  - `type = SOCK_DGRAM` for UDP, `SOCK_STREAM` for TCP
  - `protocol = 0`
  - **Return value**: socket descriptor, or -1 for error
bind(): Assign an Address to a Socket

- **bind()** is used to specify the address of the socket
  - IP addr + port number

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

- **sockfd**: the socket descriptor
- **my_addr**: a pointer to struct sockaddr_in (containing the address)
- **addrlen**: the size of struct sockaddr_in
- **Return value**: 0 for success, -1 for error

- If you don’t specify them, the system gives them a value:
  - IP address → the IP address of the running host (this is usually correct)
  - Port number → any number (of an unused port)

- **QUESTION**: When do you need to specify a port number, and when can you omit it?
Example use of socket() and bind()

- To create a UDP socket on port 1234:

```c
int fd, err;
struct sockaddr_in addr;

fd = socket(AF_INET, SOCK_DGRAM, 0);
if (fd<0) { ... }

addr.sin_family = AF_INET;
addr.sin_port = htons(1234);
addr.sin_addr.s_addr = htonl(INADDR_ANY);

err = bind(fd, (struct sockaddr *) &addr, sizeof(struct sockaddr_in));
if (err<0) { ... }
```

- For historic reasons, you are obliged to explicitly cast your struct sockaddr_in * into a struct sockaddr *

- **QUESTION:** Is INADDR_ANY equivalent to 127.0.0.1?
Sending and Receiving Datagrams

- `sendto()` and `recvfrom()` are very similar:

```c
#include <sys/types.h>
#include <sys/socket.h>

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

```c
#include <sys/types.h>
#include <sys/socket.h>

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

<table>
<thead>
<tr>
<th><code>sockfd</code></th>
<th>socket descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>*buf</code></td>
<td>buffer of message to send/receive</td>
</tr>
<tr>
<td><code>len</code></td>
<td>buffer length</td>
</tr>
<tr>
<td><code>flags</code></td>
<td>0</td>
</tr>
<tr>
<td><code>*to / *from</code></td>
<td>destination address</td>
</tr>
<tr>
<td><code>tolen / *fromlen</code></td>
<td>sizeof(struct sockaddr)</td>
</tr>
</tbody>
</table>

Return value: Number of bytes sent/received, or -1 for error
Use of `sendto()` and `recvfrom()`

- **Example:**

  (the socket is already created)

  ```c
  char msg[64];
  int err;
  struct sockaddr_in dest;
  
  strcpy(msg,"hello, world!");
  
  dest.sin_family = AF_INET;
  dest.sin_port = htons(1234);
  dest.sin_addr.s_addr = inet_addr("130.37.193.13");
  
  err = sendto(fd,
               msg,
               strlen(msg)+1,
               0,
               (struct sockaddr*) &dest,
               sizeof(struct sockaddr_in));
  
  if (err<0) { ... }
  ```

  (the socket is created and bound to a well-known port)

  ```c
  char msg[64];
  int len, flen;
  struct sockaddr_in from;
  
  flen = sizeof(struct sockaddr_in);
  len = recvfrom(fd,
                 msg,
                 sizeof(msg),
                 0,
                 (struct sockaddr*) &from,
                 &flen);
  
  if (len<0) { ... }
  
  printf("Received %d bytes from host %s port %d: %s",
          len,
          inet_ntoa(from.sin_addr),
          ntohs(from.sin_port),
          msg);
  ```
Closing a socket

- Sockets must be closed when they are no longer in use:

```c
#include <unistd.h>
int close(int sockfd);
```

- **sockfd**: the socket descriptor
- **Return value**: 0 for success, -1 for error
TCP Sockets
TCP

- Defined in RFC 793

- Popular TCP-based protocols
  - TELNET
  - FTP – File Transfer Protocol
  - SMTP – Simple Mail Transfer Protocol
  - HTTP – Hyper Text Transfer Protocol
  - SSH – Secure Shell
TCP Socket Functions

TCP client

socket()

connect()

write()

read()

close()

TCP server

socket()

bind()

listen()

accept()

read()

write()

read()

close()

3-way handshake
The TCP three-way handshake

Client

\text{syn seq=x}

\text{syn ack=x+1 seq=y}

\text{ack=y+1 seq=x+1 [data]}

Server
Creating a Socket

- Some functions are the same as in UDP

- `socket()`: creates a socket

```c
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

- `bind()`: to specify the address of a socket
  - Only useful for server sockets
  - Exactly like UDP sockets
listen(): Setting a Server Socket

- By default, TCP sockets are created as client sockets
  - A client socket cannot receive incoming connections

- Server sockets need to maintain more state
  - TCP establishes connections thanks to the three-way handshake:
    - The client sends a connection request
    - The server answers
    - The client acknowledges the server’s answer
  - Server sockets must allocate resources for handling connections

- To convert a client socket to a server socket, use listen()
  - And indicate how many not-yet-accepted connections can be supported in parallel
  - If this number is exceeded, the server will refuse connections
The interface is simple

```c
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

- `sockfd`: the socket descriptor
- `backlog`: the size of the buffer (often set to 5)
- **Return value**: 0 for success, -1 for error

**Note**
- `backlog` is **not** a limit on the number of connections established in parallel!
- It only limits the number of **pending connections** (i.e., connections before having been accepted)
Initiating a TCP connection

- Clients initiate connections to servers thanks to `connect()`:

  ```c
  #include <sys/types.h>
  #include <sys/socket.h>
  int connect(int sockfd, 
               const struct sockaddr *serv_addr, 
               socklen_t addrlen);
  ```

  - **sockfd**: the socket descriptor
  - **serv_addr**: a pointer to a struct `sockaddr_in` containing the address where to connect to
    - Obviously you must specify the destination IP address and port number
  - **addrlen**: `sizeof(struct sockaddr_in)`
  - **Return value**: 0 for success, -1 for error

- `connect()` binds the client’s socket to a random unused port

- **QUESTION**: When can it fail?
Possible Connection Errors

- **Destination Unreachable**
  - The SYN message receives an ICMP error from some intermediate router
  - The client keeps trying for some time
  - After some time, it returns an EHOSTUNREACH error to the process

- **Server reached, but no process bound to that port**
  - The SYN message receives a RST (Reset) reply from the server’s TCP
  - The client’s TCP returns an ECONNREFUSED error to the process instantly

- **Server reached, but listen backlog exhausted**
  - Like the previous case

- **No response → timeout**
  - The SYN message receives no response
  - The client’s TCP sends SYN message again (after 6sec, 24sec)
  - After some time, an ETIMEDOUT error is returned to the client process
Waiting for an Incoming Connection

- **accept()** blocks the process until an incoming connection is received
  - When a connection is received, **accept() creates a new socket** dedicated to this connection
    - The new socket is used to communicate with the client
    - The original socket is immediately ready to wait for other connections

- **accept()**:
  ```c
  #include <sys/types.h>
  #include <sys/socket.h>
  int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
  ```
  - **sockfd**: the socket descriptor
  - **addr**: a pointer to a sockaddr_in structure where the address of the client will be copied
  - **addrlen**: a pointer to an integer containing the size of **addr**
  - **Return value**: the descriptor of the newly created socket, or -1 for error
Example use of accept()

```c
int sock, newsock, res;
sockaddr_in client_addr;
socklen_t addrlen;

(the socket sock is created and bound)

res = listen(sock,5);
if (res < 0) { ... }

addrlen = sizeof(struct sockaddr_in);

newsock = accept(sock, (struct sockaddr *) &client_addr, &addrlen);
if (newsock < 0) { ... }
else
{
    printf(“Connection from %s!\n”, inet_ntoa(client_addr.sin_addr));
}
```
Writing data to a socket

- write works the same for sending data to a TCP socket or writing to a file

```
#include <unistd.h>
ssize_t write(int sockfd, const void *buf, size_t count);
```

- sockfd: socket descriptor
- buf: buffer to be sent
- count: size of buffer
- Return value: number of bytes sent, or -1 for error

- Attention: When writing to a socket, write may send fewer bytes than requested
  - Due to limits in internal kernel buffer space

- Always check the return value of write, and resend the non-transmitted data
A wrapper function (from Stevens book, page 78)

```c
ssize_t writen(int fd, const void *vptr, size_t n)
{
    size_t nleft;
    ssize_t nwritten;
    const char *ptr;

    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ( ((nwritten = write(fd, ptr, nleft)) <= 0 ) ) {
            if (errno == EINTR)
                nwritten = 0; /* and call write() again */
            else
                return -1; /* error */
        }
        nleft -= nwritten;
        ptr += nwritten;
    }
    return n;
}
```
# Reading data from a socket

- read() blocks the process until receiving data from the socket

```
#include <unistd.h>
ssize_t read(int sockfd, void *buf, size_t count);
```

- **sockfd**: socket descriptor
- **buf**: buffer where to write the data read
- **count**: size of buffer
- **Return value**: number of bytes read, or -1 for error

- **Attention**: When reading from a socket, read() may read fewer bytes than requested
  - It delivers the data that have been received
  - This does not mean that the stream of data is finished, there may be more to come
  - The end-of-file (EOF) is notified to the read by read() returning 0
Closing a TCP socket

- To stop sending data to a socket, use `close()`:  

```c
#include <unistd.h>
int close(int sockfd);
```

- Anyone can call this, either the client or the server

- This sends an EOF message to the other party
  - When receiving an EOF, `read` returns 0 bytes
  - Subsequent reads and writes will return errors
Asymmetric Disconnection

- Sometimes you may want to tell the other party that you are finished, but let it finish before closing the connection

```c
#include <sys/socket.h>
int shutdown(int sockfd, int how);
```

- **how**: SHUT_WR for stopping writing, SHUT_RD for stopping reading

- When one party has closed the connection, the other can still write data (and then close the connection as well)

<table>
<thead>
<tr>
<th>To initiate a disconnection</th>
<th>To receive a disconnection</th>
</tr>
</thead>
<tbody>
<tr>
<td>shutdown(fd, SHUT_WR)</td>
<td>read() keeps receiving data</td>
</tr>
<tr>
<td>Keep on reading the last data</td>
<td>read() receives EOF</td>
</tr>
<tr>
<td>Until receiving an EOF</td>
<td>Keep on writing the last data</td>
</tr>
<tr>
<td>close() the socket</td>
<td>close() the socket</td>
</tr>
</tbody>
</table>
Socket Options
Socket Options

- Various attributes that are used to determine the behavior of sockets.

- Setting options tells the OS/Protocol Stack the behavior we want.

- Support for generic options (apply to all sockets) and protocol specific options.
Option types

- Many socket options are **boolean flags** indicating whether some feature is enabled (1) or disabled (0).

- Other options are associated with more complex types including `int`, `timeval`, `in_addr`, `sockaddr`, etc.

- Some options are read-only (we can’t set the value)
Getting and Setting option values

- Use `getsockopt()` and `setsockopt()`:

```c
#include <sys/socket.h>
int getsockopt(int sockfd, int level, int optname,
               void *optval, socklen_t *optlen);
int setsockopt(int sockfd, int level, int optname,
               const void *optval, socklen_t optlen);
```

- **sockfd**: the socket descriptor
- **level**: the protocol this option refers to
  - SOL_SOCKET (for socket options)
  - IPPROTO_TCP
  - IPPROTO_UDP
  - IPPROTO_IP
- **optname**: the option name (a #define)
- **optval**: a buffer containing the option value
- **optlen**: the length of optval
General Options

- Protocol independent options

- Handled by the generic socket system code

- Some general options are supported only by specific types of sockets (SOCK_DGRAM, SOCK_STREAM)
Some Generic Options

- SO_BROADCAST
- SO_DONTROUTE
- SO_ERROR
- SO_KEEPALIVE
- SO_LINGER
- SO_RCVBUF, SO_SNDBUF
- SO_REUSEADDR
SO_BROADCAST

- Boolean option: enables/disables sending of broadcast messages
- Underlying Data Link layer must support broadcasting!
- Applies only to SOCK_DGRAM sockets
- Prevents applications from sending broadcasts by mistake
  - OS looks for this flag when broadcast address is specified
  - e.g., ping –b 130.37.193.255
SO_DONTROUTE

- Boolean option: enables bypassing of normal routing
  - If destination is directly attached to an interface, OK.
  - Otherwise, ENETUNREACH is returned

- Used by routing daemons (*routed*, *gated*, etc.)
  - Bypasses the routing table
  - The routing table could be incorrect
SO_ERROR

- Integer value option
  - The value is an error indicator value (similar to errno)
  - Read only!

- Reading (by calling getsockopt()) clears any pending error
  - so, you can only read it once
**SO_KEEPALIVE**

- **Boolean option**
  - enabled means that TCP sockets should send a probe to peer if no data flow for a “long time”
  - Allows a process to determine whether peer process/host has crashed

- **QUESTION:** Consider what would happen to an open telnet or ssh connection without keepalive

- Typically used by servers
  - some sort of garbage collection of terminated clients
SO_LINGER

- Value is of type:

```c
#include <sys/socket.h>

struct linger {
    int l_onoff;    /* 0 = off */
    int l_linger;   /* time in seconds */
};
```

- Controls whether and how long a call to `close()` waits for pending ACKs
  - `l_onoff = 0` → default behavior: `close()` returns immediately, and the system tries to deliver any pending packets from the send buffer
  - `l_onoff = 1` → `close()` returns when:
    - either all pending packets have been sent and acknowledged by the remote TCP stack
    - or `l_linger` seconds have elapsed

- Only for connection-oriented sockets (e.g., TCP)
SO_LINGER

- Without SO_LINGER, the node closing a connection has no way of knowing that the other peer received all sent data.

- With SO_LINGER, it can know that all sent data reached the other peer’s TCP stack.

**QUESTION:** Isn’t this what shutdown(fd, SHUT_WR) does?

- Not exactly!
- shutdown(fd, SHUT_WR) closes your own write channel, but you can still read.
- Reading till you receive EOF (read() returns 0 bytes) means that the other peer did a close()
- Therefore, shutdown() assures you that the other peer has read all your data, while SO_LINGER assures you that your data reached the other peer’s TCP stack, but you don’t know if it was read!
SO_REUSEADDR

- Boolean option
  - Enables binding to an address (port) that is already in use.

- Used by servers that are transient
  - Allows binding a passive socket to a port currently in use (with active sockets) by other processes.

- Can be used to establish separate servers for the same service on different interfaces (or different IP addresses on the same interface)
  - Virtual Web Servers can work this way

- Very useful in your assignments!
SO_RCVBUF, SO_SNDBUF

- Integer values options
  - Change the receive and send buffer sizes.

- Can be used with STREAM and DGRAM sockets.

- With TCP, this option affects the window size used for flow control
  - Must be established before the connection is made
SO_RCVLOWAT, SO_SNDLOWAT

- The stand for RECEIVE/SEND LOW WATERMARK.

- Integer values options
  - Defines the lowest number of bytes to be read/written
  - `read()` will remain blocked until at least that many bytes are available
  - Data written by `write()` will remain buffered until at least that many have accumulated

- Can be used with STREAM.
I/O Multiplexing
I/O Multiplexing

- We saw so far how a process can handle a single connection
  - `accept()` and `read()` block until something is received on a given socket

- How can a program handle multiple sockets?
  - Use multiple processes
  - Use non-blocking I/O
    - It works for `read()` but not for `accept()`

- `select()` monitors multiple file descriptors
  - It blocks the process until any one of them is ready for reading or writing

- `poll()`, `epoll()`, `libevent` are alternatives to `select()`
  - With additional flexibility but also complexity
select() [1/3]

- select() monitors several file descriptors simultaneously

```c
#include <sys/select.h>
int select(int n,
    fd_set *readfds,
    fd_set *writefds,
    fd_set *exceptfds,
    struct timeval *timeout);
```

- **n**: the highest numbered file descriptor, plus 1
- **readfds**: a list of file descriptors to monitor for reading
- **writefds**: a list of file descriptors to monitor for writing
- **exceptfds**: a list of file descriptors to monitor for exceptions
- **timeout**: a duration after which select() returns anyway.
  - Set `timeout.tv_sec = timeout.tv_usec = 0` for zero timeout (return immediately)
  - Set `timeout = NULL` for no timeout
- **Return value**: the number of (i.e., how many) descriptors ready for I/O, or 0 in case of timeout, or -1 for error
select() [2/3]

- `fd_set` is a bitset representing a list of file descriptors
  - Do not manipulate `fd_set` directly, always use special macros:

    ```c
    FD_ZERO (fd_set *set);    /* clears all bits */
    FD_SET (int fd, fd_set *set);    /* turns on bit 'fd' */
    FD_CLR (int fd, fd_set *set);    /* turns off bit 'fd' */
    FD_ISSET (int fd, fd_set *set);  /* checks if bit 'fd' is set */
    ```

- `select()` modifies the contents of `readfds`, `writefds`, and `exceptfds`.

- After `select()`:
  - File descriptors that are ready for use are turned on (in their set)
  - Non-ready descriptors are turned off

- To wait for a socket to be ready to `accept()`, put it in the read set.
Example use:

```c
int nb, fd1=5, fd2=8;
char buf[1024];
fd_set read_set;

while (1) {
    FD_ZERO(&read_set);
    FD_SET(fd1, &read_set);
    FD_SET(fd2, &read_set);

    nb = select(20, &read_set, NULL, NULL, NULL);
    if (nb<=0) { ... }
    if (FD_ISSET(fd1, &read_set)) {
        bzero(buf,1024);
        nb = read(fd1, buf, 1024);
        if (nb<0) { ... }
        if (nb==0) printf("Received EOF on fd1!\n");
        else printf("Received data on fd1: %s\n", buf);
    }
    if (FD_ISSET(fd2, &read_set)) { ... }
}
```
Server Structures
Server Structures

- Often, a server accepts connections to one (TCP) socket
  - But it wants to process several requests in parallel
  - Better use of server’s resources
  - Incoming requests can start being executed immediately after reception (not having to wait for previous client)

- Depending on its nature, a server can receive between 0 and dozens of thousands requests per second

- Several server structures can be used:
  - Iterative (i.e., not concurrent)
  - One Process Per Client
  - Preforking
  - select() loop
  - Many other variants...
Iterative Servers

- An iterative server treats one request after the other

```c
int fd, newfd;
while (1) {
    newfd = accept(fd, ...);
    treat_request(newfd);
    close(newfd);
}
```

- Simple

- Potentially low resource utilization
  - If `treat_request()` does not utilize all the CPU, resources are being wasted

- Potentially long queue of incoming connections waiting for `accept()`
  - Increased service latency
  - If the queue increases, the server may start rejecting new connections
One Process Per Client [1/2]

- A new child process is created to handle each connection
  - Also known as “One Child Per Client”

```c
void sig_chld(int) {
    while (waitpid(0, NULL, WNOHANG) > 0) {}   
signal(SIGCHLD, sig_chld);
}

int main() {
    int fd, newfd, pid;

    // socket(), bind(), and listen() have been omitted

    signal(SIGCHLD, sig_chld);
    while (1) {
        newfd = accept(fd, ...);
        if (newfd < 0) continue;
        pid = fork();
        if (pid==0) { treat_request(newfd); exit(0); }  
            else { close(newfd); } 
    }
}
```
One Process Per Client [2/2]

- This is the most common type of concurrent server
  - Several requests are treated simultaneously
  - Incoming requests are accepted and treated immediately

- QUESTION: What are the downsides?
  - It may not be suitable for highly loaded servers
    - Ok for ~1K connections per day, but for ~1M?
    - fork() takes a lot of time!

- MORE QUESTIONS!
  - Why is the \texttt{signal()} call necessary?
  - What happens if \texttt{treat_request()} needs to modify a global variable?
    - How can you obtain the desired effect?
    - e.g., to update request statistics
Preforking

- The server first creates a pool of processes dedicated to treating requests
- Each client is delegated to a child process
- No `fork()` is done per connecting client
- Performance gain!
Preforking: Each child calls accept()

- Typical example

```c
#define NB_PROC 10

void recv_requests(int fd) { /* An iterative server */
    int newfd;
    while (1) {
        newfd = accept(fd,...);
        treat_request(newfd);
        close(newfd);
    }
}

int main() {
    int fd=socket(AF_INET, SOCK_STREAM, 0);
    // bind() and listen() have been omitted

    for (int i=0; i<NB_PROC; i++) /* Create NB_PROC children */
        if (fork()==0)
            recv_requests(fd);
}
```
Preforking: Each child calls accept()

- Multiple accepts on a single socket descriptor
  - In System V types of Unix (Solaris, HP-UX, etc.), not possible!
  - However, allowed in BSD-based Unix systems

- Where it is allowed, accept() internally defines a condition variable
  - Condition: “Block until a client opens a connection”
  - Predicate: “Is a client available in the queue so I can accept it?”

- QUESTION: Can you guess some problem?
  - When a client appears, all processes waiting on that condition variable are woken up
  - Only one is allowed to run at a time, so the first one accepts the client
  - When other processes get the mutex, they first check if the predicate is still valid, and block again since the client is already accepted
  - For large process pools \( \rightarrow \) performance penalty per connecting client
Preforking: Mutex on accept()

- A solution is to protect access to accept() by a single mutex shared by all processes:

```c
void recv_requests(int fd) { /* An iterative server */
    int f;
    while (1)
    {
        /* --- ACQUIRE MUTEX --- */
        newfd=accept(fd,...);
        /* --- RELEASE MUTEX --- */

        treat_request(newfd);
        close(f);
    }
}
```

- For systems NOT supporting concurrent accepts on a single socket:
  - Solves the problem

- For the rest systems:
  - Improves performance
Preforking: accept() on parent

- Another alternative:
  - accept on the parent
  - pass socket descriptor to a free child

- Difficulties?
  - Keeping track of which children are free or loaded
  - Communicating to the children
    - Keeping open pipes
    - Using shared memory, semaphores, condition variables
    - Requires descriptor passing (out of the scope of this course)

- Advantages?
  - Parent has control over which processes to delegate clients to
  - Can achieve more efficient memory usage by using the same processes
One Thread Per Client

- Very similar to One Process Per Client
  - Attention to pay with race conditions, due to shared memory!

- Starting a thread per client is faster than preforking a pool of processes!
Prethreading

- Very similar to preforking

- Even faster than One Thread Per Client (which is already faster than Preforking)

- Locking access to `accept()` by a mutex, further improves performance
Forking vs. Threading

- If threading is faster than forking, why not always use threading?
  - The system might not support threads
  - The application might be easier to design as separate processes
  - In certain cases, One Process Per Client can be faster than One Thread Per Client

- QUESTION: When?!

- ANSWER: When the code treating a client needs to execute another program. Then, forking a unithreaded process and then doing an exec would be faster than creating a new thread, then forking a multithreaded process and finally doing the exec
Select Loop

- A single process (and single thread) manages multiple connections thanks to `select()`
  - This is quite **difficult** to do correctly
  - You must split request treatment into a set of non-blocking stages
  - You must maintain a list of data structures containing the current state of each concurrent request
    - Which stage it is in
    - All internal data it needs

- Implementing a select loop server is left to the students as exercise 😊

- For example, the *Squid Web* cache is implemented as a select loop
For servers with **low load**, it is recommended to use

- One Process Per Client
- ...or even Iterative

For servers with **extremely low load** (e.g., one invocation per day or week)

- inetd
- No need to run the server at all, while not needed
- Define the service’s protocol (TCP/UDP), port, and executable in /etc/services
- inetd listens on that port
- Upon reception of a client, it forks and execs the executable