CSC 4304 - Systems Programming
Fall 2008

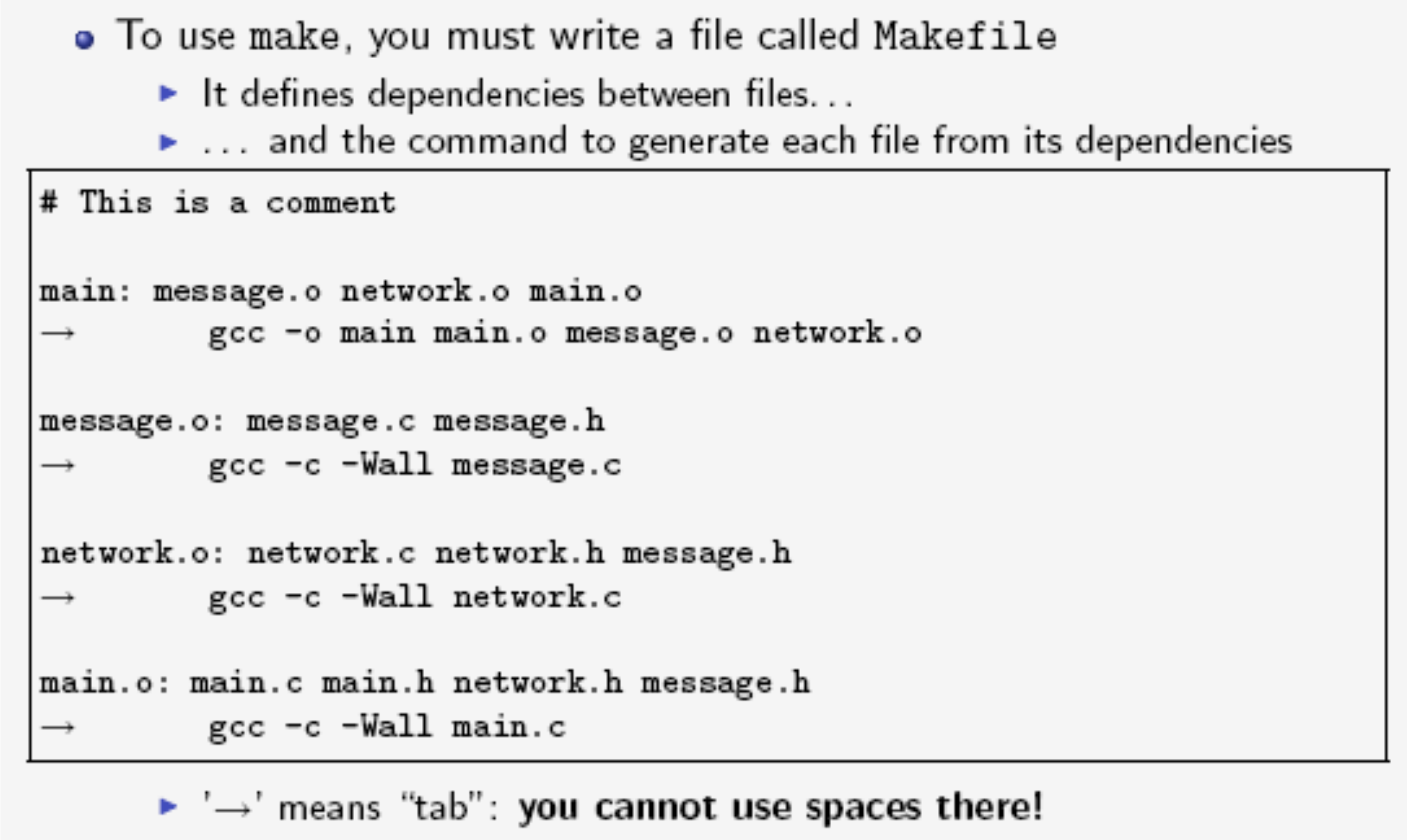
Lecture - XXIII
Final Review

Tevfik Koşar

Louisiana State University

December 4th, 2008

Using make

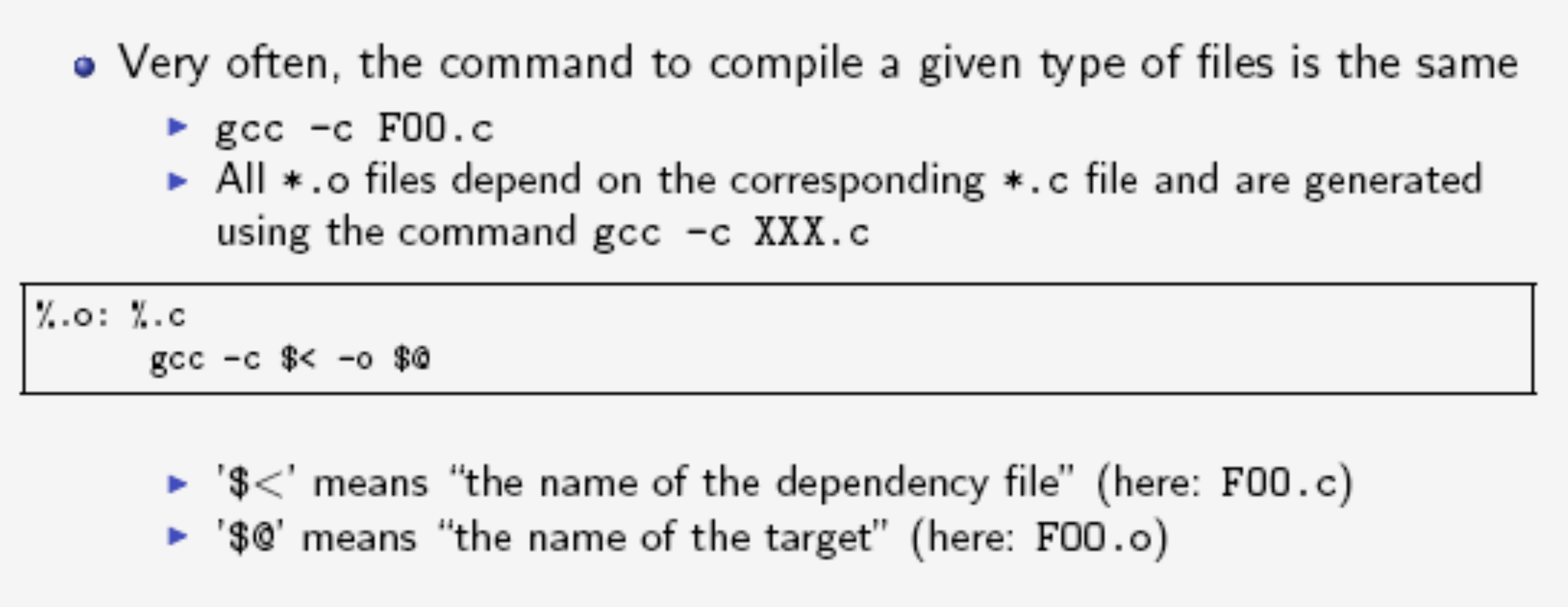


- To use make, you must write a file called Makefile
  - ▸ It defines dependencies between files...
  - ▸ ... and the command to generate each file from its dependencies

```
# This is a comment

main: message.o network.o main.o
→       gcc -o main main.o message.o network.o

message.o: message.c message.h
→       gcc -c -Wall message.c

network.o: network.c network.h message.h
→       gcc -c -Wall network.c

main.o: main.c main.h network.h message.h
→       gcc -c -Wall main.c
```

  - ▸ '→' means "tab": **you cannot use spaces there!**

Implicit Rules

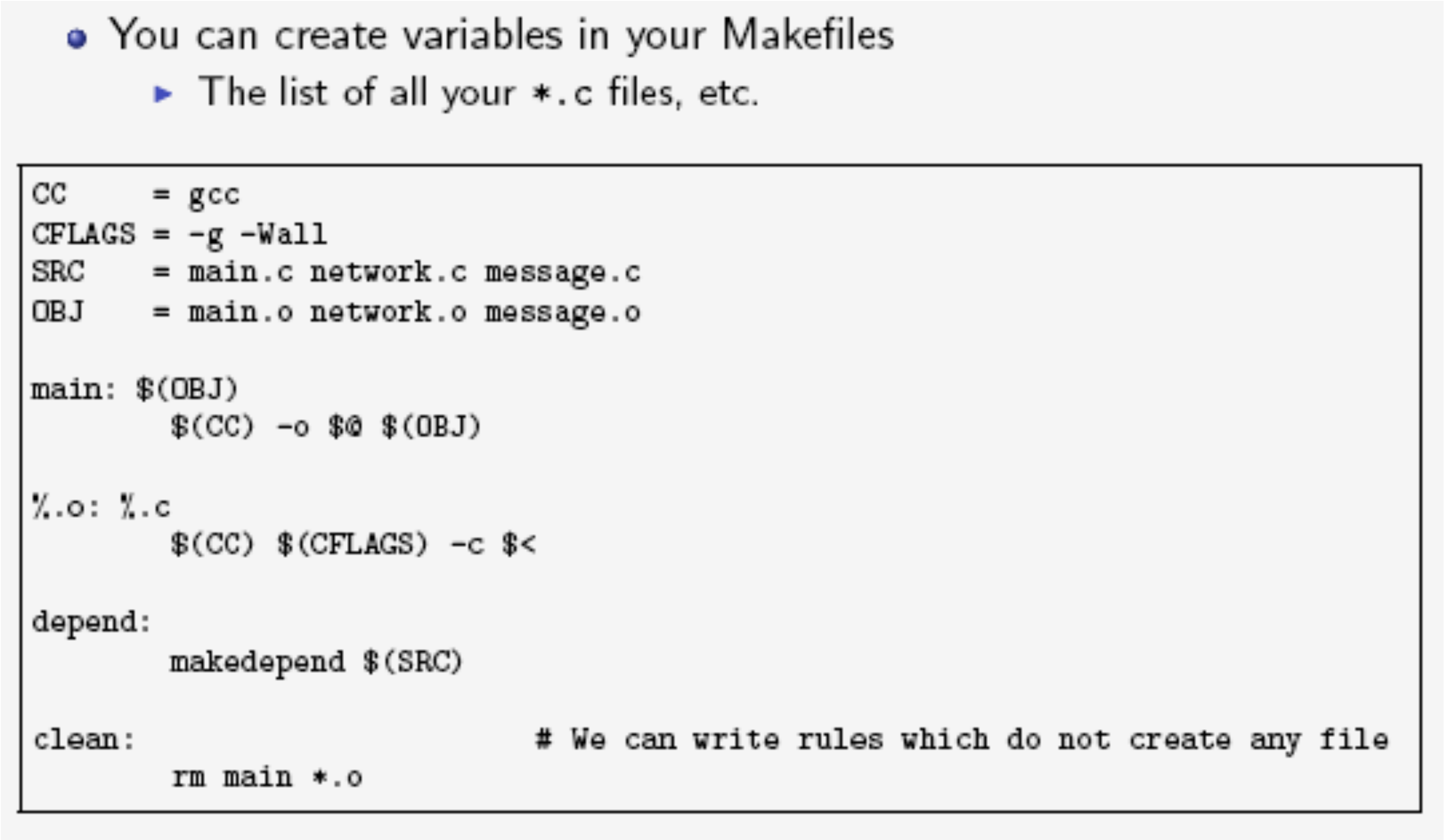


- Very often, the command to compile a given type of files is the same
  - gcc -c FOO.c
  - All *.o files depend on the corresponding *.c file and are generated using the command gcc -c XXX.c

```
%.o: %.c
        gcc -c $< -o $@
```

  - '$<' means "the name of the dependency file" (here: FOO.c)
  - '$@' means "the name of the target" (here: FOO.o)

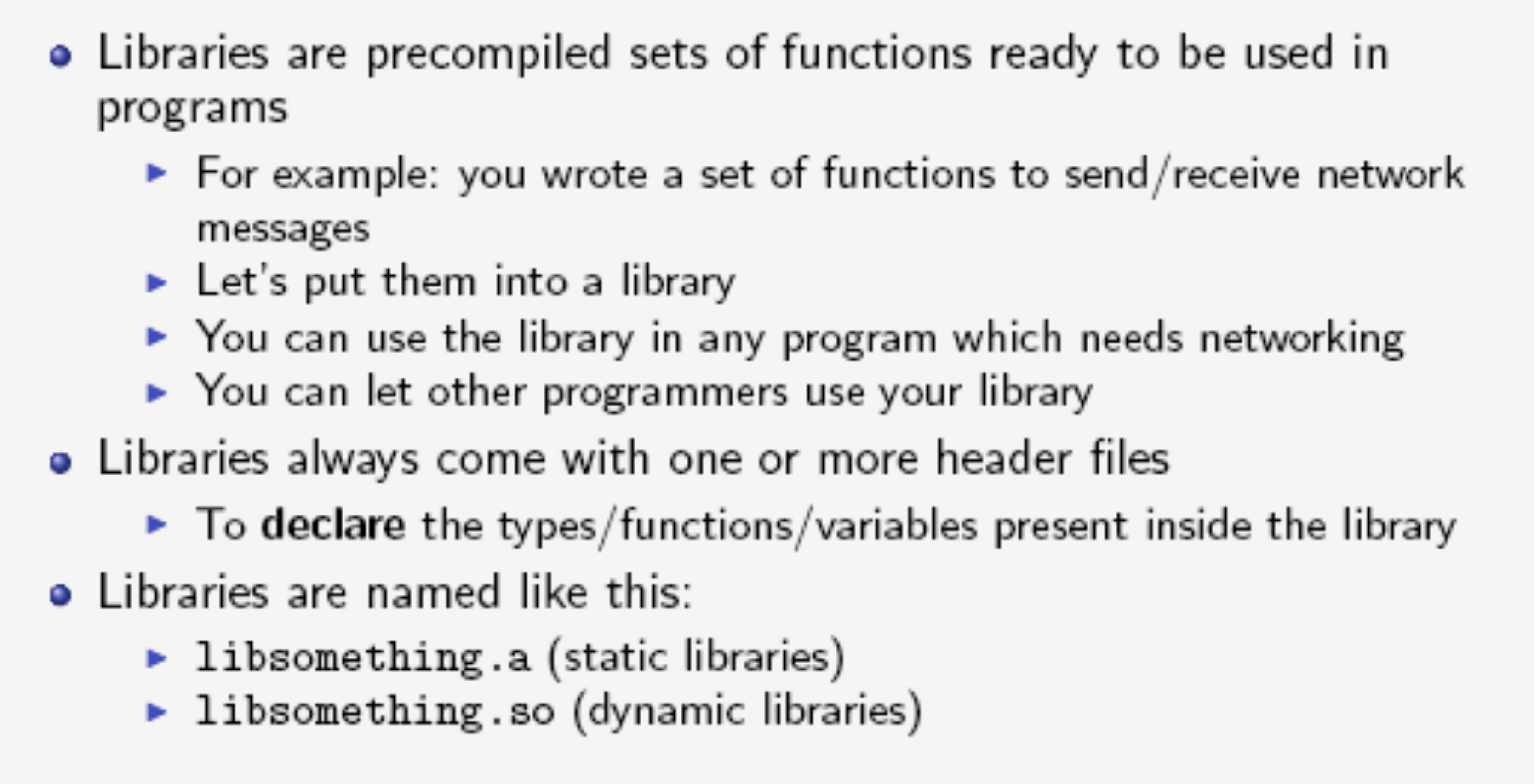
Using Variables in Makefiles



- You can create variables in your Makefiles
  - The list of all your *.c files, etc.

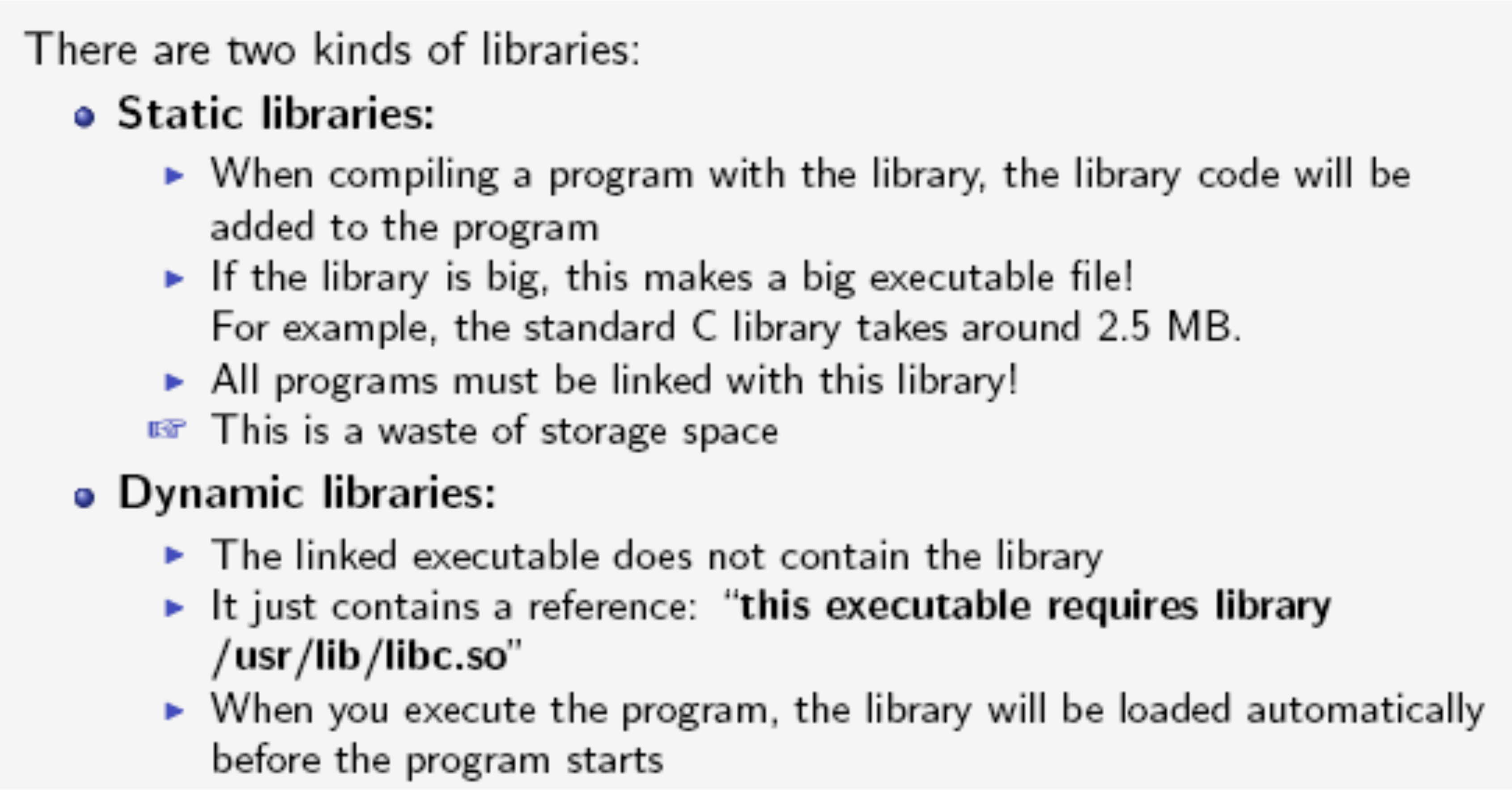
```
CC     = gcc
CFLAGS = -g -Wall
SRC    = main.c network.c message.c
OBJ    = main.o network.o message.o

main: $(OBJ)
        $(CC) -o $@ $(OBJ)

%.o: %.c
        $(CC) $(CFLAGS) -c $<

depend:
        makedepend $(SRC)

clean:                      # We can write rules which do not create any file
        rm main *.o
```

Libraries



- Libraries are precompiled sets of functions ready to be used in programs
  - For example: you wrote a set of functions to send/receive network messages
  - Let's put them into a library
  - You can use the library in any program which needs networking
  - You can let other programmers use your library
- Libraries always come with one or more header files
  - To **declare** the types/functions/variables present inside the library
- Libraries are named like this:
  - `libsomething.a` (static libraries)
  - `libsomething.so` (dynamic libraries)

Static vs Dynamic Libraries



There are two kinds of libraries:

- **Static libraries:**
  - ▸ When compiling a program with the library, the library code will be added to the program
  - ▸ If the library is big, this makes a big executable file!
    For example, the standard C library takes around 2.5 MB.
  - ▸ All programs must be linked with this library!
  - ☞ This is a waste of storage space
- **Dynamic libraries:**
  - ▸ The linked executable does not contain the library
  - ▸ It just contains a reference: "**this executable requires library /usr/lib/libc.so**"
  - ▸ When you execute the program, the library will be loaded automatically before the program starts

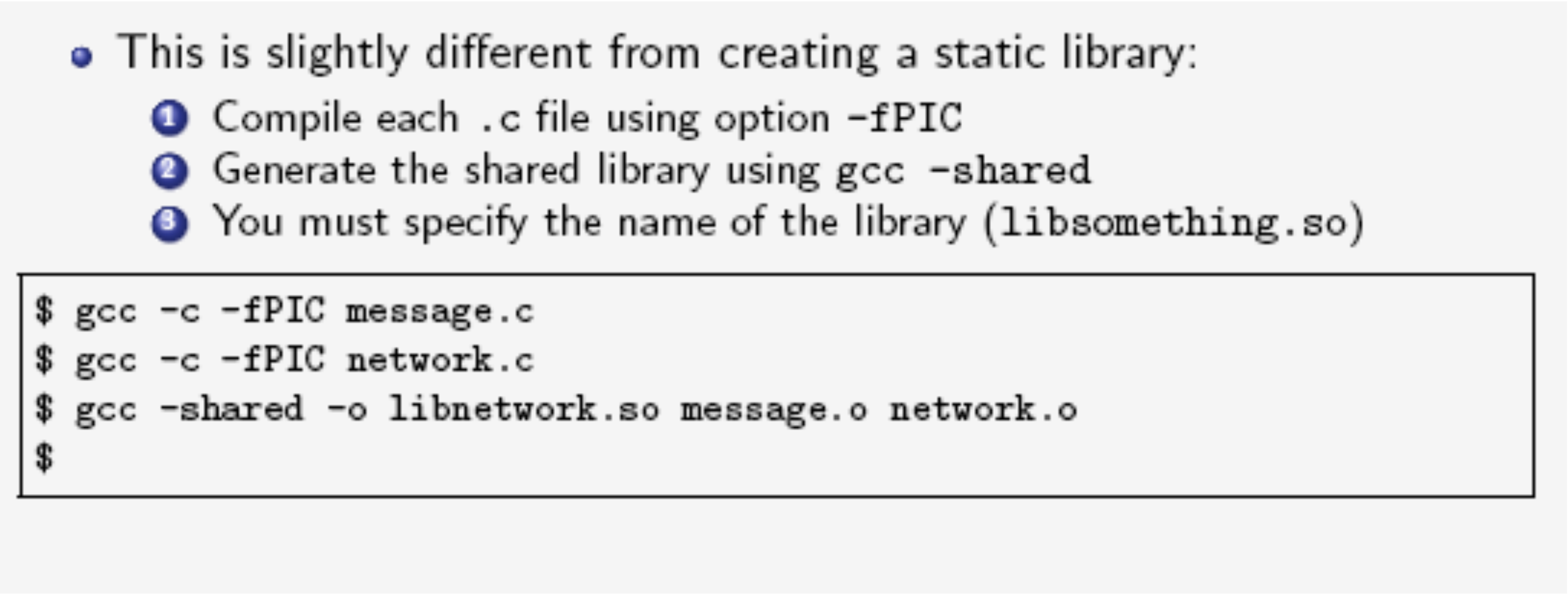
Crating a Static Library

- A static library is made of a number of *.o files

```
$ ar r libnetwork.a message.o network.o
$ ranlib libnetwork.a
$
```

- ar creates the library
  - It can also do other operations on libraries (modifying an existing library, extracting parts from a library, etc.)
  - Read the man page. . .
- ranlib creates an **index** of all functions in the library
  - And adds the index to the library

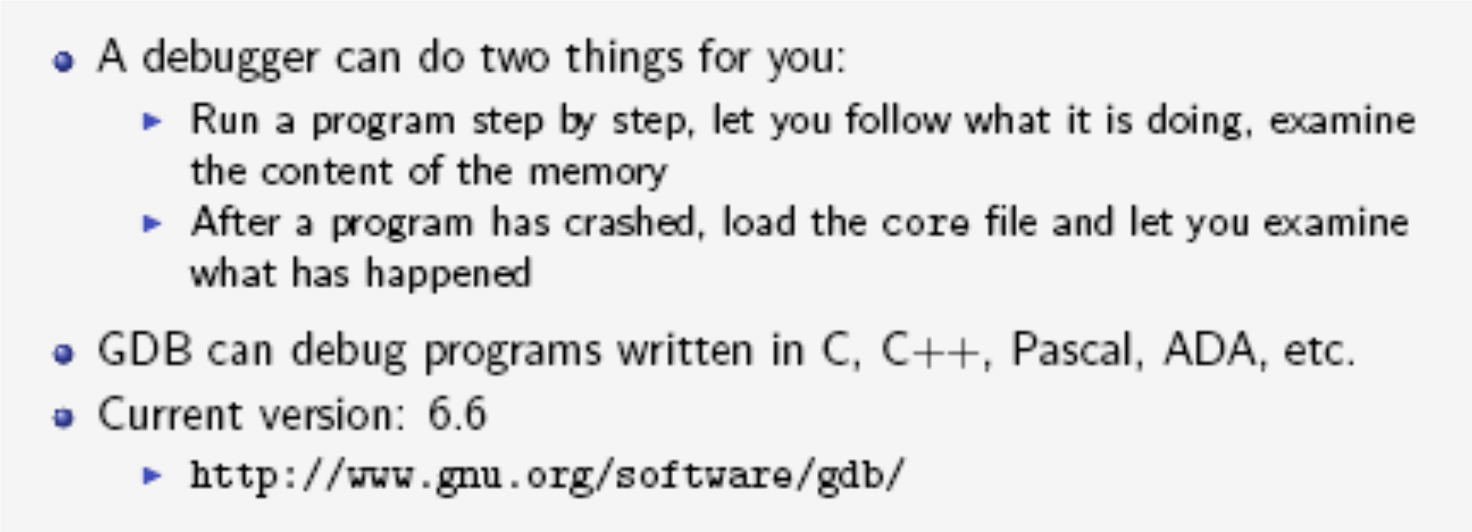
Crating a Dynamic Library



- This is slightly different from creating a static library:
  1. Compile each `.c` file using option `-fPIC`
  2. Generate the shared library using `gcc -shared`
  3. You must specify the name of the library (`libsomething.so`)

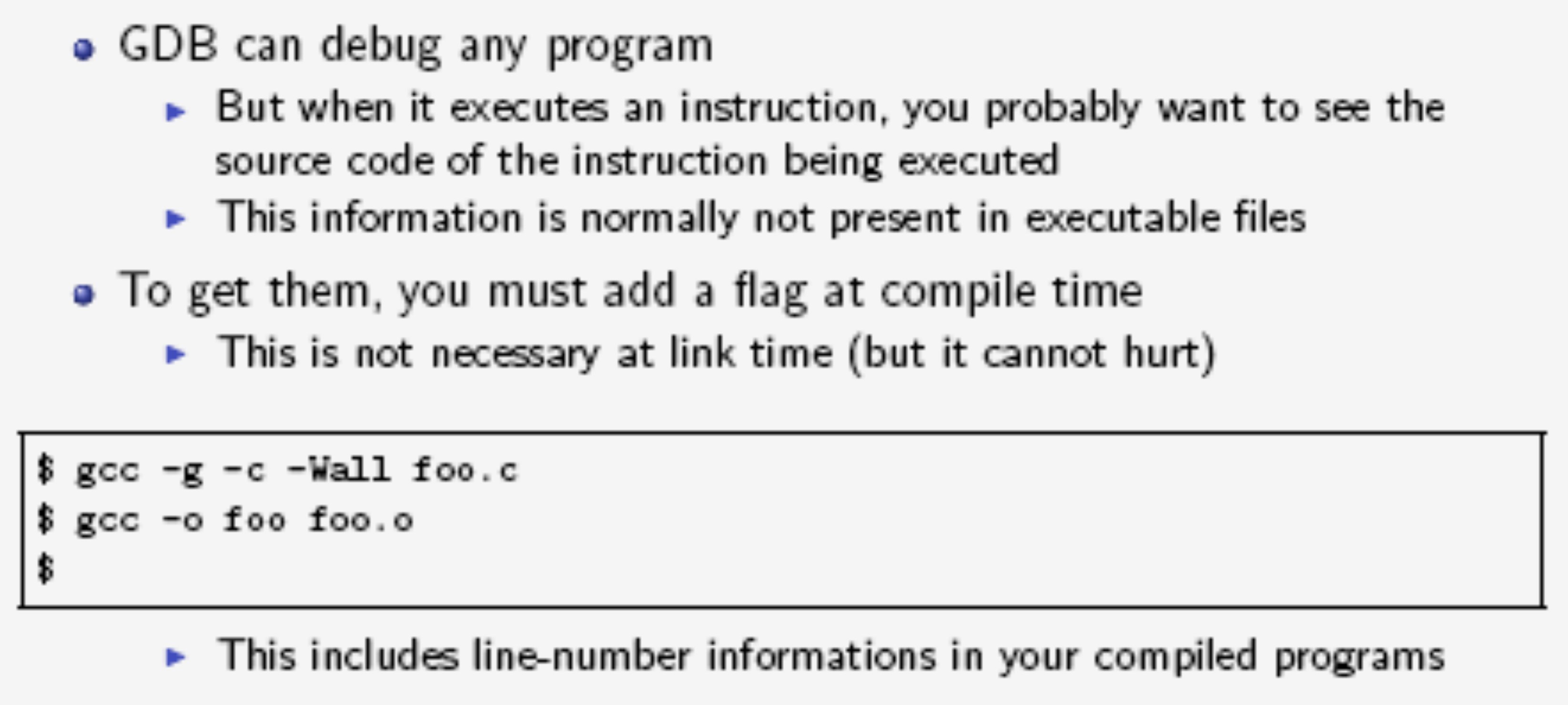
```
$ gcc -c -fPIC message.c
$ gcc -c -fPIC network.c
$ gcc -shared -o libnetwork.so message.o network.o
$
```

GDB: The GNU Debugger



- A debugger can do two things for you:
  - Run a program step by step, let you follow what it is doing, examine the content of the memory
  - After a program has crashed, load the core file and let you examine what has happened
- GDB can debug programs written in C, C++, Pascal, ADA, etc.
- Current version: 6.6
  - `http://www.gnu.org/software/gdb/`

Compiling with Debugging Info

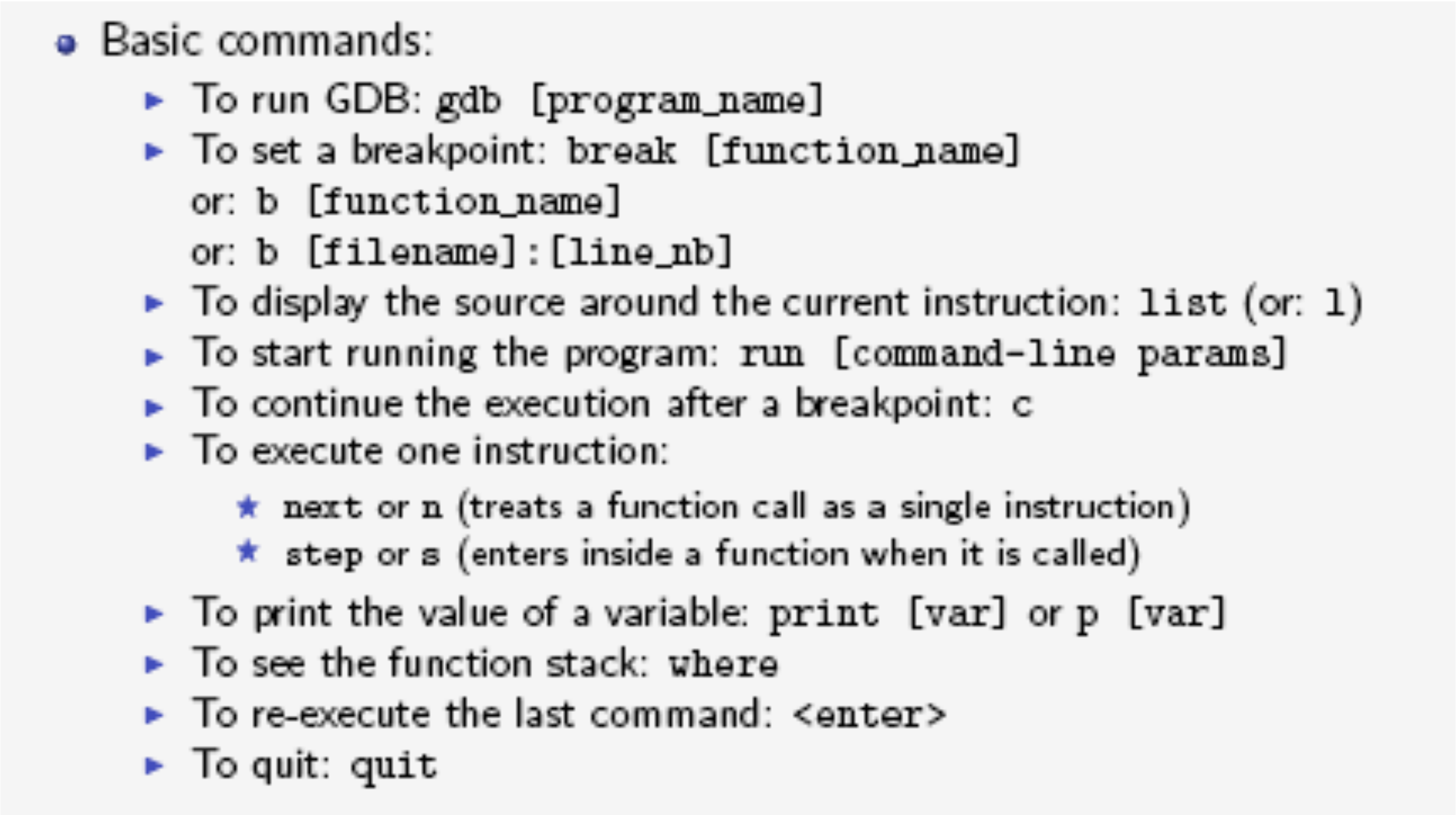


- GDB can debug any program
  - But when it executes an instruction, you probably want to see the source code of the instruction being executed
  - This information is normally not present in executable files
- To get them, you must add a flag at compile time
  - This is not necessary at link time (but it cannot hurt)

```
$ gcc -g -c -Wall foo.c
$ gcc -o foo foo.o
$
```

  - This includes line-number informations in your compiled programs

GDB Basic Commands



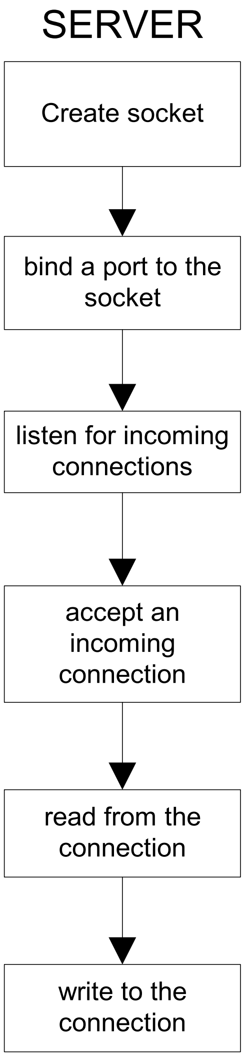
- Basic commands:
  - To run GDB: `gdb [program_name]`
  - To set a breakpoint: `break [function_name]`
    or: `b [function_name]`
    or: `b [filename]:[line_nb]`
  - To display the source around the current instruction: `list` (or: `l`)
  - To start running the program: `run [command-line params]`
  - To continue the execution after a breakpoint: `c`
  - To execute one instruction:
    - `next` or `n` (treats a function call as a single instruction)
    - `step` or `s` (enters inside a function when it is called)
  - To print the value of a variable: `print [var]` or `p [var]`
  - To see the function stack: `where`
  - To re-execute the last command: `<enter>`
  - To quit: `quit`

Creating a Socket

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

- domain is one of the *Address Families* (AF_INET, AF_UNIX, etc.)
- type defines the communication protocol semantics, usually defines either:
  - SOCK_STREAM: connection-oriented stream (TCP)
  - SOCK_DGRAM: connectionless, unreliable (UDP)
- protocol specifies a particular protocol, just set this to 0 to accept the default (PF_INET, PF_UNIX) based on the domain

Server Side Socket Details



SERVER

Create socket

bind a port to the socket

listen for incoming connections

accept an incoming connection

read from the connection

write to the connection

```
int socket(int domain, int type, int protocol)
sockfd = socket(PF_INET, SOCK_STREAM, 0);
```

```
int bind(int sockfd, struct sockaddr *server_addr, socklen_t length)
bind(sockfd, &server, sizeof(server));
```

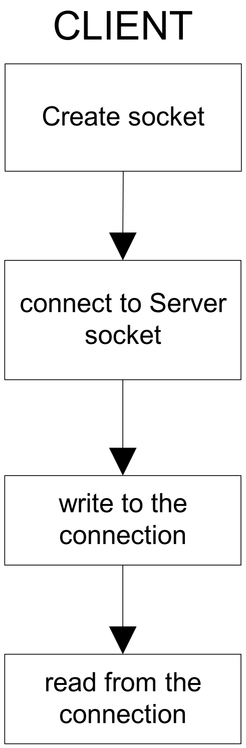
```
int listen( int sockfd, int num_queued_requests)
listen( sockfd, 5);
```

```
int accept(int sockfd, struct sockaddr *incoming_address, socklen_t length)
newfd = accept(sockfd, &client, sizeof(client)); /* BLOCKS */
```

```
int read(int sockfd, void * buffer, size_t buffer_size)
read(newfd, buffer, sizeof(buffer));
```

```
int write(int sockfd, void * buffer, size_t buffer_size)
write(newfd, buffer, sizeof(buffer));
```

Client Side Socket Details



```
int socket(int domain, int type, int protocol)
sockfd = socket(PF_INET, SOCK_STREAM, 0);
```

```
int connect(int sockfd, struct sockaddr *server_address, socklen_t length)
connect(sockfd, &server, sizeof(server));
```

```
int write(int sockfd, void * buffer, size_t buffer_size)
write(sockfd, buffer, sizeof(buffer));
```

```
int read(int sockfd, void * buffer, size_t buffer_size)
read(sockfd, buffer, sizeof(buffer));
```

Logic of a Web Server

- Setup the server
  - socket, bind, listen
- Accept a connection
  - accept, fdopen
- Read a request
  - fread
- Handle the request
  - 1. directory --> list it: opendir, readdir
  - 2. regular file --> cat the file: open, read
  - 3. .cgi file --> run it: exec
  - 4. not exist --> error message
- Send a reply
  - fwrite

An HTTP Request

- <command> <argument> <HTTP version>
- <optional arguments>
- <blank line>

- GET /index.html HTTP/1.0

Server Response

- <HTTP version> <status code> <status message>
- <aditional information>
- <a blank line>
- <content>

-
  ```
  HTTP/1.1 200 OK
  Date: Thu, 06 Nov 2008 18:27:13 GMT
  Server: Apache

  <HTML><HEAD><BODY> ....
  ```

Daemon Characteristics

Commonly, dæmon processes are created to offer a specific service.

Dæmon processes usually

- live for a long time
- are started at boot time
- terminate only during shutdown
- have no controlling terminal

The previously listed characteristics have certain implications:

- do one thing, and one thing only
- no (or only limited) user-interaction possible
- consider current working directory
- how to create (debugging) output

Concurrent Programming

- Implementation of concurrent tasks:
  - as separate programs
  - as a set of processes or threads created by a single program

- Execution of concurrent tasks:
  - on a single processor

  ➔ Multithreaded programming
  - on several processors in close proximity

  ➔ Parallel computing
  - on several processors distributed across a network

  ➔ Distributed computing

Communication Between Tasks

Interaction or communication between concurrent tasks can done via:

- Shared memory:
  - all tasks has access to the same physical memory
  - they can communicate by altering the contents of shared memory
- Message passing:
  - no common/shared physical memory
  - tasks communicate by exchanging messages

Threads vs Processes

- Heavyweight Process = Process
- Lightweight Process = Thread

Advantages (Thread vs. Process):

- Much quicker to create a thread than a process
- Much quicker to switch between threads than to switch between processes
- Threads share data easily

Disadvantages (Thread vs. Process):

- Processes are more flexible
  - They don’t have to run on the same processor
- No security between threads: One thread can stomp on another thread's data
- For threads which are supported by user thread package instead of the kernel:
  - If one thread blocks, all threads in task block.

Mutual Exclusion

- **pthread_mutex_lock**

```
// blocks until mutex is available, and then locks it
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

pthread_mutex_unlock

```
// unlocks the mutex
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Thread Example

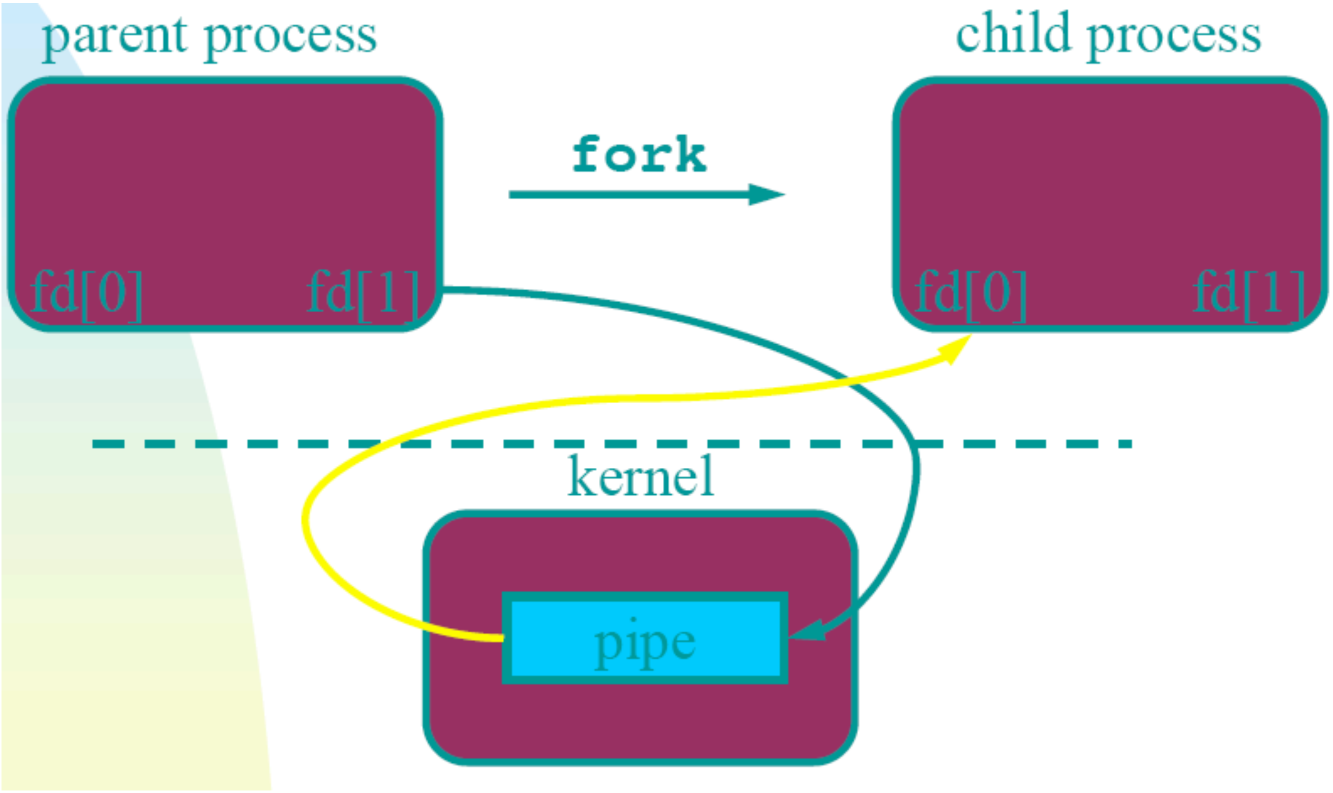
```
    int main()
{
    pthread_t thread1, thread2;  /* thread variables */


    pthread_create (&thread1, NULL, (void *) &print_message_function,
                                                            (void*)”hello “);
    pthread_create (&thread2, NULL, (void *) &print_message_function,
                                                            (void*)”world!\n”);


    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    exit(0);
}
```

Why use pthread_join?

To force main block to wait for both threads to terminate, before it exits. If main block exits, both threads exit, even if the threads have not finished their work.

Interprocess Communication

Using Pipes:



FIFOs: Named Pipes

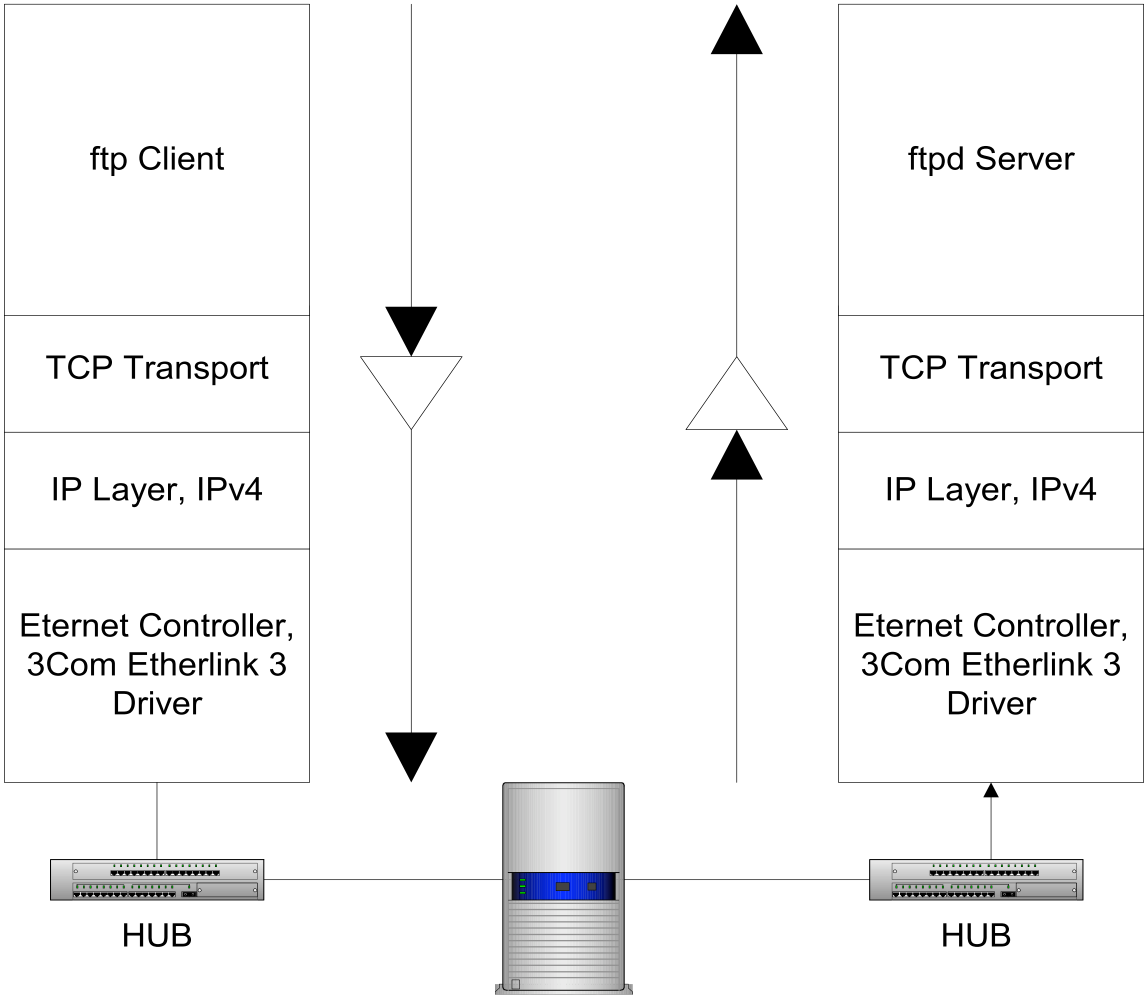
- FIFOs are “named” in the sense that they have a name in the filesystem
- This common name is used by two separate processes to communicate over a pipe
- The command mknod can be used to create a FIFO:

```
mkfifo MYFIFO (or “mknod MYFIFO p”)
ls –l
echo “hello world” >MYFIFO &
ls –l
cat <MYFIFO
```

Message Queues

- A Message Queue is a linked list of message structures stored inside the kernel's memory space and accessible by multiple processes
- Synchronization is provided automatically by the kernel
- New messages are added at the end of the queue
- Each message structure has a long *message type*
- Messages may be obtained from the queue either in a FIFO manner (default) or by requesting a specific *type* of message (based on *message type*)

Protocol Communication



ftp Client

TCP Transport

IP Layer, IPv4

Eternet Controller, 3Com Etherlink 3 Driver

ftpd Server

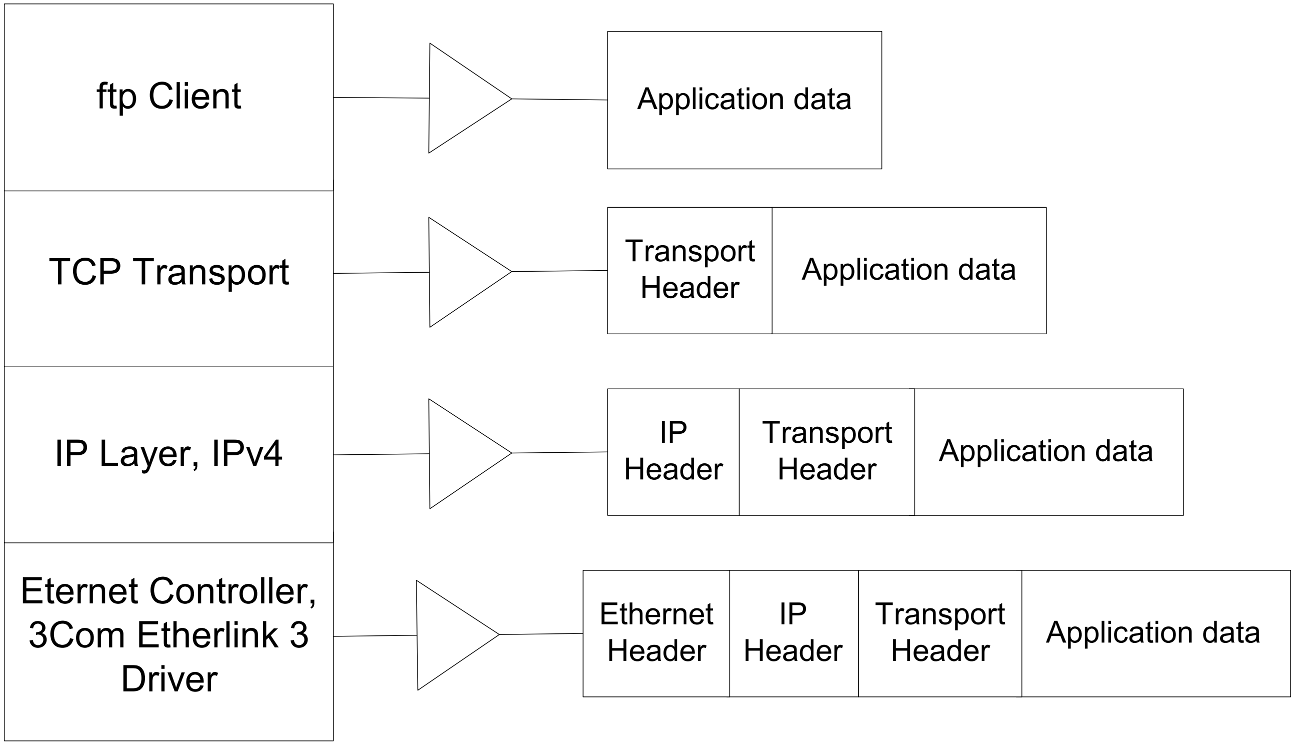
TCP Transport

IP Layer, IPv4

Eternet Controller, 3Com Etherlink 3 Driver

Data Encapsulation

- Application puts data out through a socket
- Each successive layer wraps the received data with its own header:



The Hardware (Ethernet) Layer

- Responsible for transferring frames (units of data) between machines on the same physical network

Preamble (bit sequence)	Destination Address (192.32.65.1)	Source Address (192.32.63.5)	Packet type (magic number for protocol: 0x800 = IP, 0x6003 = Decnet, 0x809B = Appletalk)	Datagram (THE DATA) (up to 12k bits)	Cyclic Redundancy Check
◄—64 bits—►	◄—48 bits—►	◄—48 bits—►	◄—16 bits—►	◄—variable—►	◄—32 bits—►

The Transport Layer

- Unix has two common transports
  - User Datagram Protocol (UDP)
    - record protocol
    - connectionless, broadcast
    - *Metaphor*: Postal Service
  - Transmission Control Protocol (TCP)
    - byte stream protocol
    - direct connection-oriented

Transport Layer: UDP

- Connectionless, in that no long term connection exists between the client and server. A connection exists only long enough to deliver *a single packet* and then the connection is severed.
- No guaranteed delivery ("best effort")
- Fixed size boundaries, sent as a single "fire and forget message". Think *announcement*.
- No built-in acknowledgement of receipt

Transport Layer: UDP

- No built-in order of delivery, random delivery
- Unreliable, since there is no acknowledgement of receipt, there is no way to know to resend a lost packet
- Does provide checksum to guarantee integrity of packet data
- *Fast* and Efficient

Transport Layer: TCP

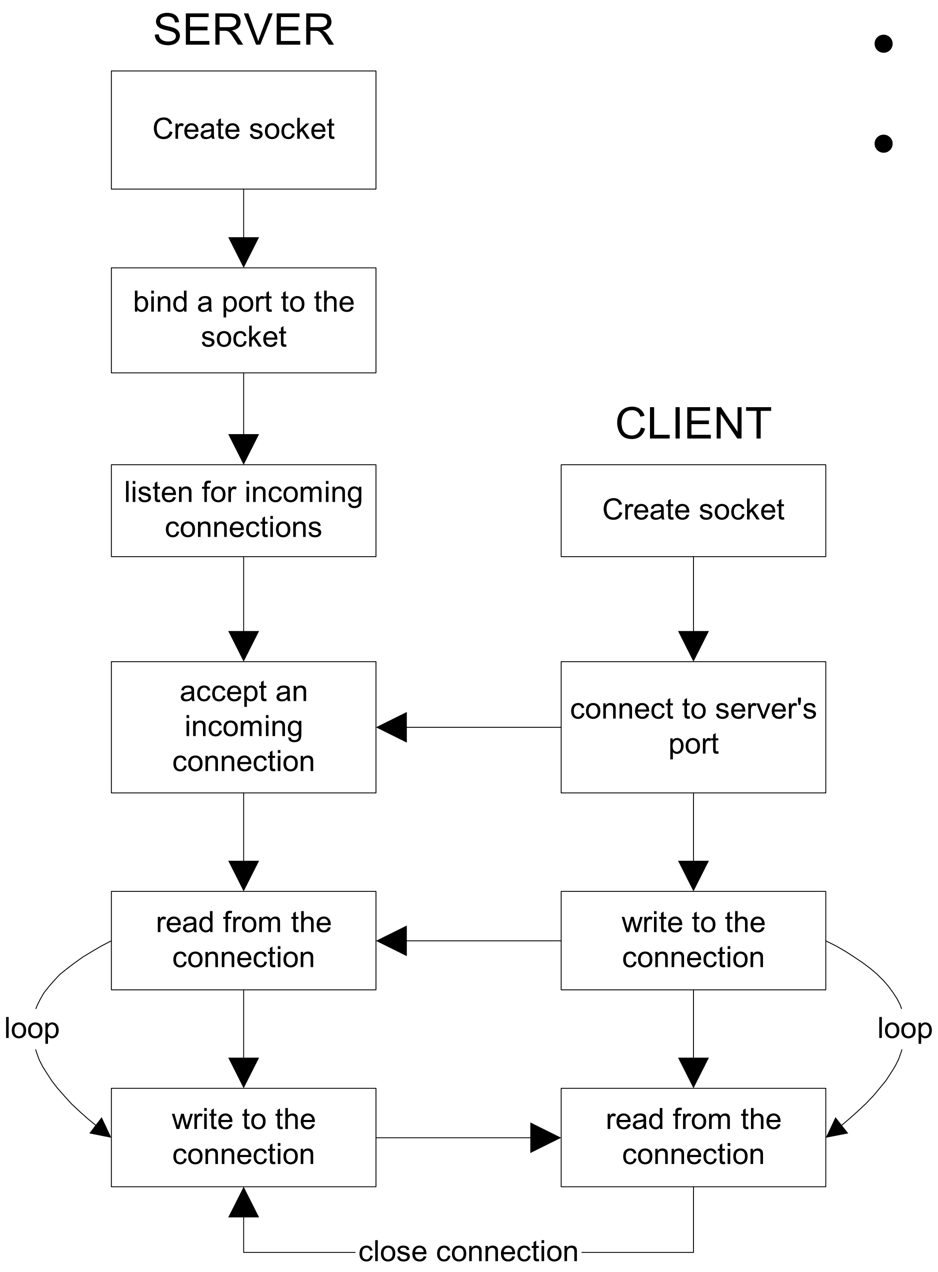
- TCP *guarantees delivery* of packets in order of transmission by offering acknowledgement and retransmission: it will automatically resend after a certain time if it does not receive an ACK
- TCP promises *sequenced delivery* to the application layer, by adding a sequence number to every packet. Packets are reordered by the receiving TCP layer before handing off to the application layer. This also aides in handling "duplicate" packets.

Transport Layer: TCP

- Pure stream-oriented connection, it does not care about message boundaries
- A TCP connection is full duplex (bidirectional), so the same socket can be read and written to (cf. half duplex pipes)
- Provides a checksum that guarantees packet integrity

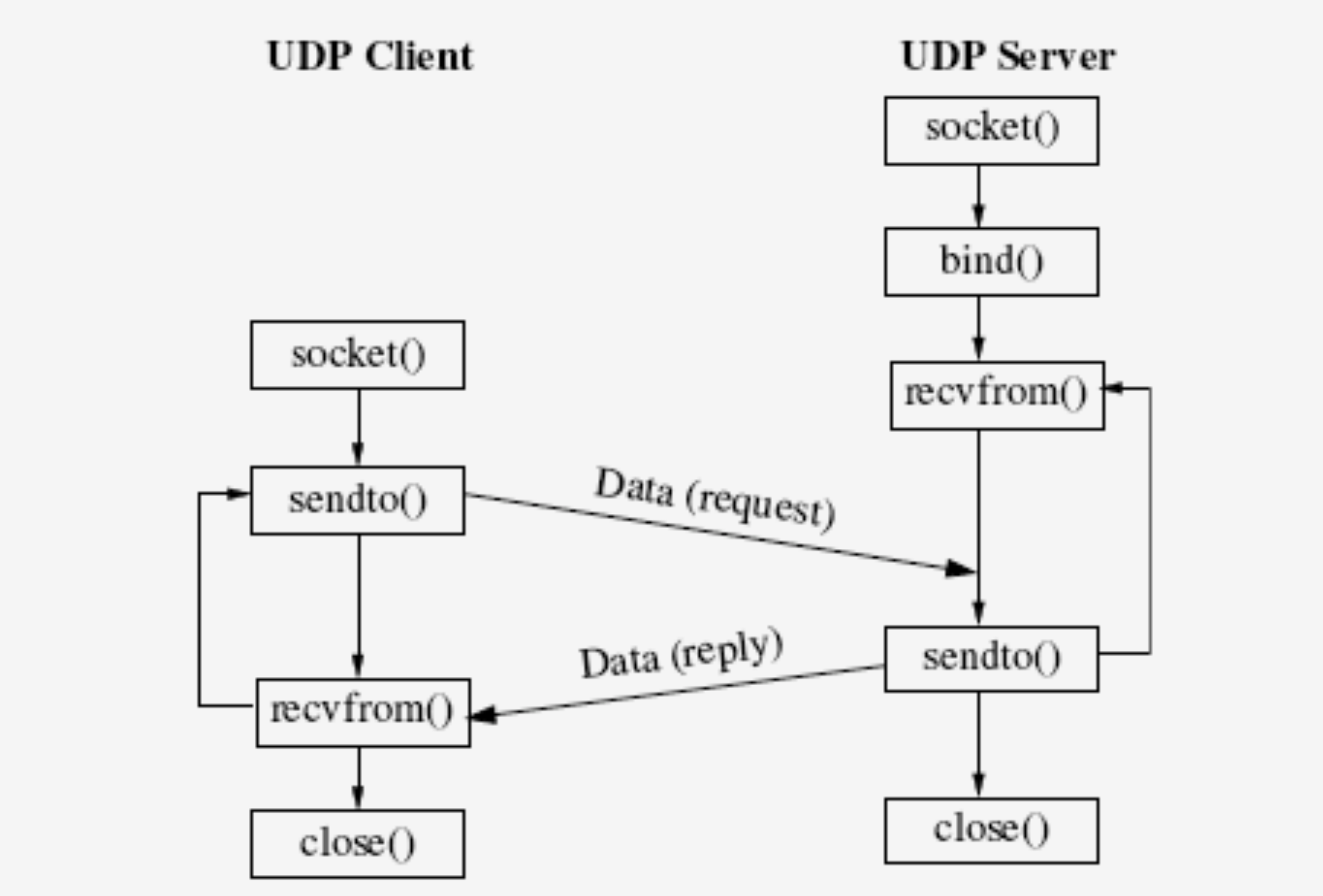
UDP Clients and Servers

- Connectionless clients and servers create a socket using SOCK_DGRAM instead of SOCK_STREAM
- Connectionless servers do not call listen() or accept(), and *usually* do not call connect()
- Since connectionless communications lack a sustained connection, several methods are available that allow you to *specify a destination address with every call*:
  – sendto(sock, buffer, buflen, flags, to_addr, tolen);
  – recvfrom(sock, buffer, buflen, flags, from_addr, fromlen);
- *Examples: daytimeclient.c, mytalkserver.c, mytalkclient.c*

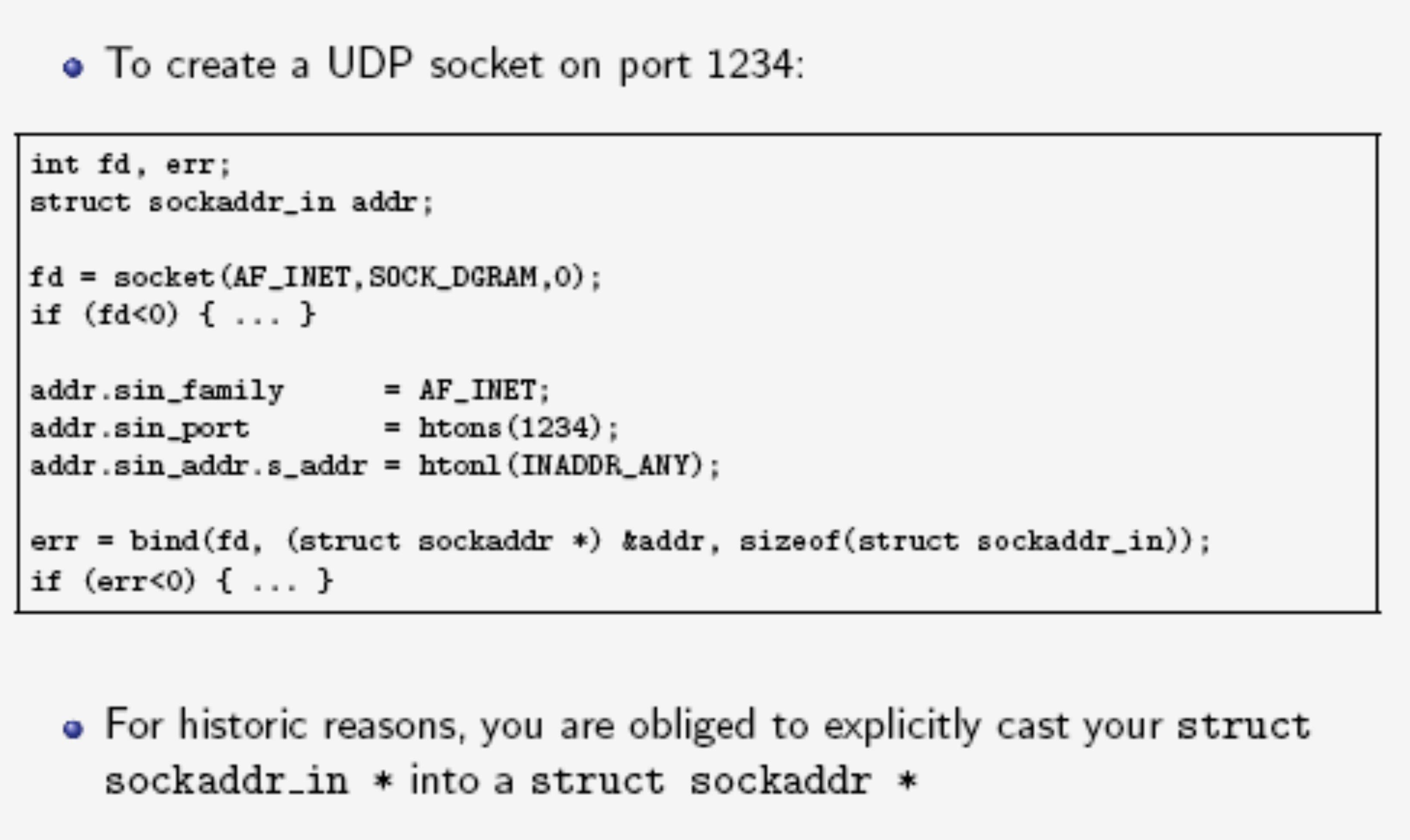


- TCP Client-Server view
- Connection-oriented socket connections

UDP Socket Functions



Creating UDP Sockets



- To create a UDP socket on port 1234:

```
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 *`

Exercise-I

Threads (True or False Questions):

- A thread cannot see the variables on another thread's stack.
- ***False*** -- *they can since they share memory*

- In a non-preemptive thread system, you do not have to worry about race conditions.
- ***False*** -- *as threads block and unblock, they may do so in unspecified orders, so you can still have race race conditions.*

- A thread may only call **pthread_join()** on threads which it has created with **pthread_create()**
- ***False*** **--** *Any thread can join with any other*

- With mutexes, you may have a thread execute instructions atomically with respect to other threads that lock the mutex.
- ***True*** -- *That's most often how mutexes are used.*

Exercise-I *(cont)*

Threads (True or False Questions):

- pthread_create() always returns to the caller
- *True.*

- pthread_mutex_lock() never returns
- *False -- It may block, but it when it unblocks, it will return.*

- pthread_exit() returns if there is no error
- *False -- never returns.*

Exercise - II

Pipes:

Let a process A invoke system("date | more"), where date and more are two programs. Is process A the parent process of the process running date?

No! Process A is the parent process a shell process, and the shell process is the parent process of the process running date.

Is the process running date the parent process of the process running more? You must provide explanation!

No! They are sibling processes. In fact, the shell process is the parent process of both of the processes.

Exercise - III

Processes:

Please provide two reasons on why an invocation to fork() might fail

(1) too many processes in the system (2) the total number of processes for the real uid exceeds the limit (3) too many PID in the system (4) memory exceeds the limit,

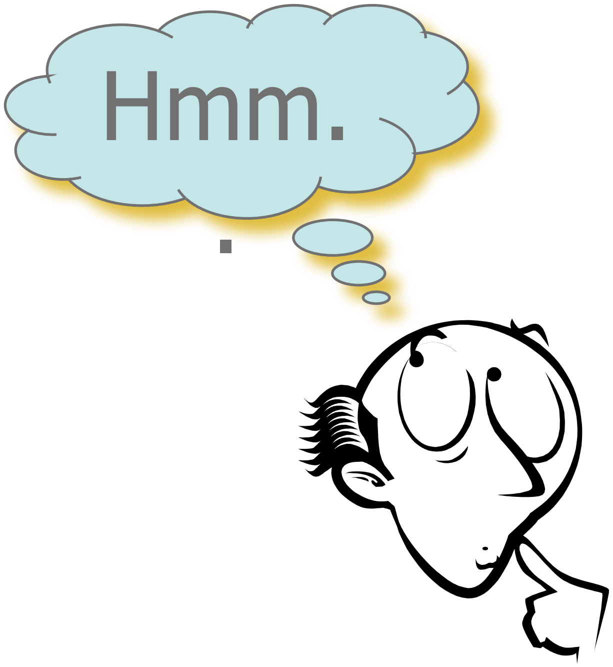
When a process terminates, what would be the PID of its child processes? Why?

It would become 1. Because when any of the child processes terminate, init would be informed and fetch termination status of the process so that the system is not cogged by zombie processes.

Final Announcements

- Final Exam:
  - Dec. 12th, @7:30am-9:30am
  - 19 Allen Hall

- Project 3:
  - Due Dec. 7th @11.59pm



Acknowledgments

- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), B. Knicki (WPI), M. Shacklette (UChicago), J.Kim (KAIST), and J. Schaumann (SIT).