Lecture 10

Network Programming

Principles

1. Get a socket id (int) with the `socket()` call

```
include <sys/socket.h>
int socket(int domain,      // AF_INET, AF_INET6, AF_UNSPEC
           int type,        // SOCK_STREAM, SOCK_DGRAM
           int protocol);   // 0 - IP
```

Some of this information is usually specified in a special structure `addrinfo` for network addresses.

```
#include <netdb.h>
struct addrinfo {
  int ai_flags;     // AI_PASSIVE
  int ai_family;    // AF_SPEC
  int ai_socktype;  // SOCK_STREAM
  int ai_protocol;  // 0
  // Set everything else to 0
  socklen_t        ai_addrlen;
  char             *ai_canonname;
  struct sockaddr *ai_addr;
  struct addrinfo *ai_next;
}
```

Optional - Specify socket behavior

If your calls will block until something comes on the network (the default), or return an error if there is no message.

```
include <sys/socket.h>
int socket(int domain,
           int type, //SOCK_STREAM | SOCK_NONBLOCK
           int protocol);
```

2. Establish a connection using the `connect()` call

```
include <sys/socket.h>
int connect(int socket,
            const struct sockaddr *address,
            socklen_t address_len);
```

The call needs a complicated structure with a lot of information you don't know.

This information is retrieved by the system from other servers or local files, based on what you provide.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
int getaddrinfo(const char *hostname,
                // name or IP address
                const char *servname,
                // port (as string) or name
                const struct addrinfo *hints,
                // what we know (not much)
                struct addrinfo **res)
                // list found by the system
                ;
void freeaddrinfo(struct addrinfo *ai);
```

3. Call `send()` to send a message

4. If an answer is expected, call `recv()` or `read()` to wait for it

5. Call `close()` to end the connection

What about writing a server?

```
connect()
```

```
bind()
listen()
accpet()
```

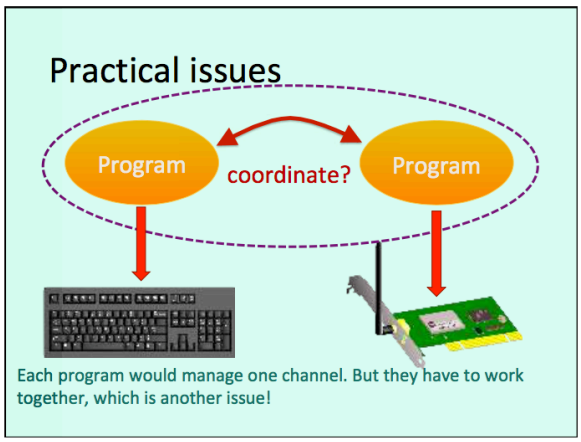
Turning it to C++

TCP/IP Network Programming Design Pattens in C++

- `TCPStream`
- `TCPConnector`
- `TCPAcceptor` (for writing a server)

Use `send()` and `recv()` rather than `write()` and `read()` as suggested in the post.

There are some errors in the test application (confuses `NULL` and `\0`).



Multiple Threads

Hyper Text Transfer Protocol (HTTP)

A HTTP server is a program that waits for requests formatted in a special way and listens on port 80 (all browsers know that).

HTTP is a "high-level" protocol, messages that must have a certain format and are sent over TCP/IP.

System Calls and Processes

Every process is identified by a process id (`pid`).

A process has easy access to two pids:

- Its own
- Its parent's

```
#include <unistd.h>
pid_t getpid(void);   // pid of the current
pid_t getppid(void);  // pid of the parent
```

Every process except process 1 is created by another process!

`ps -o pid,ppid,time,comm`

```
#include <stdlib.h>
// Create a subprocess that runs the command
int system(const char *command);
// Waits for command completion
// Returns the return code of the command
```

Signals

```
#include <signal.h>
int kill(pid_t pid, int sig);
/*
  pid
    > 0   specific process
    0     processes in the same group
```

```
    -l    other processes of the same user
*/
```

`kill -l` lists all available signals.

Special signal

`0` = test if process is alive

There may be some slight differences between systems

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGEMT       8) SIGFPE
 9) SIGKILL     10) SIGBUS      11) SIGSEGV     12) SIGSYS
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGURG
17) SIGSTOP     18) SIGTSTP     19) SIGCONT     20) SIGCHLD
21) SIGTTIN     22) SIGTTOU     23) SIGIO       24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGINFO     30) SIGUSR1     31) SIGUSR2
$
```

The "`kill -l`" command lists all available signals.
The default behavior when a program receives a signal depends on the signal.

signal `0` is a dummy signal which isn't delivered to the "target", but allows the sender to know whether the process corresponding to the pid is up and running.

Signal handler

`void handler(int sig)`

In some systems, handlers are automatically deactivated after being called.
Must reset themselves.

`SIGKILL` and `SIGSTOP` cannot be caught or ignored

```
#include <signal.h>
sig_t signal(int sig,
             sig_t func);  // SIG_IGN - Ignore,
                           // SIG_DFL Default behavior
// Returns previous settings
```

The hard way to trap a signal is calling the `sigaction()` function which, also declared in the same `signal.h` as `signal()`, is documented in section 2 (system calls) of the manual.

`sigaction()` allows for finer handling of signals than `signal()` can. For instance, nothing prevents with `signal()` a handler from being itself interrupted while processing a signal. Function `sigaction()` allows to mask interrupts and work uninterrupted when needed, and so forth.

Return from handler:

Resume but not always exactly where interrupted (eg system calls)

possible to return to a given instruction (`setjmp()` / `longjmp()`)

You will notice with the beeper that a signal "awakens" it, that is that it returns immediately from the call to `sleep()` even if it hasn't slept for 10 seconds yet. This is a fairly common behavior with waiting system calls (`recv()` springs to mind, it may be interrupted before it receives anything, it can happen with several I/O related functions).
Actually, `sleep()` returns the number of untaken seconds of sleep, and you can write the inside of the loop in this fashion to keep the beat:

```
if (remains == 0) {
     now = time(NULL);
     t = localtime(&now);
     printf("%02d:%02d:%02d beep!\n",
            t->tm_hour, t->tm_min, t->tm_sec);
     remains = sleep(nap);
} else {
     remains = sleep(remains - 1);
}
```

"Orderly termination"
"Clean shutdown"

- flush buffers and close files
- possibly delete temporary work files
- release resources

If a program suddenly quits, it may leave behind corrupted files, because by default everything isn't written instantly to files. Catching the signal allows you to close files before leaving. You may also want to remove some temporary work files, or release system resources, or close a network connection. Most "server type" programs try to put things in order before stopping.

beeper.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <time.h>

#define SLEEP_TIME  5

int         nap = SLEEP_TIME;

void handler(int sig) {
   nap /= 2;
   if (nap < 1) {
     nap = 1;
   }
}

int main() {
    time_t      now;
    struct tm *t;
    int         remains = 0;

    signal(SIGINT, handler);
    signal(SIGTERM, handler);
    while (1) {
       if (remains == 0) {
          now = time(NULL);
          t = localtime(&now);
          printf("%02d:%02d:%02d beep!\n",
                 t->tm_hour, t->tm_min, t->tm_sec);
          remains = sleep(nap);
       } else {
          remains = sleep(remains);
       }
    }
    return 0;
}
```

How to get the process id of an unrelated process?

`popen()` (pipe open) allows you to open a command as if it were a file, and to directly read from its standard output.

To check a single program

```
ps -e -o pid,command | grep prog | grep –v grep
```

Starting subprocesses

- `system()` serializes
- `popen()` strong link
- Simulations
- Parallel processing
- Daemon (session independent)

`fork()` = clone me

```
#include <unistd.h>
fork() // Takes no arguments, returns a pid_t
// -1 if error, 0 if child, child pid if parent
```

The forked process will be identical in every respect, except for pid and parent pid. It will run the same code, write to the same files and the same terminal.

SAME `stdin` / `cin`, `stdout` / `cout`, `stderr` / `cerr`

Child dies:

- parent receives `SIGCHLD`

Parent dies:

- process #1 becomes foster parent

A parent process is expected to wait for the completion of the child process.

The reason is that a process returns a status (the int return value) and that this status is supposed to be at least acknowledged by the parent process. As long as the status isn't acknowledged, the system cannot quite cleanup everything related to the completed process.

…