Architecture et Systèmes

Stefan Schwoon

Cours L3, 2014/15, ENS Cachan

Communication over network

Network Communication: Crash course

We shall briefly discuss an example how to establish a TCP/IP connection over the network.

Model: Client/server structure

- server establishes a services at a given port, waits for clients
- (multiple) clients can connect to the port and communicate with the server

Adressing in IP: ports

In the IP protocol, an Internet address consists of a *machine address* and a *port*.

A port can be thought of as fine-grained addressing:
each machine can have 2^{16} of them.

Ports 0..1023 reserved for special use, certain ports pre-defined.

E.g., 80 for HTTP protocol.

Server

The server makes the following steps (see example program):

1. Create a TCP socket (using `socket` call).

2. Connect the socket to an IP port (using `bind`).
Note: The port can be freely chosen, but numbers up to 1024 are usually reserved for system services. The client must know which port to connect to.

3. Switch the socket to *listen* mode (using `listen`).

4. Wait for a client to connect (using `accept`).
Note: accept returns a file descriptor used for exchanging data with that client.

Note that step 4 can be repeated to accept multiple clients. For each client, a separate file descriptor is created.

Client

The client makes the following steps:

1. Create a TCP socket (like the server).

2. Connect the socket to the correct port on the machine where the server is running (using `connect`).

Connect returns a file descriptor for exchanging data with the server.

Serving multiple clients

How can a server communicating with several clients at a time?

Problem: The server does not know in advance which client will send the next piece of data.

`read` waits patiently till the next piece of data arrives - but the server will block if nothing arrives from that client!

Solutions:

Create a child process (or thread) for each client.

Use the `select` system call to find a file descriptor where data is available.

Memory management

System calls

POSIX standard does not specify details of memory management; judged too machine-dependent.

Still, to handle the virtual memory of a process, Unix variants usually provide two functions:

`brk(2)` sets the size of the heap – in the virtual memory.
Direct interaction with the system, not normally used by the programmer.

`malloc(3)` (and similar routines) – process-level code that finds free space in the currently allocated heap space, will call `brk` if necessary.

Note: The OS only cares about the amount of memory requested by `brk`, which typically happens in large chunks.

How that requested memory is organized, is up to the user. Typically, `malloc` and co take over this organization and take care of fine-grained allocations of up to a few bytes. This is “invisible” on the system level.

We regard how memory allocators work.

Memory management: Malloc

A memory allocator manages the dynamic allocation of memory in the heap.

- main operations: malloc (allocate *n* bytes), returns address where memory can be used.
- user-level code contained in the standard library
- uses brk to increase virtual memory when necessary

Other operations:

- calloc (allocate and initialize to zero)
- realloc (change size of allocation, moving if necessary)
- free (release memory at given address)
- mmap (map file contents into memory)

Conflicting goals:

- minimize time (operations should be fast)
- minimize space (fragmentation, overhead)
- maximize locality (improve cache performance)
- work well in all use cases

Error detection (catching problems caused by incorrect usage):

- very limited
- use memory beyond allocated limits ($\rightarrow$ crash)
- double free ($\rightarrow$ crash)
- crashes may occur thousands of lines later $\rightarrow$ nightmare to debug

Memory blocks

Memory allocators like `malloc` typically divide memory into chunks that are *free* or *occupied* (allocated).

Simple approach: fixed-size chunks with bitmask (0=free, 1=occupied). However, fixed-size chunks are impractical.

Better: chunks of variable size

- mark beginning and end of chunks with its size, user data in between
- enables quick jump to the next chunks
- neighbouring free chunks can be easily joined
- Caution: byte alignment must be respected (pointers to words must be multiple of 4 or 8, depending on the CPU)

Time/space considerations

How to find a free chunks (of a given minimal size *n*)?

- Walk all chunks...
- Walk the free chunks (requires storing additional pointers inside the free chunks, increases speed but also overhead)...

Which free chunks to take?

- The first chunks found with size $\geq n$ (first-fit).
- Similar, but continue searching where the last search stopped (next-fit).
- The minimal free chunks with size $\geq n$ (best-fit).

Multiple bins

Compromise used in malloc:

> Multiple bins for free blocks of fixed sizes, e.g. `16`, `24`, `32`, `...`
>
> When requesting *n* bytes, look at the smallest non-empty appropriate bin and take free block from there.

Next-fit: aims to preserve locality but is observed to lead to bad fragmentation; used only in specific cases.

Description of more implementation details:
`http://g.oswego.edu/dl/html/malloc.html`

Shared memory

Unix provides a shared-memory mechanism, i.e. processes may share a segment of their memory with other processes.

Two steps (see example program):

- Set up shared memory segment (using `shmget`).
  Interface similar to file system, *key* acts as identifier.

- Integrate the shared segment into virtual memory of the process (using `shmat`).