A little history: Unix

The Unix system comes from the Multics experience (multiprogrammed-computer system) and is written for fun by Bell-Labs people (AT&T), on an unused computer.

The main features of the system are:

- **Multiuser capability**
- **Multiprocess capability (as time sharing)**
- **Peripheral access made like file access**
- **Pipes and sockets (new concepts)**
- **The "controlling tty" (new concept)**

A little history: GNU

The GNU project was born in 1984 when Richard Stallman, then MIT employee, needed to go on sharing CS culture with his own mates, without being bound by restrictive licenses and unneeded secrets.

The main features of the system are:

- **Feature 0: it can be used for whatever purpose**
- **Feature 1: it can be studied and customized**
- **Feature 2: it can be copied with no royalties**
- **Feature 3: it can be modified and redistributed**

A little history: Linux

The Linux kernel comes from the Minix (mini-unix) experience and has been written for fun by Linus Torvalds on his 386 computer with 4MB of RAM memory, in order to learn how multitasking works and to overcome the Minix limitas.

The main features of the system are:

- **Unix compatibility at functional level**
- **Unix compatibility at source level**
- **GNU GPL license for kernel code**
- **Support for dozens of platforms (over time)**

Unix Core Ideas

Basic Concepts for Data

Every OS-managed data item is a file

- Thus, peripherals are files
- Directory are files
- Communication channels are files

The Unix file can be considered the basic data class

- The methods are open, close, read, write
- The subclasses are regular files, device, directories...

The file is nothing more than an array of bytes

- The name is irrelevant to the OS, just a convention
- The text encoding is irrelevant to the OS too

Most information is stored as plain text

- Redundancy helps disaster recovery
- People can do diagnostics by simple inspection
- We avoid the need for special format-specific tools

Basic Concepts for Code

Everything is a system call

- All the OS kernel offers, code-wise, is system calls
- A program is a linear sequence of instructions and system calls
- And a program is nothing more than a data file being run

The system call interface is very simple and low-level

- It relies on some hardware mechanism for privilege escalation
- All arguments are passed through registers
- Errors are standardized in a few dozens integer codes
- Most other items are integers as well: __NR_*, fd, uid, time, ...
- Complex parameter passing, if any, happens through pointers

Mechanism, not policy

- Any user-visible mapping of those integers is policy
- The kernel interface includes as little policy as possible
- All policy choices (and blame) are left to upper levels

The Filesystem

The file is just an array of bytes (did I say this?)

- Metadata is unrelated to the file's contents and lives elsewhere
- The fs includes an "inode" struct, for metadata (size, owner, on-disk placement)
- Inodes are enumerated: device+inum identifies a unique file
- For names, we have directories: just lists of names and inums
- Names are completely arbitrary strings of bytes
  - Not really: the separator and null-byte are forbidden

This overly-simple approach brings in interesting side effects

- You can rename or remove a file that is in use, and it works
- Files can have several names, and it works
- Files can have no name at all, and it works
- The filesystem ignores local-language encoding, and it works
- Names are case-sensitive (and anydifference-sensitive)

So, the filesystem is just a database, whose data items are files

- And system programs are the queries acting on it

Permission Rules

Three bits cover all authorization needs for a file:

- Read, Write, eXecute
- All combinations make sense

A process is nothing more than the execution of a file

- Machine code files are executed by the kernel
- Text files are executed by the shell
- Files can also name their own interpreter

Three permission levels cover all authorization requirements

- User, Group, Other users
- Each user can be part of several groups

Privilege escalation of a process is managed by extra bits

- Set-uid, set-gid
- A process executing the program changes its own identity

You'll also hear about more complications introduced later

- But this basic approach works for most needs

The Controlling Tty

There is no such thing as a "system keyboard" or "screen"

- A program interacts through files, and files only
- There exists no meaningless "kbhit" function

By convention, there exists standard input, output, error

- They can be connected to a keyboard, but they are likely not

Unfortunately, the user needs some control over interaction

- Keyboard-kill operations and automatic termination on modem hang-hup
- Changes in screen size need to be notified

Thus, each process may have (or not) a controlling tty

- The tty is a serial port, with uart-class features
  - Baud rate, parity, newline conversion, ...
- The tty is also a screen, with display-class features
  - Char-grid size and little more
  - Other screen-like things travel in the data stream as escape sequences
- Special input characters received from a tty make the kernel send signals
  - The signal is sent only to processes controlled by that tty

C Library

The system call interface is a little too low-level

- Actually, C language is too low-level as well

The system, thus, usually includes a standard C library

- It implements a function-call API for the hw-specific syscall mechanism
- It offers file I/O with buffering (it helps system calls)
- It offers string operations (it helps C language)
- It gives names to numbers, by accessing system files (it adds policy)

The system library is called "C" library but it isn't specific to C.

- Most programs, most languages, rely on libc for their work
- There are several implementations of the library, both big and small
- You can write your own C library, and it's fun
- There also exist programs that do not use libc, and it's fun

The POSIX standard covers most abstraction levels

- Both system calls and standard library, for example.

The system call mechanism

The kernel exists to service system calls

System calls are usually invisible to applications

- The standard C library masks syscalls as functions
- The Linux implementation is not directly posix-compliant

The calling convention for system calls is like this:

- Syscall arguments are stored in registers
  - This applies to archs where args are passed on the stack, too
- On return, small negative value (-4095..-1) represent -errno, otherwise success
- Libc wraps this to the Posix API (-1 on error, errno set accordingly)

To enter kernel space a special machine instruction is issued

- x86: "int 0x80" (or the newer "sysenter", from PIII onwards)
- ARM: "swi" a.k.a. "svc" (with different arguments in OABI and EABI)

System calls with many arguments use a different convention

- We'll ignore such system calls here

How to issue system calls without a C library

X86:

```
int sys_write(int fd, void *buf, int count)
{
    int ret;

    asm("int $0x80" : "=a" (ret) :
        "0" (__NR_write), "b" (fd), "c" (buf), "d" (count));
    return ret;
}
```

```
00000000 <sys_write>:
   0:   53                      push   %ebx
   1:   b8 04 00 00 00          mov    $0x4,%eax
   6:   8b 54 24 10             mov    0x10(%esp),%edx
   a:   8b 4c 24 0c             mov    0xc(%esp),%ecx
   e:   8b 5c 24 08             mov    0x8(%esp),%ebx
  12:   cd 80                   int    $0x80
  14:   5b                      pop    %ebx
  15:   c3                      ret
  16:   8d 76 00                lea    0x0(%esi),%esi
  19:   8d bc 27 00 00 00 00    lea    0x0(%edi),%edi
```

ARM (OABI):

```
#define __sys2(x) #x
#define __sys1(x) __sys2(x)
#define __syscall(name) __sys1(__NR_##name) ""

int sys_write(int fd, void *buf, int count)
{
    register long ret __asm__("r0");
    register long __r0 __asm__("r0") = (long)fd;
    register long __r1 __asm__("r1") = (long)buf;
    register long __r2 __asm__("r2") = (long)count;

    asm("swi " __syscall(write) : "=r" (ret):
        "0" (__r0), "r" (__r1), "r" (__r2));
    return ret;
}
```

```
00000000 <sys_write>:
   0:   ef900004        svc     0x00900004
   4:   e12fff1e        bx      lr
```

With EABI parameter passing is simplified

- Syscall number is passed in r7
- Arguments are r0 onward (unchanged)

Processes

The Abstract Idea of Process

A process is a virtual machine, relying on system calls

- It interacts trhough memory and system calls, nothing more

A process is a sequence of instructions

- Running within a virtual memory space
- Communicating through file descriptors
- Ready to handle signals
- Carrying its own env, cwd, root, limits, ...

A thread, in Linux, is just a process

- There's no technical reason to differentiate
- The illusion of "threads" is built towards user space (but TID exists)
- Sharing (or lack thereof) of state is handled by clone(2) bits

Every process is executing a binary file

- There is no exception possible: no filesystem means no process
- Kernel threads are an exception
  - Great idea, but out of scope now

A Process' Lifetime

A process is only created as a copy of an existing process

- fork(2) takes no arguments and returns integer
- It returns 0 in the child and the child's pid in the father
- Thus, the processes share files, signals and all the rest

The child, still running the parent's code, can change itself

- Any process can change itself, the new child is not special
- A typical child closes files, changes directory, etc

A process can execute a different file, through execve(2)

- The call never returns: the process is replaced completely
- All attributes (beside the memory image) remain unchanged

A process can wait for children to die, or get notified about it

- The system sends SIGCHLD to the parent, that can use it
- The parent can actively wait(2), in blocking or non-blocking mode

```
if (fork() == 0)
        execlp("sleep", "sleep", "1", NULL);
else
        wait(NULL);
exit(0);
```

-
-

Process Attributes

Each process has the following attributes (and more)

- Execution attributes
  - Current state (running, sleeping, ...)
  - Scheduling class and execution priority
  - Memory image

- Filesystem attributes
  - A root directory and its current directory
    - Actually, nowadays we have "namespaces"
  - A controlling terminal (tty)
  - Access credentials (uid, gid, ...)

- Environment variables

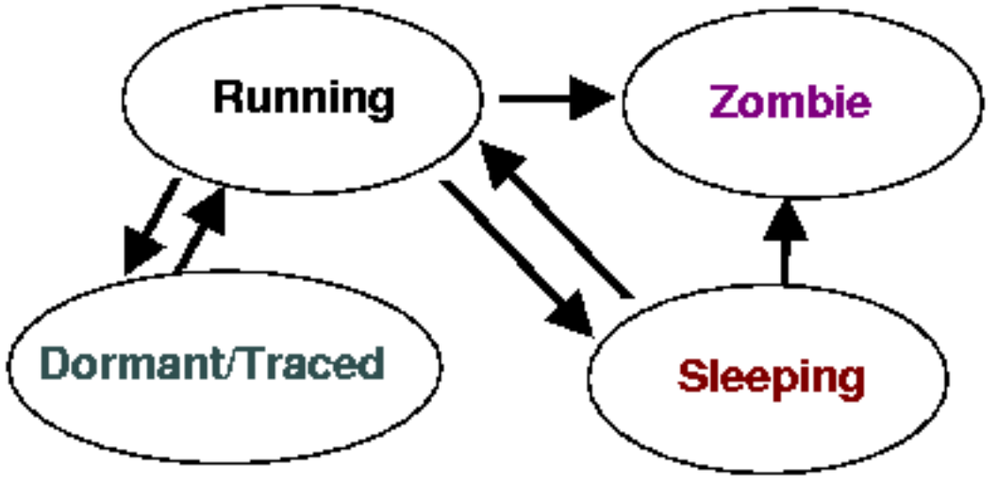
- Permission attributes
  - User and group (uid, gid)
  - Limits (see command "ulimit")
  - Unique identifiers (PID and TID)
  - It belongs to a process-group and a session

Process States

A process waiting for an event is said to be sleeping

- **In state S it is willingly waiting, can be signalled**
- **In state D it's forced to wait, can't handle signals**
  - **Because a page fault occorred**
  - **Because it's under debugger control**

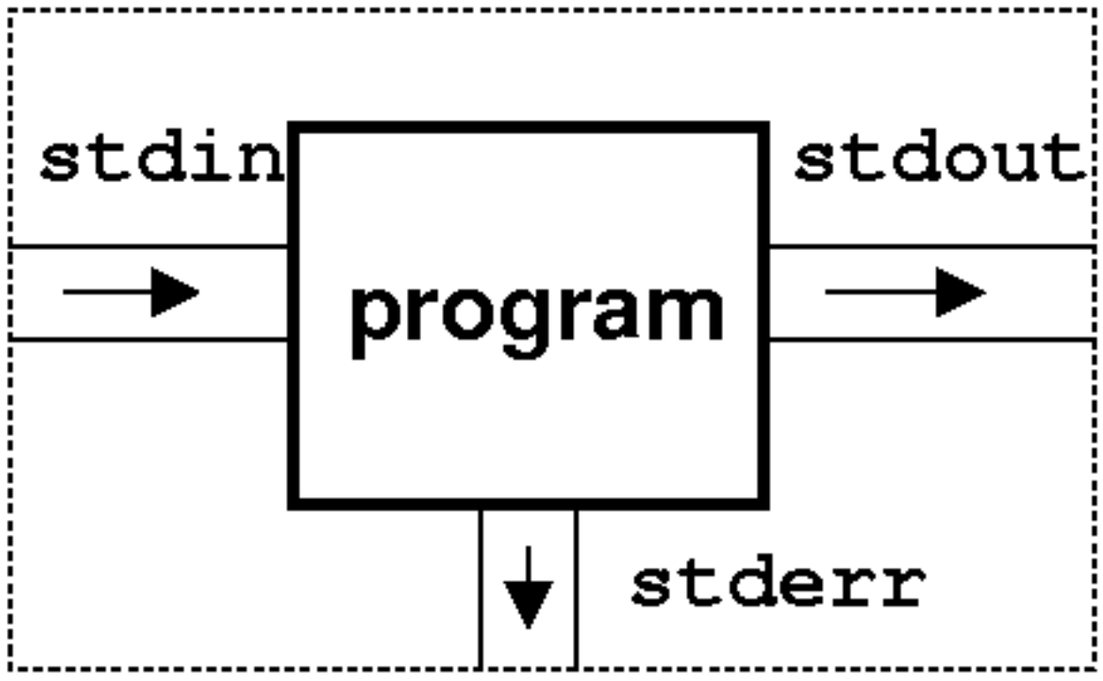
A process that exited remains in Z state until reaped



stdin, stdout, stderr

Every execution, usually, starts with 3 open files (channels)

- **It's a convention, it's not mandatory**
  - **every "interactive" program follows it**
  - **qmail chose to use a different convention**



Process Limits

- **A process can be limited in its resource use**
  - **getrlimit(2), getrusage(2) to query the system**
  - **setrlimit(2) to change the limits**
  - **the shell offers "ulimit" as internal command**

```
struct rusage {
        struct timeval ru_utime;    /* user time used */
        struct timeval ru_stime;    /* system time used */
        long ru_maxrss;             /* maximum resident set size */
        long ru_ixrss; /* integral shared memory size */
        long ru_idrss; /* integral unshared data size */
        long ru_isrss; /* integral unshared stack size */
        long ru_minflt;             /* page reclaims */
        long ru_majflt;             /* page faults */
        long ru_nswap;              /* swaps */
        long ru_inblock;            /* block input operations */
        long ru_oublock;            /* block output operations */
        long ru_msgsnd;             /* messages sent */
        long ru_msgrcv;             /* messages received */
        long ru_nsignals;           /* signals received */
        long ru_nvcsw;              /* voluntary context switches */
        long ru_nivcsw;             /* involuntary context switches */
};
```

Ptrace System Call

ptrace(2) is how a debugger can control a process

```
long ptrace(enum __ptrace_request request, pid_t pid,
            void *addr, void *data);
```

- PTRACE_TRACEME
- PTRACE_ATTACH
- PTRACE_KILL
- PTRACE_DETACH

- TRACE_PEEKTEXT, PTRACE_PEEKDATA
- PTRACE_PEEKUSER
- PTRACE_POKETEXT, PTRACE_POKEDATA
- PTRACE_POKEUSER
- PTRACE_GETREGS, PTRACE_GETFPREGS
- PTRACE_SETREGS, PTRACE_SETFPREGS
- PTRACE_CONT
- PTRACE_SYSCALL, PTRACE_SINGLESTEP

If you want, I can give you a minimal example using ptrace

Files

Access to Files

A "file descriptor" is a small integer number

- open(2), pipe(2), socket(2) create fds
- dup(2) and fork(2) clone them
- close(2) and exit(2) close them
- read(2), write(2), recvfrom(2), ...
- lseek(2), llseek(2) to move within the file
- access(2), stat(2), fstat(2) to get information
- errors are reported in the global variable errno

Information on open files

- "fuser" command
- "netstat -ap" command
- directory /proc/<pid>/fd

Blocking and Non-blocking Access

A system call can be blocking

- If you want to read data but there is no data yet
- If you want to write and there is no buffer space
- Blocking calls can return EINTR

read(2) and write(2) can transfer only part of the data

- read: if there is not enough data
- write: if the buffer is small but not full

With fcntl(2) you can set O_NONBLOCK

- read(2) will return EAGAIN instead of blocking

```
flags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

select and poll

select(2) and poll(2) allow waiting for events

- You state what set of descriptors you are interested in
- The caller is reported what descriptors are active
- You can set a timeout

Normally, main() is a loop based on select(2)

- A process must sleep as much as possible
- Asynchronous events are notified immediately
- Timeouts have the same granularity as the system clock

Since everything is a file, most situations can be attacked with a main loop based on select(2) or poll(2)

This predates multi-threading

- It may be way more efficient than multi-threading
  - Unless you really need concurrency on a multi-core system

Stdio

The standard I/O library simplifies the use of files

- fopen(3), fclose(3), fread(3), fwrite(3), fseek(3), ...
- popen(3) calls an external command
- fdopen(3) promotes an fd into a file pointer
- fscanf(3), fprintf(3), fgets(3), puts(3)

Obviously, stdio has disadvantages too:

- select(2) and poll(2) will just not work
- non-blocking files are a real pain
- scanf(3) can block unexpectedly

Buffering

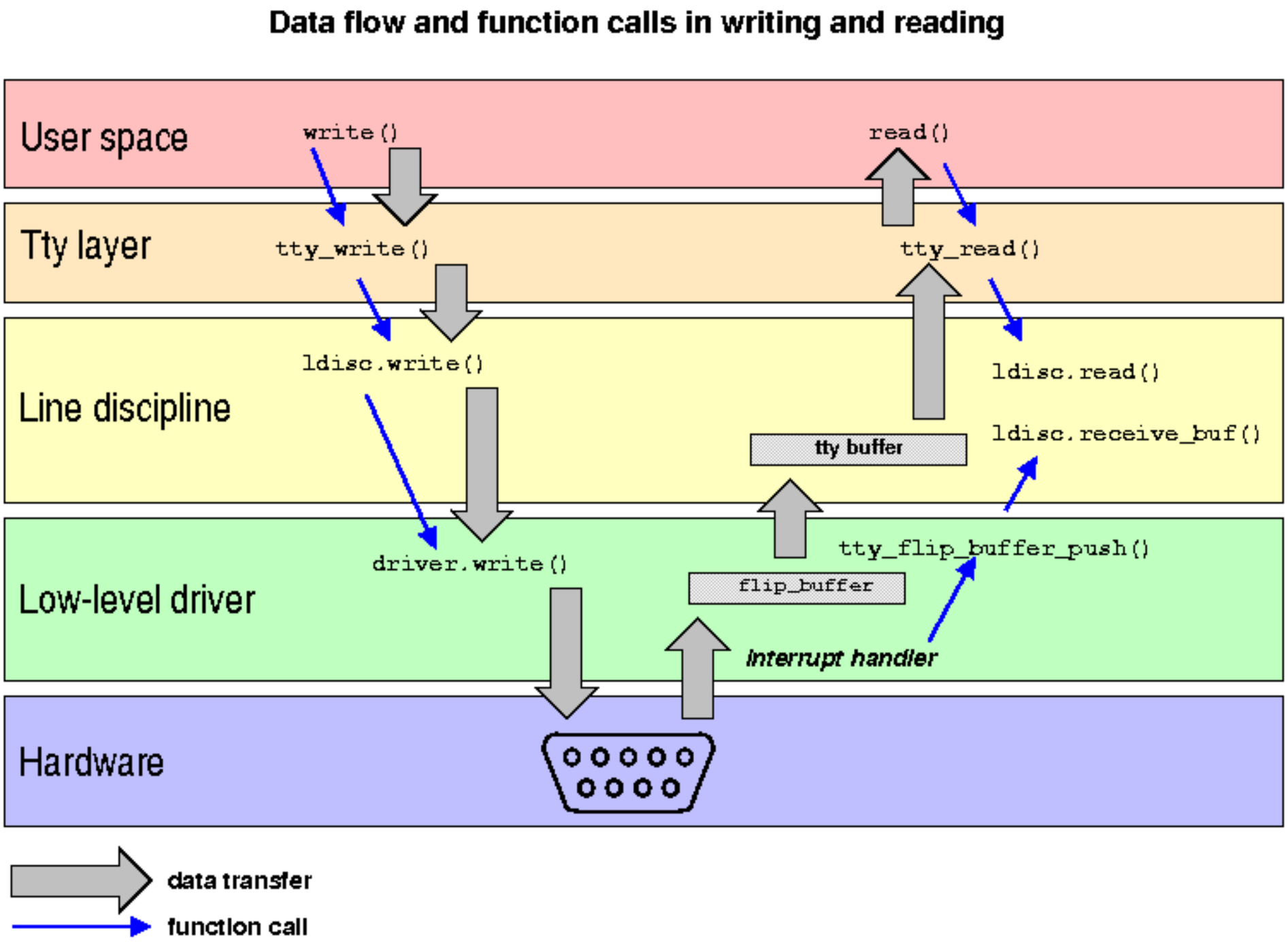
Many problems you find are related to buffering issues

- A file can be _IONBF, _IOLBF, _IOFBF
- With setvbuf(3) you can change the set up
- stdout is usually _IOLBF
- stderr is usually _IONBF
- if you open files with fopen(3), you get _IOFBF

We must also remember that a tty is special

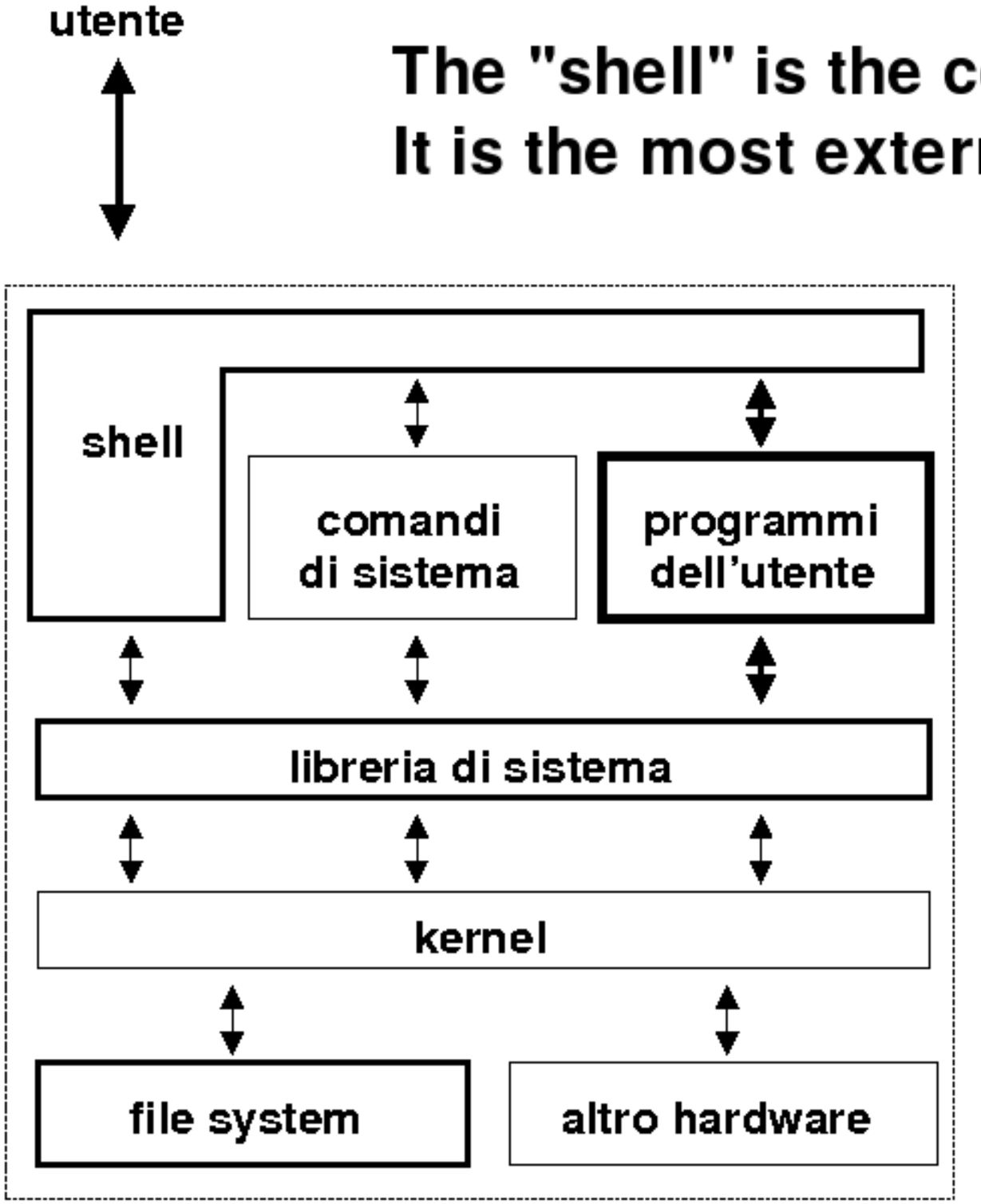
- The tty is managed by a line discipline
- Normally, it has a line buffer too
- tcgetattr(3) and tcsetattr(3) work on tty settings

Line Discipline



Shell Programming, quickly

The Command Interpreter



The "shell" is the command interpreter
It is the most external layer of a Unix system

```
% \ls -1 /bin/*sh
/bin/bash
/bin/csh
/bin/dash
/bin/fdflush
/bin/ksh
/bin/pdksh
/bin/rbash
/bin/sash
/bin/sh
/bin/tcsh
/bin/zsh
```

sh, csh, ksh, ...

There are various shells, and they only differ in details

- Syntax for uncommon operation
- Featureful/featureless in user interaction
- Executable size

The most important families are

- sh: the pristine one, the Bourne shell (by Steve Bourne)
- csh: the "C" shell, from Berkeley tradition
- ksh: Korn shell, by mr. Korn

In GNU/Linux systems we usually have these options

- bash (Bourne Again Shell)
- ash (sh-compatible, but much smaller than bash)
- dash (Debian Almquist Shell -- a modern ash)
- zsh (sh-compatible, not huge as bash nor minimal as dash)
- tcsh (csh compatible, quite out of fashion by now)

Input/Output Redirection

Most Unix commands are designed to benefit from redirection

All command interpreters allow redirection like this:

- `command < file`
- `command > file`
- `command >> file`

Redirecting stderr:

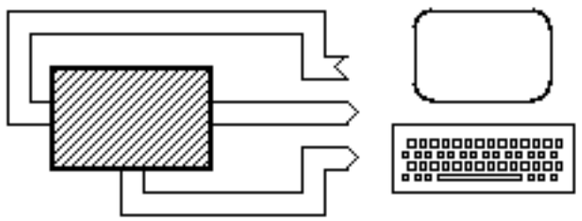
- `sh:` `command 2> file`
- `csh (stdout+err):` `command >& file`

Pretty often, redirection is all you need to perform your task

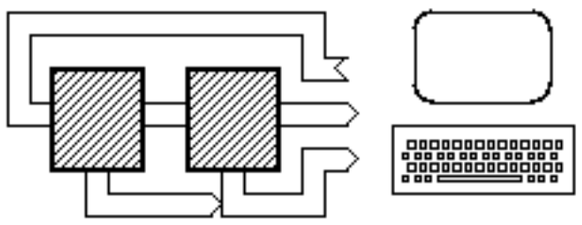
The Pipe

The Unix "pipe" is just plumbing, connecting two commands

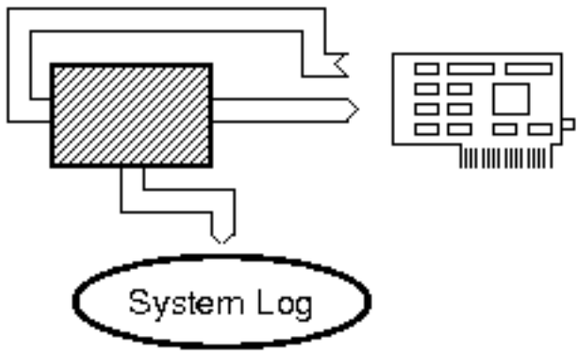
- The commands run concurrently, and results are produced immediately
- A pipe avoids the need to save intermediate results



Normal operation



Pipeline



Network Connection
through sockets

Pipe Examples

```
ls -l | sort -n -k 5

ls -l | sort -n -k 5 | tail -10

cat file.ps | psselect -p1-8 | \
        mpage -4 > small.ps

find . -name \*.h | xargs wc -l

command | tee file1 | grep str | \
        tee /dev/tty > file
```

Creating Pipes From C Language

The system call pipe() creates a plumbing pipe

- pipe(2) creates two descriptors, one WO and one RO
- There fd's are not connected to the filesystem, they only live in the process
- Thus, a pipe can only connect processes originating from the same grandparent

The C library also offsers popen(3)/pclose(3)

- They use the same argument conventions as fopen(3)/fclose(3)
- But you can't just fclose a file returned by popen
- Popen invokes a shell process, so a whole command line is fine
- The shell can be written-to or read-from
- The code is much simpler than running pipe/fork/exec/wait
- But there's less control over the details

Buffering and Blocking

The stdio library offers three different buffering policies

- _IOFBF: Full buffering (typically 4k o 16k)
- _IOLBF: Line buffering (up to the newline character)
- _IONBF: No buffering
- For details, see setvbuf(3)

Buffering interferes with pipes (and sockets, and poll/select)

- If stdin/out are not terminals, the default is _IOFBF
- So the writing process may not be actually write(2)ing
- The reading process can block waiting for data that doesn't arrive
- Interactive use, with grep(1), can be disappointing
- That's why many commands offer "-l" or force _IONBF

Example: using tee to replicate terminals

Glob and Regular Expressions

Shells expand stars and question marks ("wildcards")

- Applications see names after expansion (be careful about no match!)
- The command line can be very long (*very* long)
- This expansion is called "globbing"

But glob expressions are not regular expressions

- they are simpler
- they are much less powerful

If you need to pass special characters to commands:

- You can prefix with backslash
- You can use single quotes
- You can use double quotes

Single quotes preserve '$', double quotes expand variables

Shell Control Statements

In shell conditionals, the expression is a command

- **Success (exit 0) makes the condition true**
- **Even "[" is a command**

```
if <cmd>; then <cmd> [; <cmd> ...]; fi

while <cmd>; do <cmd> [; <cmd> ...]; done
```

"for" and "case", on the other hand, loop on strings

```
for n in <list>; do <cmd> [; <cmd> ...]; done

case <str> in
      <glob>) <cmd> ;;
      <glob>) <cmd> ;;
esac
```

Shell variables

Variables are local to the shell or "environment" variables

- Environment variables are process attributes, at OS level
- Local variables are not inherited by child processes
- The shell command "export" is used to promote a variable to the environment
- The shell expands an unexistent variable into an empty string

Special constructs (good in scripts):

- Default value: ${var:-val}
- Default assignment: ${var:=val}
- Error if missing: ${var:?errmsg}

Some variables carry a system-wide special meaning

- $PATH, for example. is known to execvp(3) and to the shell itself
- Many commands can be configured through environment variables
- The environment is a simple tool for parameter passing

The Most Important Tools (1/2)

cat

- conCATenate several files (neutral element of pipe operator)

grep

- Global Regular Expression Printer, estracts file lines

ls, cp, mv, chmod, chown, ln, mknod, mkfifo df, du, wc

- manage files, their metadata, their size

echo, pwd, touch, yes, true, false, tee

- trivial but very useful programs

less

- a pager, to look at file contents (less is better than more)

test, [

- evaluates a conditional, used in "if" and "while" loops

The Most Important Tools (2/2)

file

- reports the type of a file, from content (ignores the name)

od, sort, uniq, head, tail, basename, dirname

- useful programs that work on file content or name
- there are a huge number of them

find, xargs

- look for files (by name, date, etc), turn stdin into cmdline arguments

sed, awk, perl

- an higher level of data management, for gearheads

netcat

- like cat, but over a network (like telnet, in a way)

The Most Important Devices

/dev/null

- Black hole (empty when read)

/dev/zero

- Infinite source of zeroes (black hole when written)

/dev/random

- Infinite source of very strong random numbers (needs entropy)

/dev/urandom

- Infinite source of random numbers

/dev/tty

- The controlling terminal for this process

/dev/mem

- Physical memory (reserved to the superuser)

/dev/ports

- I/O ports of the PC (reserved to the superuser)

/dev/stdin

/dev/stdout

/dev/stderr

- The predefined streams for the current process

Files and file types, quickly

The filesystem

All files in a Unix system are part of a tree

- No system can exist without a file system
- It's a single tree, even if you have many disks
- Everything (well, almost) is seen like a file

```
/bin
/etc
/dev
/home
/lib
/mnt
/usr
/usr/bin
/usr/lib
/proc
/sys
/sbin
/tmp
/var
```

File types

Any on-disk file belongs to one of these types:

- **Regular file**
- **Directory**
- **Char device**
- **Block device**
- **Symbolic link**
- **FIFO ("named pipe")**
- **Socket**

```
laptopo% mkfifo /tmp/fifo
laptopo% ls -Fld /bin/ls /tmp /tmp/fifo /dev/ttyS0 /dev/sda \
          /usr/bin/vi /tmp/.X11-unix/X0
-rwxr-xr-x 1 root   root     118280 Mar 14  2015 /bin/ls*
brw-rw---- 1 root   disk       8,   0 May  4 08:17 /dev/sda
crw-rw---- 1 root   dialout    4, 64 May  4 08:17 /dev/ttyS0
drwxrwxrwt 7 root   root    1138688 May  5 06:45 /tmp/
srwxrwxrwx 1 root   root          0 May  4 08:17 /tmp/.X11-unix/X0=
prw-rw-r-- 1 rubini staff         0 May  5 06:45 /tmp/fifo|
lrwxrwxrwx 1 root   root         20 Apr 26  2015 /usr/bin/vi ->
                      /etc/alternatives/vi*
```

Inter-Process Communication (minimal)

Command Line and Exit code

The simplest IPC method is:

- Command-line parameter passing
  - Parameters are arbitrary strings
  - The command line can be many kilobytes long
- stdin/out/err as already-opened files
  - The new process can receive commands or data
  - The new process can send commands or data
- Environment variables for configuration parameters
  - The child inherits the parent's environment
  - We can even avoid the command line altogether
- The return value received through _exit(2)
  - 0 is generally used to mean "success"
  - Other values (1-255) mark specific errors
  - The parent can know if the child was killed
- Example: wor0if, worif

`fork(2), execve(2), _exit(2), wait(2)`

Signals

A signal is an asynchronous event

- A process sends it using kill(2) and sigqueue(2)
- A process catches it using signal(2) and sigaction(2)
- It doesn't convey any information (unless extensions are used)

Signal(7) describes the signals

- SIGTERM is control-C
- SIGCHLD reports a child terminated
- SIGSTOP blocks the process (control-Z)
- SIGCONT resurrects a blocked process
- SIGUSR1, SIGUSR2 are reserved for the user (often to {in,de}crease debug)
- SIGIO reports asynchronous input
- SIGWINCH reports a text-terminal size change

Pipes and named pipes

The pipe is a communication channel between processes

- pipe(2) creates the two endpoints, atomically
- One or both file descriptors can be passed to children
- Very useful for multi-process applications

A FIFO ("named pipe") is accessed through the filesystem

- Unrelated processes can communicate
- You may be surprised by blocking behaviours

Sockets

A socket is a file descriptor connected to the network

- What "the network" is depends on the protocol family

Sockets in the AF_UNIX family are filesystem-based

- They are not the same as localhost TCP/IP

AF_UNIX sockets, like AF_INET, offer two main socket types

- SOCK_STREAM
- SOCK_DGRAM (e.g.: syslog)

Several advantages over pipe/FIFO

- Communication between unrelated processes
- One entry point for several concurrent unmixed clients
- Support for client authentication
- Communication can be local or remote, with minimal changes

Other tools

There a are a number of other tools

- **Both simple and complex**
- **Some are obsolete and only present in text books**
- **Some are old but good and powerful**
- **Some are very modern and possibly not very stable**

I personally love mmap(2) but there's no time left

Misc ideas

The Pike rules for C programming

1. You can't tell where a program is going to spend its time.

- Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.

2. Measure.

- Don't tune for speed until you've measured, and even then don't unless one part of the code overwhelms the rest.

3. Fancy algorithms are slow when N is small (it usually is).

- Fancy algorithms have big constants. Until you know that N is frequently going to be big, don't get fancy. (Even if n does get big, use Rule 2 first.)

4. Fancy algorithms are buggier than simple ones

- and they're much harder to implement. Use simple algorithms as well as simple data structures.

5. Data dominates.

- If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident.
  Data structures, not algorithms, are central to programming.

6. There is no rule 6

Worse is better

Worse is better (Richard P. Gabriel, 1989)

- Simplicity
  - The design must be simple, in implementation and if possible in interface.
    Simplicity is the most important consideration in a design.
- Correctness
  - The design must be correct in all observable aspects.
    "Simpler" must be preferred to "correct".
- Consistency
  - The design must not be overly inconsistent.
    Consistency can be sacrificed for simplicity in some cases.
- Completeness
  - The design must cover as many important situations as is practical.
    Completeness can be sacrificed in favor of any other quality.

Other useful rules

- Follow the KISS rule (Keep it simple, stupid)
- First make it work, then make it work well (if time allows)

Other rules and notes

Always split mechanisms and policies

- Examples: X11, IPV4

Silence is golden

- Don't make your output dirty with unneeded information

Always (always) check error conditions

- It will save you hours or days of grief

Code repetition is __BAD__

- Never use cut-and-paste as a programming technique
  - You'll replicate your bugs, and will make them unfixable
  - The design of your code will be hidden
- Use tables and data structures to avoid repetition

`http://en.wikipedia.org/wiki/Unix_philosophy`

The three virtues of the programmer, according to Larry Wall

- Laziness, impatience, hubris.