Systems Programming in Linux

Jörg Faschingbauer

www.faschingbauer.co.at

jf@faschingbauer.co.at

Table of Contents

FASCHINGBAUER

Overview

FASCHINGBAUER

Overview

FASCHINGBAUER

Central Concepts

- Kernel
- Userspace
- Prozess
- File descriptor
- … and a couple more

Overview

FASCHINGBAUER

Processes

- Separate Address Spaces
- Access violations
- Attributes (UID, GID, CWD, ...)
- Resource limits
- ...

Threads - "Lightweight Processes"

Threads (aka lightweight processes) ...

- Are part of a process
- Share the address space of the entire process (for good?)
- $\rightarrow$ Synchronization mechanisms
- $\rightarrow$ Communication mechanisms
- Not originally part of Unix
- $\rightarrow$ don't behave well if one does not take care

Scheduling

FASCHINGBAUER

- Kernel grants CPU resources to processes (and threads)
- Processes and threads are equally important
- Traditional: *fair* scheduling $\rightarrow$ no guarantees who's next
- Realtime options; really fit for time critical applications

Overview

Filesystem

There is only on hierarchy, starting at the *Root Directory* ('/'). Consists of

- Directories
- Files
- Hard- and softlinks
- Device Special Files
- Extended through *mounts* at *mount points*

Everything is a File

FASCHINGBAUER

- File descriptors (and processes) are *the* central concept in Unix
- … and especially in Linux

Overview

Kernel (1)

FASCHINGBAUER

Makes sure that “Userspace” is comfortable:

- Linear address space, with swap
- Preemptive multitasking → Fairness
- No interrupts which can do harm. Well, not really: there are signals!
- Individuals are protected against each other
- Hardware is not visible as such

Kernel (2)

Facts:

- There is no process named “kernel”! Kernel is the sum of all processes running in the system, together with hardware interrupts.
- A process changes to *Kernel Mode* by issuing *System Calls*
- In Kernel Mode he can do anything he wants

Overview

User Space

Protected area where the “normal” programs live

- Per-process, infinite address spaces
- Shell
- C-Library
- Nice programming paradigms which we’ll get to know shortly

Overview

FASCHINGBAUER

Now for Some Examples

All those basic concepts are interwoven

- No process without a *current working directory*
- Who creates files? Only processes do.
- Who creates userspace at boot? Who starts the first process?
- Where would the kernel find the first program? (On the root filesystem)
- ...

Examples welcome ...

Overview

FASCHINGBAUER

The Shell, demystified (1)

FASCHINGBAUER

Starting a program, non-destructively

```
$ sleep 10
...
$
```

Here the following happens:

- Shell generates a child process and *waits* until it *terminates*
- Child *executes* `/usr/bin/sleep`
- Child *terminates*

The Shell, demystified (2)

Starting a program, destructively

```
$ exec sleep 10
```

What was that?!

Separation between Process and Executable

FASCHINGBAUER

In Windows, creating a process is executing a program:

- `CreateProcess()` create a new process by starting a program from an executable file

Unix is different:

- `fork()` creates a new process. Same executable, exact *copy* of parent's address space.
- `exec()` Loads an executable *into* the running process's address space — replacing the current content.

The proc Filesystem

FASCHINGBAUER

Virtual file system that provides a view into the system. For example:

/proc/self

```
$ ls -l /proc/self
lrwxrwxrwx 1 root root ... /proc/self -> 3736
$ ls -l /proc/self
lrwxrwxrwx 1 root root ... /proc/self/
```

Please poke around!

Price question: why is /proc/self/exe a link to /bin/ls?

Executable?

FASCHINGBAUER

Permissions

```
$ ls -l /bin/ls
-rwxr-xr-x 1 root root 109736 Jan 28 18:13 /bin/ls
```

The file's name is not `ls.exe`, but rather it is *executable*.

Executable: Shared Libraries

FASCHINGBAUER

Shared Libraries

```
$ ldd /bin/ls
 linux-vdso.so.1 =>  (0x00007fff15b69000)
 librt.so.1 => /lib/librt.so.1 (0x00007fa763546000)
 libacl.so.1 => /lib/libacl.so.1 (0x00007fa76333d000)
 libc.so.6 => /lib/libc.so.6 (0x00007fa762fe4000)
 libpthread.so.0 => /lib/libpthread.so.0 (0x00007f...
 /lib64/ld-linux-x86-64.so.2 (0x00007fa76374f000)
 libattr.so.1 => /lib/libattr.so.1 (0x00007fa762bc...
```

Executable: Memory Mappings

FASCHINGBAUER

Virtual memory is used to compose the memory layout of a process:

/proc/<pid>/maps

```
$ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 08:02 1375644 /bin/cat
0060a000-0060b000 r--p 0000a000 08:02 1375644 /bin/cat
0060b000-0060c000 rw-p 0000b000 08:02 1375644 /bin/cat
...
```

Overview

FASCHINGBAUER

Simple is beautiful

One sometimes has to think more to reach simplicity.
This pays off a thousand times.

Ok: a File is a File

A file is a file. That's simple. There are tools explicitly made to read and write files, everybody can use these.

Write to a File

```
$ echo Hello > /tmp/a-file
```

Read from a File

```
$ cat /tmp/a-file
Hello
```

Is a Serial Interface a File?

Why not? Data go out and come in!

Write into the Cable

```
$ echo Hello > /dev/ttyUSB0
```

Read off the Cable

```
$ cat /dev/ttyUSB1
Hello
```

Pseudo Terminals

- History: login via a hardware terminal, connected through a serial line
- Terminal (TTY) layer (in the kernel) implements session management
- *Pseudo Terminal*: software instead of cable

Consequentially, output to a pseudo terminal is like writing to a cable, err, file.

Write to a Pseudo Terminal

```
$ echo Hello > /dev/pts/0
```

Disks and Partitions

FASCHINGBAUER

USB Stick Backup

```
# cat /proc/partitions
major minor  #blocks  name

   8        32    2006854 sdc
   8        33    2006823 sdc1
# cp /dev/sdc1 /Backups/USB-Stick
# mount -o loop /Backups/USB-Stick /mnt
```

/proc and /sys

- Kernel has variables in memory that configure certain aspects of its operation
- Most of these variables are exposed as files

Corefiles should be named core.<PID>

```
# echo core.%p > /proc/sys/kernel/core_pattern
```

Suspend to Disk

```
# echo disk > /sys/power/state
```

Random Numbers

FASCHINGBAUER

Kernel, respectively drivers, collect entropy from certain kinds of interrupts.

Emptying the Entropy Pool

```
$ cat /dev/random
```

Overview

FASCHINGBAUER

Overview

Programming Languages C und C++

FASCHINGBAUER

- Files end with `.h`, `.c` (C) and `.cc` or `.cpp` (C++)
- Not executable
- *Compilation* creates `.o` files
- Multiple `.o` files aggregated into an *executable* or a *shared library* (`.so`), through *linking*
- Multiple `.o` files aggregated into *static library*, through *archiving*
- Compilation with (GNU-)Compiler (`gcc`, `g++`).
- Linking with `ld`, better yet with `gcc` und `g++` *frontends*.
- Archiving with `ar`.

Important Options of the GNU C Compiler

FASCHINGBAUER

`-c`	Just compile, don't link
`-o file`	Output to file `file` (default: `inputfile.o`)
`-D macro`	Preprocessor macro
`-D V=1`	Preprocessor macro with value
`-O2`	Optimization level 2
`-O0`	Optimization off
`-g`	Create debug information
`-I directory`	Append `directory` to include path
`-Wall`	Activate "almost" all warnings
`-pedantic`	ISO C/C++ pedantry
`-Werror`	Warnings become errors

Additional Warnings (Excerpt)

FASCHINGBAUER

`-Wold-style-cast`	Non-void C style casts (C++)
`-Woverloaded-virtual`	Signature mismatch (C++)
`-Wswitch-enum`	Missing case label
`-Wfloat-equal`	Comparing floating point numbers using ==
`-Wshadow`	A variable shadows another
`-Wsign-compare`	Signed/unsigned comparison
`-Wsign-conversion`	Implicit sign conversion possible

More than one ever wanted to know → `info gcc`, `man gcc`

Example: C compilers call

FASCHINGBAUER

Building an object file

```
$ gcc -c -o hello.o hello.c
```

Archiving (Static Libaries)

- Archive ⇔ Static library
- Straightforward collection of one or more object files in a single file
- Extension `.a` → `lib`*`basename`*`.a`
- Not dynamically loadable
- Linker *copies* elements into resulting executable

Creating a static library

```
$ ar cr libhello.a hello1.o hello2.o
```

Linking an Executable

FASCHINGBAUER

Linker call using `gcc` or `g++`, rather than `ld` directly.
Options:

`-o file`	Output file `file` (default: `a.out`)
`-g`	Link with debug information
`-s`	"strip" (remove symbol information)
`-L directory`	Add `directory` to library search path
`-l basename`	Library *basename*, along library search path
`-static`	Static linking (don't use shared libraries)

Example: Linking an Executable

FASCHINGBAUER

Linking, Using Separate Compilation

```
$ gcc -I../hello -c -o main.o main.c
$ gcc -o the-exe main.o -L../hello -lhello
```

Linking and Compiling in one Swoop

```
$ gcc -o the-exe main.c -L../hello -lhello
```

Library by file

```
$ gcc -o the-exe main.c ../hello/libhello.a
```

Shared Libraries

FASCHINGBAUER

- *Linked Entity*, out of one or more object files
- “Executable with multiple entry points”
- Extension `.so` → `lib<name>.so`
- Loaded dynamically at program start (*no copy* at build time)
- Ends with `.so` oder `.so.<VERSION>`
- Difference from Windows DLL: *everything* exported.

Example: Linking a Shared Library

FASCHINGBAUER

Linking, Using Separate Compilation

```
$ gcc -fPIC -c -o hello1.o hello1.c
$ gcc -fPIC -c -o hello2.o hello2.c
$ gcc -shared -o libhello.so hello1.o hello2.o
```

Linking and Compiling in one Swoop

```
$ gcc -fPIC -shared -o libhello.so hello1.c hello2.c
```

Shared Libraries - Problems

FASCHINGBAUER

- Library missing or not found
- Library does not fit (symbols missing)
- Library not *compatible* (program crashes or otherwise misbehaves) $\rightarrow$ “ABI” violation

Tricky:

- Libraries use other libraries, these again use libraries
- C++ adds more easy opportunity for incompatibilities
- C++ ABI helps, but does in no way give protection against home-made bugs (e.g., naive addition of a virtual method)

Shared Libraries - Central Libraries

FASCHINGBAUER

`libc.so.6`	C language runtime, system calls
`libdl.so.2`	Dynamic loading of libraries
`libpthread.so.0`	POSIX threads implementation
`libm.so.6`	math support
`librt.so.1`	"Realtime" (e.g. POSIX message queues)
`linux-vdso.so.1`	Kernel interface (virtual)

Shared Libraries - Diagnosis

FASCHINGBAUER

Which libraries does the shell need, and where are they found?

Bash Dependencies

```
$ ldd /bin/bash
linux-vdso.so.1 =>  (0x00007fff5e3ff000)
libncurses.so.5 => /lib/libncurses.so.5 (0x00007f6...
libdl.so.2 => /lib/libdl.so.2 (0x00007f6e1a957000)
libc.so.6 => /lib/libc.so.6 (0x00007f6e1a5fe000)
/lib64/ld-linux-x86-64.so.2 (0x00007f6e1adad000)
```

Shared Libraries — Loader Path

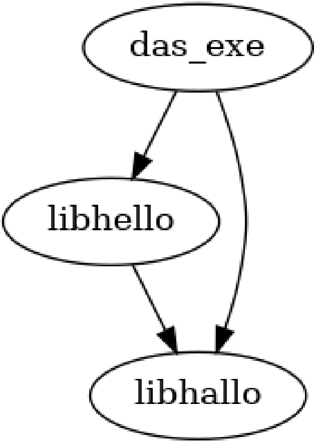
FASCHINGBAUER

Search path for shared libraries during *load time*:

1. `LD_PRELOAD` (except SUID/SGID)
2. Compiled-in `RPATH`
3. `LD_LIBRARY_PATH` (except SUID/SGID)
4. `/etc/ld.so.conf` → `/etc/ld.so.cache`
5. `/usr/lib`
6. `/lib`

Libraries — Linker Path

Linker does only `one` “pass” → **Library order is significant.**



Right

```
$ gcc ... -lhello -lhallo ...
```

Wrong

```
$ gcc ... -lhallo -lhello ...
```

Overview

System Calls

The kernel is not a library → no direct function calls, but rather “System Calls”.

- Entry points into the kernel
- Every system call has a unique number and a fixed set of parameters and registers (ABI)
- Changes context from user mode to kernel mode
- Implementation is CPU specific (software interrupt ...)
- Numbers, parameters, etc. are Linux specific
- “Kernel acts on behalf of a process”

→ `man syscalls`

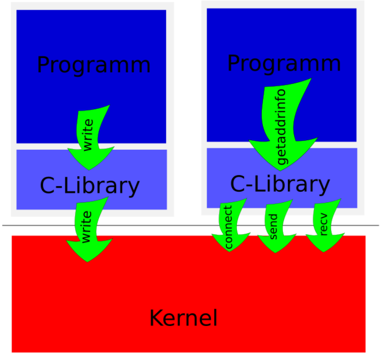
System Calls and the C-Library

FASCHINGBAUER

System calls are never used directly by a program ...

Syscall Wrapper

```
#include <unistd.h>
int main() {
    write(1, "Hallo\n", 6);
    return 0;
}
```



Library Function or System Call?

FASCHINGBAUER

Distinction is not always clear → Manual pages

System calls
(manual section 2)

- `write()`
- `read()`
- `connect()`
- ...

No system calls
(manual section 3)

- `malloc()`
- `printf()`
- `getaddrinfo()`
- ...

Manual Pages

`man [section] name.`

For example: `man man` $\rightarrow$

1 User Commands
2 System Calls
3 C Library Functions
4 Devices and Special Files
5 File Formats and Conventions
6 Games et. Al.
7 Miscellanea
8 System Administration tools and Daemons

Overview

The `errno` Variable

On error, system calls (and most C library functions) return -1 and set the *global* variable `errno`.

Error Handling with System Calls

```
ssize_t n = read(fd, buffer, sizeof(buffer));
if (n == -1)
    if (errno == EINTR)
        /* interrupted system call, retry possible */
    else
        /* abort, reporting the error */
```

errno is *global*

FASCHINGBAUER

Where's the bug?

Bad Error Handling

```
ssize_t n = read(fd, buffer, sizeof(buffer));
if (n == -1) {
    fprintf(stderr, "Error %d\n", errno);
    if (errno == EINTR)
        /* ... */
}
```

Helper Functions

- `void perror(const char *s)` Message to `stderr`, beginning with `s`
- `char *strerror(int errnum)` *Modifiable* pointer pointer to error description
- `char *strerror_r(int errnum, char *buf, size_t buflen)` Cleanest alternative

Error output

```
if (n == -1)
    perror("read()");
```

Overview

FASCHINGBAUER

Exercise: Hello World

FASCHINGBAUER

1. Write a “Hello World” and build it. (Only `main()` and `printf()` in a single file.)
2. Refactoring: divide this program into an executable containing the `main()` function, and a library which contains the rest. The library is then statically linked into the executable.
3. Add this program to out CMake build environment.

Overview

FASCHINGBAUER

Overview

FASCHINGBAUER

File Descriptors

- Universal “Handle” for everything that’s got to do with I/O.
- Type: `int`
- “File” is only one shape of I/O
- Pipes, Sockets, FIFOs, Terminals, Device Special Files
  (→ entry point into arbitrary kernel drivers)
- Linux specific ingenuities: `signalfd()`, `timerfd_create()`, `eventfd()`

Standard Filedescriptors

Number	POSIX Macro	`stdio.h` equivalent
0	`STDIN_FILENO`	`stdin`
1	`STDOUT_FILENO`	`stdout`
2	`STDERR_FILENO`	`stderr`

- Interaktive Shell: all three associated with terminal
- Standard input and output used for I/O redirection and pipes
- Standard error receives errors, warnings, and debug output

$\Longrightarrow$ Windows-Programmers: no errors, warnings, and debug output to *standard output*!!

File I/O System Calls

`open()`	Opens a file (or creates it → Flags)
`read()`	Reads bytes
`write()`	Writes bytes
`close()`	Closes the file

`open()` creates file descriptors that are associated with path names (files, named pipes, device special files, ...). Other "Factory" functions: `connect()`, `accept()`, `pipe()`, ...

`read()`, `write()`, `close()` valid for sockets, pipes, etc.

open()

FASCHINGBAUER

man 2 open

```
int open(const char *pathname, int flags, ...);
```

Swiss army knife among system calls. Multiple actions, governed by bitwise-or'ed flags:

- Create/Open/Truncate/...
- Access mode (Read, Write)
- Hundreds of others

`open()` Flags

Access Mode

- `O_RDONLY`: Write → error
- `O_WRONLY`: Read → error
- `O_RDWR`: ...

Creating a File

- `O_CREAT`: create if not exists
- `O_CREAT|O_EXCL`: error if exists

Miscellaneous

- `O_APPEND`: write access appends at the end
- `O_TRUNC`: truncate file to zero length if already exists
- `O_CLOEXEC`: `exec()` closes the file descriptor (→ later)

read()

FASCHINGBAUER

man 2 read

```
ssize_t read(int fd, void *buf, size_t count);
```

- Return value: number of bytes read (-1 on error)
- "0" is "End of File"
- Can read less than `count` (usually with network I/O)

write()

FASCHINGBAUER

man 2 write

```
ssize_t write(int fd, const void *buf, size_t count);
```

- Return value: number of bytes written (-1 on error)
- Can write less than `count` (usually with network I/O)
- Connections (e.g. pipe, socket): connection loss → `SIGPIPE` (process termination)

File Offset: `lseek()`

`read()` and `write()` manipulate the "offset" (position where the next operation begins).
Explicit positioning:

man 2 lseek

```
off_t lseek(int fd, off_t offset, int whence);
```

Positioning *beyond file size*, plus write to that position → "holes", occupying no space
Read from a hole → null bytes.

The Rest: `ioctl()`

FASCHINGBAUER

- "tunnel" for functionality not declarable as I/O
- Most commonly used to communicate with drivers
  - E.g.: "Open that CD drive!"

```
man 2 ioctl
int ioctl(int fd, int request, ...);
```

- Mostly deprecated nowadays (though easily implemented in a driver)
- Better (because more obvious): use `/proc` and `/sys`

Overview

FASCHINGBAUER

Exercise: File I/O Basics

1. Write a program that interprets its two arguments as file names, and copies the first to the second. The first must be an existing file (error handling!). The second is the target of the copy. No existing file must be overwritten.
2. Create a file that is 1 GB in size, but occupies only a couple of bytes physically.

Overview

FASCHINGBAUER

File Descriptors, Open File, I-Node

FASCHINGBAUER

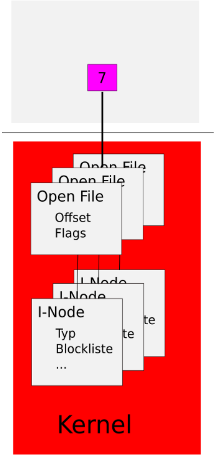
File descriptor is a “handle” to a more complex structure

File (“Open File”)

- Offset
- Flags

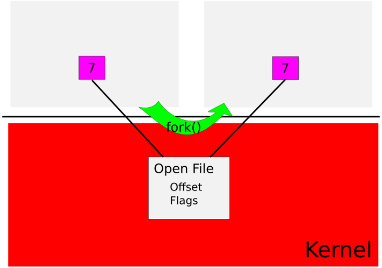
I-Node

- Type
- Block list
- ...



File Descriptors and Inheritance

- A call to `fork()` inherits file descriptors
- $\rightarrow$ reference counted copy of the same “Open File”.
- $\rightarrow$ Processes share flags and offset!
- File closed (*open file* freed) only when last copy is closed



Duplicating File Descriptors

man 2 dup

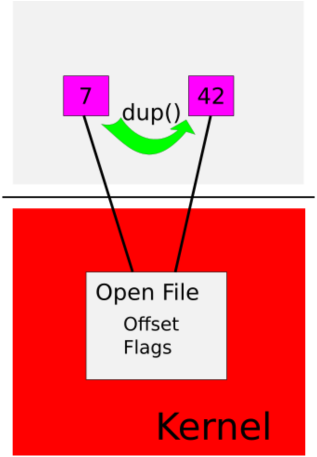
```
int dup(int oldfd);
```

- Return: new file descriptor

man 2 dup2

```
int dup2(int oldfd, int newfd);
```

- `newfd` already open/occupied → implicit `close()`



Example: Shell Stdout-Redirection (1)

Stdout-Redirection

```
$ /bin/echo Hello > /dev/null
```

- Redirection is a shell responsibility (`/bin/bash`)
- `echo` writes “Hello” to standard output.
- … and does not want/have to care where it actually goes

Example: Shell Stdout-Redirection (2)

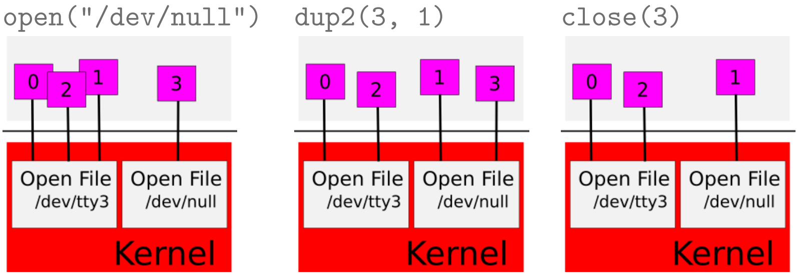
Stdout-Redirection

```
$ strace -f bash -c '/bin/echo Hallo > /dev/null'
[3722] open("/dev/null", O_WRONLY|O_...) = 3
[3722] dup2(3, 1) = 1
[3722] close(3) = 0
[3722] execve("/bin/echo", ...) = 0
```

(`fork()`, `exec()`, `wait()` omitted for clarity.)

Example: Shell Stdout-Redirection (2)

FASCHINGBAUER



Overview

FASCHINGBAUER

I/O without Offset Manipulation

FASCHINGBAUER

- `read()` and `write()` have been made for *sequential* access.
- Random access only together with `lseek()`
- Inefficient
- Not **atomic** → Race Conditions!

man 2 pread

```
ssize_t pread(int fd, void *buf, size_t count,
              off_t offset);
ssize_t pwrite(int fd, const void *buf, size_t count,
               off_t offset);
```

Scatter/Gather I/O

FASCHINGBAUER

- Often data are not present in one contiguous block
  - E.g. layered protocols
- → Copy pieces together, or issue repeated small system calls
- → Scatter/Gather I/O

man 2 readv

```
ssize_t readv(int fd,
    const struct iovec *iov, int iovcnt);
ssize_t writev(int fd,
    const struct iovec *iov, int iovcnt);
```

Scatter/Gather I/O, without Offset Manipulation

FASCHINGBAUER

Wortlos ...

man 2 preadv

```
ssize_t preadv(int fd,
    const struct iovec *iov, int iovcnt,
    off_t offset);
ssize_t pwritev(int fd,
    const struct iovec *iov, int iovcnt,
    off_t offset);
```

Attention: Linux specific

Truncating Files

FASCHINGBAUER

- Truncating a file ...
- ... or create a hole (∼ `lseek()`)

man 2 truncate

```
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

File Descriptors - Allocation

FASCHINGBAUER

Value of the next file descriptors is not arbitrarily chosen → next free slot, starting at 0.

Filedescriptor Selection

```
close(STDIN_FILENO);
int fd = open("/dev/null", O_WRONLY);
assert(fd == 0);
```

Overview

FASCHINGBAUER

Exercises: File I/O, Offset Conflict

FASCHINGBAUER

- Create a file (file descriptor `fd1`) and open it a second time (file descriptor `fd2`). Write bytes `abc` in both file descriptors. Examine the file's content. What's there and what did you expect?
- Modify the program from the previous exercise, and pass the flag `O_APPEND` to both `open()` calls. What do you notice?
- Instead of creating two independent file descriptors using `open()`, create the second from the first using `dup()`, and see what's happening.

Exercise: File I/O, `dup()`, Offset

FASCHINGBAUER

- See how duplicated file descriptors share one offset. For example, write on one of them and check the offset on the second. (*Hint:* read `man 2 lseek()` for how to get the offset associated with a file descriptor.)

Overview

FASCHINGBAUER

What Has Happened

FASCHINGBAUER

What Has Happened

- Fundamental Unix: `open()`, `read()`, `write()`, `close()`
- Semantics of file descriptors
  - Inheritance across `fork()`
  - Duplicating file descriptors
- Files can have holes, and other ridiculosities
- `strace`

What's next?

- Processes

Overview

FASCHINGBAUER

Overview

FASCHINGBAUER

Processes and Programs (1)

FASCHINGBAUER

A process has the following basic properties:

- Independently running unit
  - Instruction pointer, stack pointer, register, ...
- Separate address space
  - 32 bit pointers $\rightarrow$ 4G addressable memory
  - Virtual memory
  - Organized in stack, heap, text, initialized and uninitialized data
  - Access protection

Processes and Programs (2)

FASCHINGBAUER

A programm is a file containing the rules for composing a process's address space.

- Executable format: ELF ("Executable and Linkable Format") → `man 5 elf`
- Contains so-called "Sections"
  - Text: instruction/code
  - Data: initialized data (C: global variables which are explicitly initialized)
  - Sections for dynamic linking/loading
  - C++: constructors and destructors of global objects
  - ... and much more ...

Loader loads a program and configures its address space
→ `man 8 ld.so`

Overview

Attributes: Overview

FASCHINGBAUER

- Process ID (PID). Unique ID of every process.
- Process ID of the process's parent (PPID).
- Program name. The program file the process is running from.
- Current working directory (CWD).
- Commandline arguments.
- Environment variables
- "Credentials". A set of user and group IDsthat define permissions.

PID, PPID

man 2 getpid

```
pid_t getpid(void);
pid_t getppid(void);
```

- Every process knows about its parent → tree structure
- First process has PID 1 (called "`init`")
- `init` has PPID 0 → does not exist ("kernel")

Argument Vector

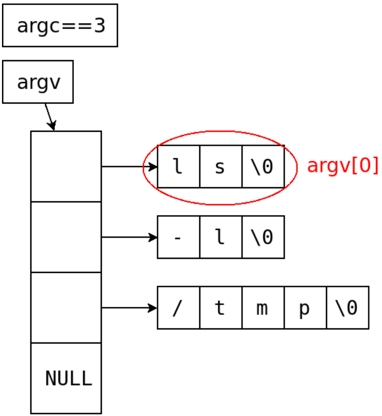
FASCHINGBAUER

main

```
$ ls -l /tmp
```

main

```
int main(int argc, char** argv)
{
    ...
}
```



Environment (1)

FASCHINGBAUER

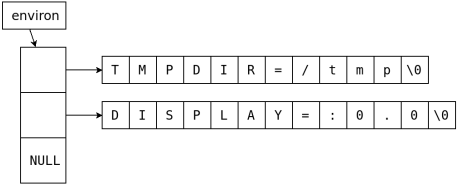
Environment variables

- Are copied from parent at process creation $\rightarrow$ “inherited”
- Prominent examples:
  - `HOME`, `USER`. Home directory; set by the login program
  - `DISPLAY`. Set by the graphical login manager (if any)

Environment (2)

man 7 environ

```
extern char **environ;
char *getenv(
    const char *name);
int putenv(char *string);
int setenv(
    const char *name,
    const char *value,
    int overwrite);
int unsetenv(
    const char *name);
int clearenv(void);
```



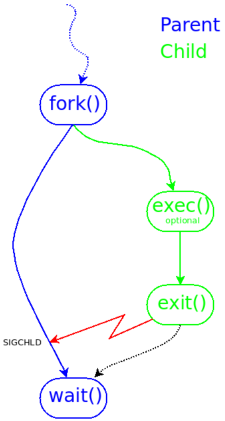
Overview

FASCHINGBAUER

Life Cycle of Processes

FASCHINGBAUER

- `fork()` creates a new process
- `exec()` sets up the process address space from an excutable file (PID remains the same) and passes control to the code
- `exit()` terminates a process → “Exit Status”
- `wait()` synchronizes the caller with the termination of a child process



Example: Shell Command

FASCHINGBAUER

$ /bin/echo Hello, seen by shell

```
$ strace -f bash -c '/bin/echo Hello'
clone(...) = 14272
[14271] wait4(-1, Process 14271 suspended
 <unfinished ...>
[14272] execve("/bin/echo",["/bin/echo", "Hello"],...
[14272] write(1, "Hello\n", 6) = 6
[14272] exit_group(0) = ?
<... wait4 resumed> [,,], 0, NULL) = 14272
```

Create Process: `fork()`

FASCHINGBAUER

`man 2 fork`

```
pid_t fork(void);
```

`fork()` splits the process in two → **two** return values.

Important:

- 1:1 Copy of the address space
- → Child runs from the same executable

`fork()` in Action

```
pid_t process = fork();
if (process == 0) {
    /* Child (green) */
}
else if (process > 0) {
    /* Parent (blue) */
}
else {
    /* Error */
}
```

Execute Program: `exec()`

FASCHINGBAUER

Executing a program

- Sets up the address space of an **existing** process
- Most work done by userspace → `ld.so`
- File descriptors remain open (→ shell I/O redirection)
- … except `O_CLOEXEC` ("Close-on-exec") file descriptor flag
- Signal handlers removed
- Memory mappings removed

Example: Shell's `exec`

FASCHINGBAUER

Shell `exec`

```
$ exec sleep 5
```

Re-mixes the address space of the running process (the interactive shell)

- `sleep` terminates
- Terminal waits until shell terminates (`wait()`)
- → Terminal terminates

exec() Variants (1)

Actual system call:

man 2 execve

```
int execve(
    const char *filename,
    char *const argv[],
    char *const envp[]);
```

- `filename` is the path to the executable (absolute or relative)
- Has nothing to do with `argv[0]` → can be set to anything

exec() Variants (2)

FASCHINGBAUER

C library wrappers:

man 3 execl

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg,
           ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
           char *const envp[]);
```

Terminate Process: `exit()` (1)

FASCHINGBAUER

Terminate, without any ado like flushing stdio buffers → raw system call

```
man 2 _exit
        void _exit(int status);
```

Attention:

- Process is really shot the hard way
- `atexit()` handlers not called
- → (e.g.) `stdio` buffers are not flushed

Terminate Process: `exit()` (2)

FASCHINGBAUER

Nicer termination: flushing buffers before termination

```
man 3 exit
void exit(int status);
int atexit(void (*function)(void));
```

- `atexit()` registers callbacks
- → in a signal handler only `_exit()` possible

Exit Status

FASCHINGBAUER

Exit status leaves parent an 8 bit number. Arbitrary, but the convention is …

- $0 \rightarrow$ Ok
- $!{=}0 \rightarrow$ Error

Exit Status and the Shell

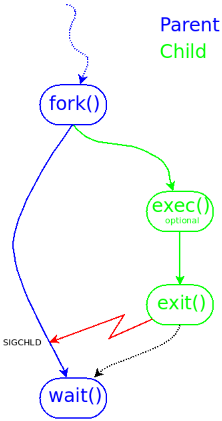
```
$ if echo Hello > /dev/null; then
>   echo $? is Ok
> fi
0 is Ok
```

Child Surveillance: `wait()`

FASCHINGBAUER

`wait()` yields information about a child process's status change

- Voluntary termination (by calling `exit()`)
- Involuntary termination (by an unexpected *signal*)
- Stopped (e.g. `Ctrl-Z` through terminal → `SIGSTOP`)
- Continued (z.B. `fg` from the shell → `SIGCONT`)



wait()

FASCHINGBAUER

Simplest form:

man 2 wait

```
pid_t wait(int *status);
```

- Waits until a child terminates
- Yields its PID as return value
- Sets `status`
- Caller has no child process altogether → Error

`waitpid()`

FASCHINGBAUER

man 2 waitpid

```
pid_t waitpid(pid_t pid, int *status, int options);
```

`pid` specifies *which* child to wait for

- `pid` > 0: wait for child with `pid`
- `pid` == −1: wait for any child
- `pid` == 0 oder `pid` < −1: process group

`options` (0 → "no particular special wishes")

- `WUNTRACED`: "stopped" is reported (default: no report)
- `WCONTINUED`: "continued" is reported (default: no report)
- `WNOHANG`: don't block; no dead child → return value 0

Exit Status According to `wait()`

Exit status: an integer carries much information

`W*` Macros in Action

```
int status;
pid = waitpid(-1, &status, WUNTRACED|WCONTINUED);
if (WIFEXITED(status))
    printf("Exited: %d\n", WEXITSTATUS(status));
else if (WIFSIGNALED(status))
    printf("Signal: %d (%s)\n", WTERMSIG(status),
        WCOREDUMP(status)?"core":"no core");
else if (WIFSTOPPED(status))
    printf("Stopped: %d\n", WSTOPSIG(status));
else if (WIFCONTINUED(status))
    printf("Continued\n");
```

→ `man 2 wait`

Orphans and Zombies

FASCHINGBAUER

Zombie:

- Process that does not exist anymore (→ cannot be killed)
- Exit status has not been fetched by parent → a program that calls `fork()` should not forget to `wait()`.
- Status in e.g. `ps` output: “defunct”
- Its only sign of existence is an entry in the kernel process table

Orphan:

- Parent terminates → children become “orphans”
- Kernel assigns them PID 1 (`init`) as parent (orphanage)

Overview

FASCHINGBAUER

Exercise: Process Life Cycle

FASCHINGBAUER

Write a program that ...

- Executes a program
- Synchronizes with its termination
- Prints all diagnostics it can get — don't forget about "stopped" and "continued"
- Example call: `starter ls -l /tmp`

Overview

FASCHINGBAUER

Overview

Signals are *poor* notifications to a process

- Number between 1 and 31
- Sent from a process to another process ($\rightarrow$ permissions)
- Hardware exception. E.g. floatingpoint, memory access ...
- Special terminal events: `Ctrl-C` (`SIGINT`), `Ctrl-Z` (*suspend*, `SIGTSTP`) ...
- Software events: timer runs off (`SIGALRM`) ...

Terminology

- *Generate*. A signal is sent.
- *Deliver*. The signal is received by a process ("delivered by the kernel"). The *signal handler* ($\rightarrow$ later) is run.
- *Pending*. A signal is pending on a process until it is delivered.
- *Blocked*. A process refuses to get a signal delivered (he "blocks" the signal).
- *Signal Mask*. The set of signals that are blocked by a process.

Default Actions

FASCHINGBAUER

All signals have a predefined “default action”

- The signal is ignored. E.g. *SIGCHLD*.
- Process termination. “Abnormal Process Termination”, as opposed to `exit()`. With or without *core dump*.
- The process is stopped or continued.

Important Signals

→ `man 7 signal`
→ `kill -l`

Signal	Default Action	Reason
SIGABRT	Terminate (core dump)	E.g. `assert()`
SIGSEGV	Terminate (core dump)	Access violation
SIGBUS	Terminate (core dump)	Access violation
SIGILL	Terminate (core dump)	Bogus function pointer
SIGFPE	Terminate (core dump)	Floating point
SIGINT	Terminate	`Ctrl-C`
SIGTERM	Terminate	Explicit `kill`
SIGPIPE	Terminate	Write to pipe/socket
SIGCHLD	Ignore	Child death

Sending Signals

man 2 kill

```
int kill(pid_t pid, int sig);
```

`pid` specifies where the signal goes to

- `pid > 0`: process
- `pid == -1`: *Broadcast*; every process the sender has permissions to. Except `init` and the sender itself.
- `pid == 0` or `pid < -1`: process group

Warning!

FASCHINGBAUER

Warning!

- Signals are no toy
- Signals are no communication medium
- Signal handlers are executing in a context that has nothing to do with normal program context $\rightarrow$ asynchronous
- One does not install a signal handler for e.g. `SIGSEGV`
- One does not ignore `SIGSEGV`
- One does not block `SIGSEGV`
- ...

Signal Set: `sigset_t`

Signal Set: eine set of signals. Signals are numbered 1 through 31 → `int` resp. `sigset_t`.

man 3 sigsetops

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

The *Signal Mask* (1)

FASCHINGBAUER

Signal Mask:

- Process attribute (more exactly: *thread*)
- Specifies which signals are blocked
- Signal that have been sent to a process but which he blocks remain *pending*

Pending signals:

- Get *delivered* as soon as they are unblocked
- Signals of the same type don't pile up at the receiver → two *SIGINT* are only *delivered* once

The *Signal Mask* (2)

Setting/modifying the signal mask:

man 2 sigprocmask

```
int sigprocmask(int how,
    const sigset_t *set, sigset_t *oldset);
```

Pending Signals:

man 2 sigpending

```
int sigpending(sigset_t *set);
```

Signal Handlers

To change the “default action” of a signal one installs a *signal handler* — Pointer to a function with the following signature:

Signal Handler

```
void handler(int sig);
```

Installing a Signal Handler

man 2 sigaction

```
struct sigaction {
    void     (*sa_handler)(int);
    sigset_t   sa_mask;
    int        sa_flags;
};
int sigaction(int signum,
    const struct sigaction *act,
    struct sigaction *oldact);
```

Special `sa_handler` **values:**

- `SIG_IGN`: ignore the signal
- `SIG_DFL`: restore default action

Effects of Signal Delivery

FASCHINGBAUER

E.g. terminate a program based upon the value of a flag (by dropping out of a loop) that is set in a signal handler. Use ...

sig_atomic_t

```
volatile sig_atomic_t flag;
```

- Blocking system calls (e.g. `read()` or `write()`) return an error when they have been interrupted by a signal
- `errno` is `EINTR`

Last Warning!

Signals are delivered asynchronously

- In a signal handler, only *async-signal-safe* functions can be used
- → practically only system calls

→ `man 7 signal`

The following functions (among many others) are not async-signal-safe:

- `printf()`, `sprintf()` (everything from `stdio.h` and `iostream`, respectively)
- `malloc()`, `free()` etc.
- `exit()` (`_exit()` can be used)
- Everything from `pthread.h`

Overview

Exercise: Signals

FASCHINGBAUER

Write a program that ...

- ... reads from `STDIN_FILENO` in a loop, and outputs what was read to `STDOUT_FILENO`. Imagine that this is a replacement for an immensely important work which can block — the program blocks on `STDIN_FILENO`.
- On program termination, the program has to do important cleanup work — it has to catch at least `SIGINT` and `SIGTERM`.
- Our cleanup work is to safely — *not* in the signal handler — write "Goodbye!" to standard output.

Overview

FASCHINGBAUER

Overview

FASCHINGBAUER

Owner and Permissions

Types of permissions

- Read (`r`)
- Write (`w`)
- Execute (`x`)

Separate permissions for

- User (`u`)
- Group (`g`)
- Others (`o`)

Permission Bits

FASCHINGBAUER

File Permissions

```
$ ls -l /etc/passwd
-rw-r--r-- ... /etc/passwd
```

Bits	Meaning
`-`	Type: regular file
`rw-`	Read- and writable for owner (root)
`r--`	Readable for group
`r--`	Readable for others

Execute Permissions

FASCHINGBAUER

Execute Permissions

```
$ ls -l /bin/ls
-rwxr-xr-x ... /bin/ls
```

Facts ...

- An executable file does not have to end with `.exe` to be executable
- ... it simply *is* executable

Directory Permissions

FASCHINGBAUER

Directory Permissions

```
$ ls -ld /etc
drwxr-xr-x ... 07:54 /etc
```

- Read permissions: *content* (list of names) is readable
- Execute permissions: to access a file (e.g. for reading), one has to have *execute permissions* on the parent directory and all directories along the path
- *The right to chdir into the directory*

Permission Bits, octal

`ls -l` Output	Binary	Shell command
`-rw-r--r--`	110100100	`chmod 0644 ...`
`-rw-------`	110000000	`chmod 0600 ...`
`-rwxr-xr-x`	111101101	`chmod 0755 ...`

System calls take an integer argument → mostly given octal

Default Permissions – `umask`

FASCHINGBAUER

The U-Mask ...

- Bit field
- *Subtracted* from default permissions at file/directory creation
- Process attribute → inherited

`umask` in Action

```
$ umask
0022
$ touch /tmp/file
$ ls -l /tmp/file
-rw-r--r-- ... /tmp/file
```

`umask`: How Does it Work?

- `umask` *subtracted* from default permissions
- `umask` is an (inherited) process attribute
- Default permissions at file creation: `rw-rw-rw-`

Default permissions	`rw-rw-rw-`	110 110 110	0666
- U-Mask	`----w--w-`	000 010 010	0022
Outcome	`rw-r--r--`	110 100 100	0644

Shell Commands

- Permission modification (set to octal value):
  `$ chmod 755 ~/bin/script.sh`
- Permission modification (differential symbolic):
  `chmod u+x,g-wx,o-rwx ~/bin/script.sh`
- Group ownership modification (only root and members of the group can do this):
  `chgrp audio /tmp/file`
- Ownership modification (only root):
  `chown user /tmp/file`
- `chmod`, `chown`, and `chgrp` understand `-R` for "recursive".

Set-UID Bit

FASCHINGBAUER

Set-UID Bit: motivation

- *Ugly hack!*
- Encrypted passwords in `/etc/passwd` or `/etc/shadow`
- Only `root` can modify
- I (`jfasch`) want to change my password
- Have to become `root`
- … but cannot

passwd

```
$ ls -l /bin/passwd
-rws--x--x 1 root root ... /bin/passwd
```

Sticky Bit

Sticky bit: motivation

- *Ugly hack!*
- Everyone has write permissions in `/tmp`
  - $\Longrightarrow$ everyone can create files
  - $\Longrightarrow$ everyone can remove files
- Chaos: everyone can remove each other's files

Sticky Bit in `/tmp`

```
$ ls -ld /tmp
drwxrwxrwt ... /tmp
```

Owner and Permissions: System Calls

FASCHINGBAUER

man 2 chown

```
int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
```

man 2 chmod

```
int chmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

Overview

Directories and Links

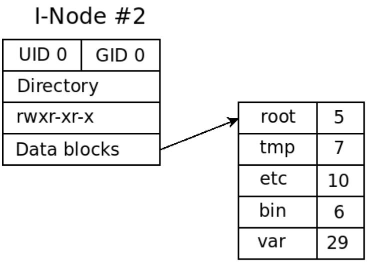
FASCHINGBAUER

- Directory: file containing pairs (`name,inodenummer`)
- Hardlink: directory entry that points to the same i-node as another entry
  - $\rightarrow$ the two are indistinguishable
- Symbolic (soft-, sym-) link: file containing the *name* of another file
  - Closest to what's called a "shortcut" in Doze (however that's implemented there)

Directory

Directory

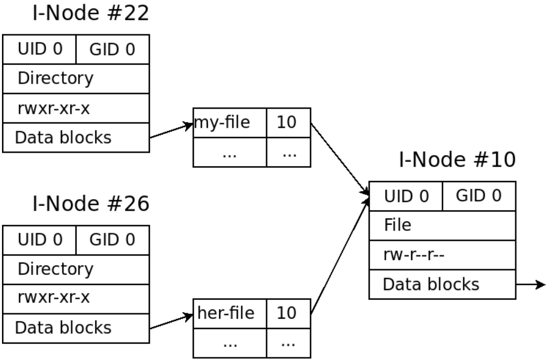
- Internally organized as a file
- Except that `read()` and `write()` are not possible
- Operations:
  - `opendir()`, `readdir()`, `closedir()`
  - `mkdir()`
  - `rmdir()`: remove entry that points to empty directory
  - `unlink()`: remove an entry that points to a non-directory



Hard Link

Hard Link

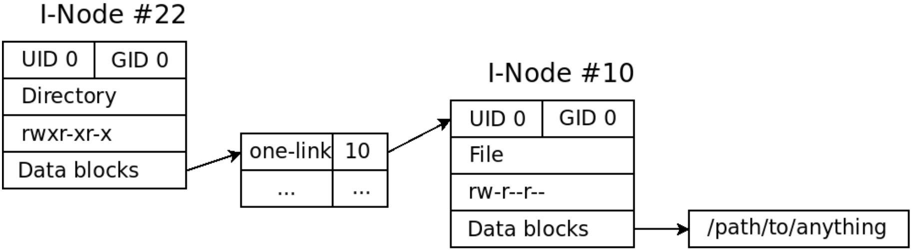
- `link()`
- Circular hard links possible → can only point to non-directories
- Only within the same file system



Soft Link

Soft Link

- “Symbolic link”, “Symlink”
- `open()`/`opendir()` on a symlink → “de-reference”
    - Operates on the pointed-to entry
- Link creation: `symlink()`
- Determine the link’s target: `readlink()`
- Target need not exist → “Dangling Link”



`unlink()` Semantics

- One can remove entries that other processes have open
  - File descriptors refer to the pointed-to *I-node*
- Only the directory entry is removed $\rightarrow$ file becomes invisible
- I-node (and associated data) remain on-disk
- I-node is freed only when last referring file descriptor is closed

Overview

FASCHINGBAUER

Overview

Why Threads?

FASCHINGBAUER

`fork()` is so beautiful

- New process
- New address space
- → no race conditions
- → simple is beautiful!

But ...

- Process creation is expensive
- Separate address space → communication is cumbersome
- Portability: Windows has no idea

Typical Uses

- Use of multiple processors for compute-intensive calculation
- One is force to use a library that blocks
    - A no-go in a GUI application for example
    - Push it in a thread, call it there, and communicate with the thread however you feel best
    - Communication $\rightarrow$ later
- Blocking I/O
    - Like the blocking library: push it in a dedicated thread
    - But there are better anti-naive solutions (Unix is not Windows)

Overview

- Creating threads
- Synchronisation: *Mutex*
- Communication: *Condition variable*
- Thread specific data (a.k.a. thread local storage)
- One-time initialization

Legal (1)

Threads of one process share the following resources:

- Process memory
- PID and PPID
- Credentials
- Open files
- Signal *handler*
- Umask, Current Working Directory, etc.
- ...

Legal (2)

FASCHINGBAUER

Threads have the following attributes of their own:

- Thread ID (TID)
  - Scheduler only cares about *threads*
  - A process is just a container (which happens to have the ID of the *main thread*)
- Stack
- `errno`
- Signal **mask**
- Thread specific data (TSD)
- ...

POSIX Thread API

FASCHINGBAUER

- POSIX thread API is not implemented in the kernel
  - User space library
  - `man 3 ...`
  - `strace` is of limited use
- `errno` is thread spezific → "semi-global"
- No PThread function sets `errno`
  - They generally return what otherwise would be `-errno`
  - *Thank you!*
- `gcc -pthread`
  - Defines macro `_REENTRANT`
  - Links `-lpthread`
  - C++: thread safe initialization of local `static`

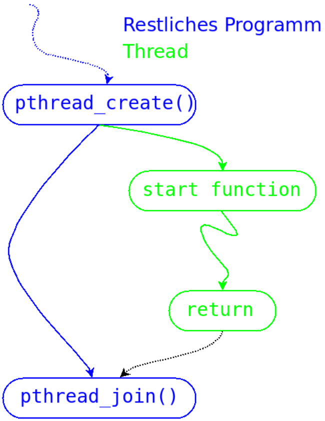
Overview

FASCHINGBAUER

Thread Life Cycle

- `pthread_create()` creates new thread
- *Start function* is called
- Thread terminates
- `pthread_join()` synchronizes with termination (fetches "exit status")

No parent/child relationship → anybody can join



Thread Creation

FASCHINGBAUER

man 3 pthread_create

```
int pthread_create(
    pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_routine) (void *), void *arg);
```

- `thread`: ID of the new thread ("output" parameter)
- `attr` → see later (`NULL` → default attribute)
- `start_routine`: thread start function, `void*/void*`
- `arg`: parameter of the start function

Thread Termination (1)

FASCHINGBAUER

Thread termination alternatives:

- Return from start function
- `pthread_exit()` from somewhere inside the thread (cf. `exit()` from a process)
- `pthread_cancel()` from outside (cf. `kill()`)
- `exit()` of the entire process → all contained threads are terminated

Don't use `pthread_cancel()` unless you know what you are doing!

Thread Termination (2)

FASCHINGBAUER

Without any further ado: the manual ...

man 3 pthread_exit

```
void pthread_exit(void *retval);
```

man 3 pthread_cancel

```
int pthread_cancel(pthread_t thread);
```

Exit Status, pthread_join()

FASCHINGBAUER

A thread's "exit status":

- `void*`, just like the start parameter → more flexible than a process's `int`.
- Parameter to `pthread_exit()`
- Return type of the start function

man 3 pthread_join

```
int pthread_join(pthread_t thread, void **retval);
```

Detached Threads

FASCHINGBAUER

Sometimes one does not want to use `pthread_join()`

- Rather, run a thread in the "background".
- "Detached" thread
- Thread attribute

man 3 pthread_attr_setdetachstate

```
int pthread_attr_setdetachstate(
    pthread_attr_t *attr, int detachstate);
PTHREAD_CREATE_DETACHED
  Threads that are created using attr will be created in a
  detached state.
```

- Detaching at runtime ...

man 3 pthread_detach

```
int pthread_detach(pthread_t thread);
```

Thread ID

- `pthread_create()` returns `pthread_t` to the caller
- Thread ID of calling thread: `pthread_self()`
- Compare using `pthread_equal()`

man 3 pthread_self

```
pthread_t pthread_self(void);
```

man 3 pthread_equal

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

“Scheduled Entities” (1)

FASCHINGBAUER

Kernel maintains “scheduled entities” (Process IDs, “1:1” scheduling)

Threads inside `firefox`

```
$ ps -eLf|grep firefox
$ ls -1 /proc/30650/task/
13960
13961
... (many more) ...
```

"Scheduled Entities" (2)

FASCHINGBAUER

Too bad:

- Scheduled entity's ID *is not the same as* `pthread_t`
- Correlation of OS threads and POSIX thread is Linux specific

man 2 gettid

```
pid_t gettid(void);
```

Overview

FASCHINGBAUER

Exercises: Thread Creation, Race Condition

FASCHINGBAUER

- Write a program that creates two threads. Each one of the threads increments *the same* integer, say, 10000000 times.
  - The integer is shared between both threads (allocated in the `main()` function). A pointer to it gets passed to the thread start function.
  - The threads don't increment a copy of the integer, but rather access *the same* memory location.

  After the starting process (the *main thread*) has synchronized with the incrementer's termination, he outputs the current value of the said integer.
  What do you notice?

Overview

Race Conditions (1)

Suppose `inc()` is executed by at least two threads in parallel:

Very bad code

```
static int global;

void inc()
{
    global++;
}
```

CPU A		CPU B		
Instr	Reg	Instr	Reg	Mem
load	42	load	42	42
inc	43	inc	43	42
	43	store	43	43
store	43		43	43

- *The variable `global` has seen only one increment!!*
- "Load/Modify/Store Conflict"
- The most basic race condition

Race Conditions (2)

Imagine more complex data structures (linked lists, trees): if incrementing a dumb integer bears a race condition, then what can we expect in a multithreaded world?

- No single data structure of C++'s Standard Template Library is thread safe
- `std::string`'s copy construktor and assignment operator are thread safe (*GCC's Standard C++ Library* → *not* by standard)
- `std::string`'s other methods are *not* thread safe
- *stdio* and *iostream* are thread safe (by standard since C++11)

Mutex (1)

FASCHINGBAUER

man 3 pthread_mutex_init

```
int pthread_mutex_init(pthread_mutex_t *mutex,
       const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- Dynamic initialization using `pthread_mutex_init()`/`pthread_mutex_destroy()`
- `attr == NULL` → default mutex (→ later)
- Static initialization using `PTHREAD_MUTEX_INITIALIZER`

Mutex (2)

FASCHINGBAUER

man 3 pthread_mutex_lock

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Simple lock/unlock must be enough
- If you find yourself using “trylock”, then something’s wrong
- *Polling is never right!*

Mutex (3)

Better code

```
static pthread_mutex_t global_mutex =
    PTHREAD_MUTEX_INITIALIZER;
static int global;

void inc()
{
    /* error handling omitted */
    pthread_mutex_lock(&global_mutex);
    global++;
    pthread_mutex_unlock(&global_mutex);
}
```

Mutex Types

FASCHINGBAUER

man 3 pthread_mutexattr_settype

```
int pthread_mutexattr_settype(
    pthread_mutexattr_t *attr, int type);
```

- `PTHREAD_MUTEX_NORMAL`: no checks, no nothing. Same thread locks mutex twice in a row before unlock → *Deadlock*.
- `PTHREAD_MUTEX_ERRORCHECK`: Deadlock check; unlocking a mutex locked by another thread → *Error*
- `PTHREAD_MUTEX_RECURSIVE`: owner can lock same mutex twice
- `PTHREAD_MUTEX_DEFAULT` → `PTHREAD_MUTEX_NORMAL`

Atomic Instructions

Simple integers don't need a mutex

fetch_and_add()

```
static int global;

void inc()
{
    __sync_fetch_and_add(&global, 1);
}
```

More → `info gcc`, GCC manual

Overview

FASCHINGBAUER

Exercises: Fixing the Race Condition

FASCHINGBAUER

- Use a mutex to protect the integer increment in the last exercise. *What do you notice?*
- Replace the mutex and the increment with a suitable atomic instruction (`__sync_fetch_and_add()`). *What do you notice?*

Overview

Condition Variable (1)

Communication:

- One thread waits for a certain event to happen
- The event is produced by another thread
- The waiting thread does not consume and CPU time while waiting (polling is dumb)
- Solution in Windows: *WIN32 Events* (auto-reset, manual-reset)

POSIX is different: *Condition Variablen*

- No state (as opposed to WIN32 Events — set/unset)
- Operations `wait()` and `signal()`
- Useless on its own
- Building block to build custom communication mechanisms around custom conditions

Condition Variable (2)

Sample conditions (*predicates*, in POSIX parlance):

- Event has been set
- Message queue is not empty anymore
- Message queue is not full anymore
- Semaphore count is not zero anymore
- ...

Condition is coupled with a state which is protected by a *mutex*. For example:

- Boolean flag “set/unset”
- Message queue implementation (linked list?)

Condition Variable: `wait()`

man 3 pthread_cond_wait

```
int pthread_cond_wait(
    pthread_cond_t *cond,
    pthread_mutex_t *mutex);
```

In an **atomic** (otherwise → "Lost Wakeup") operation

- Releases mutex
- Suspends caller until condition variable is *signaled* by another thread

Condition Variable: `signal()`

FASCHINGBAUER

```
man 3 pthread_cond_signal

int pthread_cond_signal(pthread_cond_t *cond);
```

Again, in an **atomic** operation:

- Wakes one waiter if any
- Lets him acquire the mutex

Example: WIN32 Auto Reset Event (1)

FASCHINGBAUER

Setting the event

```
void set_autoreset_event(Event* ev)
{
    pthread_mutex_lock(&ev->mutex);
    ev->value = 1;
    pthread_mutex_unlock(&ev->mutex);
    pthread_cond_signal(&ev->is_set);
}
```

Example: WIN32 Auto Reset Event (2)

FASCHINGBAUER

Waiting for the event

```
void wait_autoreset_event(Event* ev)
{
    pthread_mutex_lock(&ev->mutex);
    while (ev->value != 1) {
        pthread_cond_wait(&ev->is_set, &ev->mutex);
        /* mutex acquiriert */
    }
    ev->value = 0; /* "autoreset" */
    pthread_mutex_unlock(&ev->mutex);
}
```

Condition Variable: Checking the Predicate

FASCHINGBAUER

Use `while` **instead of** `if`, because ...

- Spurious wakeups are possible (for example if the PThread implementation is using signals internally)
- Multiple waiters are woken (broadcast)
    - Predicate is true, but the first thread invalidates it immediately

Condition Variable: Initialization

FASCHINGBAUER

man 3 pthread_cond_init

```
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *cond,
      const pthread_condattr_t *attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- Dynamic initialization using `pthread_cond_init()`/`pthread_cond_destroy()`
- `attr == NULL` → default condition variable
- Static initialization using `PTHREAD_COND_INITIALIZER`

Condition Variable: Miscellaneous

FASCHINGBAUER

man 3 pthread_cond_broadcast

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

man 3 pthread_cond_timedwait

```
int pthread_cond_timedwait(
    pthread_cond_t *cond,
    pthread_mutex_t *mutex,
    const struct timespec *abstime);
```

Overview

FASCHINGBAUER

Exercises: Message Queue (1)

FASCHINGBAUER

Write a program that ...

- ... starts a consumer thread. The consumer reads data from the queue, and writes it to Standard Output. The consumer thread should terminate by receiving a special token over the queue.
- ... starts a producer thread. The producer read data from Standard Input, line by line. Each line is sent to the consumer over the queue.
- When the producer see *end of file* on Standard Input, he inserts a *quit* token into the queue and terminates.
- The main thread joins with both threads, and terminates once both are done.

Exercises: Message Queue (2)

FASCHINGBAUER

Write a program that ...

- ... starts a consumer thread. The consumer reads data from the queue, and writes it to Standard Output. The consumer thread should terminate by receiving a special token over the queue.
- ... starts a producer thread. The producer read data from Standard Input, line by line. Each line is sent to the consumer over the queue.
- When the producer see *end of file* on Standard Input, he inserts a *quit* token into the queue and terminates.
- The main thread joins with both threads, and terminates once both are done.

Overview

One-Time Initialization (1)

Where's the bug?

Bad code

```
static X *global;

void use_global()
{
    if (global == NULL)
        global = new X;
    // ... use global ...
}
```

One-Time Initialization (2)

FASCHINGBAUER

Good code

```
static pthread_once_t global_once = PTHREAD_ONCE_INIT;
static X *global;
static void init_global() { global = new X; }

void use_global()
{
    pthread_once(&global_once, init_global);
    // ... use global ...
}
```

One-Time Initialization (3)

FASCHINGBAUER

man 3 pthread_once

```
int pthread_once(pthread_once_t *once_control,
       void (*init_routine)(void));
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

Thread Specific Data, Thread Local Storage

POSIX thread API for "Thread Specific Data" – per thread global variables → `man 3 pthread_key_create` (including example).
Non-portable alternative:

`__thread` Keyword

```
static __thread X* global;
```

Overview

Last Warning

FASCHINGBAUER

Multithreading does not go together well with `fork()`

- `fork()` copies the address space → locked mutexes
- `fork()` leaves only the calling thread alive in the child
  - All others are gone
- If you have to use `pthread_atfork()` you're lost
- `exec()` is ok — everything's gone anyway.
  - But why the hell would one do this?
- Signals are not ok at all

Last Warning

FASCHINGBAUER

Multithreading is dangerous!

- It is sexy
- It is easy — a thread is created in no time (gosh: C++11)
- There are race conditions *everywhere*
- Keep hands off cancellation
- Careful when sharing data structures → global variables aren't bad for no reason
- Debugging is nearly impossible

Last Warnung

FASCHINGBAUER

`man pthreads`: legalese that deserves reading

- "Thread-safe functions": please *please* read!
- "Async-cancel-safe functions" → don't use cancellation

Overview

FASCHINGBAUER

Overview

Scheduling

FASCHINGBAUER

Scheduler ...

- Assigns processes/threads to processors
- Decides *for how long* they will run
- *"Fair" Scheduling*: Unix tradition from the beginning
  - *Timeslices*: everyone gets their share
  - Inexact tuning opportunity: "nice" value
- *Realtime scheduling*: inherently unfair

Nice Values

Nice Value ...

- Specifies how "nice" a process is
- Between -20 (not nice) and +20 (very nice)
- +20 → only runs when noone else wants the CPU
- Non-root user can only increase nice value ("become nicer")

→ `man 1 nice`, `man 2 nice`, `man 1 renice`, `man 2 setpriority`

Overview

FASCHINGBAUER

Realtime Scheduling

FASCHINGBAUER

Realtime is not fair

- One process in an infinite loop can bring the system to halt
  - Not possible in a fair world
  - ... even when being -20 nice
- $\rightarrow$ Only root

Scheduling Policies

FASCHINGBAUER

Scheduling policies determine the scheduler's way of assigning CPUs ...

- `SCHED_OTHER`: the fair world
- `SCHED_FIFO`
  - Process get CPU immediately assigned
  - Remains on CPU until he relinquishes
  - ... or a higher prio process wants CPU
- `SCHED_RR` (Round Robin)
  - Like `SCHED_FIFO`
  - Equal prio processes: short timeslices in round robin order

Scheduling priorities

- 0 ... Reserved for good old fair processes (`SCHED_OTHER`)
- 1-99 ... Realtime priorities.

Scheduling: Examples

FASCHINGBAUER

Do nothing high-prio, FIFO policy:

chrt in Action

```
chrt -f 42 sleep 7
```

Modify scheduling attributes of existing process 4697:

chrt in Action

```
chrt -p -f 42 4697
```

Scheduling: System Calls

FASCHINGBAUER

Manipulating scheduling attributes of a process:

man 2 sched_setscheduler

```
int sched_setscheduler(
    pid_t pid, int policy,
    const struct sched_param *param);
int sched_getscheduler(pid_t pid);

struct sched_param {
    int sched_priority;
};
```

Scheduling: Threads (1)

FASCHINGBAUER

Manipulating scheduling attributes of an existing thread:

man 3 pthread_setschedparam

```
pthread_setschedparam(
    pthread_t thread, int policy,
    const struct sched_param *param);
pthread_getschedparam(
    pthread_t thread, int *policy,
    struct sched_param *param);
};
```

Scheduling: Threads (2)

FASCHINGBAUER

Start a new thread with predefinied scheduling attributes:

man 3 pthread_attr_setschedparam

```
int pthread_attr_setschedparam(
    pthread_attr_t *attr,
    const struct sched_param *param);
```

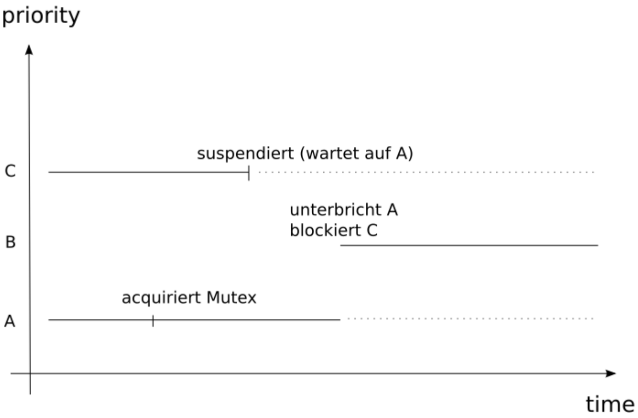
man 3 pthread_attr_setschedpolicy

```
int pthread_attr_setschedpolicy(
    pthread_attr_t *attr, int policy);
```

Overview

Priority Inversion

FASCHINGBAUER



Priority Inversion: Mutex Protocols (1)

Solution, in spoken words: at the time that C wants the mutex, A has to carry on → "protocol" between both, communicated via the mutex → Mutex Attribute

man 3 pthread_mutexattr_setprotocol

```
int pthread_mutexattr_setprotocol(
    pthread_mutexattr_t *attr,
    int protocol);
```

Priority Inversion: Mutex Protocols (2)

FASCHINGBAUER

Mutex Protocols

- `PTHREAD_PRIO_INHERIT`: A's priority is *temporarily* (until mutex is acquired) boosted to B's
- `PTHREAD_PRIO_PROTECT`: A's priority is temporarily risen to a fixed limit (→ `man 3 pthread_mutexattr_setprioceiling()`)

Overview

Overview

FASCHINGBAUER

Sockets

FASCHINGBAUER

First of all: *a socket is a file*

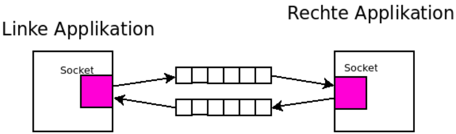
- Communication mechanism
- On the same machine or between different machines
- Different *types*: *stream* and *datagram*
- Different *families*: the “Internet” socket family is only one in many

Sockets: “Stream”

FASCHINGBAUER

Stream-Sockets

- *Connection* between two *endpoints* (sockets)
- *Reliable*: bytes are delivered, or an error occurs
- No record boundaries (stream of bytes)
- Bi-directional

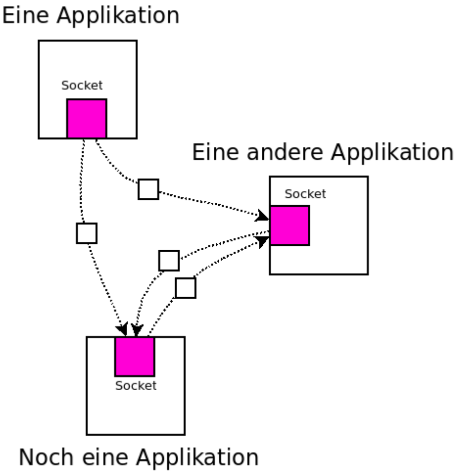


Sockets: "Datagram"

FASCHINGBAUER

Datagram sockets

- Datagrams → record boundaries
- Unreliable → datagrams can be lost or duplicated
- No connection → a socket can send datagrams to *multiple* receiver sockets



Sockets: Adress Families

FASCHINGBAUER

The Internet is not the only medium that can be communicated over → "Adress Families"

- Internet IPv4 (`AF_INET`)
- Internet IPv6 (`AF_INET6`)
- Local (`AF_UNIX`)
- Bluetooth (`AF_BLUETOOTH`)
- Novell (`AF_IPX`)
- Appletalk (`AF_APPLETALK`)
- ...

Sockets: `socket()` (1)

FASCHINGBAUER

Design principle:

- All socket system calls are *independent* of type and address family
- `socket()` ist eine generic "factory" → *file descriptor*

man 2 socket

```
int socket(int domain, int type, int protocol);
```

- `domain`: adress family (`AF_INET`, `AF_INET6`, `AF_UNIX`, `AF_BLUETOOTH`, ...)
- `type`: `SOCK_STREAM`, `SOCK_DGRAM`

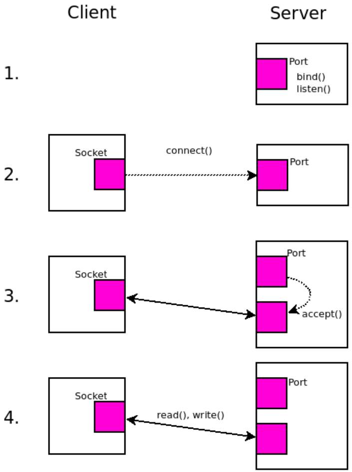
Sockets: `socket()` (2)

- `protocol`: if there are no alternatives, `protocol` is left 0

	SOCK_STREAM	SOCK_DGRAM
AF_INET	TCP	UDP
AF_INET6	TCP	UDP
AF_UNIX	-	-
AF_BLUETOOTH	L2CAP, HCI, BNEP	RFCOMM

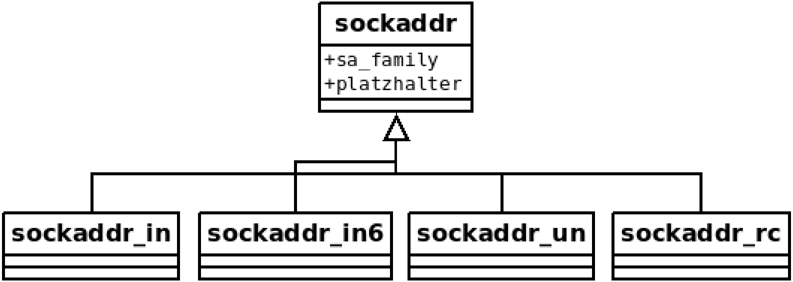
Sockets: Connection Establishment

1. Server is ready
2. Client establishes connection
3. Server accepts connection
4. Connection is ready



Sockets: Adresses

- Object oriented (well ...)
- `sockaddr` ist “Base Class” with a type field



Sockets: Server is Ready (1)

FASCHINGBAUER

Server is ready

1. Allocates socket (`socket()`)
2. Binds it to an *address* (`bind()`)
3. Activates it to accept incoming connections (`listen()`)

man 2 bind

```
int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

Sockets: Server is Ready (2)

FASCHINGBAUER

man 2 listen

```
int listen(int sockfd, int backlog);
```

- `backlog`: maximum number of yet unaccepted connections (`SOMAXCONN`)

Sockets: Client Establishes Connection

FASCHINGBAUER

Client establishes connection

1. Allocates socket (`socket()`)
2. Connects it to a server that is bound to an address (`connect()`)

man 2 connect

```
int connect(int sockfd, const struct sockaddr *addr,
            socklen_t addrlen);
```

Sockets: Server Design

FASCHINGBAUER

A server usually accepts multiple connections. Design issues:

- *Iterative*. `accept()`, followed by request treatment (`read()`, `write()`), and finally `close()`
- *Parallel*. Several possiblities:
    - `fork()`. Parent closes the accepted file descriptor, and the `accept()`s the next connection
    - Multithreaded. Just like `fork()`, but without `close()`.
    - Event driven → later.

Sockets: Adresses

FASCHINGBAUER

The key are the addresses ...

- We didn't talk about concrete address schemes
- Just roles: *client* and *server*, and who uses which system calls
- `bind()`, `connect()` and `accept()` receive anonymous `sockaddr`
- *This is intentional!*

Overview

The Internet

FASCHINGBAUER

The Internet (TCP/IP)

- Connects *networks*, which in turn connect computers
- Routing protocols
- Hardware independent addresses
- "Old" version IPv4
- "New" version IPv6 (just nobody believes)
- Domain Name System (DNS)

TCP/IP: Addresses and Ports

IP-Addresses identify machines (one machine can have multiple addresses)

- IPv4 addresses: 32 bit addresses, like `192.168.1.10`
- IPv6 addresses: 128 bit addresses, like `2001:0db8:85a3:08d3:1319:8a2e:0370:7344`

Port identifies a communicating application.

- 16 bit integer

TCP/IP: Network Byte Order (1)

Different architectures have different “byte order”

- “Big Endian”: MSB at lowest memory address
- “Little Endian”: LSB at lowest memory address

IP addresses and port numbers are part of the protocol

- *Network byte order*: big endian

All numbers that belong to addresses (port numbers!), have to be transformed into *network byte order* before putting them into address structures!

TCP/IP: Network Byte Order (2)

FASCHINGBAUER

Conversion macros: host byte order to network byte order (`hton*`) and back (`ntoh*`)

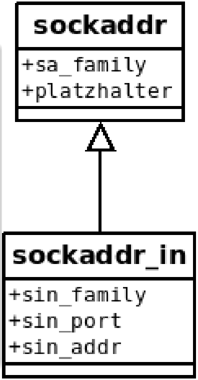
man 3 byteorder

```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

TCP/IP: Addresses (IPv4)

man 7 ip

```
struct in_addr {
    uint32_t s_addr;
};
struct sockaddr_in {
    sa_family_t sin_family;
    in_port_t sin_port; /* net bo. */
    struct in_addr sin_addr;
};
```

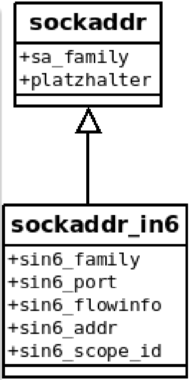


TCP/IP: Addresses (IPv6)

FASCHINGBAUER

man 7 ipv6

```
struct in6_addr {
    unsigned char s6_addr[16];
};
struct sockaddr_in6 {
    sa_family_t sin6_family;
    in_port_t sin6_port; /* net bo. */
    uint32_t sin6_flowinfo;
    struct in6_addr sin6_addr;
    uint32_t sin6_scope_id;
};
```



TCP/IP: Addresses/Constants

FASCHINGBAUER

Before use, initialize addresses: `memset(.,0,.)`!

The following constants and macros make life easier:

- `INADDR_ANY`: IPv4 address 0.0.0.0, "wildcard" address → server accepts connection from all its network interfaces
- `IN6ADDR_ANY_INIT`: IPv6 counterpart of `INADDR_ANY` (C-User: `in6addr_any`)
- `INET_ADDRSTRLEN`: maximal length of an IPv4 dotted-decimal address string
- `INET6_ADDRSTRLEN`: IPv6 counterpart of `INET_ADDRSTRLEN`

TCP/IP: Address Strings

FASCHINGBAUER

String to sockaddr_in or sockaddr_in6 and back:

man 3 inet_pton

```
int inet_pton(int af, const char *src, void *dst);
```

sockaddr_in oder sockaddr_in6 in String:

man 3 inet_ntop

```
const char *inet_ntop(int af, const void *src,
                      char *dst, socklen_t size);
```

TCP/IP: DNS Lookup, Address Conversion

FASCHINGBAUER

`getaddrinfo()`: swiss army knife, can transparently handle IPv4 and IPv6. Please read yourself!

man 3 getaddrinfo

```
int getaddrinfo(
    const char *node,
    const char *service,
    const struct addrinfo *hints,
    struct addrinfo **res);
void freeaddrinfo(struct addrinfo *res);
const char *gai_strerror(int errcode);
```

Overview

FASCHINGBAUER

Exercises: TCP/IP

FASCHINGBAUER

Write a program that ...

- ... accepts command line arguments *host* (in dotted-decimal IPv4) and *port*
- ... creates a connection to the application there
- ... reads one line from standard input, sends it over the connection, and terminates

Overview

UNIX Domain Sockets

FASCHINGBAUER

Local and cheap incarnation of an address family

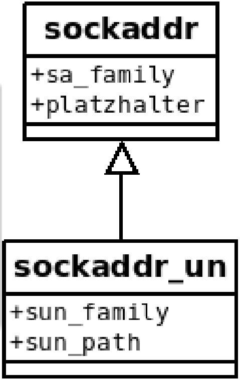
- *Address* is a path in a file system
- The usual permissions apply
  - Permission to connect to a server $\Longleftrightarrow$ Write permission on its socket
- Cheap
  - No complicated flow control between two machines
  - No big buffers on either side
  - Just a piece of kernel memory

UNIX Domain Sockets: Addresses

man 7 unix

```
#define UNIX_PATH_MAX    108

struct sockaddr_un {
    sa_family_t sun_family;
    char sun_path[UNIX_PATH_MAX];
};
```



UNIX Domain Sockets: Examples (1)

X11 uses Unix Domain sockets by default (TCP is too insecure):

X11-Server

```
$ ls -l /tmp/.X11-unix
total 0
srwxrwxrwx 1 root root 0 Feb  7 22:30 X0
```

UNIX Domain Sockets: Examples (2)

FASCHINGBAUER

D-Bus ...

- Distribution of system events ("network connected", "removable media mounted", ...)
- Communication of desktop components (Doze's COM)
- → `man 1 dbus-daemon`

D-Bus daemon, listening

```
$ ls -l /var/run/dbus
total 0
srwxrwxrwx 1 root root 0 Feb  7 22:30 system_bus_socket
```

UNIX Domain Sockets: `socketpair()`

`socketpair()`: **create a connected pair of Unix domain sockets.**

Uses include ...

- Inter thread communiction
- Testbed for protocol implementation
  - TCP, serial line, ... → need hardware
  - Unit tests, saving the need for server and/or hardware setup
- ...

man 2 socketpair

```
int socketpair(
    int domain, int type, int protocol, int sv[2]);
```

Overview

Übung: UNIX Domain Sockets

FASCHINGBAUER

- Schreiben Sie ein Programm, das wie der TCP-Client aus der letzten Übung agiert, bloß zur Kommunikation ein UNIX Domain Socket verwendet
- Passen Sie den Server gleichermaßen an — spendieren Sie ihm einen weiteren Thread, der die Kommunikation über UNIX Domain Sockets macht.
  Der Server sollte vor dem Öffnen des Ports darauf achten, ein eventuell bereits bestehendes zu löschen.

Overview

Overview

Event Loops

FASCHINGBAUER

Event driven programming ...

- Callbacks, as a reaction to events
- Many kinds of events
- e.g. GUI — a very high level
  - “Button pressed”
  - “Button released”
  - ...
- Programming paradigm: state machines

$\rightarrow$ “Main Event Loop”

Blocking System Calls

FASCHINGBAUER

Problems with blocking system calls:

- Graceful termination in a multithreaded program
  - Thread waits for input (in `read()`)
  - How do I tell him to quit his input loop?
- Same with iterative server (sits in `accept()`)
- Reactive programs (ones that do not block) have to start one thread for each blocking task → Horror!

I/O Multiplexing (1)

Wishlist:

- I want to issue a system call (e.g. `read()` on a socket) only when I know that it won't block.
- I want to be notified when that is the case.
- I want notifications on multiple such media.
- When I can do nothing without blocking, I want to block.
- I only want to wake up upon one or more notifications.

Fulfillment in Unix:

- All wishes come true
- Notifications/Events:
  - "Read now possible without blocking"
  - "Write now possible without blocking"
  - "Error"

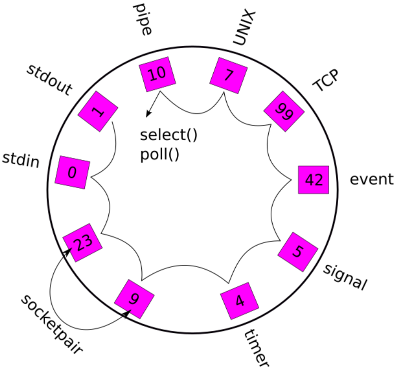
I/O Multiplexing (2)

FASCHINGBAUER

System calls for multi file descriptor surveillance

- `select()`
- `poll()`
- `epoll()` (Linux specific)

Block the caller until at least one file descriptor permits desired activity → "I/O Event"



select()

FASCHINGBAUER

man 2 select

```
int select(int nfds,
   fd_set *readfds, fd_set *writefds,
   fd_set *exceptfds, struct timeval *timeout);

void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

poll()

FASCHINGBAUER

man 2 poll

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
struct pollfd {
    int   fd;         /* file descriptor */
    short events;     /* requested events */
    short revents;    /* returned events */
};
```

Overview

FASCHINGBAUER

Exercise: `select()` and `poll()`

FASCHINGBAUER

Write the following server program ...

- The main thread has an event loop
- At the beginning, the loop maintains a single Unix domain socket — the "port". It is used to accept connections. Hint: the port "can accept without blocking" condition is signaled as *input*.
- Once accepted, connections are also maintained by the loop. The program reads from them as data arrives, and prints the data to standard output.
- Connections remain open until the client closed them. Hint: the server sees and end-of-file condition after being notified about input.

Overview

Signal Handling

FASCHINGBAUER

- Signals are no toy
- Signals are no communication medium
- Signal handlers are executing in a context that has nothing to do with normal program context → asynchronous

Why is that so complicated?

- History!
- Performance: signals save one or two CPU cycles (so they say)

→ in 99.99% of all cases you don't want it that way!

Synchronous Signal Handling: sigwaitinfo()

FASCHINGBAUER

Synchronous and blocking signal handling: wait until a signal is delivered:

man 2 sigwaitinfo

```
int sigwaitinfo(const sigset_t *set, siginfo_t *info);
int sigtimedwait(const sigset_t *set, siginfo_t *info,
                 const struct timespec *timeout);
```

Drawback: an entire thread is blocked

Synchronous Signal Handling: signalfd() (1)

FASCHINGBAUER

What if ...

1. A signal is an event? (It is)
2. I can receive events through file descriptors ...
3. ... so why can't I reveive signals through a file descriptor?

man 2 signalfd

```
int signalfd(int fd, const sigset_t *mask, int flags);
```

Synchronous Signal Handling: `signalfd()` (2)

FASCHINGBAUER

Parameters

- `mask`: set of signals I want to receive through the *signal file descriptor*
- `flags`: `SFD_NONBLOCK`, `SFD_CLOEXEC` (same semantics as the corresponding flags to `open()`)

Semantics

- `read()` blocks until a signal is delivered
- Then you read a C structure `signalfd_siginfo` → `man 2 signalfd`
- Asynchronous delivery *still does happen*
  - → switch off (block signals) with `sigprocmask()`/`pthread_sigmask()`

Synchronous Signal Handling: `signalfd()` (3)

FASCHINGBAUER

Advantages:

- Events are delivered in a natural way: `select()`, `poll()` ...
- No damn signal handler necessary

Drawback:

- Linux specific

Overview

Exercise: `signalfd()`

FASCHINGBAUER

Implement a clean shutdown of our server program

- Use `signalfd()` to create a "receive channel" for the usual shutdown signals `SIGINT` and `SIGTERM`
- Let it participate in the event loop
- Quit the event loop after the receipt of one of those
- Before terminating the program, write out a "Goodbye" message (to easily verify that everything works as intended)

Overview

Timers

FASCHINGBAUER

Traditional Unix ways to let time pass by:

- POSIX timers (`man 2 timer_create`)
  - *one-shot* oder *periodisch*
  - “Event notification” through a signal of your choice
- `nanosleep()` (`man 2 nanosleep`) to block for a given amount of time

→ Both are not satisfactory ...

- I want real events!

Timer Events (1)

FASCHINGBAUER

man 2 timerfd_create

```
int timerfd_create(int clockid, int flags);
int timerfd_settime(
    int fd, int flags,
    const struct itimerspec *new_value,
    struct itimerspec *old_value);
int timerfd_gettime(
    int fd, struct itimerspec *curr_value);
```

Timer Events (2)

FASCHINGBAUER

- Semantics of `timerfd_create()`, `timerfd_settime()` and `timerfd_gettime()` is the same as of POSIX timers (oneshot, periodic, ...)
- `read()` blocks until timer runs off. After that a `uint64_t` is read – number of timer expirations since last `read()`.

→ Pretty, simple, efficient!

Overview

Arbitrary Events: `eventfd()`

The last one: arbitrary events ...

man 2 eventfd

```
int eventfd(unsigned int initval, int flags);
```

- Content of the "file": one `uint64_t`
- `write()` (data: one `uint64_t`, the *addend*) adds the value to the existing content, *atomically*
- `read()` (conversely, into a `uint64_t` memory location) reads the eventfd's current value, and *atomically* resets it to zero
- Like all file descriptors, `select()`, `poll()` can be used

`eventfd()` Applications

Possible applications of `eventfd()`**:**

- Signaling a “Quit” flag from anywhere. For example, signal handler to main event loop.
- Inter thread communication: “I just produced 42 new elements into the queue. You may now read from the queue without blocking.”
- With a bit of fantasy, 100.000 more

Overview

Exercise: `eventfd()`

To be done!

Overview

File Change Events: inotify (1)

File Change Events: “upcalls” from kernel to userspace, as an alternative to polling → filesystem change notifications

Usage:

- Interactive file system browsers (e.g. *Nautilus*)
- Daemons (e.g. `udevd`, watching its own rules files for modification)

Again, fits nicely into the world of event driven programming!

File Change Events: inotify (2)

- File descriptor represents an "inotify instance"
- The instance contains a set of "watches": path names with an associated bitmask (type of change to watch)
- A watch is uniquely identified by a "watch descriptor"
- Events are consumed using `read()`.

→ `man 7 inotify`

File Change Events: inotify (3)

FASCHINGBAUER

Event Structure

```
struct inotify_event {
    int      wd;
    uint32_t mask;
    uint32_t cookie;
    uint32_t len;
    char     name[];
};
```

- `name`: if `len > 0`, contains path to a newly add file (relative to the directory being watched)
- `cookie`: links together related events (e.g. moves)

Overview

FASCHINGBAUER

Overview

Programs (1)

FASCHINGBAUER

Program = Instruction on how to layout the process's memory. Consists of:

- *Header*. Identifies the program type (for example, "ELF shared library")
- *Text*. Machine code.
- *Data*. Values use to initialize global variables. (Constant values, e.g. strings)
- *Relocation Tables*. Fixup addresses for dynamically loaded libraries.
- *Shared Library Informations*. Which libraries does the program need, and in which version?

Programs (2)

FASCHINGBAUER

ELF header of /bin/ls

```
$ readelf --file-header /bin/ls
  ...
  Class:                           ELF64
  Type:                            EXEC (Executa...
  Entry point address:             0x4027e0
  Start of program headers:        64 (bytes int...
  Start of section headers:        108008 (bytes...
  ...
```

Programme (3)

FASCHINGBAUER

Sections of /bin/ls

```
$ readelf --sections /bin/ls
  ...
  [11] .init     PROGBITS  00000000004021e8  000021e8
  [13] .text     PROGBITS  00000000004027e0  000027e0
  [14] .fini     PROGBITS  0000000000411d88  00011d88
  [15] .rodata   PROGBITS  0000000000411da0  00011da0
  [21] .dynamic  DYNAMIC   0000000000619e18  00019e18
  [24] .data     PROGBITS  000000000061a300  0001a300
  [25] .bss      NOBITS    000000000061a520  0001a510
  ...
```

The Program Loader `/lib/ld-linux.so.2`

FASCHINGBAUER

CPU does not execute programs from disk, but rathe from *Memory* → somebody has to take care to *load* the program into memory.

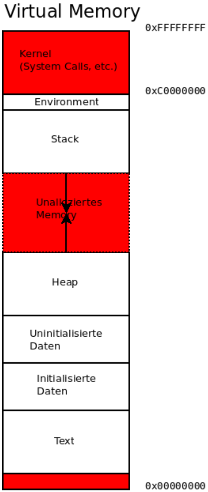
Loader `/lib/ld-linux.so.2`

- Starts a program on behalf of the kernel (`exec()`)
- Reads ELF header, sections, ...
- Sets up the virtual address space of the process
- Passes control to the "Entry Point"

Memory Layout

Memory layout of a process

- *Adress space*: 32 bit pointers → 4G adressable memory
- *Environment*: maintained by the kernel
- *Stack*: expanded *on-demand* by the kernel
- *Heap*: C-Library/`malloc()`/`brk()`
- *Uninitialized data*: global variables, initialized with all zeroes by the loader (mapping of the *zero* page)
- *Guard Page*



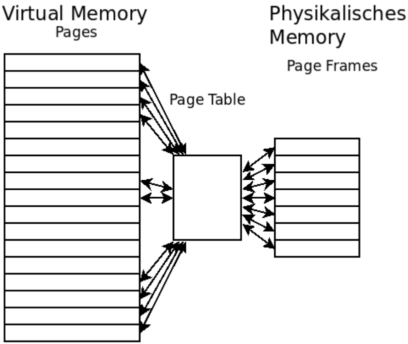
Wonderful reading: lwn.net/Articles/716603/

Virtual Memory

FASCHINGBAUER

Virtual memory

- Processes don't have *physically contiguous* memory
- Illusion "Linear Adress Space → Indirection
- *Page*: piece of virtual memory (4K)
- *Page Table*: per-process table of *allocated* pages

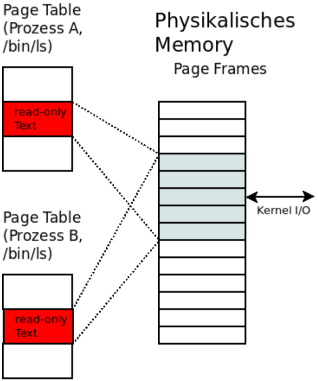


Shared Memory: "Text"

"Text": Code, executed by the CPU

- Multiple processes run the same program
- → text is *shared*
- text is not modified → *read-only*

→ *"Memory Mapping"*



Overview

FASCHINGBAUER

Memory Mappings (1)

FASCHINGBAUER

Memory Mapping: collection of contiguous pages

- Source
  - *File*. Mapped memory that is backed by a section of a file on disk.
  - *Anonymous*. Memory filled with all zeroes → `/dev/zero`.
- Visibility
  - *Shared*. Other process have access. Modification are persisted into the backing file (if any).
  - *Private*. Modifications are not persisted → *Copy-on-Write*.

Memory Mappings (2)

FASCHINGBAUER

Combinations and their meanings

- *Private File Mapping*: memory is initialized from the backing file. Copy-on-write.
- *Private anonymous Mapping*: memory allocation
- *Shared File Mapping*: modifications are visible for others, via the backing file $\rightarrow$ communication
- *Shared anonymous Mapping*: invisible for unrelated processes. `fork()` inherits mappings $\rightarrow$ memory shared with child processes.

Memory Mappings: Example (1)

FASCHINGBAUER

Once again: `proc/<PID>/maps`

```
$ cat /proc/self/maps
...
```

`r-xp`	`..`	`/bin/cat`	Text of cat
`r--p`	`..`	`/bin/cat`	Read-only data (constants)
`rw-p`	`..`	`/bin/cat`	writeable data (bss und initialized)
`rw-p`	`..`	`[heap]`	dynamically allocated memory (priv. anon.)
`rw-p`	`..`	`[stack]`	ditto

Memory Mappings: Example (2)

FASCHINGBAUER

`/lib/ld-linux.so.2` at work

```
$ strace ls
...
open("/lib/libc.so.6", O_RDONLY)        = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1...
mmap(NULL, 3508264, PROT_READ|PROT_EXEC, MAP_PRIV...
...
```

Pretty, isn't it?

Overview

FASCHINGBAUER

Creating Mappings: `mmap()` (1)

FASCHINGBAUER

`man 2 mmap`

```
void *mmap(
    void *addr,
    size_t length, int prot, int flags,
    int fd, off_t offset);
int munmap(void *addr, size_t length);
```

- Mapping backed by file `fd`, starting at `offset`, extending `length` bytes.
- `offset` and `length` should be a multiple of the page size (→ `man 2 getpagesize`).

Creating Mappings: `mmap()` (2)

Memory protection (`prot`). `SIGSEGV` when violated.

- `PROT_EXEC`
- `PROT_READ`
- `PROT_WRITE`
- `PROT_NONE`

Flags (`flags`):

- One of `MAP_SHARED`, `MAP_PRIVATE`
- `MAP_ANONYMOUS`

Flushing Mappings: `msync()`

FASCHINGBAUER

File mappings are *not* autmatically sync with the backing file (same with `write()`).

man 2 msync

```
int msync(void *addr, size_t length, int flags);
```

- `MS_SYNC`: wait until data is out on disk
- `MS_ASYNC`: don't wait

Locking: `mlock()`, `mlockall()`

File mappings need not be *resident* → can be loaded on-demand. Quite the opposite of what *realtime* is.

man 2 mlock

```
int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
int mlockall(int flags);
int munlockall(void);
```

- `MCL_CURRENT`. Lock current memory state into RAM
- `MCL_FUTURE`. Lock all that's to come.

Optimization Hints: madvise()

FASCHINGBAUER

Kernel is happy about hints on *how* the memory in the mapping will be used.

man 2 madvise

```
int madvise(void *addr, size_t length, int advice);
```

- MADV_SEQUENTIAL. Sequential access → read-ahead, freeing memory that has already been passed.
- MADV_RANDOM. Random access → no read-ahead.
- ...

Overview

FASCHINGBAUER

Overview

Inter Process Communication (IPC) (1)

Traditional Unix IPC: mechanisms to communicate between *unrelated* processes

- Semaphores
- Shared memory
- Message queues

Unrelated: not related via parent/child relationships

Inter Process Communication (IPC) (2)

History: two IPC variants ...

- *System V IPC*
  - Cumbersome, unnecessarily complex API
  - Older $\rightarrow$ more portable between Unixen
- *POSIX IPC*
  - Easy to use
  - Much of it implemented in userspace (through memory mapped files)
  - Optional feature in POSIX (fully supported in Linux though)

We're doing POSIX!

POSIX IPC: Overview

FASCHINGBAUER

IPC object names:

- System-wide visibility $\rightarrow$ just like files
- Consistently like so: `/some-object-name`

API:

- Semaphores, shared memory and message queues are opened just like files. E.g. `shm_open()`, using the same flags.
- Just like file descriptors, all types are reference counted.

Overview

Message Queues

FASCHINGBAUER

Message queue creation parameters:

- Maximum number of messages
- Maximum size of a single message
- → "Realtime guarantees"

Message priorities:

- Messages are sent with a priority
- Higher prioritized messages overtake lower prioritized messages

→ `man 7 mq_overview`

Open/Create: mq_open()

man 3 mq_open

```
mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode,
              struct mq_attr *attr);
```

In attr the only relevant members are mq_flags, mq_maxmsg and mq_msgsize.

Sending/Receiving: mq_send(), mq_receive()

FASCHINGBAUER

man 3 mq_send

```
int mq_send(mqd_t mqdes, const char *msg_ptr,
            size_t msg_len, unsigned msg_prio);
```

man 3 mq_receive

```
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
                   size_t msg_len, unsigned *msg_prio);
```

Closing/Removing: `mq_close()`, `mq_unlink()`

FASCHINGBAUER

man 3 mq_close

```
int mq_close(mqd_t mqdes);
```

man 3 mq_unlink

```
int mq_unlink(const char *name);
```

Analogy: `close()` and `unlink()`.

Notification: mq_notify()

FASCHINGBAUER

Notification: obscure feature, only shown because of its obscurity ...

man 3 mq_notify

```
int mq_notify(mqd_t mqdes, const struct sigevent *sevp);
```

Please read yourself and be disturbed!

Message Queues are Files

FASCHINGBAUER

Obvious implementation: (provided there's OS infrastructure)

- Message queues are implemented as files
- Virtual filesystem — `mqueue`

Notifications can be received more elegantly — `select()` und `poll()`!

Message Queue Filesystem: `mqueue`

FASCHINGBAUER

Message queues visible as files:

`mqueue` File System

```
# mount -t mqueue blah /mnt/mqueue
# ls -l /mnt/mqueue/my-queue
-rw------- ... /mnt/mqueue/my-queue
# cat /mnt/mqueue/my-queue
QSIZE:0          NOTIFY:0     SIGNO:0     NOTIFY_PID:0
```

Overview

Semaphores

FASCHINGBAUER

Creation parameter:

- Initial value

→ `man 7 sem_overview`

Open/Create: `sem_open()`

man 3 sem_open

```
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,
                mode_t mode, unsigned int value);
```

sem_wait(), sem_post()

FASCHINGBAUER

man 3 sem_wait

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(
    sem_t *sem,
    const struct timespec *abs_timeout);
```

man 3 sem_post

```
int sem_post(sem_t *sem);
```

Closing/Removing: `sem_close()`, `sem_unlink()`

FASCHINGBAUER

man 3 sem_close

```
int sem_close(sem_t *sem);
```

man 3 sem_unlink

```
int sem_unlink(const char *name);
```

Analogy: `close()` and `unlink()`.

Semaphores are Files

FASCHINGBAUER

- Implemented as file mappings
- `sem_t` encapsulates open file descriptor and a `void*` (the mapped memory)

`/dev/shm` is a `tmpfs` instance

Semaphore

```
$ ls -l /dev/shm/
total 1604
-rw------- ... sem.my-semaphore
```

Overview

FASCHINGBAUER

Shared Memory (1)

FASCHINGBAUER

POSIX shared memory is almost non-existing ...

- Small wrapper around existing system calls
- `shm_open()`. Does not even pretend to be something special → explicitly returns a file descriptor
- `shm_close()`

→ `man 7 shm_overview`

Shared Memory (2)

FASCHINGBAUER

Further steps:

- `ftruncate()`, to adjust the size
- `mmap()`, to create the mapping

The only reason for the `shm_*` is the "where" → `/dev/shm`

Overview

FASCHINGBAUER

Exercise: POSIX Message Queues

FASCHINGBAUER

Add a POSIX message queue to our server like follows

- The client (to be written) opens an existing message queue, sends a message, and closes the queue afterwards.
- The server ...
  - ... creates the message queue in the startup phase
  - ... receives (file descriptor based) notifications in the main loop, and reads and outputs messages just like the others
  - ... closes and removes the queue in the shutdown phase

Overview

Overview

FASCHINGBAUER

Shared Libraries - Basics

- Originally invented to replace static libraries
- Resource saving: static C library `libc.a` has around 4MB $\rightarrow$ contained in every single executable
- $\Longrightarrow$ identical code loaded in memory multiple times — once per executable
- *Shared libraries* are loaded in memory only once (code and read-only data)
- Semantics models that of static libraries

Shared Libraries - Problems

FASCHINGBAUER

- Executables don't bring the code that they have been linked against — rather, somebody else is responsible
- $\rightarrow$ mistakes happen
  - Missing libraries
  - Code compatibility ("DLL Hell")
  - ...
- Careful with C++ $\rightarrow$ one should know the language very well in order to prevent incompatibilities

Shared Libraries - Features

- Version control: different versions of the same library can co-exist
- Explicit modules loading ("plugins")

Overview

Shared Libraries - Building

FASCHINGBAUER

- “Position Independent Code” (PIC): same shared Library can be loaded at different addresses in different address spaces (processes)
- … done on purpose on most current systems (ASLR — Address Space Layout Randomization)

Shared library building

```
$ gcc -fPIC -c -o x.o x.c
$ gcc -shared -o libx.so x.o
```

Shared Libraries - Linking Against

FASCHINGBAUER

No difference here ...

- Use the library *base name*
- Linker prefers shared libraries over static libraries

Linking against shared libraries

```
$ gcc -c -o main.o main.c
$ gcc -o main main.o libx.so
# oder so:
$ gcc -o main main.o -L. -lx
```

Shared Libraries - Using (1)

FASCHINGBAUER

Executing is a bit harder ...

- Shared libraries aren't found easily
- Standard locations: `/lib`, `/usr/lib`, ...
- → Library must be *installed* there

```
$ ./main
$ ./main: error while loading shared libraries:
    libx.so: cannot open shared object file:
    No such file or directory
$ LD_LIBRARY_PATH=. ./main
```

Shared Libraries - Using (2)

FASCHINGBAUER

Shared library search path

1. `LD_PRELOAD` (ausser bei SUID/SGID)
2. `rpath` in der Shared Library selbst
3. `LD_LIBRARY_PATH` (ausser bei SUID/SGID)
4. `/etc/ld.so.conf` → `/etc/ld.so.cache`
5. `/usr/lib`
6. `/lib`

Shared Libraries - `rpath`

Compiled-in search path: `rpath`

- Executable is installed at some vendor-specific location (different from `/usr/bin` etc.)
    - Location known at build time
- One does not want to set `LD_LIBRARY_PATH` for some reason
- One does not want to edit `/etc/ld.so.conf` for some reason

```
$ gcc -Wl,-rpath,/some/funny/place -o main main.o libx.so
```

Shared Libraries - Dependencies

FASCHINGBAUER

Libraries and executables depend on libraries. Which ones?

DT_NEEDED

```
$ gcc -o main main.o libx.so
# oder so:
$ gcc -o main main.o -L. -lx
$ readelf --dynamic main
Tag        Type     Name/Value
0x00000001 (NEEDED) Shared library: [libx.so]
0x00000001 (NEEDED) Shared library: [libc.so.6]
```

- During linking, linker find the shared library that matches the *base name*
  - `-lsomething` → libsomething.so
  - More complicated though → demo time

Overview

FASCHINGBAUER

Explicit Loading - Overview

FASCHINGBAUER

Plugins: code is loaded at runtime, based on configuration or something ...

- Explicit code loading
- "Plugins"

Loader API, in the C library:

- `dlopen()`: load code from a file
- `dlsym()`: search a symbol (difficult with C++)
- `dlclose()`: close/unload
- `dlerror()`: determine error number after one occurred

Explicit Loading - `dlopen()` (1)

man 3 dlopen

```
void *dlopen(const char *filename, int flag);
```

- Loads a library, including all of its dependencies (if they aren't there already)
- `filename`: name of the library file. Path search rules as with automatic loading — except when there's a '/' in the name.

Explicit Loading - `dlopen()` (2)

FASCHINGBAUER

`flags` **are used to fine-tune behavior ...**

- `RTLD_NOW` *xor* `RTLD_LAZY`: symbols are resolved immediately (at load time), or when they are needed ($\rightarrow$ deferred error handling)
- `RTLD_LOCAL`: symbols not exported for subsequent `dlopen()` calls ("Loading Scope").
- `RTLD_GLOBAL`: the opposite of `RTLD_LOCAL`
- `RTLD_DEEPBIND`: symbols in a library are preferred over those that have been loaded previously $\rightarrow$ *self contained* libraries
  - *Careful*: default is to *not* prefer self containment

$\Longrightarrow$ Use `RTLD_LOCAL|RTLD_DEEPBIND` to load "plugin" shared objects

Explicit Loading - dlsym()

FASCHINGBAUER

man 3 dlsym

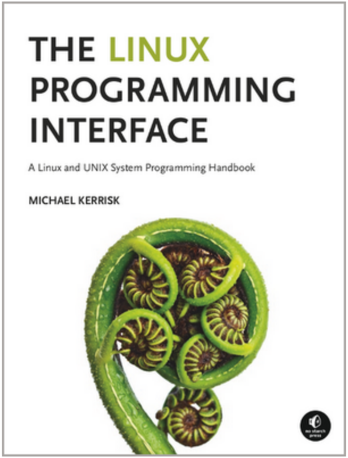
```
void *dlsym(void *handle, const char *symbol);
```

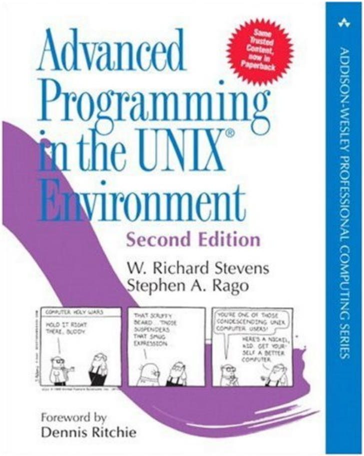
- Searches `symbol` (a C string) in library referred to by `handle`
  - `NULL` if not found
- Cast result to wanted function prototype
  - See manpage for an example

Overview

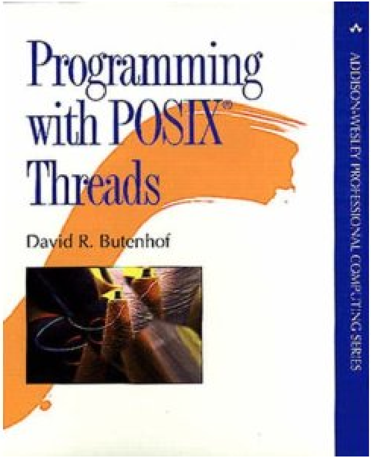
Overview

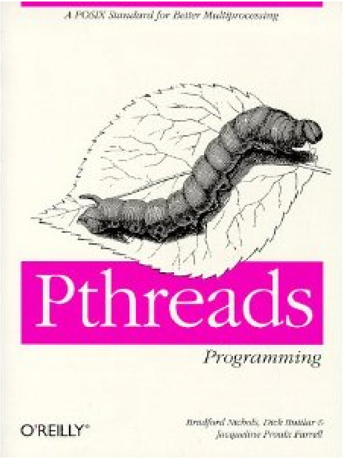
Linux/UNIX Userspace





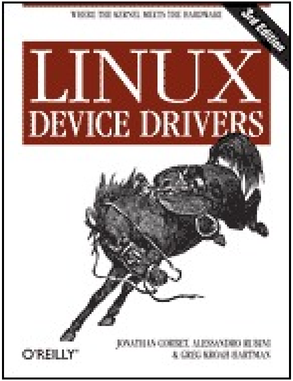
POSIX Threads

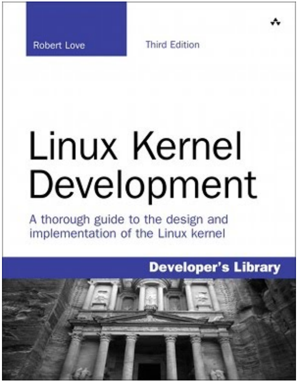


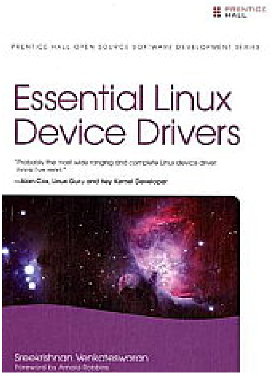


Kernel

FASCHINGBAUER







Overview

Summary

FASCHINGBAUER

We saw the good sides:

- File descriptors in all their beauty
- Processes, likewise
- Virtual memory, likewise

Fear is appropriate:

- Threads – fear is portable to other operating systems though
- Signals – fortunately there are ways other than traditional ones

There's More!

FASCHINGBAUER

Linux and Unix is a broad field. These topics could fill a couple more courses:

- File locking: locking models in the file system
- Permission system, and its Linux specific Extensions
- Pipes und FIFOs
- Shared libraries (there's more)
- Resource limits
- Linux containers
- ...

But: You Have A Basis!

FASCHINGBAUER

As always: if you have a big picture, and you understand the principles, then you can defend yourself against all that's to come.

With this in mind – ENJOY!!