

# Programmation Systèmes

## Cours 1 — Introduction

Stefano Zacchioli  
zack@pps.jussieu.fr

Laboratoire PPS, Université Paris Diderot - Paris 7

22 Septembre 2011

URL <http://upsilon.cc/zack/teaching/1112/progsyst/>  
Copyright © 2011 Stefano Zacchioli  
License Creative Commons Attribution-ShareAlike 3.0 Unported License  
<http://creativecommons.org/licenses/by-sa/3.0/>



- 1 What is system programming
- 2 UNIX concepts
- 3 System programming concepts
- 4 UNIX standards and implementations
- 5 About this course

# Outline

- 1 What is system programming
- 2 UNIX concepts
- 3 System programming concepts
- 4 UNIX standards and implementations
- 5 About this course

# Programming layered architectures

The architectures of modern computing systems are massively *layered*.

When programming, we target specific **layers**.

E.g.:

- 1 hardware level (firmware, microcode, 1GL, ...)
- 2 assembly level (assembly languages, interrupts, 2GL, ...)
- 3 **system level** (system languages, system calls, 3GL, ...)
- 4 **application level** (business-oriented, frameworks, 4GL, ...)
- n virtual architectures / virtual machines...

Each level is characterized by (or highly correlated with):

- mechanisms and APIs to interact with lower layers
- apt programming languages (and their generations)

# Which layer to target

The choice of layer reveals important trade-offs.

- 1 **Performances.** Targeting a **lower layer** might grant **better performances**.

Writing a performance critical routine in assembly might provide a several order of magnitude speed improvement when compared to programming targeting higher layers.

E.g. this technique is used for performance critical code such as device drivers, crypto-code, etc.

## Which layer to target (cont.)

The choice of layer reveals important trade-offs.

- ② **Portability**. Targeting a **higher layer** usually guarantees **better portability**, in particular better than all lower layer equivalents that might be *generated* from the chosen layer.

E.g.: a block of standard ISO C 99 code can be compiled using gcc to more than 70 different target processors.

## Which layer to target (cont.)

The choice of layer reveals important trade-offs.

- ③ **Maintainability.** Targeting a **higher layer** usually makes writing code easier and the resulting **code more maintainable**.

This is largely a consequence of the involved programming languages.

# System programming

*System programming is the art of writing system software.*  
— Robert Love

**System software** is “low level” software that interfaces *directly* with:

- the kernel of the operating system
- core system libraries (we'll be more precise in a bit)

# System programs — examples

Some examples of system software around you:

- shell
- compilers
- interpreters
- debuggers
- text editors
- system services
  - ▶ cron
  - ▶ print spool
  - ▶ power mgmt
  - ▶ session mgmt
  - ▶ backup
- network services
  - ▶ HTTP servers
  - ▶ MTA
  - ▶ DBMS
  - ▶ ...

Try `ps -auxw`, *most* of it is system-level software.

# System programming — why bother?

- there are *drawbacks* in targeting the system level (performances, maintainability, portability, etc.)
- recent years have witnessed a *shift from system programming to application programming*
  - ▶ platforms such as Java and .NET, as well as 4GL and 5GL languages, hide the system level to the programmer, in the quest for the “run everywhere” mantra
  - ▶ many programmers spend most—if not all—of their time doing *application* programming

Why should you bother learning system programming?

?

# Why system programming

- 1 many programming tasks are **inherently system-level**
  - ▶ e.g. system services, toolchain, etc.
- 2 **new system-level tasks** born on a regular basis, to cope with application-level evolution. E.g.:
  - ▶ an increasing number of new applications is written in JavaScript, both for the Web and *for desktops* (!)
  - ▶ *therefore* we need new and better JavaScript (JIT) compilers; most of their code is system-level code

## Why system programming (cont.)

- 3 **legacy** code—such as system utilities—is not going away any time soon; in some cases it is also basis for standardization (e.g. UNIX utilities)
  - ▶ in the Free Software world, the majority of *existing code* (50%+ of Debian) is system-level C code (Michlmayr et al. 2009)
- 4 even **application-level programming** benefits a great deal from system programming knowledge
  - ▶ understanding system behavior and performance bottlenecks
  - ▶ deciding when to drop-down at the system level
  - ▶ debugging portability issues
  - ▶ ...

# UNIX system programming

This course is about UNIX system programming.

We address system programming in UNIX® systems as well as “UNIX-like” implementations (e.g. Linux, FreeBSD, etc.), following various UNIX-related standards.

This is **not an introductory course** about UNIX system programming

- prerequisites:
  - ▶ UNIX proficiency as a user
  - ▶ topics covered by course “Systèmes” L3
- we review today and in the first TDs *some* of that material:
  - ▶ UNIX concepts
  - ▶ UNIX syscalls for I/O
  - ▶ system programming concepts

# Outline

- 1 What is system programming
- 2 UNIX concepts**
- 3 System programming concepts
- 4 UNIX standards and implementations
- 5 About this course

# Operating systems in a nutshell

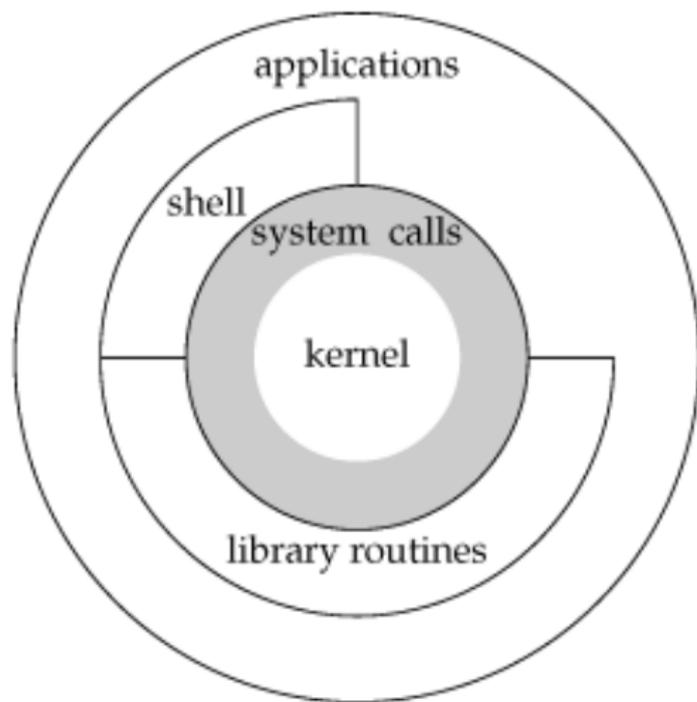
An **operating system** is the software environment that provide **services** needed to run final user programs. E.g.:

- program execution
- hardware access (e.g. read from disk, play a sound)
- file system access (e.g. open, close, read, write a file)
- memory access (e.g. allocation, memory mapping)
- network access (e.g. connect to a server, wait for connections)

The **kernel** of an operating system is the software layer that *control the hardware* and create the *environment* in which programs can run.

The kernel layer is usually thin—when compared to an entire operating system—and self-contained.

# UNIX architecture



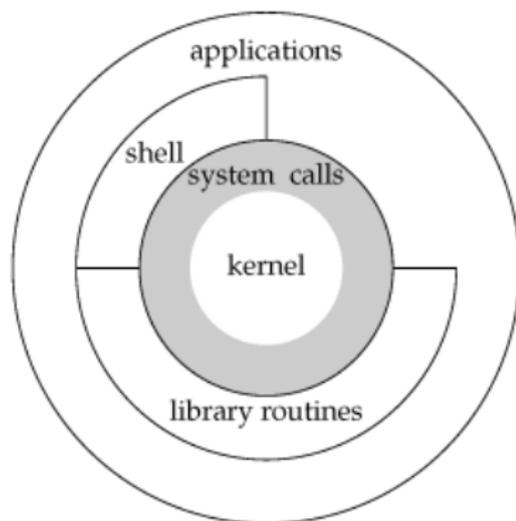
APUE, Figure 1.1

- layered architecture, with a UNIX kernel at its core
- not all layers are strictly encapsulated

# UNIX architecture — some details

The **kernel** takes care of:

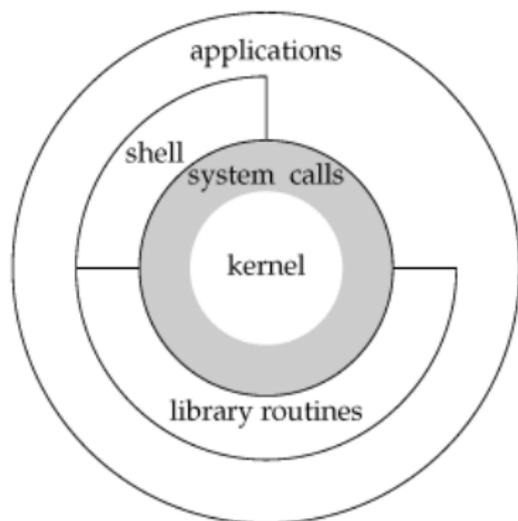
- *process scheduling*
  - ▶ highlight: preemptive multitasking
- *memory management*
  - ▶ highlight: virtual memory and process isolation
- *user management*
  - ▶ highlight: multiuser virtual private computers
- *file system provision*
- *process management*
- *access to devices*
- *networking*
- *system call API provision*



## UNIX architecture — some details (cont.)

**System calls** provide the interface (API) for programs to access kernel services.

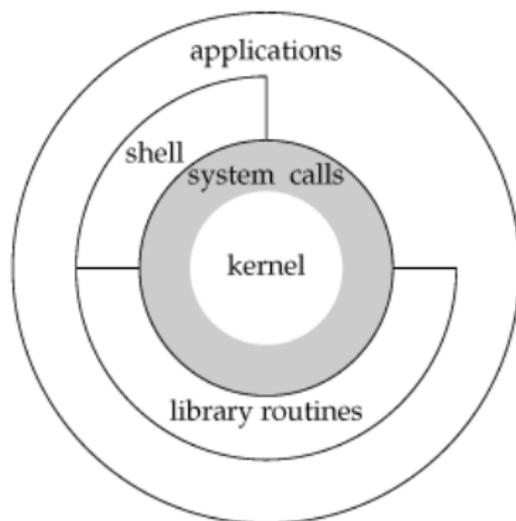
- operating systems before UNIX used to define the **system call API** in assembly, UNIX started doing so in C
- the *implementation* of system calls is part of the kernel code (AKA “**kernel-space** code”)
  - for now, we assume that we can invoke system calls as if they were ordinary C functions



## UNIX architecture — some details (cont.)

Core system libraries (“library routines” in figure) implement basic functionalities needed by almost all programs.

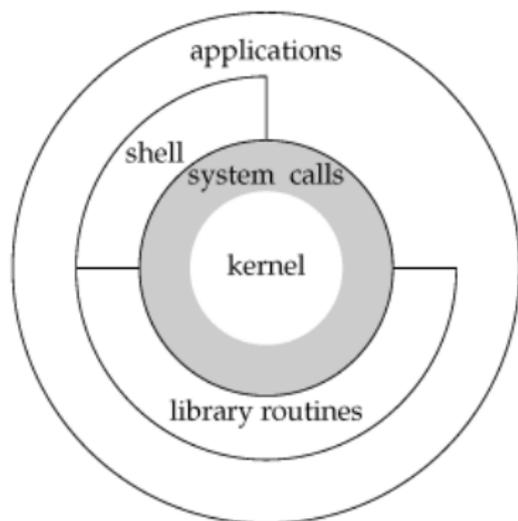
- E.g.:
  - ▶ buffered I/O
  - ▶ fine-grained memory allocation
  - ▶ time management
- those functionalities are usually implemented “lifting” system call API to richer interfaces
- the implementation is *not* part of the kernel code (AKA “**user-space** code”) and hence can be replaced more easily (but still. . .)



## UNIX architecture — some details (cont.)

The **shell** is a specific application used interactively by system users to start and control programs.

- historically, shells have been—and still are, for power users—an interactive equivalent of the system call API
- the advent of higher-level wrappers to start applications (e.g. desktop environments) have partially replaced shells. Arguably they are just another type of shells

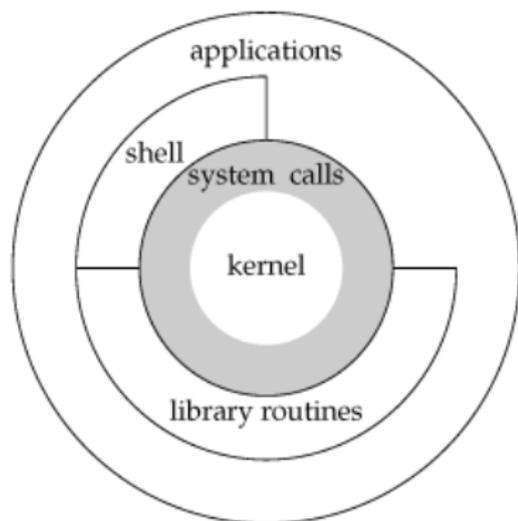


## UNIX architecture — some details (cont.)

**Applications** are the programs typically used for the productivity of the final user.

Note how applications can be built accessing various layers of what's under them:

- kernel services (via the system call API)
- core system library routines
- the shell
- other intermediate libraries (not shown)



# Logging in

- UNIX sessions start with a user of the system **logging in**, by inserting username and password
- the **user database** (usually /etc/passwd) contains information related to the login process, e.g.:

```
zack:x:1000:1000:Stefano Zacchioli , , ,:/home/zack:/bin/bash
```

- the **shell of choice** of the user (last field) is executed by the login process after authenticating the user
- all other user activities during the session stem from the shell process

# Filesystem

## UNIX directory structure:

```
/
/bin
/sbin
/etc
/dev
/home
/mnt
/lib
/root
/tmp
/usr
  /usr/bin
  /usr/include
  /usr/lib
  /usr/local
/var
  /var/log
  /var/mail
  /var/spool
  /var/tmp
/proc
/opt
/media
/srv
/boot
/sys
```

- hierarchical file system with a single root
- “*everything is a file*” mantra
- a directory is a file mapping names to (nameless) files
- (regular) file = data + metadata:
  - ▶ type
  - ▶ permissions
  - ▶ size
  - ▶ owner
  - ▶ ...

relevant syscalls: stat, fstat

# Filenames

- names within directories are called **filenames**
- syntax: all characters permitted except NULL and “/”
  - ▶ “/” is reserved as *directory separator*
  - ▶ filenames cannot span multiple directories
- special filenames
  - “.” **current directory** (wrt containing directory)
  - “..” **parent directory** (according to filesystem tree structure)

# Pathnames

## Definition (Pathname)

sequence of filenames separated by, and optionally starting with, “/”

- absolute path: a pathname starting with “/”
- relative path: all other paths

How can the kernel find the file corresponding to “/usr/lib/ocaml/pcre/pcre.mli”?

?

# Pathname resolution

## idea (OCaml-like code)

```
List.fold_left
  (fun cur_file name ->
    if not (is_dir cur_file) then
      raise Invalid_path;
    try
      List.assoc name (dir_content (opendir cur_file))
    with Not_found -> raise File_not_found)
  root_dir      (* needed to bootstrap; known by the kernel *)
  path         (* e.g. ["usr"; "lib"; "ocaml"; "pcr"; "pcr.mli"] *)
```

- pathname resolution is **performance critical**
  - ▶ it's therefore heavily cached in most kernels
- how to bootstrap pathname resolution for *relative* pathnames?
  - ▶ thanks to the notion of **working directory**
  - ▶ set to user *home directory* upon login
  - ▶ can be explicitly changed by processes

## Filename and pathnames — syscalls

- pathname resolution is performed *implicitly* by the kernel for every syscall that takes a pathname argument
- *explicit* directory browsing: `opendir`, `closedir`, `readdir`, `rewinddir`
  
- working directory manipulation: `chdir`, `fchdir`

# File descriptors

## Definition (file descriptor)

A **file descriptor** (fd) is a small non-negative integer used by the kernel to reference a file used by a process.

- fd are unique only within a process
- $\langle \text{fd}, \text{process} \rangle$  pairs act as keys to reference internal kernel data structures

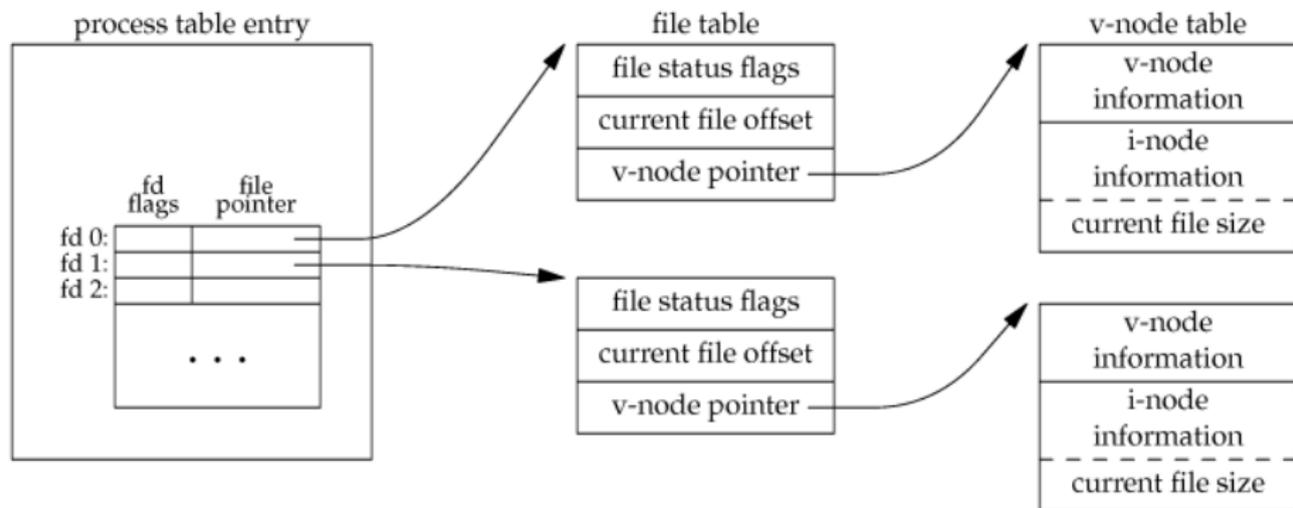
Typical “**file descriptor protocol**”:<sup>1</sup>

- 1 each time the kernel opens/creates a file for a process, it returns a file descriptor to it
- 2 subsequent actions on that file requires that the process passes the corresponding file descriptor back to the kernel

---

<sup>1</sup>According to the “everything is a file” UNIX mantra, this protocol is used for way more than regular file manipulations

# Open files and the kernel



APUE, Figure 3.6

- each process file descriptor points to an in-kernel **file table entry**
- each file table entry points to in-kernel equivalent of on-filesystem file information and associated metadata
  - ▶ in particular: the current **file offset**

## Standard file descriptors

Due to how process creation works on UNIX, shells are de facto responsible to setup part of the initial environment of new processes. To that end, shells conventionally open 3 standard file descriptors upon process creation:

**standard input** default fd where to *read input* from

**standard output** default fd where to *write output* to

**standard error** default fd where to *write error* output to

Typical values for standard file descriptors are given in `<unistd.h>`:

```
/* Standard file descriptors. */  
#define STDIN_FILENO 0 /* Standard input. */  
#define STDOUT_FILENO 1 /* Standard output. */  
#define STDERR_FILENO 2 /* Standard error output. */
```

# Redirections

By default, all standard file descriptors are connected to the terminal.

Shells offer mechanisms to redirect them to/from other resources:

- files
- processes
- other file descriptors
- in-line data (e.g. “here documents”)
- ...

see `man 1 bash`, section “REDIRECTION”

# Redirections

By default, all standard file descriptors are connected to the terminal.

Shells offer mechanisms to redirect them to/from other resources:

- files
- processes
- other file descriptors
- in-line data (e.g. “here documents”)
- ...

see `man 1 bash`, section “REDIRECTION”

## Demo

# File descriptors — syscalls

- file opening: `open`
- new file creation: `creat`
- file (descriptor) closing: `close`
- file descriptor manipulation: `dup`, `dup2`, `fcntl`, ...

# Unbuffered I/O

syscalls are available for basic, unbuffered,<sup>2</sup> I/O:

- **read** content from file to memory: read
- **write** content from memory to file: write

Read and write operations are **chunked**.

Every operation implicitly move the file offset by the amount of data read/written. Explicit displacement of the file offset is provided by the `lseek` syscall.

---

<sup>2</sup>actually, there is *some* buffering, but it happens in kernel-space; this kind of I/O is better defined “user-space unbuffered”

## Unbuffered I/O — example

Sample cat implementation using unbuffered I/O:

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

#define BUFFSIZE      4096
int main(void) {
    int    n;
    char   buf[BUFFSIZE];
    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n) {
            perror("write error");
            exit(EXIT_FAILURE);
        }
    if (n < 0) {
        perror("read error");
        exit(EXIT_FAILURE);
    }
    exit(0);
}
```

## Unbuffered I/O — example (cont.)

- chunking forces us to loop and fix a buffer size
- thanks to implicit moves, no explicit file offset accounting is needed
- data is copied to/from user/kernel-space at each read/write

# Programs and processes

## Definition (programs and processes)

- A **program** is an executable file residing on the filesystem.
- A **process** is an executing instance of a program.

Note: several different process instances of the same program might exist in memory at the same time.

- **process creation** is requested to the kernel using a syscall of the exec family
  - ▶ passing a pathname pointing to a program as an argument
- process creation *reads the program into memory and start executing it*
- each process is associated to a numeric **process ID**

## Programs and processes — demo

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

int main(int argc, char **argv) {
    printf("hello, world from process %d\n", getpid());
    exit(EXIT_SUCCESS);
}
```

```
$ gcc -Wall -o hello-pid hello-pid.c
$ ./hello-pid
hello, world from process 21195
$ ./hello-pid
hello, world from process 21196
$ ./hello-pid
hello, world from process 21199
```

# Process address space

Each process executes by default in its own **address space** and cannot access the address spaces of other processes — barring a **segmentation fault** error.

The memory corresponding to a process address space is allocated to the process by the kernel upon process creation. It can be extended during execution.

The address space of a program *in execution* is partitioned into parts called **segments**.

# Segments

**text segment** machine instructions that the CPU executes. It is read from disk upon process creation. The instruction pointer points within this segment.

**initialized data segment** (“data segment”) global variables explicitly initialized, e.g.:

```
int magic = 42; // outside any function
```

**uninitialized data segment** (“bss segment”) global variables not explicitly initialized, e.g.:

```
char crap[1024]; // outside any function
```

- doesn't take any space in the on-disk binary
- it will be initialized by the kernel at 0 / NULL
- it can be initialized efficiently using *copy-on-write*

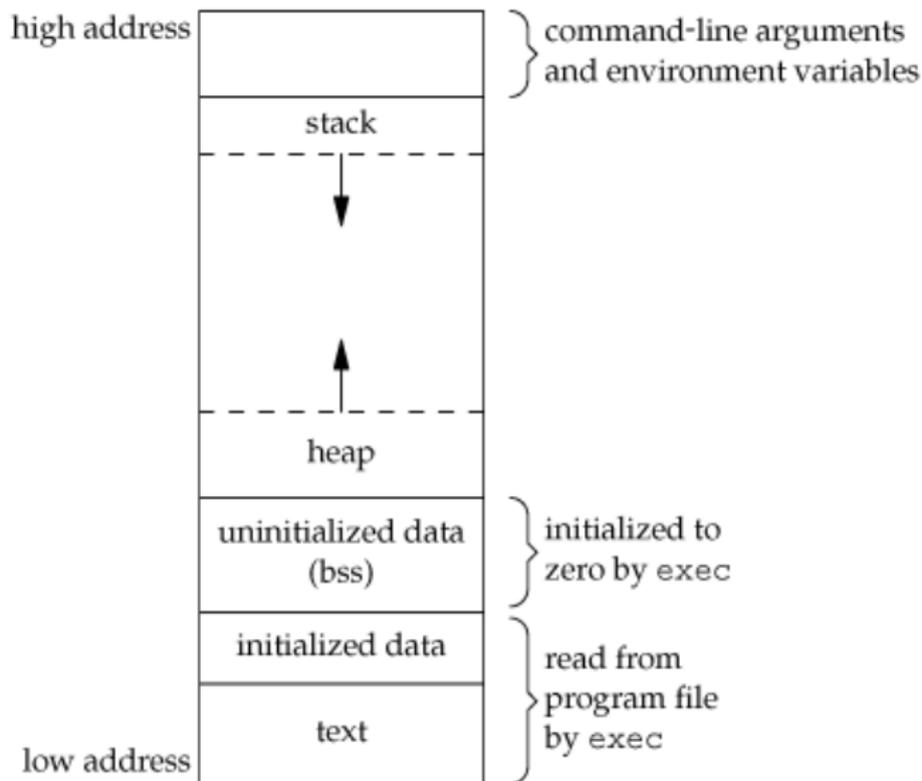
## Segments (cont.)

**stack** automatic variables, used to implement C calling convention

- can also be used for small allocations; see `alloca`

**heap** for dynamic memory allocation; see `malloc` and friends

# Typical segment arrangement



APUE, Figure 7.6

## Segment arrangement — demo

```
#include <stdio.h>
#include <stdlib.h>

int magic = 42;
char crap[1024];

void func(int arg) {
    printf("stack segment near\t%p\n", &arg);
}

int main(int argc, char **argv) {
    char *ptr;
    ptr = malloc(1);
    func(42);
    printf("heap segment near\t%p\n", ptr);
    printf("bss segment near\t%p\n", crap);
    printf("text segment near\t%p\n", &magic);

    free(ptr);
    exit(EXIT_SUCCESS);
}
```

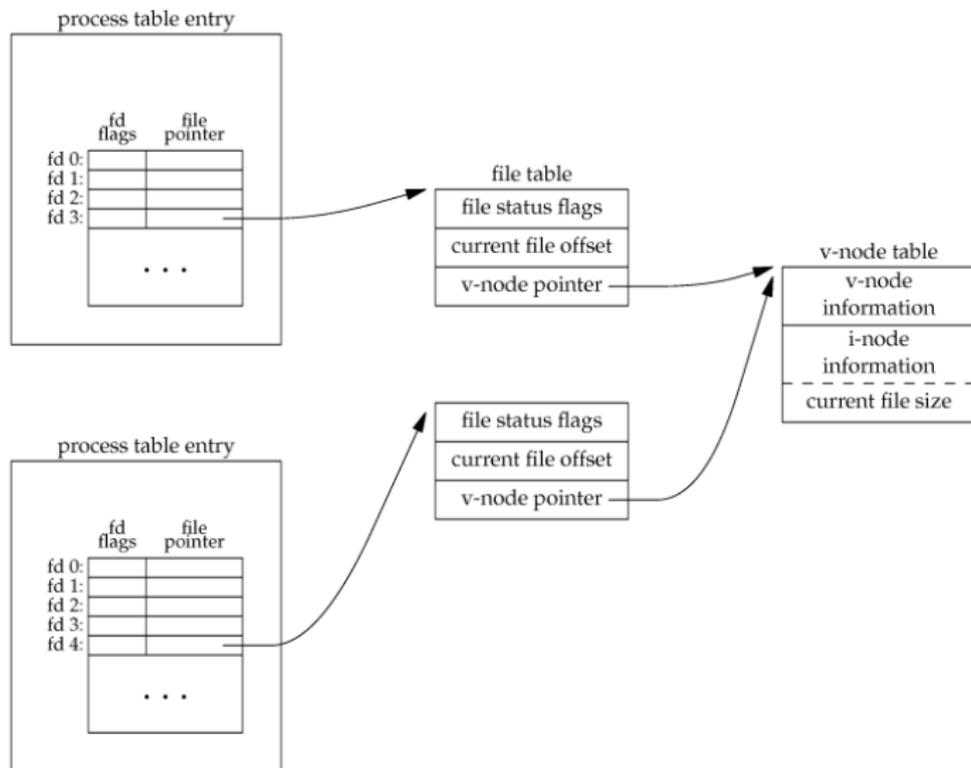
## Segment arrangement — demo (cont.)

```
$ ./segments
stack segment near 0x7ffff53ecccc
heap segment near 0x      1c52010
bss segment near 0x      600b00
text segment near 0x      600ad0
$
```

(output edited for alignment)

# Multi-process I/O — file sharing

What happens when *independent* processes act on the same file?



APUE, Figure 3.7

# Sharing resources among processes

It seems processes can share resources, such as the v-node table in the previous (degenerate) example.

Can they share more?

# Sharing resources among processes

It seems processes can share resources, such as the v-node table in the previous (degenerate) example.

Can they share more?

Yes.

- related processes can do more and share **file table entries**
- processes can share **specific memory regions** of their own address space (e.g. memory mapping, shared memory)
- pushing it to the extreme, multiple “processes” can share by default their **entire address space** by the means of threads

# Threads

By default, each process has only one **thread of control** (or “thread”), i.e. only one set of instructions being executed at any given time.

Extra threads can be added at runtime.

- pro** concurrent work on common data, without having to pass data around or setup shared memory regions; all data is “shared by default”
- pro** different threads can run in parallel on multiprocessor / multicore systems
- con** synchronization issues to avoid memory corruption; they might get very intricate (in non- purely-functional programming paradigms)

## Threads (cont.)

All threads within a process **share**:

- address space
- file descriptors
- stacks

(note the plural)

Each thread has **its own**:

- thread ID
- stack
- processor status
- instruction pointer
- thread-local storage

(unique only within the owning process)

(but others' stacks can be *accessed* !)

(to be requested explicitly)

# Outline

- 1 What is system programming
- 2 UNIX concepts
- 3 System programming concepts**
- 4 UNIX standards and implementations
- 5 About this course

# System calls

## Definition (system call)

A **system call** is a controlled entry point into the kernel, used by programs to request a service.

- during syscall execution, the processor state changes from user mode to kernel mode → so that protected kernel memory can be used
- the set of available system calls is fixed for a given platform; each syscall is identified by a unique number
- each system call accepts arguments and possibly return values, bridging user space and kernel space

For security reasons, syscalls code (kernel space) is not linked directly with user space. How can C code invoke syscalls then?

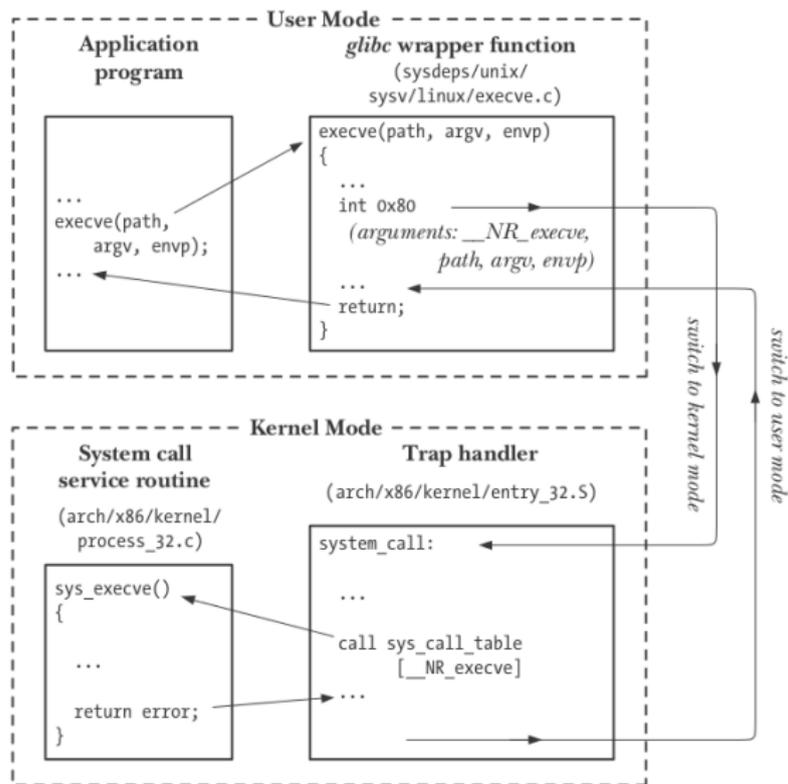
# System call invocation

- 1 program makes a syscall by invoking a **wrapper function** provided by the **standard C library**
- 2 before actual syscall invocation, arguments shall be put in specific registers; the wrapper fill those registers copying from user space
- 3 the wrapper fills a predefined register with the syscall number
  - ▶ %eax on x86-32 architectures
- 4 the wrapper executes a **trap** machine instruction
  - ▶ 0x80 on x86-32
- 5 the kernel invoke its **syscall dispatcher** routine

## System call invocation (cont.)

- 6 the syscall dispatcher
  - 1 saves processor status on the kernel stack
  - 2 **looks up** the syscall code in its syscall table
  - 3 **executes** the syscall code, passing and returning arguments via the kernel stack
  - 4 restore processor status
  - 5 put on the *process* stack the syscall return value
  - 6 return to the wrapper function
- 7 if the return value of the syscall dispatcher indicates an error, the wrapper function **sets `errno`** to the error value

# System call invocation — example



TLPI, Figure 3-1

# System call invocation — putting it all together

All this seems pretty complicated.

When developing and debugging how can we know which is which?

E.g.: `is foo(42)`

- a syscall?
- a wrapper from the standard C library?
- another user-space library function?

# strace

To understand the interaction among user- and kernel-level code, and the role played by system calls, *experimenting with existing programs* is invaluable.

**strace** allows to trace syscall invocations.

From the `strace(1)` manpage:

```
strace - trace system calls and signals
```

```
strace [ command [ arg... ] ]
```

[...] `strace` runs the specified command until it exits. It **intercepts** and records the **system calls** which are called by a process [...]. The **name** of each system call, its **arguments** and its **return value** are **printed on standard error** [...].

`strace` is a useful diagnostic, instructional, and debugging tool. [...] Students, hackers and the overly-curious will find that a great deal can be learned about a system and its system calls by tracing even ordinary programs. [...]

## A “Hello, World!” journey

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    printf("hello, world\n");
    exit(EXIT_SUCCESS);
}
```

- what system calls are invoked by the famous hello world?
- what are the respective roles of user- and kernel-level code?

## A “Hello, World!” journey

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    printf("hello, world\n");
    exit(EXIT_SUCCESS);
}
```

- what system calls are invoked by the famous hello world?
- what are the respective roles of user- and kernel-level code?

Let's strace it...

# A “Hello, World!” journey (cont.)

```
$ strace ./hello
execve("./hello", [ "./hello" ], [ /* 51 vars */ ]) = 0
brk(0) = 0x1c25000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f734db26000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=143995, ...}) = 0
mmap(NULL, 143995, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f734db02000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\300\357\1\0\0\0\0\0"... , 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1570832, ...}) = 0
mmap(NULL, 3684440, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f734d585000
mprotect(0x7f734d6ff000, 2097152, PROT_NONE) = 0
mmap(0x7f734d8ff000, 20480, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x17a00) = 0x7f734d8ff000
mmap(0x7f734d904000, 18520, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f734d904000
close(3) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f734db01000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f734db00000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f734daff000
arch_prctl(ARCH_SET_FS, 0x7f734db00700) = 0
mprotect(0x7f734d8ff000, 16384, PROT_READ) = 0
mprotect(0x7f734db28000, 4096, PROT_READ) = 0
munmap(0x7f734db02000, 143995) = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 5), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f734db25000
write(1, "hello, world\n", 13) = 13
exit_group(0) = ?
```

## A “Hello, World!” journey (cont.)

Here is a tiny teeny part of the journey, annotated with actors:

- the *shell* uses the `execve` system call (“syscall” for short from now on) to execute our program
- the *kernel* reads the object code from the “hello” binary program and create a matching process

...

- the “hello” *process* invokes the `printf()` function from the core system library
- the *core system library* (in this specific case the GNU `libc`, AKA “glibc”) invokes the `write` system call to print on the screen
- the *kernel* print on the screen

...

- (process) invokes `exit()`
- (glibc) invokes the `exit_group()` system call
- (kernel) terminates the process

## Manual sections help too

When looking at *code*, help to understand where specific functions are implemented might come from **sections** of the **UNIX programming manual**.

*The table below shows the section numbers of the manual followed by the types of pages they contain.*

- 1 Executable programs or shell commands
- 2 System calls (functions provided by the kernel)
- 3 Library calls (functions within program libraries)
- 4 Special files
- 5 File formats and conventions
- 6 Games
- 7 Miscellaneous
- 8 System administration commands
- 9 Kernel routines [Non standard]

— man(1)

# Error handling

## Error checking mantra

Thou shalt always check the return code of system calls for error.

Usually, when such an error occurs the following happens:

- 1 the syscall wrapper function returns a **negative value**
  - ▶ or NULL, for functions returning pointers
- 2 errno is set to further *explain* the error (why?)

# Error handling

## Error checking mantra

Thou shalt always check the return code of system calls for error.

Usually, when such an error occurs the following happens:

- 1 the syscall wrapper function returns a **negative value**
  - ▶ or NULL, for functions returning pointers
- 2 errno is set to further *explain* the error (why?)

## Example

open returns -1 upon failure, but there are about 15 possible different reasons for the failure. errno discriminates among them.

# errno

<errno.h> defines:

- the `errno` symbol
- `constants` (all starting with E. . .) corresponding to error classes, which can be compared with `errno` for equality

Some examples from `errno(3)`:

- `EACCES` permission denied
- `EAGAIN` resource temporarily unavailable
- `EBUSY` device or resource busy
- `EINTR` interrupted function call
- `EINVAL` invalid argument
- `ENOENT` no such file or directory
- `ENOSPC` no space left on device
- `EPRM` operation not permitted
- `EPIPE` broken pipe

sounds familiar?

## errno — tips and pitfalls

- errno is usually believed to be global and unique, but it's actually **thread local** for multi-threaded processes
  - ▶ allow to have thread-local error contexts
  
- errno is *not* cleared by functions that do *not fail*; the previous value, possibly erroneous, remains
  - ▶ you should check errno *only if an error has actually occurred*

## errno — tips and pitfalls (cont.)

- the following code is b0rked:

```
if (somecall() == -1) {  
    printf("somecall() failed\n");  
    if (errno == E...) { ... }  
}
```

**Why?**

## errno — tips and pitfalls (cont.)

- the following code is b0rked:

```
if (somecall() == -1) {  
    printf("somecall() failed\n");  
    if (errno == E...) { ... }  
}
```

*Many functions* set `errno` upon failure, so we might be checking the `errno` of someone else than `somecall()`.

The fix is to “backup” `errno` to a separate variable and *test the saved value* against `<errno.h>` constants:

```
if (somecall() == -1) {  
    int errsv = errno;  
    printf("somecall() failed\n");  
    if (errsv == E...) { ... }  
}
```

## Interlude — helper functions

To keep examples short, we'll introduce various helper functions (or simply “helpers”) overtime. Here are is the first one:

```
#include <stdio.h>
#include <stdlib.h>

void err_sys(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}
```

- `perror` print a given error message together with a human readable version of `errno` (message + “: ” + `errno` description)
  - ▶ this is why `errno` descriptions sounded familiar. . .

For compatibility with the APUE book, we'll **#include** “`apue.h`” when using helpers.

## Helper functions — example

```
#include <unistd.h>
#include "apue.h"

#define BUFSIZE 4096

int main(void) {
    int n;
    char buf[BUFSIZE];

    while ((n = read(STDIN_FILENO, buf, BUFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");
    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

# The standard C library

We've seen that the standard C library (“libc” for short) contains syscall wrappers. It contains much more than that.

- many libc functions do not use syscalls at all
- some libc functions just lift syscalls API to handier APIs
  - ▶ e.g.: **time** and timezone management
- some libc functions performs substantial extra work
  - ▶ **memory allocation**
    - ★ syscall-level: `sbrk` (it just *moves* address space boundary)
    - ★ libc-level: `malloc/free` (bookkeeping of allocated blocks)
  - ▶ **standard I/O**: buffering, higher-level operations (e.g. read a *line*)

Various libc implementations are available, popular ones:

- glibc — the GNU C library — [www.gnu.org/software/libc/](http://www.gnu.org/software/libc/)
- eglibc — the Embedded GLIBC — [www.eglibc.org](http://www.eglibc.org)

## Standard I/O — example

Same example, with C (buffered) **standard I/O**:

```
#include "apue.h"
#define BUFSIZE 4096

int main(void) {
    char buf[BUFSIZE];
    while (fgets(buf, BUFSIZE, stdin))
        if (fputs(buf, stdout) == EOF)
            err_sys("fputs error");
    exit(0);
}
```

note the **double copy** phenomenon:

- at each loop iteration data is copied to/from internal buffers of the C standard library implementation
- upon buffer flushing, read/write are used

# Which libc am I using?

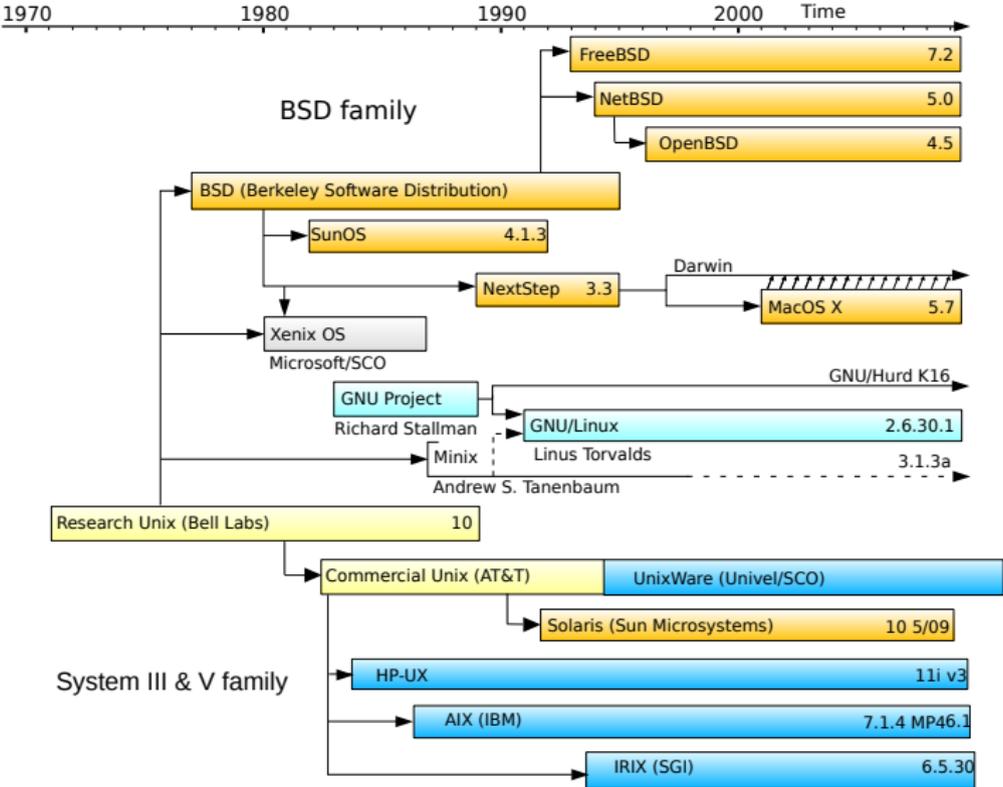
```
$ ldd /bin/ls | grep libc  
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fb406fbc000)
```

```
$ /lib/x86_64-linux-gnu/libc.so.6  
GNU C Library (Debian EGLIBC 2.13-21) stable release version 2.13,  
by Roland McGrath et al.  
Copyright (C) 2011 Free Software Foundation, Inc.  
This is free software; see the source for copying conditions.  
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A  
PARTICULAR PURPOSE.  
Compiled by GNU CC version 4.4.6.  
Compiled on a Linux 3.0.0 system on 2011-09-13.  
Available extensions:  
  crypt add-on version 2.1 by Michael Glad and others  
  GNU Libidn by Simon Josefsson  
  Native POSIX Threads Library by Ulrich Drepper et al  
  BIND-8.2.3-T5B  
libc ABIs: UNIQUE IFUNC  
For bug reporting instructions, please see:  
<http://www.debian.org/Bugs/>.
```

# Outline

- 1 What is system programming
- 2 UNIX concepts
- 3 System programming concepts
- 4 UNIX standards and implementations**
- 5 About this course

# UNIX genealogy



[http://en.wikipedia.org/wiki/File:Unix\\_history.svg](http://en.wikipedia.org/wiki/File:Unix_history.svg)

# UNIX standardization

Portability across UNIX-es has historically been quite good. During the **UNIX wars** of the 80s, the situation started getting worse.

Due to that, many high-profile users—including the US government—have heavily pushed for **UNIX standardization**.

Many (competing) standards ensued.



The C standard defines:

- **syntax** and **semantics** of the C language
- the C **standard library**

Relevant timeline:

1989 approved by ANSI (“**ANSI C**”)

1990 approved by ISO, unchanged

1999 updated and approved by ISO (“**C99**”)

- new keyword **restrict** for pointer declarations. Inform the compiler that the object referenced by a pointer is accessible *only via that pointer* within the containing scope (i.e. no pointer aliasing)
- not yet fully supported by compilers; see <http://en.wikipedia.org/wiki/C99#Implementations>

## “Portable Operating System Interface”

- family of UNIX-related standards by IEEE, updated overtime
- notion of “POSIX compliance”, that has worked pretty well

### Relevant timeline:

1988 IEEE 1003.1 — syscall API

1990 approved by ISO with the name “POSIX.1”, unchanged

1993–2000 updated by IEEE, **real-time extensions**

1996 updated by ISO, includes **pthread** (“POSIX threads”)

2001 great **merger**, combines several standards

- ISO/IEEE POSIX branches
- shell and utilities
- ISO C standard library

# Single UNIX Specification (SUS)

Initially a superset of POSIX.1, specifying additional *optional* interfaces known as X/Open System Interface (XSI). E.g.:

- encryption
- real-time threads
- XSI STREAMS
- ...

SUS defines extra interfaces and also “annotates” all POSIX.1 interfaces as either mandatory or optional for **XSI conformance**.

## The UNIX® trademark

The UNIX trademark, owned by Open Group, uses SUS as a criteria to define “UNIX systems”. To be called “**UNIX system**”, a system must pass XSI conformance.

# Some UNIX(-like) implementations

- **UNIX System V** (SysV) proprietary UNIX by AT&T (now SCO)
  - ▶ release 4 conformed to both POSIX 1003.1 and SUS
- **BSD** (Berkeley Software Distribution)
  - ▶ now evolved into FreeBSD / NetBSD / OpenBSD
  - ▶ origin of the liberal licensing movement
- **Linux**, started in 1991 by Linus Torvalds
  - ▶ nowadays the most popular UNIX-like system
  - ▶ considered to be both POSIX.1 and SUSv4 compliant
    - ★ no formal conformance though, due to the distribution model
- **Mac OS X** mixture of Mach kernel and FreeBSD
- **Solaris** UNIX system by Sun Microsystems (now Oracle)
  - ▶ historically proprietary, mixed fortune in open sourcing

# Outline

- 1 What is system programming
- 2 UNIX concepts
- 3 System programming concepts
- 4 UNIX standards and implementations
- 5 About this course**

# Objectives

Learn UNIX system programming concepts and core APIs.  
Learn how to learn more.

Specific topics:

- process manipulation
- signal manipulation
- process relationships
- inter process communication (IPC)
  - ▶ shared file descriptors
  - ▶ file locking
  - ▶ pipes, FIFOs, UNIX domain sockets
  - ▶ shared memory
  - ▶ semaphores
- multi-threading
- (terminal control)

# General info

## Équipe pédagogique

- chargé de cours: Stefano Zacchioli
- chargé de TD-TP: Gabriel Kerneis

## Horaires

- jeudi 16h30–18h30, cours magistral, amphi 6C (sauf 29/9)
- lundi 10h30–12h30, TD-TP, salles 473F et 554C
- lundi 13h30–15h30, TD-TP, amphi 6C et salle 554C

## Calendrier

- 22 septembre 2011 — début de cours
- 26 septembre 2011 — début de TD-TP

## Homepage

<http://upsilon.cc/zack/teaching/1112/progsyst/>

# Mailing list

Tous les étudiants *doivent* s'abonner a la newsletter m1 progsyst:

<https://listes.sc.univ-paris-diderot.fr/sympa/info/m1progsyst>

Toutes annonces du cours seront envoyées à cette liste.

# Validation

Le cours sera évalué:

- pour 50% par un projet<sup>3</sup>
- pour 50% par un examen

Le projet consiste à développer un logiciel, en utilisant les concepts et les techniques de programmation systèmes que nous découvrirons.

---

<sup>3</sup>qui n'est pas du contrôle continu donc obligatoire 

# Bibliography

 W. Richard Stevens and Stephen A. Rago

*Advanced Programming in the UNIX® Environment.*

(“APUE”)

Addison-Wesley Professional, 2nd edition, 2005.

- the great classic of UNIX system programming
- recommended mandatory textbook for any UNIX programming course

# Bibliography



W. Richard Stevens and Stephen A. Rago

*Advanced Programming in the UNIX® Environment.*

(“APUE”)

Addison-Wesley Professional, 2nd edition, 2005.

- the great classic of UNIX system programming
- recommended mandatory textbook for any UNIX programming course



Michael Kerisk

*The Linux Programming Interface.*

(“TLPI”)

No Starch Press, 2010.

- more recent (past POSIX.1-2008), more in-depth
- Linux-specific



Robert Love

*Linux System Programming.*

O'Reilly Media, 2007.

- Linux-specific, does not cover IPC
- full of kernel-level insights, useful to any UNIX programmer