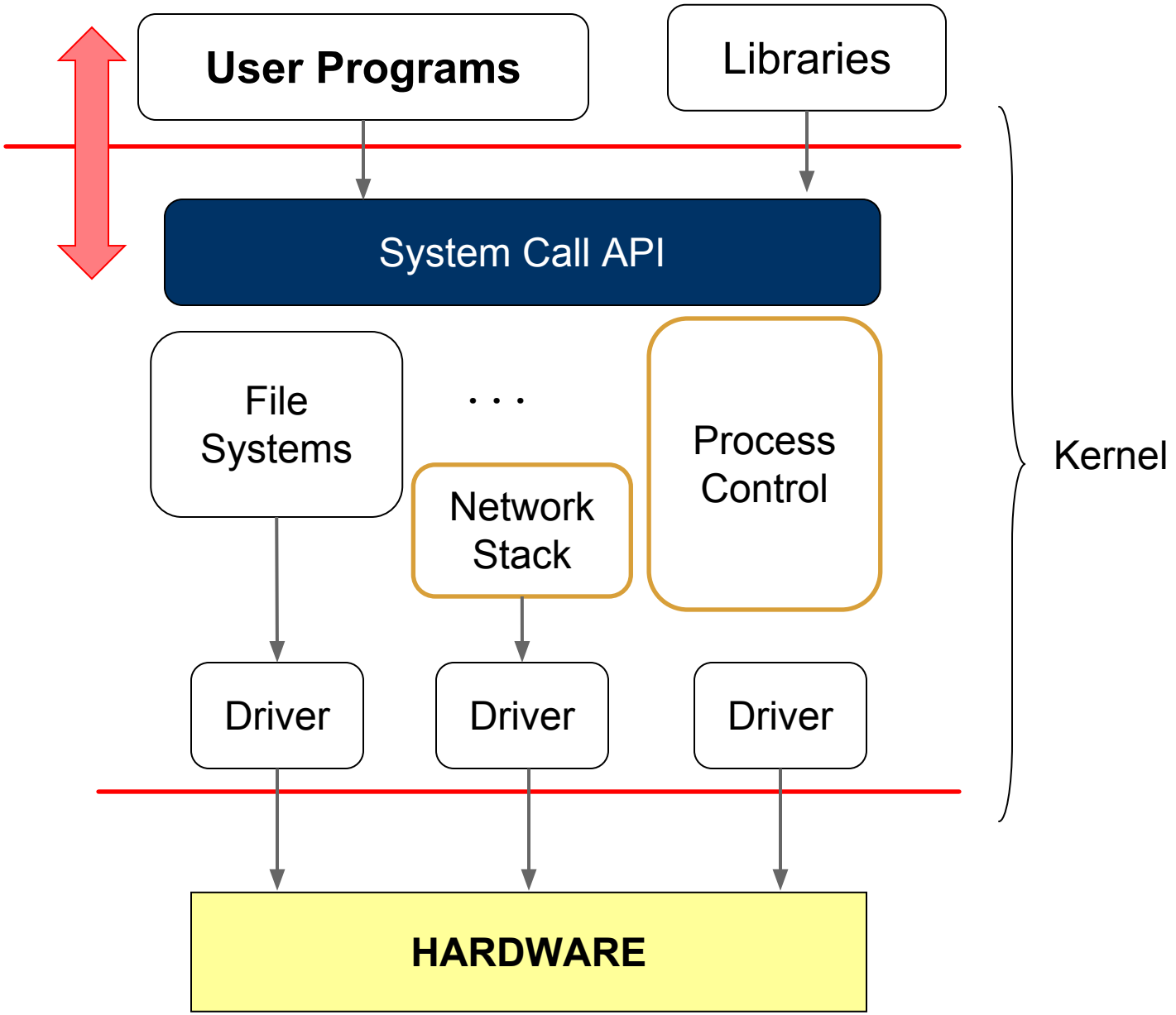
2.1. Introduction

PROFESSORS:
Rubén Santiago Montero
Eduardo Huedo Cuesta

OTHER AUTHORS:
Ignacio Martín Llorente
Juan Carlos Fabero Jiménez

Introduction: System Architecture



Introduction: Programming Standards

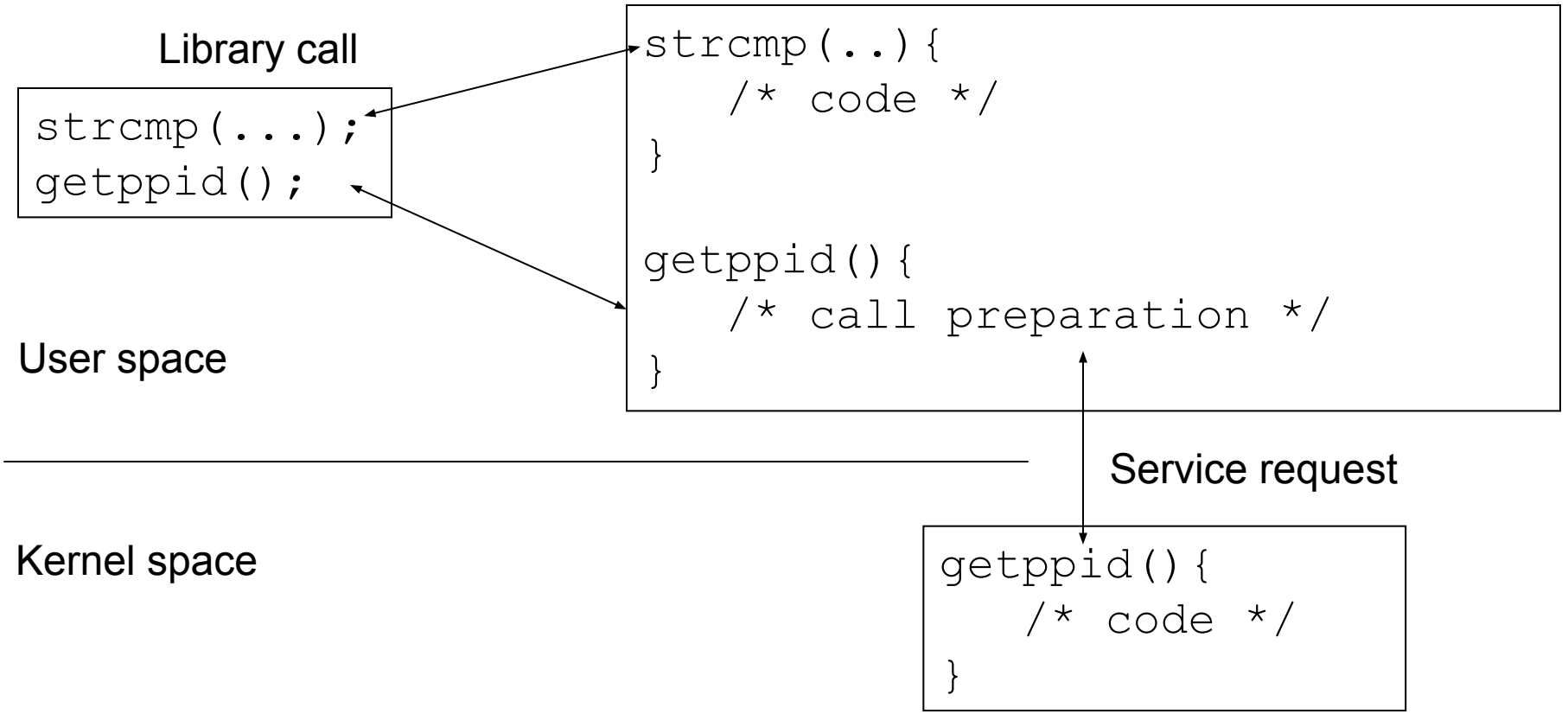
- **ANSI-C** or **ISO-C**: Programming standard adopted by ANSI (*American National Standards Institute*) and later by ISO (*International Standardization Organization*). It is the more general one. Option `-ansi` makes the compiler comply with it strictly.
- **BSD** (*Berkeley Software Distribution*): Developed during the 80s at the University of California Berkeley. Main contributions are symbolic links, sockets, `select` function...
- **SVID** (*System V Interface Definition*): Formal description of the AT&T's commercial UNIX distributions, like System V Release 4 (SVr4). Main contribution is IPC mechanisms.
- **POSIX** (*Portable Operating System Interface*): IEEE and ISO standards derived from different UNIX versions, but mainly from SVID. ANSI-C included. Besides describing system calls and C library facilities, POSIX specifies detailed semantics of a shell and a minimum command set, and also detailed bindings for various programming languages.
- **GNU** (*GNU's Not Unix*): UNIX-like OS that is free software licensed under GNU GPL (*General Public License*). The combination of GNU software and the Linux kernel is GNU/Linux.

System and Library Calls

it's usually a wrapper

From the programmer's viewpoint, there is no difference. However:

- A system call is a function of the C library that requests a service to the system (*trap*). This request is resolved in the kernel of the operating system
- A standard library call does not interact directly with the system (but may use system calls for that)



System and Library Calls

	System call	Library call
Manual section	2	3
Execution area	User/Kernel	User
Argument space	Not reserved	Dynamic/Static
Error code	-1 + `errno`	NULL + no `errno`

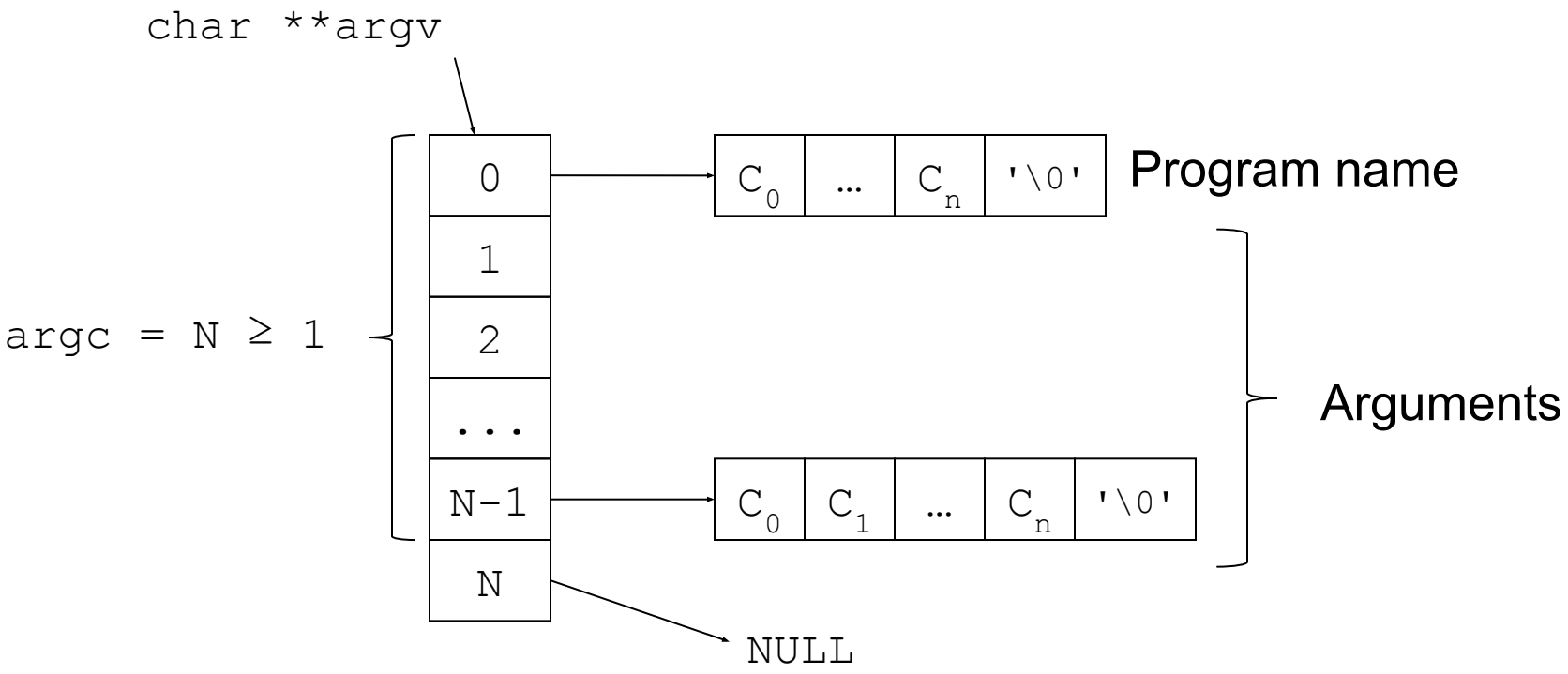
System and Library Calls

- System and library calls are documented in man pages (see `man man`):
  - Section 1: Executable programs or shell commands
  - **Section 2: System calls (functions provided by the kernel)**
  - **Section 3: Library calls (functions within program libraries)**
  - Section 4: Special files (usually found in `/dev`)
  - Section 5: File formats and conventions, e.g. `/etc/passwd`
  - Section 6: Games
  - Section 7: Miscellaneous (including macro packages and conventions), e.g. `man(7)`, `groff(7)`
  - Section 8: System administration commands (usually only for root)
  - Section 9: Kernel routines [Non standard]
- Distributions customize the manual section to their specifics, which often include additional sections
- General query format: `man [section] name`
- Manual section is specified after the name, e.g. `open(2)`
- Using `-k keyword` is useful to search for specific man pages
- It could be necessary to check files in `/usr/include/`

Program Arguments

Definition of **main** program:

```
int main(int argc, char **argv);
int main(int argc, char *argv[]);
```



Program Arguments

- POSIX recommends these conventions for command line arguments:
  - Arguments are options if they begin with a hyphen delimiter (`-`)
  - Option names are single alphanumeric characters
  - Multiple options may follow a hyphen delimiter in a single token if the options do not take arguments. Thus, `-abc` is equivalent to `-a -b -c`
  - Certain options require an argument, for example, `-o name`. The space between options and arguments is optional. Thus, `-o foo` and `-ofoo` are equivalent
  - Options typically precede other non-option arguments
  - The argument `--` terminates all options; any following arguments are treated as non-option arguments, even if they begin with a hyphen
  - A single hyphen character is interpreted as an ordinary non-option argument. By convention, it is used to specify `stdin` or `stdout`
  - Options may be supplied in any order, or appear multiple times. The interpretation is left up to the particular application program
- Long options are provided as a GNU extension, and consist of `--` followed by a name (may be abbreviated) made of alphanumeric characters and dashes
  - An argument for a long option can be specified with `--name=value`

Program Arguments

<unistd.h>
POSIX

- Parse command options:

```
extern char *optarg;
extern int optind, opterr, optopt;
int getopt(int argc, char *const argv[],
    const char *options);
```

  - `options`: String containing valid options for the program. If a colon (`':'`) follows the character, that options uses an argument
  - `optind`: Index of the next element to be processed in `argv` (initialized to 1). If different than `argc` at the end, indicates the first element in `argv` that is not an option
  - `opterr`: If not set to zero, an error message is printed when an unknown option is found
  - `optopt`: Stores the option character when an unknown option is found or a missing option argument is detected
  - `optarg`: Points to the option's argument value

Program Arguments

there is a sample program in the manual page, so we can use it in the exam and modify it to our needs

- Operation of `getopt`:
  - It permutes the contents of `argv` while scanning it so that eventually all the non-options are at the end
  - It returns the option character for the next command line option
  - When no more option arguments are available, it returns -1
    - Then `optind` is the index in `argv` of the first element that is not an option (if `optind` is lesser than `argc`)
  - If the option has an argument, `getopt` stores it in the variable `optarg`
    - Normally, there is no need to copy it, since it is a pointer into `argv`, which is not overwritten
  - If it finds an option character that was not included in `options`, or a missing option argument, it returns `'?'` and sets `optopt` to the actual option character
  - In case of error, if `opterr` is nonzero (which is the default), `getopt` prints an error message to `stderr`

System API

- *Application Programming Interface (API):* Set of functions and routines grouped for a common purpose
- General considerations for using an API:
  - Which header file do I need to include?
  - Which data type does the function return?
  - What are the function arguments?
    - Data types
    - Pass by value or pass by reference (input/output)
  - What is the meaning of the return value of the function?
  - What is the meaning of the arguments of the function?
  - How do I have to manage the memory for the variables?

System API: Trace

useful to know what a program is doing

- Trace system calls and signals:

  `strace [options] command [arguments]`

  - Executes the command intercepting system calls performed and signals received
  - Very useful, since it allows users to analyze the behaviour of programs even if their source code is not available
  - For each system call, a line is shown with the system call performed, arguments and return value
  - Options:
    - `-c`: Count time, calls, and errors for each system call and report a summary on program exit
    - `-f`: Trace child processes as they are created
    - `-T`: Show the time spent in each system call
    - `-e trace=`*`call`*: Trace only the specified system call or set of system calls (`process`, `network`, `ipc`, `signal` or `file`)
    - `-e write=`*`fd`*: Perform a full dump of all the data written to the file descriptor

Error Management

- Print a message on `stderr` output, describing the last error encountered during a call to a system or library function

<stdio.h>
POSIX+ANSI-C

```
void perror(const char *s);
```

- Output format is:

s string	:		error message	\n

- The string `s` should include the name of the function that incurred the error
- The error number is taken from the external variable `errno`, which is set when errors occur, but not cleared when successful calls are made:

<errno.h>
POSIX+ANSI-C

```
int errno;
```

- By convention, when a system call fails, it returns -1 and sets the variable `errno` to a value describing what went wrong
  - Some library functions do the same
- Return string describing error number:

<string.h>
POSIX+ANSI-C

```
char *strerror(int errnum);
```

System Information

- Get name and information about current kernel:

```
int uname(struct utsname *buffer);
```

<sys/utsname.h>
SVID+POSIX

- Returns system information in the structure pointed to by `buffer`, in the following way:

```
struct utsname {
        char sysname[];
        char nodename[];
        char release[];
        char version[];
        char machine[];
}
```

- Error code:
  - **EFAULT**: `buffer` is not valid
- Part of this information is also accessible via

```
/proc/sys/kernel/{ostype,hostname,osrelease,version,domainname}
```

System Information

<unistd.h>

POSIX

- Get configuration information at run time:

```
long sysconf(int name);
```

- Argument `name` can be:
  - `_SC_ARG_MAX`: Maximum length of the arguments to the `exec` family of functions
  - `_SC_CLK_TCK`: Number of clock ticks per second (Hz)
  - `_SC_OPEN_MAX`: Maximum number of files that a process can have open at any time
  - `_SC_PAGESIZE`: Size of a page in bytes
  - `_SC_CHILD_MAX`: Maximum number of simultaneous processes per user ID
- If `name` is invalid, -1 is returned, and `errno` is set to `EINVAL`. Otherwise, the value returned is the value of the system resource and `errno` is not changed

System Information

<unistd.h>

POSIX

- Get information about the file system:

```
long pathconf(char *path, int name);
long fpathconf(int fd, int name);
```

- Argument `name` can be:
  - `_PC_LINK_MAX`: Maximum number of links to the file
  - `_PC_NAME_MAX`: Maximum length of a filename in `path` or `fd` that the process is allowed to create
  - `_PC_PATH_MAX`: Maximum length of a relative pathname when `path` or `fd` is the current working directory
  - `_PC_CHOWN_RESTRICTED`: Nonzero if permissions cannot be changed in the file.
  - `_PC_PIPE_BUF`: size of the pipe buffer, where `fd` must refer to a pipe or FIFO and `path` must refer to a FIFO
- The limit is returned, if one exists. If the system does not have a limit for the requested resource, -1 is returned, and `errno` is unchanged. If there is an error, -1 is returned, and `errno` is set to reflect the nature of the error

User Information

- Get user and group identity:

```
uid_t getuid(void);
gid_t getgid(void);
uid_t geteuid(void);
gid_t getegid(void);
```

<unistd.h> <sys/types.h>
BSD+POSIX

- Processes have a user ID (**UID**) and a group ID (**GID**), corresponding to the identifiers of the owner of the process or, in general, to the owner of the parent process
  - These identifiers are called **real UID** and **real GID**
- Also, processes have an **effective UID** (**EUID**) and **effective GID** (**EGID**), which are actually checked to grant permissions
  - Usually, real and effective IDs coincide
  - However, when a program with the *setuid* or *setgid* bits active is executed, the process inherits these IDs from the owner and group of the program file

User Information

- Get user information accessing password databases:

```
struct passwd *getpwnam(const char *name);
struct passwd *getpwuid(uid_t uid);
```

<pwd.h> <sys/types.h>
SVID+POSIX+BSD

```
struct passwd {
    char *pw_name;   /* User name */
    char *pw_passwd; /* Password */
    uid_t pw_uid;    /* User identifier */
    gid_t pw_gid;    /* Group identifier */
    char *pw_gecos;  /* User real name */
    char *pw_dir;    /* Home directory */
    char *pw_shell;  /* Shell*/
}
```

- The function returns NULL if the user is not found or if an error is produced (e.g. **ENOMEM**: Insufficient memory to allocate `passwd` structure)
- For **shadow passwords**, `getspnam` should be used

System Time Information

- Get the time in seconds since the Epoch

```
time_t time(time_t *t);
```

<time.h>
SVID+BSD+POSIX

- The Epoch refers to 1970-01-01 00:00:00 +0000, UTC
- If `t` is non-NULL, the return value is also stored in the memory pointed to by `t`

- Functions to set and get the system time and date:

```
int gettimeofday(struct timeval *tv, struct timezone *tz);

int settimeofday(const struct timeval *tv,
    const struct timezone *tz);

struct timeval{
    long tv_sec; /* seconds since the Epoch */
    long tv_usec;/* microseconds */
}
```

<sys/time.h>
SVID+BSD+POSIX

- The `timezone` structure is obsolete and `tz` should normally be specified as `NULL`, so the corresponding structure won't be set or returned
- Only the superuser can modify the system time and date

System Time Information

<time.h>
SVID+BSD+POSIX

- Broken-down time representation, expressed in Coordinated Universal Time (UTC):

```
struct tm *gmtime(const time_t *time);
```

- Broken-down time representation, expressed relative to the user's specified timezone:

```
struct tm *localtime(const time_t *time);
```

```
struct tm {
    int tm_sec;     /* seconds 0-59 */
    int tm_min;     /* minutes 0-59 */
    int tm_hour;    /* hours 0-23 */
    int tm_mday;    /* day of the month 1-31 */
    int tm_mon;     /* month 0-11 */
    int tm_year;    /* year since 1900 */
    int tm_wday;    /* day of the week (Sun) 0-6*/
    int tm_yday;    /* day in the year (1-1) 0-365*/
    int tm_isdst;   /* daylight saving time */
};
```

System Time Information

<time.h>
SVID+BSD+POSIX

- Conversion of time information into a string:

```
char *ctime(const time_t *time);
```

- Conversion of time information into a customized string:

```
size_t strftime(char *s, size_t max,
      const char *format, const struct tm *tm);
```

  - Argument `format` is a string where:
    - `%a`: Abbreviated name of the day of the week (locale)
    - `%A`: Full name of the day of the week
    - `%b`: Abbreviated month name
    - `%B`: Full month name
    - `%d`: Day of the month as a decimal number
    - `%H`: Hour as a decimal number using a 24-hour clock
    - `%I`: Hour as a decimal number using a 12-hour clock
    - `%M`: Minute as a decimal number
    - `%S`: Second as a decimal number
    - `%p`: `"AM"` or `"PM"`
    - `%r`: Time in a.m. or p.m. notation. Equivalent to `"%I:%M:%S %p"`
  - It returns the size of the generated string, but if the length of the string (including the terminating null byte) would exceed `max` bytes, 0 is returned