Unix/Linux Programming in C

(NSWI015)

Version: November 13, 2019

Department of SISAL
Faculty of Mathematics and Physics, Charles University
Malostranské nám. 25, 118 00 Praha 1

- This is official material for the class *Unix/Linux Programming in C* (NSWI015) lectured at the Faculty of Mathematics and Physics, Charles University in Prague.

- This material is published under the Creative Commons BY-NC-SA 3.0 license and is always a work in progress, see the history on GitHub: https://github.com/devnull-cz/unix-linux-prog-in-c

- To download the latest version, go to the *releases* tab on GitHub.

- Source code referenced from this material is published in Public Domain unless specified otherwise in the files.

- The source code files can be found on GitHub here: https://github.com/devnull-cz/unix-linux-prog-in-c-src

- In case you find any errors either in the text or in the example programs, we would appreciate you letting us know. Especially do not hesitate to create new issues on https://github.com/devnull-cz/unix-linux-prog-in-c/issues.

Contents

- Introduction, Unix and C, programming tools
- Basic Unix concepts and conventions, its API
- Access rights, devices
- Process manipulation, program execution
- Signals
- Process synchronization and interprocess communication
- Network programming
- Programming with threads
- Appendix

- This lecture is mostly about Unix principles and Unix programming in the C language.
- **The lecture is mostly about system calls, i.e. an interface between a user space and system kernel.**
- For the API, we will follow the *Single UNIX Specification, version 4* (SUSv4). Systems that submit to the Open Group for certification and pass conformance tests are termed to be compliant with the UNIX standard UNIX V7. Some versions of Solaris, AIX, HP-UX a macOS on selected architectures are compliant with the previous version SUSv3 (http://www.opengroup.org/openbrand/register/xy.htm).
- The specific source code examples linked from this material are usually tested on Solaris, macOS and Linux.

Contents

- **Introduction, Unix and C, programming tools**
- Basic Unix concepts and conventions, its API
- Access rights, devices
- Process manipulation, program execution
- Signals
- Process synchronization and interprocess communication
- Network programming
- Programming with threads
- Appendix

Proprietary UNIX and Unix-like Systems

- Sun Microsystems, now Oracle: **SunOS** (defunct), **Solaris**
- Apple: **macOS** (formerly Mac OS X, Mac OS)
- SGI: **IRIX** (in maintenance mode)
- IBM: **AIX**
- HP: **HP-UX**, **Tru64 UNIX** (defunct, formerly by Compaq)
- SCO: **SCO Unix** (discontinued)
- BSD/OS: **BSDi** (discontinued)
- Xinuos (formerly Novell): **UNIXware**

Open source Unix-like Systems

- rather extensive number of **Linux** distributions
- **FreeBSD**
- **NetBSD**
- **OpenBSD**
- **DragonflyBSD**
  - all BSD variants have roots in the 4.3BSD-Lite source code
- **Minix**, micro-kernel based
- **Illumos**, based on Solaris

- Note that **Linux is a kernel**, not the whole system. In contrast to FreeBSD for example, which covers both the kernel and the userland. It is better to say a "Linux distribution" if you discuss a whole system that is built around the Linux kernel.

- FreeBSD and NetBSD forked from 386BSD (now defunct) in 1993, OpenBSD forked from NetBSD in 1995, and DragonflyBSD forked from FreeBSD in 2003. 386BSD itself was based on 4.3BSD-Lite. However, the history is much more complicated, as usual.

- Presently, the "UNIX" trademark can be only used by systems that passed conformance tests defined in the Single UNIX Specification (SUS).

- From those systems listed above, only Solaris, macOS, AIX, and HP-UX are UNIX 03 compliant (http://www.opengroup.org/openbrand/register/). Other non-certified systems, are often described as "Unix-like", even when in many cases they closely follow the standard. However, the word "Unix" is often used for systems from either group.

- The above list is a tiny fraction of the whole Unix world. Every proprietary Unix variant likely came from either UNIX V or BSD, and added its own features. This resulted in quite a few standards as well, see page 5. In the end vendors, agreed upon a small set of those.

- If you are interested in a detailed and up-to-date Unix system version history, go check https://www.levenez.com/unix/.

UNIX standards

- **SVID** (System V Interface Definition)
  - ,,purple book", published by AT&T first in 1985
  - today at version SVID4 from 1995 (SVID3 corresponds to SVR4)
- **POSIX** (Portable Operating System based on UNIX)
  - family of standards published by the IEEE organization marked P1003.xx, gradually incorporated into ISO standards
- **XPG** (X/Open Portability Guide)
  - recommendation of the X/Open consortium, that was founded in 1984 by leading UNIX platform companies
- **Single UNIX Specification**
  - standard of the The Open Group organization founded in 1996 via merging X/Open and OSF
  - today at Version 4 (**SUSv4**)
  - compliance is a requisite condition for using the UNIX trademark

- The very basic information is that the area of UNIX standards is very complex and incomprehensible on a first sight.

- AT&T allowed the producers to call its own commercial UNIX variant "System V" only if it complied to the SVID standard conditions. AT&T also published *System V Verification Suite* (SVVS), that checked whether a given system complies to the standard.

- POSIX (Portable Operating System Interface) is a standardization effort of the IEEE organization (Institute of Electrical and Electronics Engineers).

- SUSv4 is a common standard of The Open Group, IEEE (Std. 1003.1, 2008 Edition) and ISO (ISO/IEC 9945-2008).

- To certify a given system for the Single Unix Specification, it is necessary to pass a series of tests (on given architecture, e.g. 64-bit x86). The results of the tests are then evaluated. The tests themselves are unified into so called *test suites*, which are sets of automatic tests that go through the system and verify if it implements the interfaces specified in the standard. For example, for SUSv3 there are 10 such test suites.

- The interfaces specified by the POSIX.1-2008 standard are divided into 4 basic groups: XSH (System Interfaces), XCU (Shell and Utilities), XBD (Base definitions). W.r.t. number of interfaces, the biggest of them is XSH which contains more than 1000 interfaces.

- The interface groups of POSIX together with the Xcurses group, are part of the Single Unix Specification (however not part of POSIX base in the IEEE Std 1003.1-2001 standard) which includes 1742 interfaces in total, which form the Single Unix Specification (2003). The SUS interface tables are here: http://www.unix.org/version3/inttables.pdf
- Commercial UNIXes largely follow the Single UNIX Specification, compliance to this standard is the condition to use the UNIX trademark (the UNIX 98 brand corresponds to SUSv2, UNIX 03 corresponds to SUSv3, SUSv4 is UNIX V7 - do not mix it up with historical V7 UNIX). It is built on the POSIX base.
- We are going to follow SUSv4 for APIs in this lecture. The data structure definitions and algorithms used by the kernel will be mostly based on System V Rel. 4 to keep things simple.
- On Solaris there is an extensive `standards(5)` manual page, where lots of information about standards can be found in one place. Individual commands compliant to the standard are moreover placed into special directories, e.g. the `tr` program is located in `/usr/xpg4/bin/` and `/usr/xpg6/bin/` directories, in each there is a version of the program compliant to the respective standard. The options and behavior specified by the standard can be then relied upon e.g. when writing shell scripts.
- Also on Solaris, look into the

  `/usr/include/sys/feature_tests.h` header file.

POSIX

- statements like "this system is POSIX compatible" do not give any concrete information whatsoever
  - it might comply to POSIX1990 – what else ?
  - given person either does not know what is POSIX or thinks you do not know
  - the only reasonable reaction is "what POSIX?"
- POSIX is **family of standards**
- the first document is *IEEE Std POSIX1003.1-1988*, later after the introduction of extensions referred to informally as POSIX.1
- last version of POSIX.1 is *IEEE Std 1003.1-2008, 2016 Edition*
  - contains even content formerly defined by POSIX.2 (Shell and Utilities) and miscellaneous, formerly standalone extensions.

- The first document is *IEEE Std POSIX1003.1-1988*, formerly simply referred to as POSIX, then referenced as *POSIX.1*, because by POSIX it is currently meant a set of related standards. POSIX.1 in back then contained programming API, i.e. work with processes, signals, files, timers etc. It was accepted by the ISO organization (*ISO 9945-1:1990*) With small changes and is referred to as POSIX1990. IEEE reference is *IEEE Std POSIX1003.1-1990*. This standard was a great success on its own however still did not connect the System V and BSD camps, because it did not contain BSD sockets or System V IPC (semaphores, messages, shared memory). Part of the standard is the "POSIX conformance test suite (PCTS)", which is freely available.

- The POSIX brand was conceived by Richard Stallman, who founded the GNU project in 1983.

- Important extensions of IEEE Std 1003.1-1990 (they are part of IEEE Std 1003.1, 2004 Edition):

  - *IEEE Std 1003.1b-1993 Realtime Extension*, informally also known as POSIX.4, because that was its original naming before renumbering. Most of this extension is optional, therefore the claim "system supports POSIX.1b" gives even worse testimony that "system is POSIX compatible", i.e. practically zero. The only mandatory part of POSIX.4 is a simple addendum to signals compared to POSIX1990. It is therefore always necessary to state what exactly out of POSIX.4 is implemented – e.g. shared memory, semaphores, real-time signals, memory locking, asynchronous I/O, timers, etc.
  - *IEEE Std 1003.1c-1995 Threads*, see page 184.
  - *IEEE Std 1003.1d-1999 Additional Realtime Extensions*
  - *IEEE Std 1003.1j-2000 Advanced Realtime Extensions*, see page 203.
  - ...

- The POSIX standards can be found on http://www.open-std.org/. The HTML version is freely available, PDF documents have to be purchased.

Books on Unix system principles and design

1. Uresh Vahalia: **UNIX Internals: The New Frontiers**. Prentice Hall; 1st edition, 1995
2. Bach, Maurice J.: **The Design of the UNIX Operating System**. Prentice Hall, 1986
3. McKusick, M. K., Neville-Neil, G. V.: **The Design and Implementation of the FreeBSD Operating System**. Addison-Wesley, 2004
4. McDougall, R.; Mauro, J.: **Solaris Internals**. Prentice Hall; 2nd edition, 2006.
5. **Linux Documentation Project**. http://tldp.org/

- These books are about Unix internals, not about Unix system programming.

1. A great book on Unix in general and compares SVR4.2, 4.4BSD, Solarix 2.x and Mach systems. The 2nd edition, scheduled for 2005, never happened, unfortunately.
2. UNIX classic book. On UNIX System V Rel. 2, and partially 3 as well. While outdated, it is one of the best books ever written on Unix. In 1993 a Czech translation was released as **Principy operačního systému UNIX**, SAS.
3. Structures, functions, and algorithms of the FreeBSD 5.2 kernel; it is based on another Unix classic book **The Design and Implementation of the 4.4 BSD Operating System** by the same author.
4. The best book on the Solaris operating system. The system version in the book is Solaris 10.
5. Linux documentation project home page.

Books on Unix programming

1. Stevens, W. R., Rago, S. A.: **Advanced Programming in UNIX(r) Environment**. Addison-Wesley, 2nd edition, 2005.
2. Rochkind, M. J.: **Advanced UNIX Programming**, Addison-Wesley; 2nd edition, 2004
3. Stevens, W. R., Fenner B., Rudoff, A. M.: **UNIX Network Programming, Vol. 1 – The Sockets Networking API**. Prentice Hall, 3rd edition, 2004
4. Butenhof, D. R.: **Programming with POSIX Threads**, Addison-Wesley; 1st edition, 1997
5. UNIX specifications, see http://www.unix.org
6. manual pages, mainly sections 2 and 3

1. One of the best book on programming in Unix environment. Does not cover networking, that is in 3.
2. Another classic book on programming in Unix environment. Also covers networking. Not as detailed as books 1 and 3 but that could be to your advantage. We very much recommend this book, especially if you want just one. The author can see the big picture which is quite rare.
3. Unix network programming classic, one of the best on the topic; there is also volume 2, **UNIX Network Programming, Volume 2: Interprocess Communications**, covering interprocess communication in great detail.
4. Great book on programming with threads using POSIX API. Highly recommended.
5. UNIX specifications.
6. Detailed descriptions of system calls and functions.
7. A book that did not fit the slide and covers topics outside of the scope of this class: Gallmeister, B. R.: **POSIX.4 Programmers Guide: Programming for the Real World**, O'Reilly; 1st edition, 1995. A great book on real-time POSIX extensions with a beautiful cover. See also pages 132 a 135.

... Go to Amazon and search for "unix". If you ever buy anything, always check whether there is a newer edition of the same book. Note that they often still sell older releases as well.

. . . You can also buy lots of these books on Amazon in a decent second hand quality for a fraction of the original price.

. . . or you can borrow them in library of the faculty !

Manual page sections

- the convention is that “(X)” after a name means the manual page section
- for example, `chmod(2)` means a man page for the system call from a section 2, it does **not** mean a function call
- `chmod(1)` means the shell command
- use “`man <N> <name>`” to get the specific man page
- example: `man 2 chmod`
- see the `man-pages(7)` man page on Linux on what sections exist

- Different systems might have a different list of manual page sections, the numbering may not match, etc. See also `man(1)`. For example, on Solaris, the manual page section needs to be provided with the `-s` option, i.e. “`man -s 2 chmod`”.
- The `man` command uses a list of system directories to search for man pages. If you have manual pages someplace else, perhaps in a local subtree after you unpacked a tar file you downloaded and you want to check the documentation, the `-M` option may come in handy.
- Sometimes there are entries for the same name in several sections. If unsure what you are looking for, use the `-a` option to get all manual pages for that name (otherwise you get just one, usually from the first section found), and go through the individual man pages with the `q` command for `less(1)` which is usually the default pager (or possibly `more(1)`).

The C Programming Language

- virtually all Unix kernels are written in C. Only some HW dependent parts are written in assembler.
- C came into existence in the years 1969-1973, by Dennis M. Ritchie (†2011)
- it evolved from B, designed by Ken Thomson
- created as means to rewrite original Unix in a higher language. It also greatly helped **portability of the system.**
- language variants
  - original K&R C (1978-1979)
  - standard ANSI/ISO C (1989), then next C standard revisions

- The success of C eventually overcame the success of Unix itself.
- CPL ⇒ BCPL ⇒ B (Thompson, interpret) ⇒ C. Both Thompson and Ritchie worked for Bell Laboratories.
- It took many years before C reached its first standard. Most work on C happened in 1972, another peak was in 1977-1979, then in the 1980s ANSI commitee was established to provide the first standard on C. For more information on the early C history, see *Dennis M. Ritchie, The Development of the C Language* paper, available freely.
- K&R C refers to the C language as described in the first edition of *The C Programming Language* classic book by Kernighan and Ritchie, Prentice-Hall, 1978.
- In 1983 ANSI (American National Standards Institute) formed a commitee X3J11 to create the first C standard. After a long and tedious process the standard came to existence as ANSI X3.159-1989 "Programming Language C," and is mostly known as "ANSI C", or C89, and the command line name for the compiler itself was `c89`.
- The 2nd edition of the C book (1988) was updated for the upcoming standard as it used one of its final drafts. In 1990, ANSI C was adopted by ISO as ISO/IEC 9899:1990; that standard is sometimes called C90. It's the same as C89 but it renumbered its sections and removed the rationale document which was part of ANSI C. That standard was adopted back by ANSI. After C89, ANSI never got involved in the C standardization anymore, it only adopted each ISO C standard.

- The next revision of the language was released in 1999 as ISO 9899:1999, informally called C99. After that, there were three technical corrigendums, TC1, TC2, and TC3, so the current version of the C99 standard is the combined C99+TC1+TC2+TC3, WG14 N1256, dated 2007-09-07. It is a work in progress, with its current final draft located here, http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.
- After C99, C11 came, officially ISO/IEC 9899:2011.
- Some difference between C89 and C99 – inline functions, variable definitions intermixed with code, one-line comments using `//`, new functions like **snprintf**() etc.
- The ISO C standards are not free but the drafts are. The latest draft for each standard is virtually the standard itself, it just not does not say that. See http://www.open-std.org/.

Byte ordering

- byte ordering – depends on the architecture
  - $$\begin{array}{r|c|c|c|c|} \text{little endian: } 0x11223344 = & 44 & 33 & 22 & 11 \\ \texttt{addr +} & 0 & 1 & 2 & 3 \end{array}$$
  - $$\begin{array}{r|c|c|c|c|} \text{big endian: } 0x11223344 = & 11 & 22 & 33 & 44 \\ \texttt{addr +} & 0 & 1 & 2 & 3 \end{array}$$
- little endian – Intel, ARM (mostly, but it does support both)
- big endian – SPARC, MIPS, network byte ordering

- Be careful when using tools like `hexdump` that by default print out a file as 16-bit numbers. The ordering of individual bytes may not be how they are stored in a file. For example, take FreeBSD on i386. The first number in the file is character "i" which represents the lower 8 bits of the first 16-bit number, so when the first two bytes are printed out as a 16-bit number, the byte representing "i", i.e. "69", is shown as the second byte. Similarly for "kl".

```
$ echo -n ijkl > test
$ hexdump test
```

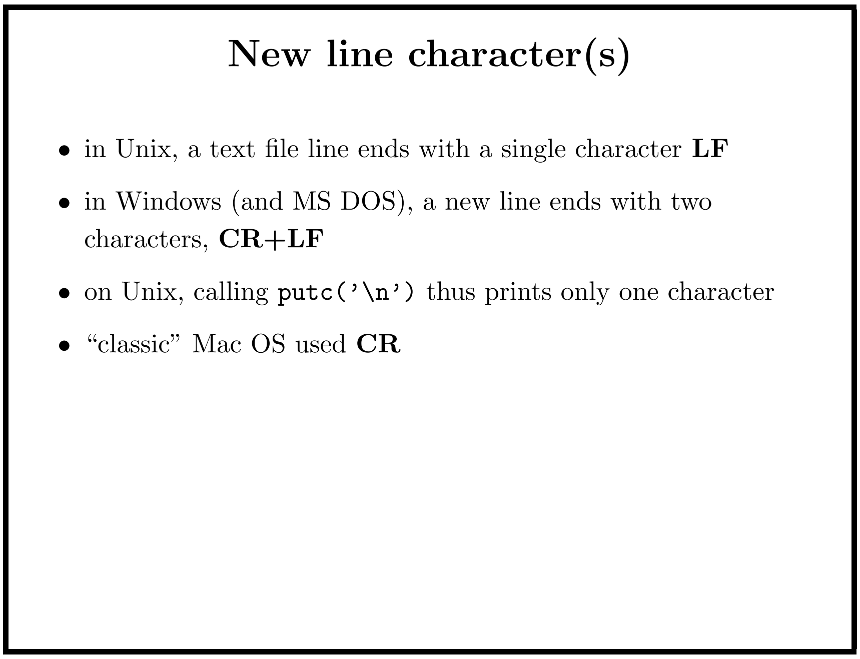
```
0000000 6a69 6c6b
0000004
```

You can use other output formats though, for example as hex bytes and characters in the same output:

```
$ hexdump -C test
00000000 69 6a 6b 6c            |ijkl|
00000004
```

- The UNIX spec does not list `hexdump` but defines `od` (octal dump). The equivalent output for the `hexdump` default output is as follows. Note that since we did that on SPARC, the output is different from the i386 output above!

```
$ od -tx2 test
0000000 696a 6b6c
0000004
```



New line character(s)

- in Unix, a text file line ends with a single character **LF**
- in Windows (and MS DOS), a new line ends with two characters, **CR+LF**
- on Unix, calling `putc('\n')` thus prints only one character
- “classic” Mac OS used **CR**

- **LF**, *line feed*, sometimes also referred to as *new line*, is a character 0x0A (10). **CR**, *carriage return*, or simply *return*, is a character 0x0D (13).
- To further confuse the enemy, “classic” Mac OS used a single **CR** as line breaks. As present time macOS comes from the Unix world, it also uses **LF** now.

- When you open a text file in classic `vi` and you see strange `^M` characters at the end of every line, it is that **CR** character from a line separator in a file brought over from a Windows system. Just get rid of them via `:%s/^V^M//g` where `^X` means Ctrl+X. ViM by default tries to be smarter in such situations but not always to your benefit.
- See the `ascii` man page for the octal, hexadecimal, and decimal ASCII character sets (i.e. up to character 127 as ASCII table has only 128 characters).

C style

- C style of the source code files is extremely important
- there are quite a few ways how to do it:

```
int
main(void)
{
        int i;
        char c = 'X';

        for (i = 0; i < 10; ++i)
                printf("%c%d\n", c, i);
        return (0);
}
```

- One of the most important thing of a C style (well, any style) is consistency. And often it is not that important what exact C style a group of coders is going to pick as it is that one specific style is chosen and then religiously followed by all in the group. A good and rigorously followed cstyle leads to a smaller number of bugs in code.
- A process that runs C style check script before integration into a source code repository automatically and refuses to accept any changesets not following the chosen C style is a working solution to avoid C style violations.
- http://mff.devnull.cz/cstyle/

C style (cont.)

- many ways how **NOT** to do it (so called assembler style):

```
int main(void) {
int i = 0; char c;
printf("%d\n", i);
return (0);
}
```

- or a schizophrenic style:

```
int main(void) {
        int i = 0; char c;
        if (1)
        printf("%d\n", i);i=2;
return (0); }
```

- A good C style of of the source code you write represents you. You will be judged by other people by the way your source code looks. Always try to write beautiful code.

Standard Utilities

cc, **c99***, **gcc**†	C compiler
CC, **g++**†	C++ compiler
ld	linker
ldd	for listing dynamic object dependencies
cxref*	generate a C program cross-reference table
sccs*	source code management
make*	for maintaining program dependencies
ar*	for managing archives
dbx, **gdb**†	debuggers
prof, **gprof**†	profilers

* SUSv4 † GNU

SUSv4

- The standard C language compiler is `c99`, required by the specification. Be careful as the default mode for `gcc` does not conform to any of the ISO C standards. You need to check the manual page for your version, look for the option `-std=` to see what is the default. For example, for version 4.2.1, the default is `-std=gnu89`, for version 7.2, it is `-std=gnu11`.
- Do not use `sccs` for source code management. Unless you are forced to use a centralized source code management (CVS, Subversion, etc.) due to historical reasons or while working on an existing project, always use a **distributed** source code management system when starting a new project. We recommend Git (`git`) or Mercurial (`hg`).
- Debuggers and profilers are not part of the standard.

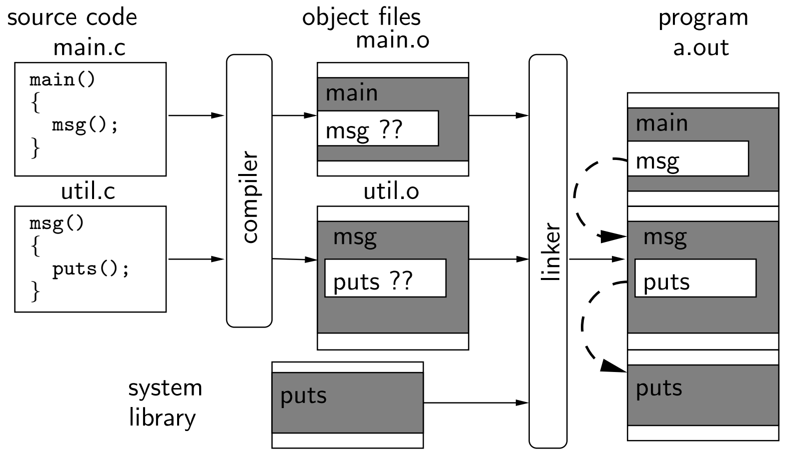
File name convention

`*.c`	the C language source code files
`*.cc`	the C++ language source code files
`*.h`	header files
`*.o`	object files
`a.out`	the default executable file name after the compilation

`/usr/include`	system header file root
`/usr/lib/lib*.a`	static libraries
`/usr/lib/lib*.so`	dynamic libraries

- Static libraries – code for used external functions is copied into a target program. Not used much nowadays.
- Dynamic libraries – the list of dynamic libraries needed are part of the program, on execution the dynamic linker (path to the dynamic linker is also part of the program, see page 30) loads them to memory and relocates pointers.
- Today, dynamic libraries are mostly used as they save disk space and you do not need to recompile all the utilities and other program on library upgrades.
- In specific situations, static libraries are still needed though, for example, in standalone binaries when booting an operating system.
- The origin of the name `a.out` is as follows. Initially, even before the C was invented, there were no libraries, no loader or link editor in the first version of the UNIX system: the entire source of a program was presented to the assembler, and the output file with a fixed name that emerged was directly executable. So `a.out` means "the output of the assembler". Even after the system gained a linker and a means of specifying another name explicitly, it was retained as the default executable result of a compilation. See *Dennis M. Ritchie, The Development of the C Language* paper, available freely.

Process of compilation



```
main()
{
  msg();
}
```

```
msg()
{
  puts();
}
```

- Non-trivial programs are often split into several source code files that contain related functions. Such files can be compiled independently, and you can even use different languages and different compilers for each file. The advantage is the speed of building, as only modified files are re-compiled (see page 26 on the `make` utility), and also flexibility, as you can use some of the files in other programs as well.

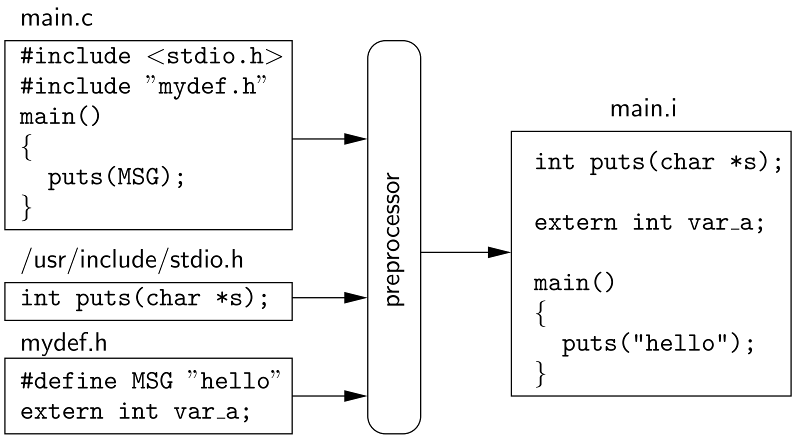
- The *compiler* compiles each file into a corresponding object file. Instead of external function pointers in the compiled code, the object file contains a table of global symbols.

- Then, the *linker* combines the built object files and used libraries into an output file. By default, it also resolves all the references to make sure all symbols used are available.

- Used code from the static libraries is copied to the executable file. When using dynamic libraries, the executable only contains a list of them, the linking process is then performed by the runtime linker (aka loader) on the program execution. For more on the dynamic linking process, see page 30.

- To select whether to use static or dynamic libraries, you use options for the linker. By default, dynamic libraries are used nowadays. The source code is same in either case. There is also a mechanism (`dlopen`, `dlsym`...) that allows to load an additional dynamic library during the program execution, and use it. For more information, see page 120.

Compilation of one file: preprocesor



main.c

```
#include <stdio.h>
#include ”mydef.h”
main()
{
  puts(MSG);
}
```

/usr/include/stdio.h

```
int puts(char *s);
```

mydef.h

```
#define MSG ”hello”
extern int var_a;
```

main.i

```
int puts(char *s);

extern int var_a;

main()
{
  puts("hello");
}
```

- The preprocessor performs macro expansion, conditional compilation, and inserts included files. It also removes comments.

- The preprocessor output can be provided via `cc -E` or calling `cpp` directly. However, some compilers have the preprocessor functionality built in so calling the external preprocessor may not get the same results. You can of course use the preprocessor for anything else where its functionality comes in handy, not just for C source code.

- In a situation where you need to fix code full of includes and conditional compilation, the output after the preprocessor phase may be very helpful to locate the problem.

- `cpp` (or `cc -E`) also allows you to see the whole tree of included files, printed on the standard error output. For that, use a separate `-H` option (not `-EH`) and redirect the output to `/dev/null`:

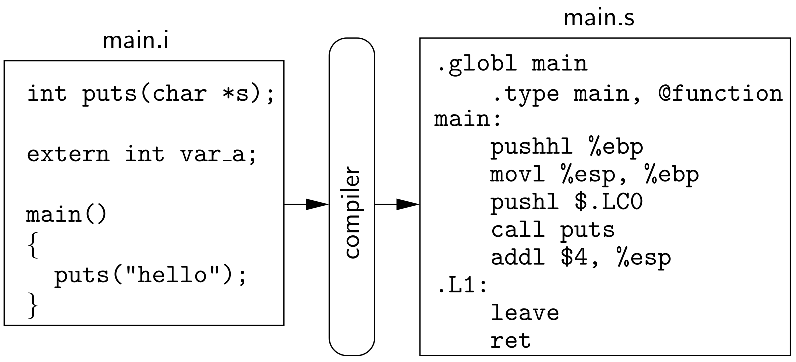
```
$ gcc -E -H tcp/connect.c >/dev/null
. /usr/include/stdio.h
.. /usr/include/sys/cdefs.h
... /usr/include/sys/_symbol_aliasing.h
... /usr/include/sys/_posix_availability.h
.. /usr/include/Availability.h
... /usr/include/AvailabilityInternal.h
.. /usr/include/_types.h
... /usr/include/sys/_types.h
```

```
.... /usr/include/machine/_types.h
..... /usr/include/i386/_types.h
etc...
```

- You cannot nest comments in C, so in order to temporarily disable code with comments without deleting it, wrapping it in another comment will not work. So, the preprocessor to your rescue – use the conditional compilation feature:

```
...
#if 0
        /* some comment */
        some_function();
        /* another comment */
        another_function();
#endif
...
```

Compilation of one file: compiler



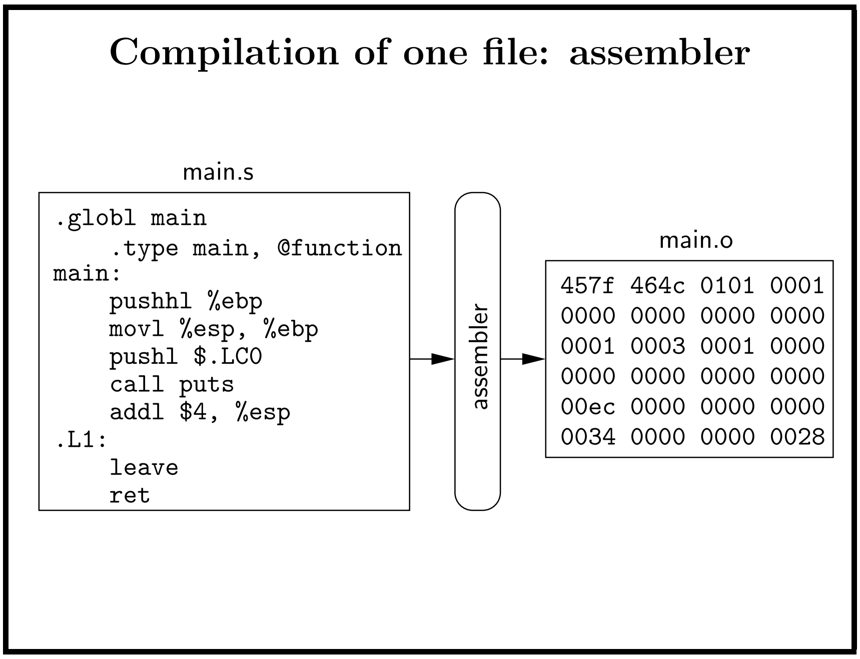
```
int puts(char *s);

extern int var_a;

main()
{
  puts("hello");
}
```

```
.globl main
    .type main, @function
main:
    pushhl %ebp
    movl %esp, %ebp
    pushl $.LC0
    call puts
    addl $4, %esp
.L1:
    leave
    ret
```

- The picture is an example output for the x86 platform, 32-bit, with AT&T syntax.
- Compilation from the C language into assembler.
- The assembler output file is the result of `cc -S`.



Compilation of one file: assembler

main.s

```
.globl main
    .type main, @function
main:
    pushhl %ebp
    movl %esp, %ebp
    pushl $.LC0
    call puts
    addl $4, %esp
.L1:
    leave
    ret
```

main.o

```
457f 464c 0101 0001
0000 0000 0000 0000
0001 0003 0001 0000
0000 0000 0000 0000
00ec 0000 0000 0000
0034 0000 0000 0028
```

- Again an example for the x86 platform, 32-bit.
- Compilation from the assembler language into the object code.
- The output file is the result of `cc -c`.

Compiler

- usage:
  `cc [`*`options`*`]` *`file`* `...`
- the most important options:

`-o` *`file`*	output file name
`-c`	only compile, do not link
`-E`	only preprocessor
`-l`	link with the specified library
`-L`*`directory`*	add a directory to search when using `-l`
`-O`*`level`*	optimization level
`-g`	compile with debug information
`-D`*`name`*	define a macro for the preprocessor
`-I`*`directory`*	add a directory to search for `#include` files

- `-l`/`-L` are actually options for the linker, i.e. the compiler will pass them on onto the linker.
- Both the compiler and linker have an extensive list of additional options that influence the generated code and what warnings are printed during the compilation/linking based on the chosen language and the standard. See manual pages for `cc`, `gcc`, and/or `ld`.

UNIX standard macros

`__FILE__`, `__LINE__`, `__DATE__`, `__TIME__`, `__cplusplus`, etc.	are standard macros for the compiler C/C++
`unix`	always defined if on Unix
`mips`, `i386`, `sparc`	hardware architecture
`linux`, `__APPLE__`, `sun`, `bsd`	operating system
`_POSIX_SOURCE`, `_XOPEN_SOURCE`	build using the specific standard

UNIX standard macros (cont.)

To build using a specific standard, you need to define one of the macros below before any `#include`. Then include `unistd.h`.

UNIX 98	`#define _XOPEN_SOURCE 500`
SUSv3	`#define _XOPEN_SOURCE 600`
SUSv4	`#define _XOPEN_SOURCE 700`
POSIX1990	`#define _POSIX_SOURCE`

- The way it works is that you use specific macros to define what you want (e.g. _POSIX_SOURCE), and then you use other macros (e.g. _POSIX_VERSION) to find out what you actually got. You always have to include unistd.h after you set the macros and use a compiler that supports what you want. For example, below we tried to compile basic-utils/standards.c which requires SUSv3, on a system supporting SUSv3 (Solaris 10), using a compiler that only supports SUSv2 (the compiler defined in SUSv3 is c99). Note that the default behavior of your compiler might be same as c89.

```
$ cat standards.c
#define _XOPEN_SOURCE   600
/* you must #include at least one header !!! */
#include <stdio.h>
int main(void)
{
        return (0);
}
$ c89 basic-utils/standards.c
"/usr/include/sys/feature_tests.h", line 336: #error: "Compiler or
options invalid; UNIX 03 and POSIX.1-2001 applications require
the use of c99"
cc: acomp failed for standards.c
```

- See the documentation for your compiler about what other macros can be used.

- See page 10 for more information on standards.

- Regarding macros for specific standards, you can find very good information in chapter 1.5 in [Rochkind]. See also basic-utils/suvreq.c.

```
int
main(void)
{
#ifdef unix
        printf("Yeah!\n");
#else
        printf("Oh, no.\n");
#endif
        return (0);
}
```

- For an example on using __LINE__, see basic-utils/main__LINE__.c

Link editor (linker)

- Invocation:
  ```
  ld [options] file ...
  cc [options] file ...
  ```
- Often used options:

`-o` *file*	output file name (default `a.out`)
`-l`*lib*	link with library `lib`*lib*`.so` or `lib`*lib*`.a`
`-L`*path*	path to libraries (`-l`*lib*)
`-shared`	create a dynamic library
`-non_shared`	create a static executable

- A linker takes one or more objects generated by a compiler and creates a binary executable, library, or another object file suitable for another linking phase.
- Note that different systems support different options. For example, `ld` on Solaris does not support `-shared` and `-non_shared`, and you have to use alternatives.
- An option `-R` allows to specify where to look for libraries when loading the executable via the dynamic linker. That path might be different from the path used during building the object, modifiable via `-L`.
- Often you do not use the linker directly at all but pass all the linker options via the compiler.

Maintaining programs (make)

- **source code**

main.c

```
#include "util.h"
main()
{
  msg();
}
```

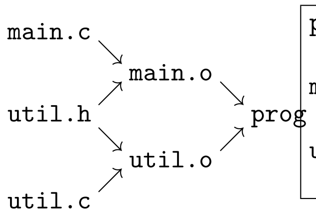
util.h

```
void msg();
```

util.c

```
#include "util.h"
msg()
{
  puts();
}
```

- **dependencies**



- **file** Makefile

```
prog : main.o util.o
        cc -o prog main.o util.o
main.o : main.c util.h
        cc -c main.c
util.o : util.c util.h
        cc -c util.c
```

.

- You could compile and link the program via one invocation of a compiler, or write a simple shell script. However, by using `make` you will only update a target if its dependencies have been modified. In other words, a well written makefile will cause to re-compile only what is really necessary after some files have been changed. You could always do something like “`make clean; make all`” but if the whole compilation process takes minutes or even hours, you really want a well written `Makefile`.

- A line “`prog : main.o util.o`” defines that before `prog` is checked, the existence of `main.o` and `util.o` needs to be checked, and also whether they are up to date. That check is performed recursively. After that, the existence of `prog` is checked and whether it is up-to-date, which means whether the last modification time is younger than that of `main.o` and `util.o`. If yes, nothing else is done. If not, the command on the next line is performed.

- `make` is usually run with an argument specifying the target to be built; if run without arguments, the first target in the `Makefile` is used. That often is `all` which if the standard Unix convention is followed, builds everything that can be built. After that, `make install` often follows, etc.

- `make` is a universal tool, useful not just for building source code. For example, to build this material from various LaTeX and other source files, `make` is used as well.

- Example: `basic-utils/Makefile01`. Note that if a non-standard make file is used, you need the `-f` option: “`make -f Makefile01`”.

Syntax of the `make` input file

- target dependencies: *`targets : [files]`*
- commands to be executed: `<Tab>`*`command`*
- comment: `#`*`comment`*
- line continuation: *`line-begin`*`\`
  `line-continuation`

- **Note that the line with a command starts with a tab, not spaces.** Every command line is executed via its own shell invocation. If multiple commands need to be executed via the same shell process, all but the last line needs to be terminated with a backslash. See the following example where the last two `echo` commands are part of the same `if` construct.

```
$ cat basic-utils/Makefile02
all:
	@echo $$$$
	@echo $$$$
	@if true; then \
		echo $$$$; \
		echo $$$$; \
	fi
$ make -f Makefile02
5513
5514
5515
5515
```

- The backslash works as a word separator and a space is inserted. See the following example: `basic-utils/Makefile07`.
- Using a double `$` suppresses the special meaning of a dollar sign, see the next slide.

- A character `@` at the beginning of a line suppresses its printout. Otherwise, `make` always prints out what is gonna be executed next.
- For a dry run, i.e. to see what would be executed, but do not execute anything, use option `-n`.
- A character `-` at the beginning of a line causes `make` to ignore a non-zero return value (usually indicating a failure), otherwise `make` reports the error and bails out right away. Example: `basic-utils/Makefile04`.

-
  ```
  test1:
          false
          echo "OK"

  test2:
          -false
          echo "OK"
  ```
- You can start building from a specific target: `make <target>`. If there is no target argument, the first target in the makefile is assumed.

Macros (`make`)

- macro definition:

  `name = string`
- continuation via a backslash inserts a space
- undefined macros are empty
- ordering of macro definitions is not important
- defining a macro on a command line:

  `make` *`target name=string`*
- macro invocation:

  `$`*`name`* (only one character *`name`*),

  `${`*`name`*`}` or `$(`*`name`*`)`
- environment variables are accessible as macros (e.g. `${EDITOR}`)

- If a macro is defined multiple times, the last definition rules, you can see an example in `basic-utils/Makefile03`. The definition on the command line goes after any definition in the Makefile. Try `make -f Makefile03 D=4`.
- You cannot define a macro recursively, see `basic-utils/Makefile05`:

```
$ cat basic-utils/Makefile05
M=value1
M=$(M) value2
all:
        echo $(M)
$ make -f Makefile05
Variable M is recursive.
```

- Often extended `make` versions are used, e.g. GNU (`gmake`) or BSD.
- To write a non-trivial `Makefile` that will work with different `make` implementations is not a simple task. Therefore projects as GNU Automake exist. For a simple conditional compilation, where based on the system we need to set different options, the following code might come in handy. A character ‘ is a back quote, and a ’ is a normal single quote:

```
CFLAGS=‘x=\‘uname\‘; \
        if [ $${x} = FreeBSD ]; then \
                echo ’-Wall’; \
        elif [ $${x} = SunOS ]; then \
                echo ’-v’; \
        elif [ $${x} = Linux ]; then \
                echo ’-Wall -g’; \
        fi‘

all:
        @echo "$(CFLAGS)"
```

- In other situations it is recommended or even needed to use utilities like `autoconf` or `automake`.
- Some `make` implementations support directives for conditional processing, e.g. BSD make. Example: basic-utils/Makefile08.bsd.
- With the `-e` option, we can force `make` to ignore a variable definition in the input file if an environment variable of the same name exists. By default `make` accepts environment variables only if they are not defined in the input file. Example: basic-utils/Makefile06.
- In general, `make` is a immensely powerful tool, just take a look at system make files of any Unix-like system. A typical feature of such build systems is that there is no documentation on how it works internally so you have to dig in deep if you need to understand or modify it – and that usually is not for the fainthearted.

Dynamic linker (loader)

- the compilation phase requires all needed dynamic libraries to check accessibility of used symbols
- **loading external shared libraries into a running process happens on program execution**. That is what a **dynamic linker** does (*run-time linker*, *loader*).
- list of required dynamic libraries is in the `.dynamic` section of an ELF object
- system by default looks for shared libraries in certain locations
- located libraries are mapped to the process address space via **mmap**() (will be later)

- An ELF object format is explained on page 109.
- In the ELF `.dynamic` section, you can add additional paths to search for the libraries using tags `RUNPATH`/`RPATH`.
- The process of an execution of a dynamically linked program works like this:
  - The kernel in `exec()` maps the program to a newly created process address space and finds out what a dynamic linker is used (see `.interp` below).
  - The kernel maps the dynamic linker to the process address space as well and calls the linker's `main()` function. As a dynamic linker is an executable program by itself, it has `main()`. You can usually also run a dynamic linker on a command line if you want to, that is mostly useful for debugging and experimenting with a dynamic linker.
  - The linker gets the list of required libraries from the program ELF header, maps those libraries to the process address space, and calls their initialization functions if those exist. All dependencies not set as *lazy* (see page 120) are mapped recursively via breadth search.
  - The linker's job is done at that point and calls the program `main()` function.
  - A process may continue to use the dynamic linker during program execution via calls like `dlopen()` etc. See page 120 for more information.
- Note that the dynamic linker does not run as a separate process (unless you run it like that) even though it has its own `main()` function. It is used within

an address space of an executed program. The program, dynamic linker, and dynamic libraries constitute a single process.

- The following examples **are from Solaris**. Finding equivalent commands and/or options on Linux is left as an exercise to the reader.
  - ELF sections are listed via `elfdump -c` (GNU has `objdump` and `readelf`). More on program sections on page 109.
  - What a dynamic linker is used is in section `.interp`, see "`elfdump -i`" and "`ld -I`". It means you could write your own dynamic linker and set it via the `-I` option for `ld` to your program. Needless to say, such an enterprise would not be an easy feat at all.
  - To list the dynamic section, use `elfdump -d`, dynamic libraries are set as `NEEDED`.
  - Finding out shared object dependencies is very easy via the `ldd` command (Solaris, Linux, BSD). It displays paths to the specific libraries (i.e. full paths) that will be used if the program is run in the same environment (see right below for more information). The command resolves the dependencies recursively so you will also see dynamic libraries that are used by other libraries and not directly by the program. To find out what exactly depends on what, use the `-v` option. macOS does not have `ldd`, use `otool -L` instead.
  - What libraries are eventually used when running the program could be different from what `ldd` shows. For example, one could use the `LD_PRELOAD` mechanism. For that reason, Solaris has a `pldd` command which provides the library dependencies for a running process. For an example on `LD_PRELOAD` with `gcc`: use already mentioned `Makefile01`, and compile `basic-utils/preload.c` like this:

    ```
    gcc -shared -o libpreload.so preload.c
    ```

    Run the program then which interposes a system call `close()` like this:

    ```
    LD_PRELOAD=./libpreload.so ./a.out
    ```

  - Most of the information listed here can be found in the manual page for the Solaris dynamic linker, `ld.so.1(1)`, and much more in the excellent *Linkers and Libraries Guide* on `docs.oracle.com`. If you use FreeBSD, its dynamic linker is `ld-elf.so.1`, on Linux distributions it is usually `ld-linux.so.1`, it is `rld` on SGI IRIX etc.
  - You can also configure the dynamic linker via setting environment variables. For example, try this on Solaris:

    ```
    LD_LIBRARY_PATH=/tmp LD_DEBUG=libs,detail date
    ```

    and to find out what options you have to debug the dynamic linker there:

    ```
    LD_DEBUG=help date
    ```

  - To tell the linker to also look for dynamic libraries in directories other than the default ones (paths from `LD_LIBRARY_PATH` are searched first), use it like the following:

    ```
    $ cp /lib/libc.so.1 /tmp
    $ LD_LIBRARY_PATH=/tmp sleep 100 &
    ```

```
[1] 104547
$ pldd 104547
104547: sleep 100
/tmp/libc.so.1
/usr/lib/locale/cs_CZ.ISO8859-2/cs_CZ.ISO8859-2.so.3
```

  – You can also edit ELF objects via `elfedit(1)` on Solaris. You can change `RUNPATH`, for example.

- In general, you should not use `LD_LIBRARY_PATH` for anything else than debugging during the development or when moving libraries between directories. You can find lots of articles on "why is `LD_LIBRARY_PATH` evil?" etc. For example, http://xahlee.org/UnixResource_dir/_/ldpath.html.

  This variable is often misused in the start-up scripts because the command(s) is/are incorrectly linked and the dynamic linker would not otherwise find the correct libraries. The typical side effect, however, is that the program(s) subsequently start(s) additional programs that use the same libraries but as all children inherit the environment, those programs are forced to use libraries from non-default directories, and possibly those contain libraries of different versions from those that the programs were initially built with. Quite often this might be hard to find when something goes bad and you start seeing unexpected behavior. Commands like `pldd` come in handy in such situations.

API vs ABI

API – Application Programming Interface

- interface used in the **source code** to use another software component like a library, OS kernel, or your own code – e.g. `exit(1)`, `printf("hello\n")` or `my_function(1, 2)`
- . . . so that the same source code could be compiled on all systems supporting a given API

ABI – Application Binary Interface

- low-level binary interface between **modules** (e.g. `a.out` and `libc.so.1` or even `a.out` and `a.out`)
- . . . so that the built module could be used wherever the same ABI is supported

- In short – an API is source code based while an ABI is binary based.
- An example of an API is one defined by the POSIX.1 standard or the set of system calls for a given system.

- An example of an ABI is the System V AMD64 ABI, followed on Solaris, Linux, FreeBSD, macOS, and other systems.
- ABI defines the calling convention interface to the called machine code, e.g. how parameters are passed (pushed on the stack, placed in registers, or a mix of both) or how return value is returned.
- API defines a set of functions, its parameters and their types, and function return values. The API may also include global variables – `errno`, for example.
- The following example represents what happens if a library ABI is changed and the new library replaces the old one. Note that the dynamic linker cannot detect that the function symbol did not change. The given change is also an API change and the problem would be fixed if `main.c` was recompiled with the correct prototype for function `my_add`. However, recompiling is often not desirable as one might end up recompiling the whole system. That is why keeping backward compatibility for library ABI is so important.

  The first result is expected, i.e. `3`:

```
$ cat main.c
#include <stdio.h>

int my_add(int a, int b);

int
main(void)
{
        (void) printf("%d\n", my_add(1, 2));
        return (0);
}

$ cat add.c
int
my_add(int a, int b)
{
        return (a + b);
}

$ gcc -shared -o libadd.so add.c
$ gcc -L. -ladd -Xlinker -R . main.c
$ ./a.out
3
```

  Now imagine a new library came, one with a modified ABI in function `my_add`. Note that we replaced the old library with the new one. Now, instead of 4 byte integers, 64-bit longs are used. When you run the program again, you will get an incorrect return value:

```
$ cat add2.c
int64_t
my_add(int64_t a, int64_t b)
{
        return (a + b);
}

$ gcc -shared -o libadd.so add2.c
$ ./a.out
-1077941135
```

- Example: lib-abi/abi-main.c (see the block comment in the file on how to use other files located in the same directory).
- To change an ABI safely, you need library versioning – if the library ABI change is not backward compatible, a bumped up version needs to prevent running the program without rebuilding it. However, in that case you need to keep the old library on the system. Note that having different versions of the same library might become a problem by itself, for example if more versions of such a library get loaded into the program address space.
- The way versioning works in ELF (see page 109) is that the library file name incorporates a version. The SONAME in the dynamic section then lists the full name of the library, including the version. When the loader searches for the libraries, it searches for the file name from the SONAME field.
- Example of what exact libraries and their versions are needed for the Secure shell client on a Linux distribution. You see that, for example, the OpenSSL library version used by the SSH client is 1.0.0.:

```
$ readelf -d /usr/bin/ssh

Dynamic section at offset 0xb0280 contains 34 entries:
  Tag        Type           Name/Value
0x0000000000000001 (NEEDED) Shared library: [libsctp.so.1]
0x0000000000000001 (NEEDED) Shared library: [libcrypto.so.1.0.0]
0x0000000000000001 (NEEDED) Shared library: [libdl.so.2]
0x0000000000000001 (NEEDED) Shared library: [libz.so.1]
0x0000000000000001 (NEEDED) Shared library: [libresolv.so.2]
0x0000000000000001 (NEEDED) Shared library: [libpthread.so.0]
0x0000000000000001 (NEEDED) Shared library: [libgssapi.so.3]
0x0000000000000001 (NEEDED) Shared library: [libc.so.6]

$ openssl version
OpenSSL 1.0.2n  7 Dec 2017

$ ls /usr/lib/libcrypto.so.1.0.0
/usr/lib/libcrypto.so.1.0.0
```

The OpenSSL library version is kept 1.0.0 even though the real version is actually 1.0.2n. That is because the micro versions ("z" in x.y.z) do not change the ABI binary compatibility (change in "y" does). So, such a number must not be in the SONAME field otherwise you would need rebuilt binaries that depend on OpenSSL, even on a binary compatible micro upgrade.

Debugger `dbx`

- Usage:

```
dbx [ options ] [ program [ core ] ]
```

- Most common commands:

`run [`*arglist*`]`	program start
`where`	print stack
`print` *expr*	print expression
`set` *var* `=` *expr*	change value of variable
`cont`	continue program run
`next`, `step`	execute a line (with/out going into function)
`stop` *condition*	set breakpoint
`trace` *condition*	remove tracepoint
`command` *n*	action on breakpoint (commands follow)
`help [`*name*`]`	help
`quit`	debugger exit

- Basic line oriented debugger with symbols; to fully use it, the program to be debugged has to be compiled with debugging data (`cc -g`). The program is then started from debugger using the `run` command, or the debugger can be connected to an already running process. `dbx` can also be used to analyze program crashes, provided a `core` file was generated.

- It is possible to find it e.g. on Solaris, it is not present by default on other systems.

- For source line debugging it is not necessary to use just the `-g` option for translation, it is also necessary to have the object files and source files available in the same location as they were used for compilation. This is typically true when debugging on the system where the code is written. For other use cases it is necessary to provide the needed files, the `dbx` command `pathmap` can help with that.

- `gdb`-compatible mode can be enabled using `gdb on`. If you want to know the `dbx` equivalent command to concrete `gdb` command, see the `help FAQ`; the very first question is "A.1 Gdb does <something>; how do I do it in `dbx` ?"

- If the option `-g` is not used, `dbx` will be still usable on Solaris, because it will print function arguments. On BSD systems and Linux distributions, the `-g` has to be used, otherwise the debuggers will not be of much use. This is demonstrated in the `debug/dbx.c` example. When compiling with `gcc` and using `gdb` the `where` command will not show the function parameters, while on Solaris with the Studio compiler and `dbx` debugger, the function parameters will be shown.

- Example: debug/coredump.c. After compilation and running the program will crash and a core dump will be generated (if permitted on the system; also see `ulimit -c`)

  ```
  $ cc coredump.c
  $ ./a.out
  Segmentation Fault (core dumped)
  $ dbx ./a.out core
  Reading a.out
  core file header read successfully
  Reading ld.so.1
  Reading libc.so.1
  program terminated by signal SEGV (no mapping at the fault address)
  0x08050a05: bad_memory_access+0x0015:   movb     %al,0x00000000(%edx)
  (dbx) where
  =>[1] bad_memory_access(0x8047ae8, 0x8047a44, ...
    [2] main(0x1, 0x8047a50, 0x8047a58, 0x8047a0c), at 0x8050a1b
  ```

  Based on the above it is possible to say in which function the program crashed. What we cannot see is the exact line in the code. For that, the program has to be compiled with debugging symbols, i.e. "`cc -g coredump.c`".

  ```
  $ cc -g coredump.c
  $ dbx ./a.out core
  Reading a.out
  core file header read successfully
  Reading ld.so.1
  Reading libc.so.1
  program terminated by signal SEGV (no mapping at the fault address)
  Current function is bad_memory_access
      8           x[0] = '\0';
  (dbx)
  ```

GNU debugger gdb

- Usage:
  gdb [*options*] [*program* [*core*]]

- Most common commands:

run [*arglist*]	program start
bt	print stack
print *expr*	print expression
set *var* = *expr*	change value of variable
cont	continue program run
next, step	execute a line (with/out going into function)
break *condition*	set breakpoint
help [*name*]	print help
quit	debugger exit

- GNU analogy of dbx. The dbx compatibility mode can be enabled using the -dbx option.

- Most platforms today offer debuggers with a graphical front end. Often they are implemented on top of gdb.

-
```
#include <stdio.h>
int main(void) {
  printf("hello, world\n");
  return (0);
}
$ cc -g main.c
$ gdb -q a.out
(gdb) break main
Breakpoint 1 at 0x8048548: file main.c, line 4.
(gdb) run
Starting program: /share/home/jp/src/gdb/a.out

Breakpoint 1, main () at main.c:4
4         printf("hello, world\n");
(gdb) next
hello, world
5         return (0);
(gdb) c
Continuing.
Program exited normally.
```

```
(gdb) q
```

- Debuggers are a great help when your program exits with a “segmentation error” – i.e. when accessing memory incorrectly. When the `-g` option is used during compilation, the debugger will be able to show exact line of code where the problem happened. Concrete example (can you tell why the program behaves in such a way?):

```
$ cat -n main.c
     1  int
     2  main(void)
     3  {
     4          char *c = "hey world";
     5          c[0] = '\0';
     6          return (0);
     7  }
}
$ gcc -g main.c
$ ./a.out
Bus error (core dumped)
$ gdb a.out a.out.core
...
Core was generated by ‘a.out’.
Program terminated with signal 10, Bus error.
...
#0  0x080484e6 in main () at main.c:5
5               c[0] = '\0';
```

Contents

- Introduction, Unix and C, programming tools
- **Basic Unix concepts and conventions, its API**
- Access rights, devices
- Process manipulation, program execution
- Signals
- Process synchronization and interprocess communication
- Network programming
- Programming with threads
- Appendix

Standard header files (ISO C)

`stdlib.h`	...	basic macros and functions
`errno.h`	...	error handling
`stdio.h`	...	input and output
`ctype.h`	...	character handling
`string.h`	...	string handling
`time.h`	...	time and date handling
`math.h`	...	math functions
`setjmp.h`	...	far jumps
`assert.h`	...	debugging macros/functions
`stdarg.h`	...	variable arguments processing
`limits.h`	...	implementation dependent constants
`signal.h`	...	signal handling

- The *header file* is a file that contains declarations (*forward declaration*) of functions, variables and macro definitions. From the preprocessor's point of view, this is a simple file in the C language.

- **These header files are not specific to UNIX. They are part of the ISO C standard, that is included in POSIX.1 (page 10) It is important to realize that every system that supports ISO C has to have these files, regardless of whether it supports POSIX.1.**

- The appropriate header file for given function can be looked up using the function's man page, e.g. this is the beginning of `memcpy` man page on Solaris:

```
Standard C Library Functions                       memory(3C)

NAME
   memory, memccpy, memchr, memcmp, memcpy, memmove,  memset  -
   memory operations

SYNOPSIS
   #include <string.h>
...
...
```

- Individual macros contained in these files are usually not explained, their meaning can be looked up in relevant specifications which are on-line. On some systems (Solaris) individual header files have their own manual page (e.g. `man stdlib.h`).

- The `assert` macro is possible to remove during the compilation using the `NDEBUG` define. Example: `assert/assert.c`.

```
cat assert.c
#include <assert.h>

int
main(void)
{
        assert(1 == 0);
        return (13);
}
$ cc assert.c
$ ./a.out
Assertion failed: 1 == 0, file assert.c, line 6
Abort (core dumped)
$ cc -DNDEBUG assert.c
$ ./a.out
$ echo $?
13
```

Standard header files (2)

`unistd.h`	...	standard symbolic constants and types
`sys/types.h`	...	data types
`fcntl.h`	...	file control options
`sys/stat.h`	...	information on files
`dirent.h`	...	directory entry format
`sys/wait.h`	...	waiting for children
`sys/mman.h`	...	memory mapping
`regex.h`	...	working with regular expressions

- These headers are part of the UNIX specification.
- Looking into these header files may be worth your while.

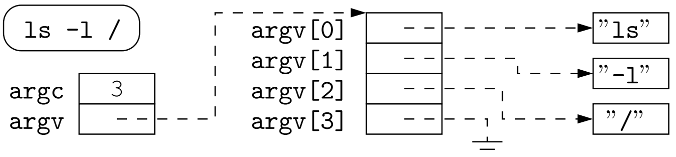
Standard header files (3)

`sys/socket.h`	...	network communication
`arpa/inet.h`	...	definitions for internet operations
`pthread.h`	...	POSIX threads
`semaphore.h`	...	POSIX semaphores
`sys/ipc.h`	...	System V IPC
`sys/shm.h`	...	System V shared memory
`sys/msg.h`	...	System V messages
`sys/sem.h`	...	System V semaphores

- There are also many other standard header files.

Function `main()`

- function `main()` is called upon program execution
- `int` **`main`** `(int` *`argc`*`, char *`*`argv`*`[]);`
  - `argc` . . . number of command line arguments
  - `argv` . . . array of command line arguments
    - `argv[0]` is an executed program name (without path)
    - the last item is `argv[argc] == NULL`
  - returning from `main()` or calling `exit()` terminates the process
  - standard return values are `EXIT_SUCCESS` (0) and `EXIT_FAILURE` (1)



- The first argument (of type `int`) is the number of arguments on a command line, including argument 0 – the program name. The second argument is an array of strings representing the command line arguments. There is a terminating `NULL` as the last array item. Note that `NULL` is different from an empty string.
- To go through all the command line arguments, you can either use *argc* or test for `NULL` in `argv[i]`.
- `argv[0]` is sometimes a source of additional information. For example, commands `cp`, `mv`, and `ln` may be linked to the same executable (Solaris). The value of `argv[0]` then tells the process what function it is supposed to perform. Another example – if the first character of a shell process `argv[0]` is set to "-" it means the shell process is supposed to act as a login shell (check the `bash` man page, section INVOCATION, for more information on what being a login shell means). In the process listing you will see "`-bash`". This convention is not part of the UNIX specification for `sh` but it had already been used in the Bourne shell on UNIX V7 (1979) and other shells followed suit.
- We already know that upon program execution, the dynamic linker eventually passes the control to function `main`, as explained on page 30. If the `main` function is missing, the compilation fails during the linking phase. When `main` finishes, it means the process finishes. You can also use functions `exit()` or `_exit()` from anywhere in the program, i.e. not just from function `main`.
- Passing the environment in the third parameter of type `char**` is not part of the normative part of the C standard, only the informative one. C compilers

typically support that though. The `main` variant with the 3rd parameter looks like this:

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

- The return value type of `main` should be always `int`. **Only lower 8 bits from that integer are used though.** It is always a non-negative number. Note that in contrast to the C convention, the `0` return value means a success in a Unix shell, and a non-zero value means a failure. A typical construct in a shell looks like this:

  ```
  if prog; then
          echo "success"
  else
          echo "failure"
  fi
  ```

  or:

  ```
  prog && echo "success" || echo "failure"
  ```

  Example: `main/return-256.c`.

- Never use `return (-1)` in `main` nor `exit(-1)`. Based on the information in the previous paragraph, from `-1` you will get `255` as the return value you get in the shell in `$?`. It just creates confusion.

- It is very reasonable to use only `EXIT_SUCCESS` (`0`) and `EXIT_FAILURE` (`1`) unless there is a valid reason for other values. Sometimes you might need more values to distinguish between failures. For example, the `passwd` command on Solaris have quite a few of them, go check its manual page if interested, section `EXIT STATUS`. Example: `main/return-negative-1.c`.

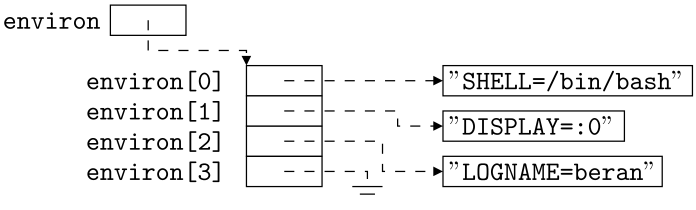
- The difference between function `exit()` and `_exit()` is that `exit` also flushes and closes streams (try it out with `printf()` **without** printing a new line), and calls functions registered via `atexit()`, and possibly other actions based on a specific system. Example: `exit/exit.c`

- Example on printing out command line arguments: `main/print-argv.c`

- If a process is killed by a signal, you can get the signal number from its return value as presented by the shell. See page 127.

Environment variables

- the list of *environment variables* is passed as
  `extern char **environ;`
- it is an array of strings terminated by `NULL` in the format:
  `variable_name=value`



environ

environ[0] ”SHELL=/bin/bash”

environ[1] ”DISPLAY=:0”

environ[2] ”LOGNAME=beran”

environ[3]

- A shell passes an executed program those variables marked as exported (in Bourne-like shells via an internal command `export` *`variable`*). After you export a variable, you do not need to export it again after its value is changed. The command `env` prints current enviroment variables. You can also add a variable to the environment of an executed program without changing the environment of your current shell:

  ```
  $ date
  Sun Oct  7 13:13:58 PDT 2007
  $ LC_TIME=fr date
  dimanche  7 octobre 2007 13 h 14 PDT
  ```

  It may not work like that in your shell as very probably your system will not have a French localization package installed.

- When replacing the current process image with a different program (page 106), the child will by default, get the full environment from its parent. You can pass a different array as an argument to the function `exec()`, including passing an empty one.
- Different commands use different environment variables. That should be documented in their respective manual pages, sometimes in a separate section named *ENVIRONMENT* or *ENVIRONMENT VARIABLES*.
- For example, the `man` command uses `PAGER`, `vipw` uses `EDITOR`, etc. The variables are usually in caps but it is just a convention. `wget` does not follow

convention and uses `http_proxy` and `https_proxy` as environment variables to set the proxy to be used, and other lower case variables as well.

- If `envp` is the 3rd argument of `main`, it is the same pointer as is in the standard global variable `environ`.

- Example: `main/print-env.c` (see below how to clear out inherited environment from the shell using the `env` command):

```
$ cc print-env.c
$ env - XXX=yyy aaa=ABC ./a.out
aaa=ABC
XXX=yyy
```

Manipulating the environment

- it is possible to replace `environ` with a different array but in general you should not do that
- `char *`**`getenv`**` (const char *`*`name`*`);`
  - return value of *name*
- `int `**`putenv`**` (char *`*`string`*`);`
  - inserts a string *`name=value`* into the environment (adds a new or modifies an existing variable)
- changes are propagated into children upon their creation
- there are also functions **setenv**() and **unsetenv**()

- Read the description for `environ` in SUSv4 before assigning a new value to the variable as there is important information there you should know.

- When using `putenv`, the string will become part of the environment, but nothing is copied. You must not use automatic variables for such strings. Use `setenv` to copy the value into the environment. Example: `main/putenv.c`.

- Changes in a child never changes its parent environment as those arrays reside in separate address spaces.

- Another difference between `putenv` and `setenv` is that in `setenv` one can say whether an existing variable should be overwritten or not. Function `putenv` always overwrites an existing variable.

- Example:

```
int
main(void)
{
        printf("%s\n", getenv("USER"));
        return (0);
}
$ ./a.out
janp
```

- As `environ` is just an array of pointers, you may find code that directly manipulates it. However, any application that directly modifies the pointers to which the `environ` variable points has undefined behavior according to SUSv4, and that is possibly exacerbated if the functions introduced above are used while doing that. Do not write code like that.
- Example: `main/getenv.c`
- Note that there is a difference between setting a variable to an empty string and removing the variable from the environment via `unsetenv`.

Command arguments processing

- common notation for shell: `program -option arguments`
- options are in the form of `-x` or `-x` ***`value`***, where `x` is alphanumeric, ***`value`*** is an arbitrary string
- multiple options can be unified: `ls -lRa`
- ’`--`’ or first argument not starting with ’`-`’ marks the end of options and arguments following are not considered options even if they start with ’`-`’.
- this form of arguments is required by the standard and can be processed using the `getopt` function.

- The arguments can of course be processed using custom functions, however the standard function is sufficient for the overwhelming majority of use cases.
- The options can be repeated, however it makes sense only in specific cases.

- The ordering of the options can be important and it depends on the application to specify the ordering.

- The UNIX standard defines 13 rules that very precisely define the naming of commands and the format of options. For example the name of the command should be lower case, 2–9 characters long and using only characters from the portable character set. Arguments without options should be possible to group after '–', etc.

- Using numbers for options (e.g. `-4` and `-6` in some `ping` implementations) is old fashioned; reportedly SUSv3 mentions that.

- Watch out for GNU command line utilities and their strange permutations of options and arguments.

- The `-W` option should be reserved for vendor options, i.e. for non-portable extensions.

- When you used all the letters of the alphabet for options, something is not right with the design of the command. One solution would be to split the processing and introduce the notion of sub-commands. See for example the `zfs` command on systems with the ZFS file-system.

Options processing: `getopt()`

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

- the function receives command line parameters and processes and returns one option. If a given option has an argument, it is set in the `optarg` variable.
- when all options are processed, it returns -1 and the `optind` variable contains the index of first unprocessed argument of the program.
- possible options are passed in the `optstring` string. If an option has mandatory argument, its character is followed by '`:`' in the string.
- on error (unknown option, argument missing), the function returns '`?`', saves the option character into the `optopt` variable.

- Usually the `getopt` function is used first to process options and then the rest is processed using custom functions; the remaining arguments are often file names and such.

- By convention the options in the `optstring` variable are sorted.

Example of using getopt()

```
struct {
    int a, b; char c[128];
} opts;
int opt; char *arg1;

while((opt = getopt(argc, argv, "abc:")) != -1)
    switch(opt) {
        case ’a’: opts.a = 1; break;
        case ’b’: opts.b = 1; break;
        case ’c’: strncpy(opts.c, optarg,
            sizeof (opts.c) - 1);
            opts.c[sizeof (opts.c) - 1] = ’\0’; break;
        case ’?’: fprintf(stderr,
            "usage: %s [-ab] [-c Carg] arg1 arg2 ...\n",
            basename(argv[0])); break;
    }
arg1 = argv[optind];
```

- It is a good custom while detecting an unknown switch or incorrect use of options to write a simple usage message, optionally with a reference to documentation, and exit the program with error, i.e. non-zero return value.

- Note how the `opts.c` string is properly terminated. Utmost care is necessary when processing potentially unbound output, especially when handling options of a program running with elevated privileges.

- It is evident that `getopt` is a stateful function. In order to process the next array of arguments or start from scratch, it is necessary to set the `optreset` variable to 1.

- The standard version of `getopt` retains the order of arguments when processing.

- When a undefined option is used, `getopt` will print an error; this can be suppressed by setting the `opterr` variable to 0.

- Example: shell script `getopt/getopts.sh` rewritten to C language using the `getopt` function: `getopt/getopt.c`

Processing long options

- first appeared in GNU library `libiberty`:

  `--name or --name=value`

- the arguments are permuted so that e.g. `ls * -l` is the same as `ls -l *`, standard behavior can be enabled by setting the `POSIXLY_CORRECT` environment variable.
- the long options are processed using the **getopt_long**`()` function using these variables and structures:

```
struct option {
    const char *name; /* name of the option */
    int has_arg; /* value: yes, no, optional */
    int *flag; /* if NULL, the function returns val,
                  otherwise returns 0 and *flag = val */
    int val; /* return value */
};
```

The version that appeared in FreeBSD (getopt_long is not part of the standard), has the following properties:

- If all long options have the short variant set in `val`, the behavior of `getopt_long` is compatible with that of `getopt`.
- It is also possible to specify the value of a long option with a space (for example `--color green`).
- If `flag` is set, `getopt_long` returns 0, which makes these long options without the short variant get processed in a single branch of `case`.
- There is also `getopt_long_only`, which allows long options to begin with a single dash (`-option`).
- The `getopt_long` function is possible to use in two ways. The first way is that each long option has a corresponding short variant – this way it is possible to introduce long options to existing programs in a **getopt compatible** way. The second way makes it possible to have independent long options. In this case the function always returns 0 (non-compatible with `getopt`) and the `*flag` variable is set to `val`.

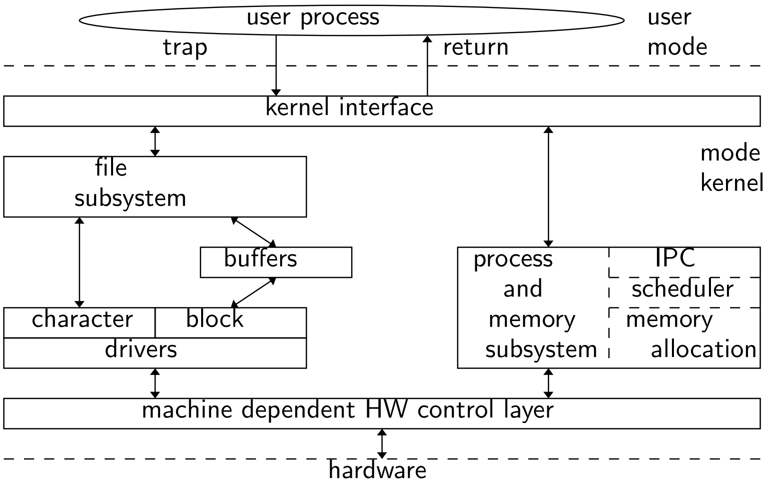
Processing long options (cont.)

```
int getopt_long(int argc, char * const argv[],
                const char *optstring,
                const struct option *longopts,
                int *longindex);
```

- `optstring` contains short options, `longopts` contains the address of an array of structures for long options (last record contains all zeroes)
- if the function hits a long option, it returns the corresponding `val` or zero (if `flag` was not `NULL`), otherwise the behavior is the same as `getopt`.
- it will also put the index of the found option in `longopts` to `*longindex` (if not `NULL`)

- See `getopt/getopt_long.c`

Classic UNIX structure



- This scheme is taken from [Bach86]. It emphasizes two central terms of the system model in UNIX – files and processes. **In modern Unixes it looks different but for the time being such model will suffice.**

- UNIX differentiates between two modes of CPU: *user mode* and *kernel mode.* In user mode the privileged intructions are not available (e.g. memory mapping, I/O, interrupt masking). These modes have to be supported by the hardware (CPU).

- The processes usually spend most of the time running in user mode. They enter kernel mode either by using a synchronous interrupt (trap) for calling kernel services or asynchronously (clock, I/O). In kernel mode, the exceptional states are handled (page fault, memory protection failure, unknown instruction etc.). Some special tasks are handled by system processes running in kernel mode all the time.

- The classic UNIX kernel is monolithic. Originally it was necessary to regenerate the kernel (i.e. compile from source code and link) whenever some kernel parameter had to be changed or add a device driver. In modern implementations it is possible to set many kernel parameters dynamically using system utilities. Also, many Unix systems extend kernel services by using *loadable kernel modules.* For example FreeBSD 5.4-RELEASE has 392 such modules.

- There are two ways to handle peripheral devices: *block devices* and *character/raw devices.* Block devices (e.g. disk drives) pass the data through *buffers* in blocks; character devices (e.g. terminals) make it possible to work with individual bytes and do not use buffering.

- **The kernel is not an individual process**, rather it is part of each user process. When kernel is executing, it is usually user space performing some action in kernel mode.

Processes, threads, program

- a **process** is a system object characterized by its context, identified by a unique number (**process ID**, **PID**); in other words ,,code and data in memory”
- a **thread** is a system object that exists inside a process and is characterized by its state. All threads within a single process share the same memory area sans registers and stack; ,,line of execution”, ,,what is running”
- a **program** is a file with a precisely defined format that contains instructions, data and service information needed for execution; ,,executable file on disk”

○ **memory** is assigned to **processes**.

○ **processors** are assigned to **threads**.

○ threads of single process can run on different processors.

- The ***context*** is the memory of a process, register contents and the kernel data structures relevant to that process.
- In other words – the context of a process is its state. When the system is executing the process, one says that it runs in the process context. The classic kernel handles interrupts in the context of interrupted process.
- The threads were introduced to UNIX later - originally there were just processes that from today’s perspective had only single thread. The possibility of using multiple threads was introduced because it was shown that it is beneficial to have multiple lines of execution within the same process on commonly shared data.
- The memory areas of processes are isolated between each other, however processes can communicate or explicitly share memory.
- While processes are kernel entities, threads can be partially or fully implemented as user libraries. In the latter case, this means that the kernel does not have to support threading at all. Threads have smaller overhead than processes.

- A system process that is running for the whole time and provides some system services (`inetd`, `cron`, `sendmail`...) is called a *daemon*. BSD system mascot is not a devil but a daemon.
- As Rob Pike mentioned in one of his lectures, this is a concurrency model that can lead to parallelism.

kernel, modes, interrupts/traps (classic UNIX)

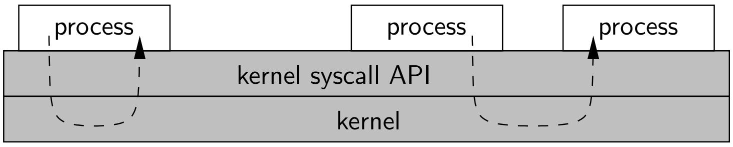
- processes typically run in user mode
- a system call will cause a switch to kernel mode
- a process has one stack for each mode
- the kernel is part of each user process, it is not a standalone process(es)
- switching to different process is called *context switching*
- interrupt handling is performed in the context of interrupted process
- the classic kernel is non-preemptive

- **The kernel is not a set of processes running in parallel with user processes, rather it is part of each user process.**
- The transition between user and kernel mode is not a context switch – the process is still running in the same context.
- The interrupted process is not necessarily the one that caused it.
- In kernel mode, a process can access kernel memory that is not accessible from user mode. Likewise, it can execute privileged instructions that would otherwise cause errors in user mode (like state register manipulations).
- An interrupt routine cannot block because it would block the process; a process can only block on its own. Modern Unixes use interrupt threads, in their context the drivers **can** block.
- The fact that the classic UNIX kernel is non-preemptive means that **one process cannot block another process**.
- An interrupt can come while handling another interrupt. If its priority is higher, it is accepted by the CPU. The sequence of accepted interrupts is saved in an *interrupt context stack*.

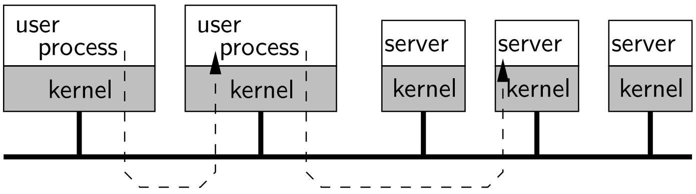
- **Modern kernels are very different in terms of interrupt handling, kernel preemption, etc.**

System calls and inter-process communication

- UNIX



- distributed OS



- If a Unix process is required to perform a system task, it will pass the control to the kernel using a system call. The kernel is code shared between all processes but accessible only to those processes that are running in kernel mode. The kernel is therefore not a standalone privileged process, it is still running in the context of a process (one that requested a system service via a system call or that was running when an interrupt came).

- Inter-process communication in UNIX is achieved using system calls, it is therefore handled by the kernel.

- There can be system processes called *kernel threads*, that are running exclusively in kernel mode. The majority of system processes run in user mode differ from the rest in that they have elevated privileges. The process scheduler switches between processes and makes it possible to run multiple processes simultaneously even on single processor system. Multi-processor systems enable true parallelism of processes and threads (it is possible for a thread to migrate between processors based on scheduling).

- In distributed operating systems the kernel is in the form of microkernel, i.e. it provides only the very basic services like processor programming, memory allocation and inter-process communication. Upper system services that are part of the kernel in UNIX (e.g. file system access) are implemented as special processes (servers) running in user mode. The kernel passes the request of a user process to a relevant server that can be running on a different network node.

- There are many microkernels today, e.g. Minix (unix-like system) or HURD that runs above the Mach micro-kernel.

System calls, functions

- In UNIX a distinction is made between **system calls** and **library functions**. This division is maintained in man pages: section **2** contains system call man pages and section **3** library functions.
  - library functions are executed in user mode, just like the rest of the program's code.
  - system calls also have the form of a function. However, a given function only processes arguments and passes the control to the kernel using synchronous interrupt instruction. Once it returns from the kernel, it will adjust the result and return it to the caller.
- this distinction is not made in the standard – from programmer's perspective it does not *usually* matter if a given function is processed by a library or kernel.

- The transition from userland to kernel can be costly; if the program executes lot of syscalls, it can have a negative effect on its performance. **A library function can but does not have to perform some system calls, however it always does some non-trivial work in user mode.**
- It is possible to perform system calls directly in the assembler.
- The kernel API is defined w.r.t. function calls of the standard library, not w.r.t. interrupt level and data structures used by these functions when passing control to the kernel. The mechanism of switching between user and kernel mode can differ not only depending on the hardware platform but also between different versions of the same system on the same hardware.

System call return values

- integer return value (`int`, `pid_t`, `off_t`, etc.)
  - `>= 0` ... success
  - `== -1` ... failure
- pointer return value
  - `!= NULL` ... success
  - `== NULL` ... failure
- after a failed syscall, the error code is stored in global variable `extern int` **`errno`**`;`
- a successful syscall never changes `errno`! It is therefore necessary to test the return value first and then check `errno`.
- error messages depending on the `errno` value can be printed with `void` **`perror`**`(const char *`*`s`*`);`
- textual representation for a given value is returned by `char *`**`strerror`**`(int` *`errnum`*`);`

- In Solaris, `errno` is in reality defined in `libc` as a dereferenced pointer to an integer (specific to a userland thread) and the value is set right after the instruction for the system call. For example, on the i386 architecture the `errno` value is stored in the `eax` register after the return from the syscall (after the `sysenter` instruction is executed). Before the instruction, the register held the system call number. It is therefore the `libc` library that is responsible for the program to see the correct `errno` value.

- The POSIX thread functions (`pthread_*`) do not set `errno`, rather they return either zero or an error code.

- For some calls, the value `-1` is semantically valid. To use such functions, it is necessary to set `errno = 0` before the call and check whether the value changed after the call. E.g. the `strtol` function returns 0 on failure, which can also be a valid value in some cases (and -1 is also a valid result value).

- It is therefore necessary to always read the appropriate man page for the system call or library function.

- Note that to test for failures of the functions from `stdio.h`, it is necessary to test using the function "`int` **`ferror`**`(FILE *`*`stream`*`)`" as it is not otherwise possible to distinguish between an error and end of a stream. Considering we are not using these functions in the lecture (except for `printf` and `fprintf`), you should not need it.

Formatted error messages: `err(3)`

- functions to display a formatted error message and optionally exit
- instead of **perror**() and **exit**(), one function suffices
- `void` **`err`**`(int` *`status`*`, const char *`*`fmt`*`, ...);`
  - prints a program name, your formatted string, and error message based on `errno`
  - exits the program with *status* return value
  - use **errx**() if `errno` is not set

  `void` **`warn`**`(const char *`*`fmt`*`, ...);`
- – same as **err**() but does not exit
- see the manual page for other functions from the same family
- originated in 4.4BSD

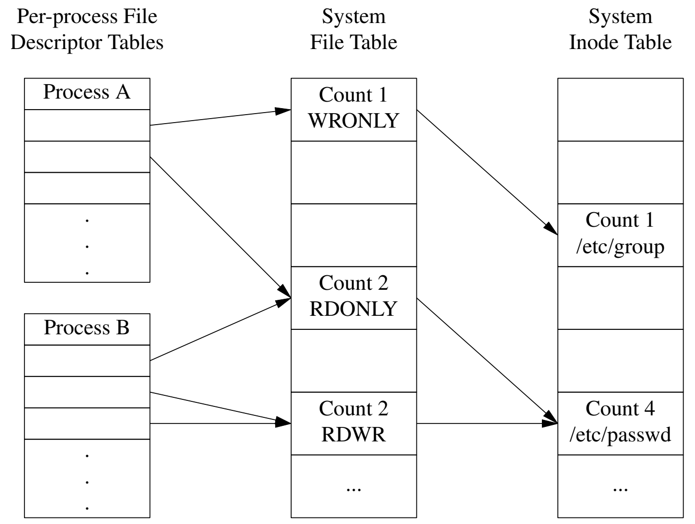
- The **errx**() function behaves as **err**() but does not use `errno` to print that extra error message. Similarly, **warnx**().
- These functions are very handy especially for smaller programs as they are easy to work with and save you some lines of code. More complex applications often use their own logging functions.
- Not part of the UNIX specification but in general you can find them almost everywhere.
- Example: `err/err.c`

File API

- before working with a file, it must be first opened via **open**() or **creat**()
- open files are accessible via *file descriptors*, numbered from 0. More descriptors can share the same file opening (read/write mode, position).
- standard file descriptors
  - 0 ... standard input (read only)
  - 1 ... standard output (write only)
  - 2 ... unbuffered error output (write only)
- for reading and writing a file: **read**(), **write**()
- position change: **lseek**(), close: **close**(), information: **stat**(), file control: **fcntl**(), access rights: **chmod**(), ...

- Every function that allocates file descriptors (not just **open**() or **creat**() but also **pipe**(), see page 112, and **dup**(), see page 70, for example), always uses the first available descriptor number. That is very important and will be later used when we work with pipes and redirect process input and output.
- A process inherits file descriptors from its parent so it does not have to open already open files. Usually at least file descriptors 0, 1, and 2 are provided.
- Functions from a header file `stdio.h` (e.g. **fopen**(), **fprintf**(), and **fscanf**()), and their file handle `FILE` are defined in the standard `libc` library and use standard system calls like **open**(), **write**(), and **read**(). From those functions, we will only use functions for printing to the terminal output like **fprintf**().

System View of Open Files



- This is a simplified view of the kernel tables that deal with files. It is modeled after a similar picture in [Bach]. Today it is more complicated but the main idea stays.

- Each process on the system has its *file descriptor table.*

- Slots in that table point to the *system file table.* There is only one such a table in kernel. This table carries the opened file mode and also the **current file position.** Each `open` call will create new entry in the table. However, file descriptor duplication or inheritance will lead to sharing of the slots.

- From the system file table, it is pointed to the *system inode table.* Nowadays the table contains so called *vnodes – virtual nodes* but that is not relevant for us now. For more info, see page 223.

- The system file table represents an additional level of indirection so that different processes can share the file position.

- When opening a file via an `open` call, a new slot in both the file descriptor and system file tables is always allocated. File position sharing in the same process is achieved via file descriptor duplication where more descriptors share the same system file table slot. If a file position sharing is needed among multiple processes, that is achieved via the `fork` call, and it is explained on page 114.

- To see how the file descriptor table looks like for given process, use `lsof` (Linux distributions, macOS) or `pfiles` (Solaris) programs. They will come handy for debugging file descriptor leaks or inheritance/redirection issues.

Opening a file: `open()`

```
int open(const char *path, int oflag, ... );
```

- opens a file `path`, returns its file descriptor. The `oflag` is an OR combination of the following flags:
  - `O_RDONLY`/`O_WRONLY`/`O_RDWR` ... open for reading, writing, or both
  - `O_APPEND` ... append only
  - `O_CREAT` ... create the file if it does not exist
  - `O_EXCL` ... fail if the file exists (for use with `O_CREATE`)
  - `O_TRUNC` ... truncate the file (write permission needed)
  - ...
- with `O_CREAT`, the third parameter *mode* defines the access mode for the newly create file

- The first available file descriptor is always used.
- When `O_CREAT` is used, the *mode* is modified using the current mask that can be changed via a shell command `umask` – those bits in *mode*, also set in the process umask, are nullified. The default umask value is typically (and historically) `022`. We recommend that you always set it to `077` in your profile script. Never do that for root though otherwise you will end up with a system in a non-supported configuration – installed software will not be possible to run by non-privileged users, what worked before may stop working, etc.
- If the *mode* argument is required and not specified, you get whatever is on the stack or CPU register. Both flags and mode are stored in the system file table, see page 60.
- Macros for use with *mode* can be usually found in the manual page for `chmod(2)`, and you can find them also in the `sys/stat.h` header file (or in another file included from it) where the standard requires them to be.
- A no longer needed slot in the file descriptor or system file table is zeroed out before its reuse.
- There are other flags as well:
  - `O_SYNC` (`O_DSYNC`, `O_RSYNC`) ... the call returns after the data is physically stored (synchronized I/O). `O_DSYNC` is for writing synchronization only, `O_RSYNC` for reading.

  - `O_NOCTTY` ... when opening a terminal by a process without a controlling terminal, the terminal being opened does not become one.
  - `O_NONBLOCK` ... if reading or writing cannot be satisfied right away, calls `read`/`write` will fail instead of getting blocked and waiting for the completion. `errno` is set to `EAGAIN` in such a case.

- One cannot use `O_RDONLY | O_WRONLY` for both reading and writing as historically, implementations used 0 for the read-only flag. The standard defines that only one of those three flags may be used.

- Is is possible to open and create a file for writing so that writing is disallowed by its mode. It will work for the initial file opening but any subsequent attempts to write will fail.

- You need write permission to use `O_TRUNC`.

- The behavior of `O_EXCL` without using `O_CREAT` at the same time is undefined.

- For file locking, the `fcntl` call is used, see page 72.

Creating a file

`int` **creat**`(const char *`*path*`, mode_t` *mode*`);`

- the function is equivalent to:
  `open(path, O_WRONLY|O_CREAT|O_TRUNC, mode);`

`int` **mknod**`(const char *`*path*`, mode_t` *mode*`, dev_t` *dev*`);`

- creates a device special file

`int` **mkfifo**`(const char *`*path*`, mode_t` *mode*`);`

- creates a named pipe

- The `open` call allows opening of a regular file, device, or named pipe. However, it (and `creat` as well) can only create a regular file, so you need the other two calls for non-regular files.

- The test of a file's existence using the flag `O_EXCL` and its subsequent creation if it did not exist, is an atomic operation. You can use that for lock files but only with the `open` call, not `creat`, because `creat` lacks the flags specification.

- You need extra privileges to create device special files (e.g. to be a root).

Reading and writing files: `read()`, `write()`

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

- attempts to read *nbyte* bytes of data from the object referenced by the descriptor *fd* into the buffer pointed to by *buf*
- returns the number of bytes actually read, 0 on `EOF`, -1 on error

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

- attempts to write *nbyte* of data to the object referenced by the descriptor *fd* from the buffer pointed to by *buf*
- number of bytes which were written is returned, -1 on error

- For any Unix system, a file is just a sequence of bytes without any inner structure.
- The **behavior of `read` and `write` depends on the type of the file** (regular, device, pipe, or socket) and whether the file is in a blocking or non-blocking mode (flag `O_NONBLOCK` on file opening, see page 62).
- For both calls there are quite a few situations that can happen, and which are listed in the next few paragraphs. If unsure, consult the standard.
- `read` returns a non-zero number of bytes less than *nbyte* if less then *nbyte* bytes remain in a file, if the call was interrupted by a signal, or if the file is a pipe, device, or socket and there is less than *nbyte* bytes available at the moment. If there is no data, a blocking `read` will block unless some data gets available, a non-blocking `read` returns -1 and sets `errno` to `EAGAIN`.
- `write` returns a non-zero number of bytes less than *nbyte* if less then *nbyte* bytes can fit into the file (e.g. disk full), if the call was interrupted by a signal, or if `O_NONBLOCK` was set and only part of the data fits into a pipe, socket, or a device; without `O_NONBLOCK` the call will block until all the data can be written. If nothing can be written, a blocking `write` blocks until writing data is possible, a non-blocking call returns -1 and sets `errno` to `EAGAIN`.
- **Important exceptions for pipes** are listed on page 66.
- If `read` or `write` returns a non-zero number less than `nbyte` due to an error, a repeated call returns -1 and sets `errno`.

- If `read` or `write` are interrupted before they manage to read or write, respectively, at least one byte, the call returns -1 and sets `errno` to `EINTR`. Note that there is a difference if the call manages to read or write, respectively, at least one byte – see the paragraphs above.

- `O_APPEND` guarantees an atomic write to the end of a file on a local filesystem, i.e. **every** write will add data to the file end (so called *append-only* file).

Closing a file: `close()`

`int` **close**`(int` *fildes*`);`

- releases `fildes`, if it was the last descriptor for a file opening, closes the file
- if the number of links is 0, the file data is released
- if the last pipe descriptor is closed, any remaining data is lost
- on process termination, an implicit `close` is called on all descriptors

- Note that the file data is released only after the number of links gets to 0 **and** if the file was opened before, when closing it. It means that you can `rm` all hard links to a specific file but if a process has the file still open, the file data remains on the disk until the process closes the file or terminates.

- If a process needs a temporary file, it can create it, `unlink` it right away, and work with it using the existing file descriptor. When the file descriptor is closed (and all of its possible duplicates), the file data is released.

- Even `close` may fail. For example, some filesystems may write the data on file closure, and if that write fails, `close` fails.

- If you forget to close file descriptors when you no longer need them, and it is a long living process, a daemon perhaps, depending on the system configuration you may hit memory starvation as the system memory will be filled with the ever growing file descriptor table.

- A very simple `cat(1)` program: `read/cat.c`

Example: copy files

```
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    char buf[4096];
    int inf, outf;
    ssize_t ilen;

    inf = open(argv[1], O_RDONLY);
    outf = creat(argv[2], 0666);
    while ((ilen = read(inf, buf, sizeof (buf))) > 0)
            write(outf, buf, ilen);

    close(inf); close(outf);
    return (0);
}
```

- The example above is complete. You can cut-and-paste, compile, and run it. Obviously, due to the slide size constraint, it is missing several error checks otherwise needed for a well behaved utility, like checking `argc` before using `argv`, or checking return values for `open`, `creat`, `read`, and `write`, and reporting possible errors.

- It is not efficient to read and write by a few bytes as each system call costs a certain overhead, independent on the size of the processed block. It is much better to read and write larger blocks, say 8-1024KB. You can `strace(1)` a `cat` command on Linux to see what blocks they use for `read`. It may be around 128 kilobytes. The following example, `read/cat.c`, has an option `-b` for setting the read block size; you can try different block sizes, including 1, on a reasonable big file (use `mkfile(1)` on BSD/macOS or `fallocate(1)` on Linux to create an arbitrarily large file), and use `time(1)` to measure the system call overhead.

- If you need to work with little pieces of data read from files, you can use stream oriented functions that internally buffer the data – `fopen`, `fread`, ... However, please do not use those functions in your semester assignment and at the exam.

- Note that we always only write as many bytes as we actually read. See page 63 for more information on when we can read/write less data than requested.

Working with a named pipe (FIFO)

- it may not be possible to create a FIFO on a distributed filesystem (e.g. NFS or AFS)
- you need to know the semantics of opening a FIFO
  - opening a FIFO for just reading will block until a writer (aka producer) shows up, unless one already exists.
  - opening a FIFO for just writing will block until a reader (aka consumer) shows up, unless one already exists.
  - this behavior can be adjusted using a flag `O_NONBLOCK`
- semantics for reading and writing is a bit more complicated, see the notes below this slide.
  - same as for a conventional, unnamed, pipe (will be later)

- A named pipe is created using the system call `mkfifo`, see page 62. Using an unnamed pipe is described later on page 113.
- A consumer is a process that opens a file/pipe for reading, a producer opens a file/pipe for writing.
- It is possible to open a named pipe for reading and writing at the same time. The same process thus can write to the pipe and then read the same data from it.
- If the pipe is not yet open for writing by any process, to open the pipe for reading without blocking on the call, one needs to use `O_NONBLOCK`, see page 62. Without that flag the consumer will block waiting for a producer.
- However, trying to read a previously opened pipe that has no producer results in a return value of 0, indicating the end of file – the process will not block waiting for a producer. It is irrelevant whether the pipe was opened in a blocking or non-blocking mode.
- When writing to a pipe without a consumer (i.e. the producer opened the pipe when there was at least one existing consumer), the kernel will send the producer a signal `SIGPIPE` ("broken pipe"). See the following example. For simplicity, we are using an unnamed pipe but a named pipe would behave in the same manner. The `date(1)` command never reads anything from its standard input so it is guaranteed that the producer, `dd(1)`, will be writing to a pipe without a consumer. If a process is killed by a signal, the shell provides a signal number added to 128 as its return value, and `kill -l` understands that:

```
bash$ dd if=/dev/zero | date
Sun Mar  2 01:03:38 CET 2008
bash$ echo ${PIPESTATUS[@]}
141 0
bash$ kill -l 141
PIPE
```

Another possibility with named pipes:

```
$ while [ 1 ]; do cat /etc/passwd; done >> /tmp/bigpasswd
^C
$ read < fifo &
[1] 65946
$ cat /tmp/bigpasswd > fifo
[1]+  Done                    read < fifo
$ echo $?
141
```

- When opening a pipe for writing only with `O_NONBLOCK` and without an existing consumer, the call returns -1 and `errno` is set to `ENXIO`. This asymmetry in opening a pipe for reading in non-blocking mode is due to the fact that it is not desirable to have data in a pipe that may not be read in a short period of time. The Unix system does not allow for storing pipe data for an arbitrary length of time. Without the `O_NONBLOCK` flag, the process will block while waiting for a consumer. By asymmetry, we mean that the system allows consumers without producers but it tries to avoid writers without existing readers.

- If you want to create a process that sits on a named pipe and processes data from producers, you need to open it with the flag `O_RDWR` even if you do not intend to write to it. If you do not use the flag, you might end up with `read` returning 0 after all producers, perhaps only temporarily, disappear, which could be solved by busy waiting. A much better solution would be to use the `select` call, see page 176.

- Writing data of length `PIPE_BUF` bytes or less (`limits.h`) is guaranteed as atomic, i.e. data will not be intermingled with data written by other writers. For example, on Linux kernel 4.x it is 4096 bytes, on Solaris 11 it is 5120 bytes, and on FreeBSD 8.2 it is only 512 bytes. It is obvious from the above that if you write less or equal than `PIPE_BUF`, you always write the whole data or fail, and if `O_NONBLOCK` is set and the whole data buffer cannot be written (for example, a pipe can hold only a limited number of bytes), the call will fail.

- A pipe has no file position, you always append data to a pipe.

- All the information here applies to a unnamed pipes as well, see page 113.

Setting file position: `lseek()`

`off_t` **lseek**(`int` *fildes*, `off_t` *offset*, `int` *whence*);

- will set the file offset for reading and writing in an already opened file associated with a file descriptor *fildes*
- based on value of `whence`, the file offset is set to:
  - `SEEK_SET` ... the value of *offset*
  - `SEEK_CUR` ... current position plus *offset*
  - `SEEK_END` ... size of the file plus *offset*
- returns the resulting offset (i.e. from the file beginning)
- `lseek(fildes, 0, SEEK_CUR)` only returns the current file position

- The first byte is at position 0. If it makes sense, you may use a negative number for setting the *offset*. Example: `read/lseek.c`.
- It is legal to move beyond the end of the file. If data is written there, the file size will be set accordingly, the "holes" will be read as zeros. Note that just changing the file position will not increase the file size.
- You can get the file size via `lseek(fildes, 0, SEEK_END)`.
- The most common operations with `lseek` are three: setting the position from the beginning of a file, setting the position to the end of a file, and getting the current file position (0 with `SEEK_CUR`).
- There is no I/O involved when calling `lseek`.
- You can obviously use the return value of `lseek` not only for subsequent calls to `read` and `write` but also for another call to `lseek`.
- Beware of files with holes as it may lead to problems with backing up the data. Example: `read/big-file.c` demonstrates that moving a sparse file may end up in an actual storage data occupation increase. It greatly depends on the system you run, what archiving utility is used, and their versions. Some utilities provide the means to preserve holes, for example, `dd` with `conv=sparse`, `tar` with `-S`, `rsync` with `--sparse`, etc.
- Beware of confusing the parameters. The second line below looks OK but the arguments are in reversed order. What is more, `SEEK_SET` is defined as 0

and `SEEK_CUR` is 1, so the file position is not moved which is not by itself a disastrous thing, which makes it more difficult to find it:

```
lseek(fd, 1, SEEK_SET);
lseek(fd, SEEK_SET, 1);    /* WRONG!!! */
```

Change file size: `truncate()`

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

- causes the regular file to be truncated to a size of precisely *length* bytes.
- if the file was larger than *length*, the extra data is lost
- if the file was previously shorter, it is extended, and the extended part reads as null bytes

- Truncating the file when opening it can be achieved via the `O_TRUNC` flag in `open`, see page 61.

Descriptor duplication: `dup()`, `dup2()`

`int` **dup**`(int` *`fildes`*`);`

- creates a copy of the file descriptor *fildes*, using the lowest-numbered unused descriptor. Returns the new descriptor.
- same as `fcntl(fildes, F_DUPFD, 0);` (will be later)

`int` **dup2**`(int` *`fildes`*`, int` *`fildes2`*`);`

- duplicates `fildes` to `fildes2`.
- almost the same as
  `close(fildes2);`
  `fcntl(fildes, F_DUPFD, fildes2);`

- We already know that the first available file descriptor is used when opening and creating file, see page 61.
- The original and duplicated file descriptors share the same slot in the system file table (see page 60), i.e. they share the file position and read/write mode.
- The equivalent for `dup2` is not fully equivalent. If *fildes* is equal to *fildes2*, `close(fildes2)` does not happen and `dup2` returns `fildes2` right away.

Example: implement shell redirection

- `$ program <in >out 2>> err`

```
close(0);
open("in", O_RDONLY);
close(1);
open("out", O_WRONLY | O_CREAT | O_TRUNC, 0666);
close(2);
open("err", O_WRONLY | O_CREAT | O_APPEND, 0666);
```

- `$ program >out 2>&1`

```
close(1);
open("out", O_WRONLY | O_CREAT | O_TRUNC, 0666);
close(2);
dup(1);
```

- Note the flag `O_APPEND` used to implement a redirection `>>`.
- Another example of `dup` use will be provided when we start working with pipes. The first redirection example from the slide (without `stderr`) is in `read/redirect.c`. In that example, the `execl` call replaces the current process image with the program passed in the first argument. We got ahead of ourselves here though, we will learn about the `exec` calls on page 106.
- To fully understand how redirection works it is good to draw the file descriptor table for each step and where the slots point to. In the 2nd example in the slide above, we have the initial state, after `close(1)` and `open("out", ...)`, and the final state, as follows:

```
+--------+              +--------+              +--------+
|   0    +-> stdin      |   0    +-> stdin      |   0    +-> stdin
+--------+              +--------+              +--------+
|   1    +-> stdout ==> |   1    +-> "out"   ==> |   1    +-> "out"
+--------+              +--------+              +--------+    /
|   2    +-> stderr     |   2    +-> stderr     |   2    +---'
+--------+              +--------+              +--------+
```

- You need to pay attention to the state of descriptors. The 2nd example above will not work if the descriptor 0 is already closed, as `open` returns 0 (the first available descriptor) and `dup` fails while trying to duplicate an already closed descriptor. Possible solutions:

```
close(1);
if ((fd = open("out", O_WRONLY | O_CREAT | O_TRUNC, 0666)) == 0)
        dup(0);
close(2);
dup(1);
if (fd == 0)
        close(0);
```

or

```
fd = open("out", O_WRONLY | O_CREAT | O_TRUNC, 0666);
if (fd != 1) {
        dup2(fd, 1);
        close(fd);
}
dup2(1, 2);
```

Manipulate file descriptors and devices: `fcntl()`, `ioctl()`

int **fcntl**(int *fildes*, int *cmd*, ...);

- file descriptor duplication
- setting descriptor and file status flags
- advisory and possibly mandatory locking

int **ioctl**(int *fildes*, int *request*, ...);

- manipulates the underlying device parameters of special files
- used as a universal interface for manipulating devices
- each device defines a set of requests it understands

- Example: close standard error output on program execution (exec call): `fcntl(2, F_SETFD, FD_CLOEXEC);`
- We will see later that descriptors are used for sockets as well, so it is possible to use `fnctl` on them, for example to set a non-blocking socket. See page 160 for more information.
- Values for *cmd* in `fcntl`:

  - F_DUPFD … duplicate descriptors
  - F_GETFD … get file descriptor flags. FD_CLOEXEC is the only descriptor flag defined in SUSv4.
  - F_SETFD … set file descriptor flags
  - F_GETFL … get file opening flags (read/write, append, …)
  - F_SETFL … set file opening flags (O_APPEND, O_DSYNC, O_NONBLOCK, O_RSYNC, and O_SYNC). Setting RO/RW is not allowed, and neither are flags for creation, truncation, and exclusive access.
  - F_GETLK, F_SETLK, F_SETLKW … for file locking

- Note that there are two kinds of flags. Flags associated with the file descriptor, having FD in the macro names (only one such exists – FD_CLOEXEC), and flags associated with the slot in the system file table of opened files, having FL in the fcntl command macros (i.e. F_GETFL). See also the picture on page 60.

- Devices may support reading and writing using read and write, and mapping files to memory (mmap, see page 116), all other operations on devices (e.g. setting parameters, locking, or eject) are performed using the ioctl system call.

- When setting flags, always get the present flags first. Even when you know the flags are zero, you never know how the code is modified in the future, and what other flags are possibly added. So, you should always use something like the following when setting a new flag:

```
flags = fcntl(fd, F_GETFL);
if (fcntl(fd, F_SETFL, flags | O_APPEND) == -1)
  ...
```

  …similarly when removing a flag.

Get file status information: `stat()`

```
int stat(const char *path, struct stat *buf);
int fstat(int fildes, struct stat *buf);
```

- for a file specified by a path or a file descriptor, returns a structure containing file information, such as:
  - `st_ino` ... i-node number
  - `st_dev` ... ID of device containing file
  - `st_uid`, `st_gid` ... user/group ID of owner
  - `st_mode` ... file type and mode
  - `st_size`, `st_blksize`, `st_blocks` ... total size in bytes, preferred block size for filesystem I/O, and number of 512 byte blocks allocated
  - `st_atime`, `st_mtime`, `st_ctime` ... time of last access, last modification, and last i-node modification
  - `st_nlink` ... number of hard links

- You may be able to instruct the system not to update the file access time when reading from a file, with granularity per file system. This option is useful on file systems where there are large numbers of files and performance is more critical than updating the file access time (which is rarely ever important). It may also come in handy if your filesystem is on a medium with a limited number of erase/write cycles – a CompactFlash, for example. If your system supports it, it will be a `noatime` option for the `mount` command. See the manual page.

- *Metadata* is information about the file – access mode, access times, length, owner, group, etc. Metadata does **not** include the file data itself, or the file name as the file data can be accessed through several different hard links and those hardlinks are in the data of directories. In other words, metadata is data about the actual file data.

- Metadata can be read even when the process has no rights to read the file data.

- These functions do not provide file descriptor flags or flags from the system file table. These functions are about file information as stored on some mountable media.

- `st_ctime` is not the creation time but the change time – the last modification of the inode.

- The UNIX norm does not specify the ordering of the `struct stat` members, nor does it prohibit adding new ones.

- Example: stat/stat.c
- You can call `fstat` on file descriptors 0, 1, and 2 as well. Unless redirected before, you will get information on the underlying terminal device (e.g. `/dev/ttys011` on macOS). Example: stat/stat012.c.

Get file status information (2)

- for a file type, in `<sys/stat.h>` there are constants `S_IFMT` (bit mask for the file type bit field), `S_IFBLK` (block device), `S_IFCHR` (character device), `S_IFIFO` (FIFO), `S_IFREG` (regular), `S_IFDIR` (directory), and `S_IFLNK` (symlink).
- macros for file type checking: `S_ISBLK(m)`, `S_ISCHR(m)`, `S_ISFIFO(m)`, `S_ISREG(m)`, `S_ISDIR(m)`, and `S_ISLNK(m)`.
- access right masks: `S_IRUSR` (owner has read permission), `S_IWGRP` (group has write permission), etc.

```
int lstat(const char *path, struct stat *buf);
```

- if *path* is a symlink, `stat()` returns information about the file the symlink refers to. `lstat()` function returns information about the link itself.

- The file type and access mode are stored together in the `st_mode` member and that is why the aforementioned macros exist.
- `S_IFMT` specifies those bits that store the file type, and the macros are not bit masks but values. So, the test for the specific file type is needed to be done like the following (extra parentheses are needed due to the operator precedence): `((st_mode & S_IFMT) == S_IFREG)`. All macros are defined in the spec and thus portable.
- Example: stat/filetype.c

Setting file times

```
int utime(const char *path, const struct utimbuf
*times);
```

- changes file last access and modification times
- cannot change i-node access time (`ctime`)
- calling process must have write permission for the file

- This call is mostly used by copy/move and also archive utilities to make sure the times are the same for the originals and copies. (e.g. `tar` or `rsync`).
- The shell interface for `utime` is the command `touch`. As mentioned in the slide, you cannot change the time of the i-node modification.

File name manipulations

`int link(const char *path1, const char *path2);`

- creates a new link (aka hard link), i.e. a directory entry, named *path2* to file *path1*. Hard links cannot span filesystems (use **symlink** for that).

`int unlink(const char *path);`

- deletes a name (i.e. a directory entry) and after deleting the last link to the file and after closing the file by all processes, deletes the file data.

`int rename(const char *old, const char *new);`

- changes the file name (i.e. one specific link) from *old* to *new*. Works within the same filesystem only.

- It is better said again – **Unix does not have a delete call for files**. Details are on page 88.
- The call `link` creates hardlinks, i.e. a relation between a file name and an i-node number. I-node numbers are unique within a filesystem so for links between filesystems, symlinks are needed. A number of hardlinks to a specific file is only limited by the size of the `st_nlink` member of `struct stat`, and the specification does not specify the size, it only says its type `nlink_t` is an integer value.
- The parameter *path2* must not exist. So, you cannot rename using the `link` call.
- `unlink` does not work on directories.
- The shell command `mv` uses `rename` to move objects within the same filesystem. To move files between filesystems, a file needs to be copied first, then `unlink`ed from the originating filesystem (the whole operation is not atomic).
- `rename` renames symlinks, not the files those symlinks point to.
- There is also a more generic call `remove`, see page 80.

Symbolic links

`int symlink(const char *path1, const char *path2);`

- make a new symbolic name from *path2* → *path1*.
- *path1* may span filesystems, and may not exist at all

`int readlink(const char *path, char *buf, size_t bufsz);`

- put maximum of *bufsz* bytes from the symlink target path to *buf*
- returns the number of bytes written to *buf*.
- *buf* is **not** terminated by `'\0'`

- The shell command `ln` uses either the `symlink` or `link` syscall depending on whether the `-s` option was used.
- Calling `unlink` on a hardlink will not release the file data if other hardlinks exists. You can delete the symlink's target in which case you end up with a *broken link*.
- `readlink` is useful in the situation where you want to `unlink` the symlink's target.
- *bufsize* is typically set as 1 byte less than the buffer size to accommodate the terminating `NULL` character.
- See `readlink/readlink.c` and fix the problem therein.

Working with directories

```
int mkdir(const char *path, mode_t mode);
```

- attempts to create an empty directory *path* with entries '`.`' and '`..`'

```
int rmdir(const char *path);
```

- deletes directory *path*. The directory **must** be empty.

```
DIR *opendir(const char *dirname);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
```

- sequentially reads directory entries
- structure `dirent` contains:
  - `d_ino` ... i-node number
  - `d_name` ... file name

- Directory items are not ordered and `readdir` is allowed to return them in arbitrary order depending on the filesystem implementation. `NULL` is returned on failure and `errno` is set. `NULL` is also returned upon reaching the end of a directory but in that case, `errno` is not changed.
- `readdir` is a stateful function. To read the directory from the beginning again, `rewinddir` can be used. If you want to read the directory from multiple threads, use the reentrant version, `readdir_r`.
- Some older systems might support reading a directory via a normal `read` call. In that case, of course, you need to know the internal representation of the directory data so the use of this is questionable. Linux does not allow it at all.
- `d_ino` might not be very useful if the directory is a mountpoint; this member contains the i-node number of the directory where the filesystem is mounted and not the root of the mounted filesystem as one might expect.
- Some system have a `struct dirent` member `d_type`. It can have values of `DT_REG`, `DT_DIR`, `DT_FIFO` etc., see the man page for `dirent`. It was a BSD specific thing and subsequently copied by other systems, including Linux. However, it is not part of SUSv4 and thus not portable, unfortunately. The portable way is to call `stat` on each directory entry.
- `rmdir` does not work on directories that are not empty. You have to remove all entries before you can delete a directory. You can use `system("rm -r xxx")` etc. but be careful to sanitize the environment and the directory name to avoid any security implications.

- The specification also defines `remove` which behaves like `unlink` for files and as `rmdir` for directories.

Example: read a directory

```c
int
main(int argc, char *argv[])
{
    DIR *d;
    struct dirent *de;

    for (int i = 1; i < argc; i++) {
        if ((d = opendir(argv[i])) == NULL) {
                warn("%s", argv[i]); continue;
        }
        while ((de = readdir(d)) != NULL)
            printf("%s\n", de->d_name);
        closedir(d);
    }
}
```

- The `ls` command is based on a similar loop. It is missing code to tell whether NULL means an error or an end of a directory.
- Example (with the correct use of `errno`): `readdir/readdir.c`

Change working directory

- every process has its current working directory. When a process refers to a file using a relative path, the reference is interpreted relative to the current working directory of the process.
- the working directory is inherited from the process parent

```
int chdir(const char *path);
int fchdir(int fildes);
```

- changes the working directory for the process

```
char *getcwd(char *buf, size_t size);
```

- stores the absolute path to the current working directory to *buf*, its *size* must be at least one byte longer than the path length.

- The descriptor for `fchdir` is from `open` called on the directory (i.e. not from `opendir`).
- There is also a function `chroot` which changes the root directory of a calling process to a new one. It is often used in various server implementations to limit access to the specific subtree. For example, for an FTP server. You have to be careful though and make sure it is not possible to escape from the chrooted subtree and access other parts of the filesystem. The function is also not a part of SUSv4.
  - Other, more generic ways of isolation are provided in available systems, and differ from each other greatly. For example, there are jails in FreeBSD, zones in Solaris, sandboxing in macOS, LXC in Linux (that is what Docker is based on there), etc.

Check permissions for access: access()

```
int access(const char *path, int amode);
```

- checks whether the calling process can access the file *path*
- *amode* is an OR-combination of constants
  - R_OK ... read permission test
  - W_OK ... write permission test
  - X_OK ... execution permission test
  - F_OK ... file existence test
- in contrast to stat(), the result depends on the process's RUID and RGID
- **never use this call, it cannot be used in a safe way. See the notes below.**

- This call applies mechanism of access right testing to a specific file for the calling process.
- The function was meant for an SUID process to verify whether the user running the process would have had access to a file if it was not for the SUID privileges. However, there is an inherent security hole in this approach. The test and the subsequent action on the file is not an atomic operation. An attacker could possibly unlink the file and immediately symlink it to a different file to what it actually had no rights to manipulate with. If the timing is right, the SUID process will operate on that other file. The correct solution is not to use the access call but return to the real UID/GID and try the operation. For example, if we succeed in opening the file under the real UID/GID and continue working with the file descriptor, the file manipulation mentioned above would not gain the attacker anything.

Setting file permissions

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

- changes permissions of file *path* to *mode*.
- can be used by the file owner or root

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

- changes file owner and group for *path*. Value of `-1` means do not change that ID.
- only root can change owners so that users could not work around quotas to disown their files
- a regular user can change a group of files he/she owns, and must belong to the target group

- The *mode* parameter does not contain the file type, of course. See `sys/stat.h` or the `chmod(2)` manual page for available bit masks.
- Only the owner of a file can change its mode. Mode like `rw-rw-rw-` has nothing to do with that.
- In some implementations, it was possible to pass the file ownership to someone else. Today this is usually not allowed. Just think about how you could work around a per-user filesystem quota with this.

Contents

- Introduction, Unix and C, programming tools
- Basic Unix concepts and conventions, its API
- **Access rights, devices**
- Process manipulation, program execution
- Signals
- Process synchronization and interprocess communication
- Network programming
- Programming with threads
- Appendix

Users and groups

```
beran:x:1205:106:Martin Beran:/home/beran:/bin/bash
```

The fields, in order from left to right: user name, hashed password (today in `/etc/shadow` or elsewhere), user ID (aka UID); primary group ID (aka GID), full name, home directory, login shell

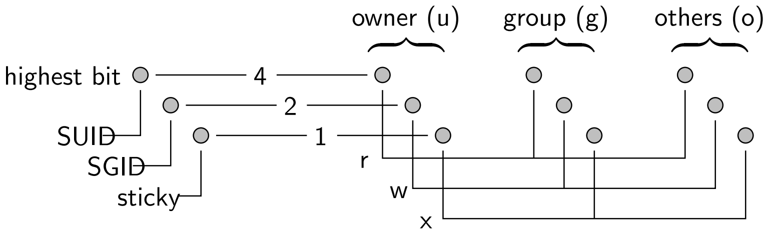
Note that a superuser (root) has always UID 0.

```
sisal:*:106:forst,beran
```

The fields, in order from left to right: group name, group password (not used today), group ID (GID), list of group members

- User information in files `/etc/passwd` and `/etc/group` are processed by various system programs, like `login` or `su`. The kernel knows nothing about these files as it only uses numeric representation of users and groups.
- Passwords are long gone from `/etc/passwd`, they are stored some place else, for example in `/etc/shadow`, which is not readable to an unprivileged user. The passwords are also salted and then hashed. On BSD based systems, e.g. FreeBSD or macOS, instead of `/etc/shadow`, `/etc/master.passwd` database is used.
- If `/etc/shadow` does exist, it is structured in a similar way as `/etc/passwd`. The file is actually used as a database, storing e.g. number of unsuccessful login attempts for given user etc.
- There are also protocols used for authentication that do not use `/etc/passwd` at all, for example NIS (Network Information Service) or LDAP (Lightweight Directory Access Protocol).
- The user group in `/etc/passwd` is called *primary* for the user. Such a group is used when files are created. Other groups a user belongs to, those in the user's entry in `/etc/group`, are called *supplementary* and provide additional group privileges to access files.

Access rights



- **SGID** on a file without the executable bit for its group means **mandatory locking** in systems based on System V
- **sticky bit** for directories: remove and rename allowed for file owners only.
- **SGID** for directory: new files will have the same group as the directory (on System V based systems; it works in a different way on BSD systems, see below)

- The sticky bit for directories means that when the directory is writable for a given user (possibly because all users can write), the user can create any file that does not exist in that directory yet. However, if the file exists but is not owned by the user, he/she can not remove nor rename it even if he/she can write to the directory. The sticky bit is denoted by "t" and is typically used for temporary directories:

```
$ ls -ld /tmp
drwxrwxrwt 9 root root 356352 Jan 27 22:37 /tmp/
```

- The SGID bit for directories on BSD based systems means that files and directories created in this directory will have the same owner as the directory itself. The filesystem must be mounted with an `suiddir` flag and the kernel may need an additional non-default option `SUIDDIR`. It also does not work for the root user. This functionality is there to support the Samba protocol.
- Originally, the sticky bit had a meaning for regular files as well but that is not used anymore.
- Some filesystems (XFS, AFS, UFS2, ZFS, and others) also support *access control lists* (ACLs) that allow for finer access right management.

Obtain user/group information

- `struct passwd *getpwnam(const char *name)`
  return structure describing user found in password database or NULL.
- `struct passwd *getpwuid(uid_t uid)`
  ditto; perform search according to UID.
- `void setpwent(void)`
- `void endpwent(void)`
- `struct passwd *getpwent(void)`
  these functions traverse password database. **setpwent** rewinds to the beginning of the password database, **getpwent** gets the current entry, **endpwent** closes the password database and free allocated resources.

- These functions work independently on what database was used to get the user information, see page 87 for more information on naming databases.
- All these functions are part of POSIX 1003.1-2008.
- **setpwent**() must be called before the first call to **getpwent**().
- There are also functions **getgrnam**() and **getgrent**() which can be used to get group information.

- To search and list naming databases, you can use the program `getent`. For example:

```
$ getent passwd root
root:x:0:0:Super-User:/root:/sbin/sh
$ getent group root
root::0:
```

Name service switch

- today's systems are not confined to only using `/etc/passwd` and `/etc/groups`
- such systems have *databases* (`passwd, groups, protocols`, ...)
- database data come from *sources* (files, DNS, NIS, LDAP, ...)
- file `nsswitch.conf` defines what databases use what sources
- library functions must support this, obviously
- it is possible to combine some sources, e.g. users may be first be searched in `/etc/passwd`, then in LDAP
- came first with Solaris, other systems borrowed the idea

- Systems using the name service switch typically use the `nsswitch.conf(4)` file to store information about what databases are supported, including the API. For example, with the `passwd` database, standard calls like `getpwnam(3)` and `getpwent(3)` use it. In general, it is not needed to process such databases manually, as there is always an API to use for that.
- Example of an existing `nsswitch.conf` on Solaris:

```
passwd:     files ldap
group:      files ldap

# You must also set up the /etc/resolv.conf file for DNS name
# server lookup.  See resolv.conf(4).
hosts:      files dns

# Note that IPv4 addresses are searched for in all of the
# ipnodes databases before searching the hosts databases.
ipnodes:   files dns
```

```
networks:   files
protocols:  files
rpc:        files
ethers:     files
```

Access rights testing

- a user is identified with a **UID** number and numbers for groups he belongs to (**primary GID**, **supplementary GIDs**)
- this identification is inherited by each process
- file F has owner (UID_F) and group owner (GID_F).
- the algorithm for evaluation of access rights for process: $P(UID_P, GID_P, SUPG)$ and file $F(UID_F, GID_F)$:

If	then Process has w.r.t. File
`if(`UID_P `== 0)`	... all rights
`else if(`UID_P `==` UID_F`)`	... owner rights
`else if(`GID_P `==` GID_F `||` $GID_F \in SUPG$`)`	... group rights
`else`	... rights of others

- The processes of the `root` user can change its user and group identity. This is used e.g. by the `login` process, which runs as `root` and after performing an authentication check successfully, runs a shell process with the identity of the given user (using the `setuid` syscall – see upcoming slides).
- The implication of the algorithm is that for the `root` user, the access rights are not relevant (it has always unlimited access – at least in classic UNIX without fine grained privileges). If the user is equal, the group/other rights are not used even though they permit more than what user rights do. Similarly the others rights are not used if the group is equal. **Therefore if a file owned by a user has the rights set to `---rwxrwx`, the user cannot read/write/execute it until he/she changes the rights.**
- More and more systems diverge from the classic model where many processes were running under a user with UID 0. A security vulnerability in such an application meant total control of the system. To thwart this, these systems employ models like *least privilege* in Solaris or *privilege separation* and *pledge* in OpenBSD.
- In order to delete a file, the user has to have write permission for the **directory** containing the file, because that is actually the “file” being changed.

The rights of the file to be deleted are not relevant; the shell might give you a warning that you are about to delete a file for which you do not have the right to write, however that is just a warning, the operation will proceed. It is quite logical – if you set a file as read-only the shell will deduce that you probably do not want to delete such a file. See the example below. **Unix systems do not have delete-like operation for a file**, the file is deleted automatically once it is no longer referenced from a directory structure and the file is not presently open by any process (see `unlink/unlink.c`).

```
$ whoami
janp
$ ls -ld janp-dir
drwx------   2 janp  staff  512 Mar 23 12:12 janp-dir/
$ ls -l janp-dir
total 0
-rw-r--r--   1 root  root     0 Mar 23 12:11 root_wuz_here.txt
$ rm janp-dir/root_wuz_here.txt
rm: janp-dir/root_wuz_here.txt: override protection 644 (yes/no)? yes
$ ls janp-dir/root_wuz_here.txt
janp-dir/root_wuz_here.txt: No such file or directory
```

- However if `root` creates its own sub-directory in the `janp-dir` directory and then creates a new file in there, the `janp` user can no longer delete the `janp-dir` directory and its contents because:
  - no directory can be deleted if non-empty, and
  - the given file created by `root` cannot be deleted because `janp` is not an owner of the sub-directory containing the file
- If the read bit is removed from a directory rights, it is not possible to read its contents, therefore you cannot list files contained therein. However if you know the name of the file in the directory and the execute bit is set, you can read the file:

```
$ mkdir foo
$ ls -ald foo
drwxr-xr-x  2 vladimirkotal  staff  68 Nov  5 14:37 foo
$ touch foo/bar
$ file foo/bar
foo/bar: empty
$ ls foo
bar
$ chmod u-r foo
$ ls foo
ls: foo: Permission denied
$ file foo/bar
foo/bar: empty
```

- There is a situation where even the execute bit for a directory is not sufficient. That is used for temporary directories where anyone can write to. However, it is not desirable to permit users to delete each others files. To achieve that there is the *sticky bit* (01000). There might be a `sticky` man page, where the sticky bit function is described. It is visible as `t` in the `ls` output:

```
$ ls -ld /tmp
drwxrwxrwt   7 root     root         515 Mar 23 12:22 /tmp
```

Real and effective UID/GID

- for each process the following IDs are distinguished:
  - **real UID** (RUID) – real owner of the process
  - **effective UID** (EUID) – user, whose rights are used by the process
  - **saved UID** – original effective UID
- similarly each process has the real, effective and saved GID.
- usually `RUID==EUID && RGID==EGID`.
- **right vesting** . . . execution of a program with the SUID (**set user ID**) bit set changes the EUID and the saved UID of the process to the UID of the program owner, the RUID stays the same.
- similarly the SGID bit changes the EGID of the process.

- **access rights checking always consults the EUID, EGID, and supplementary GIDs**
- The SUID and SGID bits are used for programs that need bigger privileges than the user who executes them. One example is the `passwd` program that needs to update files `/etc/passwd` and `/etc/shadow`, where the ordinary (i.e. non root) user process cannot modify the first and cannot write into the the second. Another example is the `su` program, which has to have the right to arbitrarily change user and group identity, which is a privilege of programs running with UID 0.
- Programs using the SUID and SGID bits should be carefully programmed to allow only the operations for which they were designed and prevent misuse of their privileges for non-authorized actions (root shell execution). Such programs used to be one of the most frequent causes of security problems in Unix systems.
- The basic rule for writing SUID/SGID programs is: **do not write them** if it is not absolutely necessary. It is not easy to produce a correct (i.e. secure) SUID/SGID program, especially of higher complexity.
- **These are the rules for ID change:**
  - an ordinary user process cannot change its RUID or saved UID (the `exec` is an exception to that, see page 106)

  - the process can always change its EUID to that of the RUID or saved UID. This guarantees that in a SUID program, it is possible to arbitrarily change the EUID between the one that enabled the process to gain ownership rights and the UID of the real user that executed the process originally.
  - **root can do everything**, and when it changes the RUID, it will also change the saved UID – it does not make sense to change just one of them when either can be used to set the EUID.

Process owner identification

- `uid_t` **getuid**`(void)`

  returns the real user ID of the calling process.
- `uid_t` **geteuid**`(void)`

  returns the effective user ID of the calling process.
- `gid_t` **getgid**`(void)`

  returns the real group ID of the calling process.
- `gid_t` **getegid**`(void)`

  returns the effective group ID of the calling process.
- `int` **getgroups**`(int` *`gidsz`*`, gid_t` *`glist`*`[])`

  – `glist` returns at most `gidsz` supplementary group IDs of the calling process and returns number of all GIDs of the process.

- The `getuid` returns real the UID, there is nothing like `getruid`.
- `getgroups`: when `gidsz == 0`, it returns the number of groups. When `0 < gidsz < #groups`, it returns `-1`.
- In Unix, there are many data types such as `uid_t`, `gid_t`, `size_t`, `pid_t`, etc. In general, these are integer types and you can often find them in the `/usr/include/sys/types.h` header file.
- Solaris has the `pcred` command that provides process identification information in a simple form:

```
$ pcred 5464
5464:   e/r/suid=1993  e/r/sgid=110
        groups: 33541 41331 110
```

Process owner change

- `int setuid(uid_t uid);`
  - for a process with EUID == 0, sets the RUID, EUID and saved-SUID to `uid`
  - for other processes it sets just the EUID, and `uid` must be either equal to the RUID or saved UID
- `int setgid(gid_t gid);`

  similar to `setuid`, for group-IDs of the process.
- `int setgroups(int ngroups, gid_t *gidset)`

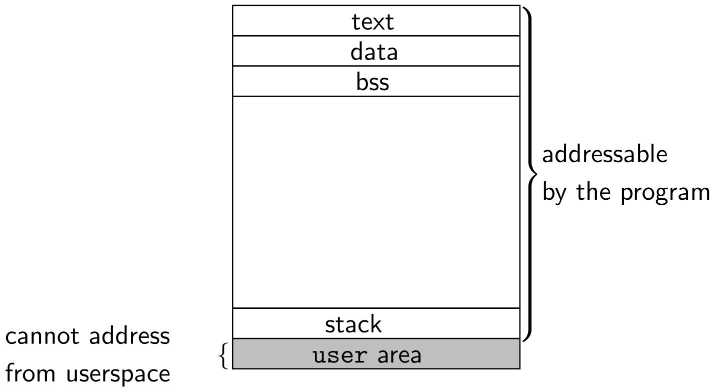
  sets the supplementary group IDs for the calling process. Can only be used by superuser process.

- W.r.t. setting the UID for a process with EUID == 0, see also the notes on page 90.
- To recap the above: a process with effective rights of a superuser can arbitrarily change its identity. The rest can only switch between its real and effective IDs.
- The *login* program uses the `setuid` syscall.
- If a process with UID == 0 wants to change its identity, it has to call `setgid` first and then `setgroups`. Only after that can it call `setuid`. Any other ordering would mean that the process would lack the rights to perform the operation in question, e.g. once `setuid` returns it would not have the rights to perform `setgid` and `setgroups`.
- `setgroups` is not part of UNIX 98 or UNIX 03.
- RUID/EUID are saved in the kernel process structure and also in the so called *u-area* (see e.g. [Bach]).
- If a root SUID program calls `setuid` for a UID other than 0, it can no longer return the EUID to 0 (this makes sense, imagine a user logging into the system). For a different behavior, `seteuid` (that sets just the EUID) would have to be used.
- Example: `setuid/screate-file.c`

Contents

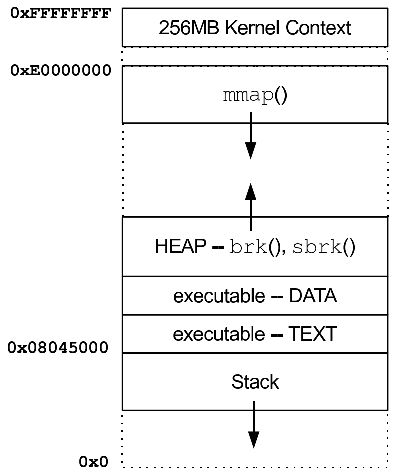
- Introduction, Unix and C, programming tools
- Basic Unix concepts and conventions, its API
- Access rights, devices
- **Process manipulation, program execution**
- Signals
- Process synchronization and interprocess communication
- Network programming
- Programming with threads
- Appendix

Process memory layout in userspace



- Each process has 3 basic segments (memory segments, not hardware segments):
  - text ... program code
  - data ... initialized variables
  - stack
- `text` and `data` sections are saved in executable file
- The sections for initialized and non-initialized variables and heap are considered as data
- It is also possible to connect segments of shared memory (`shmat`) or files (`mmap`) into the address space.
- The text is shared between all processes which execute the same code. The data segment and stack are private for each process.
- Each system can use a different layout of a process address space (and typically it is indeed so). See the next slide which also shows sections for `mmap` and *heap*.
- *bss* ... non-initialized variables (`bss` comes from the IBM 7090 assembler and stands for „block started by symbol"). While the program is running, the `data`, `bss` and heap sections (not shown in the picture) make up data segments of the process. Heap size can be changed using the `brk` and `sbrk` system calls.
- Note – by non-initialized variables are meant static variables – i.e. global variables or variables declared as `static` both in the functions and outside that are not set to a value. All these variables are automatically initialized with zeroes before the program is started. Therefore it is not necessary to store their value in the binary. Once one of these variables is initialized, it will become part of the data segment on disk.
- *(User) stack* ... local non-static variables, function parameters (on certain architectures in certain modes - e.g. 32-bit x86), return addresses. Each process has 2 stacks – one for a user mode and another for kernel mode. The user stack automatically grows according to its use (except for threads where each thread has its own limited stack).
- *User area (u-area)* ... contains process information used by the kernel which is not needed when the process is swapped out to disk (number of open files, signal handling settings, number of shared memory segments, program arguments, environment variables, current working directory, etc.). This area is accessible only to the kernel which will see just the area of a currently running process. The rest of the data needed even if the process is not currently running or while swapped out to disk is stored in the `proc` structure. `proc` structures for all processes are always resident in memory and accessible in a kernel mode.

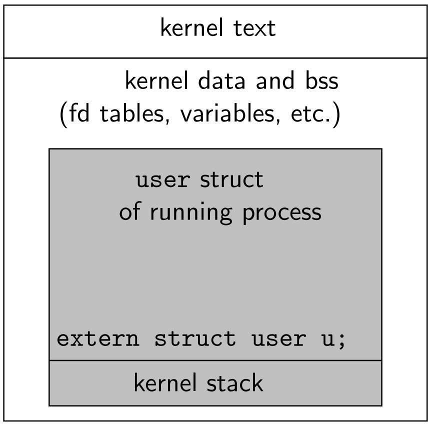
Example: Solaris 11 x86 32-bit



- The following is deductible from the image:
  - maximum size of kernel for Solaris 11 x86 32-bit is 256 megabytes
  - there is free space between kernel and memory reserved for `mmap`
  - stack grows towards lower addresses and its size is limited to 128 megabytes
- A *heap* is a part of the memory that can be extended by processes using the `brk` and `sbrk` syscalls and is used by the `malloc` function. The `malloc` allocator gradually extends the heap on demand, and manages acquired memory and distributes it to the process in chunks. When `free` is called, it does not mean that the memory is returned to the kernel; it is only returned to the allocator.
- The `mmap` area is used for mapping files into memory, i.e. also for shared libraries. Some allocators use also this memory internally, e.g. in case a process requests larger chunks of memory at once. It is possible to exclusively use just `mmap`, and that is transparent to the application. When using `mmap` it is possible to return the memory to the kernel (using `munmap`), in contrast to the `brk`/`sbrk` based implementation.
- The picture was taken from [McDougall-Mauro] and does not contain space for non-initialized variables. If you try to print the address of such a variable on this system, you will find out that both initialized and non-initialized variables share a common data segment, labeled in the image as ,,executable – DATA". Example: `pmap/proc-addr-space.c`.

- The kernel mapping is not necessary, for example on Solaris running on `amd64` architecture (i.e. 64-bit) the kernel is no longer mapped into user space.
- `brk` nor `sbrk` are part of the standard, hence portable applications should not use them; if a similar functionality is needed, they should use `mmap`, see page 116.

Process memory layout in kernel



kernel text

kernel data and bss
(fd tables, variables, etc.)

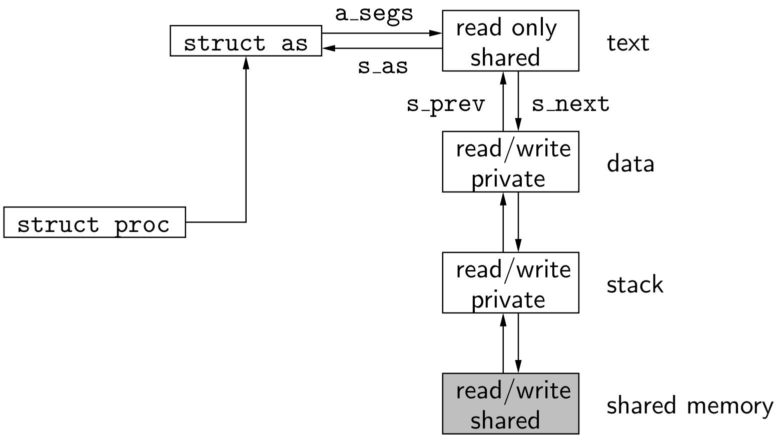
user struct
of running process

`extern struct user u;`

kernel stack

- A process will enter a kernel model either by a *trap induced by the CPU* (page fault, unknown instruction, etc.) *timer* (to invoke scheduler), *interrupt* from a peripheral device, or synchronous trap (a standard library uses it to hand over the control to the kernel to service a *system call*).
- There is only one copy of the kernel text and data in the memory, shared by all processes. The kernel text as a whole is resident in memory and not swapped out to disk.
- *kernel text* ... code of the operating system kernel, loaded when the system is booting up and is always resident in memory. Some implementations allow to add modules to the kernel during runtime (e.g. when a device is connected, matching device driver module is automatically loaded), it is therefore not necessary to regenerate the kernel and reboot the system whenever a change is needed.
- *data and kernel `bss`* ... contain data structures used by the kernel, contains the u-area of a currently running process.
- *kernel stack* ... independent for each process, is empty when the process is in the user mode (and therefore uses the user stack).

Process memory segments



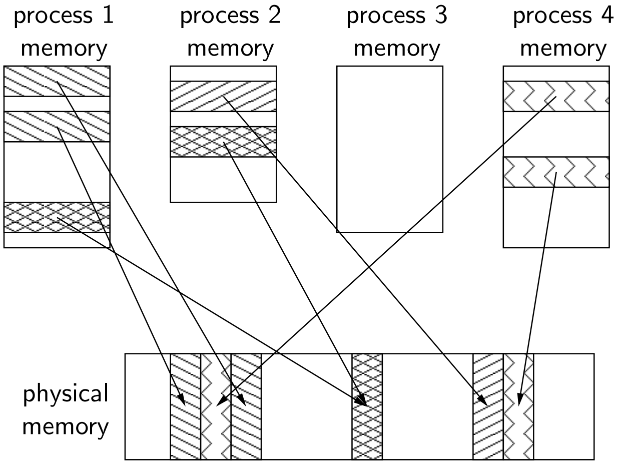
- This is how memory segments representation looks like in the kernel.

- The core feature of this architecture is a *memory object*, a mapping abstraction between a part of memory and a place where data is normally stored (so called *backing store* or *data object*). This place can be e.g. swap space or a file. Address space of a process is a set of mappings to different data objects. There exists also an *anonymous object* that does not have persistent backing store (it is used e.g. for a stack). Physical memory then serves as a cache for data of these mapped objects.

- This coarsely described architecture is called VM (*Virtual Memory*), and was introduced in SunOS 4.0. The virtual memory architecture of SVR4 is based on this architecture. More information can be found in [Vahalia], the original white paper from 1987 that introduced this architecture: Gingell, R. A., Moran J. P., Shannon, W. A. – *Virtual Memory Architecture in SunOS.*

- To determine what memory segments a memory space of a process consists of, various tools can be used: `pmap(1)` on Solaris, NetBSD and in some Linux distributions, `procmap(1)` on OpenBSD, or `vmmap(1)` on macOS.

Virtual memory



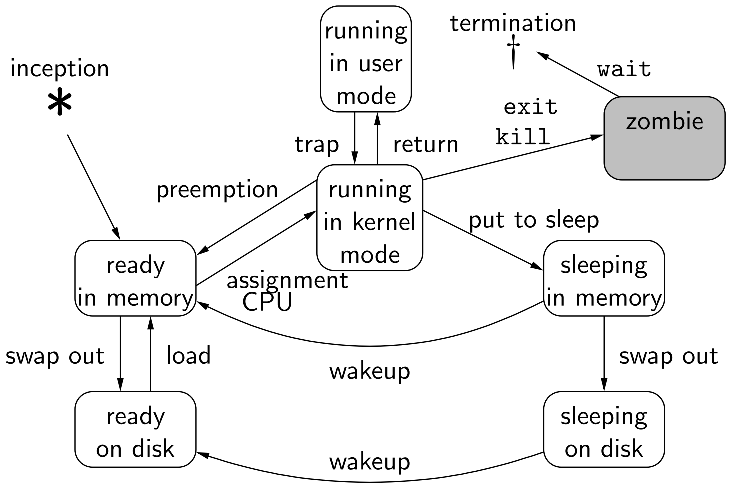
- Each process sees its own address space as a contiguous interval of (virtual) addresses from zero to some maximal value. Accessible addresses are those for which there is a mapping, i.e. there is a memory segment (see the previous slide).

- The kernel divides the memory to pages. Each page has its own location in physical memory. This location is determined by kernel page tables and the memory pages can be arbitrarily mixed w.r.t. their placement in the virtual address space of a process.

- If a page is not used it can be swapped out to disk.

- The kernel memory management ensures a mapping between virtual addresses used by processes and the kernel to physical addresses. It also reads in pages from a disk upon a page fault.

Process states



- After the process is terminated either using an `exit` call or as a consequence of a signal, it will transition to a zombie state as the kernel needs to store a return value for the process. The whole memory of the process is freed, the only remaining piece is the `proc` structure. The process can go away for good only after its parent will retrieve its return value using the `wait` call. If the original parent is no longer available, the `init` process which became the new parent will call `wait`.
- In today's Unix systems processes are usually not swapped out as whole, only individual pages are.
- A process is put to sleep if it requests it, e.g. when it is waiting on a completion of an operation with a peripheral device. The *preemption* is on the other hand an involuntary removal of a CPU by the scheduler.

Process scheduling

- *preemptive* – if a process does not give up CPU (e.g. by entering a sleep to wait on some event), the CPU is taken away after a time quantum expiration.
- processes are classified into queues according to a priority, a CPU is assigned to the first ready process from the queue with the biggest priority.
- SVR4 introduced priority queues and real-time support with guaranteed maximal response time
- contrary to the previous versions, in SVR4 a bigger number means a bigger priority

- **The premise of preemptive planning are periodic timer interrupts** which take away the CPU from the running process and pass on the CPU to the kernel (scheduler is activated).
- The other variant is non-preemptive (cooperative) planning, where process keeps running, until it gives up the CPU, i.e. until it calls such system call, that switches the context to different process. The downside of cooperative planning is that one process can block the CPU and other processes forever.
- Unix uses only preemptive planning for user processes.
- Traditional (historical) UNIX **kernel** uses cooperative planning, i.e. process running in kernel mode is not switched until it gives up the CPU by itself. **Modern Unix kernels are preemptive** – mainly because of real-time systems; where it is necessary to have the possibility to remove a CPU from a running process immediately, and not waiting until it returns from a kernel mode or enters sleep by itself. Note that UNIX was preemptive from its very beginning but its kernel was non-preemptive in the beginning.
- With a preemptive planning processes can be interrupted at any time and the CPU given to another process. Therefore a process can never be sure that a given operation (spanning more than one instruction, besides system calls with guaranteed atomicity) will be executed atomically, without being influenced by other processes. If it is necessary to ensure atomicity of an operation, processes must synchronize. This problem is avoided in a cooperative planning – the atomicity of a given operation is simply ensured by not giving up the CPU while the operation is still in progress.

Priority classes

- **system**
  - priority 60 to 99
  - reserved for system processes (`pageout`, `sched`, ...)
  - fixed priority
- **real-time**
  - priority 100 to 159
  - fixed priority
  - a time quantum corresponds to priority value
- **time-shared**
  - priority 0 to 59
  - dynamic 2 part priority, fixed user part and dynamic system part – if a process uses CPU extensively, its priority is being decreased (and time quantum increased)

- The system class is used only by the kernel, a user process running in a kernel mode retains its own planning characteristics.
- Processes in the real time class have the biggest priority and so should be configured correctly otherwise they could block the rest of the system from getting any CPU time.
- If a process in a time-shared class is put to sleep and is waiting on an event, the system priority is temporarily assigned to it. After a wake-up, such a process will be assigned a CPU earlier than other non-sleeping processes to finish the operation as soon as possible as it could hold some locks later needed by other processes.
- Fixed part of the priority in the time-shared class can be set using

  ```
  int setpriority(int which, id_t who, int prio);
  ```
  or
  ```
  int nice(int incr);
  ```

  The *which* value determines what will be in the *who* argument. If *which* is e.g. *PRIO_PGRP*, the *who* will store process group number. Note that **nice** call will return a new nice value. As -1 is a valid value, it is necessary to clear `errno` and then check if the function returns -1.

- The priority class and nice value of a given process can be displayed with the `-l` option of the `ps` command or by explicitly specifying the fields to be printed out (see the `-o` option).

- Example: priority values have different scales on different systems. E.g. on macOS 10.9 a process that had the priority value 30 will have the value decremented to 21 after increasing the nice value:

```
$ sleep 200 &
[1] 36877
$ ps -O pri,nice -p $!
  PID PRI NI   TT  STAT      TIME COMMAND
36877  31  0 s003  S      0:00.00 sleep 200
$ renice 10 -p $!
$ ps -O pri,nice -p $!
  PID PRI NI   TT  STAT      TIME COMMAND
36877  21 10 s003  SN     0:00.00 sleep 200
```

  With Linux kernel 3.10 it will look differently – the priority value will be increased after a nice value increases. However, that means the process will be running with a smaller priority.

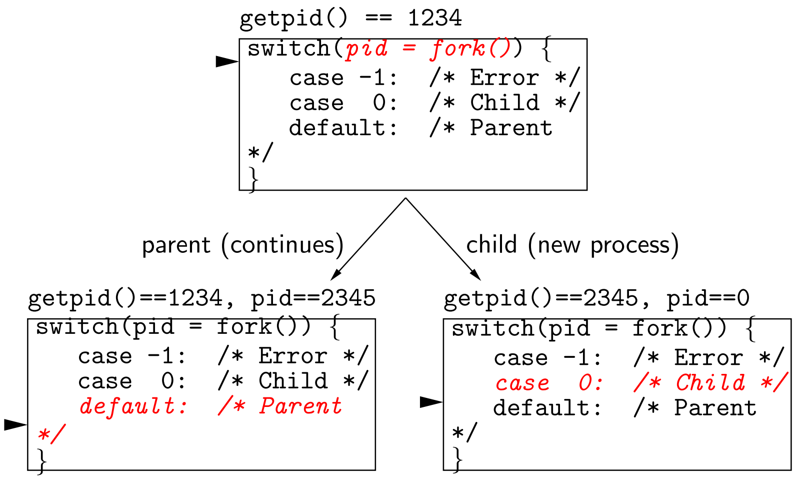
Process groups, controlling terminals

- every process belongs to a *process group*
- each group can have a leading process, so called *group leader*
- every process can have a *controlling terminal* (usually it is a login terminal)
- special file `/dev/tty` is associated with a controlling terminal of each process
- each terminal is associated with a process group called a *controlling group*
- *job control* is a mechanism for suspending, resuming, and terminating process groups and control their access to terminals
- *session* is a collection of process groups created for the purpose of job control

- When a user logs into a system, a new session is created. The session contains one process group that only has a single process – one with the user's shell. That process is also the leader of that single process group and also a session leader. In case a job control is on, each command or a pipeline will create a new process group, and one process from each group will always become a process group leader. One of the groups can be running in the foreground, the rest will be running in the background. Signals which are generated from the keyboard (i.e. those triggered by combination of keys, not by executing the `kill` command) are sent only to the group running in the foreground.

- If the job control is off, command execution in the background means that the shell will not be waiting for its completion. There exists only one group of processes, and keyboard generated signals are sent to all processes running in the foreground and background. Processes cannot be moved to the background from the foreground and vice versa.
- When a process that has a controlling terminal opens the `/dev/tty` file, it gets associated with its controlling terminal, i.e. if two different processes open this file, each will be accessing a different terminal.
- In bash a process group (job) can be stopped temporarily using `Ctrl-Z`, and can be resumed again using ,,`fg %N`" where `N` is the number from the `jobs` command listing. More information can be found in the "JOB CONTROL" section in the bash man page.

Create a new process: `fork()`



```
getpid() == 1234
switch(pid = fork()) {
   case -1:  /* Error */
   case  0:  /* Child */
   default:  /* Parent
*/
}
```

```
getpid()==1234, pid==2345
switch(pid = fork()) {
   case -1:  /* Error */
   case  0:  /* Child */
   default:  /* Parent
*/
}
```

```
getpid()==2345, pid==0
switch(pid = fork()) {
   case -1:  /* Error */
   case  0:  /* Child */
   default:  /* Parent
*/
}
```

- The child process is almost an exact copy of its parent except for the following:
  - The child process has a unique process and parent process ID.
  - If the parent had multiple threads, the child will only have one that called `fork`; will be further explained on page 193.
  - Child process resource utilization counters are set to 0.

  - `alarm` settings and file locks are not inherited.

- The file descriptor tables are exact copies in both processes. That means that more processes can share and seek a common file position. Signal masks are not changed, more on that on page 134.
- For efficiency and less memory consumption, the address space is not copied but a *copy-on-write* mechanism is used.
- The reason why the parent gets its child's PID as a return value and the child gets 0 is because it is easy for the child to get its parent PID via `getpid`. Imagine how the parent would figure out the new child PID, especially if it already spawned multiple children.
- Example: `fork/fork.c`
- There is also `vfork`, used in the past to work around the problem that the child address space was usually rewritten on subsequent `exec`. This problem was solved via already mentioned copy-on-write mechanism. See `fork/vfork.c` on how it works.
- The problems of `vfork` are largely solved by `posix_spawn`. See page 112.

Process identification

`pid_t` **getpid**`(void);`

- returns the process ID of the calling process.

`pid_t` **getpgrp**`(void);`

- returns the PGID of the calling process

`pid_t` **getppid**`(void);`

- returns the process ID of the parent process.

`pid_t` **getsid**`(pid_t` *`pid`*`);`

- returns the session ID for process `pid` (0 means for the calling process)

process groups make it possible to send signals to group of processes at once

session is a collection of processes created for (*job control*).

The processes of the session share one *controlling terminal*. Session includes one or more process groups. Maximum one group in the session runs in foreground (*foreground process group*) and has access to the controlling terminal for input and output, the rest is running in the background (*background process groups*) and have only optional access to the output or no at all. (disallowed operation with terminal will stop the process).

parent process: Each process (besides `swapper`, `pid == 0`) has a parent, i.e. process that created it with the `fork` syscall. If the parent exits before the child, its adoptive parent will become the `init` process, that will take care of the zombie after the process ends.

- To get information about running processes programmatically is possible using non-standard API (e.g. the `libproc` library on Solaris built on top of the `procfs` filesystem that is mounted under `/proc`).
- Note that using `getppid` value to check if parent exited is not portable (commonly `init` has `pid` 1 however this is not true in many virtualized/containerized environments, e.g. in PID namespaces on Linux, Zones in Solaris).
  Example: `session/getppid.c`.

Creating a new process group/session

`int` **setpgid**`(pid_t` *`pid`*`, pid_t` *`pgid`*`);`

- sets the PGID of the process specified by `pid` to `pgid`.

`pid_t` **setsid**`(void);`

- creates a new session if the calling process is not a process group leader

- For the **setpgid** syscall the following is true:
  1. pid == pgid : the process with *`pid`* will become process group leader
  2. pid != pgid : the process with *`pid`* will become process group member

- The process which is not yet process group leader can both become session leader and process group leader using `setsid`. If the process already is process group leader, `setsid` will fail. To overcome this it is necessary to call `fork` and call `setsid` in the child process. Such process does not have controlling terminal however it can acquire it by opening a terminal which is not yet controlling terminal of a session when `open` flags argument does not contain the `O_NOCTTY` flag, or using other implementation dependent way.

Execute a program: `exec`

```
extern char **environ;
int execl(const char *path, const char *arg0, ... );
```

- replaces the current process image with a new process image
- runs a program defined via *path*
- arguments that follow, including *arg0*, are given to the program via `argc` and `argv` of its `main()`
- the argument list must end with `(char *)0`, i.e. `NULL`
- *arg0* should contain the program name (i.e. not the full path)
- **open file descriptors are unaffected by exec**
  - . . . aside from file descriptors with flag `FD_CLOEXEC`

- *path* must be an absolute or relative path to the executable file. `PATH` is only used for **execlp** a **execvp** (see the one of the slides that follow), if *path* does not contain `'/'`.

- All variants of these calls are commonly just called the **exec** call. It goes without saying that one of the variants is used but usually that is not important for the sake of a discussion.

- Sometimes `argv[0]` is different from the executable file name. For example, `login` command prefixes the shell file name with `'-'`, e.g. `-bash`. The shell then knows it is supposed to function as a login shell. A login shell reads `/etc/profile`, for example.

- **exec** does not transfer the control to the program in memory directly. As described on page 30, the system (i.e. the code of the **exec** call) first maps the dynamic linker, aka loader, to the process address space. The loader then maps all dynamic libraries there as well, then finally calls the program `main()`.

- A useful exercise is to write a simple program calling `open()`, for example. When done, run the program via `truss(1)` or `strace(1)` like this: `truss ./a.out`. You will see what is being done before `open` is called in the end.

Execute a program: `exec` (continued)

```
extern char **environ;
int execl(const char *path, const char *arg0, ...  );
```

- successful **execl** never returns as the new process (program) fully replaced the address space of the calling process
  - ...the original place to return to no longer exists
- signal handlers are set to default
  - ...as the original handler code no longer exists
- the new process inherits `environ` from the calling process

- More about signals on page 123.
- **exec** does not change RUID and RGID. And for security reasons, if the executed program has a SUID bit set, the program's EUID and saved EUID are set to the UID of the executable program owner.
- Today's systems can also execute scripts that start with a line:
  `#!/`*`interpreter_path`*`/`*`interpreter_name [args]`*

Variants of the exec call

```
int execv(const char *path, char *const argv[]);
```

- like **execl** but arguments are in the *argv* array, the last item must be `NULL`

```
int execle(const char *path, const char *arg0, ...   ,
           char *const envp[]);
```

- like **execl** but instead of the global variable *environ*, the ***envp*** argument is used

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

- like **execv** but instead of *`environ`*, ***envp*** is used

```
int execlp(const char *file, const char *arg0, ...);
int execvp(const char *file, char *const argv[]);
```

- like **execl** and **execv** but `PATH` is also used for searching for the executable file

- **l** = list (i.e. list of arguments), **v** = vector (i.e. an array of string pointers), **e** = environment (i.e. environment variables are passed to the function via an argument), **p** = `PATH` is used.
- Aside from **execlp** and **execvp**, it is always needed to use the full path to the executable program, either an absolute or relative one.
- All variants aside from **execle** and **execve** are also passing to the program being executed the environment variables of the calling process, i.e. the `environ` array.
- For some unknown historical reasons, there is no “p” with “e” together in the standard. However, GNU provides **execvpe** as an extension.
- Example: `exec/exec-date.c`
- The following use of **execl** is incorrect as it is missing the mandatory argument for `argv[0]`:

  ```
  execl("/bin/ls", NULL);
  ```

  On some systems, the above has very interesting consequences. As `NULL` is taken as an expected `argv[0]`, the data on the stack are then accepted as the program arguments until the next `NULL` is found there. In the following example, run on some version of the FreeBSD system, `ls` is trying to list file names that are environment variable names and values (the environment array contain strings `<varname>=<value>`), as those were on the stack because the environment was passed to the program being executed, as we already know. In my case, the output was:

```
$ ./a.out
: BLOCKSIZE=K: No such file or directory
: FTP_PASSIVE_MODE=YES: No such file or directory
: HISTCONTROL=ignoredups: No such file or directory
: HISTSIZE=10000: No such file or directory
...
...
```

Executable file format

- **a.out** format, in early UNIX versions
- **Common Object File Format (COFF)** – AT&T System V, superseded **a.out**
- **Extensible Linking Format (ELF)** – new in SVR4, replaced both older formats
- ELF format:

ELF header
program header table
section 1
⋮
section N
section header table

- The UNIX standard does not specify what executable file format systems should use. While most of the UNIX and Unix-like systems (e.g. Linux distributions) use ELF, there are other widely used systems that do not. One example is macOS (which is a certified UNIX system) that uses the *Mach-O* file format, short for *Mach Object*. Each Mach-O file is made up of one Mach-O header, followed by a series of load commands, followed by one or more segments, each of which contains between 0 and 255 sections.
- On Solaris, the `elfdump` command allows listing sections of the ELF file in a human readable form. On Linux distributions, use `readelf`.
- The *ELF header* contains basic information about the file. Try "`readelf -h /bin/ls`" on any Linux distribution.
- The *program header table* is only present in files that are executable. For example, dynamic libraries are ELF files that are not executable. It con-

tains information on the virtual memory layout. You can list the table via "`elfdump -p`" or "`readelf -l`".

- Sections contain code, data, symbol table, relocation data, etc.
- The *section header table* contains information for the linker, see "`elfdump -c`" or "`readelf -S`".
- Some systems stuck on a format based on the original *a.out* for a long time. For example, OpenBSD moved from *a.out* to ELF in 2003 when releasing version 3.4.
- Today it is common that systems randomly arrange the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries. This technique is called *Address Space Layout Randomization* (ASLR) and its objective is to prevent an attacker from reliably jumping to, for example, a particular exploited function in memory. First introduced as a Linux kernel patch. OpenBSD was the first mainstream operating system to support ASLR by default, in version 3.4, released in 2003. Different systems apply this technique with different parameters and on different parts of a program. In general it is possible to introduce randomness into other parts of a system, for example process IDs, initial TCP sequence numbers, etc.

Program termination

void **exit**(int *status*);

- terminates a process with a return value *status*. Never returns.

pid_t **wait**(int **stat_loc*);

- waits for a child process termination, returns its PID and puts termination information into *stat_loc* which can be tested as:
  - `WIFEXITED(stat_loc)` ... process called `exit()`
  - `WEXITSTATUS(stat_loc)` ... argument of `exit()`
  - `WIFSIGNALED(stat_loc)` ... process got a signal
  - `WTERMSIG(stat_loc)` ... signal number
  - `WIFSTOPPED(stat_loc)` ... process stopped (`WUNTRACED` flag required, need **waitpid** below)
  - `WSTOPSIG(stat_loc)` ... stop signal number

pid_t **waitpid**(pid_t *pid*, int **stat_loc*, int *opts*);

- waits for a specific child process termination

- *status_loc* equal to `NULL` means to ignore the status information.
- Function **_exit** works as **exit** but it does not flush stdio streams and functions registered with the **atexit** call are not called.

- There is also `WIFCONTINUED(stat_loc)` which means a restarted process after having been stopped before. However, it is part of an extension that not all systems support.

- You can stop a process using “`kill -STOP <PID>`”, and restart it with ”`kill -CONT <PID>`”.

- *opts* in **waitpid** are an OR combination of the following flags:

  - `WNOHANG` ... does not hang if there are no processes that wish to report status
  - `WUNTRACED` ... children of the current process that were stopped due to a `SIGTTIN`, `SIGTTOU`, `SIGTSTP`, or `SIGSTOP` signal also have their status reported. Such processes are reported only once per such a situation.
  - `WCONTINUED` ... also report children of the current process that were restarted after having been stopped (and not waited for yet). Part of the same extension as `WIFCONTINUED`.
  - For the `WUNTRACED` and `WCONTINUED` flags, you should only use them in portable code if a macro `_POSIX_JOB_CONTROL` is defined in `<unistd.h>`.

- *`pid`* in **waitpid**:

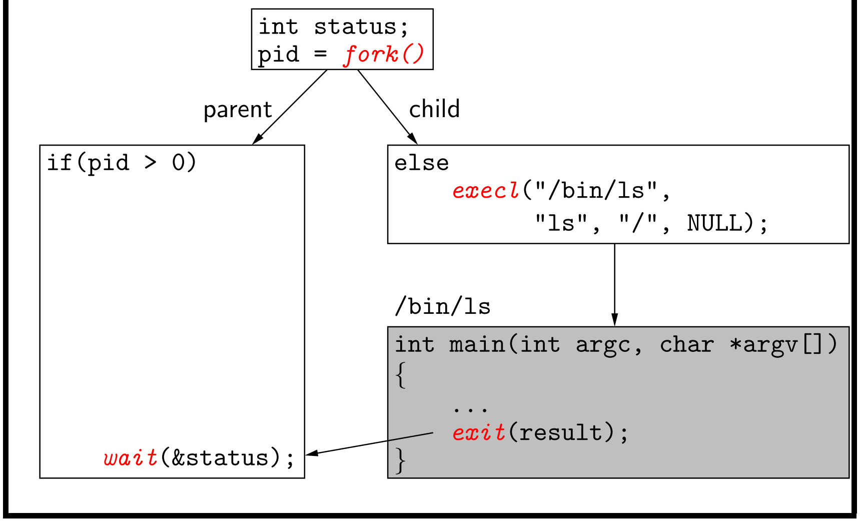
  - `== -1` ... wait for any child
  - `> 0` ... wait for a specific child
  - `== 0` ... wait for any child in the same process group as the calling process
  - `< -1` ... wait for any child in the process group of `abs(pid)`

- There are also **wait3** and **wait4** calls. These are more generic versions, also allowing to gather resource utilization statistics from exited child.

- A parent should always call one of the wait functions otherwise the system will accumulate *zombies* – terminated processes that occupy process table slots only to be waited for by their parents. Zombies could eventually exhaust all the system memory. Note that if the parent exits, its children are adopted by the `init` process that will call **wait** on such processes. However, you should always use wait for children even if you know the parent will exit soon.

- Actually, you could notify the system that the program will not wait for its children in which case such zombies will not accumulate. See page 132.

Example: start a new process and wait



```
int status;
pid = fork()
```

parent

```
if(pid > 0)



        wait(&status);
```

child

```
else
    execl("/bin/ls",
          "ls", "/", NULL);
```

/bin/ls

```
int main(int argc, char *argv[])
{
    ...
    exit(result);
}
```

- This is a typical way to start a new process and continue after its termination. The parent could also choose not to wait for the child termination right away but carry on with its life and wait for the child later.
- Note that you have to use macros from the previous slide to get the child's return value out of the status information.
- Example: wait/wait.c
- Creating new process with `fork` and replacing the process address space etc. with a new one after `exec` has been done is expensive and has other problems. To make kernel create new process directly from executable `posix_spawn` can be used. Example: exec/spawn.c

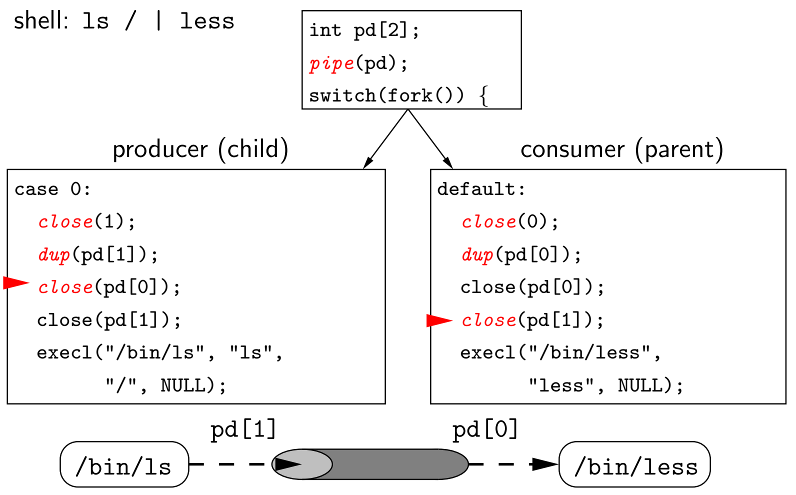
pipe()

`int pipe(int fildes[2]);`

- creates an unnamed pipe and allocates a pair of file descriptors
  - `fildes[0]` … reading from a pipe
  - `fildes[1]` … writing to a pipe
- the system makes sure that:
  - producer blocks on writing if the pipe is full
  - consumer blocks on reading if the pipe is empty
- consumer gets `EOF` (i.e. `read()` returns `0`) only if all copies of `fildes[1]` are closed.
- named pipe (i.e. FIFO, see **mkfifo**) works the same way. The difference is any process can use it.

- An unnamed pipe is created by one process and can be passed to its children only via file descriptors inherited through **fork**. That limitation can be worked around via passing an open file descriptor via a unix-domain socket. However, such a workaround is out of scope for this class.
- If the function **write** writes at most `PIPE_BUF` bytes to the pipe, it is guaranteed that the write will be atomic, i.e. such bytes will not be intermingled with bytes written by other producers.
- The SUSv3 standard does not specify whether `fildes[0]` is also open for writing and if `fildes[1]` is also open for reading. FreeBSD and Solaris provide bidirectional pipes while Linux may not. It is best to assume unidirectional pipes.
- **Important:** the same rules applied to reading and writing from/to named pipes stand for unnamed pipes as well, see page 66.
- Example: `pipe/broken-pipe.c`, `pipe/deadlock-in-read.c`

Example: a pipe between two processes



shell: `ls / | less`

```
int pd[2];
pipe(pd);
switch(fork()) {
```

```
case 0:
  close(1);
  dup(pd[1]);
  close(pd[0]);
  close(pd[1]);
  execl("/bin/ls", "ls",
        "/", NULL);
```

```
default:
  close(0);
  dup(pd[0]);
  close(pd[0]);
  close(pd[1]);
  execl("/bin/less",
        "less", NULL);
```

- Remember, open file descriptors are unaffected by **exec** aside from file descriptors with the `FD_CLOEXEC` flag – those are closed in the **successful exec** call.
- Example: `pipe/pipe-and-fork.c`
- Closing the writing descriptor `pd[1]` (see ▷) in the consumer process is required as the EOF would not be detected otherwise.
- Closing the reading descriptor `pd[0]` in the producer process is desired as well (see ▷) as if the consumer finishes prematurely the producer properly gets a `SIGPIPE`. If that file descriptor in the producer was not closed while the consumer died in the middle of processing the data, for example, the producer would not learn that the consumer was gone (as the producer would remain to be an existing reader itself), and would hang indefinitely on **write** after filling up the pipe.
- If we are not sure that the descriptor `0` was open before calling **pipe**, we have to call `dup2(pd[1], 1)` in the producer as otherwise `dup(pd[1])` could reuse the file descriptor `0` in place of expected `1`. You might also need to check if `pd[1] == 1` (i.e. standard output was closed before calling **pipe**) as in that case we could actually close one end of the pipe. Similarly, in the consumer you might need to check if `pd[0] == 0`.
- With regards to data flow, it is better to create a pipe from a child to its parent as typically the process writing the pipe finishes first, then the consumer reads the rest of the data, processes it, and then finally exits. In general, the

shell waits for the program it started, i.e. the parent, and it does not care at all about children the running program spawned during its life. If the pipe was created the other way around, the shell could print the prompt after the parent finished while the data from the child might still be flowing to the console. However, nowadays the shell waits for all processes in the pipeline. For example, the following will take 10 seconds before you get the prompt even though `cat(1)` might exit right away if `sleep(1)` closed its standard error before putting itself to sleep: `date | sleep 10 | cat`.

- For example, the original *Bourne shell* constructed a pipeline the way that the last process created a child as its producer, that producer itself created its child as its producer, and this continued until the whole pipeline was formed, i.e. the first command in the pipeline was created as the last process.
- However, in `bash`, all processes in a pipeline are direct children of the shell itself, i.e. `bash` calls **fork** that many times as is the number of programs in the pipeline. Before the prompt is printed, it waits for all processes it directly created to finish.

Shared memory – introduction

- using pipes and files to communicate between processes requires system calls
  - pros: processes cannot corrupt address space of one another
  - cons: significant syscall overhead (typically **read**, **write**)
- *shared memory* means to map a part of memory into an address space of multiple processes
- syscall overhead is gone but processes may become dangerous to one another
- memory access synchronization is needed
  - System V semaphores
  - POSIX semaphores

- Memory file mapping is one of the implementations of shared memory. A file name is used to describe the shared piece of memory.
- A memory mapped file is regularly written to disk.
- Memory based disks and filesystem are provided by today's system. They primarily rely on main memory for data storage, thus avoiding overhead of physical storage reads and writes.

Mapping files to memory (1)

```
void *mmap(void *addr, size_t len, int prot, int flags,
                int fildes, off_t off);
```

- A file section of *len* bytes, starting at position `off` in a file represented by a file descriptor `fildes`, mapped to the process space starting at address *`addr`* (`0` . . . kernel assigns the address).
- Returns the address of the mapped section, or `MAP_FAILED`.
- In *`prot`*, there is an OR combination of `PROT_READ` (read allowed), `PROT_WRITE` (write allowed), `PROT_EXEC` (execution allowed), or `PROT_NONE` (no access allowed).
- In *flags* there is an OR combination of `MAP_PRIVATE` (changes private to the process, not saved to the file), and `MAP_SHARED` (modifications are saved to the file)

- Examples: `mmap/reverse.c`, `mmap/map-nocore.c`
- in the flags **exactly one** of `MAP_PRIVATE`, `MAP_SHARED` has to be present. There is also `MAP_FIXED`, with which the kernel tries to map the section at *`addr`* even if it means to destroy already existing mapping. Without `MAP_FIXED`, the kernel will try to use *`addr`* but if there is an existing mapping, it will try to find another address to use (and returns it).
- mapping files in memory is alternative to processing files using `read`, `write`, `lseek`. After the file is mapped it is possible to work with it as a data structure in memory. The file is not being copied to the memory as a whole, rather only the pages which are accessed are allocated. If it is necessary to free some page, the contents are written back to the storage, i.e. the file. That is, if the `MAP_SHARED` is used – this way of mapping is therefore equivalent to writing back via `write(2)`) or to the swap – same copy-on-write mechanism is used (when using `MAP_PRIVATE`).
- For the file to be mapped into memory it is first necessary to open it via `open`. The access mode in `prot` cannot be "higher" than specified in the mode for `open`.
- The use of `MAP_FIXED` is not recommended, since it hampers portability of the code. Also, it removes any existing mappings at that address. Further, it does not work well with ASLR (Address Space Layout Randomization) which is frequently used in current Unix systems for dynamically linked programs.

- **warning:** w.r.t. MAP_SHARED – if the file is truncated by another process so that the truncation affects the mapped part, the access to such area results in the SIGBUS resp. SIGSEGV or signal to be sent to the process. One way how to get around it is to use mandatory locking, however it is not implemented in all the systems. In case the mapped memory is used as a parameter for write, the signal will not be sent and write will return -1 with errno set to EFAULT.

- when using MAP_PRIVATE on Solaris all the changes done by other processes that have mapped the file using MAP_SHARED are visible until the process writes to the page. – at this moment a copy of the page is created and subsequent changes by other processes are not visible anymore. On FreeBSD the changes are not visible even before the first write. The spec says: *„It is unspecified whether modifications to the underlying object done after the MAP_PRIVATE mapping is established are visible through the MAP_PRIVATE mapping."*

- the value of off+len can exceed current size of the file, however it is not possible to write beyond the end of the file and thus extend the file - the process would otherwise receive the SIGBUS or SIGSEGV signal, see example mmap/lseek.c. The signal is also received when attempting to write to file mapped read-only. (this makes sense, the assignment does not have a return value which can be tested)

- the mapping is done in whole pages, the off values have to be aligned correctly (also when using MAP_FIXED the addr). Last page after the end of file is padded with zeroes and this section is never written back to the file.

- It is possible to share anonymous mapping between processes using fork(). The only alternative is shared memory established using shmat.

- Access to the mapped region beyond the existing page of mapped object causes SIGBUS or SIGSEGV signal. This is not universally true, see example mmap/sigbus.c.

- When using MAP_FIXED the new mapping will replace any existing mapping from addr to addr+len-1, see example mmap/override.c.

- extensions not part of SUSv3:
  - MAP_ANONYMOUS flag – will create anonymous segment (without association to a file), the descriptor has to be -1. An anonymous object is mapped that has storage in swap area. This flag is frequently used by heap allocators. see also page 95.
  - in IRIX it is possible to use MAP_AUTOGROW that will automatically grow mapped object when accessing beyond its current end.

- Example of system command using file mapping is cat(1). Using mapped memory is more efficient than calling read repeatedly, saving the overhead necessary for switching to kernel and back to userspace.

Mapping files to memory (2)

int **msync**(void **addr*, size_t *len*, int *flags*);

- will write back specified pages in the region of len bytes from address `addr`. The flags is OR-combination of:
  - `MS_ASYNC` ... asynchronous write
  - `MS_SYNC` ... synchronous write
  - `MS_INVALIDATE` ... destroy mapped data, different to file contents

int **munmap**(void **addr*, size_t *len*);

- write back changes, destroy mapping of length `len` from address `addr`.

int **mprotect**(void **addr*, size_t *len*, int *prot*);

- change access rights to mapped section of a file. The values `prot` are the same as for `mmap()`.

- writing the changes to the storage is guaranteed only after `msync` or `munmap`, however other processes that have the file mapped as well will see the changes immediately.
- mapping to memory and setting access rights is used by the Electric Fence library, which is used for detection of errors when using dynamic memory.

Example: mapping files into memory

```
int main(int argc, char *argv[])
{
    int fd, fsz; char *addr, *p1, *p2, c;

    fd = open(argv[1], O_RDWR);
    fsz = lseek(fd, 0, SEEK_END);
    p1 = addr = mmap(0, fsz, PROT_READ|PROT_WRITE,
                     MAP_SHARED, fd, 0);
    p2 = p1 + fsz - 1;
    while(p1<p2) {
        c = *p1; *p1++ = *p2; *p2-- = c;
    }
    munmap(addr, fsz);
    close(fd);
    return (0);
}
```

- This program will reverse the bytes in a file.
- One of the advantages of shared memory segments is that it is possible to work with the data using pointer arithmetic. Generally it is necessary to watch out for alignment when dereferencing a pointer, e.g. on SPARC unaligned access will cause the `SIGBUS` signal, see example `mmap/aligned.c`.

Accessing dynamic link libraries

`void *dlopen(const char *path, int mode);`

- loads *path* unless already loaded, returns a **handle** or `NULL`
- in *mode* there is an OR combination of `RTLD_NOW` (immediate bind), `RTLD_LAZY` (bound when needed), `RTLD_GLOBAL` (symbols accessible from all modules), `RTLD_LOCAL` (symbols accessible through the handle only)

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

- returns the address of *symbol*

`int dlclose(void *handle);`

- closes the dynamic library

`char *dlerror(void);`

- gets diagnostic information

- For example, you could use these functions to implement a dynamically loadable plug-in modules. Modules to load could be determined either on the fly based on user input, or could be read from a configuration file.
- If you have multiple dynamic libraries that all define a symbol of the same name, you could link one of those during the build process, and other libraries can be accessed via **dlopen**.
- You need a file of a correct format. For example, on a Linux distribution, that is an ELF shared library with an `.so` suffix. With `gcc`, you will need `-shared` to build such a library there. If on macOS (file format *Mach-O*), you will need `-dynamiclib`. Libraries there have a `.dynlib` suffix.
- If *path* contains `/`, it is taken either as global or relative, if not, the dynamic linker uses its default path to search for the library, usually in `/lib` and `/usr/lib`. You can extend the list of paths to search via an environment variable `LD_LIBRARY_PATH`. Be careful when using it, see page 32.
- flags for *mode* of **dlopen**:
  - `RTLD_NOW` – all undefined symbols in the library are resolved before **dlopen** returns. `NULL` is returned if it cannot be done. That is the default.
  - `RTLD_LAZY` – only resolves symbols as the code that references them is executed. However, references to variables are always immediately bound when the library is loaded. You can force the specific behavior via environment variables `LD_BIND_NOW` and `LD_BIND_LAZY`. If there is a

conflict, NOW prevails. When a program is started, all linked libraries are bound right away unless the program or individual libraries are linked for “lazy binding” via -z lazy (gcc). See manual pages for ld a ld.so.1 (Solaris) or ld.so (Linux). Example: dyn-lib/ld-lazy.c.

  - RTLD_GLOBAL … symbols defined by the shared object will be made available for symbol resolution of subsequently loaded objects. That is the default for objects mapped when executing a program. For **dlopen**, the default is RTLD_LOCAL. It means the same library can be mapped multiple times via **dlopen** and the symbols in the mapped instances of the same library will not overlap. However, all globally mapped symbols from there are shared, e.g. errno.

- Special handle RTLD_NEXT searches the symbol only in libraries loaded after the library that called **dlsym**. Handy for redefining existing functions if we need to call the original function in our redefined one. The library with a modified function is loaded first, possibly using LD_PRELOAD, and the address of the original function can be found using

  dlsym(RTLD_NEXT, *fn_name*).

  Example: dyn-lib/rtld_next.c.

- All these functions are part of the dynamic linker that each dynamically linked program has mapped into its address space upon execution. Also see pages 30 and 106.

Example: accessing a dynamic library

```
void *handle;
double y, x = 1.3;
double (*fun)(double);
char *libname = "libm.so", *fn_name = "sin";

if ((handle = dlopen(libname, RTLD_NOW)) == NULL)
        err(1, "dlopen: %s", dlerror());

if ((fun = dlsym(handle, fn_name)) == NULL)
        err(1, "dlsym: %s", dlerror());

y = fun(x);
dlclose(handle);
```

- The code calls function **sin** from `libm.so`, the math library.
- **dlsym** returns the symbol address but always as a pointer to `void`, there is no type check, and no type information is available either. The caller must retype the returned address.
- If you use C++, note that *name mangling* is used.
- Example: `dyn-lib/dlopen.c`

Contents

- Introduction, Unix and C, programming tools
- Basic Unix concepts and conventions, its API
- Access rights, devices
- Process manipulation, program execution
- **Signals**
- Process synchronization and interprocess communication
- Network programming
- Programming with threads
- Appendix

Introduction to signals

- for notifying a process of asynchronous events, and for exception handling
- mechanism of an interrupt available in user level
- signal categories:
  - **asynchronous events** happening independently of the main program flow, e.g. a signal sent from another process, a timer expiration (`SIGALRM`), terminal disconnect (`SIGHUP`), or pressing `Ctrl-C` (`SIGINT`)
  - **exceptions** caused by a running process, e.g. attempt to access a restricted area of memory (`SIGSEGV`)

- Interrupts can be viewed as a mean of communication between the CPU and the OS kernel while signals are for communication between the kernel and processes.
- After returning from the handler, if that happens, the process continues exactly from the place where the interruption happened.
- Historically, signals were provided as a mechanism to forcefully terminate processes. That is why the function for sending signals is called **kill**.
- A nice example of an asynchronous event is while on Linux you send a `SIGUSR1` to the `dd` command to print I/O statistics to standard error output. Start `dd` first, then send a couple of `USR1` signals its way, like this:

  ```
  $ kill -USR1 $(pgrep -f "dd if=/dev/zero of=/dev/null")
  ```

  You will see something like this:

  ```
  $ dd if=/dev/zero of=/dev/null
  9179287+0 records in
  9179286+0 records out
  4699794432 bytes (4.7 GB, 4.4 GiB) copied, 1.79083 s, 2.6 GB/s
  14211424+0 records in
  14211423+0 records out
  7276248576 bytes (7.3 GB, 6.8 GiB) copied, 2.76889 s, 2.6 GB/s
  ```

Introduction to signals (continued)

- it is the simplest inter process communication – it only carries information that an event happened
  - there is a real-time POSIX extension that allows for more information, more on that later
- mostly processed asynchronously – a signal interrupts the current process flow and a *signal handler* is invoked
- signals are identified by numbers, represented by names like `SIGSEGV`, `SIGCHLD`, or `SIGKILL`
- name are usually `#defines`, see `/usr/include/signal.h` or `/usr/include/sys/signal.h`

- The real-time extension is POSIX-1003.1b. 132.
- You can process signals in a synchronous way, see **sigwait**() on page 134.

Sending signals

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

- sends signal *`sig`* to a process or a process group based on value of *`pid`*:
  - `> 0` ... to process with *`pid`*
  - `== 0` ... to all processes in the same group
  - `== -1` ... to all aside from system processes
  - `< -1` ... to processes in a group `abs(pid)`
- `sig == 0` means the system only checks whether the process has enough privileges to send a signal without sending it
- whether a process may send other process a signal depends on UID of both processes

- Traditionally, process with EUID == 0 can send a signal to any other process. However, some systems optionally provide fine grain privileges and the situation there is different even for root. That is out of scope for this class though.
- Sending a signal to another process:
  - Linux, Solaris: RUID or EUID of the process that sent the signal must match the real UID or saved SUID of the target process.
  - FreeBSD: EUID of the processes must match
- Example: `signals/killing-myself.c`
- 0 signal can be also used for a simple check for the specific process existence, see `signals/check-existence.c`.

Handling signals

- unless a process sets it otherwise, each signal triggers a specific default action, one of:
  - terminate the process (**exit**)
  - terminate and dump a core (**core**)
  - ignore the signal (**ignore**)
  - stop the process (**stop**)
  - resume the process (**continue**)
- process can either set to ignore a specific signal…
- …or can handle the signal via a user-defined function, called a **handler**

Signals `SIGKILL` and `SIGSTOP` **always** trigger an implicit action, i.e. exit or stop, respectively.

- Creating a core dump means to store the contents of the process virtual address space to a file. Usually such file has a word `core` in its file name. Some systems, e.g. macOS, may not generate core dumps by default even if its for signals with default core dump action. In that case, `ulimit -c unlimited` usually helps, and it affects the current shell only. See the present shell limits with `ulimit -a`.
- Most of the signals implicitly terminate the process, some create a core dump on top of that to enable a post-mortem analysis.
- The reason why **exec**() replaces all user set handlers to its implicit action (see also page 106) is obvious – code of the original handlers no longer exists after the **exec**() call finishes.
- You can learn the signal numbers and their names using the `-l` option for the `kill(1)` command. Without the argument, it will print the list of all signals with their corresponding numbers. Example:

  ```
  $ kill -l SIGPIPE
  13
  ```

- For each signal, you can learn what is its implicit action by checking a manual page for function **signal**() or possibly for *signal.h*. On Linux distributions, it is usually in the `signal(7)` manual page, accessible via `man 7 signal`.

Signal listing (1)

We could divide signals into a few groups. . .

Detected errors:

`SIGBUS`	bus error, e.g. wrong alignment (core)
`SIGFPE`	floating point exception (core)
`SIGILL`	illegal instructions (core)
`SIGSEGV`	segmentation violation (core)
`SIGPIPE`	write on a pipe with no reader (exit)
`SIGSYS`	non-existent system call invoked (core)
`SIGXCPU`	CPU time limit exceeded (core)
`SIGXFSZ`	file size limit exceeded (core)

- Those signals are generated on an error in a program.
- For the first four signals – `SIGBUS`, `SIGFPE`, `SIGILL`, and `SIGSEGV` the standard does not specify what exactly has to be the reason but usually those are errors detected by hardware. Try the following examples, and check the return value with "`kill -l $?`": `signals/sigsegv.c`, `signals/div-by-zero.c`.

  The Unix shell convention is that if a program is killed by a signal, `$?` will be 128 plus the signal number (`Ctrl+\` sends any foreground process a `SIGQUIT` signal):

  ```
  $ sleep 99
  ^\Quit: 3
  $ echo $?
  131
  $ kill -l 131
  QUIT
  $ kill -l $((131 - 128))
  QUIT
  ```

- **For those four signals, there are some special rules as well** (for details, see section *2.4 Signal Concepts* in SUSv4):
  - If set as ignored by function **sigaction**(), the program behavior after such a signal is delivered is undefined.
  - The behavior of a process is undefined after it returns from a signal-catching function.

  - If such a signal is masked while the signal is being delivered, the program behavior is undefined.

- Bottom line is that if a hardware generated error is real (i.e. the signal is not send via **kill** or similar functions), the process may never get over that error at all. It is not safe to ignore such errors, continue after returning from a handler, or mask such signals. You can catch such signals though, the standard does not prohibit that. However, if you do, you should do so only to deal with the situation and exit. You can check `signals/catch-SIGSEGV.c`. More information and another example can be found on page 194.

- Note: if the standard specifies any behavior as *undefined*, it means the specification does not state what should happen and that whatever happens does not violate the standard. So, if you trigger such a behavior and your computer burns down or even flies off to the Moon, possibly still in flames, that does not violate the standard either.

Signal listing (2)

Signals generated by a user or application:

`SIGABRT`	abort program (core)
`SIGHUP`	terminal line hangup (exit)
`SIGINT`	interrupt program via `Ctrl-C` (exit)
`SIGKILL`	kill program (exit, **cannot be caught or ignored**)
`SIGQUIT`	quit program via `Ctrl-\` (core)
`SIGTERM`	software termination signal (exit)
`SIGUSR1`	user-defined signal 1 (exit)
`SIGUSR2`	user-defined signal 2 (exit)

- Signal `SIGHUP` is often used as a way to let a daemon know its configuration file changed and that it should re-read it.

- `SIGINT` and `SIGQUIT` are usually generated from a terminal via `Ctrl-C` and `Ctrl-\`, respectively, and could be redefined using the command `stty` or via a function **tcsetattr**(). In order to generate core files, your system must allow it. Check command `ulimit`.

- As `SIGKILL` cannot be handled, use it only if you know what you are doing. For example, if a running process does not respond to user input nor any other signal. Many applications, mainly daemons, rely on the fact that they are sent SIGTERM on termination, which they often handle and perform actions before exit. For example, flushing the database, removing temporary files and file locks, etc. So, do not use SIGKILL right away only because ”it’s the simplest way to kill a process” as you might run in trouble.

- Example on using SIGQUIT on Solaris:

```
$ sleep 10
^\Quit (core dumped)
$ mdb core
Loading modules: [ libc.so.1 ld.so.1 ]
> $c
libc.so.1‘__nanosleep+0x15(8047900, 8047908)
libc.so.1‘sleep+0x35(a)
main+0xbc(2, 8047970, 804797c)
_start+0x7a(2, 8047a74, 8047a7a, 0, 8047a7d, 8047b91)
>
```

- SIGTERM is the default signal for command kill(1).
- SIGUSR1 and SIGUSR2 are not used by any system call and are available for use to the user.

Signal listing (3)

Job control:

SIGCHLD	child status has changed (ignore)
SIGCONT	continue after stop (continue)
SIGSTOP	stop (stop; **cannot be caught or ignored**)
SIGTSTP	stop from terminal Ctrl-Z (stop)
SIGTTIN	background read attempted from control terminal (stop)
SIGTTOU	background write attempted to control terminal (stop)

- Those used to be part of a non-mandatory POSIX extension but now they are required by POSIX.1-2008. See also page 5.

- Only one process at any given time can read from its process group control terminal but multiple processes can write to it at the same time.
- Stopping a process group from a terminal, usually via `Ctrl-Z`, is done with signal `SIGTSTP`, not `SIGSTOP`; a program thus can catch the signal.

Signal listing (4)

Timers:

`SIGALRM`	timer expired (exit)
`SIGPROF`	profiling timer alarm (exit; see **setitimer**(2))
`SIGVTALRM`	virtual time alarm (exit; also see **setitimer**(2))

Miscellaneous:

`SIGPOLL`	event occurred on explicitly watched file descriptor (exit)
`SIGTRAP`	trace trap (core)
`SIGURG`	urgent condition present on socket (ignore)

- `SIGALRM` and related function **alarm** is used for setting timer alarms, useful for implementation of timeouts, for example.

Setting actions for signals

```
int sigaction(int sig, const struct sigaction *act,
              struct sigaction *oact);
```

- assigns an action *act* for a signal *sig*, previous setting is returned in *oact*, if non-zero
- structure `sigaction` contains:
  - `void (*sa_handler)(int)` ... `SIG_DFL`, `SIG_IGN`, or a handler address (i.e. a handler function name)
  - `sigset_t sa_mask` ... signals blocked while in handler; plus *sig* is blocked by default
  - `int sa_flags` ... `SA_RESETHAND` (handler is reset to `SIG_DFL` at the moment the signal is delivered), `SA_RESTART` (restart pending calls), `SA_NODEFER` (do not block *sig* while in handler)

- for manipulating signal sets, use **sigaddset**(), **sigdelset**(), **sigemptyset**(), **sigfillset**(), **sigismember**()
- If `act == NULL` you only get the present setting and nothing is changed. If not interested in previous setting, use `NULL` for *oact*.
- If `SA_RESTART` is not set, the system call interrupted by the signal will return an error with `errno` set to `EINTR` (some calls actually return directly the `errno` value but that is not relevant now). However, resetting the call will not work for all system calls. You should consult the documentation for your system if you intend to use it. On Linux, check the `signal(7)` manual page. Example: `signals/interrupted-read.c`.
- Beware of deadlocks (will be discussed in the chapter on synchronization). If you set `SA_NODEFER`, your handler needs to be reentrant.
- **You should only use functions in a safe way from a handler**. By safe it is meant either use reentrant functions or you need to make sure the signal is not delivered in the wrong time (for example, the signal is delivered within a function and the same function is called from the handler, and the function is not ready for that). Minimal set of functions that must be ***async-signal-safe*** is listed in SUSv4 in section *System Interfaces: General Information ⇒ Signal Concepts ⇒ Signal Actions (2.4.3)*. Systems can extend the list, of course. Whether a function is safe to use in a signal handler or not should be documented in its manual page. On Linux, see also the `signal-safety(7)` manual page.

- Using static data and/or locking in a function would generally be a problem for its asynchronous signal safety. It can lead to corrupt data, deadlocks, etc. As the set of functions safe to use in a handler is limited, one way to use the signals is only to set a global variable in the handler and test it later, for example in a loop that processes some events. As a function waiting for an event is typically interruptible by a signal (see above), setting the global variable and testing it later does not experience any real delay. Example: `signals/event-loop.c`.

- There is also a simplified signal facility **signal**(). We recommend to use **sigaction**() only. Behavior of **signal**() is not specified with threads, for example. It can also behave in a very different way on different systems, some keep the handler set after delivering the signal, some reset it to `SIG_DFL`. Check `signals/signal-vs-sigaction.c` if interested.

- If your system supports a part of POSIX.1b called *Realtime Signals Extension* (RTS), it is possible to use the extension if you use flag `SA_SIGINFO`. In that case a `sa_sigaction` member of the structure `sigaction` must be used for the handler, not `sa_handler`. The new handler has three parameters and it is possible to learn the PID of a signalling process, its UID, and more. See manual page `signal.h(3HEAD)` on Solaris, the online SUS specification of the header file, or the book [POSIX.4], page 9. More information is also on page 7 and 135. Examples: `signals/siginfo.c`, `signals/sigqueue.c`.

- By ignoring `SIG_CHILD` you say you will not wait for child processes and the system will still not accumulate zombies.

Example: setting time limit for read

```
void handler(int sig)
{ write(2," !!! TIMEOUT !!! \n", 17); }

int main(void)
{
    char buf[1024]; struct sigaction act; int sz;
    act.sa_handler = handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGALRM, &act, NULL);
    alarm(5);
    sz = read(0, buf, 1024);
    alarm(0);
    if (sz > 0)
        write(1, buf, sz);
    return (0);
}
```

- You can also use timers with a finer granularity provided by **setitimer**() and **getitimer**() functions. After the timer expires, a process is sent a signal based on the value of the first argument *which*:
  - `ITIMER_REAL` ... timer decrements in real time, and `SIGALRM` is used.
  - `ITIMER_VIRTUAL` ... timer decrements in process virtual time which runs only when the process is executing. A signal `SIGVTALRM` is used.
  - `ITIMER_PROF` ... timer decrements both in process virtual time and when the system is running on behalf of the process, and `SIGPROF` is used.
- Note that we use **write**() system call to write the text. Functions like **printf**() are generally not safe to use in signal handlers, see page 131.
- Example: `signals/alarm.c`

Signal blocking

- blocked signals are delivered to the process after getting unblocked again

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

- sets the blocked signal mask and returns the old one
- *how* – `SIG_BLOCK` adds signals to block, `SIG_UNBLOCK` for removing signals, for a complete mask change use `SIG_SETMASK`

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

- returns a mask for the signals pending for delivery

- The system silently ignores attempts to block `KILL` and `STOP`.
- It is a difference to ignore and block a signal. A signal ignored will never be delivered to the process, while a blocked signal is remembered by the kernel and is delivered if the process unblocks it.
- It is implementation dependent if a signal received multiple times while blocked will be delivered only once or multiple times after unblocking.
- If POSIX.4 extension is used (see page 132 and the use of `SA_SIGINFO`), signals are delivered through a queue and multiple signals are never squeezed into one delivery.

- Both arguments *oset* and *set* may be NULL. If both are NULL in the same invocation of the function, nothing happens.

- sigpending sets the set in the argument to the pending mask. See signals/sigpending.c.

Example: blocking signals

```
sigset_t sigs, osigs; struct sigaction sa;
sigfillset(&sigs); sigprocmask(SIG_BLOCK, &sigs, &osigs);
switch(cpid = fork()) {
    case -1:
        sigprocmask(SIG_SETMASK, &osigs, NULL);
        ...
    case 0: /* Child */
        sa.sa_handler = h_cld; sigemptyset(&sa.sa_mask);
        sa.sa_flags = 0;
        sigaction(SIGINT, &sa, NULL);
        sigprocmask(SIG_SETMASK, &osigs, NULL);
        ...
    default: /* Parent */
        sigprocmask(SIG_SETMASK, &osigs, NULL);
        ...
}
```

- The example shows a situation when a child needs to use a different signal handler from its parent. If you just changed the handler in the child without blocking the signals before **fork**() a potential race would have been created – a child would have run for a short time with an unwanted handler. Note that **fork**() does not change the signal mask, see page 103.

- Out of simplicity we block all the signals in the example even that on pages 127 and 194 it is explained why it is not a good thing.

Waiting for a signal

`int` **pause**`(void);`

- suspends the caller until delivery of a signal that is not blocked or ignored

`int` **sigsuspend**`(const sigset_t *`*`sigmask`*`);`

- like **pause**(), also temporarily changes the blocked signal mask

`int` **sigwait**`(const sigset_t *`*`set`*`, int *`*`sig`*`);`

- waits for a signal from *set* (must be blocked) and puts the signal number to *`sig`*.
- the signal handler is not called

- When in **pause**() or **sigsuspend**(), a signal not blocked will call a handler if one is set, and the signal catching function returns unless the process is terminated by the signal.
- With **sigwait**(), you can implement a synchronous signal handling. First, a process blocks signals it is interested in, then when convenient, it waits for them or just checks whether there are some pending using **sigpending**().
- The **sigwait**() function was added with POSIX-1003.1c extensions (threads) and you should use this function when dealing with signals in the multithreaded application. As other POSIX thread functions, it returns an error number directly, and 0 on success.
- There are also similar functions **sigwaitinfo**() and **sigtimedwait**() defined with the POSIX-1003.1**b** extension (real-time). They work in a similar way but set `errno` on failure and it is possible to get more information upon signal delivery using a structure `siginfo_t`, see page 132.
- Example (a signal is used for synchronization of two processes communicating via a shared memory): `signals/sigwait.c`
- Do not use for synchronous signals like `SIGSEGV`, `SIGILL`, etc. More on that on pages 127 and 194.

Contents

- Introduction, Unix and C, programming tools
- Basic Unix concepts and conventions, its API
- Access rights, devices
- Process manipulation, program execution
- Signals
- **Process synchronization and interprocess communication**
- Network programming
- Programming with threads
- Appendix

Problem: conflict while sharing data

- `struct { int a, b; } shared;`
-
```
for( ; ; ) {
    /* non-atomic operation */
    a = shared.a; b = shared.b;
    if (a != b) printf("NON-CONSISTENT STATE");
    /* non-atomic operation */
    shared.a = val; shared.b = val;
}
```
- if the cycle is run in 2 processes running in parallel (or threads) that share the same structure and have different values of the `val` variable, it will lead to conflicts due to non-atomic operations.

- Operations that can be expressed in C with single statement does not have to be atomic. e.g. on RISC processors the command `a++` is typically translated into:

  ```
  load reg,[a]
  inc reg
  store [a],reg
  ```

  due to the fact that on this architecture numbers cannot be incremented directly in memory. For such cases there are sets of functions for atomic arithmetic operations (e.g. `atomic_add(3c)` on Solaris) that are much faster than classic synchronization mechanisms. More on page 205.

- In general similar problem happens when multiple processes share a system resource.

- example `race/race.c`

Conflict scenario

	Processes **A**`(val==1)` and **B**`(val==2)`	`a`	`b`
1.	initial structure state	?	?
2.	process **A** writes to member `a`	1	?
3.	process **B** writes to member `a`	2	?
4.	process **B** writes to member `b`	2	2
5.	process **A** writes to member `b`	2	1
6.	the structure is in inconsistent state and one of the processes will find out.		

- more possibilities:
  1. structure is in inconsistent state, e.g. `(1, 1)`
  2. process `B` writes `2` into member `a`
  3. process `A` reads the value of the structure `(2, 1)` earlier than process `B` writes member `b`

- Note that synchronization problems surface only after when the program is running on multiprocessor(core) machine or on more processors than what is used when development. Something to think about when testing.

Solution: mutual process exclusion

- it is necessary to ensure atomic operation on the structure, i.e. while one processes modifies the structure, the other cannot manipulate it.

Processes **A**`(val==1)` and **B**`(val==2)`	`a`	`b`
1. initial structure state	?	?
2. process **A** writes to member `a`	1	?
3. process **B** must wait	1	?
4. process **A** writes to member `b`	1	1
5. process **B** writes to member `a`	2	1
6. process **B** writes to member `b`	2	2
7. the structure is consistent.		

- It is necessary to ensure mutual exclusion also when reading, so that the process that is reading cannot read inconsistent state while another process is changing it. While writing it is necessary to exclude all other processes. While reading it is sufficient to exclude only writers.

- ***critical section*** is piece of code that can be executed only in one process (or thread), otherwise the operations can lead to inconsistent state, e.g. wrongly connected linked list, mismatched database indexes etc. It is possible to say that critical section is code which accesses or modifies resource shared with multiple processes (or threads) and therefore access to such code should be synchronized. Critical section should be as short as possible to limit contention. The second definition is more generic, it can also include situations where only one process (or thread) can change the state however if this is not happening, more processes can read simultaneously.

Problem: readers and writers conflict

- several processes running in parallel write protocol with operations to common log file. Each new record is appended to the end of the file.
- if the record writing operation is not atomic, the contents of multiple records can be interleaved.
- only single process can write.
- other processes read data from the log file.
- while reading record that is being written inconsistent data is retrieved.
- while writing, it is not possible to read. If no process is writing, multiple processes can read simultaneously.

- 2 situations are permitted: one writer or multiple readers
- On local disk the synchronization of writers can be archived via the `O_APPEND` flag, however this is not going to work on network file systems such as NFS or in case it is necessary to perform multiple `write()` operations for single record. Moreover this does not solve the situation of readers – it is still possible to read while writing.

Solution: file locking

- writer process locks the file for writing. Other processes (readers and writers) cannot work with the file and have to wait for the lock to be unlocked.
- reader process locks the file for reading. Writers have to wait for the lock, other readers can also lock the file for reading and read data.
- in each moment there can be at most one active lock for writing or multiple locks for reading. Both locks cannot be locked simultaneously.
- for efficiency each process should hold the lock for shortest time possible and if possible do not lock the whole file – only the section that is being worked with. Passive waiting is preferred, active waiting is suitable for very short time.

- 2 ways of waiting:

  active (busy waiting) – a process tests a condition in a cycle while it is not true.

  passive – a process registers itself in the kernel as a waiting for the condition and goes asleep. The kernel wakes it up if the condition becomes true.

- active waiting is justifiable only in special situations.

Synchronization mechanisms

- theoretical solution – mutual exclusion algorithms (Dekker 1965, Peterson 1981)
- interrupt disable (1 CPU), special *test-and-set* instructions
- **lock-files**
- tools offered by the operating system:
  - **semaphores** (POSIX, System V IPC)
  - **file level locking** (`fcntl()`, `flock()`)
  - thread synchronization: **mutexes** (surround critical sections, only one thread can hold the mutex), **condition variables** (the thread is blocked until another thread signals condition change) **read-write locks** (shared and exclusive locks, similar semantics to file level locking)

- Both Dekker and Peterson need to achieve the result via only shared memory, i.e. several variables shared by processes.
- Dekker's solution is presented as a first solution to the problem of mutual exclusion of 2 processes, without having to apply the mechanism of strict alternation, i.e. if second process does not express the will to enter critical section, the first can enter how many times it wants (and vice versa). Dekker's solution is not trivial, compare it with 16 year younger Peterson solution, e.g. on `en.wikipedia.org`.
- We are not going to deal with theoretical algorithms or compare hardware mechanisms used by the kernel. Instead we are going to focus on the use of file level locking (which use the atomicity of some file operations) and special synchronization primitives offered by the kernel.

Lock files

- for each shared resource there exists previously agreed file path. Locking is done by creating the file, unlocking by removing the file. Each process must check if the file exists and if yes, has to wait.

```
void lock(char *lockfile) {
    while( (fd = open(lockfile,
                      O_RDWR|O_CREAT|O_EXCL, 0600)) == -1)
        sleep(1);  /* waiting in a loop for unlock */
    close(fd);
}

void unlock(char *lockfile) {
    unlink(lockfile);
}
```

- The key is the `O_EXCL` flag.
- Because the exclusive flag is not accessible in most shells, the shell scripts usually use `mkdir` or `ln` (hard link) commands for lock file synchronization.
- example: `file-locking/lock-unlock.c` – it is to be used together with the `file-locking/run.sh` shell script.
- In case of process crash the locks are not removed and therefore other processes would wait forever. Thus it is prudent to write PID of the process that created the lock to the lock file. The process that is waiting for unlock can verify that the process with given PID number exists. If not, it can remove the lock file and retry. User level command that can do this is e.g. `shlock`(1) (on FreeBSD in `/usr/ports/sysutils/shlock`), however could cause this situation:
    - **watch out:** if multiple processes find out simultaneously that the process does not exist, it can lead to error because the operation of reading file contents and removing it is not atomic:
        1. process A reads the contents of lock file, checks PID existence
        2. process B reads the contents of lock file, checks PID existence
        3. A deletes the lock file
        4. A creates new lock file with its PID
        5. B deletes the lock file
        6. B creates new lock file with its PID

7. Now both processes think they acquired the lock.

- Another **problem** is that the `lock()` function above contains active waiting. This can be solved e.g. so that the process that acquired the lock can open named pipe for writing. Reader processes will enter sleep by reading from the pipe. The `unlock()` will close the pipe and so the waiting processes will be unblocked.
- Lock files are usually used for situations where multiple instances of the same program are unwanted.

file locking: `fcntl()`

`int` **fcntl**`(int` *`fildes`*`, int` *`cmd`*`, ...);`

- to set locks for file `fildes`: `cmd`:
  - `F_GETLK` ... takes lock description from 3rd argument and replaces it with description of existing lock that collides with it.
  - `F_SETLK` ... sets or destroys lock described by the 3rd argument. If the lock cannot be set immediately returns -1.
  - `F_SETLKW` ... like `F_SETLK`, however puts the process to sleep if it is not possible to set the lock (`W` means "wait")
- 3rd argument contains lock description and is pointer to `struct flock`

- Locking of files over NFS is done via the `lockd` daemon.
- There are 2 types of locks:

advisory locks – for correct operation it is necessary that all processes working with locked files to check locks before reading/writing. used more frequently.

mandatory locks – if a file is locked, read/write operations will automatically block the process, i.e. the lock will be applied also on processes that do not explicitly work with the lock.

- not universally recommended, do not always work (e.g. `lockd` implements just advisory locking)
- for given file they are enabled by setting the SGID bit and removing right to execute for the group (i.e. setting that otherwise does not make sense). One process sets the lock (e.g. using `fcntl`). Other

processes then do not have to check the lock explicitly because each `open/read/write` operation is checked by the kernel against the file locks and enforces waiting till the lock is explicitly unlocked by the owner process.

  - implemented e.g. on Solaris, Linux. FreeBSD does not support it. The fcntl(2) man page on Linux (2013) does not recommend it because the implementation contains errors that could lead to race conditions and therefore the consistency cannot be generally achieved.

- It is important to realize that when process exits, all its locks are released.

File locking: `struct flock`

- `l_type` ... lock type
  - `F_RDLCK` ... shared lock (for reading)
  - `F_WRLCK` ... exclusive lock (for writing)
  - `F_UNLCK` ... unlock
- `l_whence` ... same as for `lseek()`, i.e. `SEEK_SET`, `SEEK_CUR`, `SEEK_END`
- `l_start` ... start of locked region with regards to `l_whence`
- `l_len` ... length of the region in bytes, 0 means till the end of the file
- `l_pid` ... PID of the process holding the lock, used only for `F_GETLK` when returning.

- If given part is not locked when using `F_GETLK`, the `flock` structure is returned unchanged except for the first member that is set to `F_UNLCK`.
- example: `file-locking/fcntl-locking.c`
- example on how to use fcntl (it is ,,fixed" version of previous `race/race.c` example from page 137): `race/fcntl-fixed-race.c`.
- locking via `fcntl` and `lockf` has one important property that is being described in the `fcntl` man page in FreeBSD:

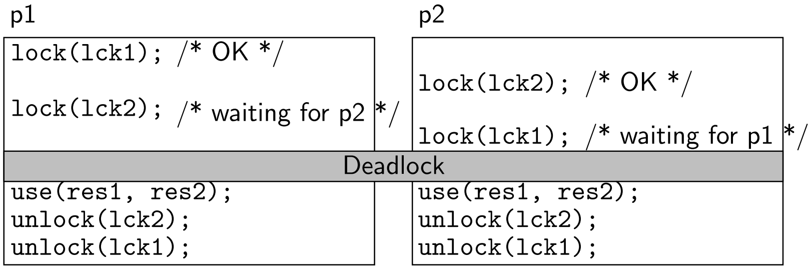
  This interface follows the completely stupid semantics of System V and IEEE Std 1003.1-1988 ("POSIX.1") that require that all locks associated with a file for a given process are removed when any file descriptor for that file is closed *by that process. This semantic means that applications must be aware of any files that a subroutine library may access. For example if an application for updating the password file locks the password file database while making the update, and then calls `getpwnam(3)` to retrieve a record, the lock will be lost because `getpwnam(3)` opens, reads, and closes the password database.*

- The `lockf` function (SUSv3) is simpler variant of `fcntl`, specifies only how to lock and how many bytes from the current position in the file. Very often implemented as `fcntl` wrapper. Beware: one cannot assume it is implemented around `fcntl` therefore avoid using these both together.

- The example `file-locking/lockf.c` demonstrates how mandatory locking works and the use of the `lockf` function.

Deadlock

- 2 shared resources `res1` and `res2` protected by locks `lck1` and `lck2`. Processes `p1` and `p2` want to have exclusive access to both resources.



p1

```
lock(lck1); /* OK */

lock(lck2); /* waiting for p2 */
```

p2

```
lock(lck2); /* OK */

lock(lck1); /* waiting for p1 */
```

Deadlock

p1

```
use(res1, res2);
unlock(lck2);
unlock(lck1);
```

p2

```
use(res1, res2);
unlock(lck2);
unlock(lck1);
```

- **watch out for the locking order !**

- Generally deadlock happens if process is waiting for an event that cannot happen. Here e.g. 2 processes are waiting for each other, for the other to release the lock however that will never happen. Next possibility is deadlock of single process that is reading from a pipe after forgetting to close the write end of the pipe. If there is no one else that has the pipe open, the read will block because the all file descriptor copies of the write end are not closed and therefore the end of file cannot happen until the reading process closed the write end however it cannot do it because it is blocked. in `read` syscall:

```
int main(void)
{
        int c, fd;
        mkfifo("test", 0666);
        fd = open("test", O_RDWR);
        read(fd, &c, sizeof(c));
        /* never reached */
        return (0);
}

$ ./a.out
^C
```

- `fcntl()` checks for deadlock and returns `EDEADLK`.
- It is best to avoid deadlock by correct programming and do not rely on the system.

IPC

- **IPC** stands for *Inter-Process Communication*
- between processes **in single system**, e.g. does not include network communication.
- **semaphores** ... used for process synchronization
- **shared memory** ... passing data between processes, brings similar problems as shared files, the solution is to use semaphores.
- **message queues** ... provide both communication and synchronization (waiting for message to arrive)
- IPC means have **access rights** (for reading/writing) similarly to files for owner, group and others.

- IPC resources continue to exist even after the process that created them is no longer around. To destroy them it is necessary to explicitly request this. From the shell the list of IPS resources can be acquired using the `ipcs` command. They can be deleted using the `ipcrm` command. The state and contents of existing IPS resources is unchanged even if no process works with them at the moment.

Semaphores

- introduced by E. Dijkstra
- semaphore is data structure that contains:
  - non-negative integer `i` (free capacity)
  - process queue `q`, that wait for free capacity
- semaphore operations:

init(s, n)

```
empty s.q; s.i = n
```

P(s)

```
if(s.i > 0) s.i-- else
put calling process to sleep and add it to s.q
```

V(s)

```
if(s.q empty) s.i++ else
remove one process from s.q and wake it up
```

- **P** is from dutch ,,proberen te verlagen" – try to decrement, **V** from ,,verhogen" – increment.
- The `P(s)` and `V(s)` operations can be made generic: the semaphore value is possible to change with any integer `n` … `P(s, n)`, `V(s, n)`.
- Allen B. Downey: *The Little Book of Semaphores*, Second Edition, on http://greenteapress.com/semaphores/
- *binary semaphore* has only values 0 or 1

Mutual exclusion with semaphores

- one process initializes the semaphore

```
sem s;
init(s, 1);
```

- critical section is augmented with semaphore operations

```
...
P(s);
critical section;
V(s);
...
```

- semaphore initialized to `n` value allows `n` processes to enter the critical section simultaneously. Here semaphore works like a lock. The same process is unlocking (incrementing the value) and locking (decrementing the value).
- In general, the increment operation can be done by different process than the one which performed the decrement operation (and vice versa). The mutexes work differently, see page 196.

POSIX API for semaphores

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

- creates or opens a new POSIX semaphore. *mode* is same as for **open**(). Use “`/somename`” for the *name*.

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

- decrement *`sem`*, if currently 0, the call will block

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

- increment *`sem`*. If *`sem`* consequently becomes greater than 0, another thread blocked in **sem_wait**() will be woken up.

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

- close *`sem`*, free its resources allocated to **this** process.

- *`oflag`* can be a combination of only `O_CREAT` and `O_EXCL`. If the semaphore already exists, an invocation with only `O_CREAT` silently succeeds, `O_EXCL` will cause an error.

- As soon as the semaphore is created, other processes can use it as well, based on *mode*.

- If the semaphore is not removed via **sem_unlink**(), it will exist (in kernel) until the system is shut down.

- There are also memory-based semaphores that do not use a name and are not identified by the returned handle – and thus use less resources. See **sem_init**() and **sem_destroy**() for more information.

- Example on POSIX semaphores (it is a fixed version of the previous example `race/race.c` from page 137): `race/posix-sem-fixed-race.c`.

Other IPC facilities

- POSIX and SUSv4 define other facilities for interprocess communication we have not mentioned yet:
  - **POSIX shared memory** accessible via **shm_open**()
  - **POSIX queues** … **mq_open**(), **mq_send**(), **mq_receive**(), …
  - **System V APIs** for queues, shared memory, and semaphores
- **sockets** come from the BSD world and allow communication in domains `AF_UNIX` (communication within the same system) and `AF_INET` (communication within the same system or across network).

- POSIX IPC API is much simpler and better designed than the System V IPC API. However, it is also much younger so you may see the System V API used a lot in existing code. For System V semaphore API and examples, see page 230.
- The POSIX API for IPC came in with extension 1003.1b (aka POSIX.4), see page 9.
- Sockets were accepted by other Unix systems as well and has been a part of the UNIX specification since version 2.
- There are other more system specific facilities. For example, *doors* is an RPC like facility on Solaris, designed to be used within the same system only. It has some interesting features - not only it can be used for interprocess communication however it can be used in kernel to request a service from a userland program.

Contents

- Introduction, Unix and C, programming tools
- Basic Unix concepts and conventions, its API
- Access rights, devices
- Process manipulation, program execution
- Signals
- Process synchronization and interprocess communication
- **Network programming**
- Programming with threads
- Appendix

Network communication

UUCP (UNIX-to-UNIX Copy Program) – first application for communication between UNIX systems connected directly or via modems, in Version 7 UNIX (1978)

sockets – introduced in 4.1 BSD (1982); socket is one end of a bidirectional communication channel created between two processes either on the same computer or across a network.

TLI (Transport Layer Interface) – SVR3 (1987); API providing network communication within the 4th layer of ISO OSI. Counterpart to the BSD sockets API.

RPC (Remote Procedure Call) – SunOS (1984); provides access to services running on a remote machine, data transferred in XDR format (External Data Representation)

- There are two main conceptual models for network communication. ISO (International Standards Organization) OSI (Open Systems Interconnect), and the Internet protocol suite (also called TCP/IP). Each one defines several layers. If we oversimplify, ISO OSI is very formal, quite complex, has 7 layers, and its specification is not freely available, while TCP/IP is simpler, not that formal, with 4 layers, and the specification is freely available via RFCs. While the layers of the two models cannot be precisely mapped onto each other, the corresponding layers are roughly as follows:

ISO OSI	TCP/IP
application presentation session	application
transport	transport
network	internet
link physical	link

  We will be working exclusively with the TCP/IP model.

- UUCP is a historical thing, fully implemented in userland without any kernel support. See the `uucp` man page or Wikipedia for more information.

- RPC is implemented as a library linked to applications, uses sockets and works on top of TCP and UDP. RPC was developed as a communication protocol for the *Network Filesystem* (NFS). There are several mutually incompatible RPC implementations.

- TLI is designed from an OSI model-oriented viewpoint, and it corresponds to the 4th layer – transport. TLI API looks similar to sockets.

- Sockets for communication within the same host are in the `AF_UNIX` domain and their names correspond to special files that represent the sockets in the filesystem. `ls -F` uses the equal sign "=" to mark a Unix domain socket.

- Sockets in `AF_UNIX` are different from local TCP/IP communication over the loopback interface `localhost` (`127.0.0.1`). See page 155 for more information on `AF_UNIX`.

TCP/IP basics

- protocols
  - **IP (Internet Protocol)** – principal communications protocol, not accessible for a non-privileged user
  - **TCP (Transmission Control Protocol)** – reliable, ordered, and error-checked delivery of a stream of bytes
  - **UDP (User Datagram Protocol)** – datagram, connection-less, unreliable
- **IP address** – 4 bytes (IPv4) / 16 bytes (IPv6), defines a network interface, not a computer
- **port** – 2 bytes, application end-points on a host
- **DNS (Domain Name System)** – translates domain names to the numerical IP addresses

- Unix mostly uses protocols from the TCP/IP family. We will cover TCP and UDP. In both protocols, one end of a communication channel is identified by an IP address and a port. Those two pieces correspond to a socket. A TCP connection is uniquely identified by a pair of sockets.

- Ports below 1024 are reserved and additional privileges are needed to use them. For example, *root* can access those. See `/etc/services` for the textual database of port numbers and corresponding service names.

- To learn about networking and the Internet in general, we very much recommend networking lectures by Jiří Peterka. You can either attend them at MFF UK or you can find those online.

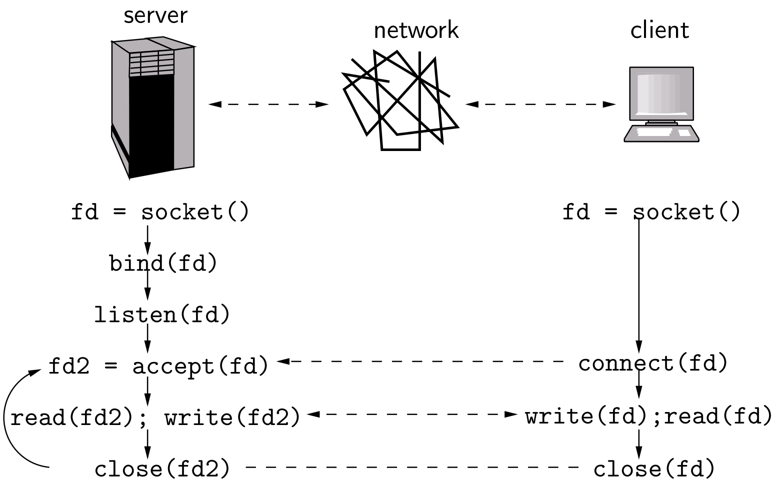
- *IP* – protocol of the internet layer within the TCP/IP network stack, provides data transfer between two interfaces identified by an IP address. It is unreliable. Provides routing and fragmentation. Defined in RFC 791. The Internet Control Message Protocol (ICMP), defined in RFC 792, is an inherent part of the IP protocol.

- *UDP* – a simple extension of the IP protocol, only adds protocol numbers. Still unreliable and datagram oriented. Defined in RFC 768.

- *TCP* – establishes connections between two points (sockets). Provides a continuous stream of data, congestion control and reliable delivery. To create a connection, a so called *handshake* must be performed. The protocol is defined in RFC 793 and other follow-up RFCs.

- *DNS* – hierarchically organized database, its structure does not have to follow the IP address structure.

Connection-oriented (TCP), sequential service



fd = socket()
bind(fd)
listen(fd)
fd2 = accept(fd)
read(fd2); write(fd2)
close(fd2)

fd = socket()
connect(fd)
write(fd);read(fd)
close(fd)

- Note that common **read**() a **write**() calls are used. To keep the picture readable, all arguments aside from a descriptor *fd* were intentionally omitted.
- The server creates one connection and does not accept a new one until the previous connection finished. That is why it is called a sequential service.
- System calls used:
  - **socket**() – creates a socket, returns its descriptor.
  - **bind**() – binds an IP address and a port number with the socket. In other words, it assigns a name to an unnamed socket. The address must be either one of IP addresses assigned to one of the host network interfaces (the host where the socket was created), in which case it will only accept connection requests over that specific interface via that specific IP address, or it can be a special value `INADDR_ANY` (for so called *wildcard sockets*, denoting the connection will be accepted on any IP address on any interface of the host.
  - **listen**() – tells the kernel to start listening on connection requests on the socket.
  - **accept**() – blocks the process until there is a connection request, then creates the connection and returns a **new** descriptor which is used for communicating with the client. The original socket descriptor can be used for another **accept**() call to serve a new connection request from another client.

- **close**() – closes the connection.
- **connect**() – the client asks to create a connection. The IP address and a port number are passed as arguments (omitted in the picture), the communication is performed through an already existing socket descriptor *fd*. In contrast to **accept**(), a new file descriptor is not created.

Creating a socket: `socket()`

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

- creates a socket, returns its descriptor
- *domain* – „where the communication will take place":
  - `AF_UNIX` ... local communication within a host, its address is a file name. Also `AF_LOCAL`.
  - `AF_INET`, `AF_INET6` ... internet communication, the address is an IP address and port pair
- *type*:
  - `SOCK_STREAM` ... connection-oriented reliable service, provides bidirectional data stream
  - `SOCK_DGRAM` ... connection-less unreliable service, transmits datagram
- *protocol*: `0` (default for a given *type*) or a valid protocol number (e.g. `6` = TCP, `17` = UDP)

- Function is declared in `<sys/socket.h>` as well as other network related functions from the previous slide.
- Sockets use the same name space as file descriptors, i.e. the same descriptor table. If you write a simple program that only calls **socket**(), it will return 3 as that will be the first available slot in the descriptor table.
- Connection-oriented communication is always bidirectional and is similar to pipes. However, note that pipes may not be bidirectional, see page 113.
- Sometimes you can see constants beginning with `PF_` (meaning *protocol family*, e.g. `PF_INET`, `PF_UNIX`, or `PF_INET6`) and used in a **socket**() call. Constants `AF_` (*address family*) are then used only for naming the sockets. While it might make a better sense, the UNIX specification only defines `AF_` constants. And if `PF_` constants exist on a system, they are defined via corresponding `AF_` constants. We do recommend to use only `AF_` constants.
- There are other domains, see the `socket(2)` manual page.
- There are also other socket types, for example `SOCK_RAW`, for full protocol access. In order to use `SOCK_RAW` you usually need additional privileges. That

is a reason why a command `ping`, which works with ICMP headers of sent packets, might need an SUID privilege:

```
$ ls -l /usr/sbin/ping
-r-sr-xr-x   1 root  bin  55680 Nov 14 19:01 /usr/sbin/ping
```

Naming the socket: `bind()`

```
int bind(int socket, const struct sockaddr *address,
         socklen_t address_len);
```

- binds a *socket* with an address
- `struct sockaddr` is universal and not used to fill out the address
  - `sa_family_t` *`sa_family`* ... domain
  - `char` *`sa_data`*`[]` ... address
- for `AF_INET`, `struct sockaddr_in` is used:
  - `sa_family_t` *`sin_family`* ... domain (`AF_INET`)
  - `in_port_t` *`sin_port`* ... port number (16 bits)
  - `struct in_addr` *`sin_addr`* ... IP address (32 bits)
  - `unsigned char` *`sin_zero`*`[8]` ...padding

- **bind**() assigns a socket its source address for packets being sent to the other side of the connection which is also the destination address for data received. The remote address is set via **connect**().
- Structure `sockaddr` is a universal type used by a kernel. For setting addresses based on a specific domain one has to use concrete structures per domain, see the next slide. Those specific structures need to be casted to the universal structure as that is the type required by the **bind**() function, for example. However, it is not recommended to use those structures as the program will only work for one specific address family – we will do it here to show you how it works though. However, you can use helper functions that convert names to addresses with no need to work those structures directly. See **getaddrinfo**() on page 171 for more information.
- You will also need other header files, see the example on page 157.
- For domains `AF_INET` and `AF_INET6`, you can use a special address that stands for any address on a given host. Such a socket can be used to accept connections on any IP address assigned to any interface on the host. That address is:

  - For AF_INET, use INADDR_ANY (4 zero bytes corresponding to 0.0.0.0)
  - With AF_INET6 the situation is more complicated. You can either use a constant variable in6addr_any or a constant IN6ADDR_ANY_INIT. However, the constant can be only used to initialize variables of type struct in6_addr, not for any assignment. Both values correspond to :: (16 zero bytes).

- You cannot bind the same address to multiple sockets.

- If **bind**() is not called, the kernel will assign one of available ports and the primary IP address on the interface through which the destination can be accessed. In general, as a client, you do not need a specific outgoing port so **bind**() is usually not needed and the call is typically used only by servers as they need a specific port number to be contacted (e.g. 443 for HTTPS). Note that some legacy services though, e.g. rsh, require the client to connect from a privileged port (0-1023). Such a client must call **bind**() to use such a port and also have enough privileges to do that.

- **The address and port must always use network byte ordering.** See page 12 where different byte orderings were explained. More information is on page 170.

Structure for IPv4 addresses

- each address family has its structure and a header file
- the structure is in **bind**() type-casted to struct sockaddr

```
#include <netinet/in.h>
struct sockaddr_in in = { 0 }; /* IPv4 */

in.sin_family = AF_INET;
in.sin_port = htons(2222);
in.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(s, (struct sockaddr *)&in, sizeof (in)) == -1) ..
```

- As already mentioned, such code is for demonstration purposes only. Unless really needed, you should not write code like this but rather use generic functions for converting names to addresses in which case you do not need to worry about address families. Most networking applications are assumed to work with both IPv4 and IPv6.

- `sin_addr` is a structure itself, of type `in_addr`. That structure must have at least one member, `s_addr` whose type must be equivalent to a 4 byte integer. That comes directly for the UNIX specification for `netinet/in.h`.
- Working with `sockaddr_in6` is a bit more complicated. It is in `netinet/in.h`, either defined in there directly or the file includes `netinet6/in6.h` with the structure definition.
- We repeat again as it is important – for both `AF_INET` and `AF_INET6`, the port and address must be in the network byte ordering, see page 170.
- As `INADDR_ANY` is defined as `0`, you may see its use without **htonl**() (see page 170 for more information on the function). Never do it that way. Next time you put an IP address there, you may forget to add **htonl**() and you are going to run into some issues right away. And again, if you write code agnostic to specific address families, you do not need to worry about network byte ordering for addresses and ports whatsoever.
- In the `AF_UNIX` domain, `struct sockaddr_un` is used, defined in `<sys/un.h>`:
  - `sa_family_t` *`sun_family`* ... domain
  - `char` *`sun_path`*`[]` ... socket name
  - The size of *sun_path* has intentionally been left undefined in the UNIX specification. This is because different implementations use different sizes. For example, 4.3 BSD uses a size of 108, and 4.4 BSD uses a size of 104. Since most implementations originate from BSD versions, the size is typically in the range 92 to 108.

Waiting for connection: `listen()`

`int` **listen**`(int` *`socket`*`, int` *`backlog`*`);`

- specifies willingness to accept incoming connections on *socket*, and the system starts listening
- maximum of *backlog* connection requests may wait in the queue to be served
- connection requests that do not fit the queue are refused (**connect**() returns an error on the other side).
- the system waits for a connection on an address previously assigned by **bind**()

- The system may silently adjust *backlog* if it is not in a supported range.
- Wildcard sockets are primarily used for servers. If you need to distinguish between interfaces, you need a socket per an IP address. This used to be used for web servers that distinguished virtual servers based on the IP address. However, that is remote past. To distinguish between virtual servers running on the same host, a HTTP header „`Host:`” is used. Similarly, the TLS protocol uses `ServerName`.
- The fact that the system starts listening on a port means that a TCP handshake is performed and data is being accepted. The data is stored in a fixed length buffer and after it is filled out, the connection is still active but the TCP window is set to 0 which means the system stopped accepting further data. The buffer size is usually a few tens of kilobytes. Example: `tcp/up-to-listen-only.c`.
- The example code uses macro `SOMAXCONN`, required by the UNIX specification to be in `sys/socket.h`. It specifies the maximum queue length for **listen**(). As far as we know, Linux, FreeBSD, macOS and Solaris use value of 128.
- This function is specific to connection-oriented protocols, so it does not work with UDP.

Accepting connection: accept()

```
int accept(int socket, struct sockaddr *address,
           socklen_t *address_len);
```

- creates a connection between the local, already listening `socket` and a remote end
- returns a new socket descriptor that can be used to communicate with the remote process. Original socket can be used to accept another connection.
- returns a remote IP address/port in *address* unless NULL
- `*address_len` is a size of the *address* structure, is updated with its real size on return

- The “remote end” is the socket on which a **connect**() is called on a remote Unix host, or it could be possibly something else on other systems. Remember that protocols and APIs are two different things.

- The newly created socket has the same characteristics as *socket*. For example, if *socket* was non-blocking, the new socket is non-blocking as well.
- If more clients running on the same host connect to the same server (i.e. using the same IP address and port), individual connections are distinguished only by the client side port number. Do remember that a TCP connection is uniquely identified by two sockets, i.e. "((addr1, port1), (addr2, port2))."
- The *address* may be `NULL` which means the caller is not interested in the remote end address. In that case, ***`address_len`*** should be `NULL` as well.
- If the code is written to be independent of an address family, it should use `struct sockaddr_storage` for *address*. Any specific address structure is guaranteed to fit in `struct sockaddr_storage`, i.e. either `struct sockaddr_in` or `struct sockaddr_in6`. Also see page 175.
- Example: `tcp/tcp-sink-server.c`

Initiating a connection: `connect()`

```
int connect(int sock, struct sockaddr *address,
            socklen_t address_len);
```

- attempts to make a connection to a remote socket waiting on *address* (of ***`address_len`*** length)
- if *sock* was not bound before, the kernel will assign an available local address based on the chosen address family
- you should close *sock* on a connection failure

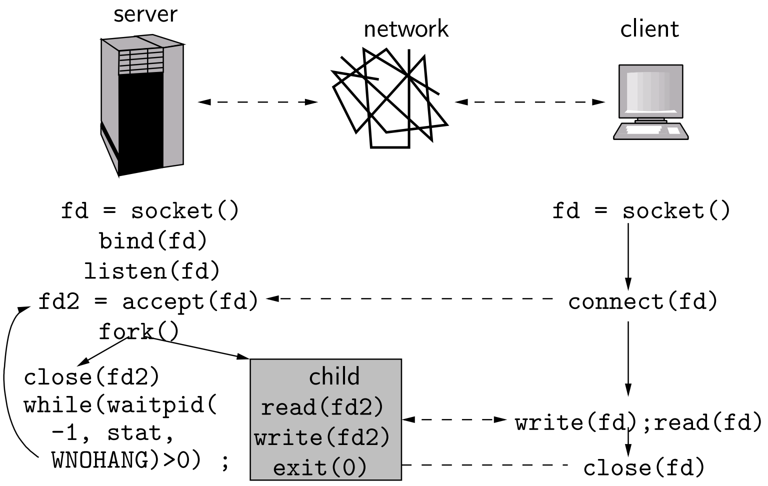
- As the UNIX specification does not say anything about the socket state after a connection failure, you should definitely close *sock* before moving on to create a new socket and trying again.
- After the connection is created, both the server and client can use normal **read**() and **write**() calls, or **send**(), **recv**, **sendmsg**(), and **recvmsg**(). Behavior of the functions is similar as if working with pipes.
- Example: `tcp/connect.c`

- For connection-less services (UDP), **connect**() may be used as well. However, it will only set the remote address so that **send**() and **recv**() which do not have a remote address parameter can be used.

- We may also call **connect**() multiple times for a connection-less service in which case each call sets the remote address again. If we use `NULL` for the remote address, the present address is reset.

- If the socket is set as non-blocking, see page 72, **connect**() will not block while waiting to connect. It will return `-1` with `errno` set to `EINPROGRESS` (= "not possible to create a connection right away") and the connection request is stored in the system queue to be processed. Until the connection is ready, subsequent **connect**()s return `-1` with `errno` set to `EALREADY`. However, using this way to test the connection readiness is not the right approach as if the background connection attempt fails, the next **connect**() tries to create a new connection and we would end up in a never ending loop. The right approach is to use **select**() or **poll**(), see pages 176 and 179. You can also find there an example using a non-blocking **connect**(), page 177.

Connection-oriented (TCP), parallel service



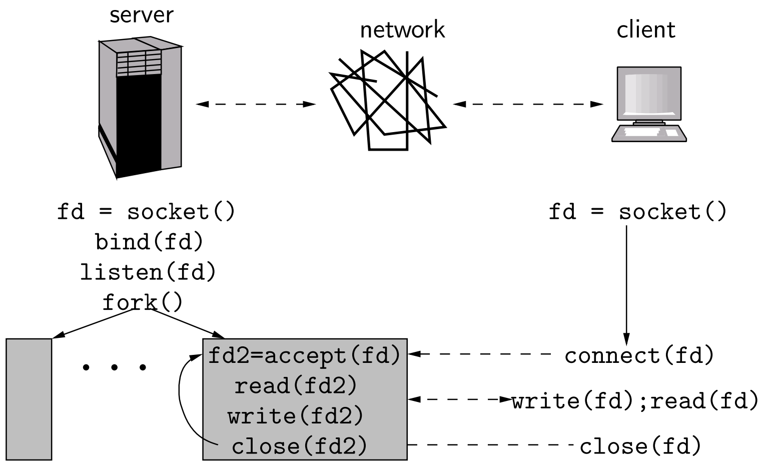
```
fd = socket()                          fd = socket()
   bind(fd)
  listen(fd)
fd2 = accept(fd) <- - - - - - - - - -  connect(fd)
   fork()
close(fd2)          read(fd2)  <- - -> write(fd);read(fd)
while(waitpid(      write(fd2)
  -1, stat,         exit(0)  - - - - -  close(fd)
  WNOHANG)>0) ;
```

- For each client connection the server accepts, a new process is created to process it. After the connection is finished, the child exits. The parent process can accept new connections meanwhile. So, multiple connections may be served in parallel.

- After **fork**()ing but before starting to use the connection, the child may **exec**() – that is how `inetd` works, see page **??**.

- As you already know, by calling **waitpid**() you are getting rid of zombies. This only works because the `WNOHANG` flag is used, otherwise the parent would be blocked and the next connection would only be accepted after one of the children exited. Another way is to set to ignore `SIGCHLD` in which case you completely avoid the grave danger of the living dead attack (see page 132). You could also catch the signal and call one of the **wait**() calls from the handler itself which is fine as the call is async-signal-safe (see page 131).

Connection-oriented service, parallel `accept()`



```
fd = socket()
  bind(fd)
 listen(fd)
   fork()

fd2=accept(fd)
  read(fd2)
 write(fd2)
 close(fd2)
```

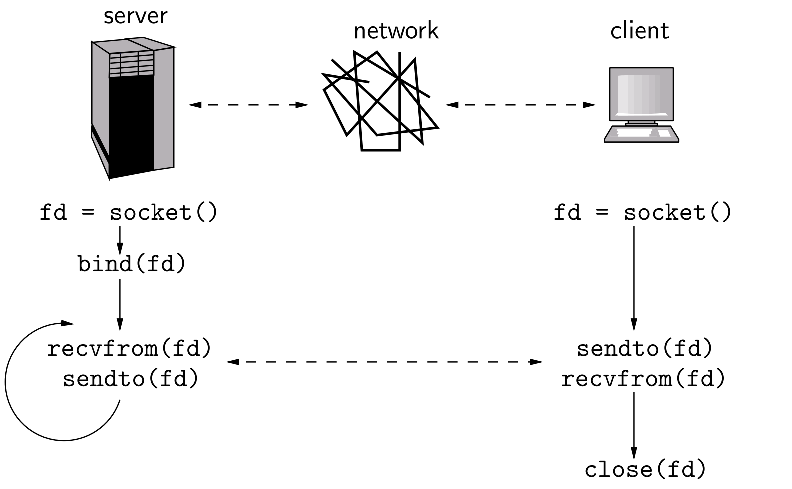
```
fd = socket()
connect(fd)
write(fd);read(fd)
close(fd)
```

- After calling **bind**() and **listen**(), the parent creates several children who sequentially serve connection requests. Kernel, from the user point of view in non-deterministic fashion, distributes the connection requests between child processes waiting in **accept**(). The parent itself does not serve any connection but possibly **wait**()s and creates new processes as needed.

- The parent should monitor the number of existing children serving connections and spawn new processes as necessary. It is a good idea for child processes to voluntarily exit after serving a certain number of connections to avoid issues like memory leaks to affect the host as a whole. Apache web server can be configured to work like this.

- All server side ways of serving the connections work with the same client. The way a client works does not depend on which approach the server chooses.

Datagram services (UDP)



fd = socket()

bind(fd)

recvfrom(fd)
sendto(fd)

fd = socket()

sendto(fd)
recvfrom(fd)

close(fd)

- Note that there is no **listen** call.
- Both the client and server use the same functions, the client is one that sends the first datagram.
- As in TCP, a client does not need **bind**() unless it requires a specific local address to bind to. The server gets the client address from the received datagram.
- In contrast to connection-oriented service, the connection-less one has less overhead and one can use the same socket to communicate with multiple remote processes.
- You can use **connect**() for UDP as well, see page 161 for more information.

Receiving message: `recvfrom()`

```
ssize_t recvfrom(int sock, void *buf, size_t len,
                 int flg, struct sockaddr *address,
                 socklen_t *address_len);
```

- receives a message from `sock`, stores it to `buf` of size `len`, puts the sender's address to *address*, and address length to `address_len`. Returns message length. If the message does not fit *len*, extra data is discarded (`SOCK_STREAM` does not divide data, nothing is discarded).
- flags `flg` can be:
  - `MSG_PEEK` ... message it considered not read, next `recvfrom` will return it again
  - `MSG_OOB` ... reads urgent (out-of-band) data
  - `MSG_WAITALL` ... waits for the buffer to fill up i.e. *len* bytes

- Mainly for `SOCK_DGRAM` sockets. Waits for the whole message, does not return datagram portion. Again, it is possible to set the socket as non-blocking.

- There does not seem to be a portable way how to get the size of the UDP datagram before reading it out from kernel buffer. On Linux the `MSG_TRUNC|MSG_PEEK` flags can be used for `recvfrom` to get the size. On other systems the generic `recvmsg` syscall can be used to at least provide awareness of the truncation (`MSG_TRUNC` flag in the `msghdr` structure). Depending on the application, using large buffer (based on lower network layer constraints) might be the answer (at the cost of wasting memory).

- `address_len` **must** be initialized with buffer size if the address is not `NULL`. `NULL` address is used to express that the caller is not interested in remote's address – however that is usually not the case when working with datagrams.

- Instead of using `recvfrom` it is possible to use `recvmsg` which is more generic.

- If `connect` was used then `recv` can be used instead of `recvfrom`.

- After successful return from `recvfrom` it is possible to reuse `address` and `address_len` unchanged for a `sendto` call.

- Like `sendto`, `recvfrom` is possible to use for connected service. That said, getting remote's address is better via `getpeername`, see page 169.

- Example: `udp/udp-server.c`

Sending message: sendto()

```
ssize_t sendto(int socket, void *msg, size_t len,
               int flags, struct sockaddr *addr,
               socklen_t addr_len);
```

- sends a message *msg* via *socket* of *len* bytes to address *addr* (of *addr_len* length).
- *flags* can carry:
  - MSG_EOB . . . finish a record (if supported by the protocol)
  - MSG_OOB . . . send urgent (out-of-band) data

- Used mainly for SOCK_DGRAM sockets, because in such situation we only have socket representing our side of the connection; see the note for accept. The remote address has to be specified which cannot be done with write. Moreover, for **datagram** service the data sent is considered as whole, i.e. either it is accepted completely or the call will block – partial write does not exist. Like with file descriptors, it is possible to set the socket as non-blocking, see page 72.
- Instead of using sendto more generic function sendmsg can be used.
- If connect was used then send can be used instead of sendto.
- Successful return from either **does not mean successful delivery of the message to the remote side**, but only insertion of the data to local buffer which is yet to be sent out.
- It is possible to use sendto for stream service, however the address will be ignored. The only reason not to use write would be to use flags. In this case it is simpler to use send.
- Example: udp/udp-client.c.

Closing socket: `close()`

`int` **close**`(int` *`sock`*`);`

- close descriptor *sock*, after the last descriptor is closed, close the socket in kernel
- for the `SOCK_STREAM` socket, `SO_LINGER` flag is important (default is `.l_onoff == 0`, use **setsockopt**() with `struct linger` to change that).
  - `.l_onoff == 0` … **close**() returns but system tries to transfer rest of the data
  - `.l_onoff == 1 && .l_linger != 0` … system tries to transfer data until timeout `l_linger` expires (in seconds), if it fails, return error, otherwise return OK after transferring data.
  - `.l_onoff == 1 && .l_linger == 0` … reset the connection

- Once closed, TCP socket can remain in transitory state which is defined in TCP protocol for connection closing. Before the socket is completely destroyed, it is not possible to use another socket with the same port, unless this behavior was overridden with the `SO_REUSEADDR` flag using the `setsockopt` function, see page 169.
- Connection reset is abnormal connection termination. In case of TCP a packet with `RST` flag is used for such termination. The remote side will detect this as end of file when reading, the reset will lead to the `ECONNRESET` error. Example: `tcp/linger.c`.

Shut down part of a connection: `shutdown()`

int **shutdown**(int *socket*, int *how*);

- shuts down a socket but does not close the descriptor, *how* can be:
  - `SHUT_RD` ... shut it down for reading
  - `SHUT_WR` ... for writing
  - `SHUT_RDWR` ... for both

- After using `shutdown` it is still necessary to close the descriptor using `close`.
- Normal TCP connection termination each side will signal that no subsequent writes will follow. This is valid for either `close` or `shutdown(fd, SHUT_RDWR)`. When using `shutdown(fd, SHUT_WR)` it is still possible to read from the socket. The remote side will get `EOF` while reading however it can still write.

Working with IPv4 and IPv6 addresses

- binary representation of IP address is hard to read
- string representation of IP address cannot be used when working with `sockaddr` structures

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

- converts string to binary representation, i.e. something usable for `in_addr` or `in6_addr` members of `sockaddr` structures
- returns 1 (OK), 0 (wrong address) or -1 (and sets `errno`)

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

- counterpart to `inet_pton`; returns *`dst`* or `NULL` (and sets `errno`)
- for both functions `af` is either `AF_INET` or `AF_INET6`

- The functions are declared in `arpa/inet.h`.
- `inet_pton` returns 1 if the conversion successfully happened, 0 if given string is not an address or -1 if *`af`* is not supported (`EAFNOSUPPORT`). `inet_ntop` returns `dst` if everything is OK otherwise returns `NULL` with `errno` set.
- Addresses and ports in binary form are stored as big endian.
- `dst` has to be sufficiently sized because there is no parameter specifying the size. This is not a problem since according to the value of `af` appropriate address structure or character array can be passed in. For maximal lengths of strings for addresses, 2 macros can be used – `INET_ADDRSTRLEN` (16) or `INET6_ADDRSTRLEN` (48). These values contain space for terminating `\0`.
- `size` is string size of `dst`. If not sufficient, the call will fail and `ENOSPC` will be set.
- `n` stands for `network`, `p` stands for `presentable`
- In the past `inet_aton` and `inet_ntoa` (`a` as `ascii`) were used for IPv4 addresses. Thanks for the functions above these are now legacy. All these calls are usually documented in the `inet` man page.
- Do realize that using these functions it is only possible to convert one address family once, either IPv4 or IPv6. When the output can be either, try one and if that fails, fallback to another. Example: `resolving/addresses.c`.

More socket functions

```
int setsockopt(int socket, int level, int opt_name,
               const void *opt_value, socklen_t option_len);
```

- sets socket parameters

```
int getsockopt(int socket, int level, int opt_name,
               void *opt_value, socklen_t *option_len);
```

- reads socket parameters

```
int getsockname(int socket, struct sockaddr *address,
                socklen_t *address_len);
```

- get local socket address

```
int getpeername(int socket, struct sockaddr *address,
                socklen_t *address_len);
```

- get address of the remote end

- The `level` value for `getsockopt` and `setsockopt` is usually `SOL_SOCKET`. For `getsockopt`, the `option_len` value **must** be initialized to the size of `opt_value`.

- `getsockname` is used when not using `bind` syscall to determine what is the (local !) address assigned to the socket by the kernel.

- `getsockopt(sock, SOL_SOCKET, SO_ERROR, &val, &len)` returns (and erases) error indication for given socket. This is mainly useful to get result of non-blocking `connect`, see page 160.

- When using `SO_REUSEADDR` it is possible to immediately create new server (i.e. to successfully call `socket`, `bind`, `listen` and `accept`) listening on address and port previously used even though there are still TCP connections in their final stage from the previous instance of the server:

  ```
  int opt = 1;
  setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
  ```

- See `SO_REUSEADDR` being used in example `tcp/reuseaddr.c`. For the demonstration it is necessary to establish at least once connection, otherwise there is nothing to wait for and repeated `bind` will succeed anyway.

Byte order

- network services can use byte ordering that differs from the native one on the system. Conversion functions (macros):
  - `uint32_t` **htonl**`(uint32_t` *`hostlong`*`);`
host → network, 32 bits
  - `uint16_t` **htons**`(uint16_t` *`hostshort`*`);`
host → network, 16 bits
  - `uint32_t` **ntohl**`(uint32_t` *`netlong`*`);`
network → host, 32 bits
  - `uint16_t` **ntohs**`(uint16_t` *`netshort`*`);`
network → host, 16 bits
- network byte order is big-endian, i.e. most significant byte first. Both network addresses and port numbers are multibyte values.

- If the local system uses the network byte order natively, the functions become no-ops.
- Simple and sufficient test is to run your program in 2 instances against each other (if possible) on two systems with different endianess.

Protocol and port numbers

```
struct protoent *getprotobyname(const char *name);
```

- returns protocol number in `p_proto` with *name* (e.g. for `"tcp"` returns 6).
- protocol numbers are stored in the `/etc/protocols` file.

```
struct servent *getservbyname(const char *name,
                              const char *proto);
```

- for service name `name` and protocol name `proto` returns port number in `s_port`.
- port numbers are stored in the `/etc/services` file.

The functions return `NULL`, if matching entry cannot be find in the database.

- The result of **getprotobyname** is handy for calling `socket`, the result of **getservbyname** is for `bind`.
- Next to `getservbyname` there is also `getservbyport` that finds service entry according to port number (in network byte order !) and function `getservent` and others. for entry traversal.
- All these functions search only ”official” lists of services and protocols, which are located in the files mentioned on the slide.
- These files define the mapping for names and numbers for standard protocols and services.
- Keep in mind that protocol is the upper layer protocol as specified in the IP packet header, (e.g. TCP, UDP, OSPF, GRE etc., see pages 11 and 14 in RFC 791), not HTTP, SSH, telnet or FTP – these are *services*, represented by port numbers.
- example: `resolving/getbyname.c`

Convert hostname to addresses: getaddrinfo()

- generates sockaddr structures from given parameters
- works with both IP addresses and ports
- can influence the behavior with *hints*

```
int getaddrinfo(const char *nodename,
                const char *servicename,
                const struct addrinfo *hint,
                struct addrinfo **res);
```

- addrinfo structure members: ai_flags (for hints), ai_family (address family), ai_socktype, ai_protocol, ai_addrlen, struct sockaddr *ai_addr (resulting addresses), char *ai_canonname, struct addrinfo *ai_next (next item in the list)

- When evaluating queries for addresses and names, the naming services are consulted according to the specification in the /etc/nsswitch.conf file.
- The structure addrinfo is defined in netdb.h.
- getaddrinfo is able to convert a string containing either a hostname or an IP address. Similarly for ports. So, whether you use a hostname or an IP address as an command line argument, you can use the same code to process that.
- The getaddrinfo function returns the results in the *res* parameter. The result is a list of sockaddr structures corresponding to given input.
- There is a difference whether the address structures are to be used for a server or client; e.g. for a server it is sufficient to specify *nodename* as NULL (meaning any address). Similarly for the member ai_flags of the addrinfo structure used for the *hint* parameter – for a server, use AI_PASSIVE to indicate the returned address structures will be used in **bind**.
- After the results are no longer needed, it is prudent to call freeaddrinfo, that will free the memory allocated by getaddrinfo.
- example: resolving/getaddrinfo.c
- In the past the functions **gethostbyname** and **gethostbyaddr** were used. These only work with IPv4 addresses and are therefore considered legacy and not recommended to use. The functions **getipnodebyname** and **getipnodebyaddr** can be used instead (**getipnode*** are found in many systems,

nevertheless they were removed from GNU `libc`, so cannot be relied on universally). What is more, they are not part of UNIX standards. **getaddrinfo** and **getnameinfo** are part of the POSIX.1-2001 standard.

- It is not strictly necessary to convert old IPv4 specific code to the generic APIs. However when writing new code, always use them even when you think the program will only use IPv4.
- IPv6 sockets may be used for both IPv4 and IPv6 communications. That means if you try to bind addresses returned by **getaddrinfo**, **bind** should return `EADDRINUSE` after its first call if you use both IPv4 and IPv6 addresses. Binding on an IPv6 address should be sufficient to accept connections over IPv4 as well. Details are in RFC 3493. If you want to restrict a socket to IPv6 only, use the following code on a socket from the `AF_INET6` family:

```
int on = 1;

if (setsockopt(s, IPPROTO_IPV6, IPV6_V6ONLY, (char *)&on,
    sizeof(on)) == -1) {
        /* deal with the error here */
}
```

Convert address to hostname: `getnameinfo()`

- counterpart to `getaddrinfo`
- gets `sockaddr` structures, therefore works with both IPv4 and IPv6 addresses, as opposed to the `gethostbyaddr` function.

```
int getnameinfo(const struct sockaddr *sa,
                socklen_t *sa_len,
                char *nodename,
                socketlen_t *nodelen,
                char *servicename,
                socketlen_t *servicelen,
                unsigned flags);
```

- `getnameinfo` converts IP address and port number from the `sockaddr` structure to strings according to configured name services (`/etc/nsswitch.conf`) and the `flags` value. It performs the actions that would have to be otherwise done by `gethostbyaddr` and `getservbyport`.

- Either `nodename` or `portname` can be `NULL` – that means the caller is not interested in its resolution.
- Unlike the above mentioned legacy functions `getnameinfo` is reentrant and is therefore safe to use with threads.
- The man page for `getnameinfo` contains the list of usable `NI_` flags for the *`flags`* parameter.
- example: `resolving/getnameinfo.c`

Example: TCP server

```
struct sockaddr_storage ca; int nclients = 10, fd, nfd;
struct addrinfo *r, *rorig, hi;
memset(&hi, 0, sizeof (hi)); hi.ai_family = AF_UNSPEC;
hi.ai_socktype = SOCK_STREAM; hi.ai_flags = AI_PASSIVE;
getaddrinfo(NULL, portstr, &hi, &rorig);
for (r = rorig; r != NULL; r = r->ai_next) {
    fd = socket(r->ai_family, r->ai_socktype,
        r->ai_protocol);
    if (!bind(fd, r->ai_addr, r->ai_addrlen)) break;
}
freeaddrinfo(rorig); listen(fd, nclients);
for (;;) { sz = sizeof(ca);
    nfd = accept(fd, (struct sockaddr *)&ca, &sz);
    /* communication with the client */
    close(newsock);
}
```

- This is generic skeleton of TCP server. The `portstr` parameter is the only input value of the program. First successful `bind` will finish traversal of the IP address list.
- Program written in such a way will work on system with IPv6 and IPv4 addresses, i.e. it will be able to accept connections for both address families on the same socket. The remote address of IPv4 clients will be represented as *IPv4-mapped IPv6 address*, (i.e. it contains embedded IPv4 address, e.g. e.g. `::FFFF:78.128.192.1`).
- Note that it is mostly not necessary to cast to the `sockaddr` structure. The only exception is the `accept` call. For the `accept` call the `sockaddr_storage` is used which is generic ”container” able to store members of both `sockaddr_in` and `sockaddr_in6` structures.
- `socket` and `bind` are called for all returned `sockaddr` structures until `bind` succeeds for one of them or the list is not finished.

- The code on the slide does not check all return values and does not free all the allocated entities, e.g. socket closing in case of `bind` error. Also, it does not deal with the situation when the list of addresses is traversed without successful `bind` (this can be detected with `r` NULL check after the end of the first cycle).

Example: TCP client

```
int fd; struct addrinfo *r, *rorig, hi;
memset(&hi, 0, sizeof (hi)); hi.ai_family = AF_UNSPEC;
hi.ai_socktype = SOCK_STREAM;
getaddrinfo(hoststr, portstr, &hi, &r);
for (rorig = r; r != NULL; r = r->ai_next) {
    fd = socket(r->ai_family, r->ai_socktype,
        r->ai_protocol);
    if (connect(fd, (struct sockaddr *)r->ai_addr,
        r->ai_addrlen) == 0)
            break;
}
freeaddrinfo(resorig);
/* communication with server */
close(fd);
```

- This is generic TCP client skeleton. The `hoststr` and `portstr` arguments are the only input data of the program. First successful `connect` will terminate the address traversal for given host.

- In the example we let the kernel will choose local port. (there is no `bind` before the `connect`).

- `connect` is called sequentially on all returned `sockaddr` structures (embedded in the `addrinfo` structures) for given host until a connection succeeds or the list is exhausted.

- Same lack of error checking and freeing or resources as for the TCP server example, i.e. in situation when the list is exhausted without successful `connect`. Also the socket is not closed when `connect` is not successful.

Waiting for data: select()

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

- determines which out of given file descriptors are ready for reading, writing or which experienced exceptional state. If no such descriptor is found, waits till `timeout` expires. (`NULL` ... means arbitrary long wait). The `nfds` parameter sets the range of file descriptor values (`0, ..., nfds-1`).
- functions for setting file descriptor masks:
  - `void` **FD_ZERO**`(fd_set *fdset)` ... initialization
  - `void` **FD_SET**`(int fd, fd_set *fdset)` ... set
  - `void` **FD_CLR**`(int fd, fd_set *fdset)` ... unset
  - `int` **FD_ISSET**`(int fd, fd_set *fdset)` ... test

- **Motivation:** to read data from multiple file descriptors, it is possible to open respective files with the `O_NONBLOCK` flag (or set the flag after `open` with `fncnl` – with the second parameter being `O_SETFL`, not `O_SETFD`, see page 72). Repetitious `read` (which is non-blocking after we set the descriptor as non-blocking, see page 61) is then called on all descriptors. Sleeping in every loop is needed in order not to hammer the kernel with syscalls. For every invocation, you either read some data right away or get `-1` with `errno` set to `EAGAIN` if no data was ready, meaning the call would have blocked if it was not for the non-blocking descriptor. However, the disadvantages are: active waiting, frequent user-kernel mode switching, possible delay (up to the wait period between the iterations of the cycle). Correct solution of this situation is to use e.g. `select` and then call `read` only for those descriptors marked as *ready*.
- Busy waiting example: `select/busy-wait.c`. Notice that newly created socket does not have the `O_NONBLOCK` flag, see page 159, so it is necessary to set it.
- A descriptor marked as *ready* means that `read` or `write` on this descriptor with a cleared `O_NONBLOCK` flag would not block, therefore there has to be some data ready or some other condition is pending (`read` can e.g. return 0 for end-of-file or `-1` for error).
- The set `errorfds` is for exceptions depending on descriptor type; for socket it is the reception of urgent data (the `URG` flag in TCP header). It does not mean that there was an error on given descriptor. The error is determined by checking `errno` after given syscall returns `-1`.

- Before calling select the sets contain descriptors of interest, after the syscall returns the sets contain only those descriptors for which specified event happened. **It is therefore necessary to refresh the sets before next call to select.** Typically the sets are implemented as bit masks however it does not have to be so. The programmer does not have to deal with that. Traversing the sets is only possible one bit at a time and using FD_ISSET.

- select is handy also for the possibility to write into a pipe or socket – waiting for the other side to consume part of the buffer and thus free space for more incoming data.

- It is possible to use NULL instead of the descriptor set. Special case is to use NULL for all the sets in which case the syscall will block until the timeout expires or a signal is delivered.

- After returning it is necessary to test each descriptor individually, there is no call that would create a set of ready descriptors.

- If networking server handles multiple ports, it is possible to call select with respective socket descriptors and then accept for descriptors that select marked as ready, i.e. there is a client connection waiting (reading readiness).

- connect on non-blocking socket will return immediately, the actual connection establishment will be reported by select as readiness for writing. See page 160.

- Another possibility to use select is networking server that handles multiple clients in single process. Using select the client descriptors are tested and those which are ready are then used for communication. To allow new clients, the descriptor used for accept is tested as well.

- Beware that select **can** change the timeval structure; there is a call pselect, that will not change the structure (besides having other properties).

- It is possible to use FD_SETSIZE for the nfds parameter which is system constant for maximal file descriptor count. This is not the best solution since this constant is only 1024 on 32-bit systems however for 64-bit programs it can be significantly bigger (e.g. 65536 on Solaris), leading to more time spent in the syscall.

- If the time argument is set to 0 (not NULL), then select can be used for so called *polling* – it will check the current state and return immediately.

- Example: select/select.c

- select can also be used to determine the state after connect for non-blocking socket. Use getsockopt with opt_name set to SO_ERROR, (see page 169) to see if the connection was successful. Example: select/non-blocking-connect.c.

Example: `select()`

- The `fd` descriptor is a socket, converts socket communication to terminal and vice versa.

```
int sz; fd_set rfdset, efdset; char buf[BUFSZ];
for(;;) {
    FD_ZERO(&rfdset); FD_SET(0, &rfdset);
    FD_SET(fd, &rfdset); efdset = rfdset;
    select(fd+1, &rfdset, NULL, &efdset, NULL);
    if(FD_ISSET(0, &efdset))
        /* exception on stdin */ ;
    if(FD_ISSET(fd, &efdset))
        /* exception on fd */ ;
    if(FD_ISSET(0, &rfdset)) {
        sz = read(0, buf, BUFSZ); write(fd, buf, sz); }
    if(FD_ISSET(fd, &rfdset)) {
        sz = read(fd, buf, BUFSZ); write(1,buf,sz); }
}
```

- Here is typical use of `select`, when it is necessary to read the data from two sources. This example assumes that the descriptors `0` and `fd` are set as non-blocking.

- Better solution would be to use `select` also for testing writing readiness. For this two buffers are required – one for each direction of communication. Respective read descriptor will be present in the read set for `select` when the buffer is empty. The write descriptor will be in the write set when the buffer is non-empty.

- `select` will put the process to sleep when the data is not read from the remote side if only testing write readiness. This can be simply simulated by a program that merely connects (TCP handshake) however does not read anything. Example: `select/write-select.c`.

Waiting for data: `poll()`

`int` **poll**`(struct pollfd` *`fds`*`[], nfds_t` *`nfds`*`, int` *`timeout`*`);`

- waits for event on one of the descriptors in the `fds` array of `nfds` items for `timeout` ms (`0` ... returns immediately, `-1` ... waits arbitrarily long).
- `pollfd` structure members:
  - `fd` ... descriptor number
  - `events` ... events to wait for, OR-combination of `POLLIN` (can read), `POLLOUT` (can write), etc.
  - `revents` ... events that happened on the descriptor, same indications as in `events`, plus e.g. `POLLERR` (error happened)

- Alternative to `select`.
- The *timeout* argument is in milliseconds.
- There are many more flags, see the man page.
- Often either `select` or `poll` is implemented using the other one. On Solaris, for example, `poll` is a system call, and `select` is then a library call implemented as a `poll` wrapper.
- `poll` is necessary to use to test descriptor equal or higher than `FD_SETSIZE`.
- Unlike `select` it is not necessary to refresh descriptor sets after each call or nullify `revents`. To deselect a file descriptor, set the `fd` member of the `pollfd` structure to `-1` (or reshuffle the array which is significantly more complex).
- Time set to -1 has the same effect as `NULL` for `select`.
- If the number of descriptors is set to 0 (`fds` should be then `NULL`), `poll` can be simply used to sleep with less than one second granularity. Example: `sleep/poll-sleep.c` That said, this "trick" does not work on macOS.
  - Another way is to use `nanosleep`, that is part of POSIX.4 extension and therefore does not have to be always available. Example: `sleep/nanosleep.c`.
- While `select` and `poll` are pretty efficient when handling many connections in parallel, internally they both have $O(n)$ algorithms inside. The famous

"C10k" problem (http://www.kegel.com/c10k.html), is about finding a solution on how to efficiently handle tens of thousands of connections (could be both server and client side - e.g. imagine web crawler) in parallel. Pretty much all Unix systems offer some sort of event driven library and/or kernel primitives to achieve that. Also, there are many ways how to approach the problem, plus some tricks that can be applied on the top.

Contents

- Introduction, Unix and C, programming tools
- Basic Unix concepts and conventions, its API
- Access rights, devices
- Process manipulation, program execution
- Signals
- Process synchronization and interprocess communication
- Network programming
- **Programming with threads**
- Appendix

Threads

- *thread* = *thread of execution*, a basic software "thing" that can do work on a computer
- classic Unix model: single threaded processes
- with introduction of threads, a process becomes just a container for threads
- advantages of multithreaded applications
  - speed-up – a typical objective is having threads on multiple CPUs running in parallel
  - more modular programming
- disadvantages
  - more complex code
  - debugging may become more difficult

- **While one has to put resources into sharing data when working with processes, one has to put resources into managing inherent data sharing if working with threads.** Note that all threads of the same process have equal access to the process virtual address space.

- Not all applications are fit for multithreading as some tasks are not inherently parallel in nature.

- Even that debuggers typically support threads, debugging changes timing so the problem may not reproduce when using the debugger. That is usually not an issue in a single threaded application.

- There is an excellent book on programming with POSIX threads by Butenhof, see page 9. You can also use an online book *Multithreaded Programming Guide* available on http://docs.oracle.com.

- An example situation when you do not want to use threads is if you want to change a real and effective UID of processes. Take OpenSSH – every connection is served by two processes. One, with maximum privileges, usually runs as root, and provides services (allocating a pseudo terminal is one of them) to a second, unprivileged process. The idea is that most of the OpenSSH code does not need any special privilege so if a bug is found in code that is run under an unprivileged user, the damage is much smaller than if such code was run with maximum privileges. This technique is called a *privilege separation* and you could not do the same thing with threads.

Implementation of threads

library-thread model (1:N)

- threads are implemented in a library. Kernel has no knowledge of such threads.
- run-time library schedules threads on processes and kernel schedules processes on CPUs

⊕ less overhead

⊖ more threads of the same process cannot run in parallel

kernel-thread model (1:1)

- threads are a first class kernel citizen

⊕ more threads of the same process can run in parallel on multiple CPUs

hybrid models (M:N)

- N library threads scheduled on M kernel threads, N >= M

⊖ too complex to implement, not really used today

- Original Unix systems used library models (sometimes called *lightweight threads* or *green threads*). Today in general most of the systems stick to the 1:1 model. There was some evolution in the past, e.g. Solaris 9 was using the M:N model and switched to 1:1 in Solaris 10.

- The drawback of the 1:1 model is that there is always some non-trivial overhead associated with thread creation as the library has to call into the kernel as well in order to create the associated kernel thread.

- Threads implemented in a library may be either preemptive or non-preemptive. To achieve preemption, you can use timers and signals. However, if the objective is more in better modular programming than real parallelism, usually non-preemptive threads do fine. Switching threads will be done when a process would normally block in system calls.

- If a system call blocks in a library implemented thread model, the whole process will block as the kernel has no knowledge there are more threads in the process. So the threading library is written the way that non-blocking calls are used, the thread context is saved after that and the library switches to another thread via **setjmp**() and **longjmp**() system calls. Example: `pthreads/setjmp.c`. Another way is to use a non-standard system call **swapcontext**() if the system provides it.

- To implement a 1:N library, there are several things for consideration:
  - how to deal with threads trying to install `SIGALRM` handler if it is already used for firing signal to trigger the dispatcher/scheduler periodically
  - how to implement separate stacks for each thread (especially when `setjmp` was chosen as a building block).
  - what should be the thread states
  - how to avoid all threads to be blocked when one threads blocks e.g. on I/O

POSIX threads (pthreads)

- first came with IEEE Std 1003.1c-1995
- POSIX thread API uses a prefix `pthread_`
- these functions return 0 (= OK) or an error number (values as for `errno`)
  - ... functions do **not** set `errno`
  - so you cannot use functions **perror**() or **err**()
- the standard also defines other functions, for example those that could not be possible to adjust for the use with threads without changing its API (e.g. `readdir_r`, `strtok_r`, etc.)
  - `_r` means *reentrant*, i.e. the function can be called by multiple threads without any side effects

- General information on POSIX is on page 7.
- There are more threading API, the POSIX thread API is just one of them. For example, there is a system call `sproc()` on IRIX, then Cthreads, Solaris threads, GNU Ptr threads (= portable), ...
- The POSIX thread API is available in different libraries on different systems. For example, on Linux you usually need `-lpthread` but on Solaris the API is part of standard `libc`. With `gcc`, instead of `-lpthread`, you can use `-pthread` and the compiler will do what is needed for the specific system (which does not have to be Linux).
- Each POSIX thread API implementation is usually built on top of the native threading library. For example, on Solaris, it's the `thr_` API functions.
- We will talk more on reentrant functions in connection with threads on page 209.
- As already mentioned, given that the POSIX thread API uses `errno` codes directly as return values, the following piece of code is not correct:

  ```
  if (pthread_create(&thr, NULL, thrfn, NULL) != 0)
          err(1, "pthread_create");
  ```

  as it will print possibly something like the following on error (unless `errno` was set by previous code which would make it even more confusing):

  - "`a.out: pthread_create: Success`" on Linux distributions

  - “a.out: pthread_create: Error 0” on Solaris
  - “a.out: pthread_create: Unknown error: 0” on FreeBSD
  - or something else based on the system and the concrete message it uses for errno equal to 0, unless errno is already set otherwise.

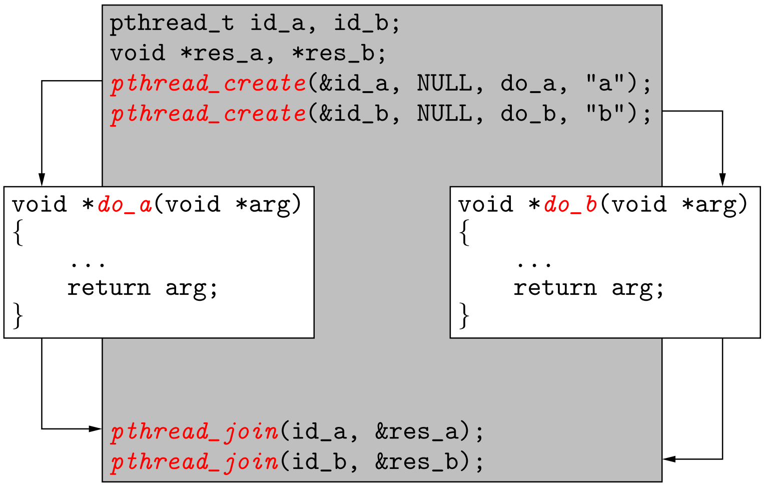
  The Linux approach might confuse the programmer as leaving errno zero does not have to mean the function did not fail, as we just showed. FreeBSD makes it obvious that something is not entirely right. Example that shows such a situation: pthreads/wrong-err-use.c. The correct code could look like this:

```
int e;

if ((e = pthread_create(&thr, NULL, thrfn, NULL)) != 0)
        errx(1, "pthread_create: %s", strerror(e));
```

- Other functions that use errno work the same with POSIX threads as each thread has its own errno. In that case, it is redefined using a function (which can either return the value or an address which is dereferenced). Check /usr/include/bits/errno.h on Linux if interested.

Example: thread creation



```
pthread_t id_a, id_b;
void *res_a, *res_b;
pthread_create(&id_a, NULL, do_a, "a");
pthread_create(&id_b, NULL, do_b, "b");
```

```
void *do_a(void *arg)
{
    ...
    return arg;
}
```

```
void *do_b(void *arg)
{
    ...
    return arg;
}
```

```
pthread_join(id_a, &res_a);
pthread_join(id_b, &res_b);
```

- This is a trivial example. The process (main thread) creates two more threads and waits for them to finish. This process thus have 3 threads in total.

Thread creation

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

- creates a new thread, puts its ID to *thread*
- with attributes from `attr`, e.g. its stack size, `NULL` means default attributes
- function *`start_fn`*() will be started in the thread using argument *`arg`*. After the function returns, the thread ceases to exist.
- with the `pthread_attr_t` objects can be manipulated using **pthread_attr_init**(), **pthread_attr_destroy**(), **pthread_attr_setstackaddr**(), etc. . .

- Once a thread is created, it is up to the scheduler when the thread will really start running. The thread can voluntarily give up the CPU by using `sched_yield` (this is POSIX function, unlike `pthread_yield`).
- If the main thread does not wait for the thread to finish, the whole program will terminate even though there are still some threads running, see `pthreads/pthread_create.c`
- Be careful not to use this:

  ```
  for (i = 0; i < N; i++)
      pthread_create(&tid, attr, start_routine, &i);
  ```

  It looks like we pass each thread its index. However, before the started thread gets scheduled to run, a next iteration might happen, modifying `i`.
- Examples: `pthreads/wrong-use-of-arg.c`, `pthreads/correct-use-of-arg.c`.
- If you need to pass only one value, you could use the following (**note that it is implementation specific in the C standard so it is not portable**);

  ```
  assert(sizeof (void *) >= sizeof (int));
  for (i = 0; i < N; i++)
      pthread_create(&tid, attr, start_routine, (void *)(intptr_t)i);
  ```

  . . . and in function `void *start_routine(void *arg)` cast the pointer back to integer.

```
printf("thread %d started\n", (int)arg);
```

Example: `pthreads/int-as-arg.c`

- If we need to pass more bytes than it is the size of the pointer, you must pass a pointer to memory where the passed data is stored, or use global variables. Accessing global variables must be synchronized, of course. More on that on page 195.

- `pthread_t` is a transparent type and its implementation is not of your concern. Usually it is an integer though used to map to the native threads provided by the system. If you create several threads, you need to pass a different address for a `pthread_t` variable otherwise you will not be able to further manipulate with the threads from the main thread, e.g. waiting for them to finish.

Thread private attributes

- instruction pointer
- stack (automatic variables)
- thread ID, available through

  `pthread_t` **`pthread_self`**`(void);`
- scheduling priority and policy
- value of `errno`
- thread specific data – a pair of

  `(pthread_key_t` *`key`*`, void *`*`ptr`*`)`
- signal mask

- A key created by **pthread_key_create**() is visible from all threads. However, in every thread the key may be associated with a different value via **pthread_setspecific**().

- Each thread has a fixed size stack which **does not automatically increase.** It is usually anywhere from 64 kilobytes to a few megabytes. If you cross that limit, the program will quite probably crash. If you want a stack of a greater size than what is the system default, you have to use *`attr`* when creating a thread. Example: `pthreads/pthread-stack-overflow.c`

- You can read more about thread specific data on page 191.

- More on per thread signal mask is on page 194.

Terminating the calling thread

```
void pthread_exit(void *val_ptr);
```

- terminates the calling thread, it is similar to **exit**() for processes

```
int pthread_join(pthread_t thr, void **val_ptr);
```

- waits for thread *thr* to finish, the value passed to **pthread_exit**() or the return value is stored in the location referenced by *val_ptr*

```
int pthread_detach(pthread_t thr);
```

- indicate that storage for the *thr* can be reclaimed when the thread terminates. **pthread_join**() can no longer be used.

- If **pthread_exit**() is not used, it is implicitly called when the thread terminates with the value from the function `return`.
- The contents of the exiting thread's stack are undefined so you should not use pointers to the function local variables from memory pointed to by *val_ptr*.
- If you do not intend to call **pthread_join**(), you need to call **pthread_detach**() or use the attributes (see below). If you do not, a memory needed to carry information for a subsequent **pthread_join**() will not be freed. It is a similar situation as with accumulated zombies. You can just call it like this in the thread function:

  ```
  pthread_detach(pthread_self());
  ```

- You can also set the thread attributes when creating the thread, using **pthread_attr_setdetachstate**() with `PTHREAD_CREATE_DETACHED` on the attribute variable and then use that in **pthread_create**(). Example on setting the attributes: `pthreads/set-detachstate.c`
- You can use `NULL` as *val_ptr* in **pthread_join**(), telling the system you are not interested in the return value.
- Any thread can wait for another thread, not just the one that created it.

- We recommend to always check the return value of **pthread_join**() to make sure you wait for the right thread. If you use an incorrect thread ID, the function returns immediately with an error.

- In contrast to waiting for processes to finish, **one cannot wait for any thread to finish**. The rationale is that since there is not parent–child relation, it was not deemed necessary. However, some system provide that functionality, e.g. on Solaris you can use `0` for a thread ID in **thr_join**(). If you needed this functionality with POSIX thread API, it is easy to set threads as *detached* and use a condition variable together with a global variable. More on that on page 199.

- Examples: `pthreads/pthread-join.c`, `pthreads/pthread-detach-join.c`

Initialization

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

- in *once_control* you pass a pointer to statically initialized variable

  `pthread_once_t once_control = PTHREAD_ONCE_INIT;`

- first thread that calls **pthread_once**() calls *init_routine()*. Other threads will not call the function, and if it has not finished yet, will block waiting for it to finish.

- you can use it for dynamic initialization of global data in libraries where multiple threads may be using the library API at the same time

- In program itself you will probably not need this function. You can just call the initialization function before you create the first thread.

- It is undefined if *once_control* is a local variable or does not have an expected value.

- This is handy for lazy initialization, i.e. only after one of the threads call into APIs of the library (as opposed to when the library is being loaded), the library becomes initialized and the semantics of `pthread_once` will make sure this will happen only once.

Cancel execution of a thread

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

- cancel *thread*. Depends on:

`int` **pthread_setcancelstate**(`int` *state*, `int *`*old*);

- sets new state and returns the old value:
  - `PTHREAD_CANCEL_ENABLE` ... cancellation allowed
  - `PTHREAD_CANCEL_DISABLE` ... cancellation requests against the target thread are held pending

`int` **pthread_setcanceltype**(`int` *type*, `int *`*old*);

- `PTHREAD_CANCEL_ASYNCHRONOUS` ... immediate cancellation
- `PTHREAD_CANCEL_DEFERRED` ... cancellation requests are held pending until a cancellation point is reached.

- Cancellation points will occur when a thread is executing functions specified in the standard, like **open**(), **read**(), **accept**(), etc. The full list is usually in the `pthread_setcancelstate` man page.
- The **pthread_testcancel**() function creates a cancellation point in the calling thread. The **pthread_testcancel**() function has no effect if the ability to cancel is disabled.
- Be very careful with the use of `PTHREAD_CANCEL_ASYNCHRONOUS` as it may lead to data inconsistency as the cancellation may happen any time, even in your critical sections.
- Cleanup functions are called on cancellation, see page 192. For example, if cancelling a thread holding a mutex, you could use the cleanup function to unlock it.
- Functions **pthread_setcancelstate**() and **pthread_setcanceltype**() provide similar functionality to threads as is manipulating a signal mask to processes.
- Example: `pthreads/pthread-setcanceltype.c`

Global variables per thread

```
int pthread_key_create(pthread_key_t *key,
                       void (*destructor)(void *));
```

- creates a key that can be associated with a value of `(void *)` type. Function *destructor()* are called for all keys whose value is not `NULL` on thread termination.

```
int pthread_key_delete(pthread_key_t key);
```

- deletes the key, does not change the associated data

```
int pthread_setspecific(pthread_key_t key,
                        const void *value);
```

- sets pointer *value* to *key*

```
void *pthread_getspecific(pthread_key_t key);
```

- returns the value of *key*

- Global variables and dynamically allocated data is common to all threads. Thread specific data provides a way to create a global variable per thread. Note the difference between that and a local variable in the thread function – as you know, in C, the local variable is not visible in other functions called from the thread function. Thread specific data is a very useful feature. Imagine you have existing multithreading code using a global storage place which suffers from heavy contention. You can easily create a thread specific data to create a storage place per thread with minimal changes to the original code.
- When you create a key, `NULL` is associated with it. If you do not need the destructor function, use `NULL`.
- Destructors are called in unspecified order on all keys with a value different from `NULL`. Its value is set to `NULL` and the original value is used as a parameter to the destructor. If, after all the destructors have been called for all non-`NULL` values with associated destructors, there are still some non-`NULL` values with associated destructors, then the process is repeated. If, after at least `PTHREAD_DESTRUCTOR_ITERATIONS` iterations of destructor calls for outstanding non-NULL values, there are still some non-`NULL` values with associated destructors, the implementation may (it usually does otherwise you could end up with an infinite loop) stop calling destructors.

Cleanup functions

- each thread has a stack of cleanup routines called when functions **pthread_exit**() or **pthread_cancel**() are called (but not when `return` is used). Routines are run from the top of the stack down.
- after cleanup functions are called, thread specific data destructors are called in unspecified order

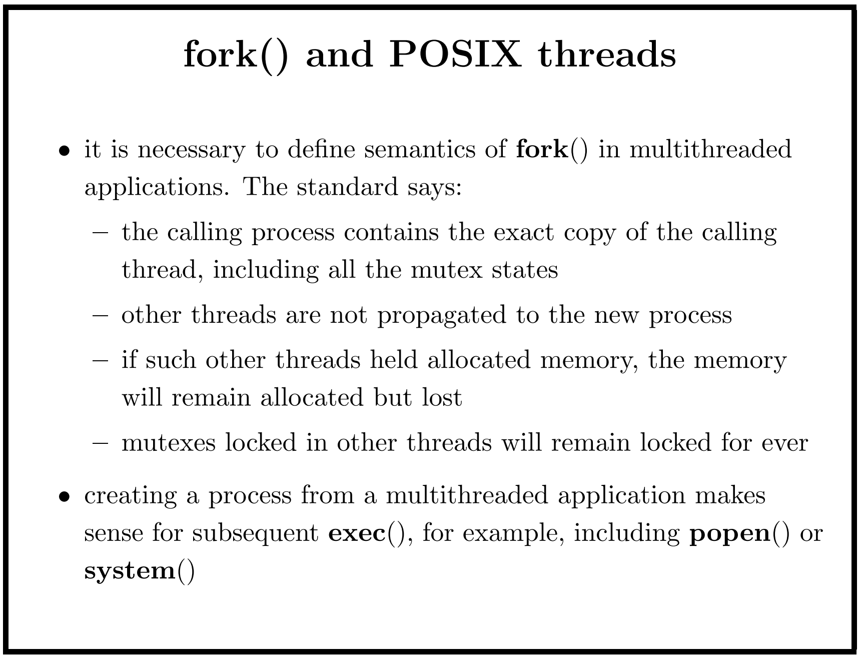
```
void pthread_cleanup_push(void (*routine)(void *),
                          void *arg);
```

- add *routine* to the top of the stack

```
void pthread_cleanup_pop(int execute);
```

- removed a routing from the top of the stack. Will call the routine if *execute* is non-zero

- Cleanup functions are called as `routine(arg)`.



fork() and POSIX threads

- it is necessary to define semantics of **fork**() in multithreaded applications. The standard says:
  - the calling process contains the exact copy of the calling thread, including all the mutex states
  - other threads are not propagated to the new process
  - if such other threads held allocated memory, the memory will remain allocated but lost
  - mutexes locked in other threads will remain locked for ever
- creating a process from a multithreaded application makes sense for subsequent **exec**(), for example, including **popen**() or **system**()

- No cleanup routines or thread specific data destructors are called for threads not propagated to the new process.

- Note that the way how **fork**() works also depends on the system used. For example, in Solaris before version 10 (i.e. before 2005), **fork**() in the `libthread` library (different from `libpthread`) was the same as **forkall**().

- Examples: `pthreads/fork.c`, `pthreads/fork-not-in-main.c`, and also `pthreads/forkall.c`

- You can use **pthread_atfork**() to set handlers that are executed before **fork**() is called in the parent process, and then after **fork**() is called both in the parent and its child. The handlers are executed in the context of the thread that calls the **fork**(). Such handlers are very useful when **fork**() is used not only as a wrapper around **exec**(). After **fork**(), all variables in the child are in the state as in the parent, so if a thread not present to the child held a mutex in the parent (see page 196), the mutex stays locked in the child, and trying to lock in the child will lead to a deadlock. However, if the parent locks all the mutexes in the *`pre-fork`* handler and then unlocks them in the *`post-fork`* handler (both for the parent and the child), you will avoid such deadlocks. That is because when locking mutexes in the *`pre-fork`* handler, other threads are still running so the mutexes held by them should be released eventually (usually each thread exits a critical section in a short time in well written code). Example: `pthreads/atfork.c`. For more on this topic, see [Butenhof].

- See page 196 on why mutexes locked in other threads on **fork**() stay locked forever.

Signals and threads

- signals can be generated for a process (the `kill` syscall) or for a thread (error conditions, the `pthread_kill` call).
- signal handling is the same for all threads in the process, the mask of blocked signals is specific for each thread, can be set with

```
int pthread_sigmask(int how, const sigset_t *set,
                    sigset_t *oset);
```

- a signal for a process is handled by one of its threads, one of those that do not have such signal blocked.
- one thread can be dedicated for signal handling using the `sigwait` call. The other threads have the signals blocked.

- If the action for a signal is set to process exit, the whole process will exit, not just one thread.
- New thread will inherit signal mask from the creator thread.
- Similarly to the use of `sigwait` with processes (page 134) – just block given signals in all threads, including the thread processing the signals using `sigwait`. **This way of signal handling in threaded environment is usually the only recommended.** And it is easy to implement as well. From a previous note, it is sufficient to mask the signals only once, in the main thread (before creating any new threads), because the mask will be inherited with each `pthread_create` call.
- Do not use `sigprocmask` (page 133) in threaded environment, because the behavior of this call is not specified by the standard in such environment. It may work, or not.
- Example: `pthreads/sigwait.c`.
- **Note** that this way of signal handling should not be used for synchronous signals such as `SIGSEGV`, `SIGILL`, etc. These signals are generated directly for a thread, so if blocked the dedicated signal handling thread may not "see" them (if it did not cause them by itself). Also, the standard does not specify if blocking these signals should actually work as mentioned on page

127. Some systems normally deliver these signals, making the process exit. The standard says:

> *If any of the `SIGFPE`, `SIGILL`, `SIGSEGV`, or `SIGBUS` signals are generated while they are blocked, the result is undefined, unless the signal was generated by the `kill` function, the `sigqueue` function, or the `raise` function.*

Example: `pthreads/sigwait-with-sync-signals.c`. This example shows that Solaris 10 and 11, FreeBSD 7.2 and a Linux distribution, that reports itself as "Gentoo Base System release 1.12.13", is the `SIGSEGV` signal delivered and the process killed even though it is masked. There used to be a system that did not deliver the signal when masked – FreeBSD 6.0. It should be possible to handle synchronous signals if you terminate the process in the handler itself (e.g. after printing a user friendly error message), see page 127, which also contains an example.

Thread synchronization in general

- most of the programs employing threads needs to share data between them
- or needs to execute given actions in certain order
- . . . all of this needs to **synchronize** running threads activity
- for processes it is necessary to make some effort to actually share data, for threads on the other hand it is necessary to maintain natural data sharing.
- will describe:
  - mutexes
  - condition variables
  - read-write locks

- Process synchronization is described on pages 136 to 150.
- By using mutexes and condition variables it is possible to construct any other synchronization model.
- The exact behavior of synchronization primitives is largely determined by the scheduler. It decides which of the threads waiting for releasing a lock will be woken up after the lock is actually released. This leads to classical problems such as a *thundering horde* (lots of threads waiting for unlock) or a *priority*

inversion (thread holding a lock has lower priority than the thread waiting for the lock).

Thread synchronization: mutexes (1)

- the simplest way how to ensure synchronized access to shared data between threads
- initialization of statically defined mutex:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- initialization of dynamically allocated mutex `mx` with attributes `attr` (these are set using `pthread_mutexattr_...`; if `attr` is `NULL`, default attributes will be used)

```
int pthread_mutex_init(pthread_mutex_t *mx,
                       const pthread_mutexattr_t *attr);
```

- after done using the mutex it is possible to destroy it:

```
int pthread_mutex_destroy(pthread_mutex_t *mx);
```

- Mutex = *mutual exclusion*
- Special form of Dijkstra semaphores – the difference between mutexes and binary semaphores is that **mutex has an owner and locked mutex must be unlocked only by the thread that acquired it.** This is not the case with semaphores. In order to check whether given mutex was locked by different thread when acquiring it, it is necessary to test return value of `pthread_mutex_lock`, and also have the lock checking set, see below.
- Mutexes are meant to be held for short time only. They are used for critical section (see the definition on page 138) implementation, similarly to lock-files or semaphores (if used like locks).
- Lock checking is governed by a mutex type. By default the mutex type is set to `PTHREAD_MUTEX_DEFAULT`. This type by itself does not define the result of (a) locking a locked mutex, (b) unlocking a mutex locked by a different thread, or (c) unlocking an unlocked mutex. Unix/Linux systems will map that macro to `PTHREAD_MUTEX_NORMAL` or `PTHREAD_MUTEX_ERRORCHECK` (ignoring the recursive type, see below). Thus, depending on a specific system, locking an already locked mutex will result in a deadlock (`NORMAL`) or not (`ERRORCHECK`). In the latter case, a return value will contain information about the error and if not tested, the program will wrongly assume the mutex is locked. For the `NORMAL` mutex type the result of (b) and (c) is not defined, for `ERRORCHECK` an error will be returned. In general you should

avoid any undefined behavior unless specifically documented by the system at hand. More information can be found in the POSIX standard or the `pthread_mutexattr_settype` man page. Checking return values of mutex functions can make the code slightly less readable however it can be wrapped in a macro. Alternatively, the checks can be used during development only. Solaris and Linux use `NORMAL` type by default, FreeBSD uses `ERRORCHECK`. Example: `mutexes/not-my-lock.c`.

- Another type is `PTHREAD_MUTEX_RECURSIVE` that holds a count of lock actions done by given thread. The remaining threads will be granted access only if the count reaches 0. This mutex cannot be shared between processes.

- What are recursive mutexes good for? Let's assume there are two libraries, `A` and `B`. There is a library function `A:foo()` acquires a mutex and calls `B:bar()`, and in turn calls `A:bar()` which tries to acquire the same mutex. Without recursive locks a deadlock will ensue. With recursive mutexes that's fine if these two calls are done by the same thread (another thread will get blocked). That is, assuming `A:foo()` and `A:bar()` are aware that the same thread can be already in the critical section.

- The behavior according to mutex types:

	`NORMAL`	`ERRORCHECK`	`RECURSIVE`
detects deadlock	N	Y	N/A
multi locking	deadlock	error	success
unlock by different thread	undefined	error	error
unlock unlocked	undefined	error	error
can be shared between processes	Y	Y	N

- Static mutex initialization using before mentioned macro will set default attributes. It is possible to use initializer function also for statically allocated mutex. If a mutex is dynamically allocated, it is always necessary to use `pthread_mutex_init`, even if the default attributes are desired or not.

- Dynamic mutexes are needed e.g. when a data structure containing a mutex protecting it is dynamically allocated. In such a case, before calling `free` with the data structure, it is first necessary to properly destroy the mutex (that can also have some memory allocated). Destroying a locked mutex is not defined by the standard.

- Copying mutexes is also not defined by the standard – the result of such an operation depends on the implementation. It is possible to copy a pointer to a mutex and work with that.

- A mutex destroy means its deinitialization.

Mutexes (2)

- to lock and unlock mutex:

```
int pthread_mutex_lock(pthread_mutex_t *mx);
```

and

```
int pthread_mutex_unlock(pthread_mutex_t *mx);
```

- If a mutex is already locked, the attempt to acquire it will result in the thread being locked (depending on mutex type). It is possible to use:

```
int pthread_mutex_trylock(pthread_mutex_t *mx);
```

. . . that will attempt to acquire the mutex and if that fails it will return error

- Locking a mutex that is being held by another thread is not correct. Sometimes a (self)deadlock can ensue, see the previous page. If you need to unlock a mutex locked by a different thread, use binary semaphores instead.

- When creating a program where efficiency is paramount, it is necessary to think about how many mutexes will be needed and how exactly they will be used. Even a library that was not written with threads in mind can be converted to be thread-safe (see page 209) by acquiring a per-library lock on any library function entry and releasing the lock before the function exits. Such a lock may be called a "giant" mutex, and it may lead to lock contention for every consumer of such a library as at any given moment only one thread may execute the library code. On the other hand, if using a large number of mutexes to synchronize access to many small sections, significant amount of time might be spent in the overhead of calling functions implementing the locking. It is therefore desired to search for a compromise. (Or use an algorithm that does not require locks at all).

- Examples: `mutexes/race.c`, `mutexes/race-fixed.c`

- Mutexes can be shared between processes so that their threads will synchronize on them. This is done by using shared memory that will be set as an attribute of such mutexes. See the `pthread_mutexattr_setpshared` man page.

Condition variables (1)

- mutexes provide synchronization for shared data
- condition variables pass information about the shared data – for example that the value has changed
- . . . and allows to put threads to sleep and wake them up
- therefore **each condition variable is always associated with exactly one mutex**
- one mutex can be associated with multiple condition variables
- using mutexes and condition variables it is possible to construct other synchronization primitives – semaphores, barriers, . . .

- In other words – condition variables are handy in a situation when a thread needs to test the state of **shared** data (e.g. number of elements in a queue) and voluntarily put itself to sleep if the state is not as desired. The sleeping thread may be woken up by another thread after the latter changed the state of the data in a way that the situation which the first thread was waiting on actually happened (e.g. by inserting an item into a queue). The second thread wakes the first one by calling a designated function. If no thread is sleeping at the moment, that function would have no effect – nothing will be saved anywhere, it is as if it never happened.
- A condition variable, which is an opaque type for the programmer, is not associated with a concrete condition like "*`n` is greater than 7*". A condition variable may in fact be compared to a flag of a certain color; if it is lifted up, it means that the threads waiting for the flag to be raised are informed (= woken up) and may use this information to its own judgment. Some threads may wait for `n` to be bigger than 7, some other may be waiting solely for `n` to change, and another then for `n` to become 99. It is only up to the programmer whether a separate condition variable will be used for all states of `n` (i.e. we would use multiple flags of different colors) or whether a single condition variable will be used (the same flag color for all situations). For the latter, the threads waiting on `n > 7` and `n == 99` must always test `n` as they know that they are woken up whenever the variable changed. If the variable is not equal to 7 the thread must voluntarily put itself to sleep again. As it is explained further, the **test is necessary to perform after an every wake-up** even if a dedicated condition variable is used for every possible state – it may happen that the system can wake up a sleeping thread (because of

various implementation reasons) without any other thread causing this; it is called a *spurious wake-up*.

Condition variables (2)

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

- initializes condition variable `cond` with attributes `attr` (they are set with the `pthread_condattr_...()` functions), `NULL` = default.

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

- destroys condition variable.

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

- waits on condition variable until another thread calls **pthread_cond_signal**() or **pthread_cond_broadcast**().

- While the condition variables are used for putting threads to sleep and waking them up, given that we work with shared data, there is always a mutex involved when working with a condition variable.

- It is necessary to test the condition after the thread locks the mutex and before the `pthread_cond_wait` is called. If the thread does not perform this operation, it could be put to sleep indefinitely because the message from another thread about the condition changing could go "unnoticed". In other words, a thread cannot enter sleep to wait for situation which happened in the meantime. It does not work like signals which are "held" by the system if they are blocked. It is important to perform this under the protection of the mutex, to be sure what is the state of the data when calling `pthread_cond_wait`.

- The condition variables API works thanks to the fact that when entering critical section the mutex is locked by the thread and the `pthread_cond_wait` **function will unlock the mutex before putting the thread to sleep**. Before exiting from the function the mutex is locked again. It may therefore happen that the thread is woken up while waiting on a condition variable and then put to sleep again when hitting a mutex already locked by another thread. There is nothing complicated about this, it is merely a mutual exclusion of threads in a critical section.

Condition variables (3)

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

- wakes up one thread waiting on condition variable `cond`.

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

- wakes all threads waiting on condition variable `cond`.

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

- waits for `pthread_cond_signal()` or `pthread_cond_broadcast()` or until the system time to reaches the `atime` value.

- One condition variable can be used to announce multiple situations at once – e.g. when inserting or removing an item to/from a queue. Because of this, it is necessary to test the condition the thread is waiting for. Another consequence of this is that it is necessary to use a broadcast in such a situation. Let us assume that both readers and writers are waiting for a condition "state of the queue has changed". If only a single wake-up event is made after an item insertion to the queue, a writer may be woken up but it is however waiting for a different event – an item removal, so it puts itself to sleep again. Thus, the message will remain in the queue until a reader is woken up.

- A thread may be woken up by another thread even in a case when a condition variable is associated with a specific event that is no longer true after the waiting thread is woken up. Let's consider this situation: a thread signals a condition change and right after that another thread locks the mutex and performs an action which invalidates the condition, e.g. removing an item from a queue while the event was "there is an item in the queue". So, the thread woken up finds the queue empty. That is another reason why the condition the thread is waiting for must be **always** tested in a cycle.

- It is also possible that a thread is woken up and the condition is not true due to a spurious wake-up already mentioned in a previous page. So again, the loop must be used.

- The `abstime` parameter of the `pthread_cond_timedwait` function is absolute time, i.e. the timeout expires when the system time reaches the value greater or equal to `abstime`. The absolute time is used so that it is not necessary to recompute the time difference after wake-up events.

Using condition variables

```
pthread_cond_t cond; pthread_mutex_t mutex;
...
pthread_mutex_lock(&mutex);
while (!condition(data))
    pthread_cond_wait(&cond, &mutex);
process_data(data, ...);
pthread_mutex_unlock(&mutex);
...
pthread_mutex_lock(&mutex);
produce_data(data, ...);
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);
```

- The first piece of code is waiting for the condition to change. If this happens, the data changed and can therefore be processed. The second piece of code (executed in a different thread) prepares the data so it can be processed. Once ready, it will signal the consumer.

- The `pthread_cond_wait` function will automatically unlock the mutex and put the thread to sleep. Once the thread is woken up, the system will lock the mutex first (this will be done by the condition variable implementation) and only after that `pthread_cond_wait` will return.

- When signalling that something has changed, it does not mean that after the change the condition will be true. Moreover, `pthread_cond_wait` can return even if no thread called `pthread_cond_signal` or `pthread_cond_broadcast`. This is another reason for testing the condition after wake-up and possibly going to sleep again.

- The mutex in the example above is unlocked only after the condition was signalled however it is not necessary. The signalling can be done after unlocking and in such a case it can be more efficient (depending on a given implementation) because the thread that has been woken up will not get immediately blocked by the mutex which is still held by the thread signalling from within the critical section.

- Example: `cond-variables/queue-simulation.c`

Read-write locks (1)

```
int pthread_rwlock_init(pthread_rwlock_t *l,
                        const pthread_rwlockattr_t *attr);
```

- creates a lock with attributes according to `attr` (set via `pthread_rwlockattr_...()` functions, *NULL* = default)

```
int pthread_rwlock_destroy(pthread_rwlock_t *l);
```

- destroys the lock

```
int pthread_rwlock_rdlock(pthread_rwlock_t *l);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

- acquires the lock for reading (more than one thread can hold the lock for reading); if anyone holds the lock for writing, the calling thread is put to sleep (`rdlock()`) or returns an error (`tryrdlock()`).

- Not a part of POSIX threads from POSIX.1c, rather part of extension POSIX.1j called "advanced real-time extensions".

- More than one thread can hold the lock for reading or at most one thread for writing (and no one for reading).

- Read-write locks are semantically similar to locking files using `fcntl` function.

- It is common that given implementation prefers writer threads to reader threads. E.g. if a lock is owned by writer and some other thread calls function **pthread_rwlock_rdlock** and there is at least one thread waiting in **pthread_rwlock_wrlock**, the writer will be given precedence. See `pthreads/pthread-rwlock.c`.

- There is maximal count of locking operations in given implementation (inferred from the type that holds the lock count). If the maximum is reached **pthread_rwlock_rdlock** returns the `EAGAIN` error, see `pthreads/pthread-rwlock-limit.c`.

Read-write locks (2)

int **pthread_rwlock_wrlock**(pthread_rwlock_t **rwlock*);

- acquires the lock for writing; Blocks if anyone owns the lock for reading or writing.

int **pthread_rwlock_trywrlock**(pthread_rwlock_t **rwlock*);

- like `pthread_rwlock_wrlock()`, however if the lock cannot be acquired it will return error.

int **pthread_rwlock_unlock**(pthread_rwlock_t **rwlock*);

- unlocks the lock

- *Interesting property*: if a thread waiting on a lock receives a signal, the signal handler will be invoked and then it will transparently continue waiting, i.e. there will be no error `EINTR`. The same is true for mutexes and condition variables.

Atomic arithmetic operations

- for architectures where the arithmetic operations are not atomic
- significantly faster than other mutual exclusion mechanisms thanks to using native CPU instructions ensuring atomicity.
- some systems or compilers supply functions for atomic operations, (e.g. `atomic_add(3c)` in Solaris `libc` or `__sync*` API in GCC)
- generally it is better to use the routines from C11 standard via *`stdatomic.h`*, e.g. addition:

```
#include <stdatomic.h>

atomic_int acnt;
atomic_fetch_add(&acnt, 1);
```

- Example `race/atomic-add.c` demonstrates the race condition problem when performing addition and its possible solutions. The program spawns two threads, each thread works with the same global variable `x`, and in a loop, each increments the variable by numbers from a sequence of 1 to `arg[1]`. The threads run in parallel, they compete for access to `x`, and each performs the following cycle:

```
for (i = 1; i < arg; ++i)
        x = x + i;
```

  After the cycle the main thread performs a check of the resulting value of `x`. If the result is not a double of a sum of the sequence, a race condition must have been hit (we can ignore an integer overflow in this case).

  The results and time consumed to complete the program are radically different for situations when the program used a plain unprotected addition, atomic arithmetic functions, and locking using mutexes. The difference in completion times between arithmetic operation functions and mutexes is especially notable on CPUs with hardware parallelism support.

- Similarly there is an API for subtraction, bit operations AND and OR, value assignment, etc.
- The atomic primitives and types in C11 is an optional feature, so should be wrapped under `__STDC_NO_ATOMICS__` negative define.

- A good description of the C11 atomic API can be found on http://en.cppreference.com/w/c/atomic. Some systems have the `stdatomic(3)` man page.

Barrier

- *barrier* is a primitive that holds group of threads together w.r.t. execution.
- all threads wait on barrier until the last thread from the group reaches it; then they can all continue together.
- typical use case is parallel data processing

`int` **pthread_barrier_init**`(pthread_barrier_t *`*`barrier`*`, `*`attr`*`, unsigned `*`count`*`);`

- initializes barrier for *count* entrances

`int` **pthread_barrier_wait**`(pthread_barrier_t *`*`barrier`*`);`

- blocks until the function is called *count* times

`int` **pthread_barrier_destroy**`(pthread_barrier_t *`*`barrier`*`);`

- destroys the barrier

- The barrier API has been defined since SUSv3, The barrier API is non-mandatory part of POSIX (it belongs to *Advanced real-time threads extension*) and hence SUS certified system does not have to implement it, which is e.g. the case of macOS. However, a barrier can be implemented using mutexes and condition variables.
- The barrier can be used e.g. in a situation when it is necessary to perform some initialization between individual phases of processing. The threads need to wait for each other because the initialization can only begin once the previous phase is over. Example `pthreads/pthread-barrier.c` shows the use in such case.
- `pthread_barrier_wait` returns the `PTHREAD_BARRIER_SERIAL_THREAD` value in the last thread that reached the barrier so e.g. a collection of results from the last phase of the run can be done.
- The barrier condition is for example value of a counter to be 0. Each thread that reaches the barrier decrements the counter which is initialized to the number of threads in the beginning. Once a thread decrements the counter and realizes it is not 0, it will enter sleep on a condition variable. If the thread is the one which discovers the counter to be 0, then instead of calling `pthread_cond_wait` it will send a broadcast which will wake up all the

threads sleeping on the barrier. pthread_cond_signal is not enough, since it is necessary to wake up all the threads, not just one. Before entering next phase the counter is initialized to previous value. This needs to be done carefully, for example it is not possible just to reinitialize the counter after the last thread reaches the barrier, because like was shown on page 199 after waking up from pthread_cond_wait the threads need to test that the counter is indeed 0 and if not they need to be put to sleep again. So it can happen that only some (or none) of the threads would wake up. How would you solve this problem ?

POSIX Semaphores – unnamed

- semaphores come from POSIX-1003.1b (real-time extensions)
- the function names do not begin with pthread_, but sem_
- possible to use them with threads

int **sem_init**(sem_t *s, int *pshared*, unsigned int *value*);

- initialize semaphore
- The **sem_post** and **sem_wait** are the same as for named semaphores (see page 148)

int **sem_destroy**(sem_t *s);

- destroys the unnamed semaphore

- Semaphores created using this API do not have a name (as opposed to those created via the sem_open function), hence they are called unnamed.
- The semaphore functions adhere to the classical UNIX semantics – they return -1 on error and set errno.
- Example: semaphores/sem.c

Typical use of threads

- **pipeline**
  - each of the threads performs its own operation on data, which is being passed between threads.
  - each thread performs different operation

  ... image processing, each thread will apply different filter
- **work crew**
  - the threads perform the same operation on different data

  ... image processing using decomposition – each thread processes different part of the image, the result is combination of the results from all the threads; barrier is useful here.
- **client – server**

- This differentiation is coarse, threads can be used in many different ways, these are the most common cases.
- In the case of client – server model, each server thread processes one request from single client.

Thread-safe versus reentrant

- *thread-safe* means that the function can be called from multiple threads in parallel without destructive consequences
  - a function which was not designed to be thread-safe can be converted into one that is – by inserting a locked section.
  - this is obviously not very efficient
- *reentrant* typically means that the function was designed with threads in mind
  - . . . i.e. it works efficiently in multithreaded environment
  - such function should avoid using static data and if possible also avoid using thread synchronization primitives because they slow down the run.

- The consequence of the above is that thread-safe is weaker property than reentrant. It is possible to write thread-safe function using synchronization primitives; rewrite existing function so that it is reentrant requires much more invention.

- The reentrant functions are also the only usable functions in signal handlers.

- These days thread-safe usually means reentrant, however it does not hurt to know the difference.

- For more on library locking see also page 198.

- There exists a number of functions that can be thread-safe, however not reentrant, e.g. `gethostbyname`. Normally this function uses static variable that is used for each call which makes it unsuitable to use in multithreaded environment - it is not thread-safe. However, on FreeBSD 6.0 it is implemented so that it implicitly uses thread-local storage for saving input data and this makes it thread safe. That said, it does not make it totally safe to use. (not mentioning that a program relying on this behavior is not portable) see example `reentrant/gethostbyname.c`. A bit better is to use reentrant version of this function called `gethostbyname_r` (if it is available on given system), which takes the address of where the result will be stored as parameter, which makes it reentrant. Much better is to use standard function `getaddrinfo` (see page) 172, which is reentrant by itself.

- Example: `reentrant/inet_ntoa.c` – shows that even function written like so does not help if it is called twice within the same call of `printf`. Each time it returns pointer with the same address (in one thread) which the `printf` only notes and only uses the contents for the final printing, therefore prints the representation of the last address used in `inet_ntoa`. On Solaris it is possible to observe this with:

```
truss -t\!all -u libnsl::inet_ntoa ./a.out
```

- Man pages on Solaris contain an item called `MT-level` in the `ATTRIBUTES` section that states whether the function can be used in multithreaded environment and any constraints. The levels are described in the attributes(5) man page.

Non-portable thread APIs

- non-portable APIs have the `_np` suffix (*non-portable*) and individual systems define their own calls.
- FreeBSD, Solaris
  - **pthread_set_name_np**`(pthread_t tid, const char *name)`

  → can give a name to a thread
- Solaris
  - **pthread_cond_reltimedwait_np**`(...)`

  → like `timedwait`, however the timeout is relative
- OpenBSD, macOS
  - `int` **pthread_main_np**`(void)`

  → find out if calling thread is the main one (= `main()`)

- Non-portable calls should be used either for system programs or debugging. You never know when the code will be executed on different system, which typically happens once you leave the company and having the time to fix it anymore.
- To find out which non-portable APIs are available on the system, try running `apropos _np`, or using brute-force (use the location of man pages on given system):

```
$ cd /usr/share/man
$ find . -name '*_np\.*'
./man3c/mq_reltimedreceive_np.3c
./man3c/mq_reltimedsend_np.3c
./man3c/posix_spawnattr_getsigignore_np.3c
```

```
./man3c/posix_spawnattr_setsigignore_np.3c
./man3c/pthread_cond_reltimedwait_np.3c
./man3c/pthread_key_create_once_np.3c
./man3c/pthread_mutex_reltimedlock_np.3c
./man3c/pthread_rwlock_reltimedrdlock_np.3c
./man3c/pthread_rwlock_reltimedwrlock_np.3c
./man3c/sem_reltimedwait_np.3c
```

Contents

- Introduction, Unix and C, programming tools
- Basic Unix concepts and conventions, its API
- Access rights, devices
- Process manipulation, program execution
- Signals
- Process synchronization and interprocess communication
- Network programming
- Programming with threads
- **Appendix**

Virtual memory implementation

- processes in UNIX use virtual addresses to access physical memory. The virtual - physical conversion is performed by hardware with the help of kernel.
- in case of memory shortage unused parts of memory are stored to (**swap**) space on disk.
- before SVR2 the `swapper` process (nowadays `sched`) swapped out whole processes.
- from SVR2 on (**demand paging**) is used with **copy-on-write**. Pages are allocated only after first use and private pages are copied only after modification. Freeing and swapping of individual pages is performed by the `pageout` process, swapping of whole processes is done only when critically low on memory.

address translation: access to invalid address or attempt to write into read-only memory will result in the `SIGSEGV` signal to be delivered.

swap: swap area is created on separate disk partition, since SVR4 it can be also in a file.

swapper: the `swapper` process tries to save a process to disk and use the free space for a process that was swapped out earlier.

demand paging: when a process requests a memory only the page is modified. The first instruction addressing the page contents will result in an exception. The kernel will handle it by allocating a page.

copy-on-write: multiple processes can share writable physical page that is however logically private to each process (such situation happens e.g. when a process is created by the `fork` system call). Until the processes only read from the memory, they access the shared page. First attempt to write will cause an exception. The kernel will copy the page and assign a copy of the page to the process that is already a private and the process can change it. The other process keep using the original page.

The pages to be swapped out are searched for by the *NRU* (not recently used) algorithm: each page has the `referenced` and `modified` flags, both zero initially. First access will set `referenced`, a change will set `modified`. Both flags are periodically nullified. The pages that are neither modified or used are freed first. The program code pages and pages of mapped files are not stored into swap area, they are fetched from given file.

File system

- directories form a tree and together with files form an acyclic graph (one file can have multiple references).
- each directory also contains reference to itself '`.`' (dot) and to its parent directory '`..`' (two dots).
- using file system interfaces, it is possible to access other system entities such as:
  - peripheral devices
  - named pipes
  - sockets
  - processes (`/proc`)
  - memory (`/dev/mem`, `/dev/kmem`)
  - pseudo files (`/dev/tty`, `/dev/fd/0`,...)
- from the kernel's perspective, each regular file is an array of bytes.
- all (including network) drives are connected to single tree.

- Devices, files in /proc, terminals, memory etc. are of one type - special files. More types: regular file (hardlink), directory, named pipe, socket, symbolic link.

- A newly created directory has 2 references to itself – one from its parent directory and one of itself, '.':

  ```
  $ mkdir test
  $ ls -ld test
  drwx------   2 janp     staff        512 Mar 23 12:46 test
  ```

- The root user can in some systems, create a cycle in a directory structure. However that might confuse file system traversal tools, hence the cyclical structure is not used much. Symbolic links to directories work everywhere.

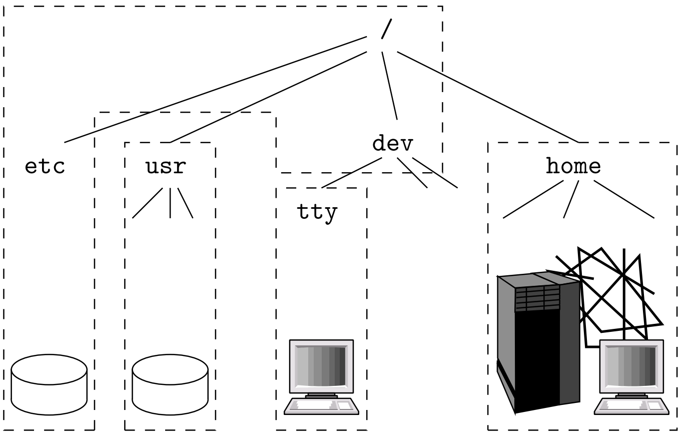
- Named pipes (see page 62) can be used between processes that do not have a "family" relationship. They work as unnamed pipes otherwise.

- The sockets noted above are in the UNIX domain, i.e. they are used for communication within one system. The INET domain sockets, used for network communication (see page 151) are not visible in the file system.

- Debuggers use process images available through /proc. On many Unix-like systems, the /proc tree contains information about kernel and running processes in the form of text files. However, some systems like Solaris, have those files in a binary form only and provide special commands to read those (like pargs, pldd, etc.).

- Modern Unix systems contain a special *devfs* filesystem that reflects an actual system configuration w.r.t. connected devices. E.g. after connecting a USB stick, a related device will appear under /dev. After physically disconnecting it, the item will disappear.

Hierarchical file system



- A *file system* is a data structure to control how data is stored and retrieved. Without it the stored data would have no structure, ownership, etc. The filesystem structure provides for storing directories (folders), files, metadata, etc.

- Each filesystem has a specific structure type it uses, for example `ext4` (Linux specific), `XFS` (used on Linux but coming from SGI IRIX), `JFS` (used on Linux but came from IBM AIX), `UFS` (BSD systems), `FAT32` (Win), `ZFS` (Solaris born, then ported to other systems), `APFS` (macOS and iOS since 2017), etc. A filesystem can be either used on local or remote storage, and in case of a remote storage network filesystem protocols like `NFS` or `AFS` are used to access the data. Note that these network file systems do not define the filesystem structure itself, they only provide for accessing existing filesystems remotely. Each filesystem also has its limits, a largest file size or the maximum size of the filesystem itself, for example.

- Unix does not have A, B, C, D... disks as Windows and other systems. All filesystems are mounted to a single directory hierarchy on any Unix system, as shown on the slide where you can see the root filesystem and three other file systems, mounted on `/usr`, `/dev/tty`, and `/home` directories. You could also further mount other filesystems on directories that are part of these non-root filesystems.

- Each filesystem mounted to the common hierarchy may be formatted using a different filesystem type. However, that is largely transparent to a user traversing the hierarchy. There are some exceptions though. For example, `FAT32` does no provide user and access rights for files so those are faked when such filesystem is mounted.

- Upon the system boot, the root filesystem is mounted first, other filesystems are later mounted via the `mount` command, usually from specific startup services based on the system you use. The startup services sometimes use file `/etc/fstab` as a source of information about what filesystems to mount. You can also use `mount` manually. To unmount a filesystem, the `umount` command is used.
- Some systems also provide for auto mounting where filesystems are mounted during the first access attempt, and may be automatically unmounted after a period of inactivity. Such functionality is usually called an *automounter*. See `autofs`, `automount`, or `amd` for more information.

Typical layout of system directories

`/bin`	...	basic system commands
`/dev`	...	devices
`/etc`	...	configuration
`/lib`	...	basic system libraries
`/tmp`	...	public directory for temporary files
`/home`	...	root of home directories
`/var/adm`	...	administrative files (not on BSD)
`/usr/include`	...	header files for C
`/usr/local`	...	locally installed software
`/usr/man`	...	man pages
`/var/spool`	...	spool (mail, printing,...)

- The `/bin`, `/lib`, `/sbin` directories contain commands and libraries needed for the start of the system when only root filesystem is mounted. The rest of the commands is located typically in `/usr/bin`, `/usr/lib` and `/usr/sbin`. `/usr` may be a separate file system, possibly not accessible during system startup.
- The `s` letter in `sbin` means ,,system", not SUID. It contain binaries not needed by typical user.
- The `/usr` directory tree contains files that are not changing while the system is running and are not dependent on given machine. This property should make it eligible for read-only sharing. On your own machine it will be typically read-write, though.

- The `/lib` directory typically contains libraries needed for programs from the root file system. If all libraries were in `/usr/lib` and `/usr` was separate file system, a problem with mounting `/usr` could paralyze complete system, because no program could be run. In some systems the root file system contains base set of programs that are statically linked. For example FreeBSD has such binaries under the `/rescue` directories. It also has separate directories `/lib` and `/usr/lib`.

- The `/var` directory tree contains data that change while the system is running and are specific for given machine.

- There can be differences in directory layout found between installations of the same operating system.

- The `hier(7)` manual page on FreeBSD and Linux describes directory hierarchy on these systems. Solaris uses `filesystem(5)`.

Peripheral device access

- The `/dev` directory contains special device files. A process opens special file via `open()` and communicates with the device via `read()`, `write()`, `ioctl()`, etc.
- special files are divided into:
  - **character** . . . data is transferred directly between process and device driver, e.g. serial ports
  - **block** . . . data is passed through system cache (buffer cache) in blocks of given size, e.g. disks
- special device identifies concrete device with two numbers
  - **major** . . . number of device driver in the kernel
  - **minor** . . . number within single device driver

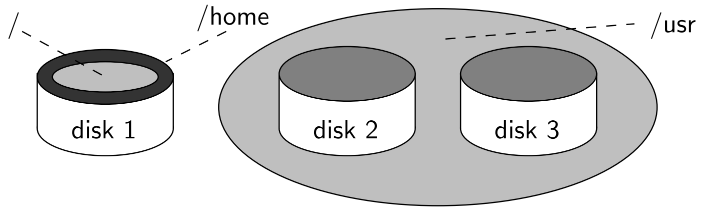
- The cache speeds up operations with peripheral devices. When reading the data is searched in the buffer first. When not present there, they are being read from disk. For the next read they are already available in the buffer. For writing the data is stored into a buffer that is written to the disk later. It is possible to force immediate write to disk. In today's systems the cache is not a standalone structure, rather it is part of virtual memory system.

- Disks in Unix are usually accessible using both character and block interface. The former is used by `mkfs` when creating the file system and also by `fsck` when performing file system consistency check. The latter is used for normal reads and writes. Some systems (FreeBSD) do not have block devices under `/dev`.

- In the past, the contents of the `/dev` directory had to be refreshed by a shell script (`MAKEDEV`) or by hand whenever hardware configuration changed. Today most of the systems populate the directory on the fly as kernel detects addition or removal of hardware components (see ***devfs*** on page 214).
- Immediate write to disk can be forced by using the `O_DIRECT` command via the `fcntl` system call.

Physical layout of file system

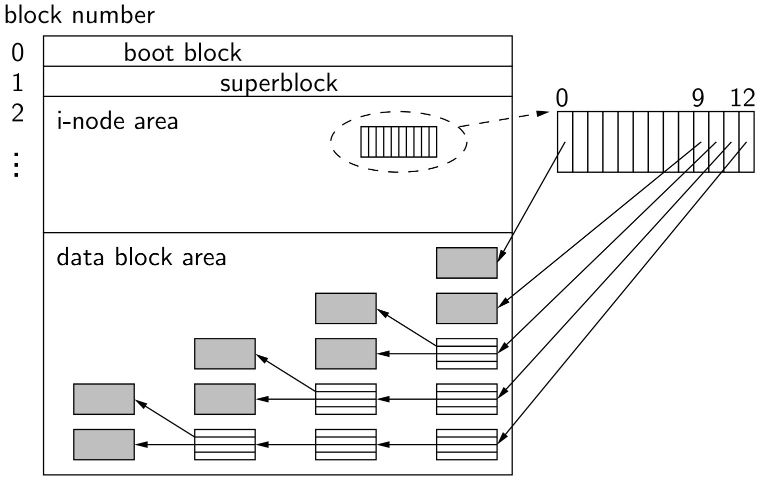
- **filesystem** can be created on:
  - **partition** – part of disk, one disk can have multiple partitions
  - **logical volume** – can be used to connect multiple partitions (that can reside on distinct disks) into one file system.
- more choices: striping, mirroring, RAID



- the expression **file system** has multiple meanings:
  - one file system, i.e. what the `mkfs` command creates
  - whole hierarchy of file systems (the output of the `mount` command)
  - way of organizing given file system and its corresponding kernel module that manipulates the data (UFS2, Ext3, XFS, ...)
- striping means that data blocks that follow each other are stored in parallel on distinct drives, boosting the speed of read operations.
- mirroring stores copies of data to multiple disks to provide redundancy.
- parity disks: data is saved to e.g. 2 drives and the 3rd drive is used for storing XOR value of the first two. If any single disk fails, the data can be reconstructed.

- individual RAID (Redundant Array of Inexpensive Disks) levels include striping, mirroring and parity disks.
- The terminology has to be used with care. For example what is meant by *partition* in the DOS world, is being called *slice* in BSD. There the partitions are defined within one slice and the file systems are created therein.
- ZFS combines both volume manager and file system. It was ported from Solaris to BSD systems.

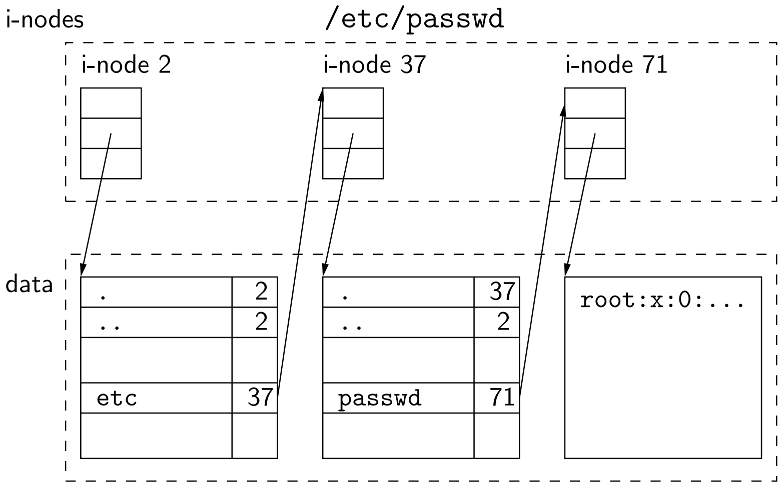
The s5 file system organization



- the original UNIX file system used by default until System V Release 3; used in BSD until 4.1
- properties:
  - 512, 1024 or 2048 byte blocks
  - single (non-duplicated) superblock
  - data area divided into *i-node* and *data block* parts
  - with 1024 byte blocks the theoretical size is over 16 GB
- *boot block* – for OS boot loader
- *superblock* – basic information about the file system: number of blocks for i-nodes, number of file system block, list of free blocks (continues in the free block area), list of free i-nodes (after it is exhausted the i-node table is searched), locks for the lists of free data blocks and i-nodes, modification flag (used for checking whether the file system was correctly unmounted), time of the last update, device information.

- *i-node* – file type, access rights, owner, group, times of last access, modification and i-node modification, number of references, file size, 10 pointers to data blocks and 3 indirect pointers. Note that the time of file creation is not stored.
- maximal file system size: 2113674 blocks, .i.e approximately 1 GB when using 512 byte blocks
- file names – max. 14 characters (14 + 2 = 16, .i.e. power of 2 and therefore seamless storage of directory entries into blocks)
- the file system was able to utilize the disks only to approximately 2% and read operations were able to fetch tens of kilobytes per second.
- for comparison – MS-DOS 2.0 from 1983 supported only FAT12, with maximal file system size of 16 MB. The maximal size up to 2 GB was made possible only in version 4.0 (1988); this version also introduced disk caches, i.e. feature that UNIX has had since its inception in 1970.

Directory structure navigation

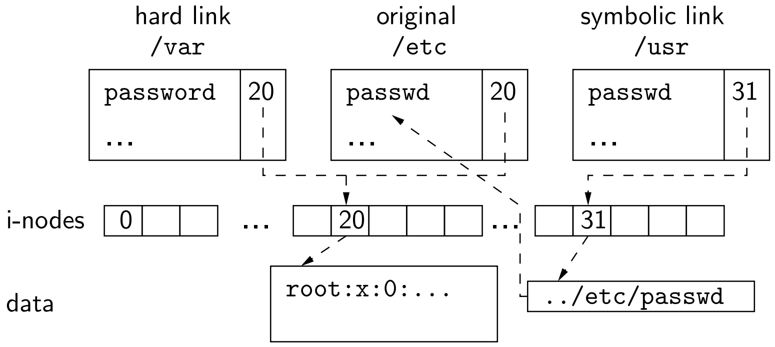


.	2
..	2
etc	37

.	37
..	2
passwd	71

- if a path begins with '/', the navigation starts in the root directory, otherwise it starts in the working directory of given process.
- the i-node number of root directory is typically 2. 0 is reserved for empty i-node and 1 used to be a file into which bad blocks were inserted to avoid them being used.
- path that has multiple consecutive slashes is still valid, i.e. `///a///b///c` is equivalent to `/a/b/c`.

Links



Hard links can be created within one (logical) file system.

- what is “normally” visible in directory listings are hard links. W.r.t. files there are **only** hardlink and symlinks.

  hardlink

  – reference to the same i-node
  – actually second name of a file
  – no difference between original and hard link
  – can be created only within one file system
  – not possible to create for directories

  symbolic link (symlink, softlink)

  – only reference to the real path of file of different type (it is marked as ’l’ in the `ls -l` output), i.e. the type of symbolic differs from ordinary file. Its data contain a simple string – name of the path, either relative or absolute.
  – different behavior for original and symlink (e.g. upon deletion)
  – watch out for relative and absolute paths when moving symbolic link
  – can point to directory or non-existing file

- the simplest way how to verify that two links point to the same file is to use the `-i` option of the `ls` command, that will show the i-node number.

```
$ ls -i /etc/passwd
172789 /etc/passwd
```

File system enhancements

- the goal: lower file fragmentation, reduce head moves by storing i-nodes and data blocks closer
- UFS (Unix File System), originally Berkeley FFS (Fast File System)
- divided into cylinder groups, each contains:
  - superblock copy
  - cylinder group header
  - i-node table
  - bitmaps for free i-nodes and data blocks
  - data blocks
- blocks of size 4 to 8 kilobytes, smaller parts stored into block fragments
- file names up to 255 characters

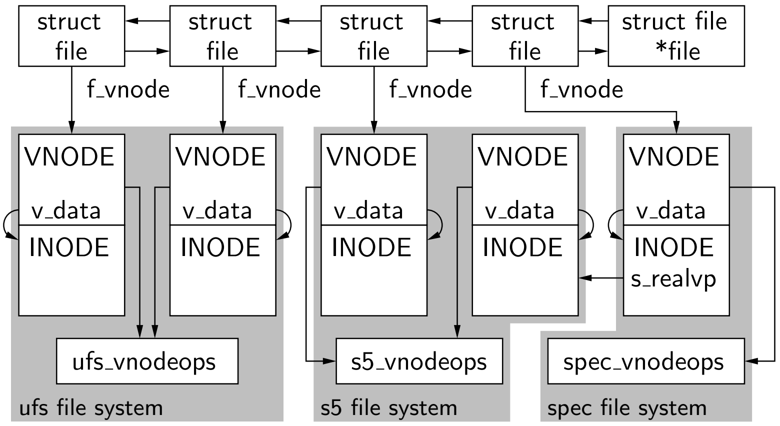
- The superblock in each cylinder group was shifted so that superblocks do not share the same platter.
- more file systems: UFS2, Ext3, ReiserFS, XFS, ZFS etc.
- UFS was still 32-bit, that reflected on maximum file length and maximum file system size. The 32-bit notation means that i-node numbers are represented as 32 bit integers. This gives the theoretical file system limits.
- journaling (XFS, Ext3, ReiserFS) – an effort to reduce the risk of data loss in case of crash and also to speed up the recovery after crash.
- ZFS – modern 128-bit file system developed in Sun Microsystems, since Solaris 10; also present in FreeBSD since version 7.

The development in directory entry management

- maximum file name length of 14 characters was not enough
- FFS – up to 255 characters; each entry also contains its length
- new file systems use B-trees for internal directory structure representation
  - considerably speeds up the work with directories that contain huge number of files
  - XFS, JFS, ReiserFS, . . .
- UFS2 introduced backward compatible *dirhash* for speeding up the access to directories with lots of files

- The principle of *dirhash* is that after first read of a directory a hash structure is created in memory. Subsequent directory accesses are then comparable to the implementations using B-trees. This way an enhancement is made without changing on-disk layout of the file system. Such enhancements cannot be made forever, eventually on-disk format has to change or transition to another file system is necessary.
- small files are often stored in i-nodes, which saves disk I/O

Virtual File System



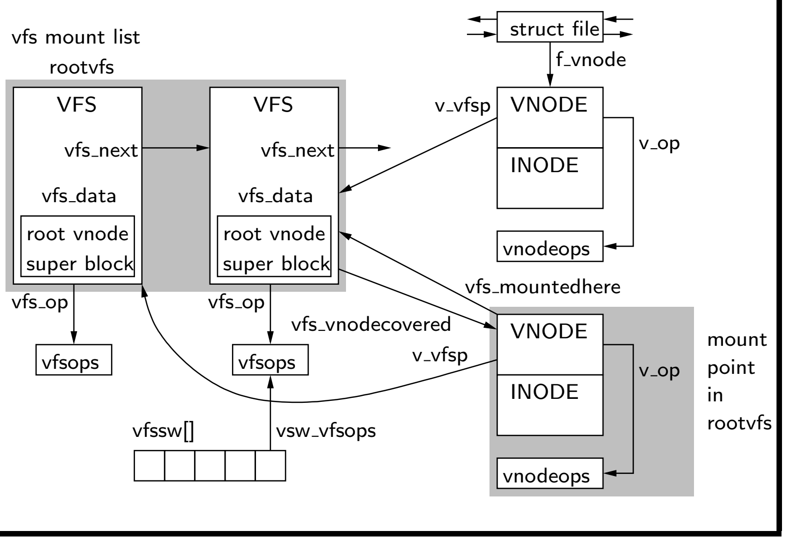
- The *FFS* file system introduced in 4.2BSD was historically the second unix file system. Some manufacturers of unix system started to prefer it considering its better performance and new features, others remained with *s5fs* from compatibility reasons. This deepened the problem with already insufficient interoperability between different unix systems. For some applications neither of these file systems was enough. Gradually the need to work with non-unix systems started appearing, e.g. with *FAT*. With growing popularity of computer networks the demand for file sharing between systems started to increase. This lead to the inception of distributed file systems – e.g. *NFS* (Network File System).

- Given the above described situation it was just a matter of time when fundamental changes in the file system infrastructure will happen to support multiple file system types simultaneously. Several different implementations from multiple manufacturers were made; in the end the de facto standard became the *VFS/vnode* architecture from Sun Microsystems. Today practically all unix unix-like systems support VFS, even though with often non-compatible changes. VFS appeared for the first time in 1985 in Solaris 2.0; soon it was adopted by BSD – FFS with VFS support started to be called UFS.

- the main idea: for each open file there is a `file` structure; it would be actually one slot in the already known system table of open files. This shows to *vnode* (*virtual node*). Vnode contains one part which is independent on given file system and one part dependent that could be e.g. the *inode* structure. This is specific for each type of file system. **Each file system type implements concrete set of functions for individual file operations**. This set is referenced by each vnodes corresponding to given file system. **This set of functions define vnode interface.** When e.g. `open` is called, the kernel will call the corresponding implementation depending on file system type (e.g. from the *ext2fs* module). Implementation dependent part of vnode structure is accessible only from functions of given file system; for kernel it is opaque. See next slide for functions that operate on file system level. **This set defines VFS interface.** These **two sets together** constitute the vnode/VFS interface, generally referred to as VFS.

- For special file types the situation is a bit more complicated, in SVR4 the `file` structure points to `snode` (*shadow-special-vnode*), that defines operations with a device (using the *spec* file system) and using the `s_realvp` pointer it refers to real vnode for the operations with the special file; this file is necessary for example for checking file access rights. Each device can have multiple special files, hence more `snodes` and corresponding real vnodes. All such `snodes` for one device have the `s_commonvp` pointer to one common `snode`, this is however not captured in the picture. When opening a special file, an item corresponding to the special file is searched in hash table of `snodes` of opened devices according to major and minor device number. If the `snode` is not found, new one is created. This `snode` will be then used for operations with the device. More in [Vahalia].

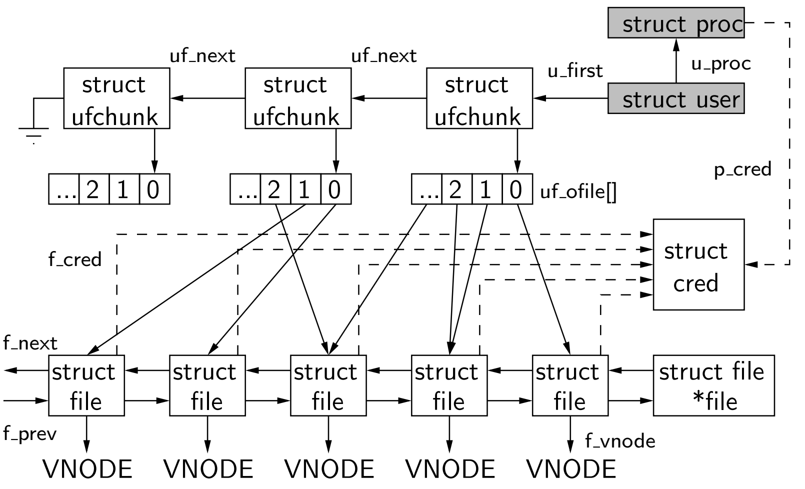
File system hierarchy



- The `vfs` structure contains implementation independent information about given filesystem, similarly to how vnode works for files. This structure represents once concrete physical filesystem, currently mounted to the hierarchy of files. In this linked list there can be more structures of the same filesystem type.

- `rootvfs` – root file system reference
- `vfsops` – function table for given file system type
- `vfssw[]` – array of pointers to the `vfsops` tables for all file system types supported by the system. This table is used for selection based on filesystem type upon `mount` syscall.
- `v_vfsp` – reference from vnode to filesystem (the `vfs` structure), on which the file represented by the vnode lies
- `v_vfsmountedhere` – only in vnode representing a mount point (directory where root of different file system is mounted); points to the `vfs` structure represented mounted filesystem
- `v_vnodecovered` – pointer to the vnode of directory where the filesystem is mounted to

System view of open files II.



- The `proc` and `user` structures are created by the kernel for each process. They contain service information about the process.
- The `ufchunk` structure contains `NFPCHUNK` (usually 24) *file descriptors*, after it is full new `ufchunk` is allocated.
- The `file` structure (*file opening*) contains the mode of file (whether it is opened for reading, writing, etc.), number of descriptors that refers to it, the `vnode` pointer and file position. One opening of file can be shared by multiple file descriptors, if the original descriptor was copied e.g. with the `fork()` or `dup()` syscalls.

- The `cred` structure contains user and group identity of the process that opened the file.
- One vnode corresponding to one file can be shared by multiple file structures if given file was opened multiple times.
- Not all vnodes are associated with the table of opened files. E.g. when executing a program it is necessary to access the executable file and hence a vnode is allocated for that.

Filesystem consistency check and repair

- If a filesystem is not correctly unmounted before the system is halted, the data can be inconsistent.
- The `fsck` command is used to check and correct file system. It progressively tests possible inconsistencies:
  - multiple references to the same block
  - references to blocks outside of data region of given file system
  - incorrect number of inode references
  - incorrect size of files and directories
  - invalid inode format
  - blocks that are neither used or free
  - invalid directory contents
  - invalid superblock contents
- the `fsck` operation is time consuming.
- journaling (e.g. XFS in IRIX, Ext3 in Linux) a transactional (ZFS) file systems do not need `fsck`.

- The data is written to the drives from the memory with some delay. To save all file system buffers the `sync()` syscall can be used. The buffers are saved periodically by special system process (or daemon).
- The `fsck` command only checks metadata. If a data corruption happened it cannot tell, let alone do something about it.
- Example of `fsck` run on **unmounted** filesystem:

```
toor@shewolf:~# fsck /dev/ad0a
** /dev/ad0a
** Last Mounted on /mnt/flashcard
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Cyl groups
24 files, 8848 used, 12951 free (7 frags, 1618 blocks, 0.0% fragmentation)
```

More ways for ensuring filesystem consistency

- traditional UFS – synchronous metadata writes
  - an application creating new file is waiting till the inode is initialized on disk; these operations are done with disk speed, not CPU speed
  - however asynchronous writes more often cause metadata inconsistencies
- solutions to metadata inconsistency problem:
  - *journaling* – one group of operations dependent on each other is written to a journal first; if a problem is encountered the journal can be "replayed"
  - metadata blocks are written to non-volatile memory first
  - *soft-updates* – follows dependencies between pointers to disk structures and writes the data using the *write-back* method so that the data are always consistent on disk.
  - *transactions*

- filesystem *metadata* = i-nodes, directories, free block maps

- *ext2* uses asynchronous metadata writes even by default and when in synchronous mode it is much slower to UFS.

- Dependent operations are for example deleting an item from directory and deleting disk inode. If the inode is deleted first and then the directory entry then if there is outage between these two operations then inconsistency follows – the link points to disk file that does not exist. It is not a problem to avoid this when using synchronous metadata writes (we know when a what is being written, the ordering of writes is therefore under our control) however when using the write-back method it is necessary to solve dependencies of the blocks because with classic synchronization of cache buffers the kernel is not interested in which blocks is written first.

- Often the block dependencies for a cycle. Soft updates can recognize such cycle and break it by performing *roll-back* and after the write is done it performs *roll-forward.*

- The soft updates performance is comparable to that of UFS with asynchronous metadata writes.

- Theoretically soft updates guarantee that it is not necessary to use `fsck` after the reboot, i.e. that the filesystem is in bootable state. It is however necessary to use so called *background fsck* for correcting non-grave errors – this is considered to be one of the big drawbacks of soft updates, especially given how sizes of disks grow over time. Example of an error that does not block booting would be a block that is marked as used however is not used by any file.

- soft updates are not always recommended for the root filesystem. The problem is that metadata loss on root file system (see the 30 second period of writes) can be more dangerous than in `/usr`, `/home` etc. Another disadvantage is that after deleting huge file soft updates do not free up the space immediately.

- Example: it is necessary 4 writes for secure rename operations when using synchronous writes – increase the number of references in inode, create new directory entry, delete old entry, decrease the number of references. If a system goes down between each of these a dangerous situation will ensue. For example 2 references from directories to inode with reference count of 1 is a problem because after deleting one of the references it would look like the file is still on disk, when its data has been deleted already. It is not hard to imagine what would it mean if the file contained important data – e.g. backup. Opposite situation, i.e. one reference to inode with reference count equal 2 is not correction situation either however it does not pose a threat to mount the filesystem and normally use it. In the worst case the file looks as it is no longer on disk while it still exists. For soft updates the rename operation will form a cycle because it is necessary to write the increased reference count first, then directory blocks and then decreased reference. And because increase/decrease is done for the same inode, then for the write it is necessary to do roll-back to (say) 2, write inode to disk, write directory blocks and then roll-forward to reference of 1. During this operation a lock is held for the inode so that no one will read older data. It is simple to show that it is not possible to write any of these 3 blocks in a cycle to match the end of the rename operation so that the roll-back is truly needed – we could consider that the inode did not really change and it is not necessary to decide whether to write it or not; the write of directory reference without increasing reference count in the inode could get us into situation which is described above.

System V semaphore API

```
int semget(key_t key, int nsems, int semflg);
```

- returns identifier of an array of `nsems` semaphores associated with the `key` key (`IPC_PRIVATE` key … private semaphores, each call returns different identifier). `semflg` is OR-combination of access rights and constants `IPC_CREAT` (create if does not exist), `IPC_EXCL` (error, if exists).

```
int semctl(int semid, int semnum, int cmd, ...);
```

- controlling function, optional fourth argument `arg` is of the `union semun` type.

```
int semop(int semid, struct sembuf *sops, size_t nsops);
```

- generic operations P and V.

- How to get the key for `semget` is explained on page 233.
- The most interesting property of System V semaphore implementation is that given syscall does not operate on one semaphore but on **semaphore array**, atomically. Most of the time you will need just one semaphore, i.e. array of one item. For such use the System V semaphores are needlessly complex.
- Access rights are only for reading and writing; the execute bit does not make sense in this context.
- Similar API (function for creating, control and operations) is followed by the other System V IPC mechanism as well.
- Once the array of semaphores is created by a process, the other processes can use `semctl()` and `semop()`, without calling `semget()` beforehand. This is valid also for the semaphores created with the `IPC_PRIVATE` key, for which it is not possible to call `semget()`, because it would create new semaphore array. This is done so that private semaphores can be inherited across `fork` as well.

System V semaphore API: `semctl()`

- `semnum` … number of semaphore in the array
- possible values of `cmd`:
  - `GETVAL` … returns semaphore value
  - `SETVAL` … **sets semaphore to value** `arg.val`
  - `GETPID` … PID of the process that performed the last op.
  - `GETNCNT` … number of processes waiting on bigger value
  - `GETZCNT` … number of processes waiting for zero value
  - `GETALL` … saves the values of all semaphores to `arg.array`
  - `SETALL` … sets all values according to `arg.array`
  - `IPC_STAT` … saves number of semaphores, access rights, times of the last `semctl()` and `semop()` into `arg.buf`
  - `IPC_SET` … sets access rights
  - `IPC_RMID` … destroys the semaphore array

- The `semctl(semid, semnum, SETVAL, arg)` call corresponds to generic semaphore operation `init(s, n)`.

System V semaphore API: `semop()`

- operation is performed atomically (i.e. either it will be done for all semaphores or none) on `nsops` semaphores according to the array `sops` of structures `struct sembuf`, that contains:
  - `sem_num` ... semaphore number
  - `sem_op` ... operation
    - ∗ `P(sem_num, abs(sem_op))` for `sem_op < 0`
    - ∗ `V(sem_num, sem_op)` for `sem_op > 0`
    - ∗ waiting on zero semaphore value for `sem_op == 0`
  - `sem_flg` ... OR-combination
    - ∗ `IPC_NOWAIT` ... when the operation cannot be performed it does not wait and returns error
    - ∗ `SEM_UNDO` ... upon process exit undo the semaphore operations

- The atomicity across the array avoids deadlock e.g. in this situation: two processes A and B will be using two semaphores for synchronization of access to two system resources. A will lock in the $(0, 1)$ order and process B will use the $(1, 0)$ order. In the moment when process A locks semaphore 0 and B locks 1, deadlock will happen because neither process can continue due to cyclical dependency. Using atomic operation of both semaphores means the operation will be successful only for one process that acquires both semaphores, the other will wait.

- `SEM_UNDO` ensures that upon process exit the semaphore that were locked by the process (used as locks) will become unlocked.

Creating System V IPC resources

- one process creates the resources, other connect to it.
- after being done with the resource, it has to be destroyed.
- `semget()`, `shmget()` and `msgget()` have the key identifying the resource as a first argument. Group of processes that want to communicate has to agree on common key. Different groups of communicating processes should have different keys.

`key_t` **ftok**`(const char *`*path*`, int `*id*`);`

- returns key derived from `path` and `id` number. Returns the same key for the same `id` and arbitrary path referencing the same file. Returns different keys for different `id` or different files on the same volume.

`ftok()` notes:

- Only the 8 lowest bits are used from the `id` argument.
- It is left unspecified whether the same key will be returned across file deletion. Usually not because the key often reflects inode number.
- Different keys for different files are not always guaranteed. E.g. on Linux the key is determined using the combination of 16-bit inode number, 8 bits of `id` and 8 bits of minor device number. Same key is returned if the inode numbers have the same 16 lower bits.
- If unrelated processes need to use the same semaphore, **the file name used for the key needs to be agreed on beforehand.**
- Example: `race/sem-fixed-race.c` (this is the `race/race.c` example from page 137 fixed with semaphores)

Books on UNIX history

- Peter Salus: **A Quarter Century of UNIX**, Addison-Wesley; 1st edition (1994)
- Libes D., Ressler, S.: **Life With Unix: A Guide for Everyone**, Prentice Hall (1989)
- **Open Sources: Voices from the Open Source Revolution**, chapter **Twenty Years of Berkeley Unix From AT&T-Owned to Freely Redistributable**; O'Reilly (1999); on: http://oreilly.com/openbook/opensources/book/index.html

. . . lots of material on this topic is online

- The chapter on the BSD UNIX from *Open Sources* written by Marshall Kirk McKusick is excellent.

UNIX prehistory

- 1925 – **Bell Telephone Laboratories** – research in communication (e.g. 1947: transistor) within AT&T
- 1965 – BTL s General Electric and MIT development OS **Multics** (MULTIplexed Information and Computing System)
- 1969 – Bell Labs initiates a project, **Ken Thompson** writes assembler, basic OS and file system for PDP-7
- 1970 – Multi-cs ⇒ Uni-cs ⇒ Uni-x
- 1971 – UNIX V1, and ported to PDP-11
- December 1971 – first edition of *UNIX Programmer's Manual*

- AT&T = American Telephone and Telegraph Company
- Multics was a system that significantly influenced the development of operating systems. It includes many innovative ideas, some of which were not always accepted positively. It greatly influenced UNIX, that adopted many ideas and tried to fix the shortcomings. The main difference was mainly in that UNIX was designed as simpler system.
- After BTL left the Multics project, GE sold its computer division to Honeywell including the Multics project, that was further developed under its patronage (virtual memory, multiprocessors, . . .) till 1985. The last Multics installation worked in the Canadian Department of National Defense and the system was used actively for example during the Persian gulf war. Definitive shutdown was made 31st October 2000. More information can be found on http://www.multicians.org.
- Before the work on the development environment for PDP-7 started, Thmopson wrote the *Space Travel* program, that was developed in other environment (Honeywell 635) and transferred on tape to PDP-7.
- In overall there were 10 editions of this manual, corresponding to ten UNIX versions developed in BTL.
- UNIX V1 did not have the`pipe` syscall !!!
- The version 1 manual can be found on http://man.cat-v.org/unix-1st/. It is worth taking a look, especially how its structure influenced the appearance of today's manual pages.

- **Also note that UNIX is roughly 10 years older than DOS.**
- The Multics system had 9 main goals, as described in the *Introduction and Overview of the Multics System* article from 1965. Most interesting goal was probably a request for uninterrupted system run.
- Multics was written in the PL/I (Programming Language #1), therefore earlier than UNIX was rewritten to C !
- Multics gained as a first system the B2 security level in 1980. For couple of years it was the only system with this security level.
- GE was founded in 1892 by merging two companies, one of which was Edison General Electric Company founded in 1879 by Thomas Alva Edison (inventor of bulb, film camera, ...); currently its subsidiaries cover many areas, including the supply of one motor type for Airbus 380 or banking.
- PDP = Programmed Data Processor. First type, *PDP-1*, was sold for $120.000 in era, when other computers cost over one million. That was a strategy of the DEC corporation – the *computer* term meant expensive machine needing a hall and team of people to operate. That's why PDP was not calling its machines computers but *PDPs*.
- PDP-11 is legendary machine of DEC corporation, gradually versions were developed from PDP-1 to PDP-16, except PDP-2, PDP-13. There are some PDP-11 systems still running today and companies that manufacture spare parts.

UNIX history, continued

- February 1973 – UNIX V3 contained the *cc* compiler (the C language was created by **Dennis Ritchie** for UNIX)
- October 1973 – UNIX presented to the public in *The UNIX Timesharing System* article in ACM conference
- November 1973 – **UNIX V4 rewritten to C**
- 1975 – UNIX V6 was the first UNIX available outside of BTL
- 1979 – UNIX V7, for many "the last true UNIX", contained *uucp*, Bourne shell; kernel size was just 40KB !!!
- 1979 – UNIX V7 ported to 32-bit VAX-11
- 1980 – Microsoft introduces XENIX, based on UNIX V7

- ACM = Association for Computing Machinery, founded in 1947. UNIX was presented by Ken Thompson and Dennis Ritchie.
- **The act of rewriting UNIX to C was possibly the most important moment in the history of this system** ⇒ it was much easier to port UNIX to different architectures.
- The legendary book *A commentary on the Unix Operating System* by John Lions was found on version 6.
- Microsoft did not sell XENIX directly, it was licensed to OEM companies (Original Equipment Manufacturer) such as Intel, SCO a others. Other companies then ported XENIX to 286 (Intel) and 386 (SCO, 1987). It is possible to find interesting information on the web describing these times and then positive attitude of Microsoft towards UNIX.
- UNIX V7 had some 188 lines of source code in circa 1100 files (determined using `find`, `wc` and `awk` across all files matching `*.[cshy]`).
- If you are interested in UNIX history, see the "unix" on Wikipedia and you will find lots of content just by clicking through.
- UNIX was multiuser system (accounts with passwords) in 1973 with multiprocessing support and process protection and paging. It had signals, pipes, hierarchical file system with mount points, file rights (user/group/other r/w/x), hard links, devices accessible as files. The kernel was written in C and its image occupied 26 Kilobytes in memory. For file operations there were `open()`, `close()`, `read()`, `write()`, `seek()`, `pipe()` system calls. For process manipulation there were `fork()`, `exec()`, `wait()`, `exit()`. In total there were 48 syscalls, out of which 35 exist till today.

UNIX divergence

- mid 70's – releasing UNIX to universities: mainly to **University of California v Berkeley**
- 1979 – **BSD Unix (Berkeley Software Distribution)** is being developed from UNIX/32V (the mentioned port to VAX) provided to Berkeley. version 3.0; last version 4.4 in 1993
- 1982 **AT&T**, owner of BTL, can enter the computer marked (forbidden till 1956) and comes with version *System III* (1982) till *V.4* (1988) – so called *SVR4*
- UNIX International, OSF (Open Software Foundation), X/OPEN, . . . are conceived
- 1991 – Linus Torvalds begun OS Linux development, kernel version 1.0 was finished in 1994

- UNIX is universal operating system working on multitude of computers from embedded and mobile systems (Linux), through personal computers till servers and supercomputers.
- UNIX V3 = *UNIX version 3*, UNIX V.4 = *system 5 release 4* etc., e.g. UNIX V3 != SVR3.
- UNIX System III is therefore not UNIX V3; in those days (late 70's) there were multiple groups in BTL that contributed to the UNIX development. The "Vx" versions were developed in *Computer Research Group*, other groups were *Unix System Group* (USG), *Programmer's WorkBench* (PWB). Another branch of UNIX was Columbus UNIX also in BT. The System III version is based on these early versions.
- UNIX has forked into two main branches: AT&T and BSD, individual manufacturers were coming with their own modifications. **Individual clones adopted features from each other.**
- System V R4 has circa 1.5 million lines of code in circa 5700 files (determined using `find`, `wc` and `awk` across files names matching `*.[cshy]`).
- Berkeley university was granted UNIX license as one of the first in 1974. During several years students (one of which was Bill Joy, later founder of Sun Microsystems and the author of C-shell) created SW package *Berkeley Software Distribution* (BSD) and were selling it in 1978 for $50. These early BSD versions contained just SW and utilities (first version: Pascal compiler, the *ex* editor), not the system or its changes. That came with the 3BSD

version. The 4BSD version was conceived in 1980 already as a project financed by the DARPA agency and led by Bill Joy. It suffered problems with insufficient performance and the tuned 4.1BSD came into existence in 1981 as a result.

- 4.1BSD should have been originally 5BSD, however after AT&T raised concerns that its customer could confuse 5BSD with System V, BSD transitioned to the 4.x BSD versioning scheme. It was common that rather write its own code, the Berkeley developers looked around first for what is already done. In this way BSD took virtual memory from Mach or NFS-compatible code developed on one Canadian university.
- The hardware manufacturers were shipping UNIX variants for their own computers and commercialization made the situation worse w.r.t. diversification of this system.
- In the 80's the first effort for standardization came into existence. Standard specifies how the system should behave externally (for user, programmer and administrator), it is not dealing with implementation. The goal is portability of applications and users. All systems remotely looked like UNIX however upon closer look there were different in many important properties. For example System V and BSD differed in filesystem, network architecture and virtual memory architecture.
- When in 1987 the AT&T and Sun Microsystems companies (whose then SunOS was based on BSD) joined their effort to develop single system that would contain the best of each, next to enthusiastic responses it also prompted fear between many other unix system manufacturers that were afraid that it would mean great business advantage for both companies. So came the Open Software Foundation into existence (do not confuse with FSF) and founding members were between others Hewlett-Packard, IBM a Digital. This system OSF/1 that arose from this partnership was not very successful and it was shipped only by Digital that renamed it to Digital UNIX. It is interesting to note that the system was based on the Mach microkernel. After the acquisition of Digital by Compaq it was renamed to Tru64 and supported by Hewlett-Packard, that was merged with Compaq in 2002. In the mean time AT&T and Sun responded by founding UNIX International. This period of 80's and 90's is called **Unix Wars** – the fight over what will be the "standard unix".
- OSF and UI became great rivals however they were soon met by unexpected opponent – Microsoft.
- (1992) 386BSD founded on *Networking Release 2*; Bill Jolitz created 6 missing files and put together functional BSD system for i386. This system was a based for *NetBSD* and *FreeBSD* (and others patterned on these systems).
- (1995) 4.4BSD-Lite Release 2, after which CSRG was disbanded. It piloted the development of BSD branch for almost 20 years. More can be found in the BSD chapter mentioned above.

The End.