Laboratorio C+Unix: Slides
Anno accademico 2018/19

Enrico Bini

December 9, 2018

Contents

1 Introduction to Unix systems

Operating System (OS)

- An operating system is the software interface between the user and the hardware of a system.
- We say that the operating system manages the available **resources**.
  - Whether your operating system is Unix-like (Linux), Android, Windows, or iOS, everything you do as a user or programmer interacts with the hardware in some way.
- the components that make up a Unix-like operating system are
  1. device drivers: make the hardware work properly
     (coded in C and assembly),
  2. the kernel: CPU scheduling, memory management, etc. (coded in C)
  3. the shell: allows the interaction with OS
  4. the file system: organizes all data present in the system
  5. applications: used by the user
     (coded in fancy languages: Java, python, or else)

Installing a Unix-like (Linux) machine

1. Download the ISO image of any Linux distribution. For example, Xubuntu:
   `https://xubuntu.org/`
2. If you want to install a Linux virtual machine over Windows/Mac, then download some virtual machine emulator. For example, VirtualBox
   `https://www.virtualbox.org/`
3. Launch the virtual machine after attaching the ISO image to the virtual disk of the virtual machine. By doing so, the virtual machine "believes" that you have inserted the Linux installation CD.

1.1 Shell

Shell

- The shell is a command line interpreter that enables the user to interact with the operating system.
- The shell is used almost exclusively via the command line, a text-based mechanism by which the user interacts with the system.
- Terminals (the "black window") allows the user to enter shell commands
- When entering commands in a terminal, the button "TAB" helps to complete
- The real hacker uses the terminal only. The mouse and the graphic interfaces are for kids: is it more efficient to use 10 fingers over a keyboard? Or one finger over a strange device?
- Exercise: open a terminal and try

```
cat /etc/shells
echo $SHELL
```

Help on commands

- Unix manual pages (or `man` pages) are the best way to learn about any given command
- man pages are invoked by "`man <command>`"
  - Space to scroll down, `b` to scroll up, `q` to quit
- `man` pages are divided in sections

Sec.	Description
1	General commands
2	System calls
3	Library functions, covering in particular the C standard library
4	Special files (usually devices, those found in `/dev`) and drivers
5	File formats and conventions
6	Games and screensavers
7	Miscellanea
8	System administration commands and daemons

- if same entry in more section, it is returned lower section
- try: `man printf`, `man 1 printf`, `man 3 printf`

1.2 File system

File system

- The file system enables the user to view, organize, store, and interact with all data available on the system
- Files have names: file extension does not imply anything about the content, it is just part of the name
- **Files** are arranged in a tree structure
- **Directories** are special files which may contain other files
- The root of the tree is "`/`"
- The **full pathname** of a file is the list of all directories from the root "`/`" until the directory of the file
- "`.`" is the current directory
- "`..`" is the parent directory
- "`~`" is the home directory of the user
- Files may be **links** to other files: command `ln` to create links

File types

- Files are an abstraction of anything that can be viewed as a sequence of bytes: the disk is a (special) file
- More in general, there are 7 types of files:
  1. (marked by "`-`" in `ls -l`) regular file: contains data, are on disk
  2. (marked by "`d`" in `ls -l`) directories: contains names of other files
  3. (marked by "`c`" in `ls -l`) character special file: used to read/write devices byte by byte (`stat /dev/urandom`)
  4. (marked by "`b`" in `ls -l`) block special file: used to read/write to devices in block (disks). Try `stat /dev/sda1`
  5. (marked by "`p`" in `ls -l`) FIFO: a special file used for interprocess communication (IPC)
  6. (marked by "`s`" in `ls -l`) socket: used for network communication
  7. (marked by "`l`" in `ls -l`) symbolic link: it just points to another file

```
stat <some-file>
df
cat /dev/sda1
```

Directory content

`/bin`	common programs, executables (often subdirectory of `/usr` or `/usr/local`)
`/boot`	The startup files and the kernel
`/etc`	contains configuration files
`/home`	parent of home directory of common users
`/tmp`	place for temporary files, cleaned upon reboot
`/root`	home directory of the administrator
`/lib`	library files
`/proc`	information on processes and resources (only on some Unix-like machines)
`/dev`	contains references to special files (disks, terminals, etc.)

Commands to navigate the file system

`cat`	Concatenate: displays a file
`cd`	Change directory: moves you to the directory identified
`cp`	Copy: copies one file/directory to specified location
`du`	Disk usage
`echo`	Display a line of text
`file`	Identifies the file type (binary, text, etc)
`grep`	Print lines matching a pattern
`head`	Shows the beginning of a file
`ls`	List: shows the contents of the directory specified
`mkdir`	Make directory: creates the specified directory
`more`	Browses through a file (has an advanced version: `less`)
`mv`	Move: moves the location of or renames a file/directory
`pwd`	shows the current directory the user is in
`rm`	Remove: removes a file
`sort`	Sort lines of text
`tail`	Shows the end of a file
`touch`	Creates a blank file or modifies an existing files attributes

Navigating in the file system

- Try navigating the file system with
  - `ls` to show the content of the current directory.
    Useful option `-a`, `-l`, `-R`, `-tr`.
    My standard command is `ls -latr`
  - `cd` to change directory
  - by pressing "TAB", all alternatives are shown (it is the most pressed button by Unix users)

- Try creating
  1. a file `tmp1.txt`
  2. a link to it with name `tmp2.txt` (with `ln`)
  3. modify `tmp2.txt` and view `tmp1.txt`
- Try creating a symbolic link to a directory with `ln -s`

Input/Output redirection

- To work properly, every command uses a source of input and a destination for output. Unless specified differently
  - the input is read from the keyboard
  - the output is written to the terminal
- Unix allows the **redirection** of the input, output, or both
  - redirection of the input from a file (with "`<`")
  - redirection of the output to a file (with "`>`")
  - redirection of the input or output to another command ("pipe" with "`|`")
- Examples:
  - `ls > my_list`
  - `wc < my_list`
  - `ls -latr | less`
  - `du -a | sort -n`

Metacharacters

- **wildcards** are special characters that can be used to match multiple files at the same time
  - `?` matches any one character
  - `*` matches any character or characters in a filename
  - `[ ]` matches one of the characters included inside the `[ ]` symbols.
- Examples
  - `ls *.tex`
  - `ls *.[tl]*`
  - `ls *t*`
  - `ls ?t*`

1.3 Accounts

Accounts

- Unix is a multi-user systems: more than one user can use "simultaneously" the available resources (computing capacity, memory, etc.)
  - Once upon a time there were single-user operating systems such as MS-DOS
  - In applications where the resources must be used by a single application, multi-user is not needed (example: embedded systems)
- **accounts** are used to distinguish between different type of usage of resources
- There are three primary types of accounts on a Unix system:
  - the root user (or superuser) account,
  - system accounts, and
  - user accounts.

All accounts

- `cat /etc/passwd` to see all accounts. Seven colon-separated ":" fields:
  1. login name
  2. crypted password (today passwords are in `/etc/shadow`, accessible only with `root` privileges)
  3. numeric user ID
  4. numeric group ID
  5. a comment field
  6. the home directory
  7. the shell program

Root accounts

- The root account's user has complete control of the system: he can run commands to completely destroy the software system as well as some hardware component
- The root user (also called root) can do absolutely anything on the system, with no restrictions on files that can be accessed, removed, and modified.
- The Unix methodology assumes that root users know what they want to do, so if they issue a command that will completely destroy the system, Unix allows it.
- People generally use root for only the most important tasks, and then use it only for the time required and very cautiously.

"With great power comes great responsibility"

- command `sudo` allows running a command as another user (even `root` if allowed)
- command `su` allows becoming another user (even `root` if allowed)

System accounts

- System accounts are specialized accounts dedicated to specific functions

  `cat /etc/passwd`

  - the "`mail`" account is used to manage email
  - the "`sshd`" account handles the SSH server
  - ...
- they assist in the running of services or programs that the users require
- they are needed because often running some services (mail, SSH, ...) requires **some** root privilege. Hence:
  - running these services with user privilege is not possible
  - running these services with root privileges is too risky
  - that's why system accounts are useful
- main access to hackers: accessible to user, but with some root privileges
- services running with system accounts **must** be super safe!

User accounts

- user accounts are needed to allow users to run applications system resources and are "protected" by passwords
- most common passwords

```
123456    qwerty    password    987654321    mynoob    666666    18atcskd2w    1q2w3e4r    zaq1zaq1
                                                                                    zxcvbn
```

- Some users may be fully trusted and the OS would like to give them the possibility to do anything
- Some others may be authorized to do only a subset of the possible actions
- How are privileges managed?

Groups

- users with similar privileges are assigned to the same **group**
- the administrator (`root`) can then manage all the users belonging to the group by simply assigning privileges to the group
- an account may belong to more than one group, if needed
- for each group, the file `/etc/group` lists:
  1. group name
  2. group password (very rarely used. From `man gpasswd`: "Group passwords are an inherent security problem since more than one person is permitted to know the password. However, groups are a useful tool for permitting co-operation between different users.")
  3. group ID
  4. list of users belonging to the group
- `groups bini` shows the groups a user belongs to

File ownership, permission

- Each "file" (which may be the disk and the terminal and other strange things) has
  - an `owner` and
  - a `group`
- Permissions are divided in three subsets:
  - `u` permissions of the user (owner)
  - `g` permissions of the users in the group
  - `o` permissions to all others
- Permissions are of three types:
  - **read** (`r`) if the file can be read
  - **write** (`w`) if the file can be written
  - **execute** (`x`) if the file can be executes ("search" permission id directory)
- Octal representation of permissions
- Examples:
  - `ls -l` to view permission (try it is `/dev/`)
  - `chmod` to change permissions of a file
  - `chown` to change owner and group of a file

2 C Language: preliminaries

2.1 Overview

C vs. Java `https://media.giphy.com/media/iedFwEvN4OZvW/giphy.gif` Founding principles of C programming:

1. Trust the programmer.
2. Don't prevent the programmer from doing what needs to be done.
3. Keep the language small and simple.
4. Provide only one way to do an operation.
5. Make it fast, even if it is not guaranteed to be portable.

Efficiency is favoured over abstraction
(no objects or fancy stuff)

- C is the standard language for: device drivers, kernel. Widely used in embedded systems (all contexts where high efficiency is a must)

How to write a C program

1. verify the presence of `gcc` by
   `gcc -v`
   on a terminal. If not installed, then
   `sudo apt-get install gcc`
2. Edit a program by a text editor (notepad, emacs, gedit on GNOME, kate on KDE, ...)
   - you should know what the editor writes into the saved file
   - sophisticated development environment "helps" you to write the code. Sometime they take decisions for you and you don't know about it
3. Compile the program by `gcc`
   - If no compilation error, execute the program
   - If errors, try to understand the errors, fix them and recompile

My first C program

1. Create and edit the following program
   `hello.c`
2. Compile it with

```
gcc hello.c
```

- By default the executable is `a.out`
- Launch it by `./a.out` (why not just `a.out`?)
- Usually we want the executable to have a name similar to the program. We do it by the "`-o`" option

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

- C comments are text between "`/*`" and "`*/`"
- Exercise: add comments and compile again
- Many dialects of C. The project must be written using the ISO C90 standard by compiling with
  `gcc -std=c89 -pedantic progetto.c -o progetto`

Basic structure of a C program

1. Pre-processor directives (`#include <stdio.h>`)
   - `#include ...` is used to add libraries
   - in the example `#include <stdio.h>` is needed to use the function `printf()`
2. Declaration of types (not in `hello.c`)
3. Declaration of global variables (not in `hello.c`)
4. Declaration of functions (not present in `hello.c`)
5. `main` function: the first function invoked at execution

Coding style

- C is powerful. C programs must be clean and understandable
- It is highly recommended to adopt a coding style
- It is suggested: the Linux kernel coding style
  (may be useful if one day you'll write kernel code)
  `https://www.kernel.org/doc/html/v4.10/process/coding-style.html`
- In short:
  - indentation made with TAB (8 characters long).
    - TAB is one byte only (ASCII character number 9). Not 8 spaces (8 bytes!!). C programmers like to be efficient and not to waste bytes, energy, ...
  - no new line before "{" (unless first brace of a function)
  - new line after "}", unless there is a continuation of the previous statement
  - do your best to stay in 80 columns
- Check example at the website

2.2 Variables

Variables

1. the **declaration** of a variable informs the compiler of the size of the variable and its type (Example: `int a;` informs that `a` is an integer)
2. the **identifier** is the "name"of the variable.
   - identifiers may be composed by alphanumeric characters and underscore "_". Cannot start with a number. Cannot be a reserved C keyword (`for`, `while`, etc.)
3. in C variables are viewed as ways to read/write to memory according to some representation (integer, floating point, etc.). The amount of memory used depends on the type of the variable.
   - A variable is **never** empty: it always has the value of the bytes in the memory
   - **Do not** assume that the initial content of a variable is zero (or else). Always **initialize it**.
4. A variable has a **value**. The value is the interpretation of the bytes in memory according the variable type

Variable types

- Possible types of C variables are:
  - `char`, `short`, `int`, `long` are integer types of incresing length
  - `float`, `double` are floating point types
  - pointers are addresses of memory (will be investigated later)
- the size (in bytes) of these types is highly machine dependent
- the operator `sizeof()` returns the number of bytes of the type
  - `sizeof(int)`, number of byte of any variable of type `int`
  - `sizeof(a)`, number of byte of the variable `a`
- Variables are declared on **top of the function** using them (no "on-the fly" declaration)
- Try to declare a variable after the `printf` in
  `hello.c`
  and compile by
  `gcc -std=c89 -pedantic hello.c -o hello`

Variable assignment, constants

- Variables are assigned by "`=`"

  ```
  a = b + 5;
  ```

- Variables may be initialized at the time of declaration

  ```
  int a = 0;
  ```

- Constants are declared by "`const`"

  ```
  const double pi = 3.14;
  ```

  - constants are normal variables (they occupy some space in memory) and cannot be assigned to any value. They keep their initial value

2.3 Input/output

Printing to the terminal

- The classic function fo print is `printf`
- It needs the directive `#include <stdio.h>` to be used
- `printf` can print strings, the value of variables and special characters
- The format is

  ```
  printf(<format-string>, <list-of-expressions>)
  ```

  `test-printf.c`
- For each expression in the list, the format string must specify how this expresion should be printed.
- Format specificators **must** be as many as the expressions

`%d`	print integer, base 10
`%o`	print integer, base 8
`%X`	print integer, base 16
`%e`	print floating point, notation `1.23e1`
`%f`	print floating point, notation `12.3`
`%s`	string of characters
`%c`	the ASCII character

- `man 3 printf` for full reference

Printing: escape character

- The `<format-string>` may contain escape characters to print non ASCII standard characters

`\n`	new line
`\t`	tab
`\"`	character "
`\'`	character '
`\\`	character \
`%%`	character %
`\uXXXX`	Unicode character coded by the 4 hex digits `XXXX`
`\UXXXXXXXX`	Unicode character coded by the 8 hex digits `XXXXXXXX`

`test-printf.c`

- Exercise: write the code that prints the sizes of all primitives types: `char`, `short`, `int`, `long`, `float`, `double`

Reading input from the keyboard Several functions to read data from the input:

- `fgets()` reads a string of characters, which can then be converted by `atoi()` or `atof()`
- `fscanf()` read formatted string (similar to `printf()`).
  - requires to know more advanced topics (pointers)
  - we will intrduce `fgets`, first
- Syntax

```
char s[80];

fgets(s, sizeof(s), stdin);
```

  - `s[80]` is a pre-allocated array of characters (**string** of characters)
  - reads a string from `stdin` (**standard input**)
  - store the string up to `sizeof(s)-1` characters into `s`
  - the string is read until EOF (end-of-file, `Ctrl+D`) or newline
  - more details on arrays in later lectures
- `man fgets`

Strings in memory, converting string into numbers

- A strings is stored as an array (sequence) of characters, terminated by the null character (`0`)
- Converting a string into an integer

```
int a;

a = atoi(s);
```

  - stores in the integer variable `a` the value represented by the string `s`
- Converting a string into an floating point number

```
double a;

a = atof(s);
```

  - stores in the floating point variable `a` the value represented by the string `s`

`test-read.c`, try with input from file

3 C: types and cast

3.1 Integers

Size of integers

- Integer types (`char`, `short`, `int`, `long`) may be:
  - `signed` representation in bytes is interpreted as number with sign: if negative in two complement (default for int)
  - `unsigned` representation interpreted as positive number
- Examples on 8 bits

binary	`signed` value	`unsigned` value
`11111111`	-1	255
`00000010`	2	2
`10000000`	-128	128
`10000001`	-127	129

- An unsigned variable is declared by
  `unsigned int a;`

Limits on integers

- List of limits

	signed		unsigned	
num. bytes	min	max	min	max
n	-2^{8n-1}	$2^{8n-1}-1$	0	$2^{8n}-1$
1	-128	127	0	255
2	-32768	32767	0	65535
4	-2147483647	2147483648	0	4294967295
8	$\approx -8 \times 10^{18}$	$\approx 8 \times 10^{18}$	0	$\approx 16 \times 10^{18}$

- the header file `/usr/include/limits.h`, which may be included by `#include <limits.h>`, contains the allowed minimum and maximum of all types

Integer constants

- In C code integer constants are
  - sequences of digits without a decimal dot "`.`"
  - if they start with "`0x`", they are interpreted in hexadecimal
  - if they start with "`0`", they are interpreted in octal
  - `char` constants may be written as `'a'` to represent the ASCII code of that character
- Best expression to write the ASCII code of the digit `n` is
  `'0'+n`
- What are the following expressions?

```
c_small = 'a'+i-1;          /* lowcase */
c_big = 'A'+j-1;            /* upcase */
c_up = 'A'-'a'+c_down;
```

3.2 Floating-point numbers

Floating point representation

- Two types for floating-point representation: `float`, `double`
- A floating-point number n is represented by
  - one bit for *sign* s of the number;
  - "biased" *exponent* e
    (biased exponent introduced to give a special meaning to $e = 0$)
  - *fraction* f, that is the sequence of digits after the "1,";

  in this order.
  Standard IEEE 754-1985
  The value of the represented value is

$$n = (-1)^s \times (1.f) \times 2^{e-\mathsf{bias}}$$

type	# bytes	# bits	# bit (e)	# bit (f)	bias
`float`	4	32	8	23	127
`double`	8	64	11	52	1023

Floating point representation: examples (`float`)

$$n = (-1)^s \times (1.f) \times 2^{e-\mathsf{bias}}$$

- Program to show representation
  `test-float.c`
  just compile it, run it, and enter a floating-point number.

  (more details on its code, later)

number	sign	exp	biased exp (e)	fractional
$1{=}1.0 \times 2^0$	`0`	0	127	`0...0`
$5{=}4+1=1.01 \times 2^2$	`0`	2	129	`0100...0`
$-0.25{=}(-1)^1 \times 1.0 \times 2^{-2}$	`1`	-2	125	`0...0`

- Special cases

number	sign	exp	biased exp (e)	fractional
0	`0/1`	—	0	`0...0`
$\infty{=}1.0/0.0$	`0`	—	255	`0...0`
$-\infty{=}-1.0/0.0$	`1`	—	255	`0...0`
`NaN`$=0.0/0.0$	—	—	255	$\neq 0$

Floating point: constants in C

- Floating-point constants are written in C with the decimal dot "."
- Examples:

```
double a;

a = 10.0;
a = .3;
a = 84753933.;
a = 918.7032E-4;
a = 4e+12;
a = 3.5920E12;
```

3.3 Type conversion

Automatic type conversion

- In expressions with operands of different types, each operand is converted in the most expressive format
- Order of expressiveness

```
char < short < int < long < float < double
```

- Example of automatic conversion in expressions

```
if (3/2 == 3/2.0) {
        printf("VERO :-)\n");
} else {
        printf("FALSO :-(\n");
}
```

- It is printed `FALSO :-(`

Conversion by assignment

- An expression assigned to a variable is converted to the type of the assigned variable
- Assignments to same type of smaller size are truncated
- Example of conversion by assignment

```
double a=1025.12;
int i;
unsigned char c;

i = a;  // i gets 1025 (fractional part truncated)
c = a;  // c gets 1    (least significant byte of int)
```

Explicit conversion: cast

- The programmer may specify a type conversion explicitly: *cast*

```
(type) expression
```

- Example of explicit conversion in expressions

```
if (3/2 == (int)(3/2.0)) {
        printf("VERO :-)\n");
} else {
        printf("FALSO :-(\n");
}
```

- It is printed `VERO :-)`
- The content of variable may be **altered** after a (explicit/implicit) type conversion

Example: type conversion

- `test-celsius.c`

3.4 Operators

Operators on numbers

- Arithmetic operators
  - `*` multiplication
  - `/` division (integer if both operands are integer)
    - at the end of the next code, what is the value of `x`?

      ```
      int a = 15, b = 6;
      x = a/b*b;
      ```
  - `%` remainder of integer division
  - `+`, `-` sum and subtracion
- Bit-wise operators
  - `~`, binary NOT
  - `&`, binary AND
  - `|`, binary OR
  - `^`, binary XOR

The shift operator

- `>>`, `<<` shift operator (fast way to divide or multiply by 2)
- the shift operator should be applied to unsigned number
- if applied to signed numbers the result depends on the architecture

Operators with conditions

- Comparison operators
  - `==` "equal to" (**WARNING**: not `=`)
  - `!=` "different than"
  - `<`, `<=`, `>`, `>=`
- Logical operators
  - `!`, logic NOT
  - `&&`, logic AND
  - `||`, logic OR
- Example of operations among conditions

  ```
  cond = x >= 3;
  cond = cond && x <= 10;
  if (cond) {...}
  ```

  more readable than

  ```
  if (x >= 3 && x <= 10) {...}
  ```

  especially when the condition is long.

Operators for assignment

- `++`, `--` increment and decrement
  - the value of `a++` is `a` and then `a` is incremented
  - when evaluating `++a`, the value of `a` is first incremented. Hence the value of the expression is `a+1`
- `=`, assignment (yes, assignment in C are expressions), the returned expression is the value being assigned
- `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, `|=`, compact assignment
  - `<expr1> = <expr1> <operator> <expr2>` can be written as `<expr1> <operator>= <expr2>`

3.5 Control constructs

Control constructs: basics

- The available constructs to control for the execution flow are:
  - `if ( expr ) <instr>`
  - `if ( expr ) <instr> else <instr>`
  - `while ( expr ) <instr>`
  - `for ( expr ; expr ; expr ) <instr>`
  - `do <instr> while ( expr ) ;`
  - `switch ( )  case:  ....`
  - `break ;`
  - `continue ;`
  - `return [ expr ] ;`
- Above `<instr>` stands for
  - an expression terminated by "`;`"
  - a control construct
  - a block with curly braces: from "{" to "}"
- The syntax `[...]` denotes an optional argument

"if" control

```
if (<cond-expr>) {
    // block TRUE
    ...
}
```

```
if (<cond-expr>) {
    // block TRUE
    ...
} else {
    // block FALSE
    ...
}
```

- "block TRUE" is executed if `<cond-expr>` is not zero
- "block FALSE", if present, executed if `<cond-expr>` is zero

Conditions of floating point numbers

Never use an "equal to" `==` (or "different than" `!=`) comparison with floating point numbers

- Floating point numbers have an approximation error
- What will it happen with the next code?

```
double d1 = 1e30, d2 = -1e30, d3 = 1.0;
printf("%lf\n", (d1 + d2) + d3);
printf("%lf\n", d1 + (d2 + d3));
```

- an equality with floating point variables should be always checked by

```
double a, b, tol;
...
tol = 1e-6;    /* relative tolerance */
if (fabs(a-b) < tol*a) { ... }
```

`while` **loop**

```
while (<cond-expr>) {
    // body of the loop
    ...
}
```

- body of the loop repeated until `<cond-expr>` becomes **zero** (which represent "**false**")
- if `<cond-expr>` is zero the loop is never executed
- if `<cond-expr>` is always non-zero (not necessarily 1), it loops forever

```
while (1) {
    // forever-loop
    ...
    break;
    ...
}
```

`for` **loop**

```
for (<expr1>; <expr2>; <expr3>) {
    // body of the loop
    ...
}
```

- more natural for looping a known number of times
- `<expr1>` is evaluated the before the first execution of the `for`
- `<expr2>` is evaluate at the beginning of every loop. If zero, then exit the `for`
- `<expr3>` is evaluated at the end of every loop
- Classic example (`n`-times loop)

```
for (i=0; i<n; i++) {
    // body of the loop
    ...
}
```

switch construct

```
switch (letter) {
case 'A':
case 'a':
        // 'a' action
        break;
case 'M':
case 'm':
        // 'm' action;
        // also 'k' action must be done
case 'K':
case 'k':
        // 'k' action
        break;
default:
        break;
}
```

4 Processing of C file

From C program to executable

- A C program (which is a text file) becomes an executable after a sequence of transformations
- Each transformation takes a file as input and produces a file in output
- `gcc` is called the "compiler", however it makes the next 4 steps (compiling is just one step)
  1. **Pre-processing**: the pre-processor syntatically replaces *pre-processor directives* (starting with "`#`", `#include`, `#define`, `#ifdef`, ...)
  2. **Compiling**: the compiler translates the C code into assembly code
     - the assembly of a different architecture may be specified ("*cross-compiling*"): it is used when the machine where the code is written is different than the machine where the code will be executing
  3. **Assembling**: the assembler translates assembly instructions into machine code or *object code*
  4. **Linking**: links to the invoked libraries are added

4.1 Pre-processor

Pre-processing

input The original C program (text file) written by the programmer

output Another text file with all pre-processor directives being replaced (still a C program)

- The pre-processor replaces text typographically
  - the input file may not be necessarily a C program
- Pre-processor directives starts with the symbol "`#`"
- Pre-processor directives are not indented: they always begin at the first character of the line
- Brief list of directives is:
  - `#define`, defines a "macro" to be replaced
  - `#include`, insert another file
  - `#if`, `#ifdef`, insert/remove portions of text depending on conditions

`#define` directive

- `#define` is used to define macros and costants.
  Classic example:

  ```
  #define VEC_LEN 80

  int v[VEC_LEN], i;

  for (i=0; i<VEC_LEN; i++) {
    /* something */
  }
  ```

  If `VEC_LEN` is changed, it is sufficient to change the value **only in one place** and not **everywhere** the length of the vector is used

- by convention macro names are always un UPPER CASE
- a macro can be defined at invocation time with `gcc -D`

  ```
  gcc -D PI=3.14
  ```

  is equivalent to add

  ```
  #define PI 3.14
  ```

  at the head of file

`#define` directive

- `#define` can be used to define parametric macros, which may seem functions but are not!!

  ```
  #define SQUARE(x)   x*x
  a = SQUARE(2)+SQUARE(3);   // after replacement 2*2+3*3
  ```

  what happens with

  ```
  #define SUM(x,y)   x+y
  a = SUM(1,2)*SUM(1,2);
  ```

  it is expanded in

  ```
  a = 1+2*1+2;    // which is 5, not 9
  ```

- macro with parameters **must always** have round brackets

  ```
  #define SUM(x,y)   (x+y)
  a = SUM(1,2)*SUM(1,2);    // expanded as (1+2)*(1+2)
  ```

`#define` directive: long macros

- `#define` macros must fit in one line!
- long definitions are possible but the character `\` must be used to break the line
- Example:

  ```
  #define  EXCHANGE(type,a,b)  {\
                                type aux;\
                                aux = a; \
                                a = b; \
                                b = aux; }
  ```

- If `v` is a parameter of a macro, `#v` is the string of `v`. Useful for printing a variable in debugging

  ```
  #define PRINT_INTV(v) printf("%s=%i\n",#v,v);
  PRINT_INTV(var1);    // printf("%s=%i\n", "var1", var1);
  ```

`#define`: macros or variables?

1. Is it preferrable using a macro in this code

   ```
   if (INPUT_VAR > 4) ...
   ```

   and compling with `gcc -D INPUT_VAR=10`

2. or

   ```
   fgets(s, sizeof(s), stding);
   input_var = atoi(s);
   if (input_var > 4) ...
   ```

- Efficiency: better using a macro, one less variable, more efficient code
- If the user needs to change the variable often at run-time, better a variable

Pre-processing: `#include`

- `#include` is used to include an external file
  - if the included file is in angular brackets
    `#include <stdio.h>`
    the file is searched in standard paths (usually `\usr\include\`)
  - if the included file is in double quotes
    `#include "my_header.h"`
    the file is first searched in current directory (used to include user-defined headers)
- `#include` is usually used to include *header files*
- when a library is used:
  - `<library>.c` contains the implementation (may be hidden to the user)
  - `<library>.h`, the header file, contains the list of available functions (visible to the user)
- usually the header file containts
  - `#define` directives
  - functions declarations
  - type declarations

Pre-processing: conditions

- portions of code may be conditionally inserted by
  - "`#if`, `#else`, `#endif`" directives

    ```
    #if int-const
      // code inserted if non-zero
    #else
      // code inserted otherwise
    #endif
    ```

  - "`#ifdef`, `#ifndef`, `#else`, `#endif`" directives

    ```
    #ifdef macro
      // code inserted if macro is defined
    #endif
    #ifndef macro
      // code inserted if macro is not defined
    #endif
    ```

- conditions of `#if` cannot be specified by C variables!! (must be evaluated at compile time, not run time)

Pre-processing: how to avoid multiple inclusions

- It may happen that a C program includes the following header files

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

- however, they both include

```
#include <features.h>
```

  which would give a "double definition" warning/error for many functions/variables

- to prevent multiple inclusions, all header file starts and ends as follows (example: `/usr/include/strings.h`)

```
#ifndef _STRINGS_H
#define _STRINGS_H
// content here
#endif
```

- try

```
less /usr/include/strings.h
```

Temporarily removing code

- the directive `#if` offers a convenient way to add and remove code
- this is useful for testing purpose

```
#if 0
  // code not inserted
#endif
#if 1
  // code inserted
#endif
```

Pre-defined macros for debugging

- To support the debugging, the following macro are predefined

`__FILE__`	string expanded with the name of the file where the macro appears; useful with programs made by many files
`__LINE__`	integer of the line number where the macro appears
`__DATE__`	string with the date of compilation
`__TIME__`	string with the time of compilation

- Standard example of debugging code to:

```
#ifdef DEBUG
#define MY_DBG printf("Arrived at line %i in file %s\n",\
                      __LINE__,\
                      __FILE__)
#else
#define MY_DBG
#endif
```

Running Pre-processor only

- By running

  ```
  gcc -E filename
  ```

  the pre-processor only is executed on `filename` and the output is written to `stdout`
  Hence, by

  ```
  gcc -E filename > after-pre-proc
  ```

  the output of the pre-processor is written to `after-pre-proc`
  `test-preproc.c`

4.2 Compiling, assembling, linking

Compiling

- After the pre-processor is run, the C program (text file) is traslated into a sequence of assembly instructions (still a text file)
- `gcc` can be stopped after the pre-processor and the compilation by
  `gcc -S ....`
- by default `gcc -S <filename>.c` saves the assembly instructions in `<filename>.s`
- Try `gcc -S test-read.c` and view `test-read.s` by a text editor
- `gcc -O2` performs some optimizations (such as loop unrolling): optimizations depends very much on the architecture
- `gcc -g` add debugging symbols (used by the debugger `gdb`)
- Try compiling by
  `gcc -S -g -O2 test-read.c -o test-read.s`

Assembling, linking

- *Assembling* is the translation from the assembly instructions (still a text file readable by a text editor) into machine code (binary file, not ASCII), also called object code
- default name is `<filename>.o` (object file)
- `gcc` can be stopped after the assembling with `-c option`
- Try

  ```
  gcc -c hello.c
  hexdump -C hello.o
  ```

- The last step of `gcc` is *linking*: the pieces of the code are linked together, possibly including calls to library functions when used

  ```
  gcc hello.c -o hello
  hexdump -C hello
  bless hello
  ```

`gcc` options

- `-E`, stop after pre-process, default output: `stdout`
- `-S`, stop after compiling, default output: `<filename>.s`
- `-c`, stop after assembling, default output: `<filename>.o`
- `-o <filename>`, to specify the output `<filename>` explicitly
- `-O`, `-O2`, `-O3`, optimize the code (loop unroll, etc.)
- `-g`, add symbols (useful for debugging)
- `-Wall`, shows all warnings
- `-D` , defines a macro
- `-I <dir>`, search directory `<dir>` before standard include directories
- just type the command `man gcc`

5 Pointers and arrays

5.1 Pointers

Pointers: declaration

- A pointer is a variable like all the others (integers, floating point numbers, etc)
- A variable "pointer" is a sequence of bits, which is interpreted as an address in memory
- Declared by specifying the type of the variable it points to

  ```
  <type> * <identifier>;
  ```

- Example

  ```
  int *pi1, *pi2, i, j;
  ```

  declares `pi1` and `pi2` as pointers to integer, while `i` and `j` are just integers.
- Usually names of pointers contain "`p`" or "`prt`"

Operations with pointers: dereferencing

- *dereferencing*: from the pointer to the variable it points to
- the unary operator `*` applied to a pointer returns the variable pointed by the pointer `p`
- `*p` is the variable pointed by `p`, which is the variable at the address `p` in memory

  ```
  int *p_a, *p_b;

  ...
  p_b = p_a;
  *p_a = 1;
  *p_b = 2;
  printf("%i\n", *p_a);  // what is the printed value?
  ```

- **Warning**: "`*`" is used to both declare a pointer and to dereference it

Operations with pointers: "address of"

- *address of*: from a variable to its address in memory
- the unary operator `&` applied to a variable returns the address of the variable
- `&v` is the address in memory of the variable `v`

```
int *p, v;  // p is pointer to int, v is int

v = 2;
p = &v;
printf("%i\n", *p);
```

Initialization of pointers

- Pointers (as all variables) must be initialized before being used
- Examples of errors

```
int a;
int *p;

a = *p;  // ERROR: p is not initialized and it points to
         //   unknown memory location. Hence, a is
         //   assigned unknown value

*p = 5;  // ERROR: we don't know where p points to.
         //   Hence, this assignment may produce a
         //   run-time error "Segmentation fault" if
         //   the memory area pointed by p is not
         //   available to the user
```

Generic pointer

- C allows to define a generic pointer by

  ```
  void * p;
  ```

  `p` is a simple address of a memory location, however no type of the pointed variable is specified
- It is possible to have

  ```
  int v=4;
  void * p;

  p = &v;
  ```

  however, it is not possible to dereference it by `*p`. The compiler doesn't know how to interpret the byte at the memory location pointed by `p`.

Null pointer

- The macro `NULL` represents a pointer (address in memory) which is invalid
- The value of the `NULL` macro is zero. After

  ```
  int * p;

  p = NULL;
  ```

all bits of the variable `p` are zero.

- a `NULL` pointers cannot be dereferenced: it does not point to any useful memory location
- **WARNING**
  - `void *` is a type of pointers
  - `NULL` is a possible value of a pointer

Casting pointers

- pointers are data types as the others
- they may be casted to other type of pointers
- pointer to different data types all have the same length: the length of a memory address
  - by casting a pointer, the address is never truncated (addresses always take the same amount of memory, regardless of the pointed data)
- `test-ptr-cast.c`

5.2 Arrays

Arrays

- An *array* is a contiguous area of memory allocated to several variables of the same type
- Arrays are declared by

  `<type> <identifier>[<size>];`

  for example

  `char s[10];`

  declares the variable `s` as an array of `10` `char` variables.
- Indices of the elements of the array span from `0` to `<size>-1`
- Elements are `s[0]`, `s[1]`, ..., `s[<size>-1]` and are stored contiguosly in memory
- The length of an array is **not** saved in the data structure
- The programmer must record the length of the array in some way
  - by storing a special character terminating the useful content (such as in `'\0'` terminated strings)
  - by recording the length in another (additional) variable

Initialization of arrays

- Array may be initialized in different ways

1. The size of the array may be unspecified and determined by the length of the initialization, as follows

   ```
   char v1[] = {'C', 'i', 'a', 'o', 0};
   char v2[] = "Ciao";
   ```

   are equivalent and create an array of 5 bytes
   (strings are arrays of characters terminated by 0)
2. If the size is specified, as in

   `int v[10] = {3, -1, 4};`

   then all following elements are set equal to zero. Hence,

   `int v[100] = {0};`

   is a convenient way to initialize all elements of the vector to zero.

Arrays and pointers

- Technically, the name of an array is a pointer

```
int v[10], *p;

v[0] = 23;  // same as *v = 23;
p = v;      // same as p = &v[0];
p[1] = 5;   // same as v[1] = 5;
```

- the difference betwen a pointer `p` and an array `v` is that
  1. the name of arrays is **constant**, it cannot be assigned to a value

     ```
     v = &v[1];   // ERROR
     v = p;       // ERROR
     ```

  2. at declaration time
     - `int v[10]` allocates a contiguous area to store 10 variables of type `int`
     - `int * p` allocates a variable `p` to store only a pointer. The area to allocate the pointed integers must be separately allocated (by `malloc` or else)
  3. `sizeof(p)` is the size of the address `p`,
     `sizeof(v)` is the size of the array `v`

Pointer arithmetics

- If `p` is a pointer, `(p+i)` is a pointer as well pointing to the address `p+i*dim`, with `dim=sizeof(*p)`
- `*(p+i)` is equivalent to `p[i]`
- Example: assuming that the following variables are declared

  ```
  int v[10] = {1, 9, 3}, *q = v + 4;
  ```

  among the following expressions, which one is correct?
  For the correct ones, what is the action taken?

  ```
  v = q++;
  *q = *(v+1);
  *q = *v+1;
  v++;
  q[4] = *(v+9);
  v[1] = *(q+8);
  (*v)++;
  ++(*v);
  *(q-3);
  ```

Examples

- `test-array.c`

5.3 Memory allocation

Memory and segmentation fault

- Segmentation fault is a common error which may happen during run time
- The segmentation fault error is signaled by the operating system when the user attempts to read/write to some memory areas where the user has no right to access to
- The following code tries to read and write everywhere
- `test-seg-fault.c`

Dynamic allocation of memory

- The OS offers the possibility to dynamically allocate memory

```
#include <stdlib.h>

void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```

  - `malloc` allocates `size` bytes in memory
  - `calloc` allocates `nmemb` elements of `size` bytes in memory and set them to zero
  - `realloc` changes the size of previously allocated area
  - `malloc`, `calloc`, `realloc` all return a `(void *)` pointer to the memory area
  - `free` frees the memory previously allocated

- standard code to allocate an array of `N` elements

```
p = malloc(N*sizeof(*p));
```

Memory must be freed

- All memory areas allocated by `malloc`, `calloc` and `realloc` must be released by `free`
- standard code to deallocate a memory area pointed by `p` is

```
free(p);
```

- A special care must be taken to free a memory area before the pointer to the area is lost

```
p = malloc(N*sizeof(*p));
...
p = &v;  // the ref to the allocated mem is lost!!
```

- To avoid forgetting to free the memory, it is recommended to write the `free` code immediately, possibly at the bottom of the file.

Static vs. dynamic allocation

- Is it better static allocation

```
int v[100];
```

- or, dynamic allocation

```
int * v;
v = malloc(100*sizeof(*v));
```

- used by same syntax: `v[10] = 412;`
- Dynamic allocation can use less memory than static allocation (static allocation requires overallocating, by dynamic allocation memory can be allocated when needed)
- Static allocation is faster since it avoids expensive system calls such as `malloc` and `free`
- Unless, gigabytes of memory is necessary, usually static allocation is preferred for efficiency
- Example of m=usage of `malloc`
  `test-malloc.c`

6 scanf, getchar, playing with memory

scanf: a printf-like method to read the input

- `fgets(...)+atoi(...)` require to invoke two functions and a preallocated string buffer
- `scanf` allows to read from `stdin` a string and stores the converted input into the pointed variable
- Standard example of usage

```
int n;

scanf("%i", &n);
```

- Input format is similar to the `printf`
- The input is read until a "white-space": space, tab, newline
- do `not` use `scanf` with "`%s`" to read a string: you may get a segmentetion fault (by writing over more than the allocated memory). `fgets` should be used to read strings
- `man scanf` for more format conversions and specifications

getchar(): read a character from stdin

- `getchar()` reads a character (1 byte) from `stdin`

```
#include <stdio.h>

int getchar(void);
```

- It returns an `int` (not a `char`!! Why?)
  - when a valid character is read, it is returned (only the least significant byte is used)
  - when the end of the file is reached (by pressing Ctrl+D, for example) a special macro `EOF`, which spans over all bytes, is returned
- Standard loop to read bytes

```
int c;

while ((c = getchar()) != EOF) {
    /* (char)c is the character to be used */
}
```

string.h: Copying memory blocks

- to copy `n` bytes from the memory pointed by `src` to the memory pointed by `dst`, we can use

```
void *memcpy(void *dest, const void *src, size_t n);
```

  - we must have access to both `*src` and `*dest`
  - troubles if two memory areas overlap
- to fill the first `n` bytes pointed by `p` with the character `c`, use

```
void *memset(void *p, int c, size_t n);
```

  - the memory area pointed by `p` must be allocated
  - `bzero(p,n)` is the same as `memset(p, 0, n)`

`string.h`: **Strings manipulation**

- to know the length of a string `s` (counting the characters until `'\0'`)

```
size_t strlen(const char *s);
```

- to concatenate string `src` to string `dest`

```
char *strcat(char *dest, const char *src);
```

  remember: `dest` **must** be allocated enough space

- to copy string `src` to string `dest`

```
char *strcpy(char *dest, const char *src);
```

  remember: `dest` **must** be allocated enough space

- to compare strings `s1` and `s2` in alphabetic order

```
int strcmp(const char *s1, const char *s2);
```

7 C: Functions

Functions

- Often there are portions of a program that are needed in several places at different times
- In C it is convenient to wrap these portions of programs in *functions*
- When you are copying/pasting portions of code, then you need a function for that code
- As in mathematics, a C function takes something in input and produces something in output
- A C function is characterized by
  - a *name*
  - a list *input parameters*
  - one *output parameter*
  - the *body* of the function
- input/output parameters may be `void`: they may not exist

Example of a function

```
int min(int a, int b)
{
        if (a<b) {
                return a;
        } else {
                return b;
        }
}
```

- "`min`" is the name of the function
- the function takes two integer parameters as input
- the function returns one integer as output
  - a `return` statement indicates the returned output
  - if the function has a `void` output, no `return` is needed
- the body of the function is between `{` and `}`

Declaration of a function

- The compiler requires that a function is declared before being used
- The *declaration* of a function is a line of code with
  - the *output parameter*
  - the *name* of the function
  - the list of the *input parameters*
  - terminated by a semi-colon "`;`"

  Notice: the compiler (step "2" of `gcc`) doesn't need to know the body of the function to compile, just its declaration!! However, to produce an executable, the linker (step "4" of `gcc`) of course needs to know the code of the body
- Example of function declaration

```
int min(int a, int b);
```

- The names of the input variables are not needed in the declaration

```
int min(int, int);
```

- function declarations are also called *prototypes* of the function

Definition of a function

- The definition of a function includes
  1. its declaration and
  2. its body
- The definition of a function has all the information of the declaration
- Why are declaration useful?
- Libraries of functions expose to the user the declaration of the functions only (in the header file, such as `stdio.h`)
- The implementation of the functions (function bodies) may be intentionally hidden to the user
- `test-declare-fun.c`

Invocation of functions

- The declaration or the definition of a function **must** appear above its first usage
  - otherwise the compiler doesn't recognize the function name
- A function is invoked by passing the parameters in accordance to the declaration

```
int min(int, int);

int main() {
    int a;

    a = min(4,-2);
}
```

- at compile time, only the function declaration is needed
- a function `fun` with `void` list of parameters is invoked by `fun()`

Passing parameters to functions

- When a function is invoked, the invocation parameters are **copied** into additional variables
- A function can use and modify the paramaters
- These modifications, however, are **not visible** outside the function

```
int mul(int x, int y) {
    x *= y;  /* we can use variable x */
    return x;
}

int main() {
    int a, b=4;
    a = mul(b,3);
    /* what is the value of b? */
}
```

Returning from a function

- the keyword `return` is used to return the value of a function
- once `return` is executed, no other statement of the function is executed
- there may be more than one `return` in the function body: the first one that is encountered is the one executed
- functions with `void` output:
  - has not `return` statement: it completes once the closing bracket "`}`" is reached
  - may have a `return;` with no value

Returning more information

- A limit of C functions is that they return only one variable
- If returning a large data is needed, then a pointer to such a data can be passed as parameter
- Example

```
void sort(int * v, unsigned int n)
{
/*
 * Sorting elements v[0],...,v[n-1]
 */
}
```

- The ponter only is copied internally to the function
- The pointed structure, however, is the original one and may be modified

Passing `const` parameters

- Sometimes it is needed to pass a large amount of data to functions (a long vector, etc)
- To avoid copying all the data onto the stack as parameters it is advisable to pass only a reference to the data (a pointer)
- In this way, however, the function may accidentally (or maliciously) modify the data
- To pass a pointer to a data structure that we don't want to modify we use the keyword `const` before the parameter
- For example (`man 3 printf`)

```
int printf(const char *format, ...);
```

8 Composite data types

Structures: declaration

- primitive data types: `int`, `char`, `double`, etc
- collection of homogeneous data: arrays
- collection of heterogeneous data: **structures**
- How to declare a structure? Example:

```
struct point {
    double x;
    double y;
};
```

- Then variables of that type are declare by

```
struct point p1, p2;
```

Structures: initialization

- Initialization by listing values within curly braces `{...}` separated by commas

```
struct info {
    int id;
    char *name;
    int age;
};

struct info el1 = {3, "Aldo", 45};
```

widely used in code.

Structures: usage

- Each field of a `struct` is referred by the "dot" notation

```
struct info {
    int id;
    char *name;
    int age;
};

struct info v1;

v1.id = 10;
```

- When structures are dynamically allocated, each field of a pointer to a struct is referred by the notation "`->`"

```
struct info * p;

p = malloc(sizeof(*p));
p->age = 35;
```

Assignment of structures

- `struct` may be assigned

```
struct info a, b;

a = b;
```

- however, they cannot by tested with the equal sign. The following code is incorrect

```
struct info a, b;

if (a == b) {
    ...
}
```

- Example:
  `test-matrix.c`

Unions

- The `union` data type is declared similarly to `struct`

```
union my_union_t {
    double f;
    unsigned long i;
};
```

- however all fields **overlaps in memory**!!
- `test-union.c`
- hence, `sizeof(<union>)` is the size of the largest field
- unions are used to store alternatives
- `union` used to save memory (especially in embedded systems)

Enumerations

- Enumerations are used to define "labelled constants"
- A labelled constant is an integer constant with a name
- Example of declaration

```
enum month {Gen, Feb, Mar, Apr, May, Jun,
            Jul, Aug, Sep, Oct, Nov, Dec};
```

- `test-enum.c`
  - The value of the first constant is set to zero unless explicitly specified by the programmer (for example, with "`Gen = 1`")
  - From the second constant, the value is incremented unless the programmer specifies explicitly another value (for example, with "`May = 2`")
- The purpose of `enum` is to improve readability of code
- variables of `enum` type are replaced by their value in the assembly code

Defining new types

- `typedef` allows defining "new" types (to rename an old type)

  ```
  typedef <old-type> <new-type>;
  ```

- Used to hide the real type used
  - good: when you do not trust who will read your code
  - bad: when you trust who will read your code (it may be complicated to go through many include files to understand the type of a variable)
- for example, `/usr/include/stdint.h` has many integer types defined which specifies the exact size of the integer
- often types are also defined by pre-processor macros with `#define`. What are the differencs?
- `typedef_define.c`

Creating a list with `struct` and `typedef`

- In C, dynamic lists are created by using
  - a `struct` for the element of the list
  - the `struct` also has a pointer to the next element

```
typedef struct node {
        int value;
        struct node * next;
} node;

typedef node* link;
```

- Example of program implementing lists
  `test-list.c`

9 C: more on operators

Conditional and comma operators

- The conditional "`?:`" is a ternary operator

  ```
  <expr1> ? <expr2> : <expr3>
  ```

  returns:
  - `<expr2>` if `<expr1>` is non-zero
  - `<expr3>` if `<expr1>` is zero
- The comma "`,`" operator

  ```
  <expr1> , <expr2>
  ```

  `<expr1>` and `<expr2>` are evaluated in this order and `<expr2>` is returned
  - sometime used in `for` loops in the first and third expressions, when more assignments or increments are needed

    ```
    int v[VEC_LEN], *p, i, somma = 0;

    for (p = v, i = 0; i < VEC_LEN; p++, i++) {
        somma = somma + *p;
    }
    ```

Precedence of operators

- Operators (such as "+" or "*") are used to combine operands and then produce expressions
- When evaluating a complex expression, in what precedence are operators evaluated?

```
a = 2;
b = 3;
c = a+a         *b;
```

- In math we know that multiplications are made before addition
- This is called **precedence** of operators
- C operators have a precedence (to be illustrated later on)

Associativity of operators

- When combining operators of the same precedence, in what order do we proceed?

```
a = 2;
b = 3;
c = 4;
d = a - b - c;
a = b = c = d;
```

- **Associativity** can be "right-to-left" or "left-to-right"

Table Precedence/Associativity 1/3 Available at `http://en.cppreference.com/w/c/language/operator_precedence`

Prec.	Operator	Description	Associativity
1	`++ --`	Suffix/postfix incr. and decr.	Left-to-right
	`()`	Function call	
	`[]`	Array subscripting	
	`.`	Struct/union access	
	`->`	Struct/union access via pointer	
2	`++ --`	Prefix increment and decrement	Right-to-left
	`+ -`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(type)`	Type cast	
	`*`	Indirection (dereference)	
	`&`	Address-of	
	`sizeof`	Size-of	

Table Precedence/Associativity 2/3 Available at `http://en.cppreference.com/w/c/language/operator_precedence`

Prec.	Operator	Description	Associativity
3	`* / %`	Mul., div., and remainder	Left-to-right
4	`+ -`	Addition and subtraction	Left-to-right
5	`<< >>`	Bitwise left shift and right shift	Left-to-right
6	`< <=`	Comparison $<$ and $\leq$ respectively	Left-to-right
	`> >=`	Comparison $>$ and $\geq$ respectively	Left-to-right
7	`== !=`	For relational $=$ and respectively	Left-to-right
8	`&`	Bitwise AND	Left-to-right
9	`^`	Bitwise XOR	Left-to-right
10	`|`	Bitwise OR	Left-to-right
11	`&&`	Logical AND	Left-to-right
12	`||`	Logical OR	Left-to-right

Table Precedence/Associativity 3/3 Available at `http://en.cppreference.com/w/c/language/operator_precedence`

Prec.	Operator	Description	Associativity
13	`?:`	Ternary conditional	Right-to-Left
14	`=`	Simple assignment	Right-to-Left
	`+= -=`	Assign. by sum/difference	
	`*= /= %=`	Assign. by prod./quot./remainder	
	`<<= >>=`	Assign. by bitwise left/right shift	
	`&= ^= |=`	Assign. by bitwise AND/XOR/OR	
15	`,`	Comma	Left-to-right

10 Pointer to pointer, array of pointers

Pointers to pointers

- C allows the declaration of pointers to pointers, for example by

  ```
  int **p;
  ```

- in this case:
  - `p` is a pointer (of type `int **`) pointing to a memory address containing a variable of type `int *`
  - `*p` is a pointer (of type `int *`) pointing to a memory address containing a variable of type `int`
  - `**p` is an `int`
- Also variables of type

  ```
  int **** p;
  ```

  are possible. Then `p` is a pointer to a pointer to a pointer to a pointer (4 times!!) to a variable of type `int`
- I never saw in the code more than 2 levels of dereferencing
- `test-ptr-ptr.c`

Array of pointers

- Pointers are variables: they can be stored in arrays as any variable
- An array of pointers is declared by

  ```
  <type> *v[<size>];
  ```

  which statically allocates an array of `<size>` pointers to `<type>`
- Example of initialization:

  ```
  char * p[] = {
          "defghi",
          "jklmnopqrst",
          "abc"
          };
  ```

  initializes:
  - a vector `p` with three pointers `p[0]`, `p[1]` and `p[2]`
  - three strings pointed respectively by `p[0]`, `p[1]` and `p[2]`
- `test-array-ptr.c`

Usage of array of pointers: command-line arguments

- When commands are invoked at the shell, they may have a sequence of space-separated "*command-line arguments*"
- Example:

  ```
  gcc my_file.c
  ```

  `gcc` is the command, `my_file.c` is the first (and only) command-line argument
- Command-line parameters can be read and used within a program
- We have been writing the `main` as

  ```
  int main() { /* body */}
  ```

  however, to read command-line arguments it must be written as

  ```
  int main(int argc, char *argv[]) { /* body */}
  ```

  - `argc`: number of space-separated strings at command line
  - `argv`: array of pointers to each string

  `test-command-line.c`

Array of arrays

- C allows the static allocation in memory of arrays of arrays (sometime called "bi-dimensional arrays")

  ```
  <type> v[<DIM1>][<DIM2>];
  ```

  allocates an array of `DIM1` *elements*.
  Each *element* is an array of `DIM2` contiguous variables of type `<type>`
- Overall, it is allocated for `v` a contiguous amount of memory of `sizeof(<type>)*DIM1*DIM2` bytes
- Example:

  ```
  int v[10][3];
  ```

  - `v` is an array of 10 arrays of 3 `int` variables
  - `v[0]` is the array of 3 `int` at position `0` in `v`
  - `v[i][j]` is the `j`-th element of the `i`-th array in `v`
- In memory all elements are contiguous: the element `v[i][DIM2-1]` is followed by `v[i+1][0]`
- The first element in memory is `v[0][0]` then `v[0][1]` and so on
- The last element in memory is `v[DIM1-1][DIM2-1]`
- **WARNING**: elements are addressed by `v[i][j]` and not by `v[i,j]`

Array of arrays: pointer arithmetics (difficult!!)

- Example:

```
int v[DIM1][DIM2];
```

- `v` and `v[i]` may be used as arrays
  - Remember: "the name of an array can be used as pointer"
  - `v` points to its first element, which is the array `v[0]` of `DIM2` `int`.
    The value of the "pointer" `v` is `&v[0][0]`
  - `v[i]` points to its first element, which is `v[i][0]`.
    The value of the "pointer" `v[i]` is `&v[i][0]`
  - `v[i][j] = *(v[i]+j) = *(*(v+i)+j)`
  - `v[i]` = `*(v+i)`
- `test-bi-array.c`

11 Error handling and environment variables

How to handle errors

- The invocation of functions may fail (example: `malloc` fails if memory is not available ...)
- The most frequently used method to inform about the failure of a function is to return an invalid value (example: `getc` returning `EOF`, `malloc` returning `NULL` ...)
- If the invalid value is returned, the calling function knows that an error happened, but doesn't know why
- To inform about the type of error the global variable

```
int errno;
```

  declared in `errno.h` is used
- When a function fails, it sets the global variable `errno` accordingly. By doing so, who called the failed function is informed
- Possible values of `errno` (error codes) are described at man pages
- the function `strerror(errno)` (declared in `string.h`) takes as input the error code and returns a string describing the error
  `test-error.c`

Environment variables

- Each process has an associated array of strings called the list of **environment variables**
- Environment variables enable the exchange of information between the program and the "environment"
- Env. variables are a way to pass parameters to the application
- Stored as name=value pair
- Example of environment variable are:
  - `HOME`: home directory
  - `LOGNAME`: user name
  - `PATH`: list of directories where executables are searched for
- The user can set environment variables by the command `export`
  `export USERVAR=4`
  The value of the variables is always a string
- they can be shown by the command
  `printenv`

Environment variables in C

- the list of environment variables can be accessed using the global variable `char **environ`.
- `environ` points to a NULL-terminated array of pointers to strings.
- the function

```
char * getenv(const char * name)
```

returns the string of the variable `name`
`test-env.c`

12 Files and file descriptors

Files and file descriptors Two different interfaces to files

1. *streams* of type

   `FILE *`

   (`FILE` is a `struct` defined in `stdio.h`) and

   - input/output is buffered to improve performance (incovenient to write on a disk byte by byte)

2. *file descriptors* of type

   `int fd;`

   a file descriptor is just an integer (which is the index in a table managed by the operating system)

   - lower level interface
   - not buffered
   - file descriptors are used for more general purpose than writing on a disk file (interprocess communication, communicate via TCP/IP, etc.)

12.1 Streams

Opening/closing a stream

- Before being used streams must be opened by

```
FILE * fopen(const char *path, const char *mode);
```

  - `path` is the path of the file to be open
  - `mode` is a string (not a character) specifying the read/write opening mode. Check
    `man fopen`
    for full description
  - a pointer `FILE *` is returned

  Example: opening `"my_file.txt"` in read mode

```
FILE * my_f;

my_f = fopen("my_file.txt", "r");
```

- After its usage, a stream must be closed by

```
int fclose(FILE * stream);
```

  if streams are not closed, the OS may be unable to open new files

Reading-from/writing-to a stream

- For each open file, the OS keeps and updates a *file position indicator*
- Reads/Writes happen at the current position

```
int fscanf(FILE *stream,const char *format,...);
int fgetc(FILE *stream);
char *fgets(char *s, int size, FILE *stream);
```

- `fscanf`, `fgetc`, `fgets` read from a stream at current position
  - `fgetc` and `fscanf` returns `EOF` if end-of-file is reached
  - `fgets` returns `NULL` if end-of-file is reached
- Functions to write to a stream

```
int fprintf(FILE *stream,const char *format,...);
int fputc(int c, FILE *stream);
int fputs(const char *s, FILE *stream);
```

- After a read/write, the position is incremented by the number of bytes read/written
- `test-file.c`

Controlling the position over a file

- The position over a file can be controlled by `fseek()`

  ```
  int fseek(FILE *stream, long offset, int whence);
  ```

  sets the file pointer of `stream` as follows:
  - if `(whence == SEEK_SET)`, position is set equal to `offset`
  - if `(whence == SEEK_CUR)`, position is moved by `offset`
  - if `(whence == SEEK_END)`, position is moved by `offset` from the end

  notice that `offset` may be negative (to move the position backward)
- To know the current position over a file

  ```
  long ftell(FILE *stream);
  ```

  - The first byte of a file is at position `0`
  - The last byte of a file is at position `<size>-1`
  - When the position is equal to `<size>`, then we reached the end-of-file

Standard streams: `stdin`, `stdout`, `stderr`

- `stdin`, `stdout`, and `stderr` are all streams (of type `FILE *`) defined by the operating system with a special usage
- `stdin` is "standard input" and it is the stream of characters entered by the keyboard
- `stdout` is "standard output" and it is the stream of characters printed on the terminal
- `stderr` is the "standard error" stream. It is used to print error messages and it is printed on the terminal as well

Redirecting error (and output) streams of commands

- To redirect `stdout` to a file, truncate if existing
  `COMMAND 1> filename`
- To redirect `stdout` to a file, append if existing
  `COMMAND 1>> filename`
- To redirect `stderr` to a file, truncate if existing
  `COMMAND 2> filename`
- To redirect `stderr` to a file, append if existing
  `COMMAND 2>> filename`
- To redirect `stdout` and `stderr` to a file, truncate if existing
  `COMMAND &> filename`
- To redirect `stdout` and `stderr` to a file, append if existing
  `COMMAND &>> filename`

Buffering of streams

- The interaction between the (fast) processor and the (slow) devices may degrade the performance
  - it is not convenient to write a single byte to the disk every time `fputc()` is invoked
- I/O may be buffered: "buffered" read/write are delayed until the "buffering" condition is true. Three types of buffering
  1. unbuffered: all I/O operations happen immediately
  2. block buffered: I/O operations are executed when the buffer is full
  3. line buffered: I/O operations are executed when newline `'\n'` read
- `stdout` is line-buffered
- `stderr` is not-buffered (normally, we want to see the error messages as soon as they happen): during debugging use `stderr`
- other files are block buffered, unless specified differently

Controlling the buffering

- To force the buffer to be written to the device

```
int fflush(FILE *stream);
```

  by `fflush(NULL)`, all open output streams are flushed.

- to change the buffering, launch `setvbuf` before any operation

```
int setvbuf(FILE *stream, char *buf, int mode,
            size_t size);
```

  `man setvbuf` for more information

- Examples

```
/* set no buffering to stream */
setvbuf(stream, NULL, _IONBF, 0);
```

12.2 File descriptors

File descriptors and files

- **Streams** (not files) are of type `(FILE *)` (the name "`FILE`" is only for historical reasons)
- **File descriptors** are of type `int` (sometime called "I/O streams")
- A file descriptor (fd) identifies a source/destination of a sequence of bytes
- File descriptors are a lower level interface than streams
  - `int STDOUT_FILENO` (declared in `unistd.h`) is the fd of the stream `FILE * stdout`
- File descriptors are more general than streams
  - all streams have a file descriptor
  - there may be file descriptors which are not streams
- File descriptors are opened by different functions depending on their usage:
  - `int open(...)` (not `fopen(...)`) binds a file in the file system to the returned descriptor
  - `int socket(...)` binds the data coming-from/going-to a UDP/TCP (and others) connection to the returned descriptor
  - `pipe(...)` creates a "pipe": two descriptors attached to each other (more details later in the course)

Opening/closing a file descriptor of a file

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

- `open(...)` opens a file and returns a fd (`man 2 open` for details)
  - `pathname`, a string with the pathname of the file
  - `flags`, specifies how to open (about 20 flags).
    Flags are set by making the bitwise OR "`|`" among the selected macros
  - Each macro has one "1" bit only (have a look in `/usr/include/libio.h`)
    - **must include** one among `O_RDONLY`, `O_WRONLY`, `O_RDWR`
    - `O_APPEND`, file opened in append mode
    - `O_CREAT`, create the file if doesn'e exist. The read/write/execute permissions of the new file are set by `mode`
    - `O_TRUNC`, if file exists, it is truncated
- After being used, file descriptors must be closed

```
int close(int fd);
```

  otherwise we may run out of available file descriptors

- `test-open-many.c`

Opening a stream from a fd and viceversa

- given an open fd, it is possible to get a `FILE *` associated to `fd` by

```
FILE *fdopen(int fd, const char *mode);
```

- given an open stream `FILE * fp`, it is possible to get its file descriptor by

```
int fileno(FILE * fp);
```

- Notice that the structure `FILE` already has a file descriptor into the `struct`
- Try the following command

  ```
  echo "#include <stdio.h>" | gcc -E - > stdio_full.h
  ```

  and search for the `struct FILE` in the produced file `stdio_full.h`

Reading from a file descriptor

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

- reads from the file descriptor `fd` up to `size` bytes and store them to `buf`
- it returns the number of bytes actually read (it may be less than `size`)
- if it returns zero, then end-of-file is reached
- if it returns -1 then an error has occured and `errno` can be inspected to know the error type. Example of errors are:
  - `EBADF`, invalid file descriptor for reading
  - `EIO`, hardware error (such as bad block)

Writing to a file descriptor

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

- write `size` bytes from the buffer `buf` to the file descriptor `fd`
- it writes immediately the data, not buffered as `fprintf`
- formatted output over a file descriptor `fd` by

  ```
  int dprintf(int fd, const char *format, ...);
  ```

- WARNING: by mixing `fprintf` and `write` to the same fd/stream you must be careful
  - `fprintf` uses a buffer to write, while `write` doesn't
  - the output written by `fprintf` may be delayed w.r.t. the output made via `write`
- `test-buf.c`

Positioning over a file descriptor

- This position over a file descriptor is controlled by `lseek()`

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

- set the file pointer of `fd` as follows:
  - if `(whence == SEEK_SET)`, position is set equal to `offset`
  - if `(whence == SEEK_CUR)`, position is moved by `offset`
  - if `(whence == SEEK_END)`, position is moved by `offset` from the end
- notice that `offset` may be negative (to move the position backward)
- File descriptors of different types (not associated to files) do not allow positioning by `lseek(...)`

13 Scope of variables

Scope of a variable

- The *scope* of a name (variable or function) is the portion of code where that name may be used
- The scope of a name (variable) declared within a function is restricted to that functions
- Parameters of functions have the same scope of a variable declared inside a function
- The scope of a name (variable) declared outside any function (*global variable*) is from the place of declaration until the end of the file
- If a variable with the same name is declared both outside and inside a function, the one inside the function prevails

Global variables

- Global variables are declared outside any function
- Global variables are visible to all functions
- Usage of global variables:
  - good: when many functions share a large amount of data, the usage of global variable is more efficient (it prevents parameter passing)
  - bad: the code relying much on global variable may be:
    1. hardly portable (functions are not really an "isolated" piece of code)
    2. hard to comprehend (one needs to search for the defintion of a global variable in many places)
- If a function uses a global variable as "parameter", it is highly recommended to add a comment on top of the function
- Names of global variable should be highly informative to avoid the reader to browse much code:
  - `number_students` is good
  - `n` is bad

14 Storage classes

14.1 Variables on the BSS

Variables on the BSS

- BSS is a read-write memory segment
- Size of BSS is decided at compile time (depending on the size of allocated variables, plus some padding for alignment)
- Two ways to allocate variables over the BSS
  1. global variables
  2. local variables declared with the `static` qualifier

```
void func(void) {
  static int my_static_var;
  ...
}
```

- Allocated: at the begin of the program
- Deallocated: at the end of the program

`static` variables within functions

- Scope: same as local variables (only within the function)
- Lifetime: same as global variables (from the start to the end of the program)
- Typical usage: to keep some state between consecutive invocation of a function

```
void func(void) {
  static int count_invocations = 0;

  ++count_invocations;
  ...
}
```

- Example:
  `test-static.c`

14.2 Variables on the stack

Content of the stack

- The stack is a memory area with LIFO (Last-In First-Out) policy
  - `push` assembly instruction stores data to top of the stack
  - `pop` assembly instruction extracts data from the top of the stack
- Main purpose is to store parameters and return address of function invocation
  - when a function is invoked (`call` assembly instruction),
    1. the parameters of the function invocations are pushed to the stack
    2. the return address is pushed to the stack
    3. then the control flow goes to the invoked function
  - when we return from function (`ret` assembly instruction))
    1. the return address is fetched from the stack
    2. the control goes back to the invoking function
- `test-stack.c`

Variables on the stack

- When variables are declared at the top of a function, they are allocated onto the stack (unless the `static` qualifier is pre-fixed)
- Variable are allocated over the stack by reducing the stack pointer as needed by the size of the variables
- Allocated: when the function is entered
- Deallocated: when returning from the function
- Hence, we cannot rely on the their initial value
- The prefix `auto` in variables declaration, such as in

```
void func(void) {
  auto int my_stack_var;
  ...
}
```

  may be used. However, since it is the default allocation it is rarely (never?) used explicitly
- `test-stack-killer.c`

14.3 Variables on the heap

Variables on the heap

- The heap is a memory area available to the program an managed by the operating system
- A program may require memory over the heap by the `malloc()`/`calloc()` system calls
- The memory area is then accessed via the memory pointer returned by the `malloc()`/`calloc()`
- Allocated: when `malloc()`/`calloc()` is invoked
- Deallocated: when `free()` is invoked (or at the end of the program)
- `test-var-alloc.c`

14.4 Variables stored in processor registers

`register` variables

- The compiler may be informed that some variable **should be** allocated to a register of the processor by adding the keyword `register` at the declaration

```
register int my_register_var;
```

- `register` variables are used for frequently accessed variables: access time to a register if 10–100 times faster than access to memory
- The number of register is limited: the compiler cannot guarantee the allocation to a register

14.5 External variables

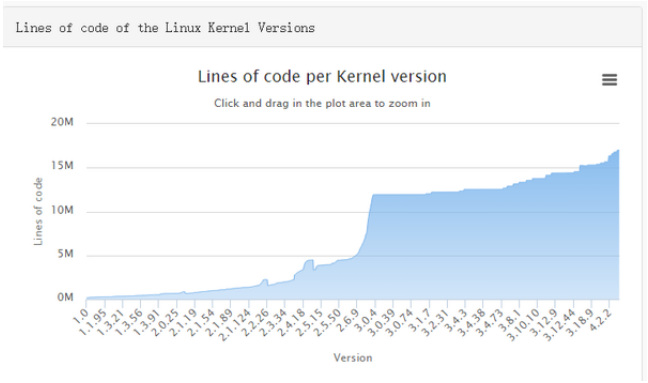
`extern` variables

- `extern` variables are allocated in other filed
  - another program
  - the operating systems
  - ...
- also functions can be `extern`. If so, they must be declared in other modules
- the compiler assumes that such a variable exists
- the linker may give an error if the variable is not found anywhere
- Examples:
  - `errno` is declared `extern`
  - `environ` is declared `extern`

15 Modules

Issues with a single long program

- Large programs (measured in number of lines or number of functions) may require many different functionalities
- Having the entire code on a single file may be problematic



- If a small modification is made on one function the entire file needs to be recompiled

Modules

- Solution: group functions and data that cover a specific functionality in a single source file (a *module*)
  - the "granularity" of a module is similar to the one of *objects* in object-oriented programming
- A module is usually composed by
  1. a header file `module-name.h` that contains
     - data types (`typedef`, `struct`, etc.)
     - function prototypes with comments that illustrate the function
  2. the source code `module-name.c` that contains the definition of the functions
- After a module is ready, it can be compiled separately by
  `gcc -c module-name.c`
  which produces the object file `module-name.o` (not an executable)
- Any program `my-program.c` that needs this module must
  1. add `#include "module-name.h"` before using any feature of the module (to inform the compiler of the content of the module)
  2. be compiled with the object of the module by
     `gcc my-program.c module-name.o`
- By doing so the module may be used, but its code is not disclosed

Example: the "`matrix`" module

- How to transform the single file program
  `test-matrix.c`
  into a module?
- The new module "`matrix`" must be composed by
  1. a header file `matrix.h` with:
     - the prototypes of the functions
     - the declaration of any needed data type
     - the declaration of any global variable
  2. the body of the module `matrix.c` with
     - the body of all functions with prototype in `matrix.h`
- The module (only) may then be compiled by
  `gcc -c matrix.c`
- Any program (such as `test-matrix-module.c`) which wants to use the module, must include only

  ```
  #include "matrix.h"
  ```

  and be compiled by
  `gcc test-matrix-module.c matrix.o`

Object dump

- `objdump` shows the content of an object file
- Examples
  1. try `objdump -d matrix.o` to see the assembly code of the module
  2. recompilie with
     `gcc -c -g matrix.c`
     and then `objdump -S matrix.o` to see source code and assembly
- `objdump` may be used also for reverse engineering on executables. Try
  `objdump -d a.out`

The `ar` utility

1. The `ar` utility is used to **ar**chive many single files in a unique one
2. In programming, `ar` is used to store many object files into a unique *library*
3. Example: show the content of an archive by

   ```
   ar t /usr/lib/x86_64-linux-gnu/libc.a | less
   ```

4. Example: extract one object file by

   ```
   ar x /usr/lib/x86_64-linux-gnu/libc.a printf.o
   ```

5. Example: show the code of the object by

   ```
   objdump -d printf.o
   ```

16 The `make` utility

Introduction to `make`

- After some source file is modified, `make` is a utility to help programmers to re-compile only the module which are affected by the modification
- When the source code of a module is modified, what must be re-compiled?
  1. the module itself
  2. all modules that use such a modified module
- Remembering the dependencies between files in a large project composed by thousands of files is somewhat complicated
- `make` is a command that provides this support

Usage of `make`

- `make <target>` tries to "make" the target
- each target has a "recepie" and needs some "ingredients" (similar to Minecraft)
- how the target can be made depends on the target
  - to make an omelette you need eggs
  - to make an executable, you need source files
  - if the file `<target>.c` is present in the current directory, by invoking
    `make <target>`
    it is executed
    `cc <target>.c -o <target>`
    `cc` is a symbolic link pointing to gcc
- targets are made according to *rules* ("recepies")
- rules to make any target are:
  1. *implicit* (they are standard. Example: "to make an executable you need a scource file")
  2. *explicit* rules are specified in a file named `Makefile` or `makefile` which is searched

Implicit rules

- Implicit rules are standard "recepies": everybody knows that to make an omelette you need eggs
- Examples of implicit rules
  1. To make an object file it is needed a source file. Try
     `make test-static.o`
  2. To make an executable, we need to compile an object file. Try
     `make test-static`
- The environment variables
  1. `CFLAGS` is used to compile (to make an object file)
  2. `LDFLAGS` is used to link objects and make an executable
- Try to set some environment variables and re-run `make`

Content of `Makefile`

- Explicit rules are written in a text files with name "`Makefile`" or "`makefile`", which is searched by `make` in the current directory
- Explicit rules explain how to make non-standard, project-dependent "recepies": how would `make` know how to make your own project?
- The format of an explicit rule is:

  ```
  target : ingredient1 ingredient2 ingredient3
  [TAB]   recepie-to-make-target
  ```

  The recepie **must** start after the TAB character (not 8 spaces)
- Everything from the charater `#` until the end of line is a comment
- If the `target` is more recent than all its ingredients, then the recepie is not made
- Example of `Makefile` for the module "matrix"
  `Makefile`

17 Process control

17.1 Process creation

Processes

- A process is an instance of an executing program
- In operating systems, a process is identified by a Process ID (PID)
- The command `ps` is used to view information on the processes
  `man ps`
- the command `top` shows a live update of CPU/mem consumed by processes
  `top`
- the command `kill` can send a "signal" to a process. One special signal is `SIGKILL` (more details on signals, later on)
  `kill -KILL <some-PID>` or `kill -9 <some-PID>`
- the command `kill` can also be used to stop or continue a process
  1. start a candidate process (a browser)
  2. get its PID
  3. `kill -STOP` PID
  4. try to use that application
  5. `kill -CONT` PID
  6. the application should be back to life

Process ID and Parent Process ID

- Processes are identified by PIDs. The system call

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

  returns the PID of the calling process (`pid_t` is an integer type)

- each process has a parent: the process that created it. The function

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

  returns the PID of the parent process (parent's PID = PPID)

- the PPID of each process represents the tree-like relationship of all processes on the system. The parent of each process has its own parent, and so on, going all the way back to process "`init`" (with PID=1), the ancestor of all processes

- to see the tree of all processes, try

  ```
  ps axjf | less
  ```

  (btw, the number of options of `ps` is uncountable)
  `test-getpid.c`

Process creation: `fork()`

- The `fork()` syscall allows a process (called "parent") to create a "child" process
- The child process is a copy of the parent
  - the OS makes a copy of all process memory of the parent: stack, BSS, and heap segments, I/O buffers included!!
  - the child executes over the copy: data modified by the child is not seen by the parent!!!
  - (sharing data among processes is possible via different methods)
- "fork": the parent process is split in two "branches"
- **SUPER IMPORTANT:** `fork()` returns two different values in child and parent processes!!!
  - in parent: the PID of the child on success (or `-1` on error)
  - in child: it returns `0`

Is the child or parent code?

- A frequent difficulty is in understanding what code we are writing: child? parent? both?

```
/* Executed only once */
if (fork()) {
    /* Executed by parent only */
} else {
    /* Executed by child only */
}
/* Executed twice: by both parent and child */
```

- Remember, the returned value of `fork()` is used to determine what process "we are":
  1. if returned 0, then we are in the child code
  2. if returned a positive number, we are in the parent code (and the value is the PID of the just created child)

`test-fork.c`
`test-fork-buf.c`
`test-fork-for.c`

Sequential programming is lost!

- We are used to programs that run a sequence of instructions
- After `fork()`, the sequence which is actually running is determined by the **scheduler**, which is not under the control of the application programmer
  - The program must then be correct, **regardless** the order of execution of processes
  - If some ordering among processes is needed to ensure correctness, then a kind of synchronization is needed
- Very difficult to write concurrent programs. Hence, it is recommended to follow this practice:
  1. do not start writing code immediately
  2. first think about your solution: how many processes? How do they communicate?
  3. write on paper your ideas and then
  4. write small portions of code to be fully tested
  5. expand the code small step by small step

17.2 Invoking an external executable (`execve`, `system`)

Replacement of a process with `execve()`

```
#include <unistd.h>

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

- `pathname`, filename to be launched;
- `argv`, arguments passed to the launched program (`NULL`-terminated list)
- `envp`, variables of the environment of the launched program (`NULL`-terminated list)
- if successful, `execve()` does not return, otherwise returns -1 with `errno` set accordingly

Effect/usage of `execve()`

- the PID is preserved
- stack, data, heap of the calling process are replaced by the ones of the program called by `execve()`
- can be used to create child processes that execute a given program

```
switch(fork()) {
  case -1:
    /* Handle error */
  case 0:
    /* preparation of the child environment */
    execve("child_command", child_args, child_env);
    exit(EXIT_FAILURE);
  default:
    /* parent code */
}
```

Launching a command with `system()`

```
#include <stdlib.h>

int system(const char *command);
```

- `system()` calls an arbitrary shell command
- `system` cerates a child process and then it waits for its termination
- if things go right, the system call `system` returns the exit status of the invoked command
- otherwise, some errors
  `man system`
- `test-execve.c`

17.3 Process termination, waiting for child termination

Process termination

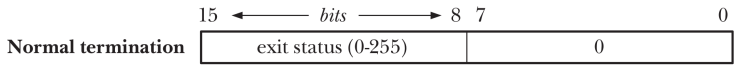
- When a process terminates
  1. all streams are flushed, and all file descriptors are closed
  2. a `SIGCHILD` signal is sent to the parent (more info about signals later)
  3. any child of the terminated process is assigned a new parent (the granparent or `init` PID=1, depending on the OS)
  4. the resources (memory, open file descriptors) are released
  5. the `exit` status truncated to 8 bits (`& 0xFF`) is stored
- A process can be terminated by
  1. the `exit(status)` system call
  2. the reception of a signal (example: sent by pressing `Ctrl+C`)
- the returned value `status` gives information about the outcome of the program. It must be between 0 and 255
- two macros (defined in `stdlib.h`) may be used:
  - `EXIT_SUCCESS` (usually 0)
  - `EXIT_FAILURE` (usually 1)

Parent `wait()` child

- The parent "can" wait for the termination of any child process by invoking the system call
  `pid_t wait(int *status)`
- A process invoking
  `child_pid = wait(&child_status);`
  checks if any child has terminated
  - if the process has no child, `wait()` returns -1 and `errno` is set to `ECHILD`
  - if the process has some terminated child, `wait()` immediately returns the PID of any terminated child and eliminate this child process from the list of children
  - If child processes exist, but none of them has terminated yet, the parent process moves to the "waiting state", waiting for the first child to terminate

Format of returned child `status`

- Once the parent correctly returns from `wait(&status)` (meaning that a child has terminated), the variable `status` is filled with information about the child process
- the format of status is as follows



- there is a more comprehensive way to wait for child processes by `waitpid()` (iilustrated next)
- the macro `WEXITSTATUS(status)` extracts the status from the value `status` written by `wait(&status)`
- `test-fork-wait.c`

Orphans and zombies

- The parent and the child processes are different and will terminate at different instants

1. If a parent terminates before the child, all child processes become *orphans*. A new process (either `init` with PID 1 or the granparent, depending on the OS) will adopt all of them
2. If a child terminates all resources are released, but an entry in the process table is kept with
   - its PID
   - its exit status, and
   - the statistics of the used resources
3. This entry in the table is hold by the OS until the parent executes a `wait()`.
4. Between the termination of the child and the execution of `wait()` by the parent, the child process is a "zombie".

Zombies

- A zombie cannot be killed by any signal (not even `SIGKILL`). This makes sure that the parent can access the exit status by a `wait()`
- If the parent terminates without executing a `wait()` all its terminated child processes (zombies) are "adopted" by another process (depending on the OS), which immediately executes as many `wait()` as needed to make the zombies R.I.P.
- An excessive number of zombies may fill the process table up, and prevents the creation of new processes
- Since zombies cannot be killed, the only way to remove them from the system is to kill the parent, which will trigger their adoption and the consequent `wait()`, which finally erases the zombies from the process list
- Why doesn't the OS free the child processes as soon as they terminate?
  `wait()` is a basic synchronization mechanism: parent is blocked until the termination of a child process

Waiting for a specific child process

- We showed that by calling `wait()` a parent waits for the completion of any child process
- To wait for a specific child process, the next system call can be used

```
pid_t waitpid(pid_t pid, int *status_child, int options)
```

- After the call to `waitpid(...)`, the parent process waits for the termination the child process `pid`. If `pid == -1`, it waits for any child process.
  `waitpid(-1,&status_child, 0)` is equivalent to
  `wait(&status_child)`
- The returned value is:

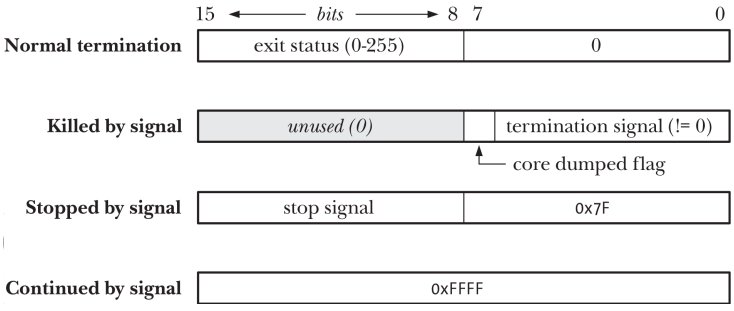
- the PID of the process whose status is reported;
- -1 if an error occured. If so, `errno` has the following values:
  - `ECHILD` no child with PID `pid` to wait for,
  - `EINVAL` invalid `option` argument

Options of `waitpid()`

- The following options can be specified as bitwise OR (`|`) of the following flags
  - `WNOHANG` "Wait NO HANGing": if no child process has terminated, `waitpid()` will **not wait** for the termination of the specified `pid`. Rather it continues, and it returns `0` to indicate this condition
    - `waitpid(ch_pid, &ch_stat, WNOHANG)` just checks termination, doesn't block the parent process if child process `ch_pid` isn't terminated
    - the actual termination of a child process can then be handled by catching the signal `SIGCHILD`, sent to the parent any time a child process terminates
  - `WUNTRACED`: `waitpid()` returns also if the selected child processes have stopped (by some signal)
  - `WCONTINUED`: `waitpid()` returns also if the selected child processes have continued (by some signal) after they were stopped

Extracting information from the returned status

- Once it is returned from `wait(&status)` or `waitpid(pid,&status,options)`, the value of `status` is filled and gives information about the status of the child



- macro exists (declared in `sys/wait.h`) to extract this information
  `man 2 wait`
  `test-fork-waitpid.c`

18 Signals

Signals

- **Signals** are software interrupts delivered to processes.
- Signals can be generated by user, software, or hardware events
- Example of signals are:
  - `SIGFPE` "Floating Point Exceptions" such as division by zero
  - `SIGILL` trying to execute an "Illegal instruction"
  - `SIGINT` used to cause program interrupt (`Ctrl+C`)
  - `SIGKILL` causes immediate program termination
  - `SIGTERM` polite version of terminating a program (`SIGTERM` can be handled by the user)
  - `SIGALRM` received when a timer (set with `alarm(int seconds)`) has expired
  - `SIGCHLD` sent to a parent when a child terminates
  - `SIGSTOP`/`SIGCONT` stop/continue a process
  - `SIGUSR1`/`SIGUSR2` user-defined signals

  `man 7 signal` for a full list
  `test-signal-all.c`

Sending a signal to any process

```
#include <sys/types.h>
#include <signal.h>

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

- `signum`, the ID of the signal
  - by sending the signal 0 (*null signal*) we can test the existence of a `pid`
    - if `(errno==ESRCH)`, `pid` doesn't exist
    - if `(errno==EPERM)`, `pid` exists but no permission to send signals
    - if successful, `pid` exists and we can send signals
- `pid`, the target process
  - if -1 the signal is sent to all processes for which the calling process has permission to send
- `test-signal-kill.c`
  save all you data, invoke it, and say goodbye...

Sending a signal to myself

1. `raise(signum)`

```
#include <signal.h>

int raise(int signum);
```

   to send signal `signum` to myself now

   - equivalent to

```
kill(getpid(), signum);
```

2. `alarm(int sec)`

```
#include <unistd.h>

unsigned int alarm(unsigned int sec);
```

   to send `SIGALRM` to myself after `sec` seconds

   - returns the seconds remaining until any previously scheduled alarm was due to be delivered, or zero if there was no previously scheduled alarm

Signal handler: default action Each signal has a default handler (`man 7 signal`)

- **`Term`**, terminate the process.
- **`Ign`**, ignore the signal.
- **`Core`**, terminate the process and dump "core". The core dump file contains the image of the process memory at the time of termination and can be used by a debugger (such as `gdb`) to inspect the causes of termination
- **`Stop`**, stop the process
- **`Cont`**, continue the process if it is currently stopped.

Signal handler: user-defined action

- The user-defined signal handler is a function that is called **asynchronously** at the time of signal delivery, wherever in the code this happens
- At the time of signal delivery
  1. the state of the process is saved (registers, etc.)
  2. the function of the handler is executed
  3. the state of the process is restored
- The signal handler is very similar to an interrupt handler
- **synchronous** signal delivery is possible
  - the process may want to know the precise moment of signal delivery and wait for it, if needed

  out of the scope of this course

Signal handler: definition of `sigaction`

- The user may set a signal handler by the `sigaction()` system call

```
#include <signal.h>

int sigaction(int signum,
              const struct sigaction *act,
              struct sigaction *oldact);
```

1. `signum`, the number of the signal to be handled
2. `act`, new handler of the signal, if `NULL` handler unchanged
3. `oldact`, pointer to the old handler, if `NULL` no handler returned

- WARNING: `sigaction` is both a sys call and a `struct`

```
struct sigaction new, old;

sigaction(signum, &new, NULL); /* set new handler to new */
sigaction(signum, NULL, &old); /* current handler in old */
sigaction(signum, &new, &old);  /* do both */
```

Format of the `sigaction` structure

- Signal handlers are specified by a `sigaction` data structure
  `man 2 sigaction`

```
struct sigaction {
  void     (*sa_handler)(int signum);
  sigset_t sa_mask;
  int sa_flags;
  /* plus others (for advanced users) */
};
```

1. `sa_handler`, function handling the signal. It must be declared as

   ```
   void signal_handler(int signum);
   ```

   Technically, the field `sa_handler` is a pointer to a function
2. `sa_mask`, mask of signal blocked during the execution of the handler
3. `sa_flags`, bitwise OR-ed flags modifying the behavior of the signal
   `test-signal-handle.c`

User-defined signal handlers 1/2

1. user-defined signal handlers must be attached to the corresponding signal **before** the signal may be released (for example, if `SIGCHLD` is going to be handled, first attach the handler to the signal, then invoke `fork()`)
2. often a single function handles many signal and a `switch(signum)/case` selects the proper action for the signal

```
void handle_signal(int signum) {
// signum, signal that triggered the handler
  switch (signum) {
  case SIGINT:
    /* handle SIGINT */
    break;
  case SIGALRM:
    /* handle SIGALRM */
    break;
  /* other signals */ } }
```

User-defined signal handlers 2/2

1. user-defined signal handlers of parents are inherited by child processes
2. user-defined signal handlers are **cleared** after `execve`
3. global variables are visible:
   (a) both in the handler code (executed asynchronously upon the reception of a signal)
   (b) and in the rest of the code (executed according to the "normal" flow of the program)
   - good: global variables offer a way to inform the "`main`" program of the occurrence of signals
   - bad: special care must be taken when invoking functions that use global variables

Global variables and signal handlers

- some functions (including `printf(...)`) use global data structure. The asynchronous arrival of signals may corrupt this data and produce unexpected behavior
  - `man 7 signal` >> "Async-signal-safe functions":
    "If a signal interrupts the execution of an unsafe function, and handler calls an unsafe function, then the behavior of the program is undefined"
- functions that can be safely used within a handler are marked by "AS-Safe" (Asynchronous Signal-Safe) in the GNU libc documentation: `write()` is AS-Safe, `printf()` isn't, etc.
- `errno` is a global variable, which may be overwritten within the signal (if any function writing to `errno` is used). It is recommended to save it and restore it at the end of the handler

Signals lifecycle

- Signals are *generated* by hardware or software events and are sent to a process
  1. signals may be *blocked*
  2. if the signal is blocked, then it remains *pending*, until it is *unblocked*. As soon as unblocked, it is immediately *delivered* to the process,
  3. if a signal is generated again while it is already pending, then it is *merged*: after unblocked, it will be delivered only **once**!!
  4. if the signal is not blocked, it is immediately *delivered* to the process.
- Once a signal is delivered, a process can *handle* it in several ways:
  1. ignore the signal,
  2. perform the default action, or
  3. perform a user-defined action (specified by the `sigaction` system call).

Setting signal masks

- During its execution, each process has its own *signal mask*: a newly created process inherits the parent's mask
- The signal mask is the collection of signals that are currently *blocked*
- The signal mask of a process can be updated by the system call `sigprocmask(...)` (details later)
- Signal masks are of type `sigset_t`. Functions to manipulate sets:

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

`man sigsetops` for more details

Signal mask of a process

- `sigprocmask(...)` is used to set the signal mask of a process

```
#include <signal.h>

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

- `oldset` is the old mask
- the new mask is set according to:
  - if `(how==SIG_BLOCK)`, then signals in `set` are blocked
  - if `(how==SIG_UNBLOCK)`, then signals in `set` are removed from the existing mask
  - if `(how==SIG_SETMASK)`, then `set` becomes the new signal mask
- `test-signal-mask.c`

Signal mask during a handler

- The signal that triggered the handler is masked during the handler
  - unless the flag `SA_NODEFER` is set
- `sa_mask` field of `sigaction` sets the mask during the handler
  - when the handler returns, the set of blocked signals is restored to the value before its execution, regardless of any manipulation of the blocked signals made in the handler

```
struct sigaction sa;
sigset_t  my_mask;

sa.sa_handler = handle_signal; /* the handler */
sa.sa_flags = 0;         /* No flags */
/* Set an empty mask */
sigemptyset(&my_mask);
/* Add a signal to the mask */
sigaddset(&my_mask, signal_to_mask_in_handler);
sa.sa_mask = my_mask;
/* Set the handler */
sigaction(SIGINT, &sa, NULL);
```

Signal masks: why

- Signals arrive asynchronously and may interrupt the program execution at any time
- The programmer must take special care in preventing signals to interrupt the execution in places that leave the status in an inconsistent status
  `test-signal-non-atomic.c`
- type `sig_atomic_t` is an integer and it is guaranteed by the compiler to be accessed atomically (by a single assembly instruction)
- In practice, we can assume that
  - `int` and
  - pointers

  are atomic
- To mask or not to mask signals?
  1. to mask to avoid inconsistent data
  2. not to mask to increase responsiveness and reduce the risk of signal merge

Merged signals

- If a signal is generated while it is still pending for being handled, the newly generated and the pending one are *merged* into one
  - the presence of a pending signal is stored by a flag only, not by a number (of pending signals)
  - a signal handler cannot be reliably used to count the number of collected signals!
- `test-signal-merge.c`

Unblocking the delivered signal during its own handler

- When a signal `signum` is delivered to a process, during its handler, the signal `signum` is automatically blocked until the handler returns
  - **unless** the flag `SA_NODEFER` is set when setting the handler as in

```
void handle_signal(int signal);

int main() {
  struct sigaction sa;
  sigset_t  my_mask;

  sa.sa_handler = &handle_signal;
  sa.sa_flags = SA_NODEFER;  /* allow nested invocations */
  sigemptyset(&my_mask);     /* do not mask any signal */
  sa.sa_mask = my_mask;
  sigaction(SIGUSR1, &sa, NULL); /* set the handler */
```

- `test-signal-nodefer.c`

Pausing, sleeping

1. `pause()`

```
#include <unistd.h>

int pause(void);
```

   pause the process until a signal is caught

2. `sleep(sec)`

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

   the process sleeps for `sec` seconds.
   If the process caught a signal while sleeping, then `sleep` returns the remaining second to sleep

Delivery of signals to a suspended process

- Signal handler executes asynchronously:
  1. the state of the process is saved (registers, etc.)
  2. the function of the handler is executed
  3. the state of the process is restored
- What happens when a signal is delivered to a suspended process?
  - "suspended process": process waiting on `wait()`, suspended on `pause()` or `sleep()`. Counted as "sleeping" in `top`

  1. the process is not executing, since it is suspended on some system call
     - its state is already saved
  2. the function of the handler is normally executed
  3. upon the return of the handler, **two** possible behaviors:
     - the system call returns an error with `errno` set equal to `EINTR`, **OR**
     - the system call is automatically restarted
- Which behaviour among the two depends
  - on the OS and
  - on the flag `SA_RESTART`, in the system call `sigaction`
- check `man 7 signal` and the `man` page of the suspending system call (`sleep(...)`, `wait(...)`, etc.)
- `test-signal-when-susp.c`

"Synchronization" by `sleep`

- Scenario: parent process must wait that child completes some work
- Here's a temptating (wrong) code to synchronize the two processes

```
...
if (fork()) {
    /* PARENT */
    sleep(10);
    /* now the child has finished */
} else {
    /* CHILD */
    /* do my work before the parent check */
}
```

- why wrong?
  1. we don't know is the child will take less than 10 seconds
  2. the OS may decide not to schedule the child for more than 10 seconds
- `sleep(sec)` only guarantees that the process sleeps for `sec`
- `sleep` is used mostly to show output slowly to the user

19 System V: Inter-Process Communication (IPC)

Inter-Process Communication (IPC)

- Processes may communicate via IPC *objects*
  - *message queues* allow processes to send and receive messages
  - *shared memory* allows processes to view a common area of memory where all processes can write/read
  - *semaphores* enable only a few process to access a shared resource or enable synchronization
- IPC objects are implemented by (at least) two standards:
  1. System V, older standard: first released by AT&T in 1983,
  2. POSIX, more recent standard inspired by System V, rapidly spreading and adopted by many

  They are both available in many Unix-systems, such as Linux
- The course will adopt System V standard, in the future we may switch to POSIX
- Some documentation can be found at
  `http://www.tldp.org/LDP/lpg/node7.html`
  (Warning: it is dated 1995!)

IPC objects: persistence

- IPC objects are *persistent*: they survive in the kernel space even after all the processes (creating or accessing the object) have terminated
  - this is good: IPC objects enable the communication between
    1. processes that just know the "name" of the IPC object (such as in FIFOs)
    2. even different invocations of the same executable
  - this is bad: it is worse than forgetting to `free` a dynamically allocated memory (by `malloc`)
    * if not explicitly removed (when needed), they can quickly fill up the memory
- It is possible to create, list or erase current IPC objects at command line
  - the command `ipcmk` creates an IPC object (only Linux, not standard)
  - the command `ipcs` shows the status of current IPC objects
  - the command `ipcrm` erases the specified IPC object

IPC objects: IDs and keys

- Any System V IPC *object* (message queue, shared memory, or semaphore) is identified by a unique identifier (ID) of type `int`
  - uniqueness is per type: there may be two IPC objects of different type with the same ID
- Processes that are willing to use the same IPC object for communication, must both know its ID
  1. Processes may get the ID from a common *key*
- For each type of IPC object the `???get()` function returns the ID from a key

1. `int msgget(key_t key, ...)` to get a message queue
2. `int shmget(key_t key, ...)` to get a shared memory
3. `int semget(key_t key, ...)` to get a semaphore

- How can two processes agree on a key?
  1. the key can be hard-coded (via `#define`)
  2. the key can be `IPC_PRIVATE` to create a new object (not really private since it may be shared, unfortunate choice of name)
  3. the key can be `getppid()`, if object shared among siblings

Getting an IPC object from a key

- All three types of IPC objects have a similar method to get the ID
- Any process accessing the object should call the `???get()` functions to get the ID, unless the ID is inherited from parent by `fork()`

```
int msgget(key_t key, int flags);
int shmget(key_t key, size_t size, int flags);
int semget(key_t key, int nsems, int flags);
```

- the IPC object identifier associated to `key` is returned. It may be:
  - the ID of a new object just created by calling `???get()`
  - the ID of an existing object previously created by others

  Check next slide for the precise behaviour
- `flags` specifies the read/write permissions of user/group/others in the standard octal form
  - `0400` read to user
  - `0020` write to group
  - `0666` read/write to everybody
  - ...
- Also `flags` may include macros in bitwise OR

Four ways to "get" an IPC object

1. setting `key` equal to `IPC_PRIVATE`, read/write for user

   ```
   id = msgget(IPC_PRIVATE, 0600);
   ```

   - a **new** object created ("`PRIVATE`" is misleading: other processes may use it if they know the ID)
2. setting a specific `key`, r/w for user, only read for everybody else

   ```
   id = msgget(key, 0644);
   ```

   - the ID of the **existing** object associated to `key` is returned
   - `-1` returned and `errno=ENOENT` if no IPC object exists with `key`
3. setting a specific `key`, `IPC_CREAT` flag is set, r/w for user and group

   ```
   id = msgget(key, IPC_CREAT | 0660);
   ```

   - if IPC object with `key` **exists**, same as `msgget(key, flags)`
   - if IPC object with `key` does **not exist**, it is created

4. setting a specific `key`, and `IPC_CREAT`, and `IPC_EXCL` flags are set

   ```
   id = msgget(key, IPC_CREAT | IPC_EXCL | flags);
   ```

   - if IPC object with `key` **exists**, return `-1` and `errno=EEXIST`
   - if IPC object with `key` does **not exist**, same as

     ```
     id = msgget(key, IPC_CREAT | flags);
     ```

20 System V: message queues

Lifecycle of a message queue

1. A message queue Q is created by process A
2. Q is opened for being used (send/receive) by processes $P_1, \ldots, P_n$
3. Processes $P_1, \ldots, P_n$ send and receive messages over Q as needed
   - sent messages are enqueued to the tail
   - received messaged are searched from the head (may pick messages other than the first one)
4. Sender processes send messages even if nobody will ever receive them
5. Receiver processes cannot know when senders have finished
6. Once sender processes have finished sending their messages, the message queue will persist in the kernel and receiver processes remain blocked waiting for a message until the queue exists
7. It is necessary to determine correctly the condition that allows the deallocation of the message queue

Example of usage of a message queue

- Many processes (even of different users) receiving, modifying and sending a message over the queue
  `test-rcv-snd.c`
- One process creating the queue and sending the first message
  `test-msg-start.c`

Creating/accessing a message queue

- The system call

  ```
  int msgget(key_t key, int msgflag);
  ```

  returns the identifier of a message queue associated to `key`
  - `msgflag` is a list of ORed ("`|`") options including:
    * read/write permissions (least significant 9 bits) in standard octal form
      (the "execute" (x) permission is ignored)
    * `IPC_CREAT`: if queue exists, its ID is returned; if it doesn't exists, it is created
    * `IPC_EXCL` (used only with `IPC_CREAT`): the call fails (with `errno=EEXIST`) if the queue exists
- Message queues are persistent objects: they will survive to the death of the creator, they must be erased expicitly

Message format

- Messages **must** start with a `long` value: the type of a message
  - the type **must** be strictly positive (not zero)
  - the type can be used to select messages to be read
- For example, the default message structure

```
struct msgbuf {
  long mtype;           /* type of message */
  /* my personal message goes here */
};
```

- The user can define any message structure as long as:
  1. the first `sizeof(long)` bytes are reserved to the message type
  2. the total length of the message does not exceed the maximum
     `cat /proc/sys/kernel/msgmax`
- Messages of length 0 are also acceptable. If so only `sizeof(long)` bytes are sent
- Do not use pointers in a message: pointers live into the memory of a process. A pointer written by another process does not make sense

Sending a message to a queue

- Messages are sent by the `msgsnd()` system call

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

- The caller must have write permissions on the queue to send a message
  - `msqid`, the ID of the message queue where the message is sent
  - `msgp`, pointer to the message structure
  - `msgsz`, size of the message content (excluding `sizeof(long)` bytes of the heading type)
- If queue is full
  - the call `msgsnd()` blocks until some space for the message is made, or
  - if flag `IPC_NOWAIT` is set, it returns `-1` with `errno = EAGAIN`

Receiving a message

- To receive a message from the queue `msqid` and copy it to the buffer pointed by `msgp` the `msgrcv()` system call is used

```
int msgrcv(int msqid, void *msgp,
           int msgsz, long mtype, int msgflg);
```

- Process must have read permissions on the queue to receive a msg
- `msgsz` is the size of the message (without type) copied to the buffer
- The received message is selected as follows:
  - if `(mtype == 0)`, the first message in the queue is selected
  - if `(mtype > 0)`, the first message of type `mtype` is selected
  - if `(mtype > 0)` and `MSG_EXCEPT` flag is set, the first message of type **different than** `mtype` is selected
  - if `(mtype < 0)`, the first message in the queue of the **lowest** type less than or equal to `mtype` is selected (low types have a high priority)

- If no message is selected
  - the call `msgrcv()` blocks until a selected message arrives, or
  - if flag `IPC_NOWAIT` is set, it returns `-1` with `errno = ENOMSG`
- the received message is erased from the queue (unless `MSG_COPY` flag)

Errors on sending/receiving messages

- Both `msgsnd()` and `msgrcv()` may fail and return `-1`
- The error code `errno` is as follows:
  - `EACCES`: no permission to operate
    - tried `msgsnd()`, but no write permission
    - tried `msgrcv()`, but no read permission
  - `EIDRM`: the message queue was removed (see later how to remove)
  - `EINTR`: the process caught a signal while sleeping
    - on full queue for `msgsnd()`, or
    - no selected message available for `msgrcv()`
  - `ENOMEM`, `E2BIG`: system limits reached

Controlling (and deleting) a message queue by `msgctl()`

- The system call `msgctl()` enables several actions to be performed on the message queue

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

  - `msqid`, is the ID of the queue
  - `cmd`, describes the action to be taken over the queue
  - `buf`, is a parameter for the action (see next for details)
- To remove and deallocate the queue `msqid`

```
int msgctl(int msqid, IPC_RMID, NULL);
```

  - after the queue is removed, processes blocked on `msgrcv()` on the queue `msqid` will be unblocked with `errno=EIDRM`

Controlling (and deleting) a message queue by `msgctl()`

- To get the status of the queue

```
int msgctl(int msqid, IPC_STAT, struct msqid_ds *buf);
```

it will return the data structure of the queue (`man msgctl`)

```
struct msqid_ds {
  struct ipc_perm msg_perm; //Owner,permission
  time_t msg_stime;       //Time of last msgsnd
  time_t msg_rtime;       //Time of last msgrcv
  time_t msg_ctime;       //Time of last change
  msgqnum_t msg_qnum;      //Cur # msg in queue
  msglen_t msg_qbytes;//Max bytes allowed in Q
  pid_t msg_lspid;         //PID of last msgsnd
  pid_t msg_lrpid;         //PID of last msgrcv
};
```

Example 1 of usage of a message queue

- Many processes (even of different users) receiving, modifying and sending a message over the queue
  `test-rcv-snd.c`
- One process creating the queue and sending the first message
  `test-msg-start.c`

Example 2 of usage of a message queue

- The parent process:
  1. Checks the max size of messages
  2. Create a queue
  3. Forks `NUM_PROC` sender child processes
  4. Forks a receiver process
  5. Waits for the sender processes to terminate
  6. Waits for the queue to be empty
  7. Deallocate the queue, waits for the receiver, then exit
- Each sender child process:
  1. Sends `NUM_MSG` to the queue of type from `1` to `NUM_MSG`
- The receiver process:
  1. Receives all messages from the queue and prints them

  `test-ipc-msg-fork.c`
- Try the following changes:
  - The receiver receives only messages with type `< NUM_MSG/2`
  - `NUM_MSG` receiver processes receive one type of message only (from `1` to `NUM_MSG`). Useful when the queue represents "events of some type" and receivers are managers of events

21 System V: semaphores

Why semaphores?

- In concurrent programming (many processes running simultaneously), the output depends of the input **and on the scheduling decisions**
- synchronization primitives are used to constrain the possible schedules
- *semaphores* are synchronization primitives

Semaphores: terminology

- *resource*: term to denote a system resource: a printer, a memory segment, an I/O device, etc.
- *shared resource*: a resource used by more than one process
- *number of concurrent accesses* to a resource: the maximum number of processes which can access a resource
- *Semaphores* are used to *protect* shared resources
- Semaphores allow setting the number of concurrent accesses to shared resources

How does a semaphore work?

- For any semaphore s, the kernel records a *value* denoted by $v(s)$ **always** ≥ 0
- The value $v(s)$ of a semaphore represents the number of available accesses to the resource protected by s
- Any process can perform the following actions on a semaphore s:
  1. Initialize the value $v(s)$ with some integer a (number of allowed concurrent accesses to the resource)
     - $v(s) \leftarrow a$
  2. Use, if available, the shared resource protected by the semaphore s
     - if $v(s)$ equals 0, block the process until $v(s) > 0$
     - decrement $v(s)$ and use the resource
  3. Release a resource being used
     - increment $v(s)$ (it is never blocking)
  4. Wait until $v(s)$ equals zero. A process waiting until $v(s)$ is zero can be used to
     - "refill" the resource
     - have many processes waiting for the same "green light"

Creating/accessing an System V semaphore

- System V implements a **set of semaphores**: each operation onto a set of semaphores is **atomic**
- The system call

```
int semget(key_t key, int nsems, int semflag);
```

  returns the identifier of a set of `nsems` semaphores associated to `key`

  - `semflag` is a list of ORed ("|") options including:
    - read/write permissions (least significant 9 bits)
    - `IPC_CREAT`:
      (1) create a new semaphore associated to the `key`, if it doesn't exist
      (2) return the existing semaphore associated to the `key`, if it exists
    - `IPC_EXCL` (used only with `IPC_CREAT`): the call fails (with `errno=EEXIST`) if the semaphore exists
- When created semaphores are not initialized to any value: they must be explicitly initialized by `semctl(...)` (see later)
- Semaphores are persistent object: they will survive to the process death, they must be erased expicitly

Accessing and releasing a resource

- Access/realease of a semaphore are called *semaphore operations*
- A single operation on a semaphore is described by a dedicated data structure `sembuf`

```
struct sembuf {
  unsigned short sem_num;   // sem number
  short          sem_op;    // sem operation
  short          sem_flg;   // operation flags
}
```

- An array of operations over the semaphore `s_id` are performed by

```
int semop(int s_id, struct sembuf * ops, size_t nops);
```

  - `ops`, array of the operations
  - `nops`, number of the operations (size of array)
  - **Notice**: the `nops` operations are made all together atomically
  - The call blocks is the operation cannot be made

Accessing and releasing a resource

```
struct sembuf {
  unsigned short sem_num;   // semaphore number
  short          sem_op;    // semaphore operation
  short          sem_flg;   // operation flags
}
```

- To access a resource protected by semaphore the value of the semaphore must be **decremented** by setting `my_op.sem_op = -<num-res>;`
  - Important: the process **blocks** if `<num-res>` resources are not available
- To release a resource protected by semaphore the value of the semaphore must be **incremented** by setting `my_op.sem_op = <num-res>;`
  - Important: the process **never blocks** when increasing the resources
- `sem_num`, indicates the index of the semaphore in the set

Waiting until semaphore is zero

```
struct sembuf {
  unsigned short sem_num;   // semaphore number
  short          sem_op;    // semaphore operation
  short          sem_flg;   // operation flags
}
```

- If processes $A_1, A_2, \ldots, A_n$ must wait that another process B reaches some given point, then
  1. a semaphore is initialized with value `1`
  2. all processes $A_1, A_2, \ldots, A_n$ "waits for zero" with a semaphore operation `my_op.sem_op = 0;`
  3. process B decrements the semaphore by one by `my_op.sem_op = -1;`
     - the value of the semaphore becomes zero and the processes $A_1, A_2, \ldots, A_n$ will be unblocked

Controlling (and initializing) a semaphore

- The system call `semctl()` enables several actions to be performed on semaphores

```
int semctl(int s_id,int i,int cmd,/* arg */);
```

  - `s_id`, is the ID of the semaphore set
  - `i`, is the index of the semaphore in the set
  - `cmd`, describes the action to be taken over the semaphore
  - the optional fourth argument depends on the type of command
- To set (initialize) or get the value of the `i`-th semaphore in a set

```
int semctl(int s_id,int i,SETVAL,int val);
int semctl(int s_id,int i,GETVAL);
```

  - if `GETVAL`, the value of the `i`-th semaphore is returned;

Semaphore: getting information, removing

- To know how many processes are blocked

```
int semctl(int s_id,int i,GETNCNT);
```

  returns the number of processes waiting for the `i`-th semaphore to increase

- To know the process who last accessed a resource

```
int semctl(int s_id,int i,GETPID);
```

  returns the PID of the last processes who executed a `semop(s_id,...)` operation on the `i`-th semaphore

- To deallocate the semaphore `s_id`

```
int semctl(int s_id,/*ignored*/,IPC_RMID);
```

  - when a process is blocked on a `semop(id,...)` and the sempahore is removed by `semctl(id,...,IPC_RMID)` the process is unblocked with return value `-1` and `errno` is set to `EIDRM`

Semaphores and signals

- When a process is blocked on a `semop(...)` and it receives a non-masked signal
  1. the handler is executed
  2. the `semop()` system call returns `-1` and `errno` is set to `EINTR`

Semaphores: Examples

1. Tanti processi che vogliono cucinare condividendo le risorse di una cucina
   `test-sem-cook.c`

22 System V: shared memory

IPC shared memory

- System V implements *shared memory*: remember, when allocating by `malloc` you are allocating over the heap (which is private to the process!)
  - If memory is allocated by `malloc` and then
  - a process is forked
  - the two processes **do not share** the allocated memory
- Shared memory is a fast way for processes to communicate: no kernel structure (buffers, queues, etc) mediating the access to the shared memory:
  - this is good: fast way to implement IPC
  - this is bad: high risk of inconsistent behavior if memory is read "in the middle" of a write
- Once a process writes to a shared memory segment, the data written becomes immediately accessible to the other processes accessing the shared memory segment
- Simplicity of usage: assignments are made with the same syntax of private memory: no special function to access, no `write`, `read`, `msgsnd`, `msgrcv`, just "`=`"

Lifecylce of a shared memory segment

1. **Creation** by `shmget(...)`: size of memory must be specified
   - the shared memory segment is allocated over an area accessible to all processes
2. **Attaching** the shared memory area to the process address space
   - after a shared segment is attached to a private address space, it can be normally used by the process
   - any data written to the shared memory segment becomes immediately visible to other processes sharing the same segment
3. **Detaching** the segment from the process address space
   - the shared memory is no longer visible, but it still exists (it is a **persistent** object)
4. **Deallocation** of the shared memory segment

Creating a shared memory segment

- The system call

```
int shmget(key_t key, size_t size, int shmflg)
```

  returns the identifier of a shared memory segment associated to `key` of size **at least** `size` (the allocated size must be a multiple of `PAGE_SIZE`)
  - `shmflag` is a list of ORed ("|") options including:
    - read/write permissions (least significant 9 bits)
    - `IPC_CREAT`:
      (1) create a new shm segment associated to the `key`, if it doesn't exist
      (2) return the existing shm ID associated to the `key`, if it exists
    - `IPC_EXCL` (used only with `IPC_CREAT`): the call fails (with `errno=EEXIST`) if the shm segment exists
- Shared memory segments are persistent object: they will survive to the process death, they must be erased expicitly

Attaching a shared memory area to a process

- To attach a shared memory segment to the address space of a process

```
void *shmat(int shmid, NULL, int shmflg)
```

  - `shmid`, ID of the shm object
  - second argument used for advanced features: setting to NULL is safe
  - `shmflg`, flags
    - `SHM_RDONLY`, uses the shared memory in read-only mode
    - plus others for advanced settings
- it returns a pointer to the shared memory segment
- Typical usage

```
struct my_data * datap; //shared data struct

shm_id = shmget(IPC_PRIVATE, sizeof(struct my_data), 0600);
datap = shmat(shm_id, NULL, 0);
// From now on, all processes accessing to
// datap->something, read/write in shared mem
```

Detaching a shared memory

- A shm segment is detached by

```
int shmdt(const void *shmaddr);
```

- `shmaddr` is the address of the segment we want to detach, previously returned by a `shmat` call
- **Implicit detaching** of a shm segment occurs when:
  1. the process terminates
  2. the control flow passes to another process by `exec()`
- detaching is **not** deallocation

Control (and deallocation) of a shm segment

- A shared memory segment is controlled by

```
int shmctl(int shmid, int cmd,
           struct shmid_ds * buf);
```

  - `shmid`, the ID of the shared memory object
  - `cmd`, is the command to be made (`IPC_STAT`, `IPC_RMID`, ...)
  - the third argument may be used depending on the command `cmd`
- To mark a shared memory for deallocation

```
int shmctl(int shmid, IPC_RMID, NULL);
```

  - **Important**: the actual deallocation happens only when the last process is detached from the shared memory segment
  - Deallocating the shm segment immedately would create problems to the processes still using the segment
    - these problems cannot be detected by some `errno` (as for message queues), because the access to memory segment is made by assignments "`=`", not by any function calls

Example on shared memory

- `test-ipc-shm.c`