Laboratorio C+Unix: Notes from the slides
Anno accademico 2019/20

Enrico Bini

November 29, 2019

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

- Follow directions on the README, available on
  `https://informatica.i-learn.unito.it/`

1.1 Shell

Shell

- shell (Italiano = "guscio"), versus kernel (Italiano = "nucleo")
- The shell is a command line interpreter that enables the user to access to the services offered by the kernel
- 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". Icon: ) 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`

System calls

- system calls ("syscalls" for short) are the "access point" to the kernel: the way programs ask the kernel for any service
- Example of services asked to the kernel:
  - reading a file from the disk,
  - reading the keyboard,
  - printing over the screen,
  - reading from the network card
  - ...
- syscalls are identified by a unique number
- `strace <command>` shows all system calls happening when invoking `<command>`
- `strace -wC <command>` also shows a summary of the invoked system calls

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> to view status and type of a file
- cat /dev/sda1 to show the content of the disk...

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
`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 file's 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 output of command A as input to command B (“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 (used to store the name of the user or the name of the service associated a system account)
  6. the home directory of the account
  7. the default shell
- Command `usermod [OPTIONS] <username>` to change any among the fields above and more
- `usermod -c "Nuovo Nome" bini` to change the comment field into “`New Name`”

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
  - web servers run as dedicated account
  - ...
- they assist in running 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
- `cat /etc/group` to view the list of group. Each row has:
  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.")
  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)
- `chown` to change the owner of a file
- `chgrp` to change the group of a file
- `chmod` to change the permissions of a file
- Example: `chmod u+rw <filename>` adds read/write for the owner
- Example: `chmod o-r <filename>` remove write for the others

File permission, octal representation

- File permissions are often represented in octal (base 8)

user			group			other			octal
r	w	x	r	w	x	r	w	x	
1	1	1	1	1	0	1	0	0	=764

- Equivalent commands
  - `chmod u=rwx,g=rw,o=r <filename>`
  - `chmod 764 <filename>`
- 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 the C compiler `gcc` by
   `gcc -v`
   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 by `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`

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 as in "`} else {`"
  - do your best to stay in 80 columns
- Check example at the website

The opposite of coding style: Obfuscated C Code

- some guys enjoy write "obfuscated code"

```
         #define/*__Int3rn^ti[]n/l_()I3fusc^t3|]_C_C<>I7E_C[]nt3st__*/L/*__MMXVIII__*/for
     #include/*!"'()*+,-./12357:;<=>?CEFGHIJKLMNSTUVWXYZ[]^_`cfhijklmnrstuvwxyz{|}*/<stdio.h>
  char*r,F[1<<21]="~T/}3(|+G{>/zUhy;Jx+5wG<v>>u55t.?sIZrC]n.;m+:l+Hk]WjNJi/Sh+2f1>c2H`)(_2(^L\
 -]=([1/Z<2Y7/X12W:.VFFU1,T77S+;N?;M/>L..K1+JCCI<<H:(G*5F--E11C=5?.(>+(=3)Z-;*(:*.Y/5(-=)2*-U,\
/+-?5'(,+++***''EE>T,215IEUF:N`2`:?GK;+^`+?>)5?>U>_)5GxG).2K.2};}_235(]:5,S7E1(vTSS,-SSTvU(<-HG\
-2E2/2L2/EE->E:?EE,2XMMMM1Hy`)5rHK;+.T+?[n2/_2{LKN2/_|cK2+.2`;}:?{KL57?|cK:2{NrHKtMMMK2nrH;rH[n"
"CkM_E21-E,-1->E(_:mSE/LhLE/mm:2Ul;2M>,2KW-+.-u).5Lm?fM`2`2nZXjj?[n<YcK?2}yC}H[^7N7LX^7N7UN</:-\
ZWXI<^I2K?>T+?KH~-?f<;G_x2;;2XT7LXIuuVF2X(G(GVV-:-:KjJ]HKLyN7UjJ3.WXjNI2KN<l|cKt2~[IsHfI2w{[<VV"
"GIfZG>x#&#&&$#$;ZXIc###$&$$#>7[LMv{&&&&#&##L,l2TY.&$#$#&&$,(iiii,#&&&#$#$?TY2.$#$1(x###;2EE[t,\
SSEz.SW-k,T&&jC?E-.$##      &#&57+$$#      &&&W1-&$$7W  -J$#$kEN&#&      $##C^+$##W,h###n/+L2YE"
"2nJk/H;YNs#$[,:TU(#$   ,:   &&~H>&#   Y;   &&G_x&#2;   ,mT&$YE-#&   5G   $#VVF$#&zNs$$&Ej]HELy\
CN/U^Jk71<(#&:G7E+^&#  l|?1  $$Y.2$$  7lzs  WzZw>&$E    -<V-wE(2$$  G>x;  2zsW/$$#HKt&$$v>+t1(>"
"7>S7S,;TT,&$;S7S>7&#>E_::U  $$'",op  ,*G=  F,*I=957+F  ;int*t,k,O,  i,   j,T[+060<<+020];int M(
int m,int nop){;;;return+   m%(0+nop  );;}  int*tOo,w,  h,z,W;void(C)  (int n){n=putchar(n);}int
f,c,H=11,Y=64<<2,Z,pq,X   ;void(E/*d  */)(  int/*RP*/n  ){L(Z=k+00;  Z;  Z/=+2+000)G[000]=*G*!!f
|M(n,2)<<f,pq=2,f=+06   <f?++pq,++pq  ,G++  ,z:f+001,n  /=2;;}void  (V)(  int/*opqrstabd*/n){C(n
%Y);;C(n/Y+00);;}void  J(){L(pq--,pq   =j   =O=-1+0;++  j<240;I[6+   (h   +6+j/12/2*2+M(j/2,2))*
W+M(j/2/2,+06)*2+w*014      +00+M(00+      000+j,002      +00)]=000      +00+k)k=M(G[j/2/2+(*r-+
32)**"<nopqabdeg"],/*4649&96#*/3);/*&oaogoqo*/;}/*xD%P$Q#Rq*/int/*dbqpdbqpxyzzyboo3570OQ*/main()
{L(X=Y-1;i<21*3;i++,I++)L(r=G,G+=2;*G++;)*G>=13*3?*G-*r?*I++=*G:(*I++=r[1],*I++=r[2]):1;L(j=12,r
=I;(*I=i=getchar())>-1;I++)i-7-3?I-=i<32||127<=i,j+=12:(H+=17+3,W=W<j?j:W,j=12);L(;*r>-1;r++)*r-
7-3?J(),w++:(w=z,h+=17+3);C(71);C(73);V('*'*'1'*7);C(57);C(32*3+1);V(W);V(H);C(122*2);L(V(i=z);i
<32*3;)C(i++/3*X/31);C(33);C(X);C(11);L(G="SJYXHFUJ735";*G;)C(*G++-5);C(3);V(1);L(V(j=z);j<21*3;
 j++){k=257;V(63777);V(k<<2);V(M(j,32)?11:511);V(z);C(22*2);V(i=f=z);V(z);V(W);V(H);V(1<<11);r=
  G=I+W*H;L(t=T;i<1<<21;i++)T[i]=i<Y?i:-1;E(Y);L(i=-1;++i<W*H;t=T+Z*Y+Y)c=I[i]?I[i]*31-31:(31<
    j?j-31:31-j),Z=c[t[c]<z?E(Z),k<(1<<12)-2?t[c]=++k,T:T:t];E(Z);E(257);L(G++;k=G-r>X?X:G-r
        ,C(k),k;)L(;k--;C(*r++/*---#$%&04689@ABDOPQRabdegopq---*/));}C(53+6);return(z);}
```

- by Yusuke Endoh at 2018 International Obfuscated C Code Contest
- download `2018.c` and `smily.txt`
- compile by `gcc 2018.c -o 2018`
- run by `./2018 < smily.txt > smily.gif`
- open `smily.gif` with any image viewer

2.2 Variables and memory

What is a variable?

- What is a variable?
- In C, a view closer to implementation is taken
- In C, the best way to think of a variable is as a **portion of memory**
- In some sense, variables "do not exist". Only the memory exists! "Variables" are just a convenient way to refer to pieces of memory.
- Often times, in C we care only about:
  - the **amount** of memory taken by a variable
  - **where** in the memory is a variable allocated
  - the **value** of a variable
  - the type of a variable is of little importance:

    the C is a **weakly typed** language

Memory: the abstraction

- In C the memory is abstracted as a loooong sequence of **bytes**
- Memory is **byte-addressable**: at every memory address, there is only one byte

address	content
...	...
`7FFF0040671A8107`	`A7`
`7FFF0040671A8108`	`E8`
`7FFF0040671A8109`	`03`
`7FFF0040671A810A`	`00`
`7FFF0040671A810B`	`00`
`7FFF0040671A810C`	`50`
...	...

- addresses in memory are represented by a machine-dependent number of bytes: in the slide by 8 bytes (16 hex digits)
  - by 8 bytes long addresses, it is possible to address up to $2^{8\times 8} = 2^{64} \approx 16 \times 10^{18}$ bytes (16 billions of GB)
- about a billion times larger than current size of large RAMs
- the address space is not used only to store data
- files or I/O devices (video) may be mapped onto a portion of the address space

Memory: endianness

- How to store variables needing more bytes?
  - by using contiguous memory locations
- Example
  - an `int` variable `var` is represented over 4 bytes
  - if stored at address `7FFF0040671A8108`, then it occupies the cells at:
    1. `7FFF0040671A8108`
    2. `7FFF0040671A8109`
    3. `7FFF0040671A810A`

4. `7FFF0040671A810B`

– its value is `var = 1000` (1000 decimal = 4 bytes `00 00 03 E8`$_{16}$)

little-endian: starts from **least** significant byte

...	...	
7FFF0040671A8108	E8	var
7FFF0040671A8109	03	
7FFF0040671A810A	00	
7FFF0040671A810B	00	

big-endian: starts from **most** significant byte

...	...	
7FFF0040671A8108	00	var
7FFF0040671A8109	00	
7FFF0040671A810A	03	
7FFF0040671A810B	E8	

x86 processors: multi-byte data is stored as **little-endian**

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. An optional **initialization** by a constant

```
int a;                  /* not initialized */
int b = 91;             /* initialized */
int c = my_function();  /* wrong init: not a constant */
```

4. A variable has a **value**. The value is the interpretation of the bytes in memory according the variable type
5. a variable is a portion of memory. 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**.

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
- 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`
- Check the size of the type of variables on your machine

  `test-sizeof.c`

2.3 Output: basics

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

3 Arrays and strings

Arrays

- An array is **not a class** (as in Java)
- An *array* is a **contiguous** area of memory allocated to **several variables** of the **same type**
- An array is declared by

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

it has size **sizeof(<identifier>) = sizeof(<type>)*<size>**

- Example:

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

declares the variable `v` as an array of `10` `int` 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 contiguously in memory

address	content	variable
...	...	...
0080FC	...	
008100	v[0]	v
008104	v[1]	
...	...	
008124	v[9]	
008128	...	

- $\texttt{8100} + 40_{10} = \texttt{8100} + 28_{16} =$ `8128`

Arrays: length

- The length of an array is **not** saved in the data structure
  - **Do not ever try** to invoke the "method" `length()` with an "array" object
  - "methods" and "objects" **do not exist** in C
- The programmer must record the length of the array in some way
  - by storing a special character terminating the useful content (such as in strings, which are terminated by the byte `0`)
  - by recording the length in another (additional) variable

Strings: arrays of bytes terminated by 0

- The String "object" or "class" **does not exist** in C
  - (again, "object" and "class" **do not exist** at all in C)
- The term "string" in C is often used to denote
  1. an array of `char`, as declared by:

     `char s[100];`
  2. the bytes of such an array are interpreted as ASCII codes of characters
  3. the byte `0` is written in `s` after the last character, to terminate the string
- A string may be printed by the `%s` placeholder of the `printf` as in

```
printf("The string s is \"%s\"\n", s);
```

Strings: manipulation by including `string.h`

- By including the library `#include <string.h>` some useful function strings to manipulate strings may be used
  1. The following function returns the number of bytes in `s` before the terminating byte `0`

```
strlen(s);
```

  2. to append string `src` to string `dest`

```
strcat(dest, src);
```

  - `dest` **must** be allocated at least `strlen(dest)+strlen(src)+1`

- otherwise (quoting from `man strcat`): "If `dest` is not large enough, program behavior is unpredictable; **buffer overruns are a favorite avenue for attacking secure programs**."

3. to append **up to** `n` bytes of `src` to string `dest`

```
strncat(dest, src, n);
```

   - if no `0` byte terminating `scr` among the first `n` bytes, only first `n` bytes are concatenated
   - it prevents the user to write arbitrary-long data

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.

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 the value represented by the string `s` in the integer variable `a`

- Converting a string into an floating point number

```
double a;

a = atof(s);
```

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

Reading input from the keyboard: `fgets()`

- the function `fgets(...)` reads a string of characters
- `#include <stdio.h>` must be added on top to use it
- 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

- `man fgets`

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

4 Pointers to memory

Pointers

- All variables are represented by sequence of bytes
  - `int`, `long` are interpreted as integer in two-complement
  - `float`, `double` are interpreted as floating point numbers according to the standard IEEE 754-1985
- A pointer variable is interpreted as **an address in memory**
- Declared by specifying the type of the variable it points to

  `<type> * <identifier>;`

- only the pointer is allocated, **not the variable it points to**!!
- Example

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

  declares `pi1` and `pi2` as pointers to integer, `i` and `j` are just integers.

- Usually names of pointers contain "`p`" or "`prt`"

address	content	variable
...	...	
000000000000A808	000000000000C800	`pi1`
...	...	
000000000000C800	`<something>`	`int`-type variable
...	...	

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;

/* allocate int at *p_a */
p_b = p_a;
*p_a = 1;
*p_b = 2;
printf("%i\n", *p_a);
```

address	content	variable
...	...	
008100	008200	p_a
008108	008200	p_b
...	...	
008200	1	*p_a
...	...	

- **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);
```

Operations with pointers: casting

- "casting" a variable is an explicit type conversion

```
double f = 0.21;
int a;

a = (int)f;  /* data loss due to truncation */
```

- by casting pointers, it is changed the type of pointed data

```
double f = 0.21;
int * pa;

pa = (int *)&f;
```

- pointers 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

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`.

Arrays and pointers

- Technically, the name of an array is a **constant** pointer
- A pointer `<type> * p;` is used to refer elements of the array at `p`

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

  v[0] = 23;  // same as *v = 23;
  p = v;      // same as p = &v[0];
  p = &v[1];
  p[1] = 5;   // same as v[2] = 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 to `<type>`, `(p+i)` is a pointer to `p[i]` of the array `p` of elements of type `<type>`
- The address pointed by `p+i`, then is `p+i*dim`, with `dim=sizeof(*p)`
- Example: assuming that the following variables are declared

  ```
  int v[10] = {1, 9, 1000}, *q = v + 3;
  ```

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

  ```
  q = v+1;
  v = q+1;
  q++;
  *q = *(v+1);
  *q = *v+1;
  q[4] = *(v+2);
  v[1] = (int)*((char *)q-3);
  q[-1] = *(((int *)&q)-9);
  v[-1] = *(--q);
  ```

address	content	variable
...	...	...
008100	1	
008104	9	
008108	03E8=1000_{10}	v
00810C	0	
...	...	
008124	0	
008128	00810C	q
...	...	

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`*

5 C: types

5.1 Integers

Integers: `signed`, `unsigned` representations

- Integer types (`char`, `short`, `int`, `long`) may be:
  1. **`signed`**: bytes are interpreted as number with sign: if negative in two-complement
     - by default all integer types are signed
  2. **`unsigned`**: bytes representation interpreted as positive number
     - `unsigned` variables must be declared explicitly as in

```
unsigned int a;
```

- Examples on 8 bits

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

- having both signed and unsigned integers in the same expression is a **bad idea**

  `test-sign.c`

Integers: limits

- 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	$\approx 16\times 10^{18}$

Integers: constants

- In C code integer constants are
  1. 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
     - otherwise they are interpreted as decimal
  2. single characters within `'` (as in `'a'`) to represent the ASCII code of that character
- Best expression to write the ASCII code of the digit `n` is
  `'0'+n`
- *`test-int-const.c`*

5.2 "Boolean"

The type boolean does not exist

- Although conditions do exist
- When evaluated as condition, a numerical expression `<expr>` is

  false if `<expr>` is equal to zero

  true otherwise
- Example of a for loop

```
/* Compact way to run 10 iterations */
for (i=10; i; i--) {
    /* body of the for loop */
}
```

5.3 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: limits

type	min	max
`float`	1.17549×10^{-38}	3.40282×10^{38}
`double`	2.22507×10^{-308}	1.79769×10^{308}

Floating point: constants

- Floating-point constants are written in C with the decimal dot "`.`" or with the letter **`e`** (or **`E`**)
- Examples:

```
double a;

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

Floating point: imprecise arithmetic

- The finite number of bits to represent real numbers introduces an approximation error
- The approx error may even lead to violation of basic properties, such as the associativity of addition

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

- Also, if a floating point number needs to be tested if it is equal to zero **never** use `== 0` or `!= 0`
- Always, test proximity to zero (not equality) by some code as

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

5.4 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 = i;  // 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*

6 Operators and control

6.1 Operators

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 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: useful to get/set bits of a representation
  - `~`, binary NOT
  - `&`, binary AND
  - `|`, binary OR
  - `^`, binary XOR

```
if (x & 0x80) {
    /* the MS bit of the LS byte is 1  */
}
```

```
x = x ^ 0xFF /* flip the LS byte */
```

The shift operator

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

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

```
if (x = 0) {
    /* never taken, x = 0 is always false */
}
```

- `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, `|=`, compact assignment
  - `<expr1> = <expr1> <operator> <expr2>` can be written as
    `<expr1> <operator>= <expr2>`

6.2 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

`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;
    ...
}
```

`do-while` **loop**

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

`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**

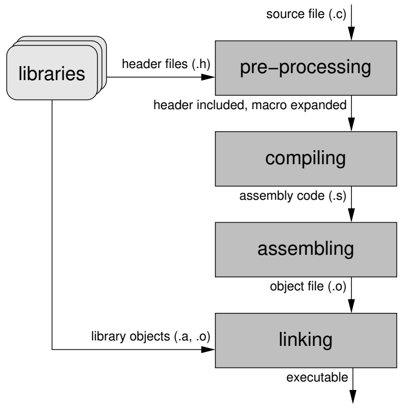
```
switch (letter) {
case 'A':
case 'a':
        // 'a' action
        break;
case 'M':
case 'm':
        // 'm' action;
        // also 'k' action must be done
case 'K':
case 'k':
        // 'k' action
        break;
default:
        break;
}
```

7 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 as 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
3. **Assembling**: the assembler translates assembly instructions into machine code or *object code*
4. **Linking**: object code is linked to the library code



7.1 Pre-processing

Pre-processing: overview

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

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

- The pre-processor replaces text typographically
  - the input file may not be necessarily a C program
- The "instructions" of the pre-processor are called *directives*
- 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

Pre-processing: `#define` directive, constants

- `#define` is used to define costants and macros. 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. Example: `gcc -D PI=3.14` is equivalent to add at the head of file

```
#define PI 3.14
```

- Empty constants are possible: they are removed from the source file

```
#define EMPTY_CONST
```

Pre-processing: `#define` directive, macros

- `#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);// replaced by 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))
```

Pre-processing: `#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; }
```

  to be used as

  `EXCHANGE(int, a, b);`

- 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);
```

Pre-processing: `#include` directive

- `#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*
- A header file exports some functions of a library
- The *C standard library*, often called `libc` (`glibc` is the GNU `libc`) collects many useful functions
  - `stdio.h`, functions for input/output, files, etc.
  - `string.g`, string handling, copying blocks of memory
  - `math.h`, mathematical functions (sin, cos, pow, etc.)
  - `errno.h`, to test error codes set by functions
  - `limits.h`, architecture-dependent min/max values of different types
  - `stdlib.h`, random numbers, memory allocation, process control
  - `ctype.h`, for testing the type of characters (upper/lower case, etc.)

Pre-processing: conditional inclusion

- 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 pre-processing 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`

Pre-processing: 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-processing: 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

- A good example of debugging code is:

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

Pre-processing: the `NULL` pointer macro

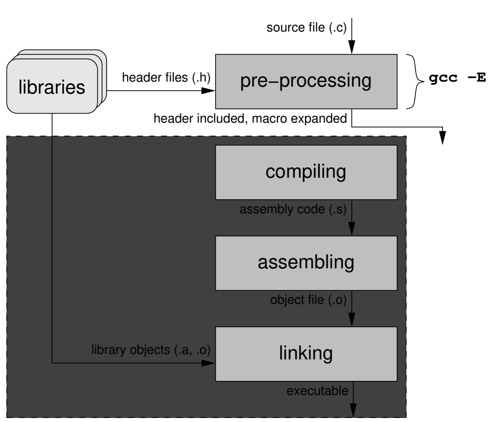
- The macro `NULL` represents a pointer (address in memory) which is invalid
- The value of the `NULL` macro is zero. After

  ```
  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

Pre-processing: invoking preprocessor only

- By running
  `gcc -E filename`
  the pre-processor only is executed on **`filename`** and the output is written to the terminal (`stdout`)



- Hence, by
  `gcc -E filename > after-pre-proc`
  the output of the pre-processor is written to `after-pre-proc`

  `test-preproc.c`

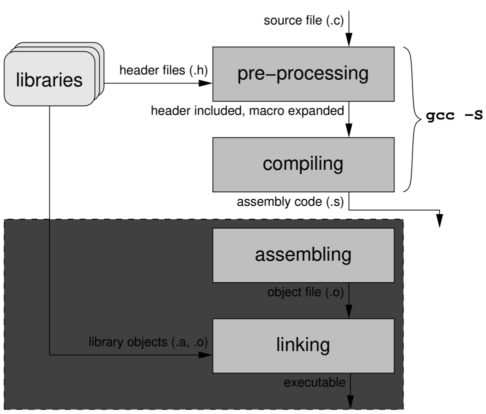
Pre-processing: options

- `-E` stop after pre-processing and produce the output to the terminal (`stdout`). Must be redirected to file is it is needed to save it
- `-D` , defines a macro
- `-I <dir>`, search directory `<dir>` before standard include directories

7.2 Compiling

Compiling: invoking compiler only

- 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`

Compiling: options

- billions of options for compiling `man gcc`

1. Cross-compiling: produce the assembly for different architectures:
   - `-m32` 32-bit architectures
   - `-marm` ARM architectures
2. Optimization of the code
   - `-O2` some typical optimizations (such as loop unrolling): optimizations depends very much on the architecture
   - `-Os`, optimize the size of the object file
3. Debugging
   - `-g`, add debugging symbols (used by the debugger `gdb`)
   - `-O0`, no optimization (optimized code is hard to debug)
4. Try compiling by
   `gcc -S -g -O0 test-int-const.c`

Compiling: syntax to be used for the exercises/project

1. `-std=c89`, select the ANSI C standard (the first standardized C in 1989)
   - variables are declared only at the top of the block. Not allowed to declare variables "on the fly" as in

     ```
     for (int i=0; i<10; i++) /* no C89 standard */
     ```

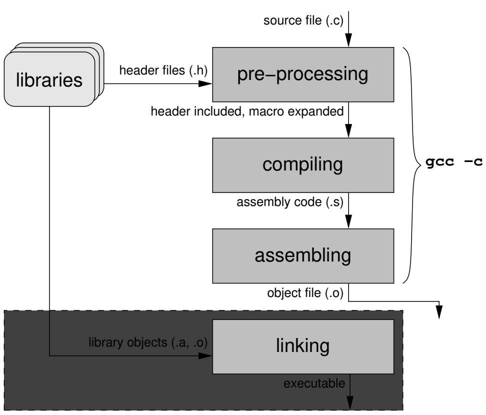
   - no comment
     `// commento`
     only
     `/* commento */`
     accepted

2. `-pedantic` rejects programs not conforming to the ANSI C standard

7.3 Assembling

Assembling

- *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`

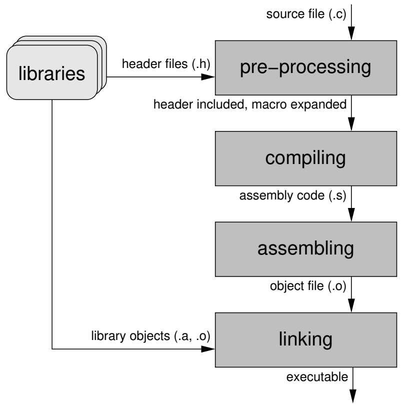


- Try
  `gcc -c test-int-const.c`
  `hexdump -C test-int-const.o`

7.4 Linking

Linking

- Last step of `gcc` is *linking*: the pieces of code are linked together
- The linker needs one (and only one) function `main(...)` to be defined: going to be the first code to be exec
- try next commands to modify an executable
  `gcc test-int-const.c -o test`
  `bless test`
  `./test`



- Options
  - `-l<lib-name>`, to link it with the library `<lib-name>`.
    Example: `-lm` to link with the math library
    `man sin`

    test-no-link.c

8 C: Functions

Functions

- *Functions* are used to break down a complex problem into smaller ones
- If you find yourself copying/pasting lines of code which "do something", then you may need a function for that code
- Functions **are not** parametric macros
- As in mathematics with
$$f : \underbrace{\mathbb{R}^2}_{\text{input}} \to \underbrace{\mathbb{R}}_{\text{output}},$$
  a C function gets an input and produces an output
- A C function is characterized by
  1. the *declaration of the function* (aka function `prototype`), which holds information about
     - the *name* of the function (mandatory)
     - the list of types of *input parameters* (optional)
     - the type of one *output parameter* (optional)
  2. the *body of the function*, which is the code that processes the inputs to produce the output
- the `void` type is specified for missing input, output or both

Functions: declaration (or prototype)

- The **compiler** requires that a function is declared before being used
- The *declaration* of a function (or *prototype*) is a line of code with
  - the type of one *output parameter* (`void` if none)
  - the *name* of the function (mandatory)
  - a comma-separated list of types of *input parameters* within round brackets (optional)
  - 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!!

- Example of function declaration

```
/*
 * Sorting the array v of length num
 */
void sort(int * v, unsigned int num);
```

and an equivalent way without the parameter names

```
void sort(int *, unsigned int);
```

Functions: definition (or body)

- The definition of a function includes
    1. its declaration and
    2. its body

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

- The body is needed by the **linker** only (step “4” of `gcc`)
- 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 body may be intentionally hidden to the user

test-declare-fun.c

Functions: invocation

- The declaration or the full 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()`

Functions: passing parameters

- When a function is invoked, the invocation parameters are **copied** into additional variables
- A function can use and modify the paramaters
- These modifications, however, have **no effect** 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? */
}
```

Fuctions: how to modify a parameter?

- Often times it is needed that a function modifies one or more parameters. Example: to sort an array
- However, parameters are always copies: any change to a parameter is lost after returning
- Solution: if some data needs to be modified by a function, then we declare a function that receives a **pointer to the data**, not the data itself
- Through the pointer the original data may be modified
- Example

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

- The ponter only is copied internally to the function

Functions: 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 as parameters (which is inefficient), 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, ...);
```

Functions: returning

- 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

Functions vs. macros

- **stage of gcc**: macros expanded by preprocessor, functions are compiled
- **type checking**: in macros, the type of operands is **not** checked
- **efficiency**: macros may be more efficient than functions, no parameters passing, no call instruction
- **size of executable**: if macros are used used the size of the executable grows
- **parameters**: macros are expanded by the pre-processor, if a paramenters is modified it remains modified after the macro as well. A modification of parameters within functions isn't seen outside
- **return value**: macros do not return any value. Still, a macro may be an expression
- **recursion**: obviously, no recursion with macros
- **debugging**: programs with many macros may be harder to debug

Functions vs. macros: example

- If we have both a function and a macro computing the minimum

```
#define MIN(x,y) ((x) < (y) ? (x) : (y))
int min(int x, int y) {
  return x < y ? x : y;
}
```

- What is the difference between

```
min_val = min(min(a,b),min(c,d));
min_val = MIN(MIN(a,b),MIN(c,d));
```

- If the function `min` is used:
  - 3 calls, 3 comparisons
- If the macro `MIN` is used:
  - 0 calls, 5 comparisons

```
MIN(MIN(a,b),MIN(c,d)) /* becomes */
(a < b ? a : b) < (c < d ? c : d) ?
(a < b ? a : b) : (c < d ? c : d)
```

Functions vs. macros: conclusion

- Macros may be a good replacement of functions when:
  1. the lines of code are few (say, 10)
  2. the function code is used many times
  3. high efficiency is needed
  4. no return value, nor recursion is used
  5. we are ready to hard-to-debug errors
  6. `gcc -E` is your friend
- Macros may be good for:
  1. computing the minimum between two values
- Functions may be good for:
  1. sorting an array

9 Scope of variables

Scope of a variable

- The *scope* of a name (variable or function) is the portion of code where that name is visible and then it 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 declared outside any function (*global variables* or *function declarations*) 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 "isolated" pieces of code
    2. hard to comprehend/debug: when the reader finds a global variable, it may be not obvious where it is declared
- If a function uses a global variable as input or output, it is **strongly recommended** to add a comment on top of the function
- Names of global variables should be highly informative to avoid the reader to browse much code:
  - `number_students` is good
  - `n` is bad

10 Storage classes

Storage classes

- All C variables have a **storage class**, which determines where variables are stored
- Normally, variables are stored in memory. Three possible areas of memory:
  1. variables over the BSS (Block Standard by Symbol, historical acronym)
  2. variables over the stack segment
  3. variables over the heap
- Moreover,
  4. variables may be stored in registers
- Finally,
  5. the storage class may be decided elsewhere in the code

Memory segments

- Depending on the needs, the OS assigns a few memory segments to *processes* (which are programs in execution)
- Segments are mapped over the process address space
- Each segment has:
  1. a start and end address (meaningful in the process address space)
  2. flags that determine the access modes:
     - read: it can be read
     - write: it can be written
     - execute: it contains code which may be executed
     - private/shared: if it isn't/is shared among other processes
- To view the memory mapping of a process, try:
  `ps -Af | grep sh` to get a Process ID (PID)
  `cat /proc/<PID>/maps` to print the memory map of `<PID>`

10.1 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

10.2 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

10.3 Variables on the heap

Variables on the heap: dynamic allocation

- The heap (in Italiano "mucchio", "cumulo") is a memory area available to the program upon specific request to the operating system
- The program may ask the OS some memory via the following calls

```
#include <stdlib.h>

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

  which is returned via a pointer `(void *)`

  - `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

- Allocating memory via `malloc` is called *dynamic memory allocation* because the size of the allocated memory is decided at runtime
- When variables are declared the size of memory is decided at compile time (*static memory allocation*)

Standard ways for dynamic allocation

- Standard code to allocate an array of `num` elements

```
int * p;

p = malloc(num*sizeof(*p));
/* better than malloc(num*sizeof(int)) */
```

- `calloc(...)` has a slightly different syntax and it clears the memory (set all bytes equal to zero)

```
int * p;

p = calloc(num, sizeof(*p));
```

- After the allocation, the memory can be used as needed

```
p = calloc(num, sizeof(*p));
for (i=0; i<num; i++) {
  p[i] = i*i;  /* using array notation */
}
```

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);
```

- `free(p)` is error, if `p` not returned by `malloc/calloc`
- 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;  /* ref to 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.
- Lifetime of memory allocated onto the heap
  - Allocated: when `malloc()/calloc()` is invoked
  - Deallocated: when `free()` is invoked (or at the end of the program)

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`
- Example of usage of `malloc`

  test-malloc.c

10.4 Variables in memory: comparison

Allocation of data in memory

- Let us have a look to the following examples:
  - *test-var-alloc.c*
  - *test-show-addr.c*
- Remember the difference between

```
char v[] = "string0";
char * p = "string1";
```

  - `string0` may be modified (it belongs to a page with “`w`” permission)
  - `string1` may not be modified (no “`w`” permission)

10.5 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

10.6 External variables

`extern` variables

- `extern` variables are allocated in other files
  - another program
  - the operating systems
  - ...
- also functions can be `extern`. If so, they must be declared in other modules
- `extern` variables are declared by

```
extern int my_extern_var;
```

- the compiler assumes that such a variable exists: it does not allocate space in memory for it
- the linker may give an error if the variable is not found anywhere

11 Composite data types

11.1 Data structures: `struct`

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;
};
```

- Each piece of data is called *field* of the `struct`
- In the example, the `struct point` has 2 `double` fields with names `x` and `y`
- The name of the type is "`struct point`". Hence, 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};
```

- the initialization of each field must follow the order of declaration.

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 accessed by pointers, each field of the pointed `struct` is referred by the notation "`->`"

```
struct info * p;

p = malloc(sizeof(*p));
p->age = 35; /*same as (*p).age = 35 */
```

Structures: byte alignment, padding

- How much memory is allocated to a struct? Where?

```
struct myrecord {
    int field1;
    double field2;
    /* more fields */
};
```

- Normally, fields are allocated in memory in the order they are declared
- Amount of memory of a `struct` **may be more** than sum of memory of each field

$$\texttt{sizeof}(\texttt{myrecord}) = \texttt{sizeof}(\texttt{field1}) + \texttt{sizeof}(\texttt{field2}) + \\ + \cdots + \text{"}\texttt{padding}\text{"}$$

- "padding" may be added to align the fields to "good" memory boundaries (multiples of 4, 8, or 16)
- *`test-struct.c`*

Structures: assignment

- `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) {
    ...
}
```

11.2 "Overlapping data structrures": `union`

Unions

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

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

- however all fields **overlaps**, starting from the **same address**!!
- *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)

11.3 Enumerating constants: `enum`

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

11.4 Defining new data types: `typedef`

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
  `gedit /usr/include/stdint.h`
- often types are also defined by pre-processor macros with `#define`.

```
#define MY_TYPE double

MY_TYPE my_var;
```

- Differences: macro-defined type is just a replacement by the pre-processor

11.5 Dynamic lists

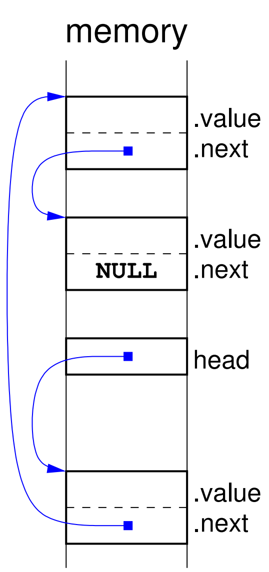
Dynamic lists by `struct`, `typedef`, `malloc`, ...

- In C, dynamic lists are created by
  - defining the element of the list by a `struct`

```
typedef struct node {
  int value; /* or any data */
  struct node * next;
} node;

typedef node* list;
```

  - the `struct` has a pointer to the **next element**
  - setting a pointer `head` to the **head** of the list
  - the `.next` field of last element has value `NULL`
  - node insertion:
    1. new node allocated by `malloc(...)`
    2. the new node is properly linked
  - node removal:
    1. node is unlinked
    2. node memory deallocated by `free(...)`

test-list.c



12 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`
Starting from **highest** 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

13 `scanf`, copying 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);
```

- ’i’: reads an integer(hex: if it starts with `0x`, octal: it starts with `0`, decimal: otherwise)
- 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

`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 (check `bcopy(...)` or `memmove(...)` in case of overlap)

- to fill the first `n` bytes pointed by `p` with the character `c`, use

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

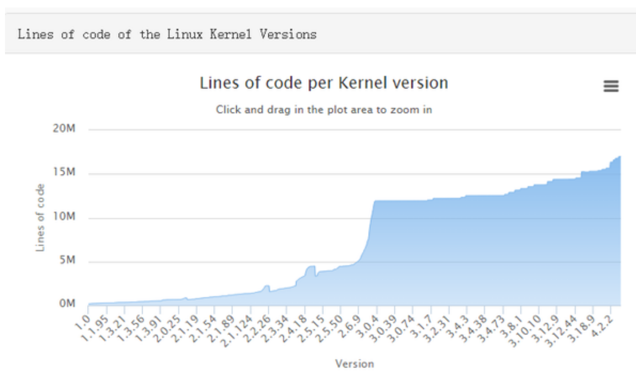
  - the memory area pointed by `p` must be allocated
  - `bzero(p,n)` is the same as `memset(p, 0, n)`

14 Modules

14.1 Modules: overview

Issues with a single long program

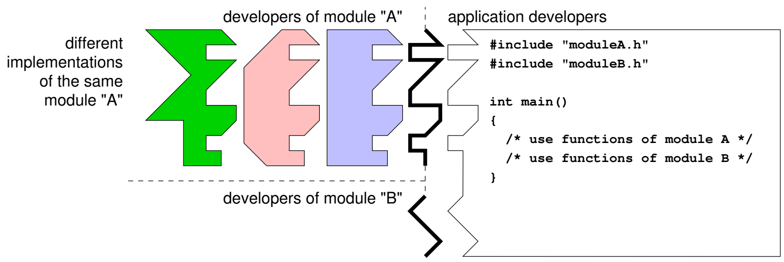
- Large programs (measured in number of lines or number of functions) may require many different functionalities
- Having the entire code on a single file may be problematic



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

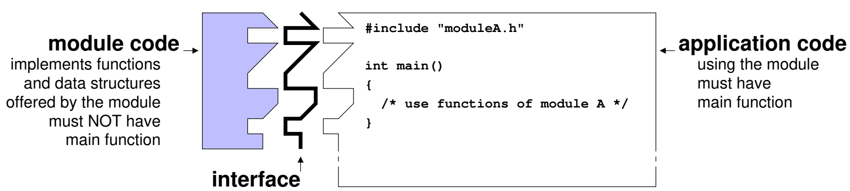
Modules

- Solution: to 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
- Development of a large project (example: the Linux kernel):
  1. the large project is split down into smaller "modules" (example: the Linux scheduler, the memory management, the I/O management, . . .)
  2. possibly different teams develop each single module
- The *interface* describes the features offered by the module (a sort of "contract" between the developers and the user of the module)



14.2 Modules in C

Modules in C



1. **interface**: the *header* file (example: `"moduleA.h"`)
   - lists functions and data types (`typedef`, `struct`,. . .) of the module
   - it is included by the `#include` directive
   - it is **never** compiled alone: there is no executable code
2. **module code**: implementation of module (example: `"moduleA.c"`)
   - contains the implementation of the functions listed in the header file
   - does **not** contain a `main()` function: the main function is the first function to be executed
   - **only compiled** by `gcc -c`, which produces the object file
3. **application code** (example: `application.c`)
   - it uses the module by including the header file
   - it **must** contain a `main()` function
   - compiled and **linked** with the object file of the module

Modules in C: the interface

- The interface of a module in C is the *header file*.
  Example: `module-name.h`
- It is included by all programs using the module by

```
#include "module-name.h"
```

- To avoid multiple inclusion it starts/ends as follows:

```
#ifndef _MODULE_NAME_H
#define _MODULE_NAME_H
/*
 * List of data types and functions offered
 * by the module with EXPLANATORY COMMENTS!!
 */
#endif /* _MODULE_NAME_H */
```

- it doesn't contain any executable code (no assignments, `for`, `if`, ...)
- it is **never** compiled. **Never, ever** write something like:
  `gcc module-name.h`

Modules in C: the implementation

- It describes how the module is implemented.
  Example: `module-name.c`
- It also includes its own header

```
#include "module-name.h"
/*
 * Implementation of the functions listed in
 * module-name.h
 */
```

- May be less commented: it is read by the module developers, **not** by the module users
- A module is **compiled only** by (notice the flag `-c`)
  `gcc -c module-name.c`
  which produces the object file `module-name.o` (not an executable)
- The implementation may be **hidden** to the user, who **only** needs
  1. the header file `module-name.h` (to compile his code)
  2. the object file `module-name.o` (to link his code)

Modules in C: application code

- The application is what you launch from the terminal
- If it wants to use the module, it must include its header file

```
#include "module-name.h"
/* Application dependent functions */

int main() {
  /* Application code */
}
```

- the pre-processor directive `#include "module-name.h"` allows using the module functions and data types and compiling without errors
- the code of the module functions is then added during the linking stage, it is **not** compiled with the application
- To do so, the application is compiled together with the object file `module-name.o` by
  `gcc application.c module-name.o -o application`

Example: the "`matrix`" module

- Let us have a look to a module implementing some matrix operations
  - *matrix. h*, the header file of the module (the interface)
  - *matrix. c*, the implementation of the module
  - *application. c*, an example of code using the module
- The module (only) may then be compiled (**not** linked) by
  `gcc -c matrix.c`
- Any program (such as
  application. c) which wants to use the module, must include only

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

  and be compiled (**and linked**) by
  `gcc application.c matrix.o`

14.3 Libraries

Libraries: the `ar` utility

1. When the modules to be used are many it may become complicated even to write to command line to compile
   `gcc app.c mod1.o mod2.o mod3.o ....`
2. The term *library* is often used to denote a collection of modules
3. The `ar` utility is used to **ar**chive many single files in a unique one
4. In programming, `ar` is used to store many object files into a unique *library*
5. Example: show the content of the Standard C Library (`libc`) by
   `ar t /usr/lib/x86_64-linux-gnu/libc.a | less`
6. Example: extract one object file by
   `ar x /usr/lib/x86_64-linux-gnu/libc.a printf.o`

Object dump

- `objdump` shows the content of an object file
- The format used to show the object is an ELF (Executable and Linkable Format) file
- Examples
  1. to see the assembly code of the module `matrix`, try
     `objdump -d matrix.o`
  2. recompilie by
     `gcc -c -g matrix.c`
     and then try the next command to see source code and assembly
     `objdump -S matrix.o`
- `objdump` may be used for reverse engineering on executables: understanding from the binaries what the program is doing
- Example: show the assembly code of the printf by
  `objdump -d printf.o`

15 Pointers: endgame

Array of pointers

- Pointers are variables
  - Arrays of pointers can be declared and used as arrays of 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 -c my_file.c -o my_file`

  - the command is `gcc`
  - 4 command-line arguments follow
- Command-line arguments 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

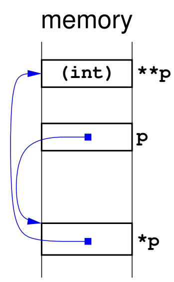
  test-command-line.c

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*

Pointers to functions

- The code of functions is in memory
- It is then possible to declare *pointers to functions*
- A pointer to a function is the address of code of the function in memory
- A pointer to a function is declared by:

```
<type> (* var_name) (<param_types>);
```

  - `var_name`, name of the function pointer;
  - `type`, type returned by the function;
  - `<param_types>`, list of input types;

- Different than a function returning a pointer to `<type>`

```
<type> * var_name(<param_types>);
```

- Arrays are also possible, by:

```
<type> (* var_name[LEN]) (<param_types>);
```

test-fun-ptr.c

16 Files and file descriptors

Files and file descriptors Two different interfaces to files

1. *streams* of type

```
FILE * my_f;
```

(`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
  - data to be written to disk is stored to a memory area (*buffer*) only
  - the buffer is written to the disk (*flushed*) depending on the buffering policy

2. *file descriptors* of type

```
int my_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.)

16.1 Streams

Streams: opening/closing

- 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. Example: `"rw"` 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

Streams: reading

- For each open file, the OS keeps and updates a *file position indicator*
- Every reading happens at the current position, which is incremented by the number of bytes read

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

- they all read from the current position
- `fgetc` reads and returns the byte (`char`) read into a (`int`). If end-of-file is reached, then the `int` non `char`-representable `EOF` macro is returned (tipycally of value `-1`)
  - the byte `0xFF` can be distinguished by `EOF`
- `fgets` reads an array of up to `size` bytes. Returned `NULL` if end-of-file is reached.
- `fscanf` read by `scanf/printf` format

Streams: writing

- Every writing happens at the current position, which is incremented by the number of written bytes

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

- `fputc` writes the byte `c`, casted to `char`, to the file
- `fputs` writes the zero-terminated string `s` **without** the terminating `0` byte
- `fprintf` writes to file by the `printf` format

Streams: 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

  test-file.c

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

Streams: buffering

- 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

Streams: 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.

- the function `setvbuf` changes the buffering policy of `stream`

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

  if `mode` is:

  1. `_IONBF`, stream is unbuffered (every single byte is written/read immediately)
  2. `_IOLBF`, the buffer is written as soon as newline is found
  3. `_IOFBF`, write to disk only whe buffer is full

  `man setvbuf` for more information

- Examples

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

16.2 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
- 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)

File descriptors linked to standard streams

- `stdin`, `stdout`, and `stderr` are standard streams opened by the OS and allowing the program to:
  - read from keyboard (from `stdin`)
  - write normal output to terminal (to `stdout`)
  - write error messages to terminal (to `stderr`)
- Standard file descriptors are associated to these streams:
  - the integer `0` is the file descriptor of `stdin`
  - the integer `1` is the file descriptor of `stdout`
  - the integer `2` is the file descriptor of `stderr`

Redirecting `stdout` and/or `stderr`

- 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
  ```

Opening/closing a file descriptor of a file

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

- `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
    - **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
    - `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

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

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(...)`

17 Error handling

Errors: the errno global variable

- The invocation of functions may fail. Examples:
  - `malloc` fails if memory is not available
  - `open` fails if the file to be read does not exist
- If the call to a function fails, the caller is informed by an invalid returned value. Examples:
  - `malloc` returns `NULL` if failing (invalid pointer)
  - `open` returns `-1` if failing (invalid file decriptor)
- If the invalid value is returned, the calling function knows that an error happened, but **doesn't know why**
- To inform about the cause of the error the global variable

```
int errno;
```

  declared in `errno.h` is set by the failing function
- When a function fails, it sets the global variable `errno` accordingly.
- The caller may get more details about the reasons of failure by inspecting the value of `errno`

Errors: values of errno

- The `man` pages of the failing function list all possible values of `errno` which may be set, in the section "`ERRORS`" usually at the bottom of the `man` page. Example: `man 2 open`
- These values are pre-processor macros set equal to non-zero integers.
- If `errno == 0`, then the previous call was successful
- the function (declared in `string.h`)

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

  returns a pointer to a string describing the error with code `errno`
- *test-error-open.c*

18 Environment variables

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

```
extern char ** environ;
```

- `environ` points to a NULL-terminated array of pointers to strings.
- the function

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

  accessible by including

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

  returns the string of the variable `name`

  `test-env.c`

19 The `make` utility

Introduction to `make`

- A large project may be built by linking very many object files "`.o`"
- If a project is built by linking many objects, it may be complicated to remember all *dependencies*
  - when the code of the module "A" is modified, what other modules need to be recompiled?
- 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
- `make` is a command that provides the support to re-compile only the source files that are affected by the change
- More in general, `make` can help to automate sequences of commands which may depend on each other

Basic usage of `make`

- By launching
  `make hello`
  the utility `make` interprets `hello` as an executable to be made
  - If `hello.c` is **not** present it returns an error
    `make: *** No rule to make target 'hello'. Stop.`
  - If `hello.c` is present in the current directory it compiles by
    `gcc hello.c -o hello`
  - If `make hello` is launched again, `hello.c` is **not re-compiled** again.
    `make: 'hello' is up to date.`
- Compiling by `make` is influenced by the some environment variables:
  - `CC`: is the string with the name of the C compiler
  - `CFLAGS`: is added to the compilation flags
  - `LDFLAGS` is added to the linking flags (example: `LDFLAGS="-lm"`)
- Try

1. `export CFLAGS="-std=c89 -pedantic"`
2. `make hello`

- in thise case `hello` depends on `hello.c` only
- `make` always prints to `stdout` what it did

Targets and implicit rules

- Whatever follows `make` is called the *target* of `make`
  - `make hello`
    "`hello`" is the target to be made
- Unless "explicit rules" are set (see next slides), `make` tries to guess how to make a given target from its name
  1. `make hello`
     tries to make the executable `hello` from the object file `hello.o` or from the source `hello.c` by
     `gcc hello.c -o hello`
  2. `make hello.o`
     tries to make the object file from a source file `hello.c` by
     `gcc -c hello.c -o hello`
- The ones above are called *implicit rules*: the rule to make the target is guessed from its name if standard naming is adopted

Explicit rules: `Makefile`

- Some executable may depend on object files (compiled modules) in a way that cannot be guessed by make
- Explicit rules explain how to make non-implicit, project-dependent targets
- Explicit rules are described in a text file with name "`Makefile`" or "`makefile`", which is searched by `make` in the current directory
- If the `Makefile` has an explicit rule for a given target, the corresponding implicit rule (if any) is overridden

`Makefile`: syntax

- Example of `Makefile` for the "`matrix`" module. Remember:
  - *matrix.h*, the header file of the module (the interface)
  - *matrix.c*, the implementation of the module
  - *application.c*, an example of code using the module

  Makefile
- Everything from the charater `#` until the end of line is a comment
- The `Makefile` may optionally start with project-dependent declarations of variables
- The format of an explicit rule is:

  ```
  target : ingredientA ingredientB    # dependencies
  [TAB]   recepie-to-make-target
  ```

  The recepie **must** start after the TAB character (not 8 spaces)

Makefile: invoking an explicit rule

- If the following explicit rule is listed in the `Makefile`

  ```
  target : ingredientA ingredientB    # dependencies
  [TAB]   step1
  [TAB]   step2
  [TAB]   step3
  ```

  then by launching
  `make target`

  - if a file "`target`" exists and is more recent than "`ingredientA`" **and** "`ingredientB`" then nothing is made (it means that "`target`" was made already)
  - otherwise `make` is invoked for any ingredient that is newer than `target`
  - when all ingredients are made, all commands (`step1`, `step2`, `step3`) are executed

- If no "ingredient" is specified, then the commands are always executed

- Try to modify some files of the "`matrix`" module and launch `make`

20 Process control

20.1 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

```
#include <unistd.h>

pid_t fork(void);
```

- The child process is a copy of the parent
  - the OS makes a **copy of all memory** of the parent process: 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 (semaphores, etc.)
- 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
  6. trying to fix bugs too quickly, may actually inject more bugs

20.2 Waiting for termination of child processes

Waiting for child termination by `wait(NULL)`

- The parent can wait for the termination of any child process by invoking the system call (better if the parent process always does wait for child processes)

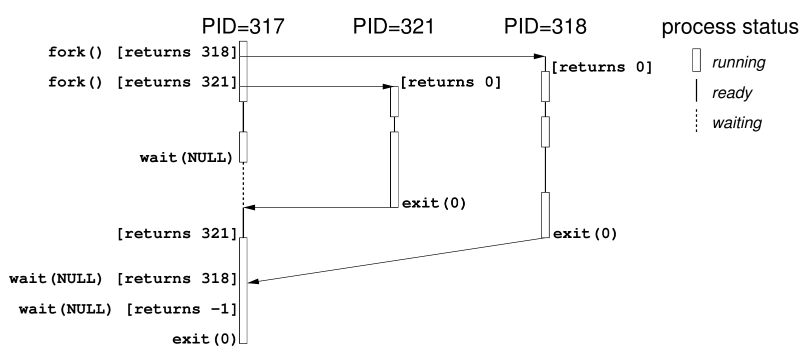
```
pid_t wait(NULL)
```

- `wait(NULL)` returns
  - `-1` if all child processes are terminated (or never had any child process) or
  - the PID of any terminated child process
- Standard code to wait for the termination of all child processes is

```
while (wait(NULL) != -1);
```

- *`test-fork-for-wait.c`*

`wait(NULL)`: possible interactions



`wait(NULL)` is a blocking system call

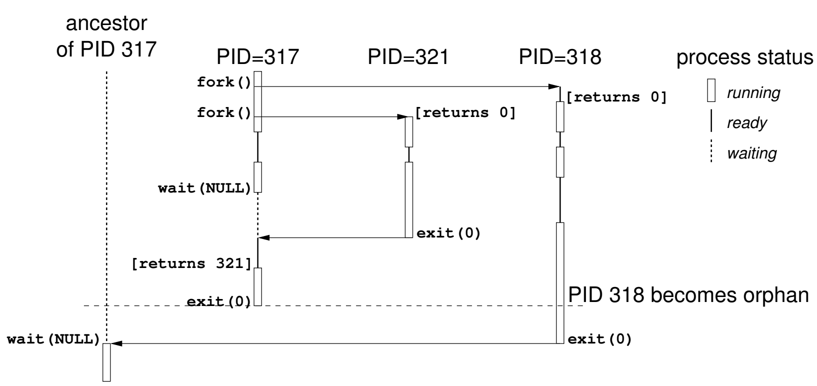
1. the parent process for any child process to terminate
2. it returns the PID of any terminated child
3. it returns `-1` if the process has no child

Process termination

- A process terminates and "returns a status" when
  1. it is returned from the `int main(...)` function by `return status`
  2. the system call `exit(status)` is invoked
- 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)
- A process may also be terminated by a signal (example: sent by pressing `Ctrl+C`). If so, no status returned
- When a process terminates
  1. all streams are flushed, and all open 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 to 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

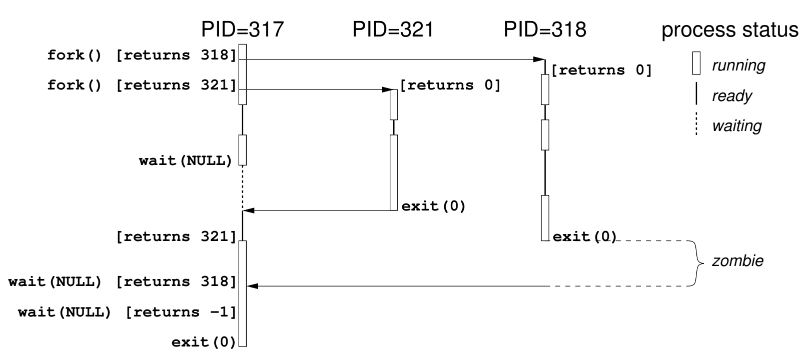
Orphans: parent process terminating before children

- The parent and the child processes are different and will terminate at different instants
- If a parent terminates before the child, all child processes become *orphans*.
- A new process (either `init` with PID 1 or some ancestor, depending on the OS) will adopt all of them



Zombies: process terminates before parent's `wait(...)`

- From termination to the parent's `wait()`, the process is a "zombie".



- 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
- This entry is held by the OS until the parent executes a `wait()`.

Zombies are not healthy

- A zombie cannot be killed by any signal (not even `SIGKILL`). They exist to 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) also becomes orphans:
  - when the ancestor adopts zombie processes, it 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 by holding a PID, 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: `wait(NULL)` allows the parent to be certain that the returned PID terminated

Retrieving more information about the child termination

- The parent process can get information about the terminated child process by:

```
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

	15 ← *bits* → 8	7 0
Normal termination	exit status (0-255)	0

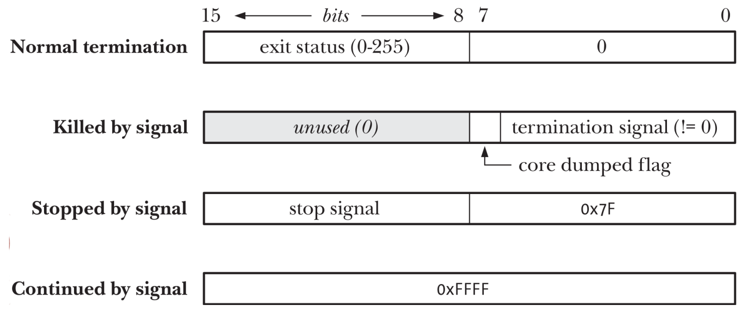
- there is a more comprehensive way to wait for child processes by `waitpid()` (iilustrated next)
- the macro `WEXITSTATUS(status)` extracts the status from the value `status` written by `wait(&status)`
  - the macro `WEXITSTATUS(status)` is just

```
#define WEXITSTATUS(x) ((x) >> 8)
```

test-fork-wait.c

Extracting information from the returned status

- Once it is returned from `wait(&status)`, 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`

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, the parent process does `not` wait 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

test-fork-waitpid.c

20.3 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`*

21 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

21.1 Sending signals

Sending a signal to any process

1. From a C program

```
#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`
    (a) if `(errno==ESRCH)`, `pid` doesn't exist
    (b) if `(errno==EPERM)`, `pid` exists but no permission to send signals
    (c) if successful, `pid` exists and we can send signals
- `pid`, the target process
  - if -1 the signal is sent to all (allowed) processes

2. From command line: `kill`, signal, and PID
`kill -INT <PID>`
`kill -SIGINT <PID>`
`kill -2 <PID>`

- If no signal is specified, then SIGTERM is the default
- `sudo kill -9 -1` is an interesting experience. . .

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);
```

asks the OS to send me `SIGALRM` 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

21.2 Handling signals

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
- Some signals (example: `SIGKILL`) **cannot** be handled by the user. Otherwise, an immortal process may be allowed by handling `SIGKILL`
- **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

```
#define _GNU_SOURCE /* necessary from now on */
#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`

```
sigaction(signum,&new,NULL); /*set new handler*/
sigaction(signum,NULL,&old); /*get cur handler*/
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; /* illustrated later */
  int sa_flags;     /* illustrated later */
  /* plus others (for advanced users) */
};
```

- `sa_handler` is a pointer to a function declared as

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

- If standard behavior is required, all bytes of "other fields" must be set to `0`

  test-signal-fpe.c

  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 `SIGALRM` is going to be handled, first attach the handler to the signal, then invoke `alarm(...)`)
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) {
   /* signal signum 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

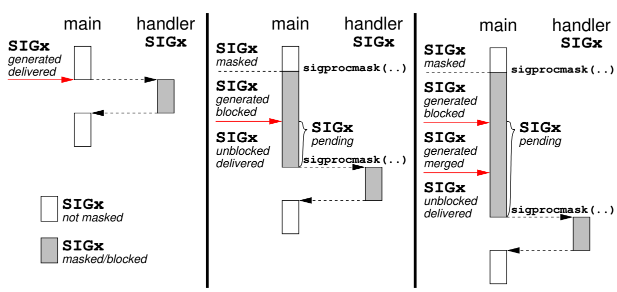
Global variables and signal handlers

- 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(...)` is **AS-Unsafe**, because it uses global variables (the output buffer)
- Every time you use any library function in a signal handler **check** that it is **AS-safe** at GNU libc documentation
- `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

21.3 Lifecycle of signals: delivering, masking, merging

Signals lifecycle

- Signals are *generated* by hardware or software events and are sent to a process
  1. Any process may set a *signal mask* to postpone signal handling
  2. if a signal arrives to a process when it is masked, the the signal is
  3. signals may be *blocked*
  4. if the signal is blocked, then it remains *pending*, until it is *unblocked*. As soon as unblocked, it is immediately *delivered* to the process,
  5. if a signal is generated again while it is already pending, then it is *merged*: after unblocked, it will be delivered only **once**!!
  6. if the signal is not blocked, it is immediately *delivered* to the process.



Setting signal masks

- During its execution, each process has its own *signal mask*
- a child process inherits the parent's signal 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 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

Signal mask during a handler

- The signal that triggered the handler is masked during the handler
  - unless the flag `SA_NODEFER` is set (to be explained later)
- `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

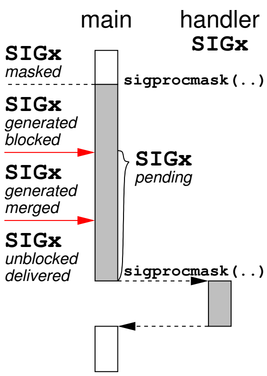
```
/* How to mask a signal during SIGINT handler */
struct sigaction sa;
sigset_t  my_mask;

bzero(&sa, sizeof(sa));      /* clean sa struct */
sa.sa_handler = handle_signal;   /* set handler */
sigemptyset(&my_mask);     /* Set an empty mask */
```

```
/* Add a signal to the sa_mask field struct sa  */
sigaddset(&my_mask, signal_to_mask_in_handler);
sa.sa_mask = my_mask;
/* Set the handler */
sigaction(SIGINT, &sa, NULL);
```

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

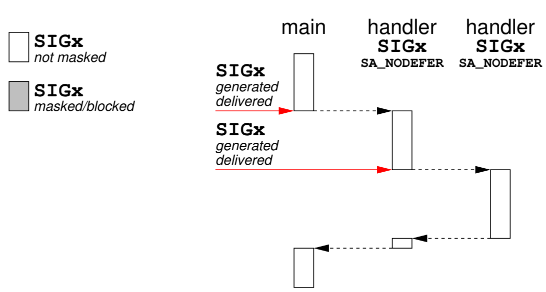


Unblocking the delivered signal during its own handler

- When a signal `signum` is delivered to a process, during its handler, the signal `signum` is automatically blocked until the handler returns
- The default behavior may be changed by setting the `SA_NODEFER` flag of the `struct sigaction`

```
bzero(&sa, sizeof(sa));
sa.sa_handler = handle_signal;
sa.sa_flags = SA_NODEFER; /* nested signals */
sigaction(SIGUSR1, &sa, NULL);
```

- *test-signal-nodefer.c*



21.4 Getting a signal when in waiting state

Entering the waiting state: pausing, sleeping

1. `pause()`

```
#include <unistd.h>

int pause(void);
```

the process stays in *waiting* state until a signal is caught

2. `sleep(sec)` (deprecated)

```
#include <unistd.h>

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

the process sits in *waiting* state for `sec` seconds. If the process caught a signal while sleeping, then `sleep` returns the remaining second to sleep
Deprecated because from `man 3 sleep`: "`sleep()` may be implemented using `SIGALRM`; mixing calls to `alarm()` and `sleep()` is a **bad idea**."

3. `nanosleep`

```
#include <time.h>

struct timespec my_time;

my_time.tv_sec = 1;
my_time.tv_nsec = 234567000;
nanosleep(&my_time, NULL);
```

sleeps for `1.234567` seconds

"Synchronization" by `sleep`/`nanosleep`

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

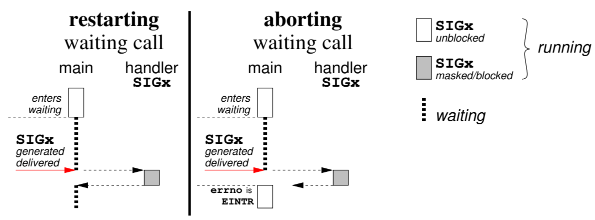
```
...
if (fork()) {
    /* PARENT */
    sleep(10);
    /* now the child has finished */
} else {
    /* CHILD */
    /* do my work before the parent check */
}
```

- 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`
- `nanosleep` is used mostly to show output slowly to the user

Delivery of signals to a *waiting* 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 *waiting* process?
  - "waiting process": process waiting on `wait()`, on `pause()`, or `sleep()`, etc... Counted as "sleeping" in `top`

  1. the process is not executing, since it is waiting 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:
     - **restarting**: the system call is restarted when the handler returns, **OR**
     - **aborting**: the system call aborts, `errno` is set to `EINTR`



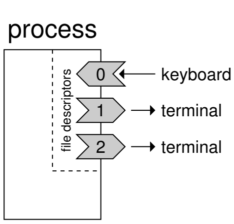
Signals when waiting: selecting the desired behavior

- Which behaviour among "**restarting**" or "**aborting**"?
- The default is "**aborting**"
- If the "**restarting**" behavior is desired, then consider
  1. setting the flag `SA_RESTART` in the `sa_flags` field of the `struct sigaction`
  2. checking if `(errno == EINTR)` after the waiting call and possibly re-invoke the call
- Unfortunately, different calls have different behavior
- It is then recommended to check the full documentation at
  `man 7 signal`
  Section "Interruption of system calls and library functions by signal handlers"
- *`test-signal-when-wait.c`*

22 Pipes

File descriptors: indices of source/sink of bytes

- file descriptor 0: read bytes from keyboard
- fd 1: write to terminal
- fd 2: write (errors) to terminal

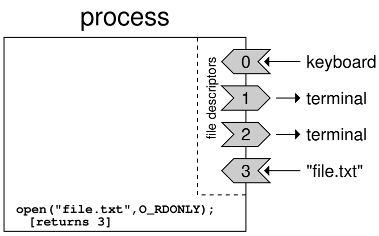


- If a process opens a file by

```
open("file.txt",O_RDONLY);
```

then it gets a new file descriptor (3 in the example)

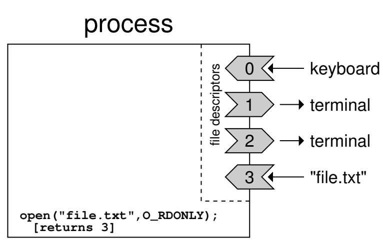
- the process can read from the file by specifying the file descriptor 3



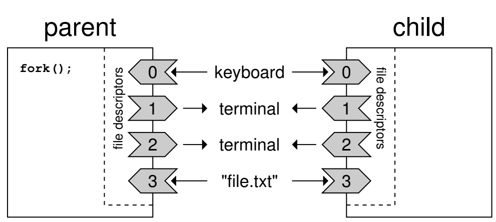
- file descriptors identify sources/sinks of bytes
- closing a file descriptor by `close(...)` means:
  1. to cut the link between the file descriptor and what it is linked to
  2. to release the entry in the file descriptor table

File descriptors: copied on `fork()`

- When a process forks a child, all file descriptors are copied
- Before `fork()`



- After `fork()`



- If a process closes its file descr. the others can still access

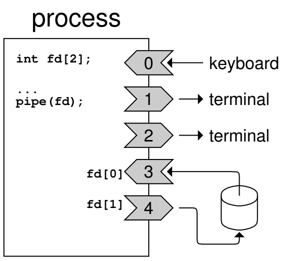
Pipes: the C interface

- Pipes are **uni-directional** byte streams
- Pipes are opened by

```
int pipefd[2]; /* declaring array of 2 int */

/* the call pipe sets two file descriptors */
pipe(pipefd);
```

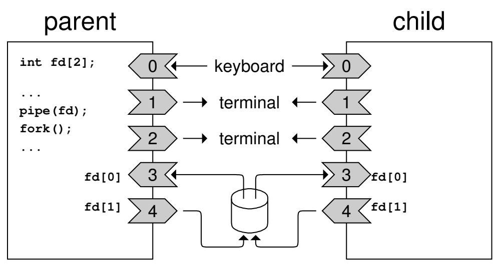
- if successful (by returning 0) it opens two file descriptors in `pipefd`:
  - `pipefd[0]` is fd of the read end of the pipe
  - `pipefd[1]` is fd of the write end of the pipe

  anything that is written to `pipefd[1]` can be read from `pipefd[0]`



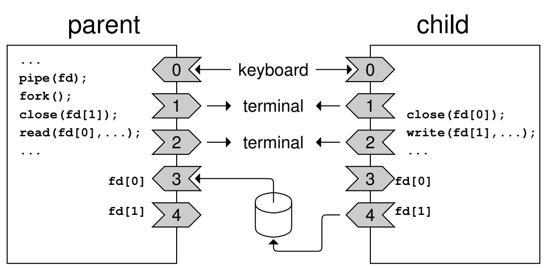
Two processes communicating via pipe [T]

- When a process forks a child after creating a pipe, a communication channel between parent and child is created



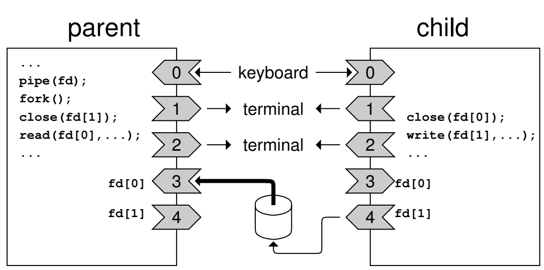
[T]

- If the two processes close the unused file descriptor, a uni-directional channel is created



- If unused file descriptors are not closed, then we run into problems (explained later)

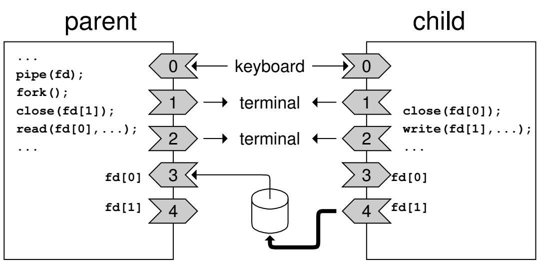
Reading from a pipe




```
char buf[100];  /* stores bytes read from pipe */
int num_bytes;

num_bytes = read(pipefd[0], buf, sizeof(buf));
```

- Reading consumes the data, which will be unavailable for next `read()`
- After a `read(...)` from a pipe:
  - if data is present, it is stored in `buf`, returned number of read bytes
  - if no data and some write end is open, it waits for some writes
  - if no data and no write end is open, it returns zero

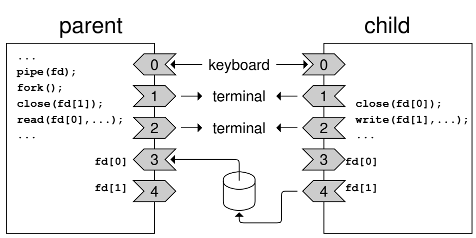
Writing to a pipe




```
char buf[100]; /* stores bytes written to pipe */
int num_bytes;

num_bytes = write(pipefd[1], buf, sizeof(buf));
```

- After a `write(...)` to a pipe:
  - if the pipe is full, the process waits for some `read(...)`
  - if enough space, it returns the number of bytes written
  - if no read end is open, a signal `SIGPIPE` (default action: `Term`) is generated (to notify that the written data will never be read)

Necessary to close unused file descriptors of pipes



- file descriptors of **unused write ends must be closed**
  - The "end-of-file" value is returned to the reader (`read()` returning 0) only when the **last** file descriptor of any writer is closed
  - if the write ends of a pipe are not closed, then the reader will wait on `read()` believing the some writer will `write()`
- file descriptors of **unused read ends must be closed**
  - When a writer tries to `write()` to a fd where all the readers have closed their read end, it gets a `SIGPIPE` signal
  - if some read end is left open, the signal `SIGPIPE` is not sent and the writer believes that somebody will read its data

Writing/Reading via pipes: examples

- Normally, the **writer** decides that a pipe is no longer needed
  1. the writer closes its write end
  2. the reader reads all the data until `read(...)` returns zero
- The size of the pipe is `PIPE_BUF` (4096 bytes on my machine):
  - reading/writing data not greater than `PIPE_BUF` is **atomic**
- If a process is waiting on `read(...)` or `write(...)` and it gets a signal, it returns `-1` and `errno` is set to `EINTR`
- Pipes do not allow file positioning (`lseek()` on a pipe will fail with `errno=ESPIPE`). Never try

```
lseek(pipefd[0],...);
```

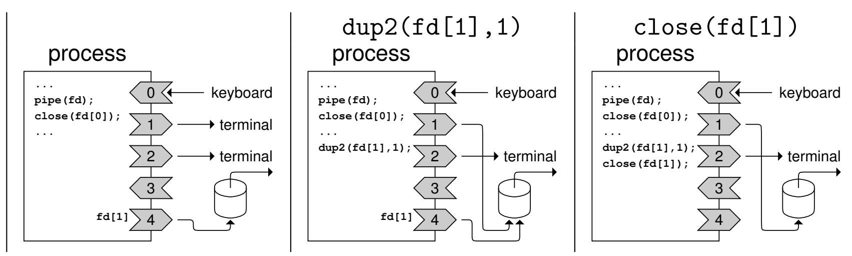
- Examples of pipe usage
  - *test-pipe-single.c*, single writer, single reader
  - *test-pipe-kids.c*, many writers, single reader, comment atomicity

Copying a file descriptor onto another

- It is possible to "copy"a file descriptor onto another one (which is then overwritten)

```
int dup2(int fd_src, int fd_dst);
```

  which copies `fd_src` onto `fd_dst`. If `fd_dst` was previously open, then `dup2()` also close it.
- Example of redirecting `stdout` to another process via pipe



- whatever the process sends to fd 1 (`printf()`, ...) goes to the pipe
- *test-fd-redir.c*

23 FIFOs

Pipes from command line

- `ls -latr` and `wc -w` are two commands
- by launching `ls -latr | wc -w`
  1. the shell forks two processes: PID1 and PID2
  2. the shell connects via pipe the output of PID1 to the input of PID2
  3. PID1 then runs `exec("ls -latr",...)`
  4. PID2 then runs `exec("wc -w",...)`
- we remark that the two process are not aware of the presence of the pipe. It is the shell (their parent process) which connected the streams differently

Pipes and Named pipes (FIFOs)

- Pipes are identified by file descriptors: they can be used only among processes sharing an ancestor
- Named pipes, called FIFO (First In First Out), solves this issue
- FIFOs are pipes with a global visible name in the file system
- Any process knowing the name of the FIFO can access to it

FIFO

1. Open two terminals: terminal A and terminal B, both well visible on the screen
2. term A: `mkfifo my-1st-fifo`
3. term A: `ls -latr`, you can notice "`p`" in the 1st column
4. term A: `ls > my-1st-fifo`, to write something to the pipe
5. term B: `cat my-1st-fifo`, to print the content of `my-1st-fifo`

- the last two commands can also be exchanged
- try with two terminals doing `cat my-1st-fifo`, and then one doing `ls > my-1st-fifo`
- Comments: the write blocks until some process reads and viceversa
- In C, a FIFO can be created by

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

  which creates a FIFO with at `pathname`, with read/write/execute permissions as specified by `mode`

24 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 `ipcs` shows the status of current IPC objects
  - the command `ipcs -l` shows the system limits on the resources
    * also available in the `/proc/sys/kernel/` file system
  - the command `ipcmk` creates an IPC object (only Linux, not standard)
  - 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 semget(key_t key, ...)` to get a semaphore
  3. `int shmget(key_t key, ...)` to get a shared memory
- 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 (Example: `msg` queues)

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);
     ```

Typical issues due to persistence

1. Say that your program creates and uses some IPC object
2. Say that your program crashes or it never ends and you have to stop it by `Ctrl+C`
3. Then you fix it and you launch it again

The IPC object of the second run still has the same content it had after the first run!!!

- Possible fixes:
  1. "get" the object with `IPC_PRIVATE` key it always returns a new object
     - may run out of memory
  2. install `Ctrl+C` handler that cleans up objects
  3. remove old objects at command line by `ipcrm`

25 System V: message queues

Queues vs. pipes

- Message queues offer an IPC facility similar to pipes

	pipes	**message queues**
unit of data	byte	message (any user-defined data structure)
terminology	write, read, file descriptors	send, receive, IDs
lifecycle	closed after all read-/write file descriptors are closed	*persistent*: stay alive even after all processes (creator, senders, receivers) terminates
read blocks	if empty & some write ends are open	always if empty
write blocks	if full	if full
deallocation	implicitly after all fd are closed	must be made **explicitly** by the user
abstraction	low	high

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
   - no `SIGPIPE`-like method when all read ends are closed (as in pipes)
5. Receiver processes cannot know when senders have finished
   - no `EOF`-like method when all write ends are closed (as in pipes)
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

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`
- After processes have finished sending, they cannot close their "write end" as in pipes
  - Message queues are "closed" (erased) separately once they are no longer needed

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 by the rules above
  - 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 of usage of message queues

- Sender process sends a message of type `argv[1]` to a queue. The text of the message is read from `stdin`

  test-ipc-msg-snd.c

- Receiver process receives a message of type `argv[1]` and prints its content to `stdout`. If a "special" message of type `MSGTYPE_RM` is received, then the message queue is erased

  test-ipc-msg-rcv.c

- they share a common header file

  ipc-msg-common.h

Example 2 of usage of a message queue

- The parent process:
  1. Create a queue
  2. Forks `NUM_PROC` sender child processes
  3. Forks a receiver process
  4. Waits for the sender processes to terminate
  5. Waits for the queue to be empty
  6. 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

26 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 an **array of semaphores**: each operation onto an array of semaphores is **atomic**
  - an array is useful for code fragments that need more than one resource
- The system call

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

  returns the identifier of an array 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

Operation on a single semaphore

- Access/realease of a semaphore are called *semaphore operations*
- An operation on a **single** 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;         /* 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 in `ops` ($\leq$ size of semaphores' array)
  - **Notice**: the `nops` operations are made all together atomically
  - The call blocks if **any** of the operations cannot be made

Sem. op. to access/release a resource

```
struct sembuf {
  unsigned short sem_num;     /* sem number */
  short          sem_op;   /* sem operation */
  short          sem_flg;          /* 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 array

Sem. op. to wait until semaphore is zero

```
struct sembuf {
  unsigned short sem_num;     /* sem number */
  short          sem_op;   /* sem operation */
  short          sem_flg;          /* 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 same 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

Don't wait forever

- If a resource protected by a semaphore is unavailable, a process may:
  1. wait until the resource is available again (as seen before), or
  2. decide to do something else
- The flag `IPC_NOWAIT` may be set in a semaphore operation

```
struct sembuf sop;
sop.sem_flg = IPC_NOWAIT;
semop(.., &sop, 1);/*dont wait*/
```

- When executing the `semop()`
  - if the resource is available, get it as usual
  - if unavailable don't wait, return `-1`, and `errno` set to `EAGAIN`
- If only waiting for some time is desired

```
#include <time.h>
struct timespec {
    time_t tv_sec; /* seconds */
    long tv_nsec; /* nanoseconds */ }
semtimedop(/*same as semop*/, struct timespec * timeout);
```

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);
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`
- Even if the flag `SA_RESTART` was set in the signal handler by the `sigaction(...)`, an interrupted `semop(...)` will always fail with `errno` set to `EINTR`

Wrong ways to wait for a semaphore

- Do not loop forever testing the value of a semaphore

```
sop.sem_flg = IPC_NOWAIT;
do {
    semop(.., &sop, 1);
while (errno == EAGAIN);
```

Semaphores: Examples

1. Tanti processi che vogliono cucinare condividendo le risorse di una cucina

   test-sem-cook.c

2. Processi figli che scrivono nella pipe in modo ordinato

   test-pipe-round.c
   which uses a small module for handling semaphores

   - *my_sem_lib.h* (header file)
   - *my_sem_lib.c* (implementation of the functions)

27 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 is 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

- Many child processes filling a shared table

- Each process needs to get a unique entry in the table, then it can write without conflict

  - *Makefile*
  - *test-shm.h*
  - *test-shm-parent.h*
  - *test-shm-child.h*

28 Debugging by `gdb`

Debugging

- Debugging is very helpful to find issues in programs

- `gdb` is the debugging engine

- as everything in Unix/Linux, it is very powerful and very cryptic

Launching `gdb`

1. As examples, we debug the code of the simple sender and receiver of messages

   - *ipc-msg-common.h*
   - *test-ipc-msg-snd.c*
   - *test-ipc-msg-rcv.c*

2. To properly debug a program, it must be compiled with the flags:

   - `-g`, to add extra information to the object files
   - `-Og`, the select the best code optimizations for the debugger (alternatively, even switching the optimizations off by `-O0` is a valid alternative)

3. To run a program within the debugger
   `gdb test-ipc-msg-snd`

4. Even if the executable should have some command-line options, just ignore it

5. By launching `gdb ...` the gdb environment opens

`gdb` commands 1/2

- It appears the prompt
  `(gdb)`
- here `gdb` commands may be entered
  - `help`, help on commands
  - `list`, list the source code
    - `list <line-number>`, list the source code starting from `<line-number>` of the current file being debugged
  - `break <line-number>`, insert a breakpoint at line `<line-number>` (always insert a breakpoint before running)
    - `break <filename>:<line-number>`, insert a breakpoint at line `<line-number>` in file `<filename>`
  - `info b`, show current breakpoints. Each breakpoint is identified by a numeric ID
  - `del <ID>`, delete the breakpoint number `<ID>`
  - `run`, run the executable (until the first breakpoint)
    - `run <command-line-args>`, run the executable with the specified command-line arguments

`gdb` commands 2/2

- `next`, execute a line of code: if a function call, invokes the call
- `step`, execute a line of code: if a function call, step in the function
- `cont`, continue the execution until the next breakpoint,
- `print <expression>`, evaluate and print `<expression>`
- `bt`, "backtrace" shows all the called functions on the stack
- `quit`, to quit the debugger

Attaching `gdb` to a running process

- It is also possible to attache `gdb` to a running process by
  `gdb -p <PID>`
- If the process is running some system call, you may need `sudo` superpowers by
  `sudo gdb -p <PID>`

Index