Compilation & Memory Management

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

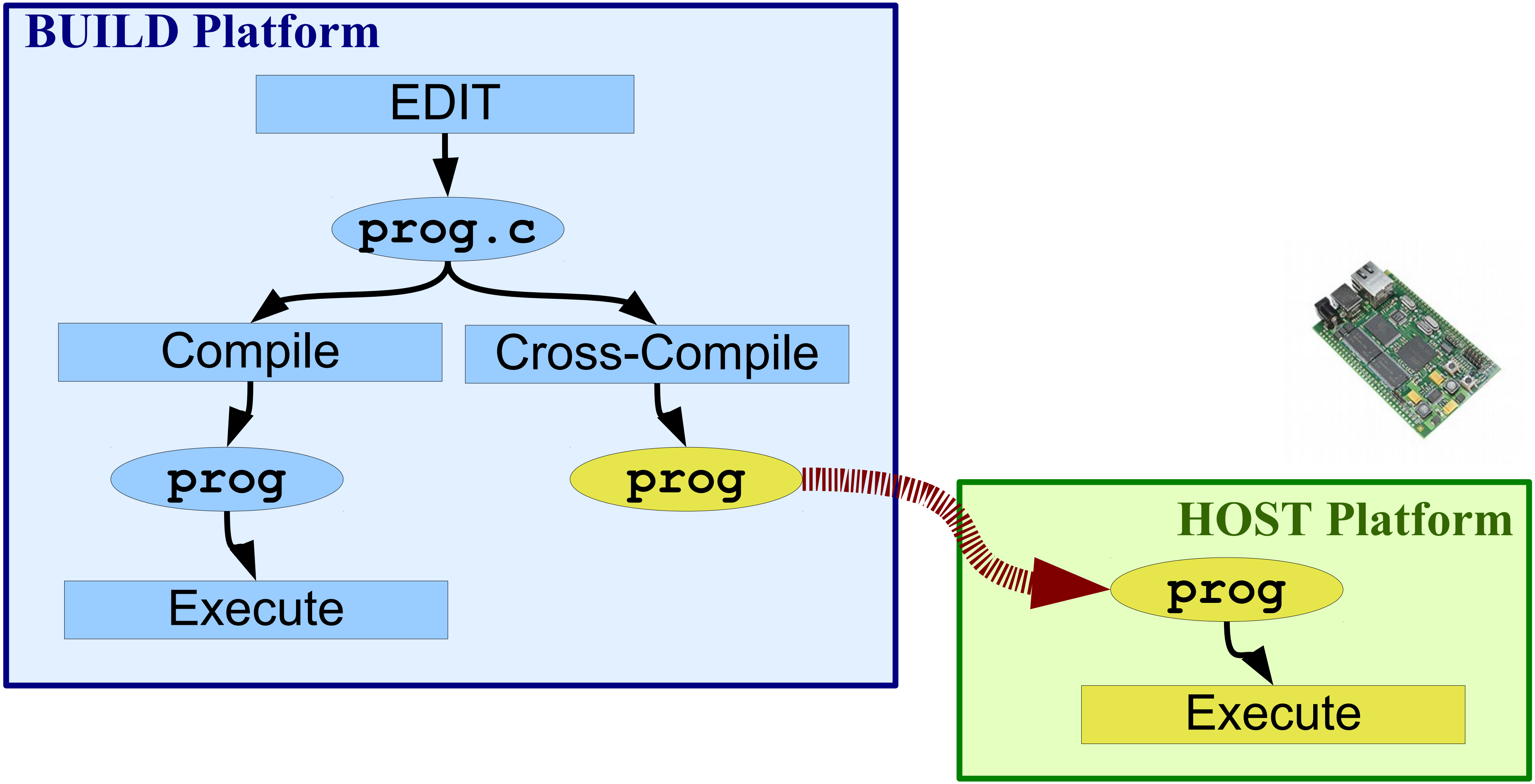
Mário de Sousa

msousa@fe.up.pt

Cross-Compiling

- On embedded systems, it is not always possible to have:
  - a compiler,
  - an editor,
  - a debugger,
  - ...
- How do we develop software for embedded systems?
  ans: Cross-compiling

Cross-Compiling



BUILD Platform

EDIT

`prog.c`

Compile

Cross-Compile

`prog`

`prog`

Execute

HOST Platform

`prog`

Execute

Compiling, Linking & Memory Management of C Programs

- Compilation of C Programs
- Shared Libraries
- Memory Management in C Programs
- Memory Management by Operating Systems

Programming in C

- Why C?
  - Most OSs in current use are written in C
  - Standard POSIX defines calls to OS in C language
    - POSIX – Portable Operating System Interface
    - POSIX - IEEE Std 1003.1
    - POSIX has Real-Time extensions available
- This lecture will NOT be teaching C!
  - We will focus on the compilation process of C programs

C programming language: Example

```
#include <stdio.h>

int main() {
  printf(“Hello World!”);
}
```

- C code organization is based on functions
- C programs start by executing the main() function
  - All C programs have one, and only one, main() function.

How to Execute Programs: Compiler

- CPUs do not understand high level programming languages (such as C)
- How can we execute a program written using a high level programming language?
  - By compiling it...
    - Use another program (i.e. a compiler) that converts the original program into another equivalent program that the CPU understands.
    - Every time you wish to execute the high level program, we simply execute instead the program crested by the compiler.
    - Example: C, Pascal, Fortran, Ada, ...

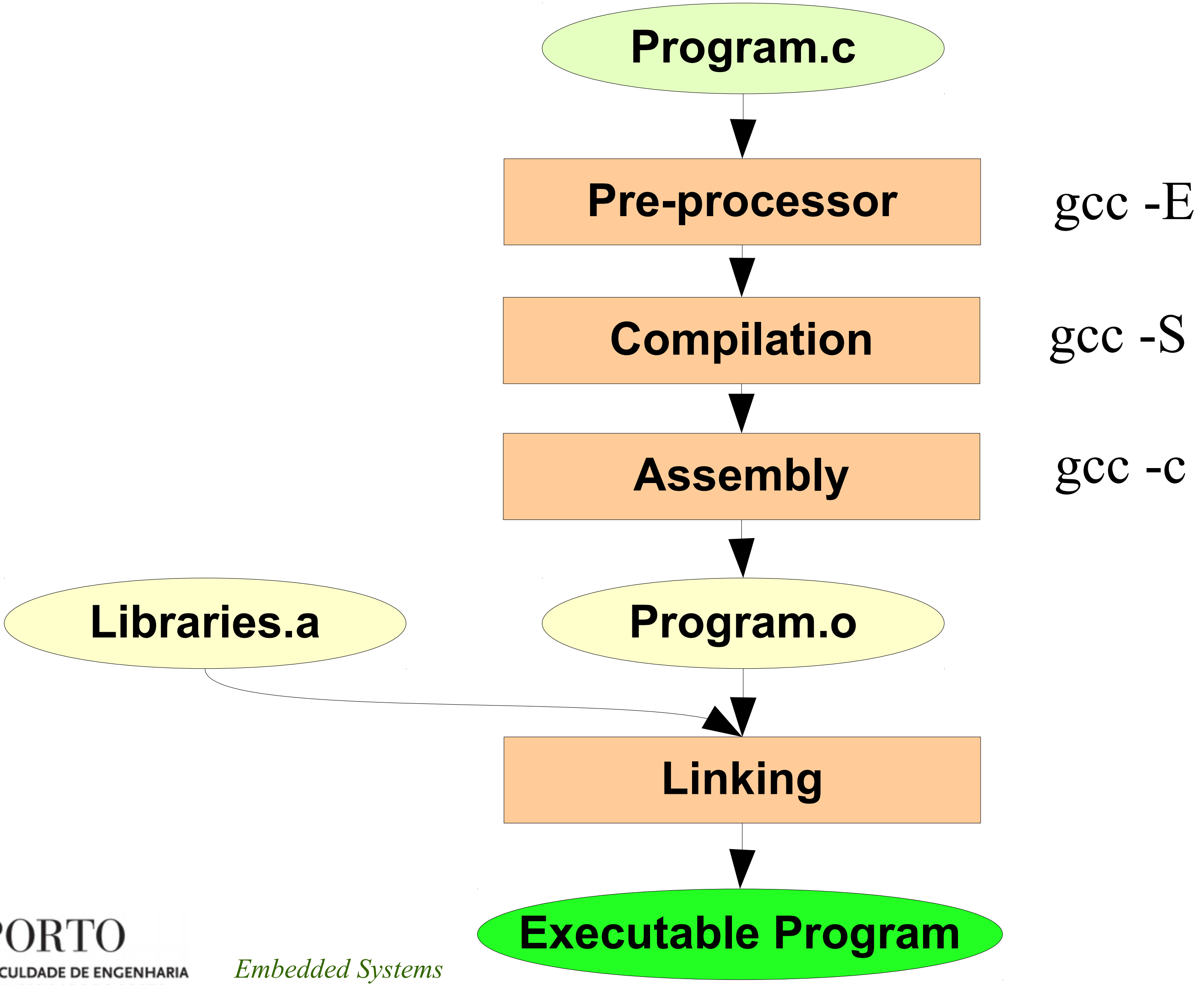
How to Execute Programs: Interpreter

- How can we execute a program written using a high level programming language?
  - By interpreting it:
    - Use another program (i.e. the interpreter) that interprets the high level instructions, and for each instruction will immediately execute the necessary CPU instructions that would produce the same effect.
    - Every time you wish to execute the high level program, we execute the interpreter and ask it to interpret the high level program.
    - Example: Python, Perl

How to Execute Programs: Virtual Machine

- How can we execute a program written using a high level programming language?
  - By compiling and interpreting the result!
    - Use a compiler that will generate an equivalent program in an intermediate level programming language (i.e. *bytecode*).
    - Every time you wish to execute the high level program, we execute the interpreter and ask it to interpret the intermediate level program.
    - Example: JAVA

Stages of Compiling a C Program


Program.c

Pre-processor

gcc -E

Compilation

gcc -S

Assembly

gcc -c

Libraries.a

Program.o

Linking

Executable Program

Stage 1: Pre-processing

- Processing done purely at the textual level:
  - Remove comments
  - Process pré-processing directives
    (i.e. those commands preceded by '#', ex: '#include')

- Example:

```
#include <stdio.h>
#define MESSAGE “Hello World!”

int main() {
  printf(MESSAGE);
}
```

Stage 1: Pre-processing

- Pre-process...

  ```
  gcc -E hello.c -o hello.cpp_out
  ```

- Result:
  - The command '#include <stdio.h>' is replaced by the contents of the file '/usr/include/stdio.h'
  - This file includes other '#include' commands, which menas other files will be included in the resulting output!
  - All occurrences of 'MESSAGE' will be replaced by '”Hello World!”'

    ```
    printf(“Hello World!”);
    ```

Stage 2: Compiling

- Converts the C program into Assembly

- Compile...

```
gcc -x cpp-output -S hello.cpp_out
```

(or more simply)

```
gcc -S hello.c
```

- The resulting file (hello.s), contains:

```
LC0:
 .ascii "Hello, world!\12\0"
main:
 pushl   %ebp
 movl    %esp, %ebp
 subl    $8, %esp
 andl    $-16, %esp
```

```
movl    $0, %eax
subl    %eax, %esp
subl    $12, %esp
pushl   $.LC0
call    printf
addl    $16, %esp
leave
ret
```

Stage 3: Assembly

- Convert the Assembly code into Machine code
  - The resulting machine code is relocatable:
    i.e. it is independent on the memory address on which it is located.
    (by using, for example, relative jumps, and including unresolved symbols...).

- Assembling...

```
gcc -x assembler -c hello.s
```

(or more simply)

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

Stage 3: Assembly

- Analyses of resulting file hello.o

```
readelf -h hello.o
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00
  Class:       ELF32
  Data:        2's complement, little endian
  Version:     1 (current)
  OS/ABI:      UNIX - System V
  ABI Version: 0
  Type:        REL (Relocatable file)
  Machine:     Intel 80386
  Version:     0x1
  Entry point address:  0x0
(...)
```

ABI-> Application Binary Interface

Reversing Stage 3: Dis-assembly

- `Objdump -d hello.o`

```
Disassembly of section .text:
00000000 <main>:
 0:   55             push   %ebp
 1:   89 e5          mov    %esp,%ebp
 3:   83 ec 08       sub    $0x8,%esp
 6:   83 e4 f0       and    $0xfffffff0,%esp
 9:   b8 00 00 00 00 mov    $0x0,%eax
 e:   29 c4          sub    %eax,%esp
10:   83 ec 0c       sub    $0xc,%esp
13:   68 00 00 00 00 push   $0x0
18:   e8 fc ff ff ff call   19 <main+0x19>
1d:   83 c4 10       add    $0x10,%esp
20:   c9             leave
21:   c3             ret
```

Comparing...
Assembly with Disassembly

`hello.s`

```
main:
 pushl   %ebp
 movl    %esp, %ebp
 subl    $8,   %esp
 andl    $-16, %esp
 movl    $0,   %eax
 subl    %eax, %esp
 subl    $12,  %esp
 pushl   $.LC0
 call    printf
 addl    $16,  %esp
 leave
 ret
```

`hello.o` (disassembled)

```
<main>:
push %ebp
mov  %esp,%ebp
sub  $0x8,%esp
and  $0xfffffff0,%esp
mov  $0x0,%eax
sub  %eax,%esp
sub  $0xc,%esp
push $0x0
call 19 <main+0x19>
add  $0x10,%esp
leave
ret
```

Comparing...
Assembly with Disassembly

`hello.s` → **`hello.o`** (disassembled)

```
main:
 (...)
 pushl   $.LC0
 call    printf
```

```
<main>:
(...)
push $0x0
call 19 <main+0x19>
```

- Machine code in hello.o is **NOT** executable!
  - It contains unresolved symbols
    - ex. `$.LC0` (reference to string `“Hello World!”`)
  - Contains addresses as an offset to location of `main()`, these are not the final addresses (reclocatable).
    - ex. `call 19 <main+0x19>`

Stage 4: Linking

- Linking relocatable code of several files and libraries. May or may not result in executable code.
- linking...

  ```
  gcc -static -o hello hello.o
  ```

  (or more simply)

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

- Result: executable file → hello

Stage 4: Linking

- Analysing the resulting hello file...

```
readelf -h hello
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00
  Class:       ELF32
  Data:        2's complement, little endian
  Version:     1 (current)
  OS/ABI:      UNIX - System V
  ABI Version: 0
  Type:        EXEC (Executable file)
  Machine:     Intel 80386
  Version:     0x1
  Entry point address:   0x8048100
(...)
```

Stage 4: Linking

- Analysing the resulting hello file...

```
objdump -d hello
8048204 <main>:
8048204: 55             push %ebp
8048205: 89 e5          mov  %esp,%ebp
8048207: 83 ec 08       sub  $0x8,%esp
804820a: 83 e4 f0       and  $0xfffffff0,%esp
804820d: b8 00 00 00 00 mov  $0x0,%eax
8048212: 29 c4          sub  %eax,%esp
8048214: 83 ec 0c       sub  $0xc,%esp
8048217: 68 c8 fa 08 08 push $0x808fac8
804821c: e8 9f 05 00 00 call 80487c0 <_IO_printf>
8048221: 83 c4 10       add  $0x10,%esp
8048224: c9             leave
8048225: c3             ret
```

Comparing...

hello.s → **hello.o** → **hello**

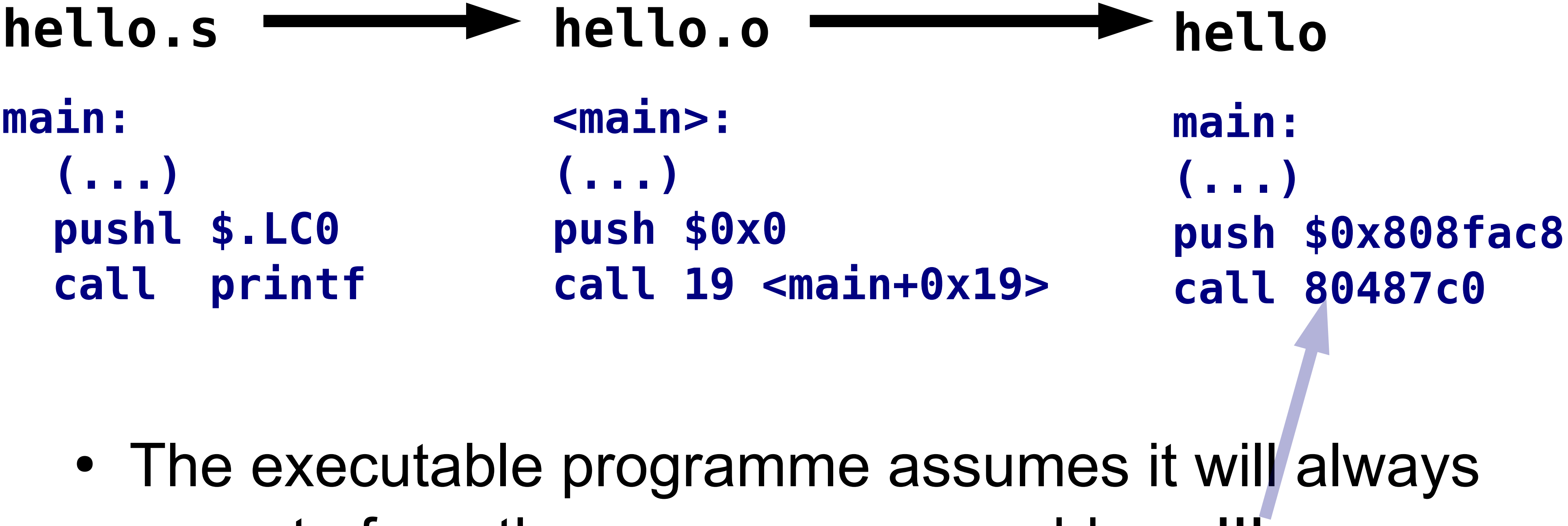
```
main:
  (...)
  pushl $.LC0
  call  printf
```

```
<main>:
(...)
push $0x0
call 19 <main+0x19>
```

```
main:
(...)
push $0x808fac8
call 80487c0
```

- Machine code of hello file **is executable**!
  - All symbols were replaced by memory addresses
    - ex. `$.LC0  -  $0x808fac8`
  - The addresses are absolute
    - ex. `call 19 <main+0x19>  -  call 80487c0`

Comparing...



hello.s

```
main:
  (...)
  pushl $.LC0
  call  printf
```

hello.o

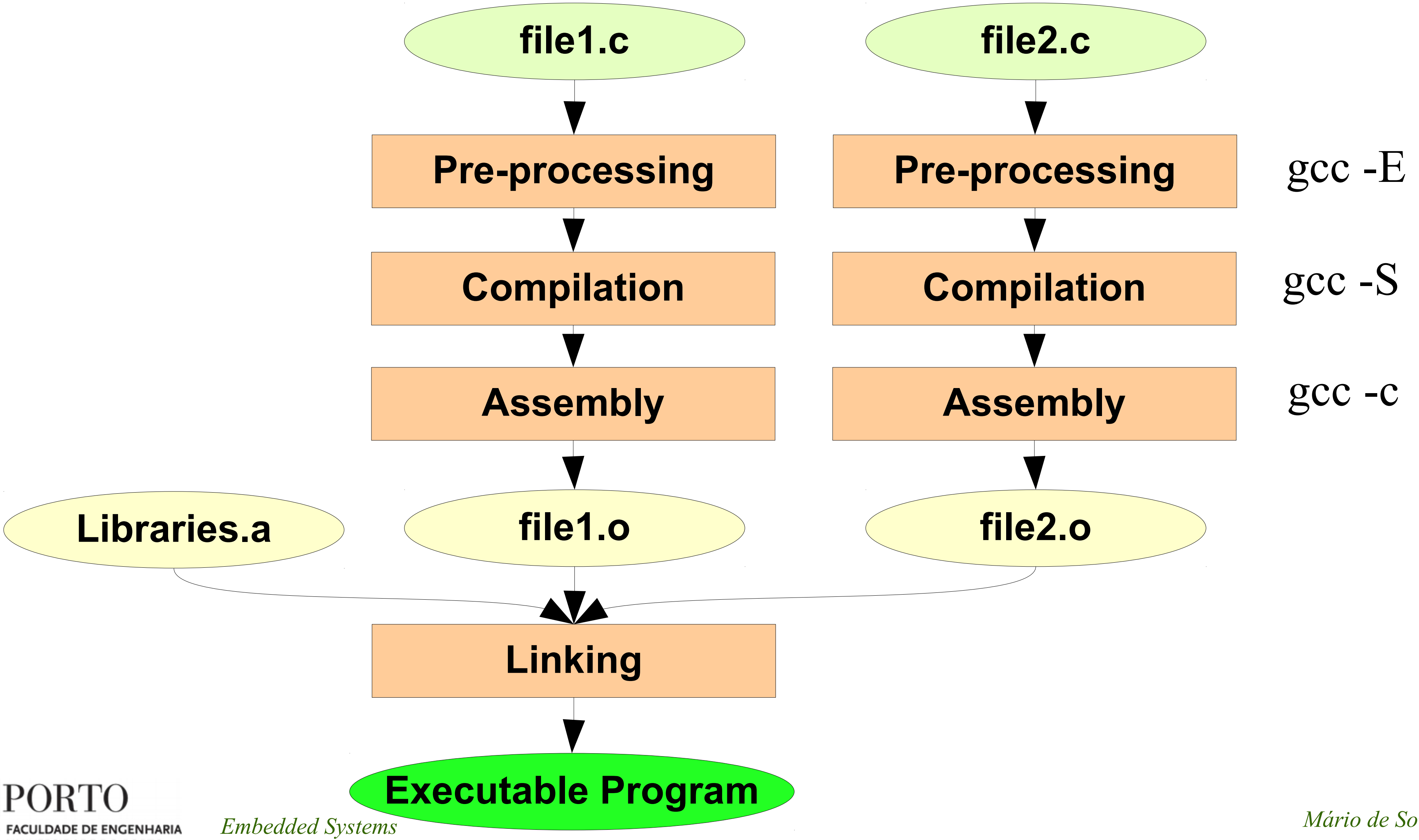
```
<main>:
(...)
push $0x0
call 19 <main+0x19>
```

hello

```
main:
(...)
push $0x808fac8
call 80487c0
```

- The executable programme assumes it will always execute from the same memory address!!!
- What if we want to execute several instances simultaneously?

Stages of Compiling a C Program



file1.c

file2.c

Pre-processing

Pre-processing

gcc -E

Compilation

Compilation

gcc -S

Assembly

Assembly

gcc -c

Libraries.a

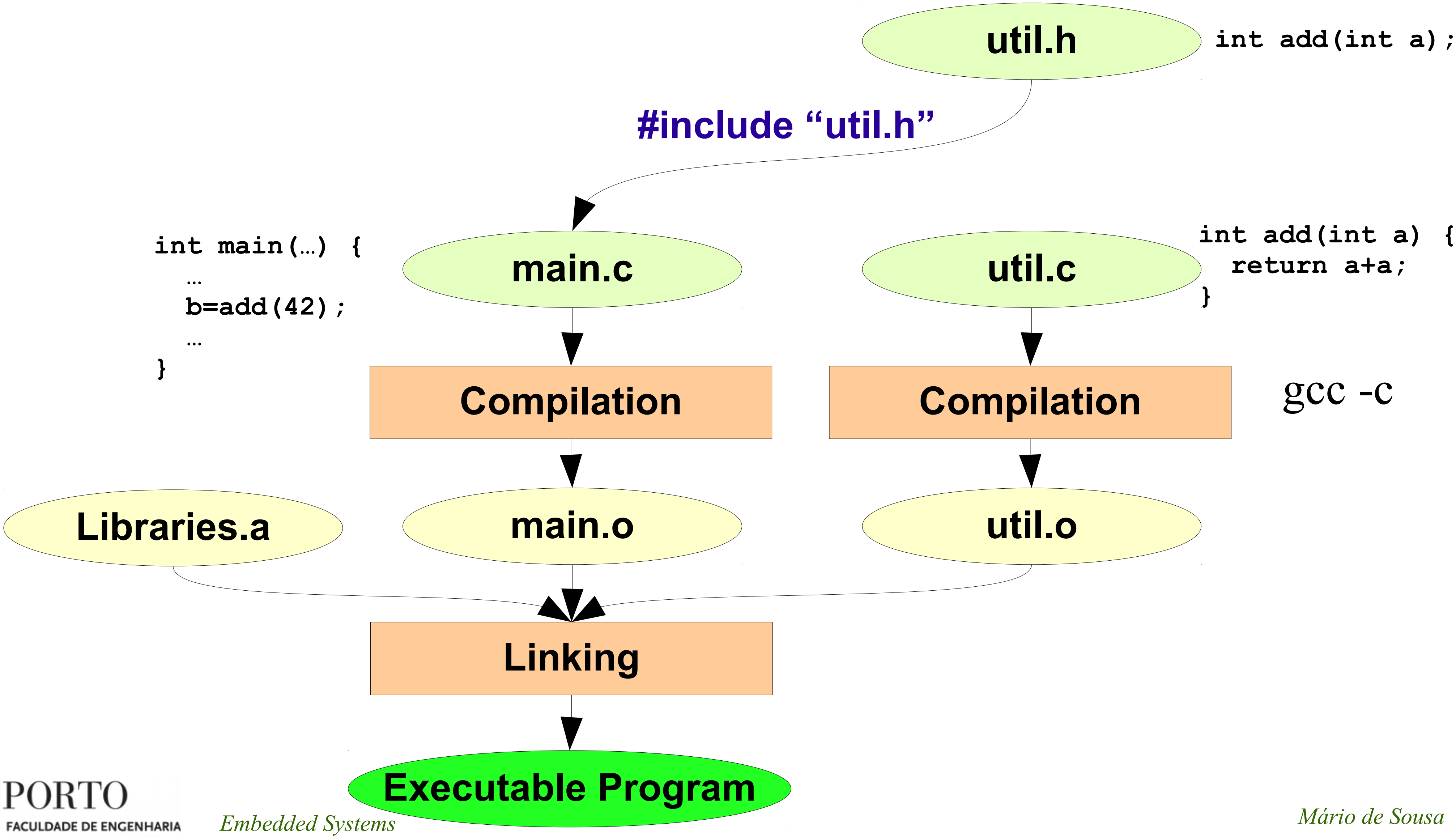
file1.o

file2.o

Linking

Executable Program

Stages of Compiling a C Program



util.h

```
int add(int a);
```

#include “util.h”

```
int main(…) {
  …
  b=add(42);
  …
}
```

main.c

util.c

```
int add(int a) {
  return a+a;
}
```

Compilation

Compilation

gcc -c

Libraries.a

main.o

util.o

Linking

Executable Program

Compiling, Linking & Memory Management of C Programs

- Compilation of C Programs
- Shared Libraries
- Memory Management in C Programs
- Memory Management by Operating Systems

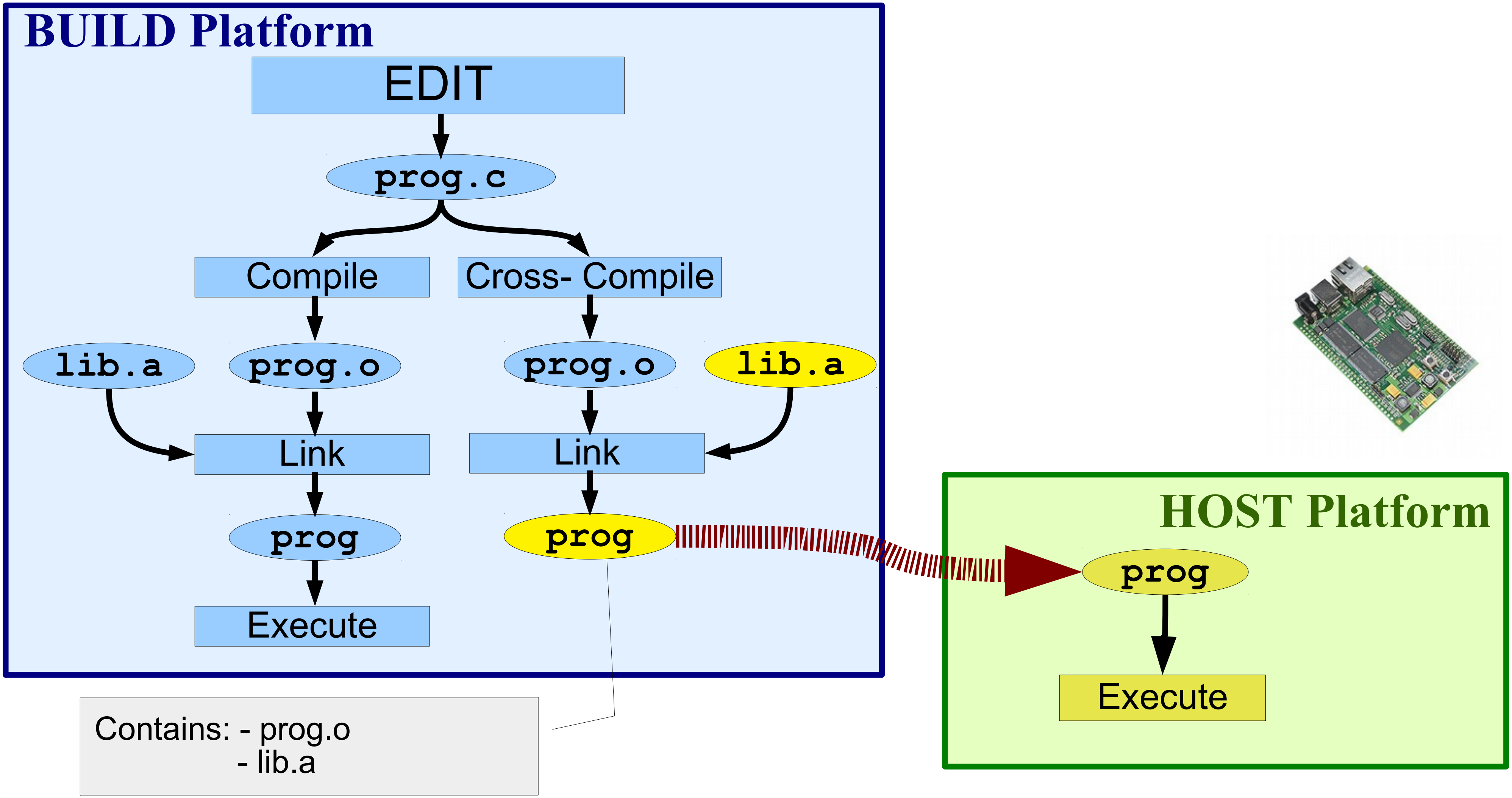
Using Libraries

- hello.c uses the `printf()` function from the C library
- Option 1 – **Static linkage**
  - The `printf()` function is copied into the final executable file (hello) during linking
  - `gcc -static -o hello hello.o`
- Option 2 – **Dynamic Linkage**
  - The executable file merely contains a reference to the library that contains the code for `printf()`
  - The code for the `printf()` function is read from the library every time the executable 'hello' program is run.
  - `gcc -o hello hello.o`

Dynamic Linking

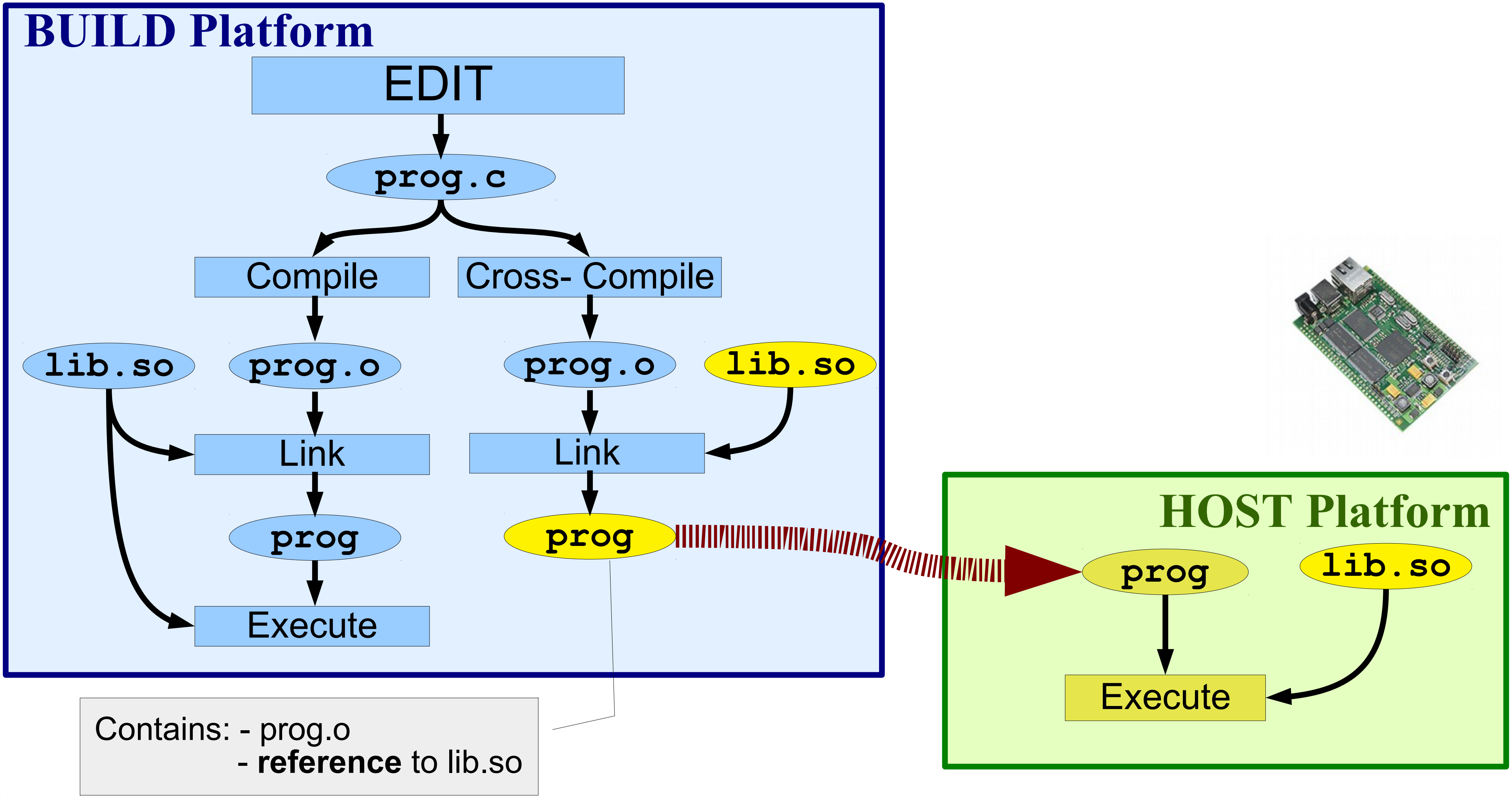
- Libraries for dynamic linking
  - In UNIX environments, these are known as S*hared Libraries.* In Linux, the files have the '.so' extension (*Shared Object*)
  - In Windows, these are known as DLL – *Dynamically Linked Library*

Cross-Compiling with Static Libraries



Contains: - prog.o
- lib.a

Cross-Compiling with Dynamic Libraries



Contains: - prog.o
- **reference** to lib.so

Linking
Static vs Dynamic

- Disk usage
  - Static linking results in larger executable program files. The same code is stored multiple times, once in each program that uses that function!
  - hello: dynamic → 11 kB, static → 475 kB.

- Removal of bugs in the library
  - Dynamic linking only requires that the libraries themselves be recompiled
  - Static linking requires recompilation of all programs that use the library.

Linking
Static vs Dynamic

- Timing Overheads
  - Lazy loading of dynamically linked libraries (delay loading the library until the time when the function is first called)
    => longer initial execution TIME
    dangerous for programs with Real-Time requirements.
  - Enabling/Disabling lazy binding → at **compile time**

```
gcc -o hello hello.o -Wl,-znow     ← load all libraries at startup
gcc -o hello hello.o -Wl,-zlazy    ← use lazy binding
```

NOTES

- ‘gcc’ does not do the linking. It simply calls the default **linker**, usually ‘ld’
- any text following ‘-Wl,’ is passed directly to the linker ‘ld’, and will be interpreted as a linker command line option
- ‘ld .. -znow’ ← link and set executable to NOT use lazy binding when it is run
- ‘ld .. -zlazy’ ← link and set executable to USE lazy binding when it is run

Linking
Static vs Dynamic

- Timing Overheads
  - Lazy loading of dynamically linked libraries (delay loading the library until the time when the function is first called)
    => longer initial execution TIME
    dangerous for programs with Real-Time requirements.
  - Enabling/Disabling lazy binding → at **execution time**

```
gcc -o hello hello.o  ← compiled to use use lazy binding (default)
set LD_BIND_NOW = 1   ← disable lazy binding
./hello               ← run program
```

NOTES

- The dynamic libraries are loaded at runtime by the dynamic **loader** ‘ld.so’
- The dynamic loader ‘ld.so’ follows the options set in environment variables
- Environment variable ‘`LD_BIND_NOW`’ disables lazy binding
- Environment variable ‘`LD_DEBUG`’ enables debug mode

Dynamic Linking

- How can you know which libraries a compiled program uses?

```
$ gcc -c -o hello hello.o
$ ldd hello

libc.so.6 => /lib/i686/libc.so.6(0x40024000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

- What is this library → `ld-linux.so` ?
  - Contains the code responsible for loading into memory the functions in shared libraries!
  - As an exception, this shared library is loaded into memory directly by **`execve()`**

Dynamic Linking

- More command line utilities...

`ar`	creates static libraries.
`objdump`	used to display all the information in an object binary file.
`strings`	list all the printable strings in a binary file.
`nm`	lists the symbols defined in the symbol table of an object file.
`ldd`	lists the shared libraries on which the object binary is dependent.
`strip`	deletes the symbol table information.

Dynamic Linking:
controlled by program at runtime

- To programmatically load a dynamic library:
  Linux/POSIX

```
// Load the library
void* my_library = dlopen("my_lib.so", RTLD_LAZY);
if (my_library == NULL) ERROR;

// Load reference to variable
int *my_int_var_ptr = (int *)dlsym(my_library,"my_int_var");
if (my_int_var_ptr == NULL) ERROR;
*my_int_var_ptr = 42;  ← Access the variable!

// Load reference to function
typedef void (*my_func_ptr_t)(void);
my_func_ptr_t my_func = (my_func_ptr_t)dlsym(my_library,"my_func");
if (my_func == NULL) ERROR;
my_func();   ← Call the function!

// Unload the library
dlclose(sdl_library);
```

Dynamic Linking:
controlled by program at runtime

- To programmatically load a dynamic library:
  Linux/POSIX

```
// Special Case → reference all symbols of current executing process
void* this_process = dlopen(NULL, 0);
```

Dynamic Linking:
controlled by program at runtime

- To programmatically load a dynamic library:

  Windows

```
// Load the library
HMODULE my_library = LoadLibrary(TEXT("my_lib.dll"));
if (my_library == NULL) ERROR;

// Load reference to variable
FARPROC initializer = GetProcAddress(my_library,"my_int_var");
if (initializer == NULL) ERROR;
int *my_int_var_ptr = (int *)initializer;
*my_int_var_ptr = 42;  ← Access the variable!

// Load reference to function
FARPROC initializer = GetProcAddress(my_library,"my_func");
if (initializer == NULL) ERROR;
typedef void (*my_func_ptr_t)(void);
my_func_ptr_t my_func = (my_func_ptr_t)initializer;
my_func();   ← Call the function!

// Unload the library
FreeLibrary(sdl_library);
```

Compiling, Linking & Memory Management of C Programs

- Compilation of C Programs (revisited): Portability
- Shared Libraries
- Memory Management in C Programs
- Memory Management by Operating Systems

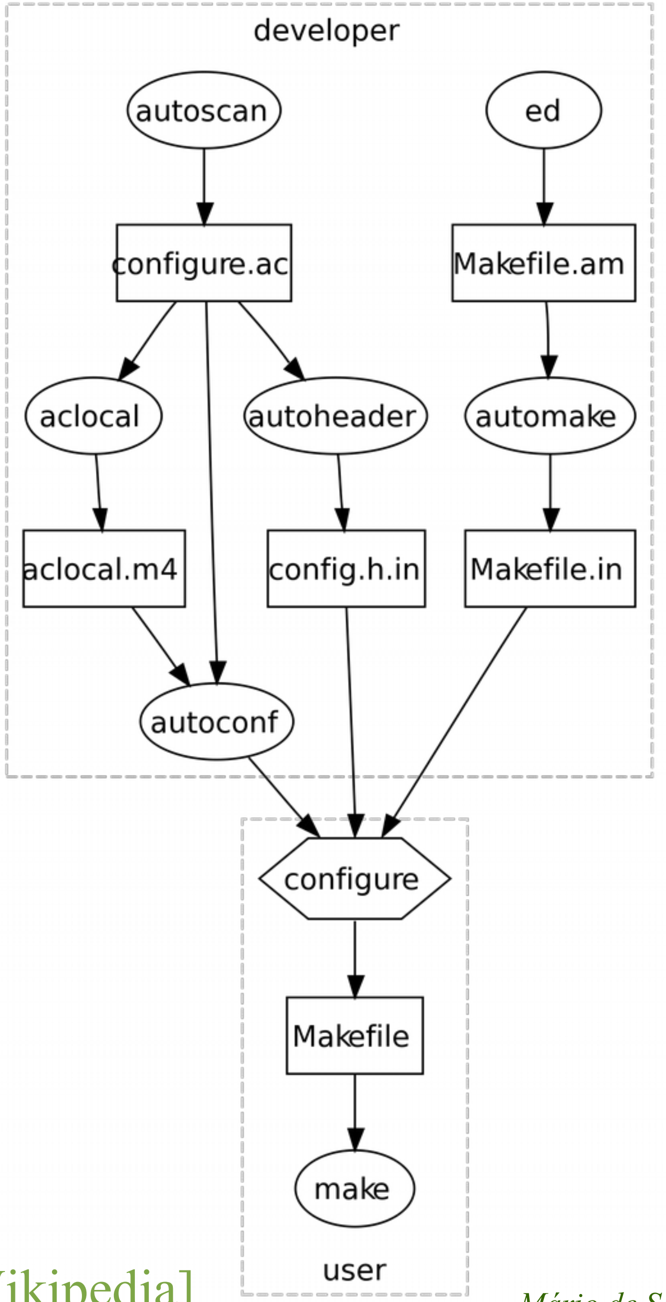
Portable Compilation

- Computers may have many different architectures...
  - endianness
  - word size
  - alignment
  - default signedness
  - No MMU (Memory Management Unit)
  - Library location (in the file system hierarchy)
- How to portably compile programs?

  → Use auxiliary tools!!
  - example: GNU Build System

Portable Compilation

- GNU Build System
  - make, automake,
  - autoconf, autoscan, autoheader, aclocal,
  - libtool
- A user, who wishes to compile the project, simply executes:
  - ./configure
  - make
  - make install



[Wikipedia]

Portable Compilation

- **`./configure`**
  - Determines the current computer's architecture, and configures the '**`Makefile`**' appropriately
    - determines whether called functions are present in the libraries
    - Determines library location
    - ...
- **`make`**
  - Compiles the program, using the configurations defined in the '**`Makefile`**'

Portable Compilation

- `./configure --host=avr32-linux`
  - Determines the architecture of the computer that will **execute** the program, and configures the '`Makefile`' accordingly.
    - existence of specific functions in the libraries
      (may require **compilation** and execution of programs... where? In the Build Platform!)
    - Library **location**
    - …

NOTE: which compiler to use?

NOTE: libraries must be in the same location in both the Build and Host Platforms!!!!

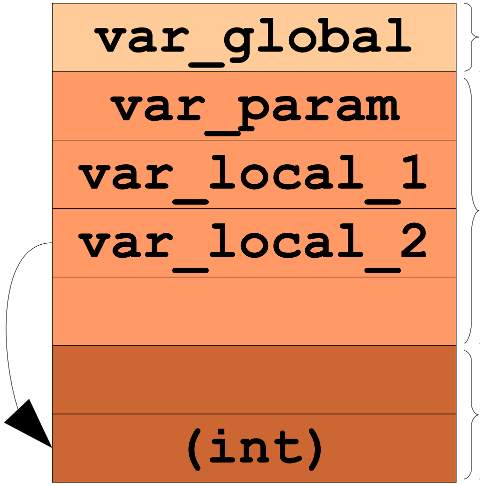
- `make`
  - Compiles the program, using the configurations defined in the '`Makefile`'

Compiling, Linking & Memory Management of C Programs

- Compilation of C Programs
- Shared Libraries
- Memory Management in C Programs
- Memory Management by Operating Systems

Memory Management in C Programs

- A program uses memory to store:
  - Global variables
  - Local Variables
  - Dynamically allocated memory (`malloc()`)



Global Variables

Local Variables
STACK

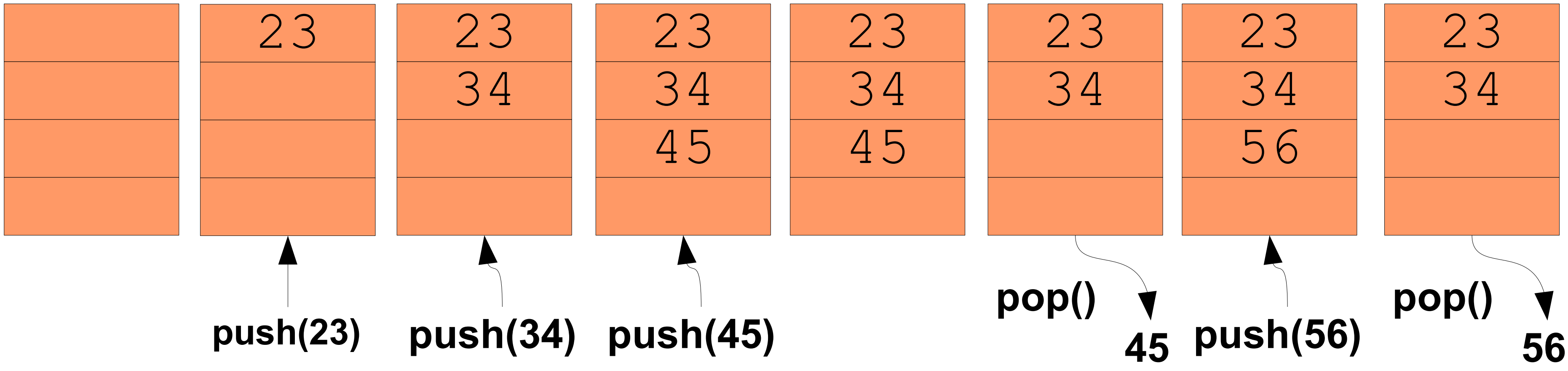
Dynamically Allocated Memory
HEAP

```
int var_global;

void foo(int var_param) {
  int var_local;
  int *var_local2 =
          malloc(sizeof(int));
}
```

STACK Management

- The number of local variables will change during program execution. It depends on the currently active functions!
- The memory area used for local variables is managed as a *stack*
  - *Stack:* LIFO (Last In First Out) data structure



Recursive Functions & the Stack

```
int sum(int n) {
  if (n == 0) return 0;
  return n + sum(n-1);
}
int main() {return sum(2);}
```

`main()`	`sum(2)`	`sum(1)`	`sum(0)`	`return 0`	`return 1`	`return 3`	`return 3`
	n=2	n=2	n=2	n=2	n=2	3	
		n=1	n=1	n=1	1		
			n=0	0			

C Functions & the stack

- In C, the stack is use for:
  - Passing of function arguments/parameters
  - Store the return address (a.k.a. program counter)
  - Store local variables
  - Store the return value
  - Temporarily store the CPU registers.
- Each compiler will establish a convention regarding how the above information is organized/stored in the stack when a function is called.
  - This structure is known as a *Stack Frame* or *Activation Record*
  - When calling OS functions, a specific organization must be used –> known as ABI – Application Binary Interface

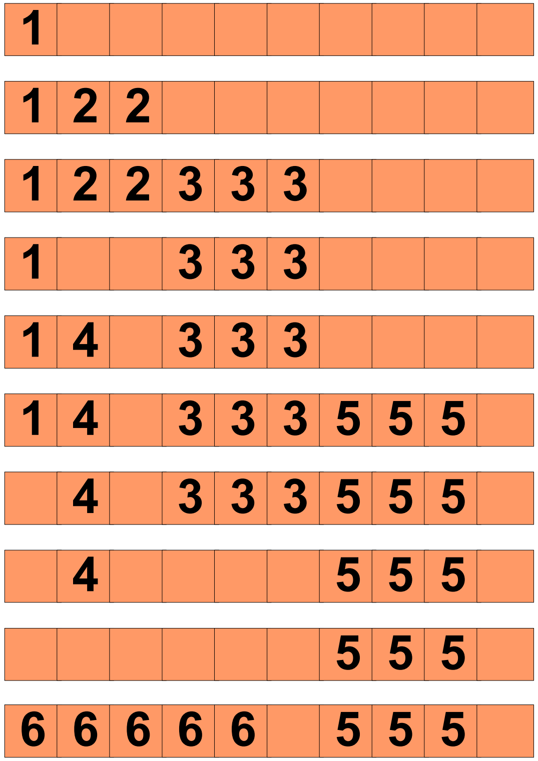
HEAP Management

- Sequence by which memory os allocated and freed is ‘random’ (i.e. unknown by the heap manager). Memory is allocated in blocks...
- Heap memory management will depend on the specific implementation of:
  `malloc(), free(), realloc(), calloc(), memalign()`

NOTE: You can overload the default implementation of these functions with your own version!

HEAP Management: Example

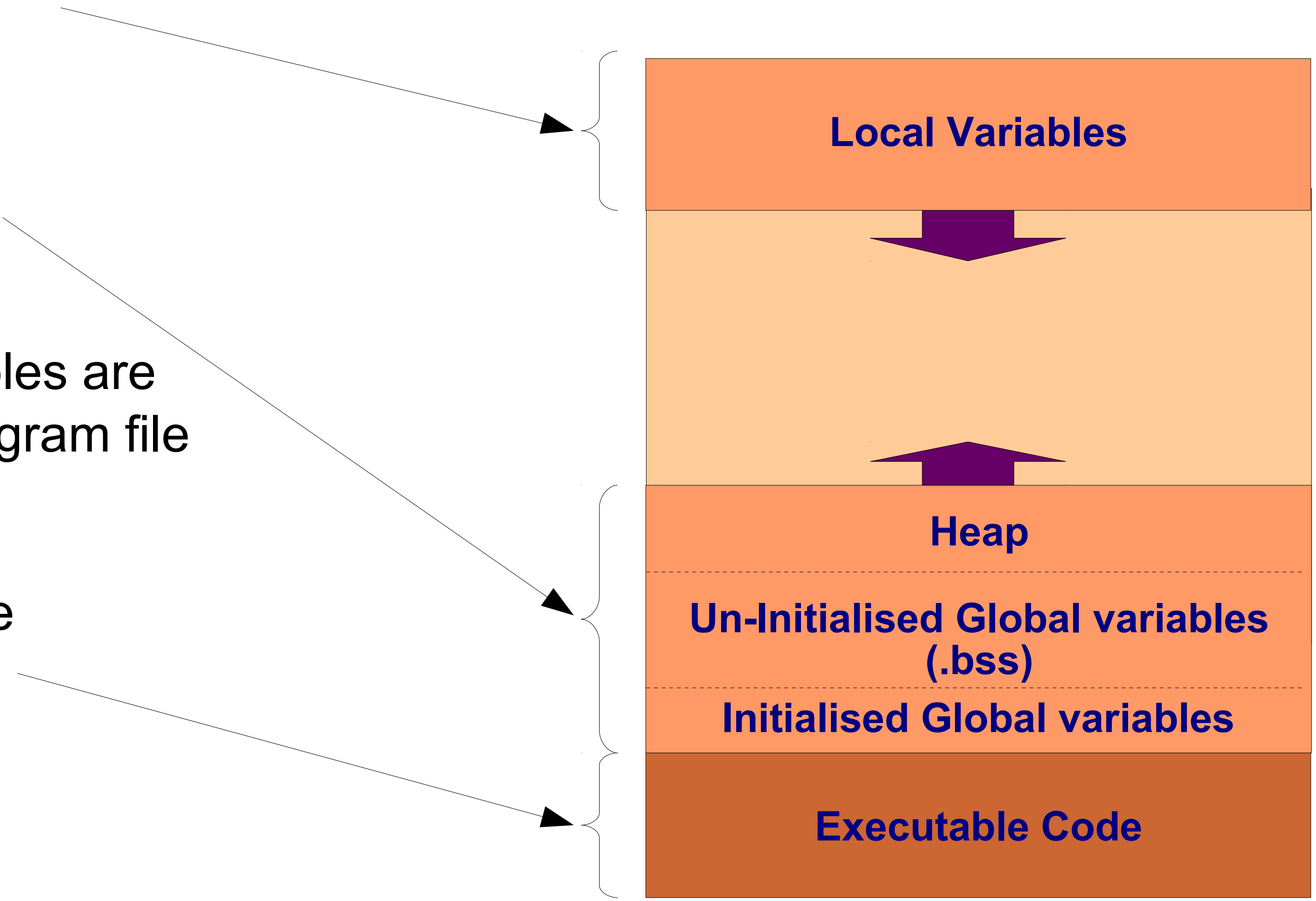
```
int main() {
 char *v1 = malloc(1);
 char *v2 = malloc(2);
 char *v3 = malloc(3);
 free(v2);
 char *v4 = malloc(1);
 char *v5 = malloc(3);
 free(v1);
 free(v3);
 free(v4);
 char *v6 = malloc(5);
}
```



1									
1	2	2							
1	2	2	3	3	3				
1			3	3	3				
1	4		3	3	3				
1	4		3	3	3	5	5	5	
	4		3	3	3	5	5	5	
	4					5	5	5	
						5	5	5	
6	6	6	6	6		5	5	5	

Assuming malloc() uses first-fit algorithm!

Unix Process: Memory Use

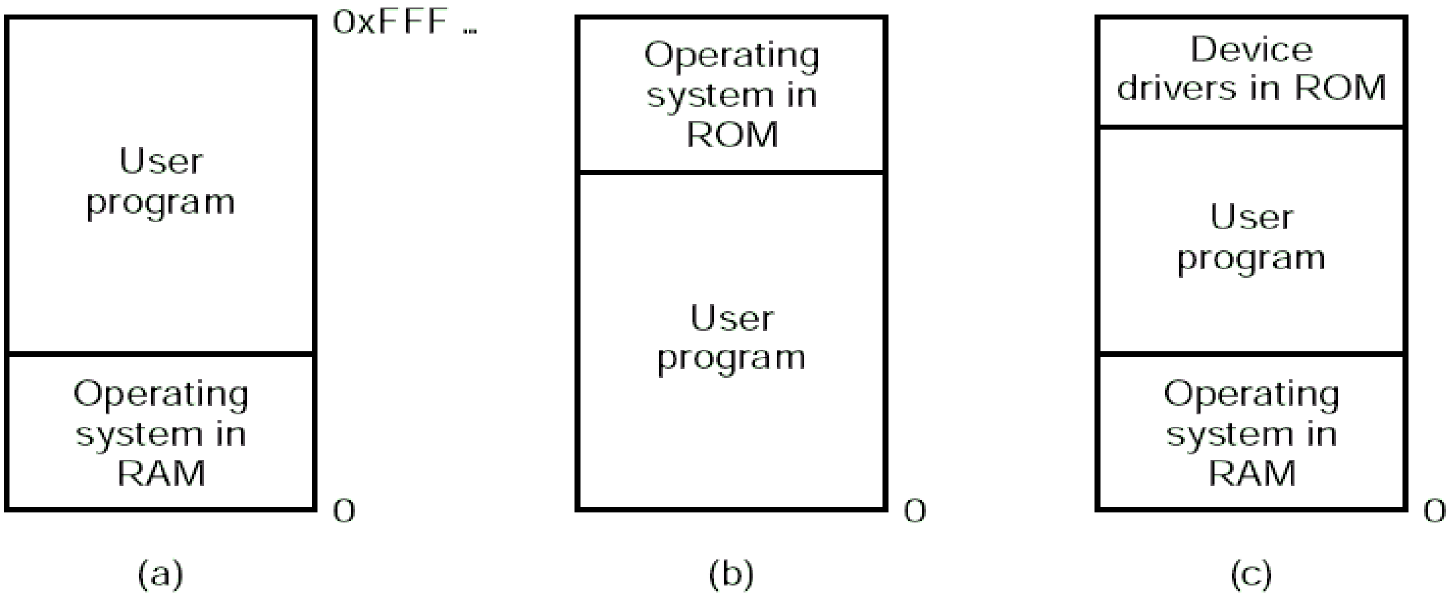


- Stack Segment
  - Variable size
- Data Segment
  - Read/Write
  - Variable size
  - Initialised variables are loaded from program file
- Text Segment
  - Executable code
  - Read-only
  - Fixed size

Compiling, Linking & Memory Management of C Programs

- Compilation of C Programs
- Shared Libraries
- Memory Management in C Programs
- Memory Management by Operating Systems

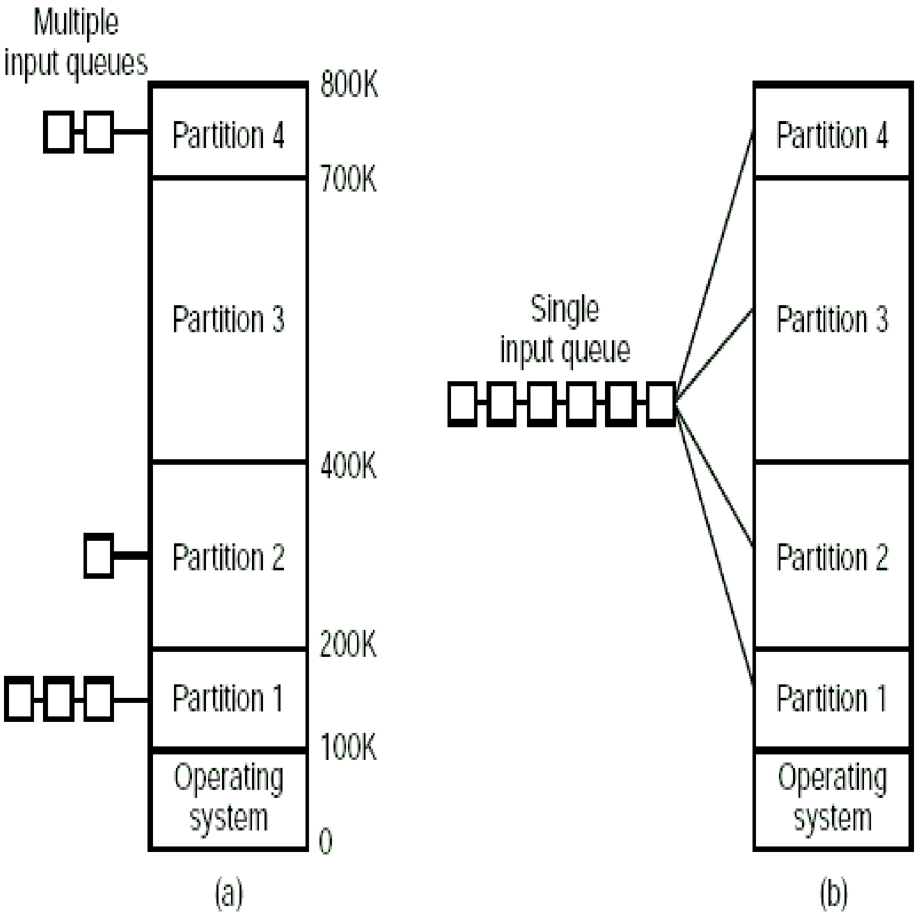
Memory Management in Single Process Operating Systems.



- Typical of micro-controllers with no OS
- Also used by MS-DOS.
- Device-Drivers in ROM -> a.k.a. BIOS.

Picture from “Modern Operating Systems”, Andrew Tanenbaum

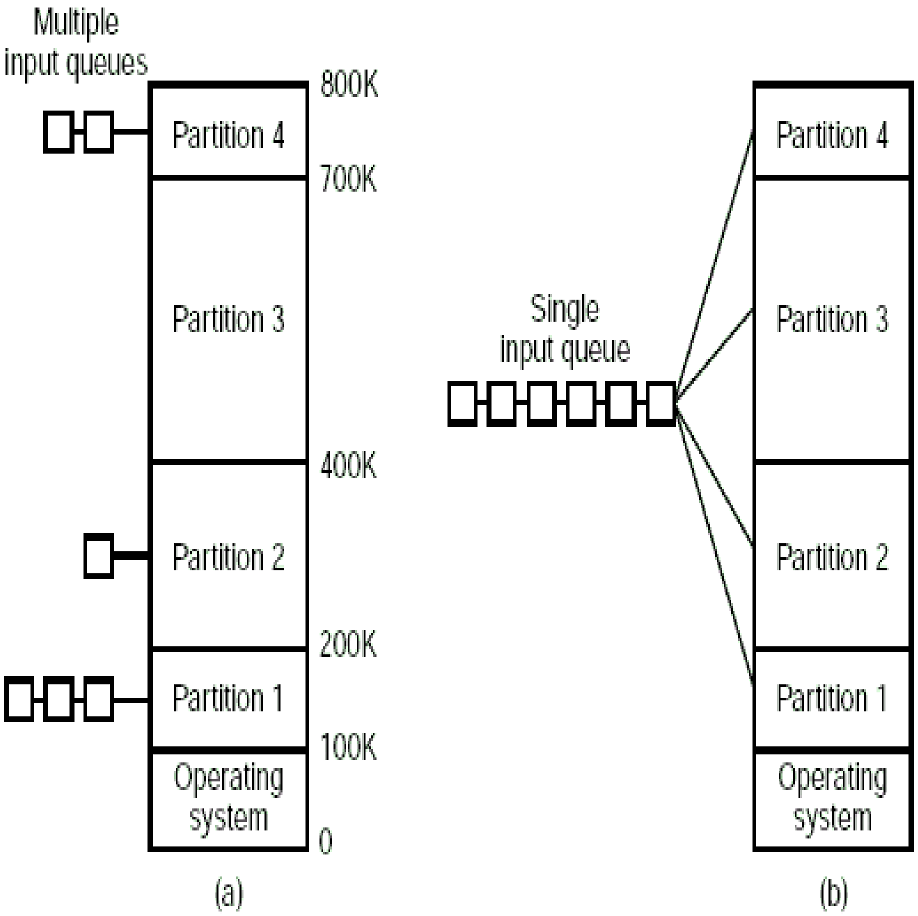
Multi Process Operating Systems:
-> Partitioned Memory



- Memory divided into partitions (possibly differing sizes)
- OS will attribute a partition to each process. (choose smallest free partition big enough for process)
- When run out of partitions -> maintain a pending process queue

Picture from “Modern Operating Systems”, Andrew Tanenbaum

Multi Process Operating Systems:
-> Partitioned de Memory



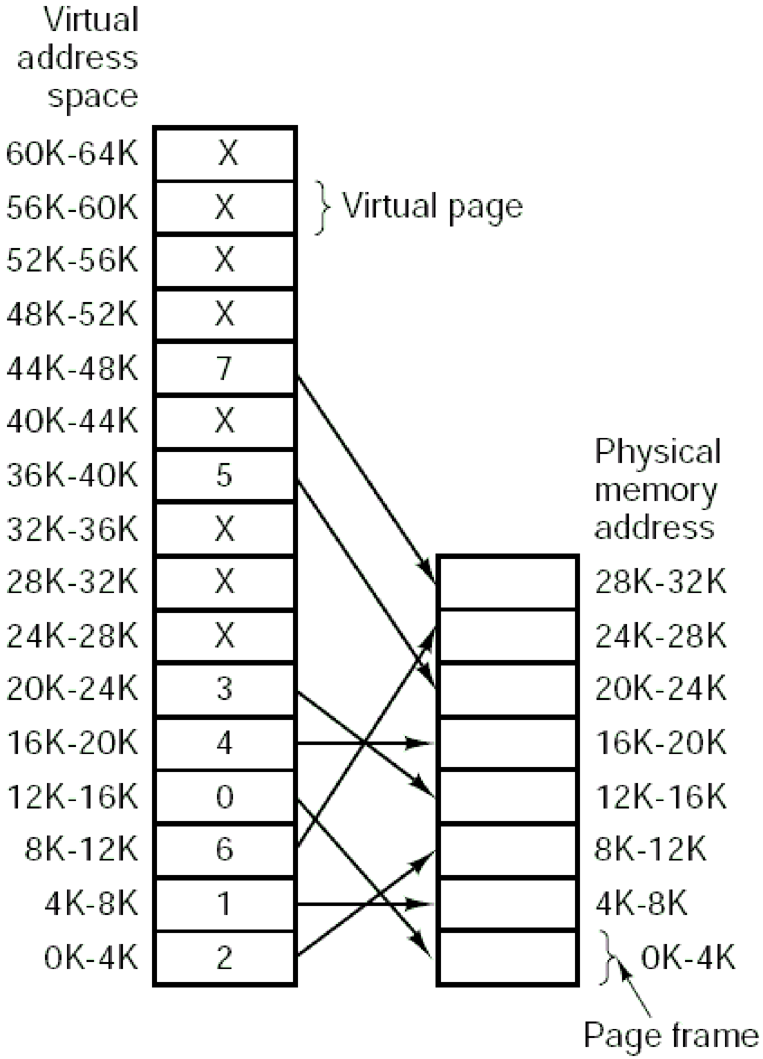
- Drawbacks:
  - Program size limited to size of largest partition.
  - Program size always smaller than computer's RAM memory.
  - Ineficient use of memory (programs must completely reside in RAM memory)

Picture from "Modern Operating Systems", Andrew Tanenbaum

Multi Process Operating Systems:
-> Virtual Memory

- Main idea: the same process gets several partitions...
  (use smaller partitions)

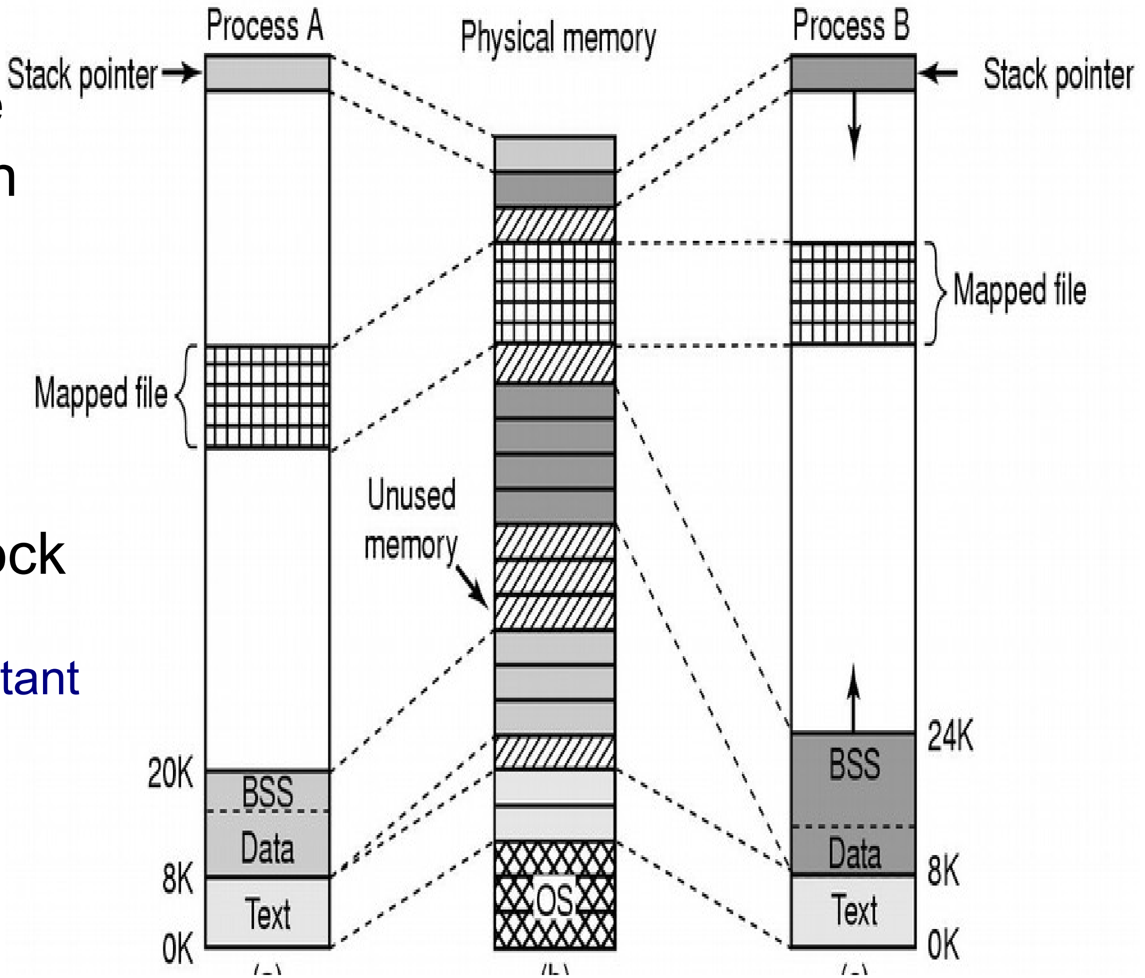
- Each process gets its own virtual address space



Picture from “Modern Operating Systems”, Andrew Tanenbaum

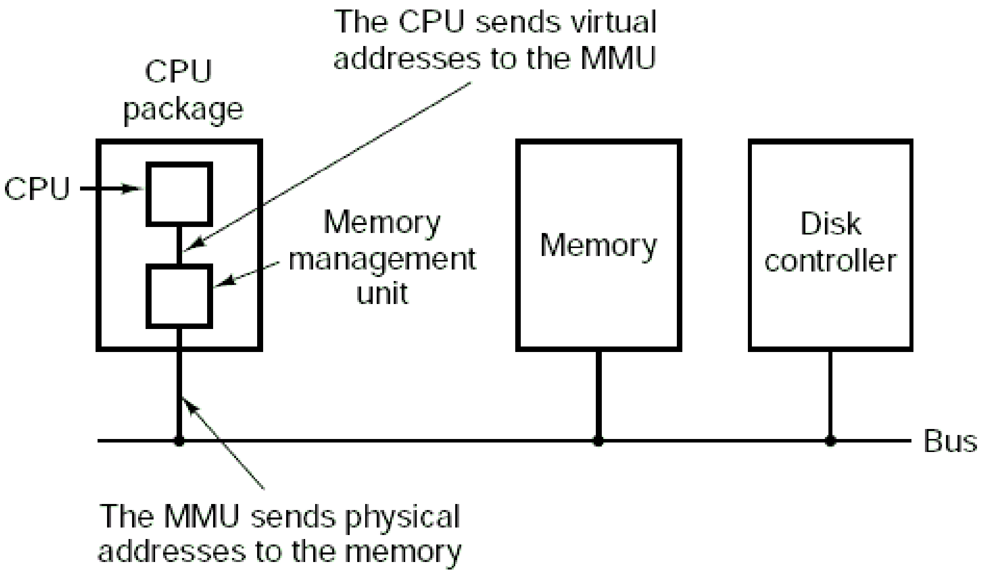
Multi Process Operating Systems: -> Virtual Memory

- It is now possible to:
  - have two program execute distinct executable code, in the same virutal address!!
  - Have two instances of the same program share the same physical memory block (read-only: memory block will contain executable code or constant global variables).



Picture from "Modern Operating Systems", Andrew Tanenbaum

Multi Process Operating Systems:
-> Virtual Memory



- Conversion from virtual to physical address is done in hardware

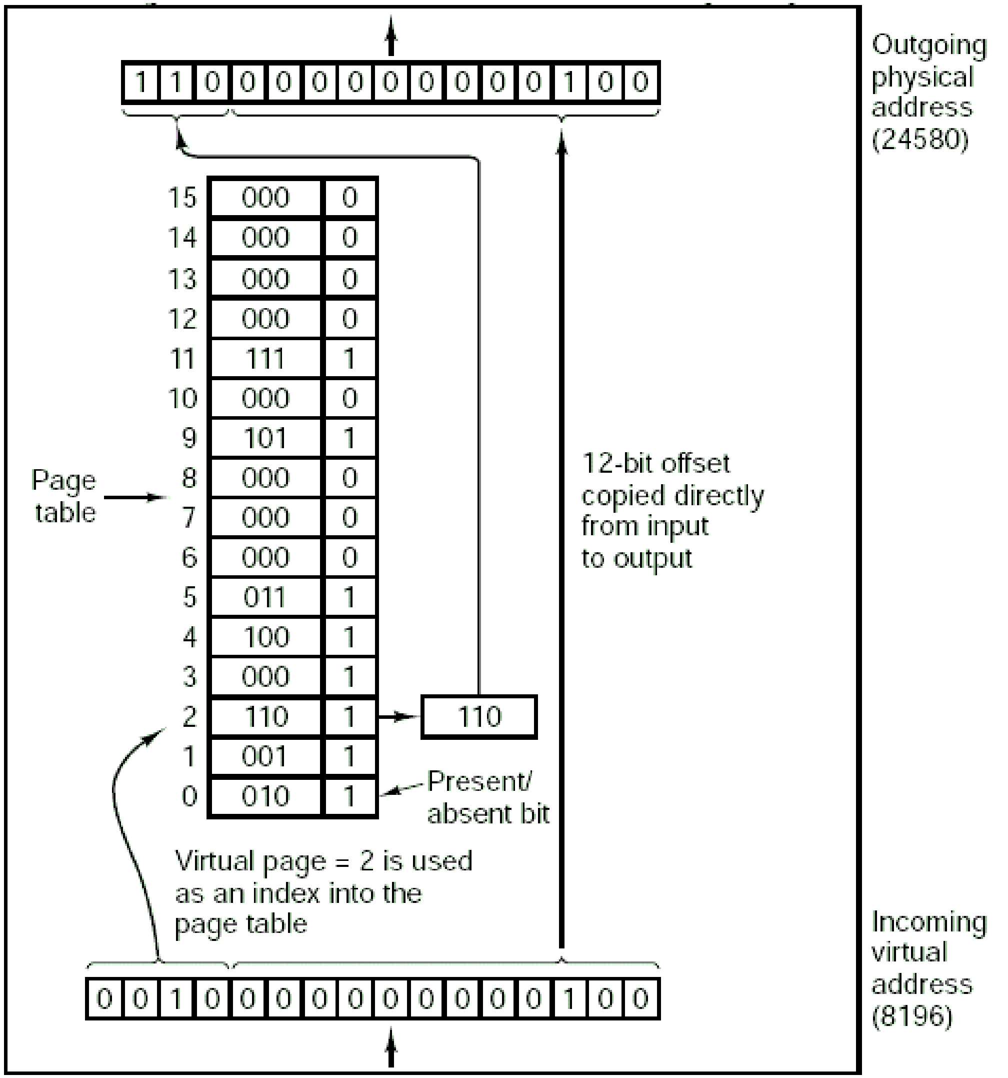
MMU: Memory Management Unit

(Doing it in hardware allows program to run at full speed!)

Picture from “Modern Operating Systems”, Andrew Tanenbaum

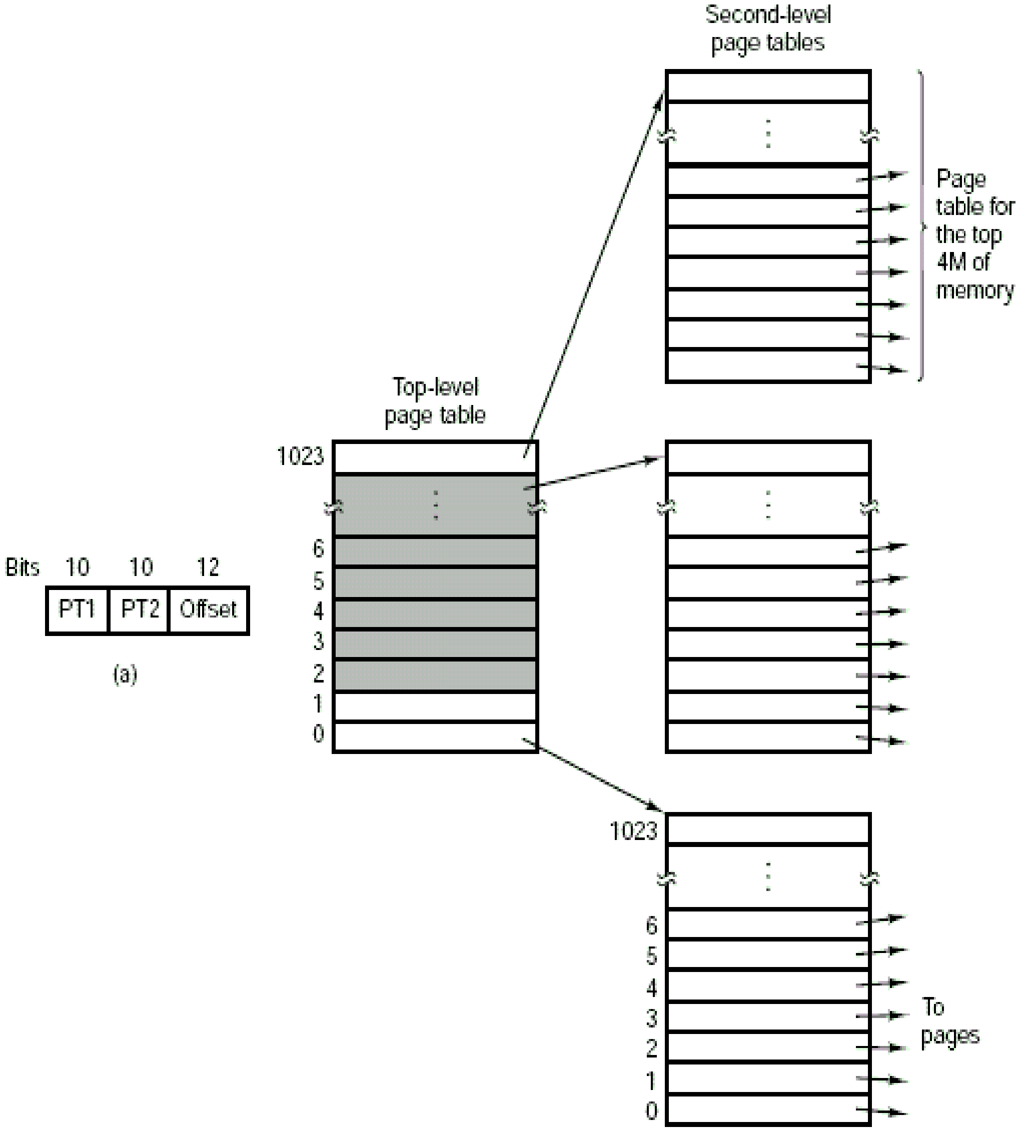
Multi Process Operating Systems:
-> Virtual Memory

- Mapping is done using only the most significant bits of virtual address
- Uses a 'Page Table' for conversion
- Each process has its own unique ‘Page Table’!!



Picture from “Modern Operating Systems”, Andrew Tanenbaum

Multi Process Operating Systems: -> Virtual Memory



- Example:
  - CPU with 32 bits addresses,
  - 4 Kbyte page size (12 bits),
    => page table will need 4 Mbyte!!!

- Usually, each program will only use a small fraction of the virtual address space.
  - Use a multi-level page table architecture

Picture from “Modern Operating Systems”, Andrew Tanenbaum

Multi Process Operating Systems:
-> Virtual Memory

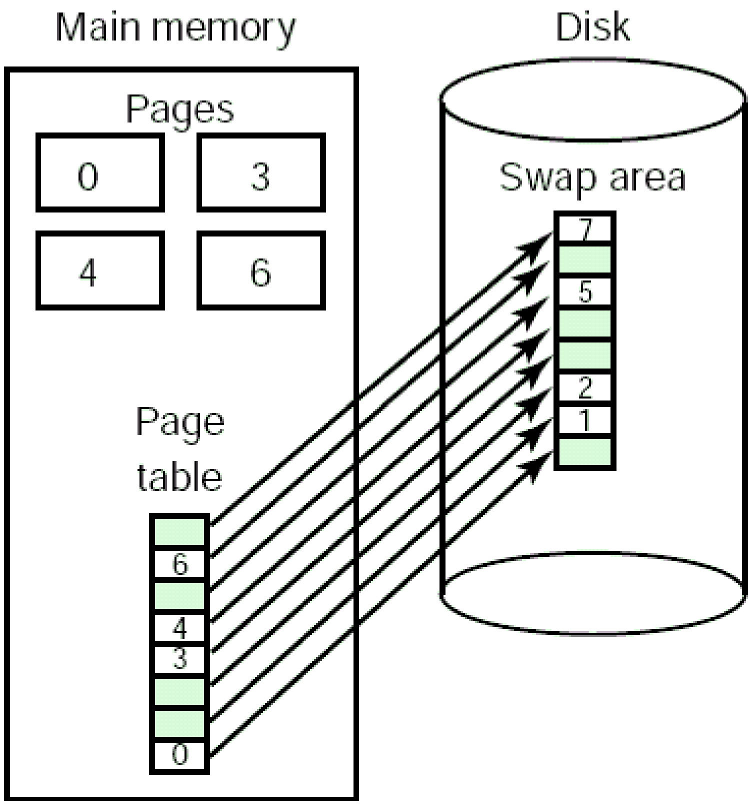
- Drawbacks:
  - Each memory access requires aditional memory reads (from the page tables) to convert to the real address space => SLOW!!!.
- Solution:
  - Use a special cache memory for the page table. Cache will maintain copies of the most recently used page tables.
  - Known as: Translation Loookaside Buffer (TLB).

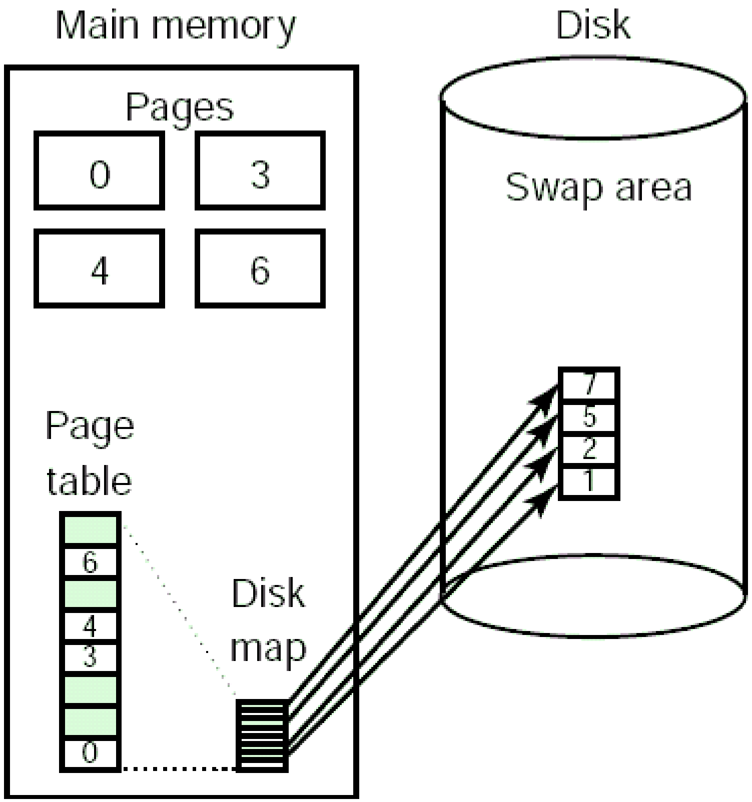
  - When a process switched occurs, all entries in the TLB become invalid! (remember, each process has its own unique page table)

Multi Process Operating Systems:
-> Virtual Memory

- Advantage:
  - Better use of available memory (smaller memory blocks)
- **BUT**:
  - The memory used together by all programs cannot exceed available RAM memory!
- Solution: Paged Memory
  - Less often used memory blocks are stored on hard disc!
  - May be stored
    - in a file (less efficient, more flexible)
    - in a specific hard disk partition (more efficient, less flexible)

Paged Memory





Picture from “Modern Operating Systems”, Andrew Tanenbaum

Bibliography

- “Modern Operating Systems”, Andrew Tanenbaum
  - Chapter 4: Memory Management
  - Chapter 10.4: Memory Management in Unix
- “Compiler Construction: Principles and Practice”, Keneth C. Louden
  - Chapter 7: Runtime Environment
    (7.1: Memory Organization During Program Execution, 7.2: ...)
- “Linkers and Loaders”, John R. Levine
- GNU manuals: gcc, autoconf, libtool
- http://www.freesoftwaremagazine.com/books/autotools_a_guide_to_autoconf_automake_libtool