Runtime Environments

Abhijat Vichare

ACM India Summer School
on
Compilers for AI/ML Programs

June 2024

Overall Plan

- Stage 1: The Operational View
  - Working “backwards”: process ➡ program
  - Working “forwards”: program➡ process
  - The Toolchain
- Stage 2: The Conceptual View
  - Creating
    - an executable, (and executing) a process, and the runtime
    - Key Idea: Binding, and the main Take Home

      No object must be unbound at the interpretation instant

Overall Plan

- Stage 1: The Operational View
  - Working “backwards”: process ➡ program
  - Working “forwards”: program ➡ pro
  - The Toolchain

What we need

- Stage 2: The Conceptual View
  - Creating
    - an executable, (and executing) a process, and the runtime
    - Key Idea: Binding, and the main Take Home

No object must be unbound at the interpretation instant

Overall Plan

- Stage 1: The Operational View
  - Working “backwards”: process ➡ program
  - Working “forwards”: program ➡ process
  - The Toolchain

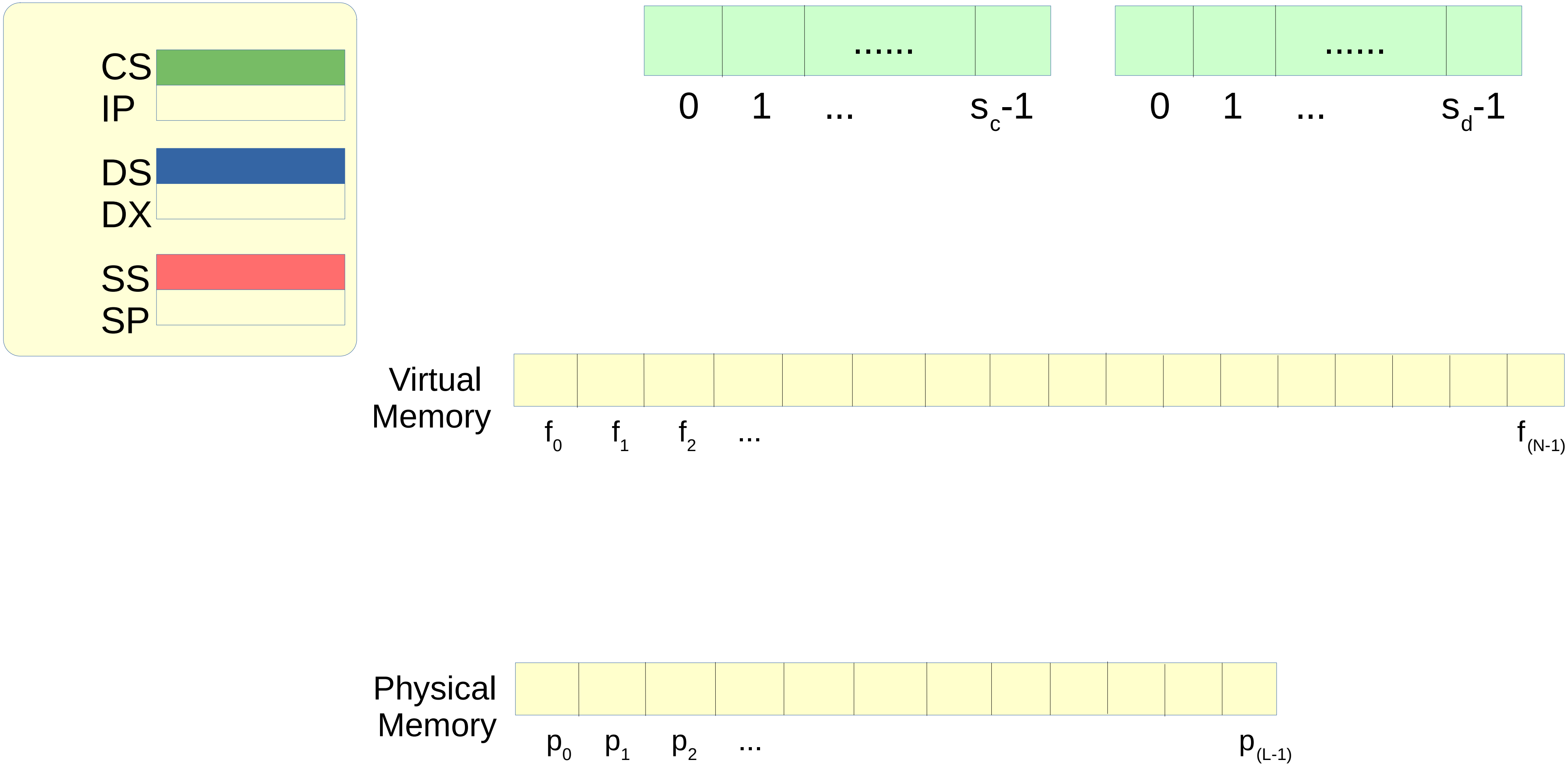
What we do

- Stage 2: The Conceptual View
  - Creating
    - an executable, (and executing) a process, and the runtime
    - Key Idea: Binding, and the main Take Home

      No object must be unbound at the interpretation instant

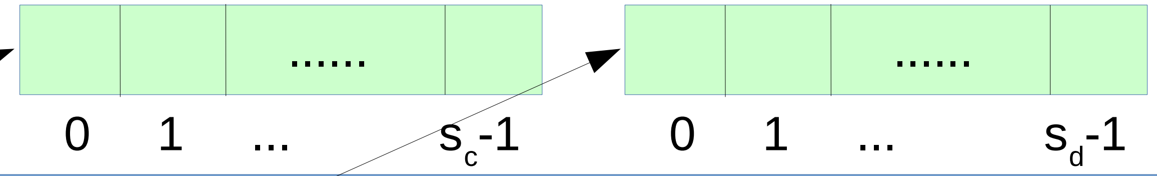
Operational View

“Program in memory”: Expectations of the Hardware



Process ► Program

“Program in memory”: Binary Code and Data

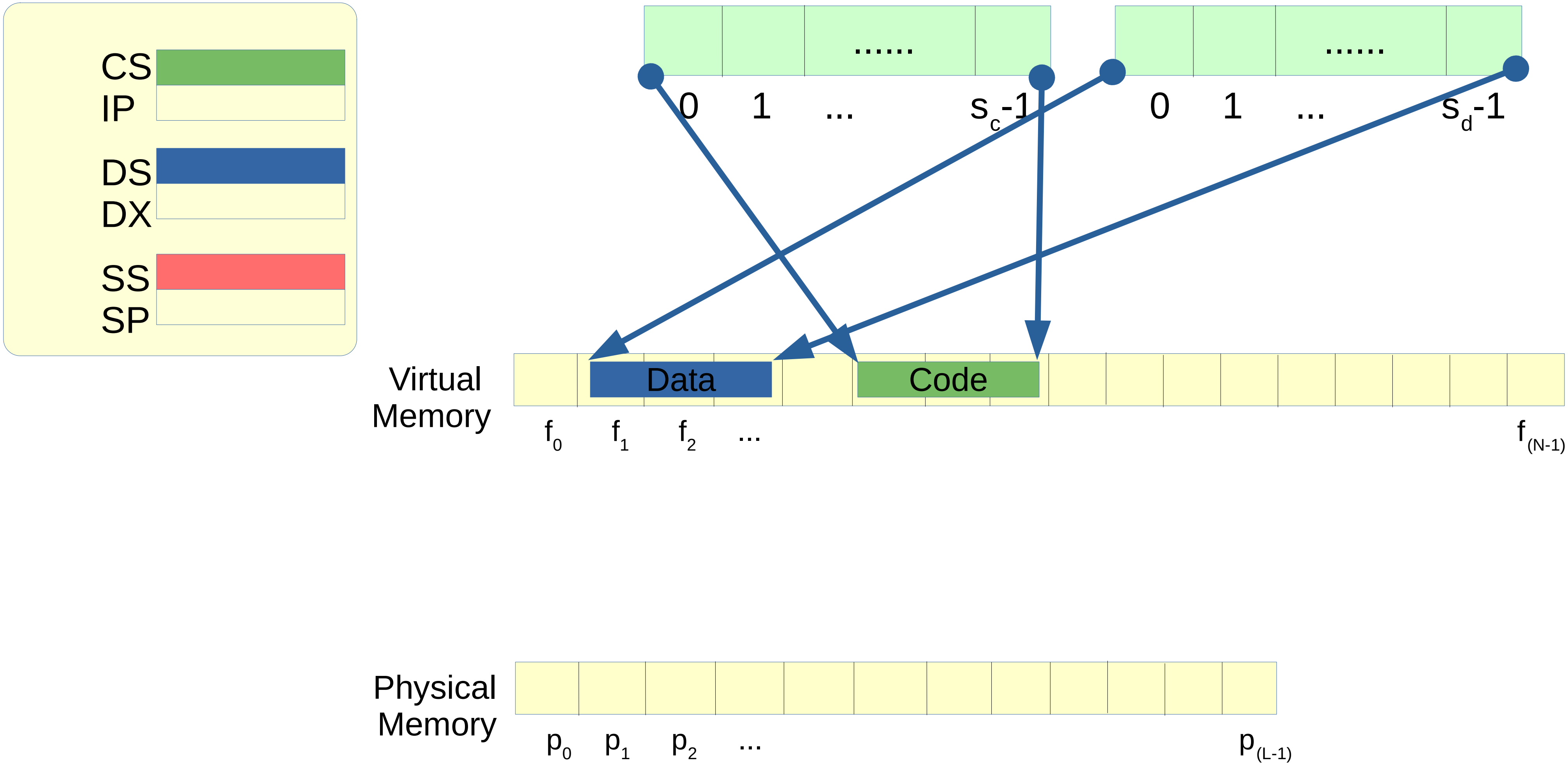


Code: Instructions must be in binary, i.e. opcodes. Length: s_c

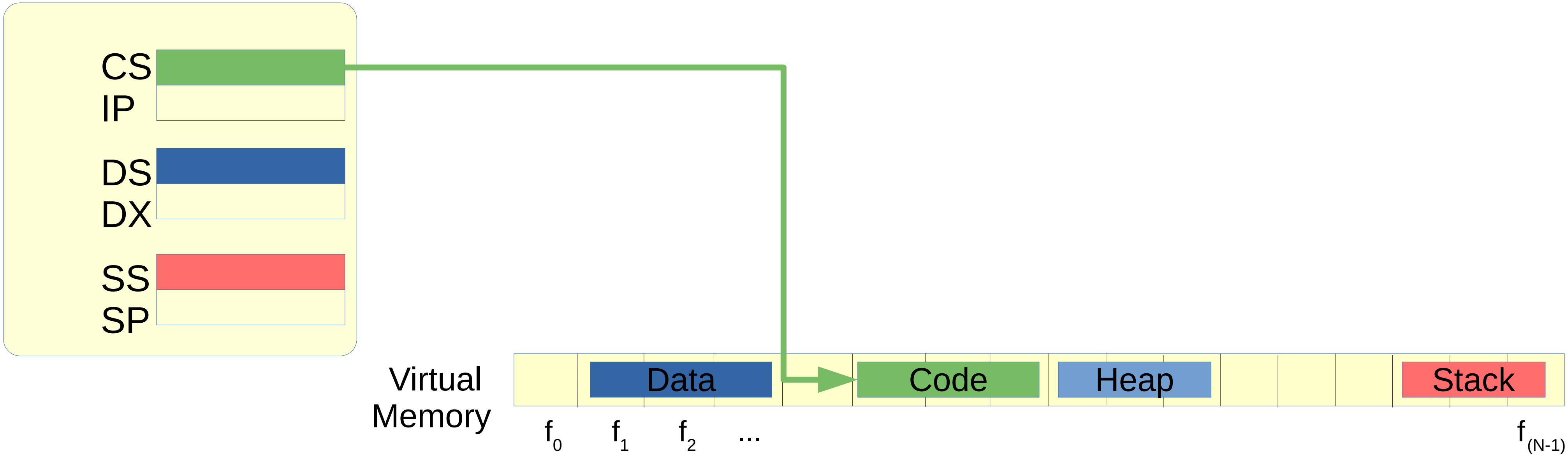
Data: Each data type must be finally represented in binary. Length: s_d

Process ► Program

“Program in memory”: Rules of Primary Memory Use

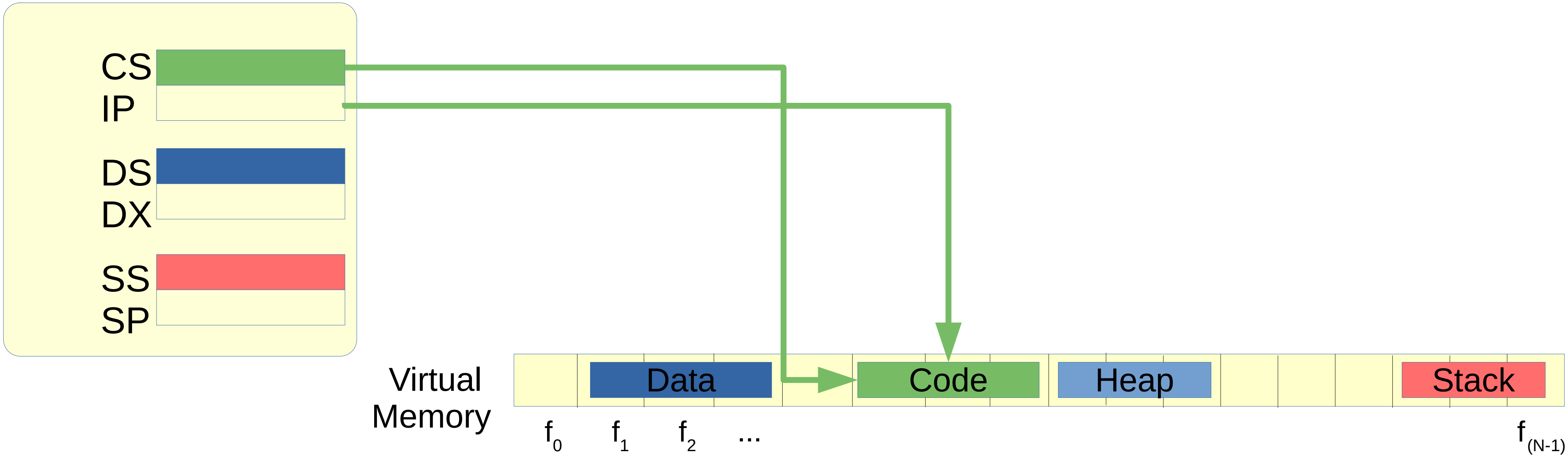


“Program in memory”: Rules of Primary Memory Use



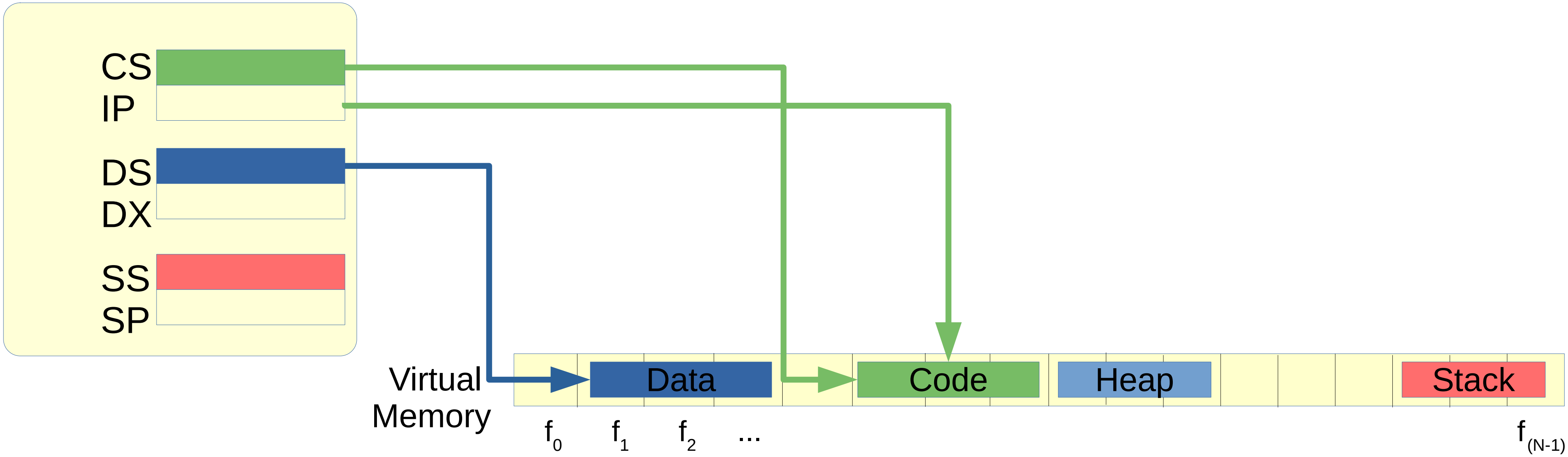
Process ► Program

“Program in memory”: Rules of Primary Memory Use



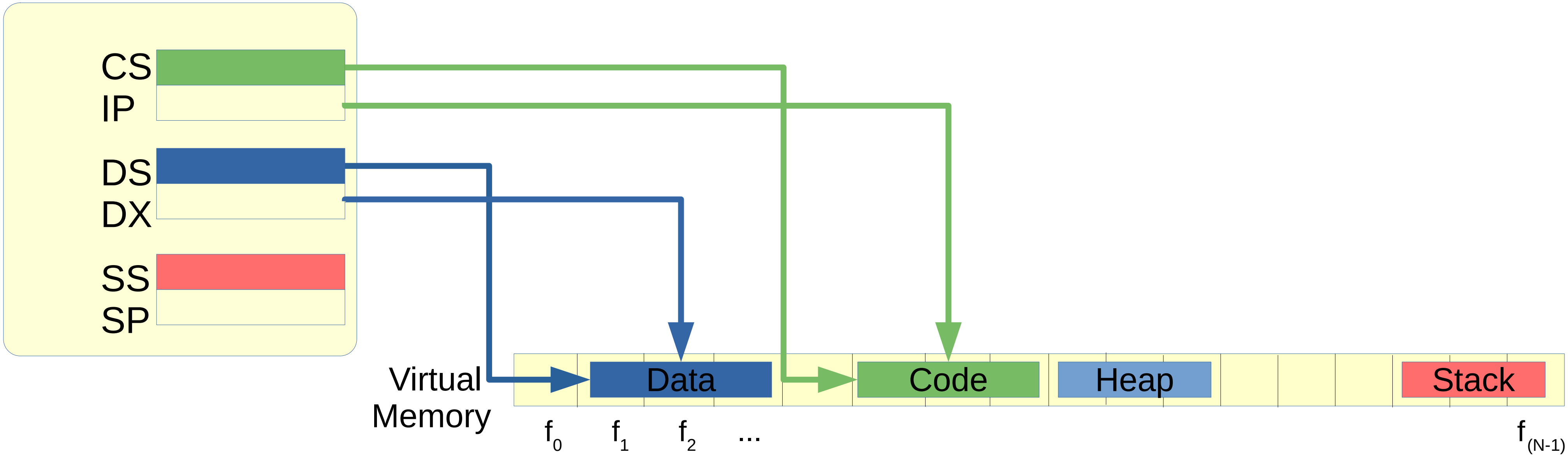
Process ► Program

“Program in memory”: Rules of Primary Memory Use



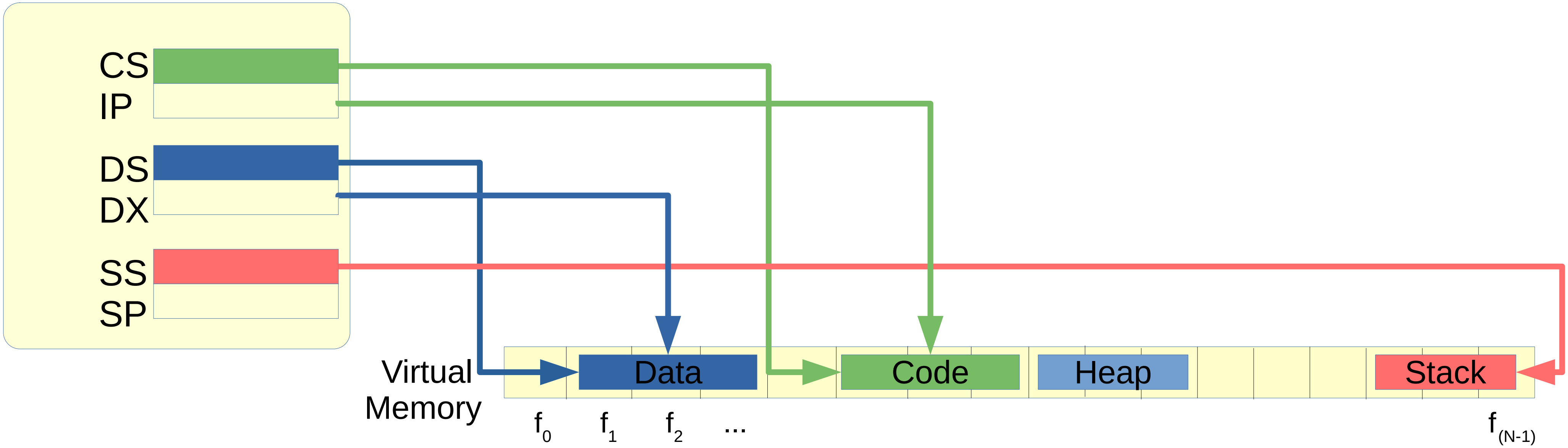
Process ► Program

“Program in memory”: Rules of Primary Memory Use



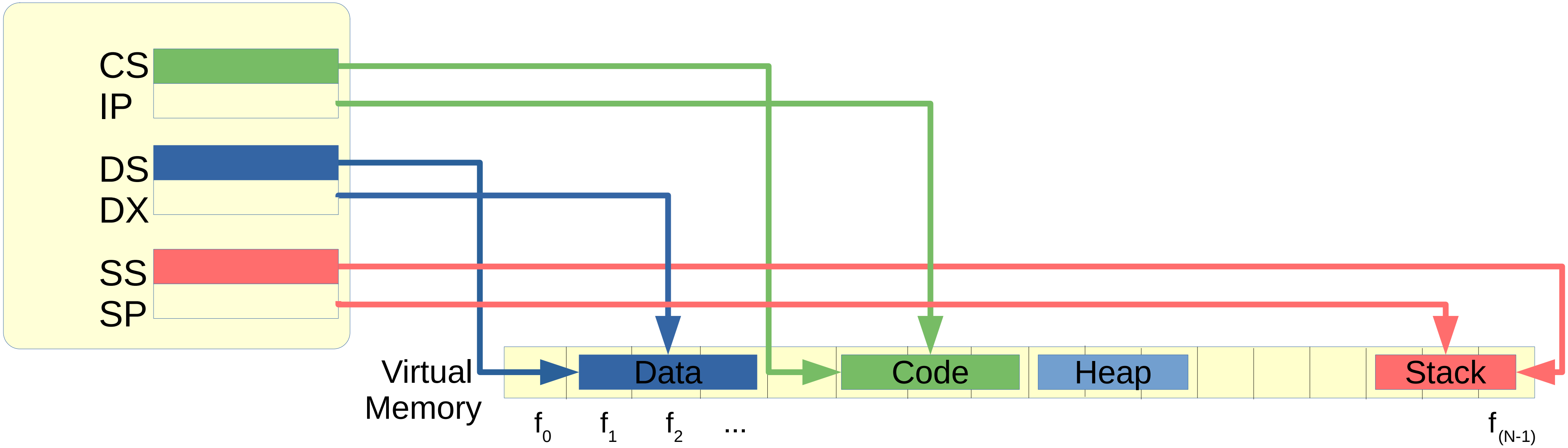
Process ► Program

“Program in memory”: Rules of Primary Memory Use

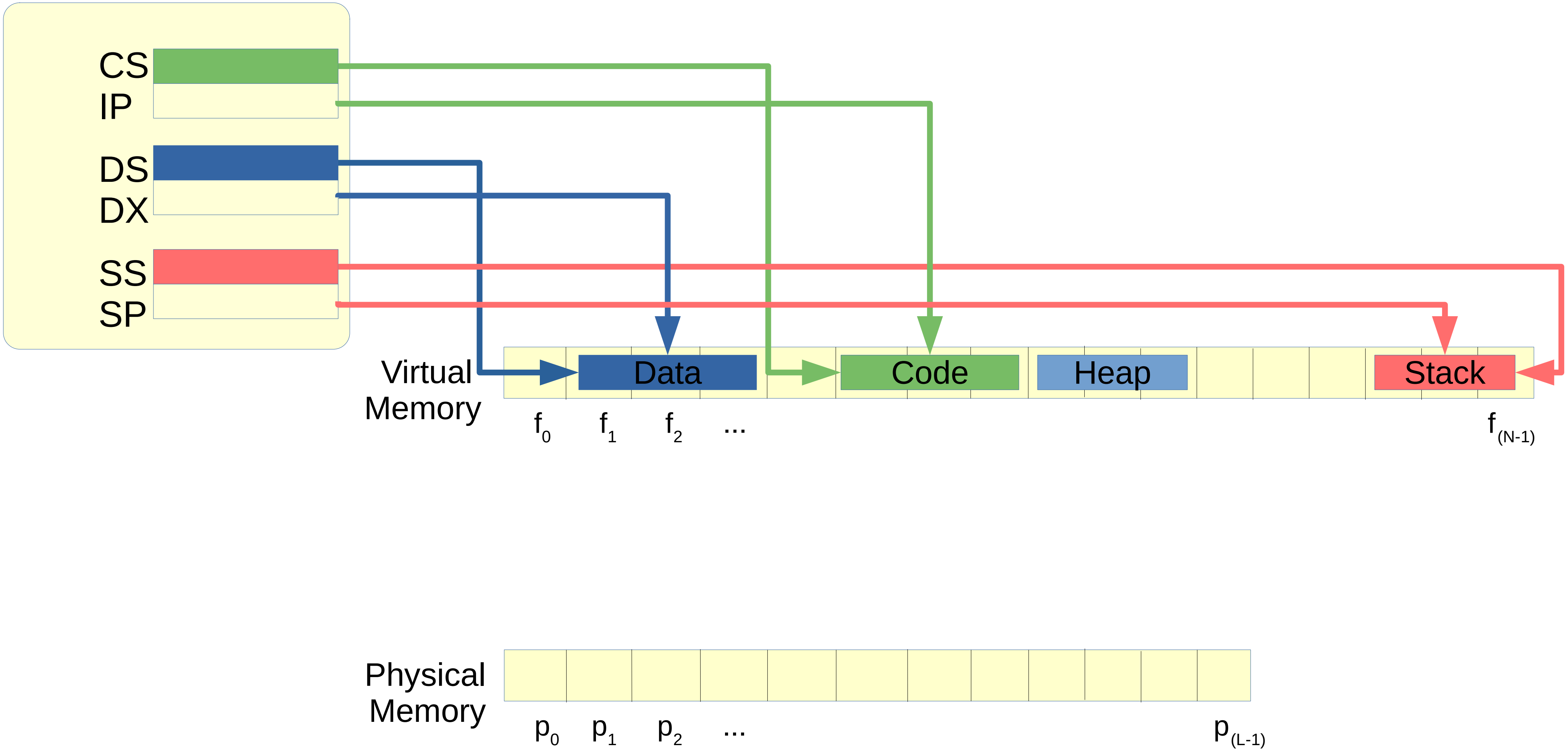


Process ► Program

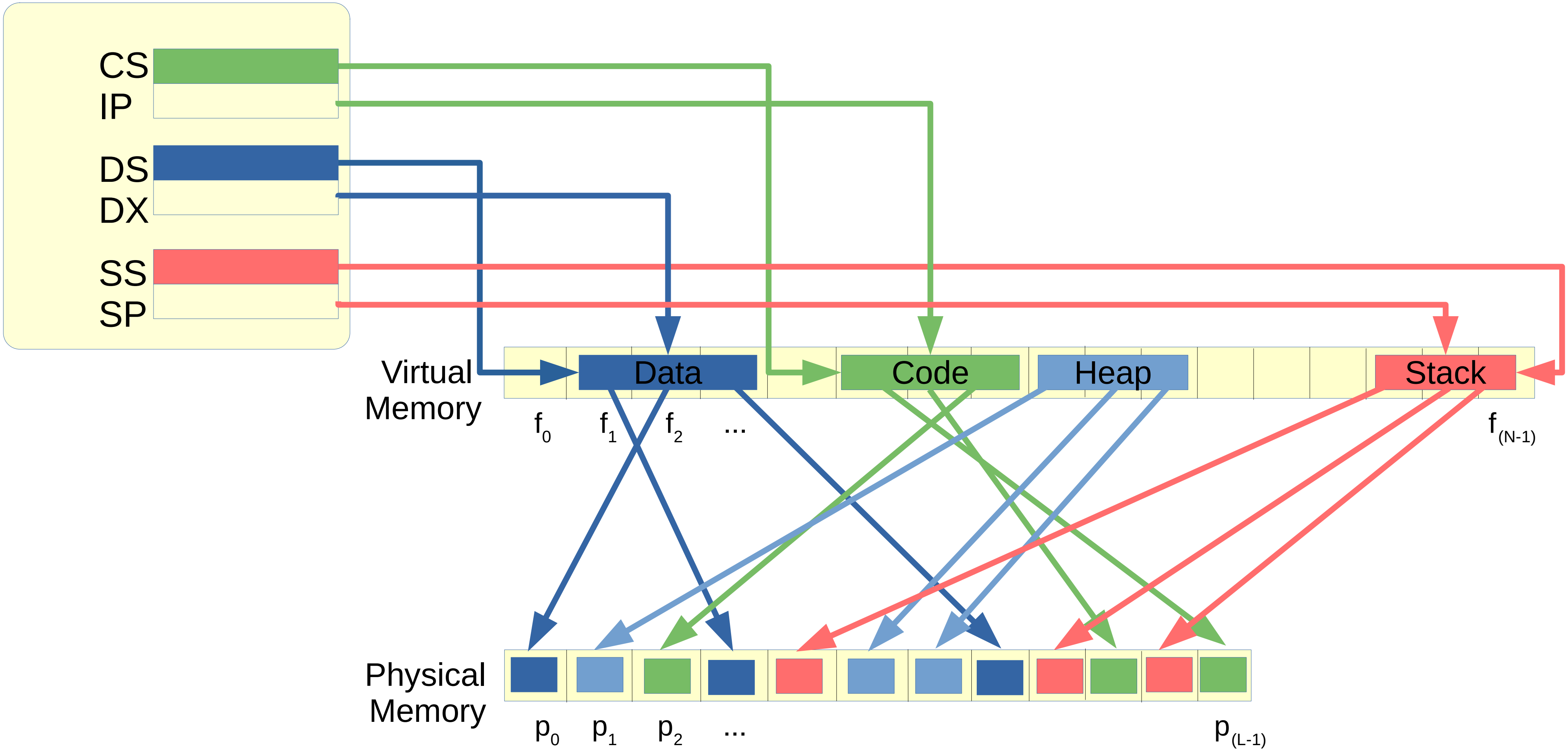
“Program in memory”: Rules of Primary Memory Use



“Program in memory”: Rules of Primary Memory Use

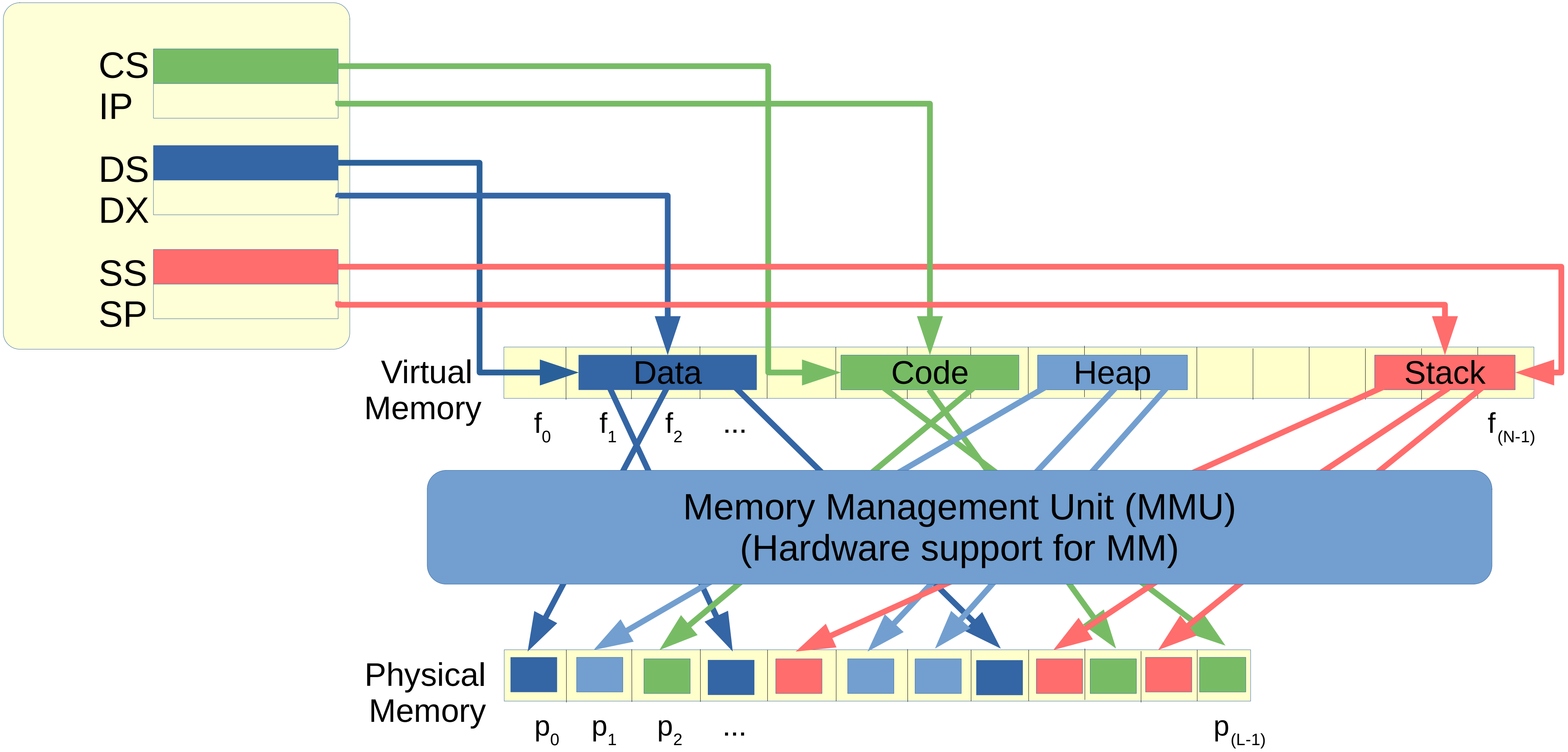


“Program in memory”: Rules of Primary Memory Use

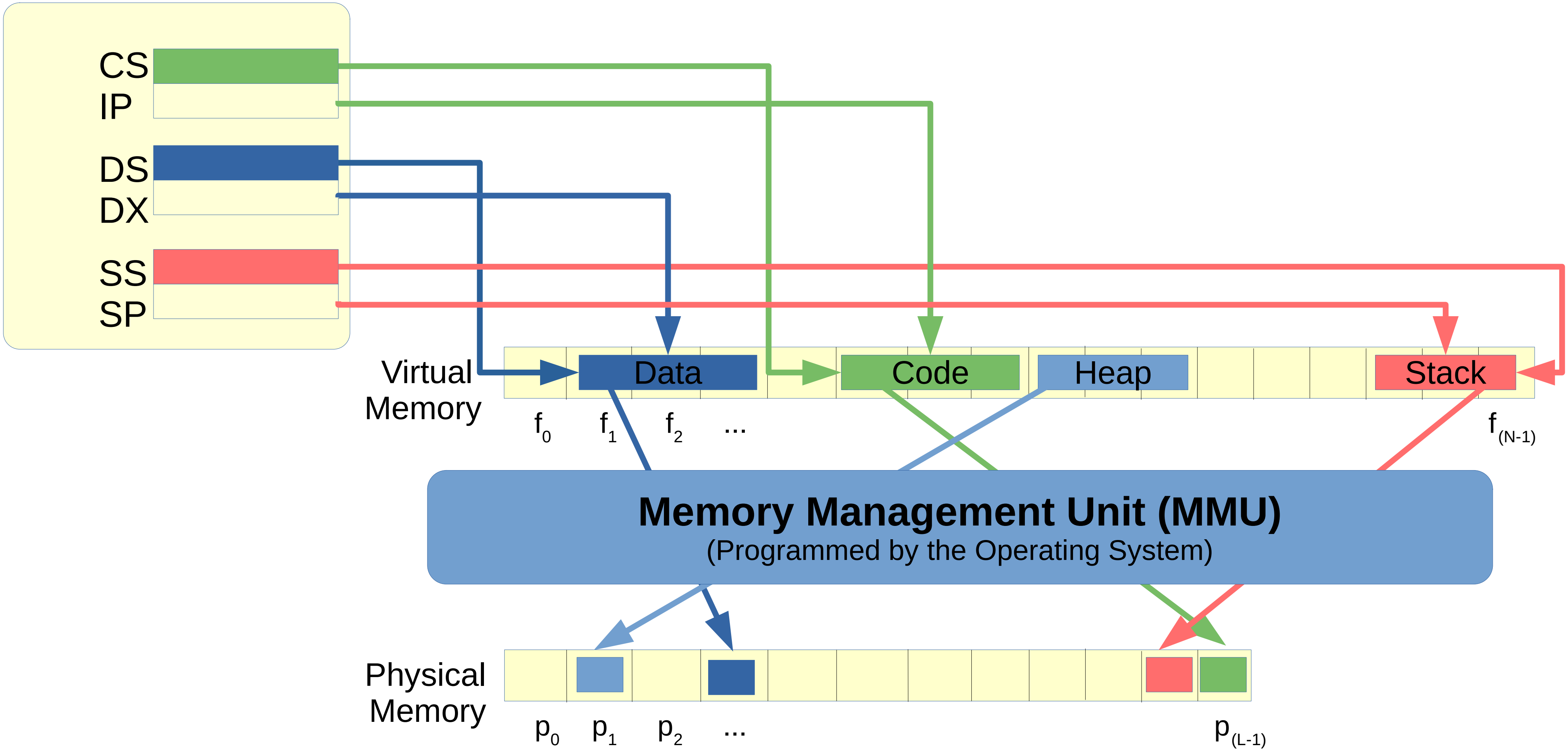


Process ► Program

“Program in memory”: Rules of Primary Memory Use

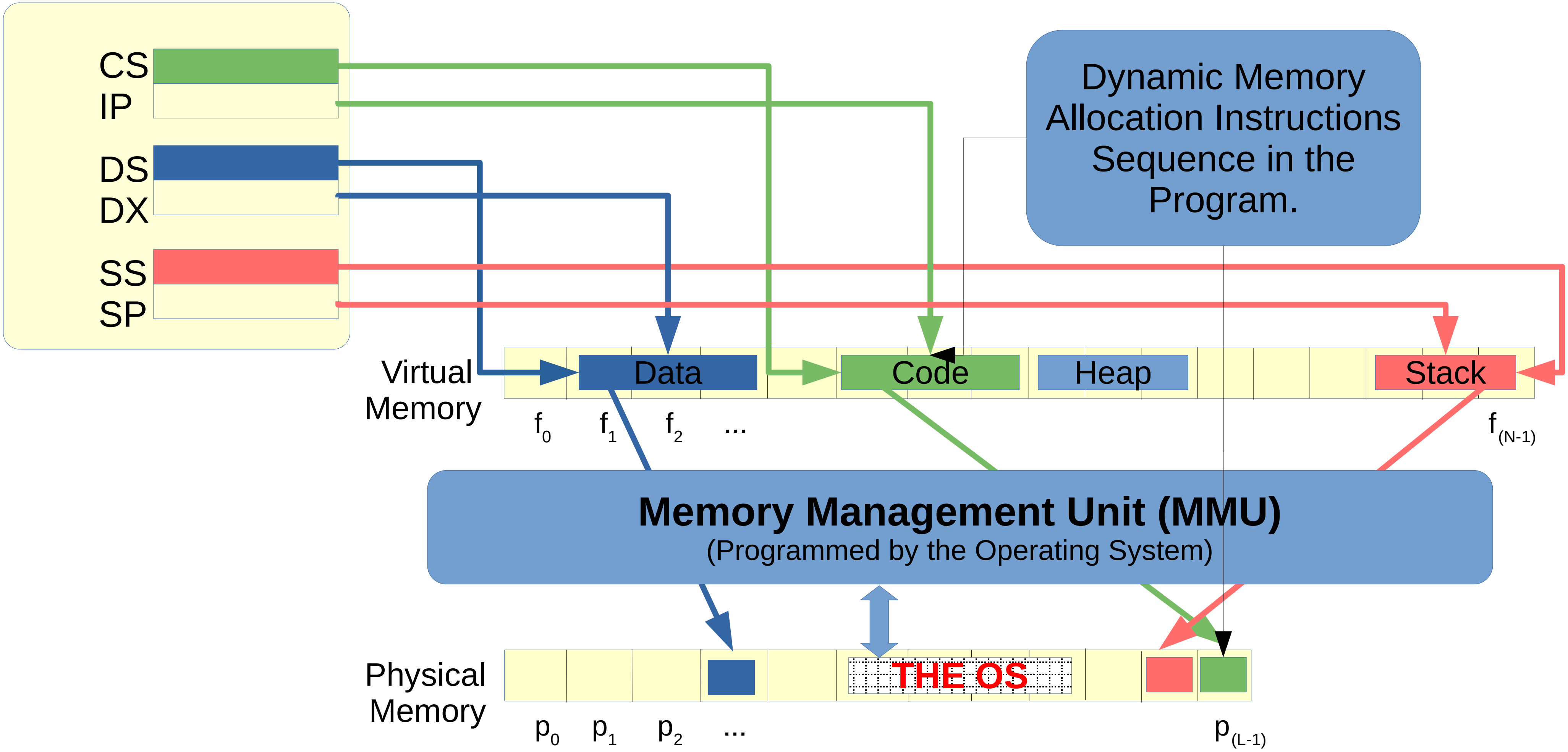


“Program in memory”: Rules of Primary Memory Use

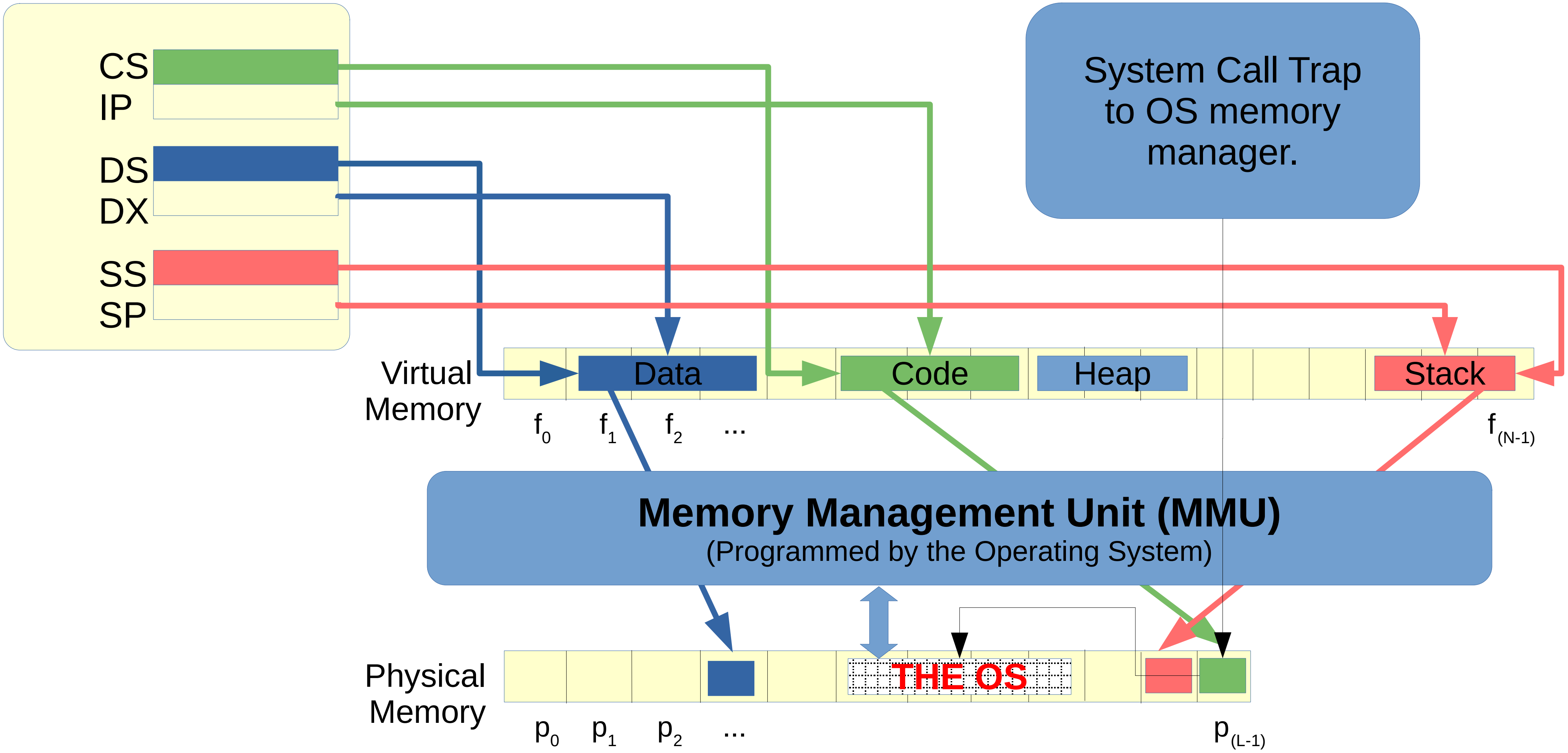


Process ► Program

“Program in memory”: Execution using CPU + Mem

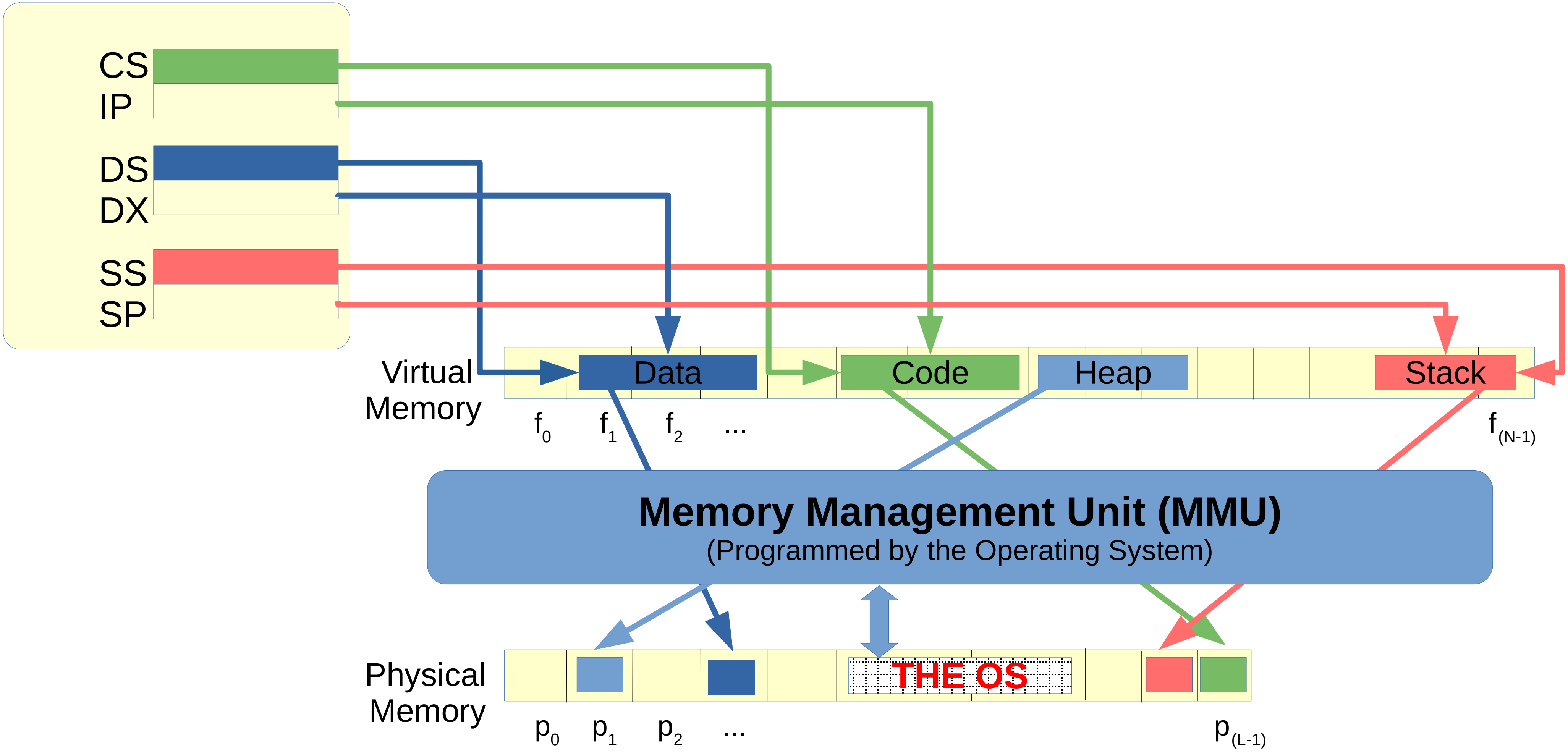

CS
IP
DS
DX
SS
SP
Dynamic Memory Allocation Instructions Sequence in the Program.
Virtual Memory
Data
Code
Heap
Stack
f_0
f_1
f_2
...
$f_{(N-1)}$
Memory Management Unit (MMU)
(Programmed by the Operating System)
Physical Memory
THE OS
p_0
p_1
p_2
...
$p_{(L-1)}$

"Program in memory": Execution using CPU + Mem



Process ► Program

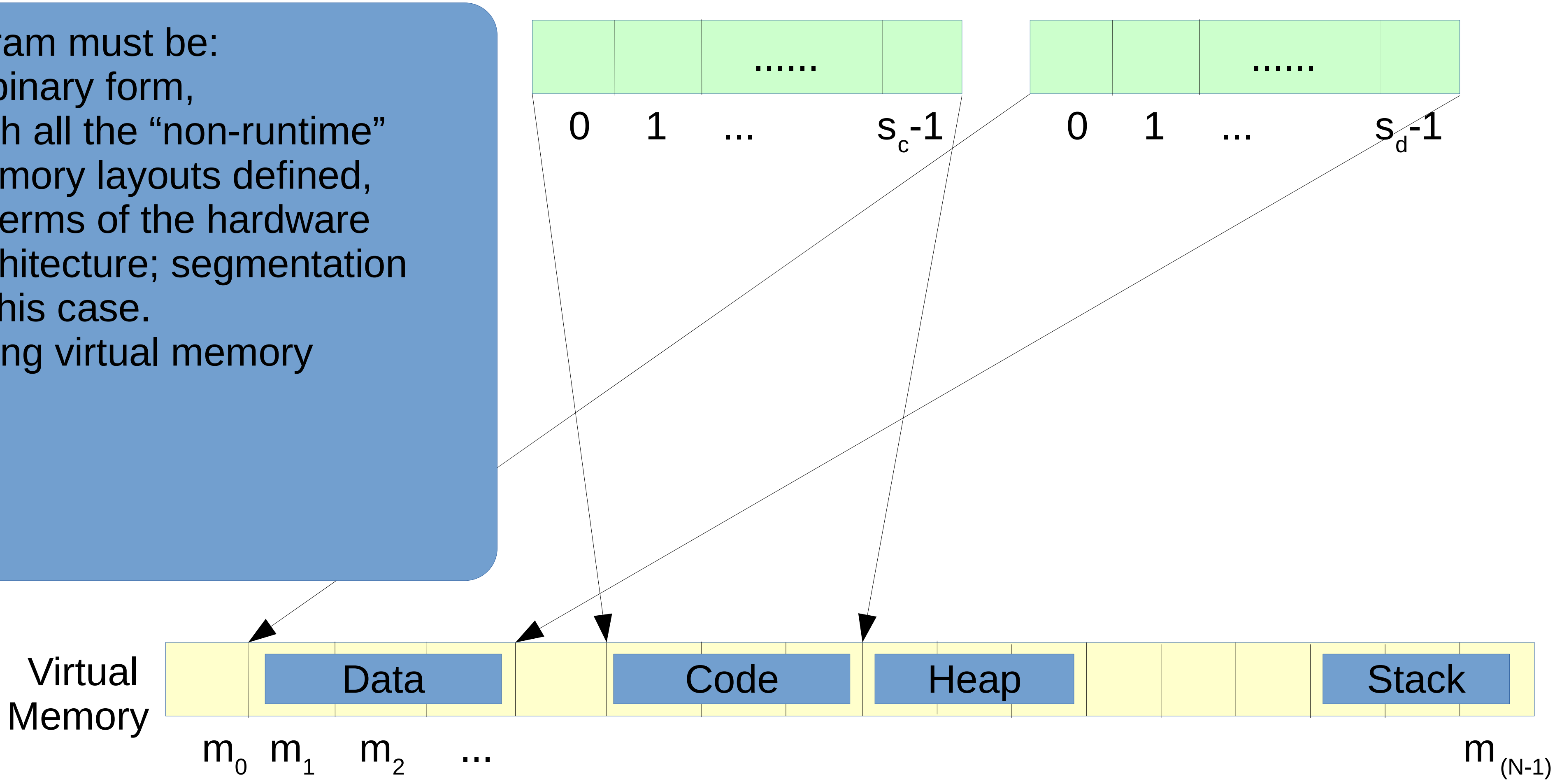
“Program in memory”: Execution using CPU + Mem



“Program in memory”: Expectations Summary

Program must be:

- In binary form,
- With all the “non-runtime” memory layouts defined,
- In terms of the hardware architecture; segmentation in this case.
- Using virtual memory

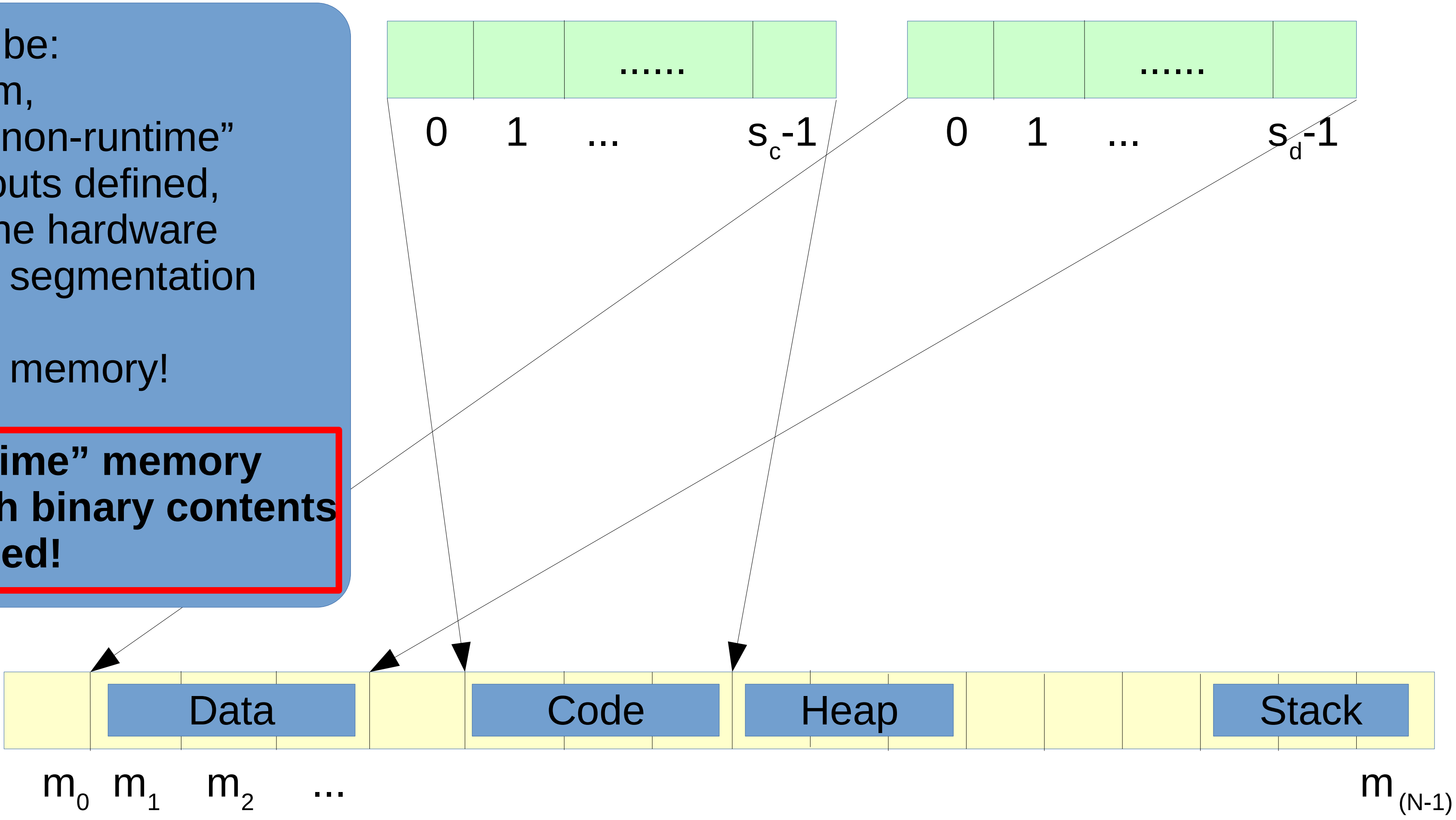


"Program in memory": Expectations Summary

Program must be:

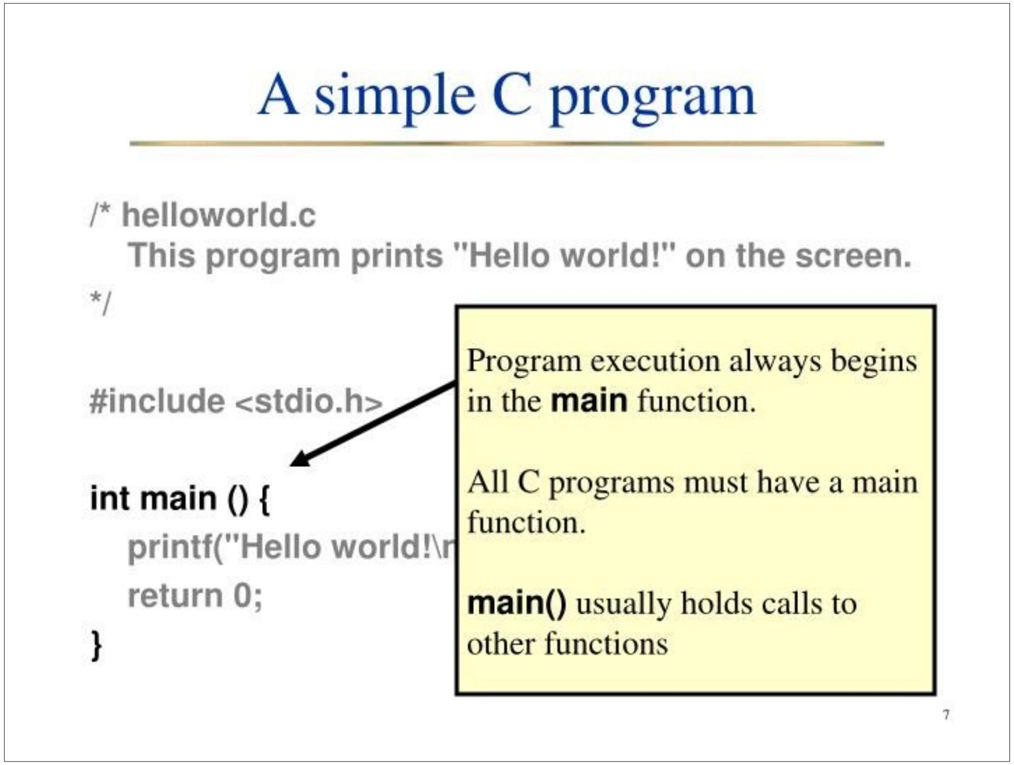
- In binary form,
- With all the "non-runtime" memory layouts defined,
- In terms of the hardware architecture; segmentation in this case,
- Using virtual memory!

All "compile time" memory segments with binary contents must be defined!



Programmers and Programming

- Simple Sample C program
- OS: GNU/Linux
  - Ideas are same for other OSes
  - Details differ
- Assume
  - x86 ASM programming
  - x86 based Architecture



A simple C program

```
/* helloworld.c
   This program prints "Hello world!" on the screen.
*/

#include <stdio.h>

int main () {
   printf("Hello world!\n
   return 0;
}
```

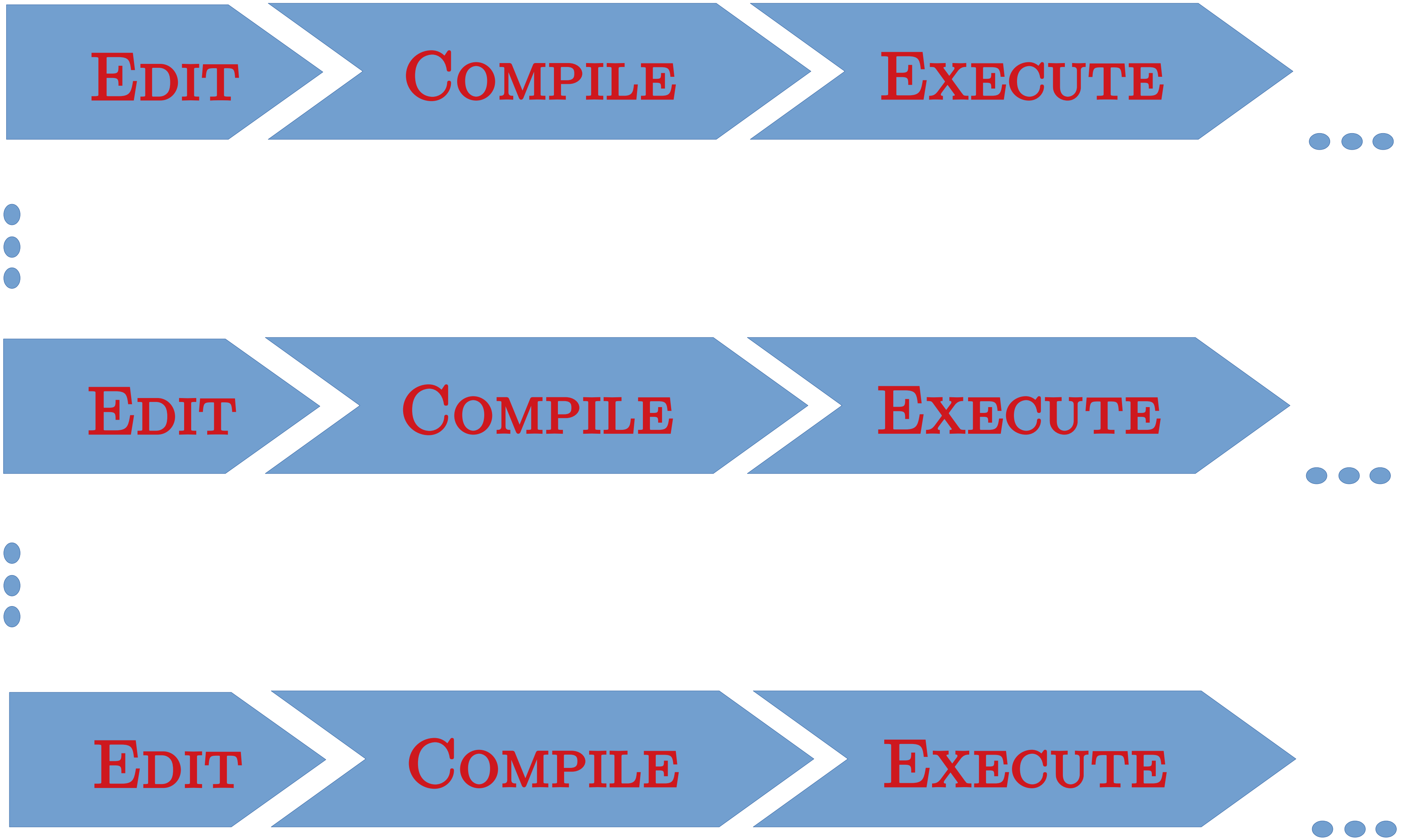
Program execution always begins in the **main** function.

All C programs must have a main function.

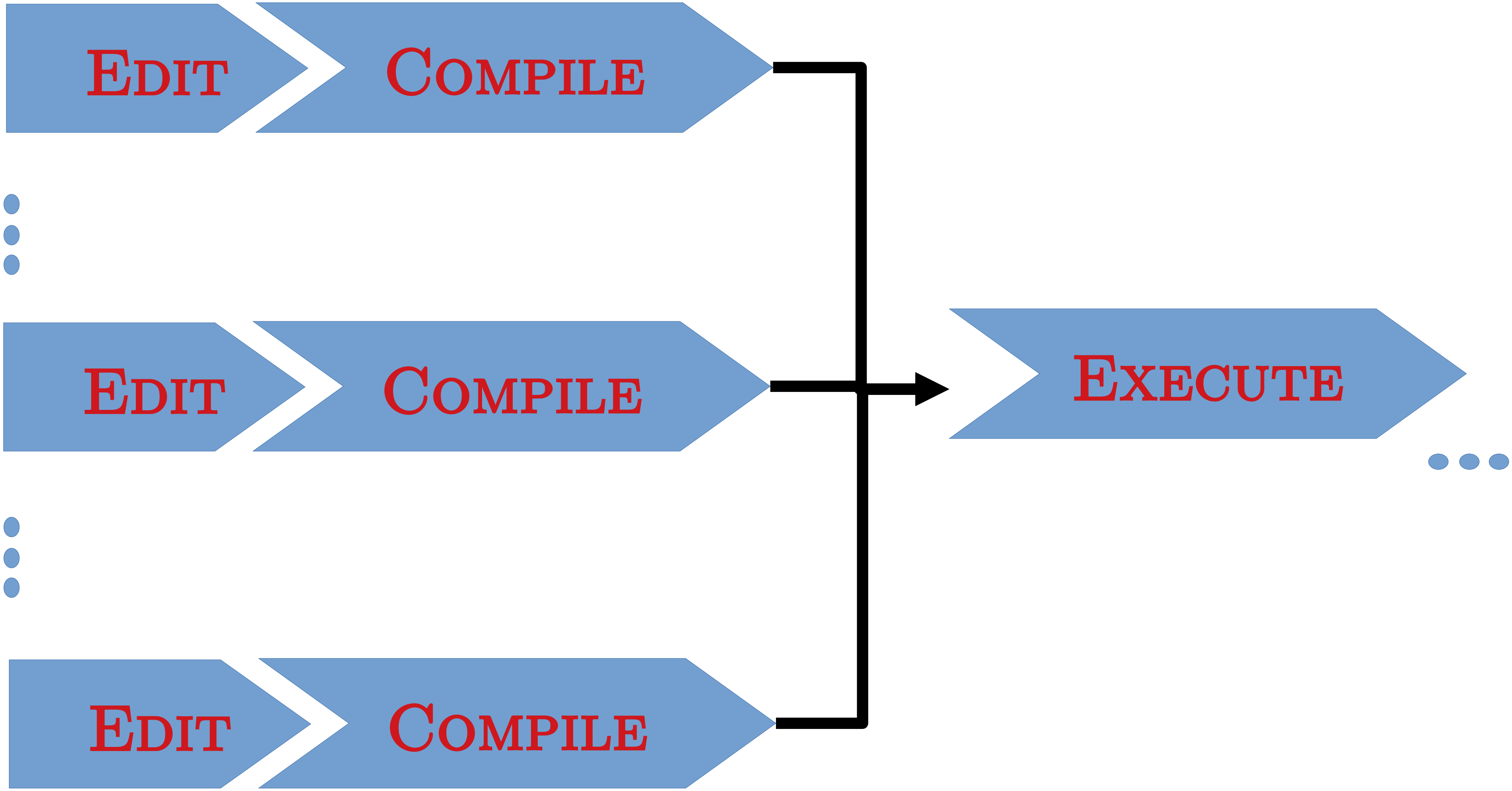
main() usually holds calls to other functions

7

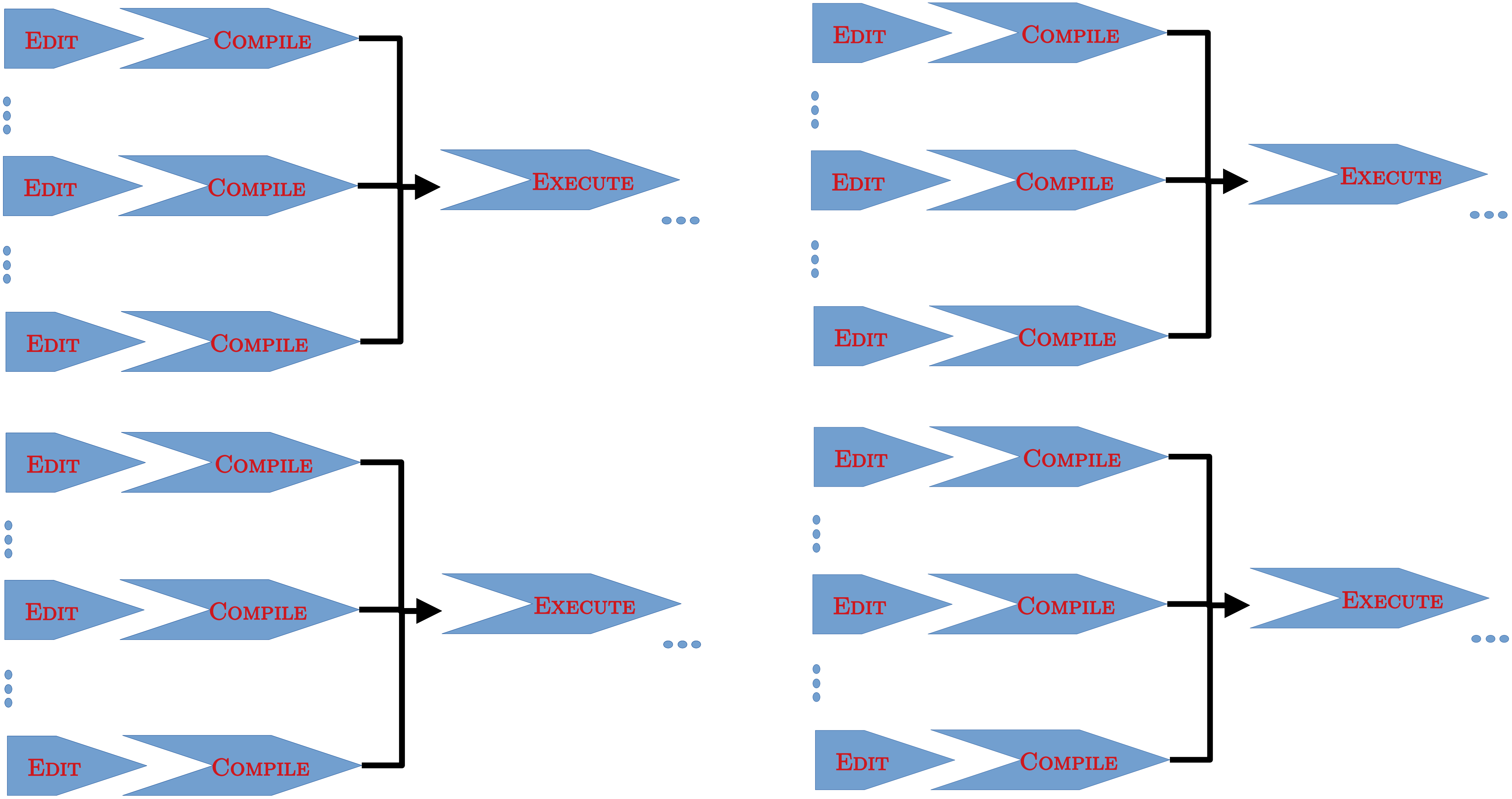
Typical (Student) Work Sequence



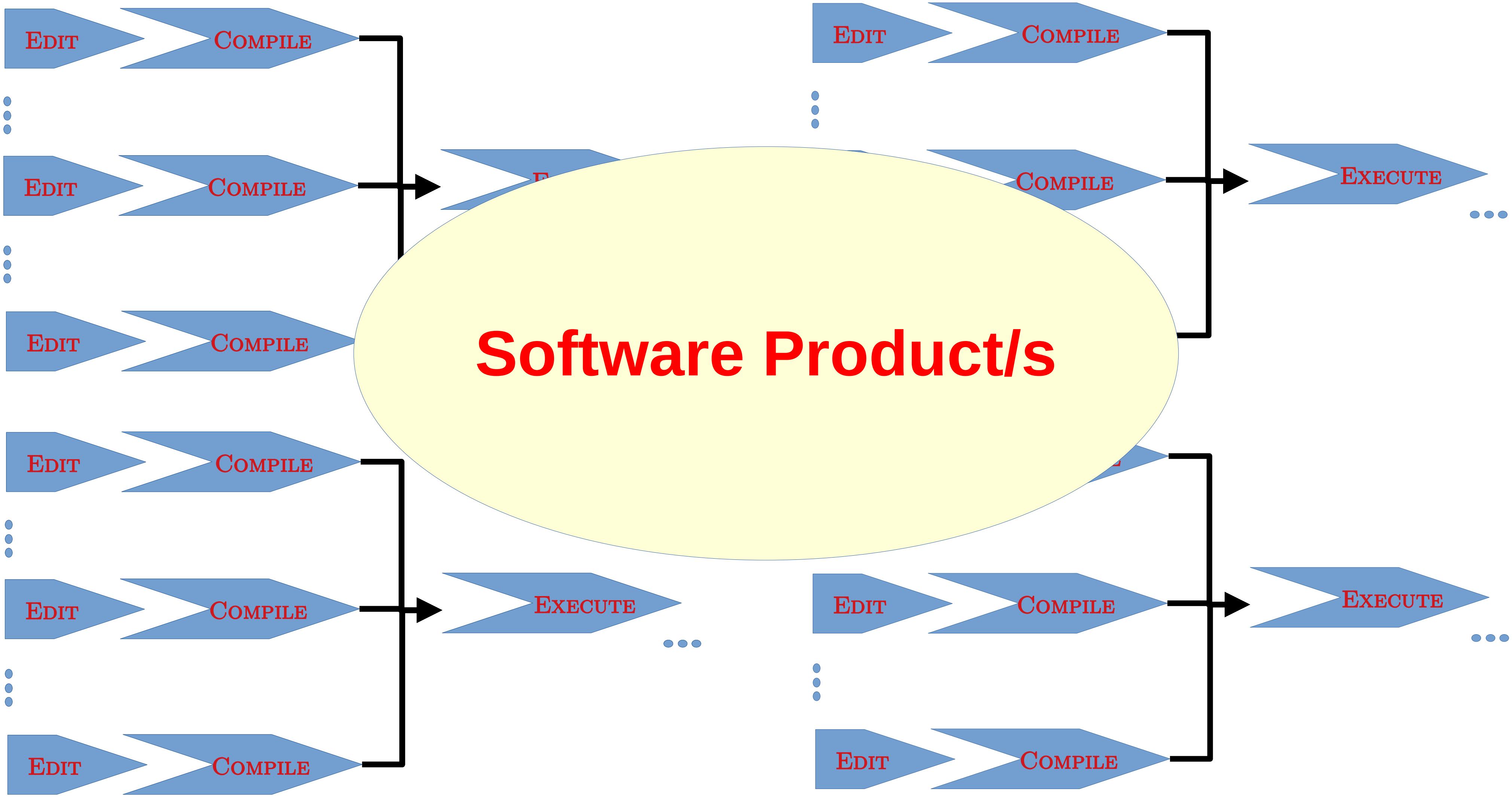
Typical (Devel.) Work Sequence



Typical (Org.) Work Sequence



Typical (Org.) Work Sequence



Program ► Process

- Program ► Process
  - Source Program ► Binary Program
  - Binary Program ► Process

Program ► Process

- Program ► Process
  - Source Program ► Binary Program
  - Binary Program ► Proce

We assume a translations,
i.e. **compilation**,
based approach.

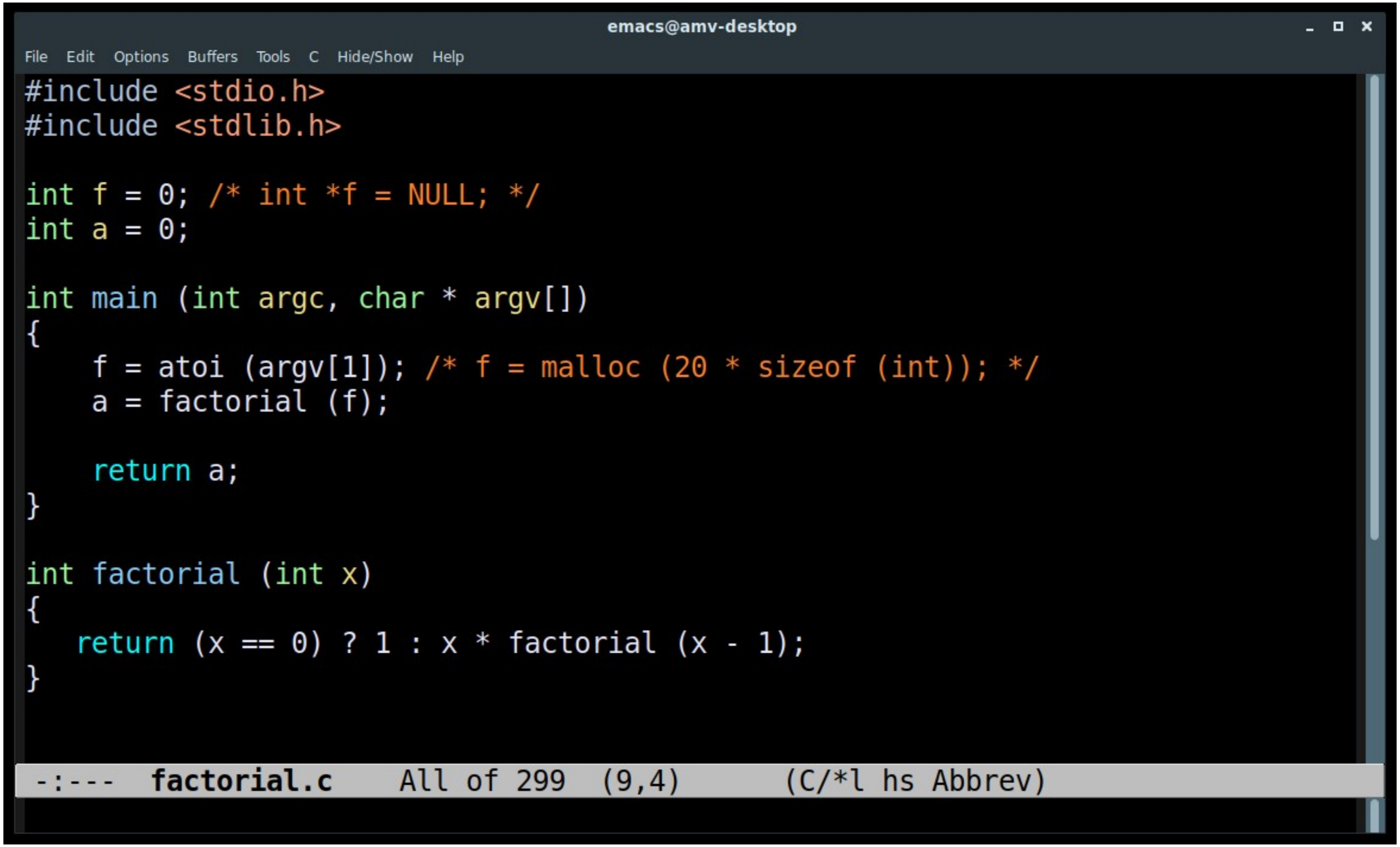
Program ► Process

- Program ► Process
  - Source Program ► Binary Program
  - Binary Program ► Process

We need to gradually transform our source program to its equivalent binary version in a way such that the target hardware + OS can execute the binary.

Program ► Process

Program ► Process: Basic Compiler Driver, e.g. gcc



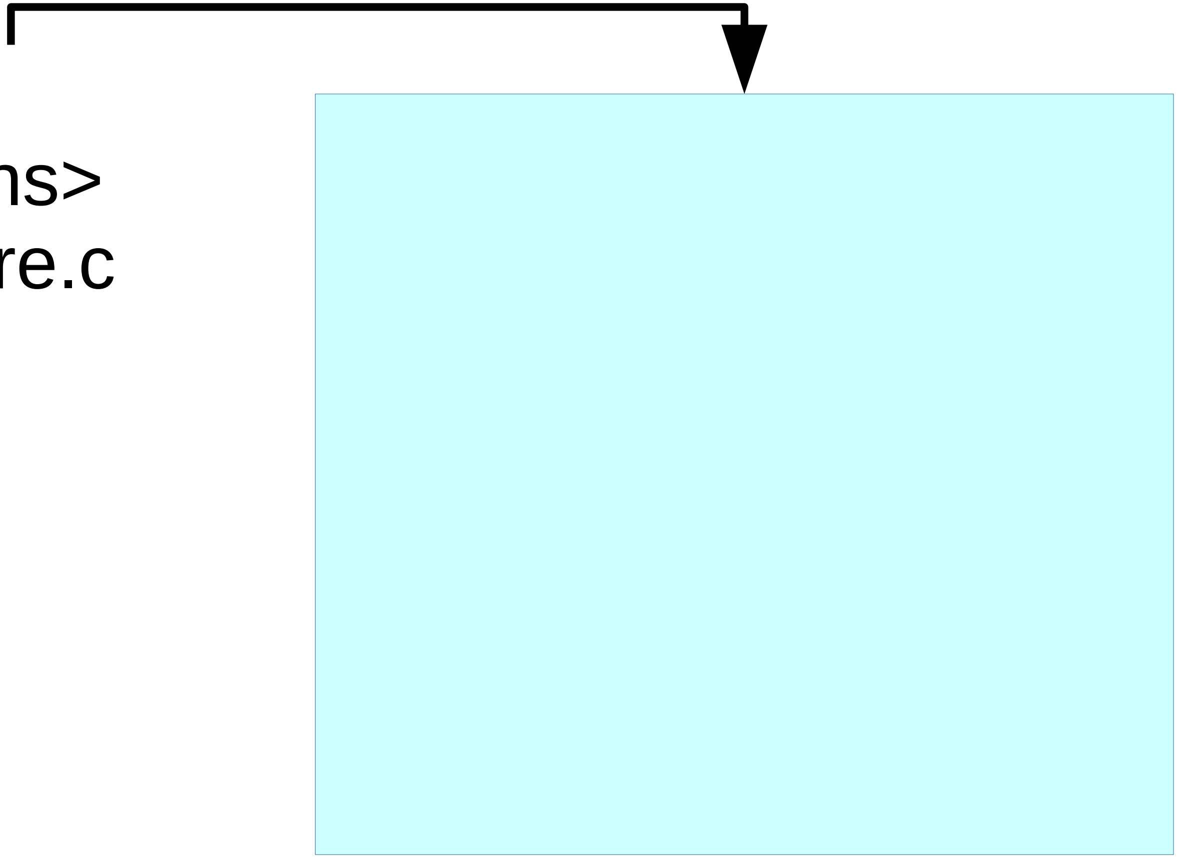
```
#include <stdio.h>
#include <stdlib.h>

int f = 0; /* int *f = NULL; */
int a = 0;

int main (int argc, char * argv[])
{
    f = atoi (argv[1]); /* f = malloc (20 * sizeof (int)); */
    a = factorial (f);

    return a;
}

int factorial (int x)
{
   return (x == 0) ? 1 : x * factorial (x - 1);
}
```

```
-:---  factorial.c    All of 299  (9,4)      (C/*l hs Abbrev)
```

Program ► Process

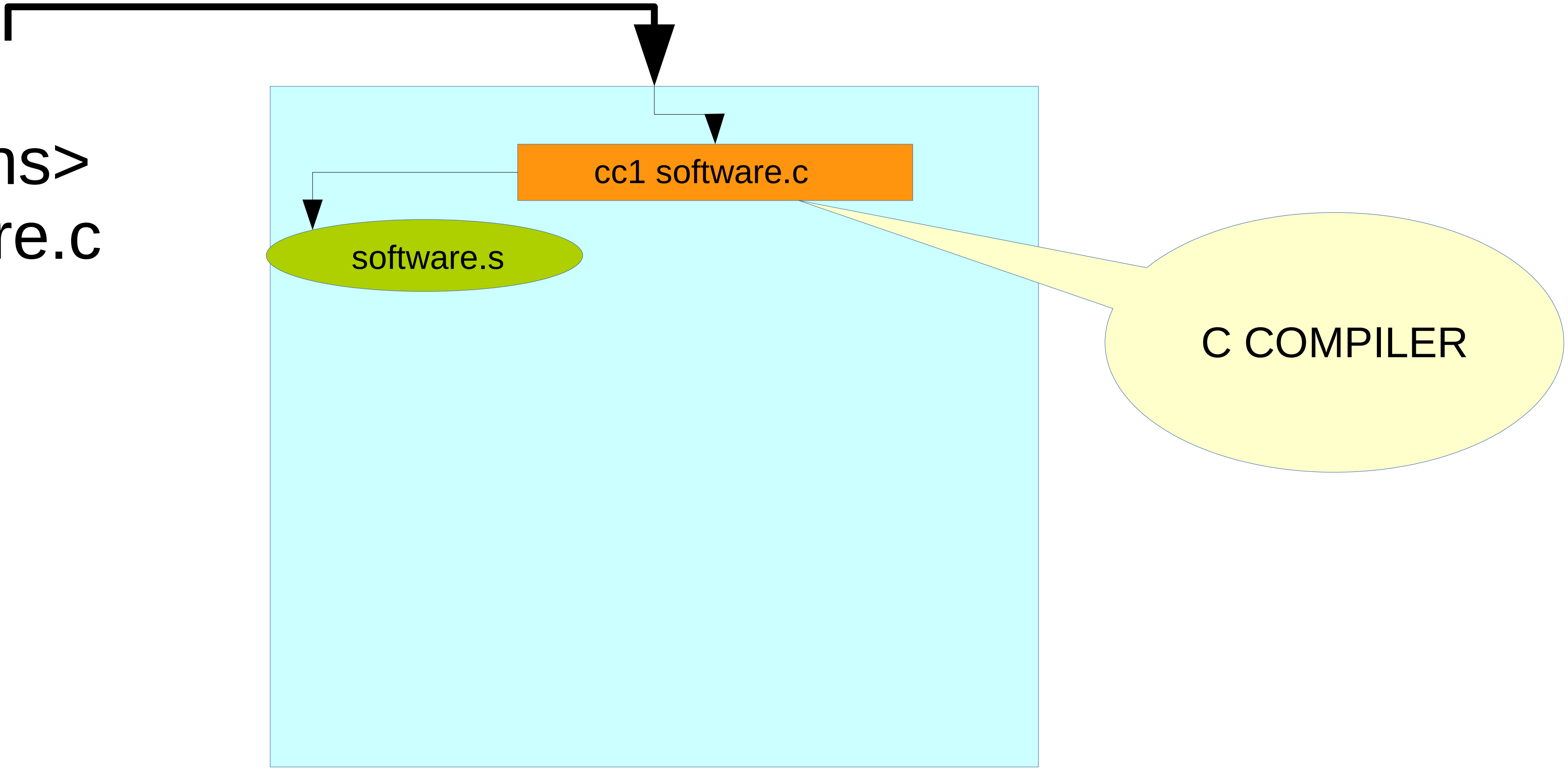
Program ► Process: Basic Compiler Driver, e.g. **gcc**

gcc -<options> software.c



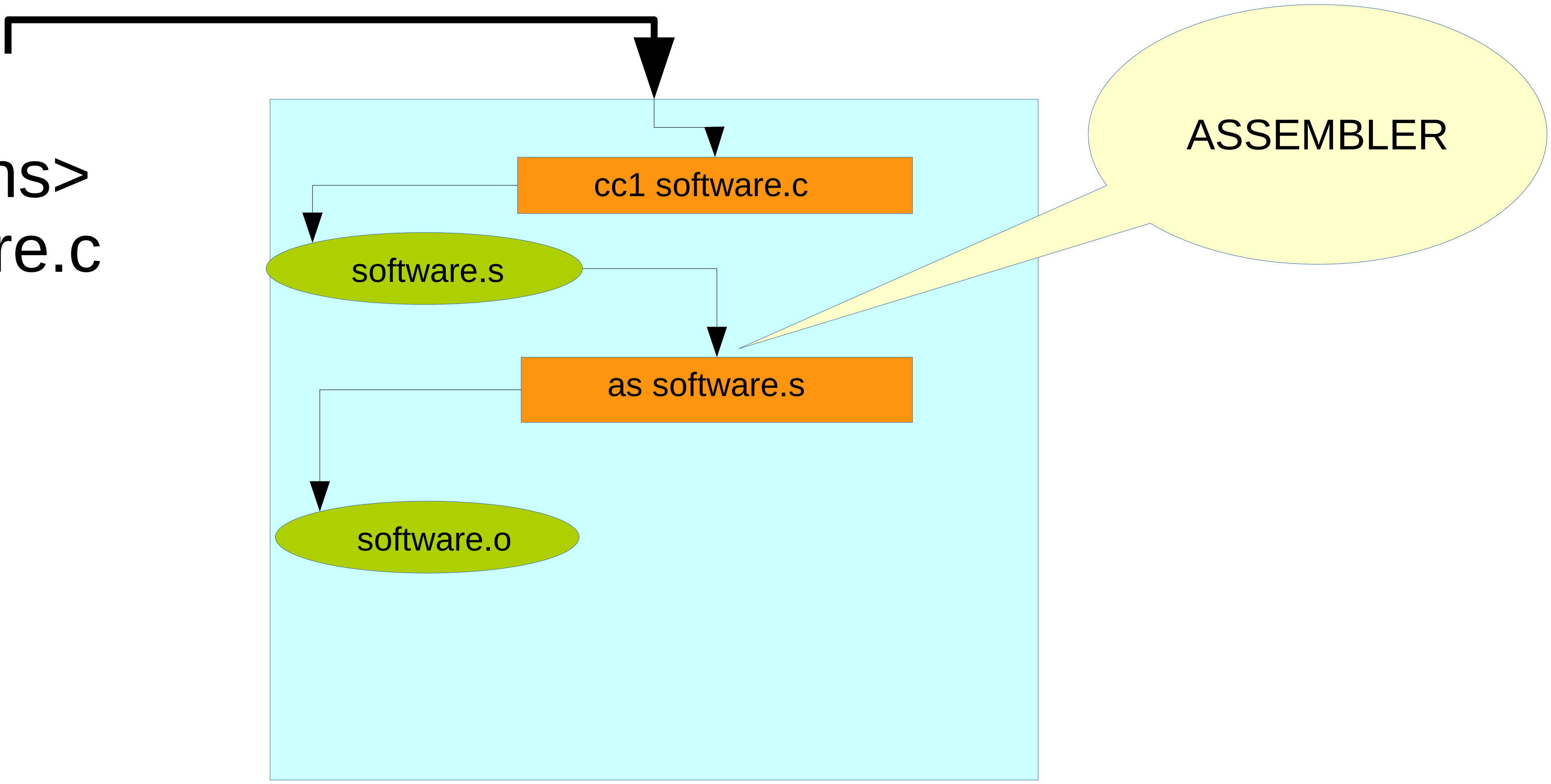
Program ► Process: Basic Compiler Driver, e.g. gcc

gcc -<options> software.c



Program ► Process: Basic Compiler Driver, e.g. **gcc**

gcc -<options> software.c



cc1 software.c

software.s

as software.s

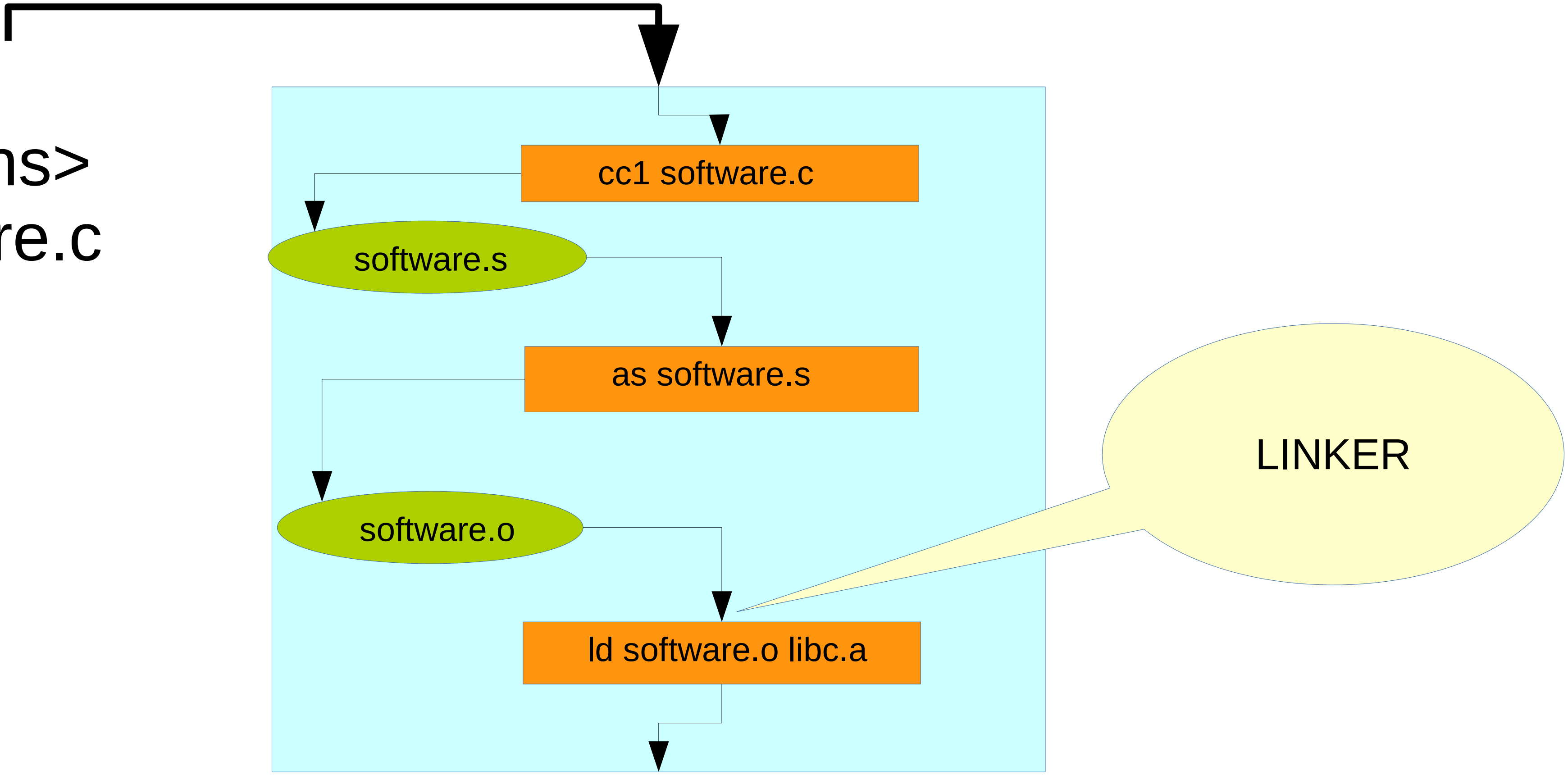
software.o

ASSEMBLER

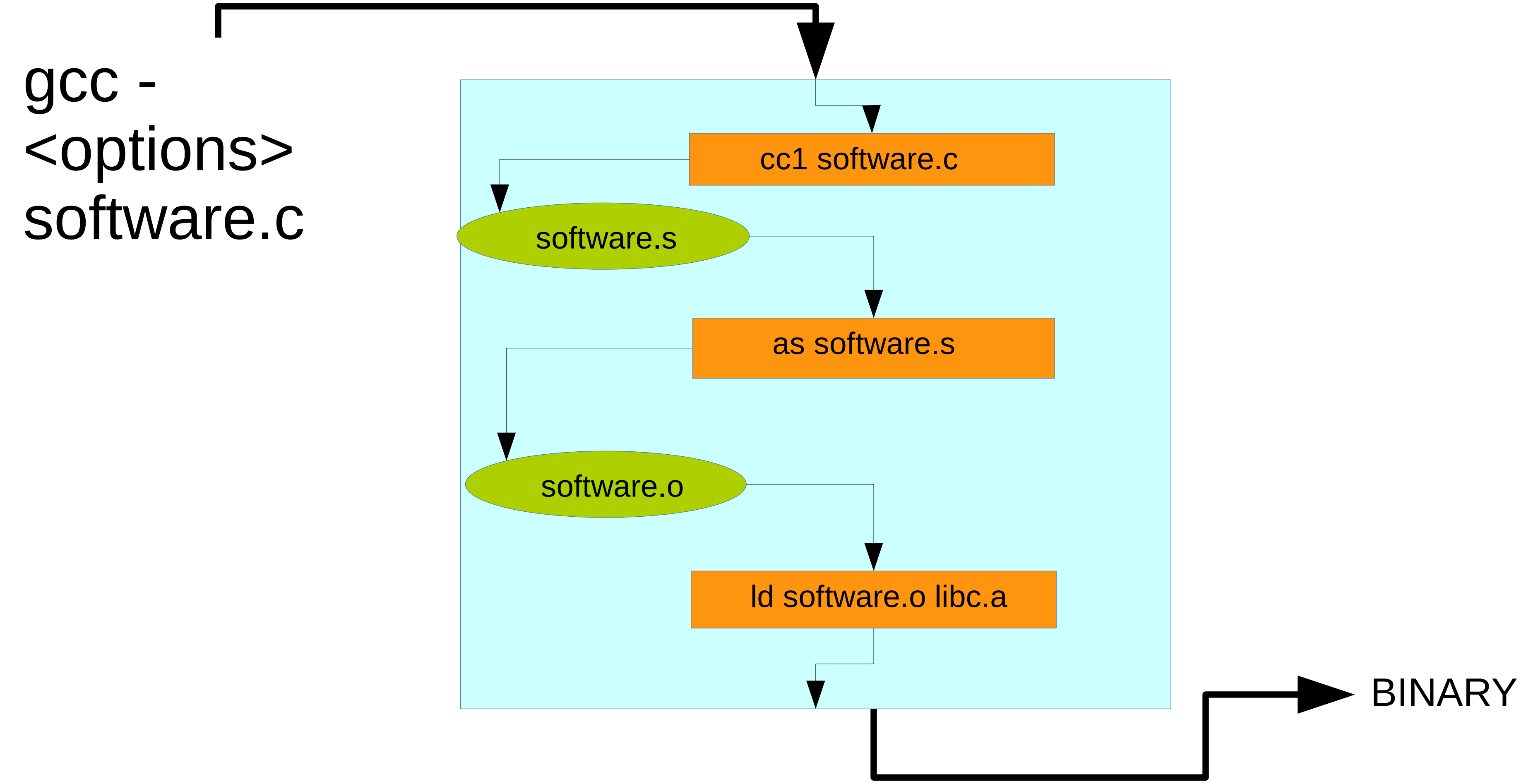
Program ► Process

Program ► Process: Basic Compiler Driver, e.g. **gcc**

gcc -<options> software.c

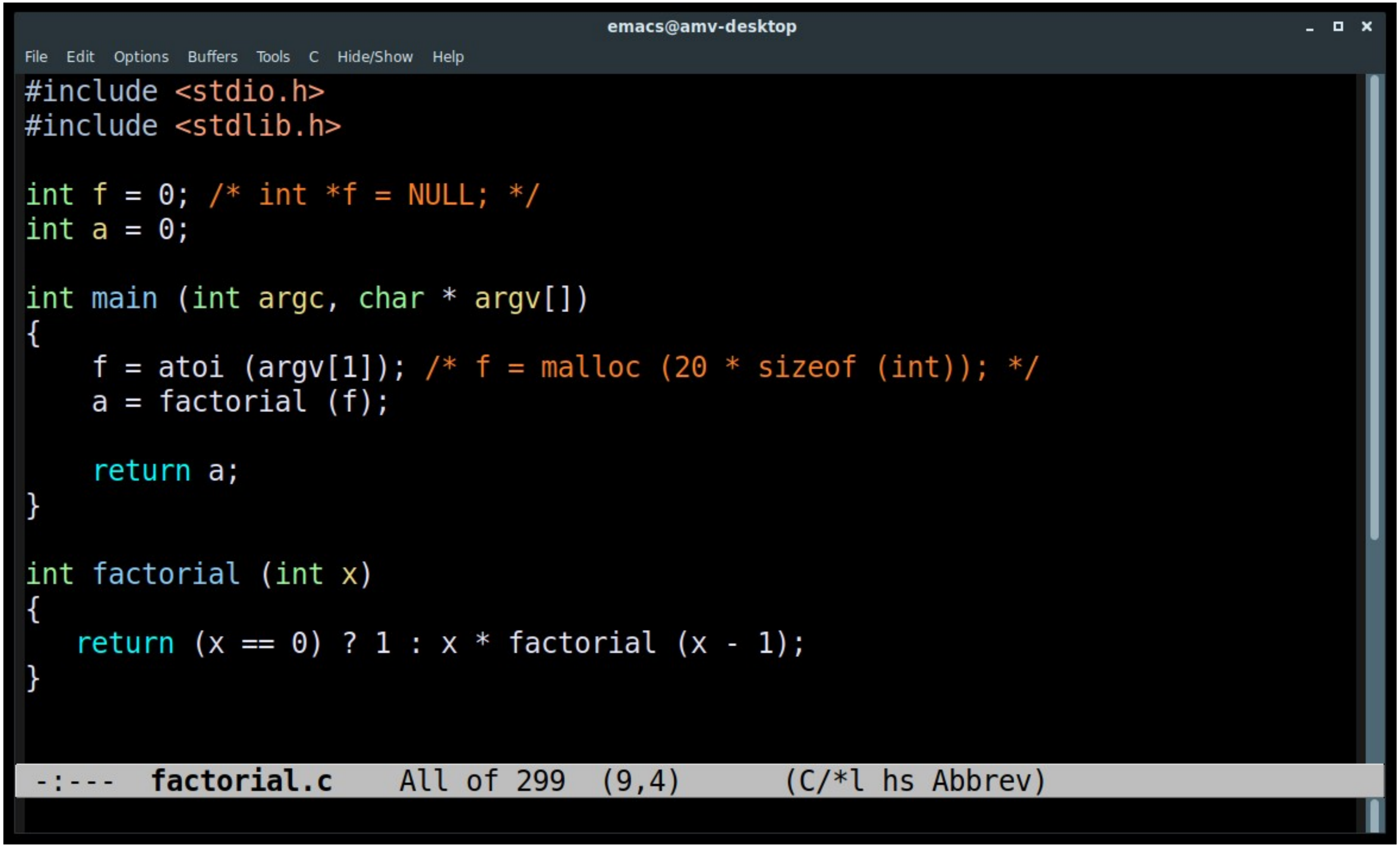


Program ► Process: Basic Compiler Driver, e.g. **gcc**



Program ► Process

Program ► Process: Basic Compiler Driver, e.g. **gcc**

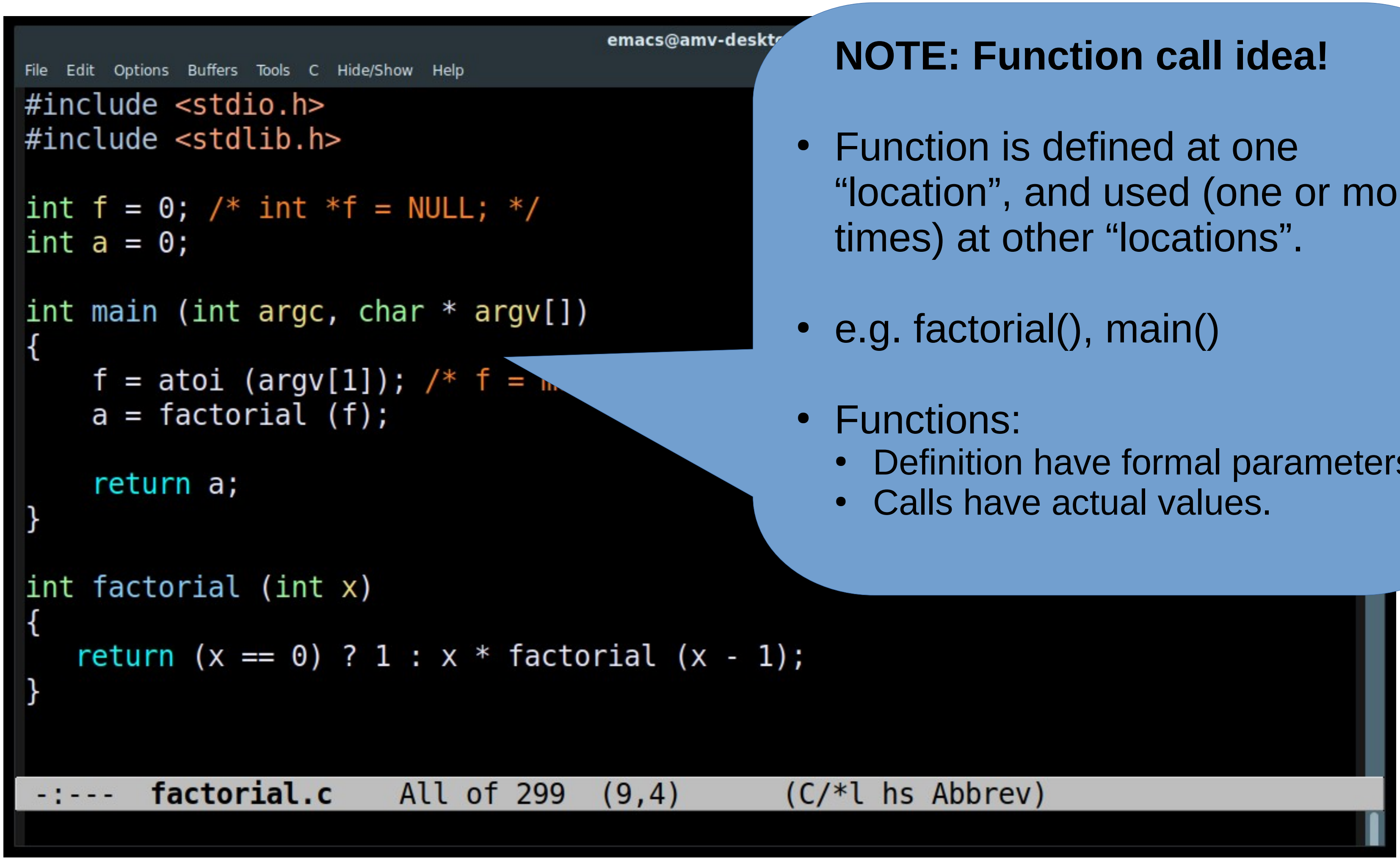


```c
#include <stdio.h>
#include <stdlib.h>

int f = 0; /* int *f = NULL; */
int a = 0;

int main (int argc, char * argv[])
{
    f = atoi (argv[1]); /* f = malloc (20 * sizeof (int)); */
    a = factorial (f);

    return a;
}

int factorial (int x)
{
   return (x == 0) ? 1 : x * factorial (x - 1);
}
```

```
-:---  factorial.c    All of 299  (9,4)       (C/*l hs Abbrev)
```

Program ► Process

Program ► Process: Basic Compiler Driver, e.g. **gcc**



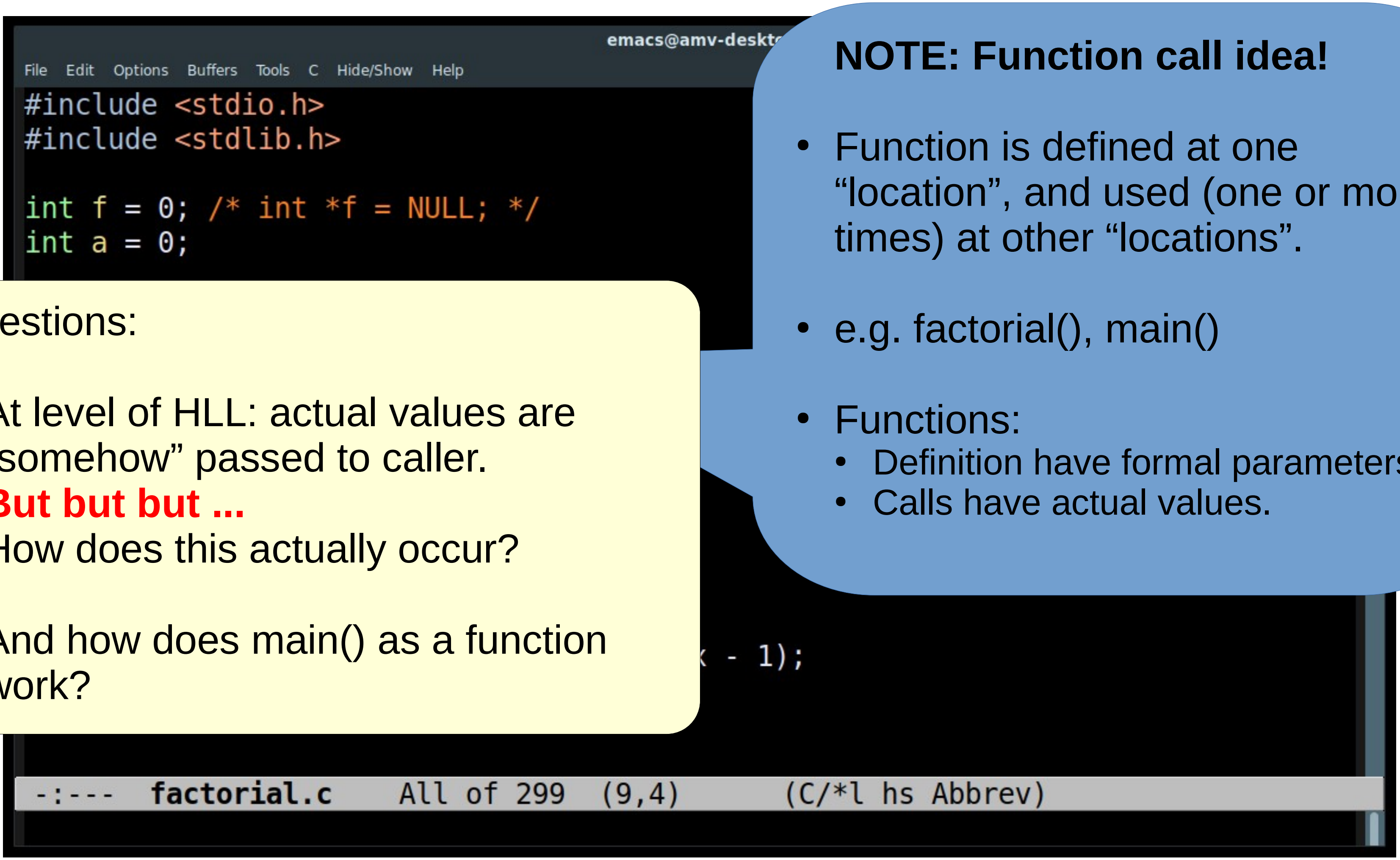
```
#include <stdio.h>
#include <stdlib.h>

int f = 0; /* int *f = NULL; */
int a = 0;

int main (int argc, char * argv[])
{
    f = atoi (argv[1]); /* f = m
    a = factorial (f);

    return a;
}

int factorial (int x)
{
   return (x == 0) ? 1 : x * factorial (x - 1);
}
```

```
-:---  factorial.c    All of 299  (9,4)      (C/*l hs Abbrev)
```

NOTE: Function call idea!

- Function is defined at one “location”, and used (one or more times) at other “locations”.
- e.g. factorial(), main()
- Functions:
  - Definition have formal parameters.
  - Calls have actual values.

Program ► Process: Basic Compiler Driver, e.g. gcc



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

int f = 0; /* int *f = NULL; */
int a = 0;

                                     - 1);

-:---  factorial.c    All of 299   (9,4)        (C/*l hs Abbrev)
```

NOTE: Function call idea!

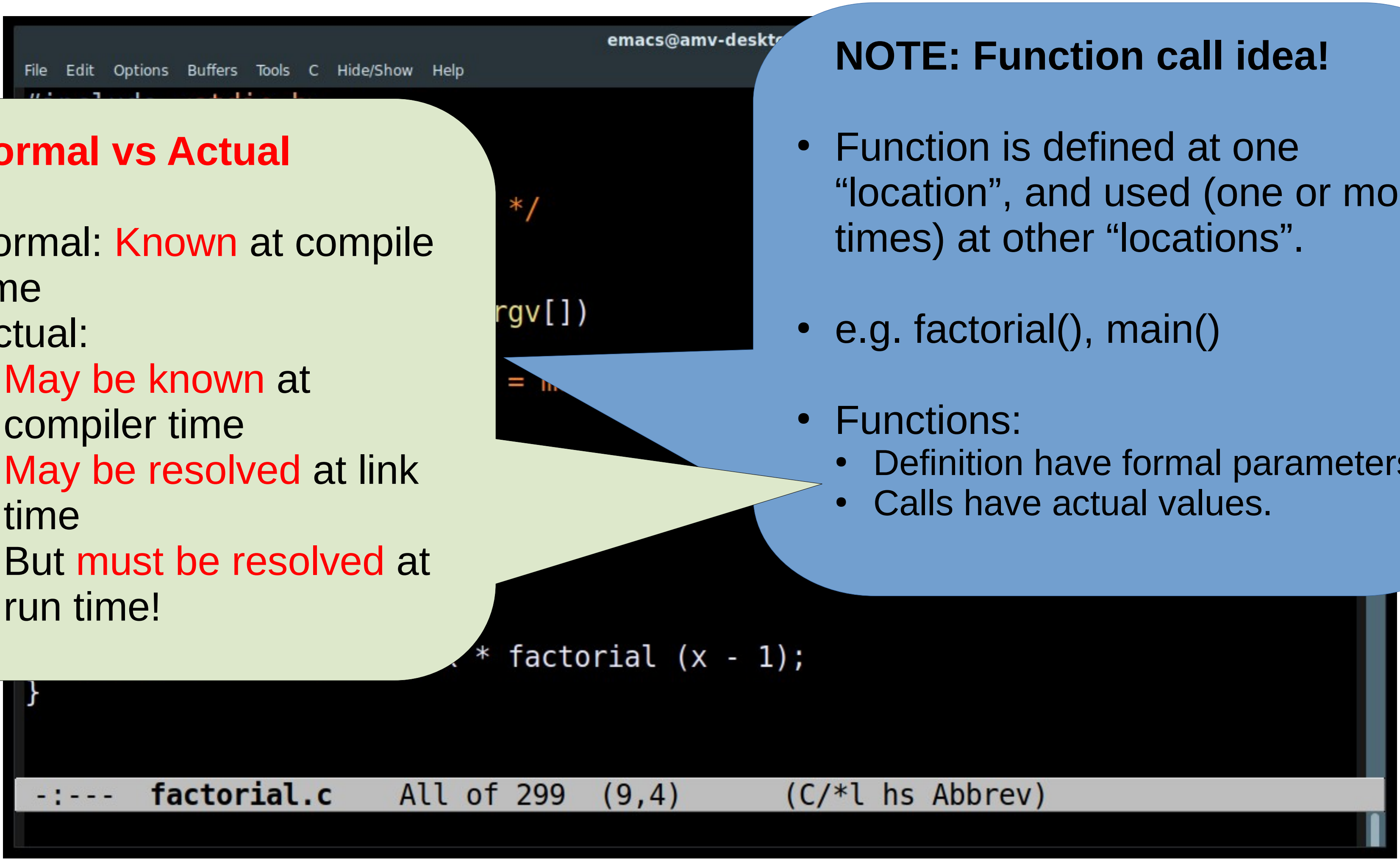
- Function is defined at one “location”, and used (one or more times) at other “locations”.
- e.g. factorial(), main()
- Functions:
  - Definition have formal parameters.
  - Calls have actual values.

Questions:

- At level of HLL: actual values are “somehow” passed to caller.
- **But but but ...**
  How does this actually occur?
- And how does main() as a function work?

Program ► Process

Program ► Process: Basic Compiler Driver, e.g. gcc



```
*/
rgv[])
= m
* factorial (x - 1);
}
-:---  factorial.c    All of 299   (9,4)      (C/*l hs Abbrev)
```

Formal vs Actual

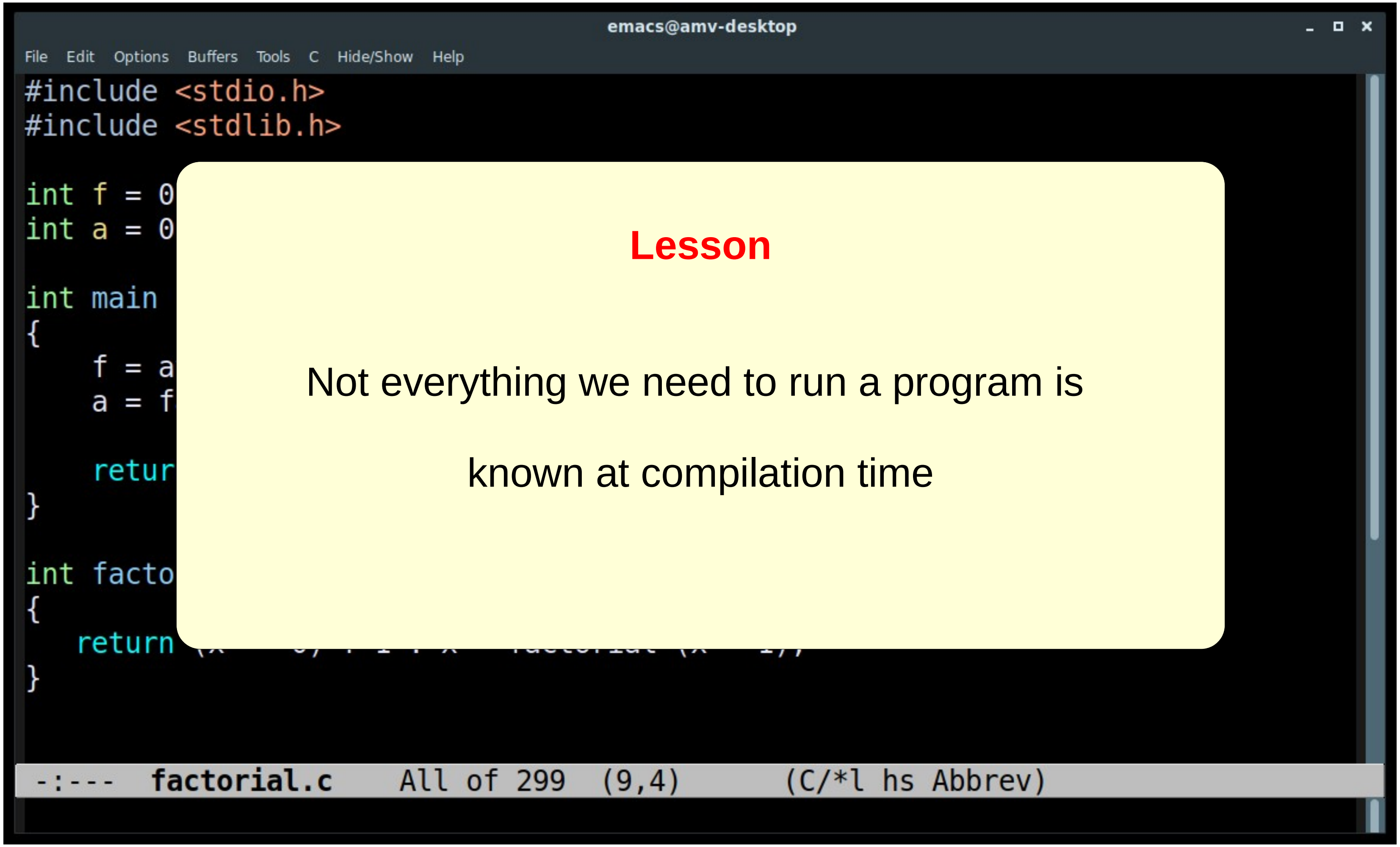
- Formal: Known at compile time
- Actual:
  - May be known at compiler time
  - May be resolved at link time
  - But must be resolved at run time!

NOTE: Function call idea!

- Function is defined at one “location”, and used (one or more times) at other “locations”.
- e.g. factorial(), main()
- Functions:
  - Definition have formal parameters.
  - Calls have actual values.

Program ► Process

Program ► Process: Basic Compiler Driver, e.g. **gcc**



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

int f = 0
int a = 0

int main
{
    f = a
    a = f

    retur
}

int facto
{
   return
}
```

```
-:---  factorial.c    All of 299  (9,4)      (C/*l hs Abbrev)
```

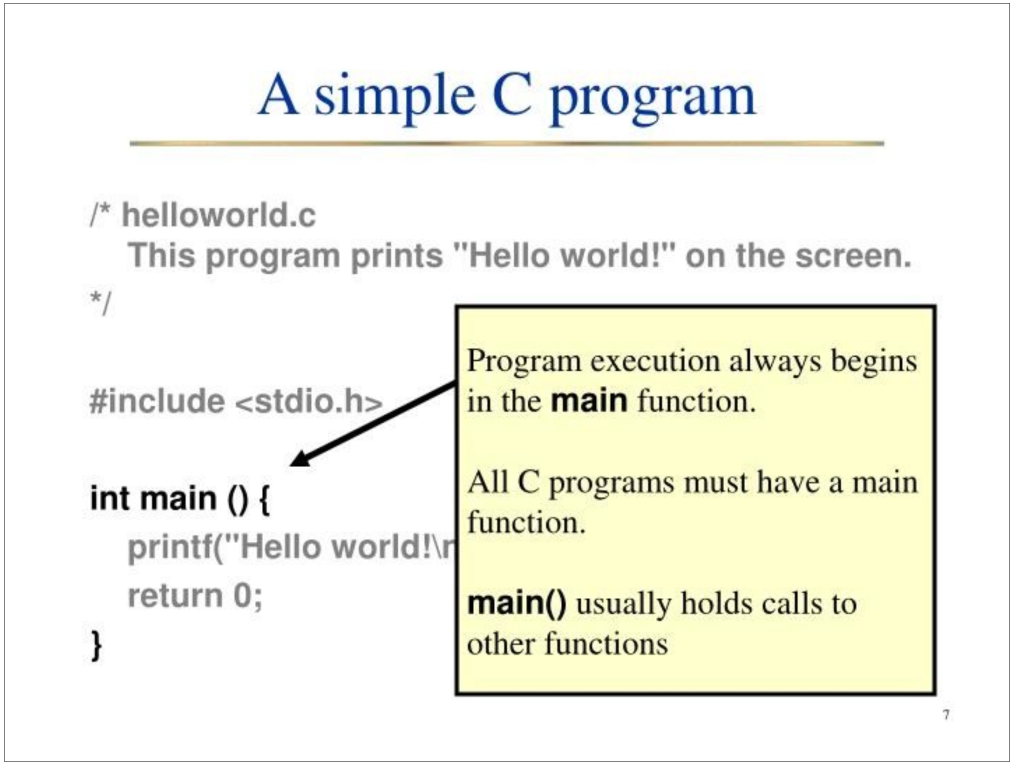
Lesson

Not everything we need to run a program is

known at compilation time

“Preface”

- Simple Sample C program
- OS: GNU/Linux
  - Ideas are same for other OSes
  - Details differ
- Assume
  - x86 ASM programming
  - x86 based Architecture



A simple C program

```
/* helloworld.c
   This program prints "Hello world!" on the screen.
*/

#include <stdio.h>

int main () {
   printf("Hello world!\n
   return 0;
}
```

Program execution always begins in the **main** function.

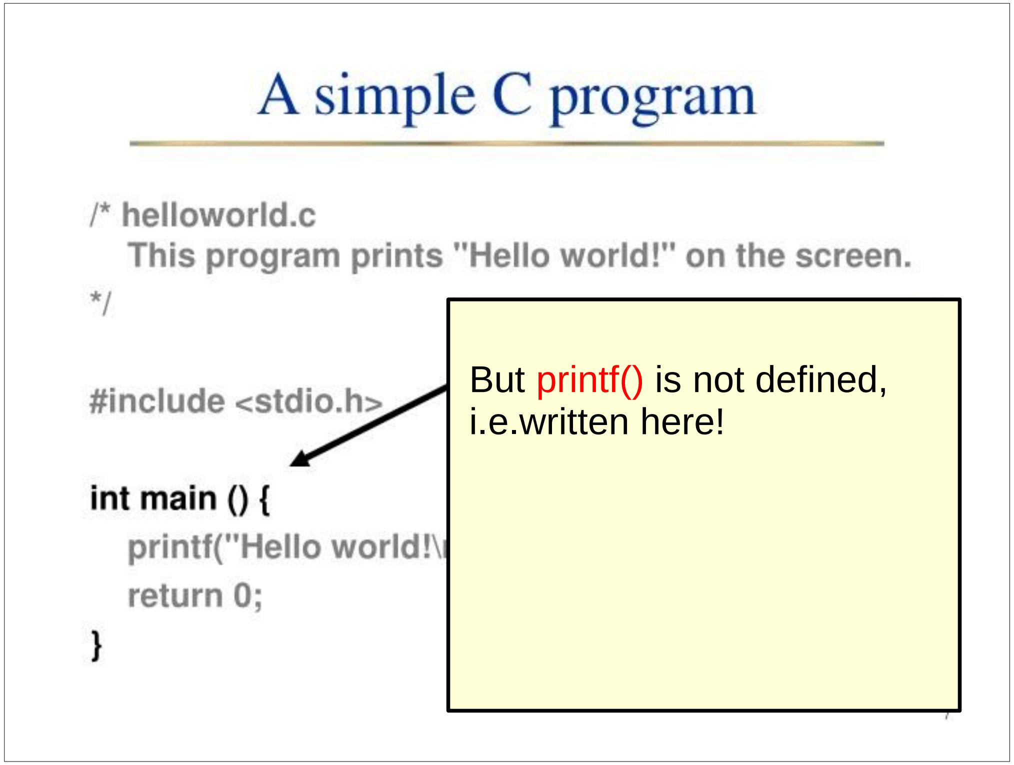
All C programs must have a main function.

main() usually holds calls to other functions

7

“Preface”

- Simple Sample C program
- OS: GNU/Linux
  - Ideas are same for other OSes
  - Details differ
- Assume
  - x86 ASM programming
  - x86 based Architecture



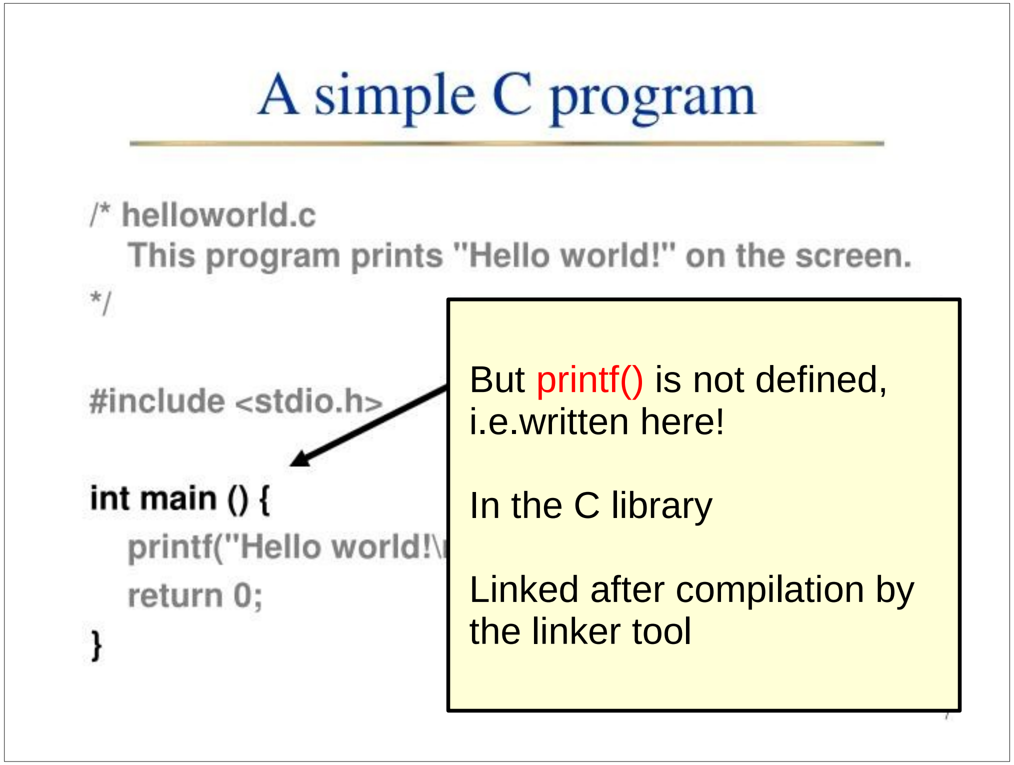
A simple C program

```
/* helloworld.c
   This program prints "Hello world!" on the screen.
*/

#include <stdio.h>

int main () {
   printf("Hello world!\
   return 0;
}
```

But printf() is not defined, i.e.written here!

“Preface”

- Simple Sample C program
- OS: GNU/Linux
  - Ideas are same for other OSes
  - Details differ
- Assume
  - x86 ASM programming
  - x86 based Architecture



A simple C program

```
/* helloworld.c
   This program prints "Hello world!" on the screen.
*/

#include <stdio.h>

int main () {
   printf("Hello world!\
   return 0;
}
```

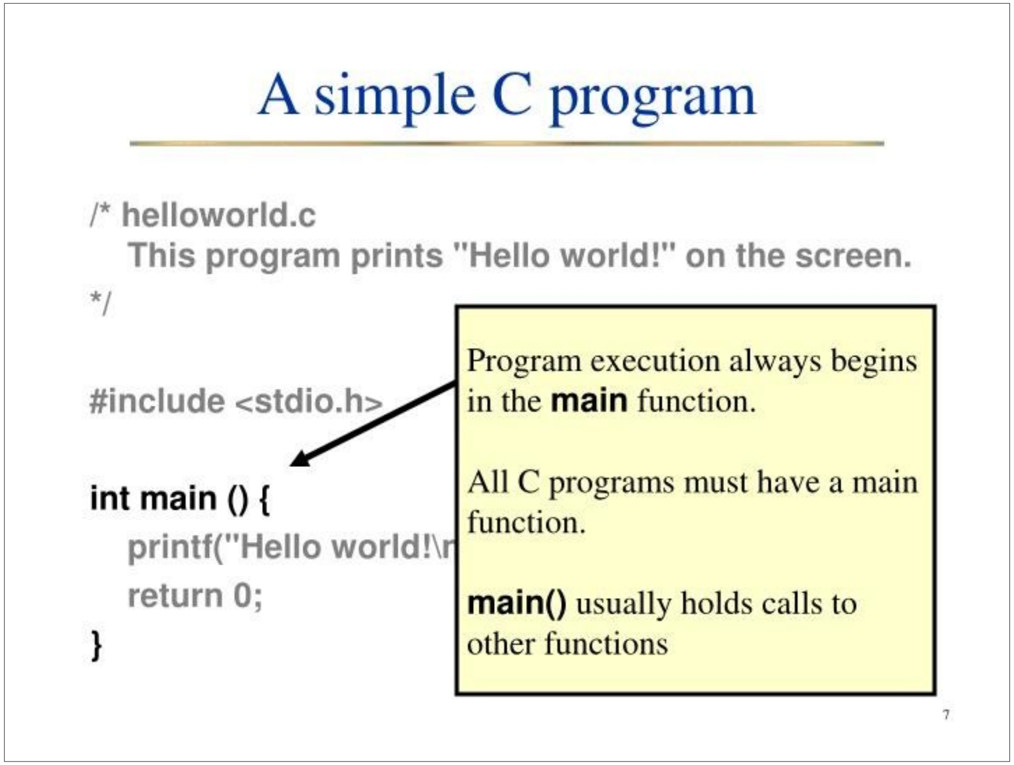
But printf() is not defined, i.e.written here!

In the C library

Linked after compilation by the linker tool

“Preface”

- Simple Sample C program
- OS: GNU/Linux
  - Ideas are same for other OSes
  - Details differ
- Assume
  - x86 ASM programming
  - x86 based Architecture



A simple C program

```
/* helloworld.c
   This program prints "Hello world!" on the screen.
*/

#include <stdio.h>

int main () {
   printf("Hello world!\
   return 0;
}
```

Program execution always begins in the **main** function.

All C programs must have a main function.

main() usually holds calls to other functions

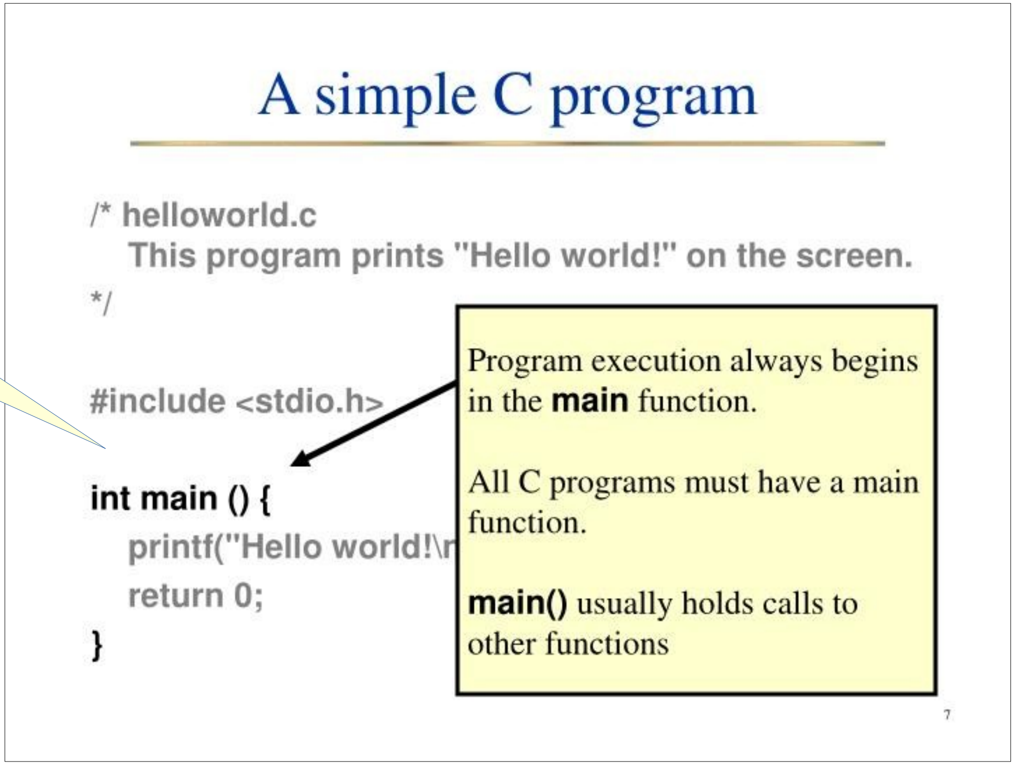
7

“Preface”

- S program
- O
  - Ideas are s r other OSes
  - Details differ
- Assume
  - x86 ASM programming
  - x86 based Architecture

QUESTION

If main() is a function,
Who “**calls**” it, and **how**?



A simple C program

```
/* helloworld.c
   This program prints "Hello world!" on the screen.
*/

#include <stdio.h>

int main () {
   printf("Hello world!\n
   return 0;
}
```

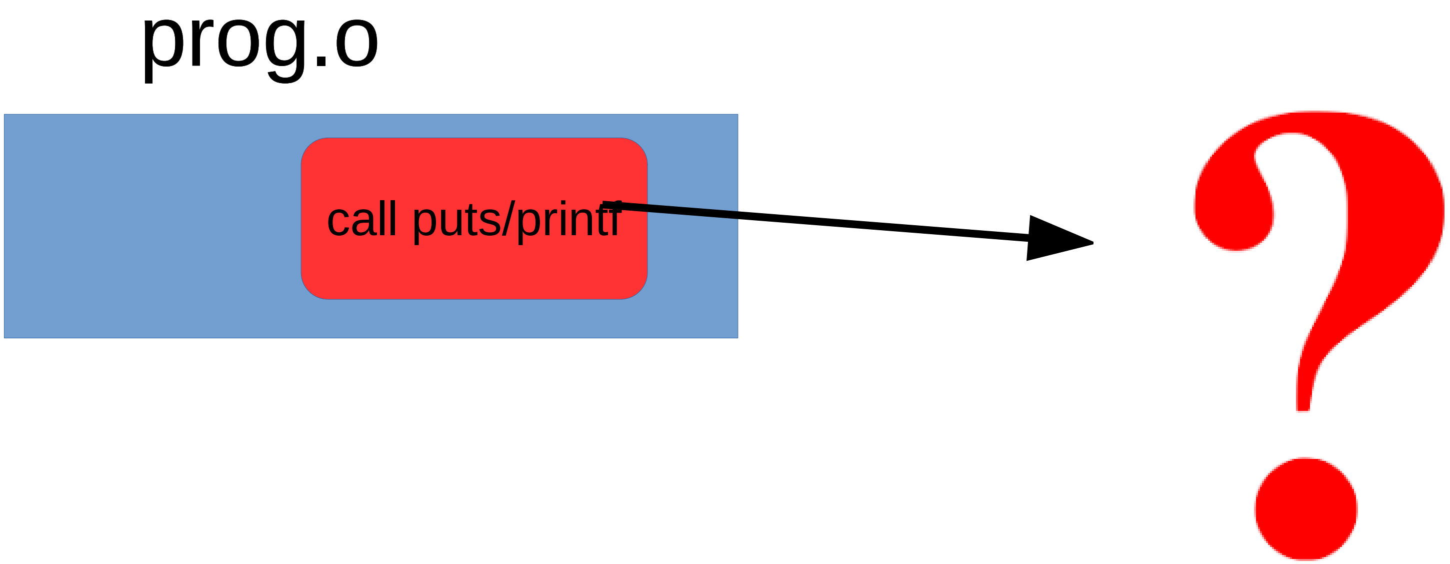
Program execution always begins in the **main** function.

All C programs must have a main function.

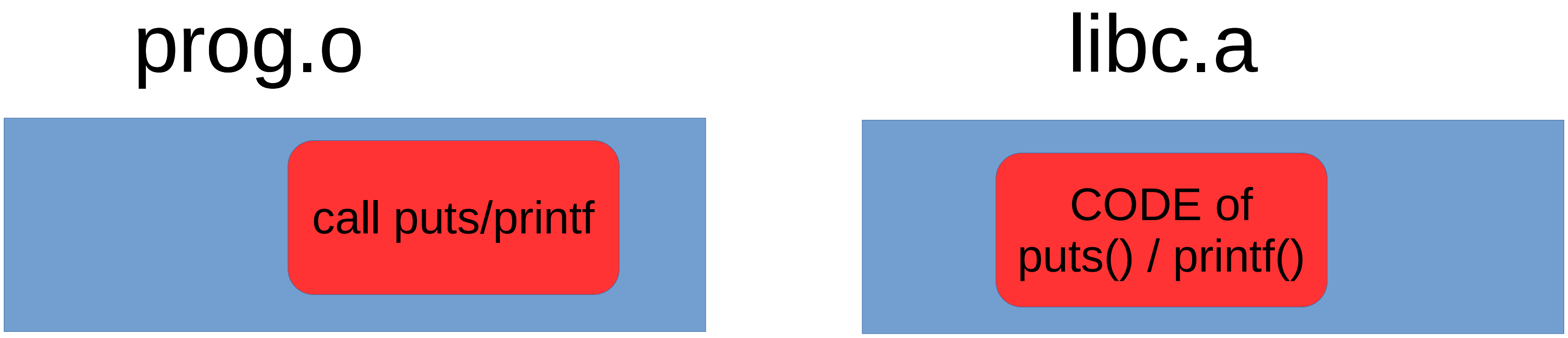
main() usually holds calls to other functions

7

Intro to "Linker" solution



Intro to "Linker" solution



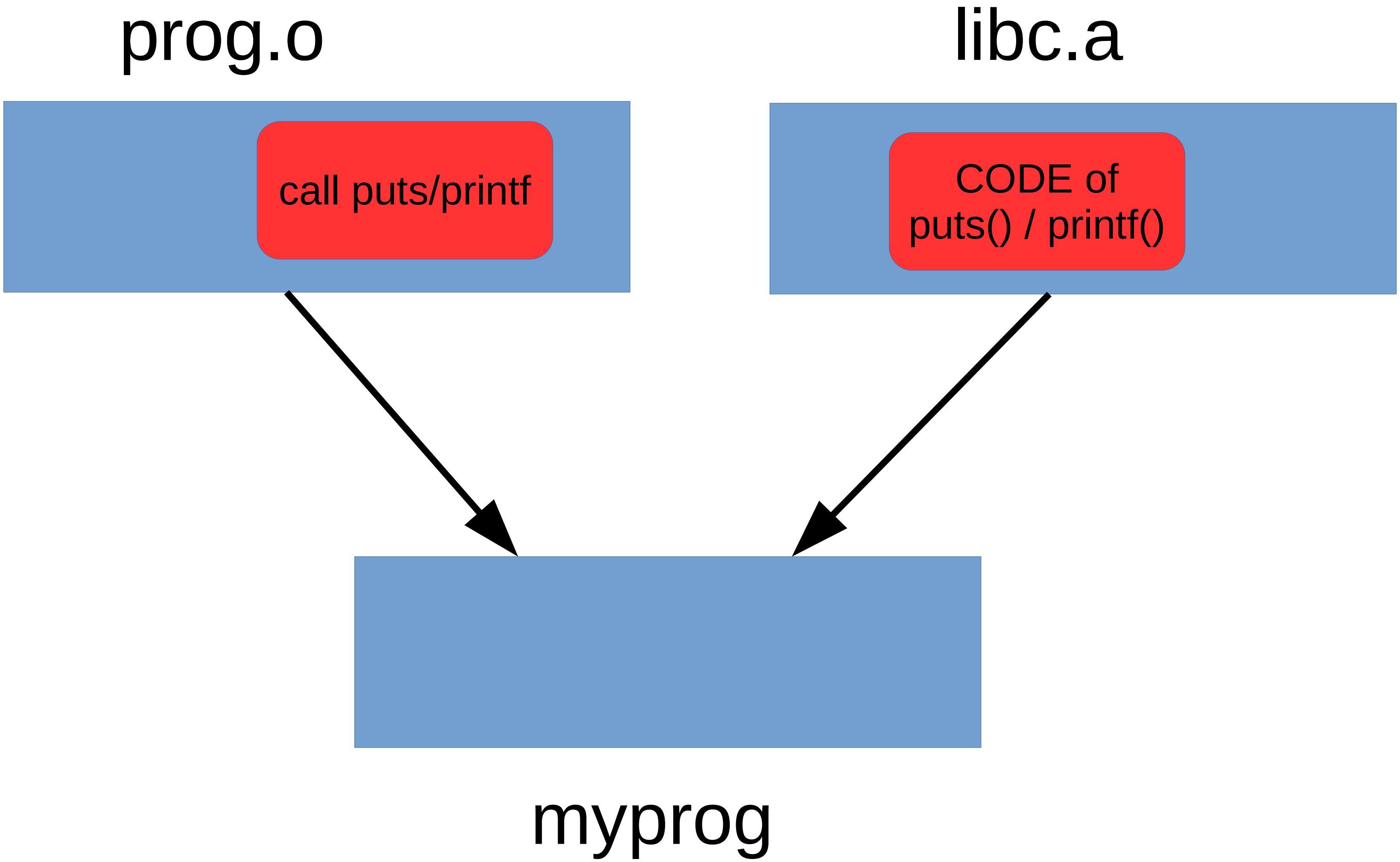
prog.o

call puts/printf

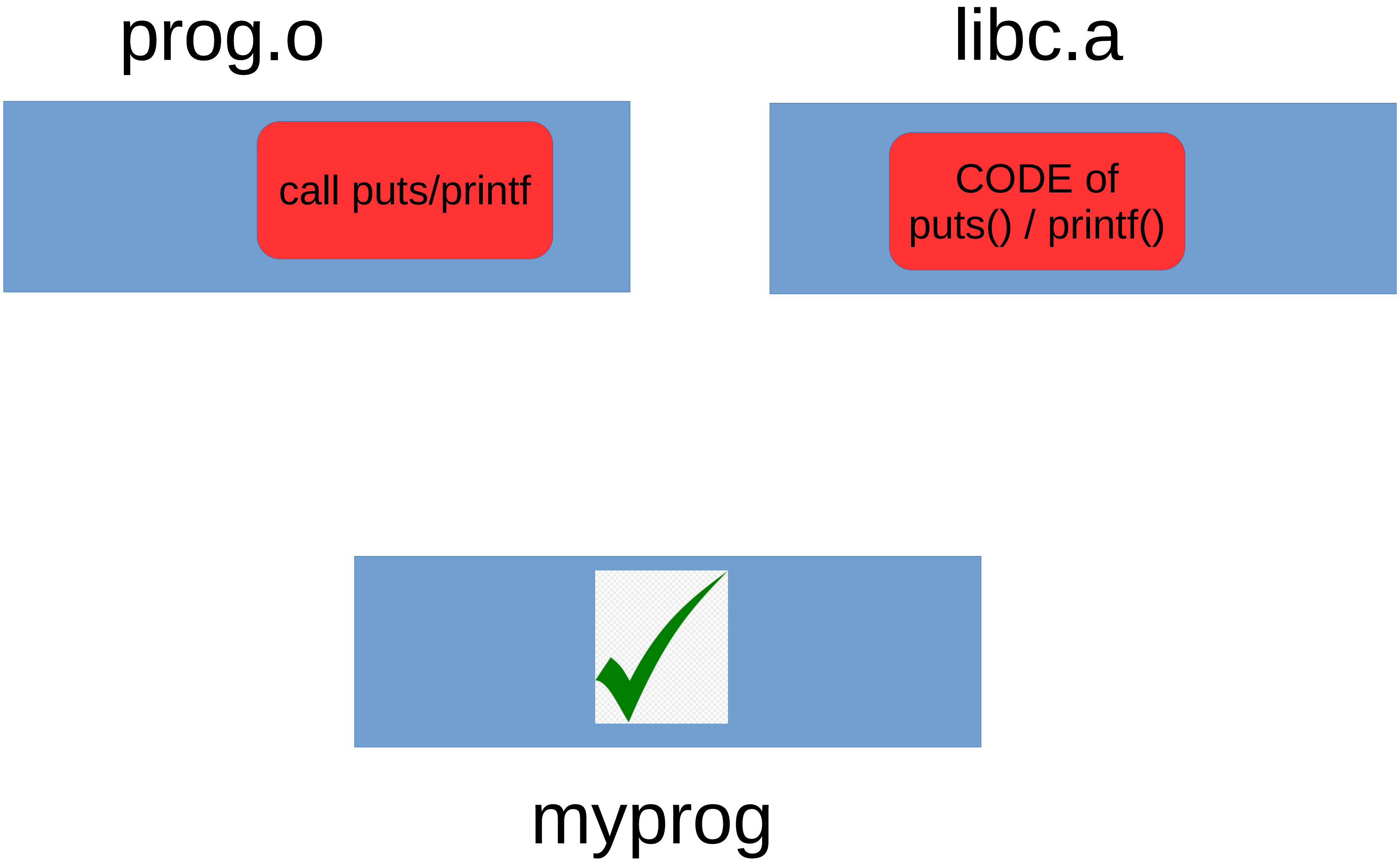
libc.a

CODE of puts() / printf()

Intro to "Linker" solution



Intro to "Linker" solution



prog.o

call puts/printf

libc.a

CODE of puts() / printf()

✓

myprog

Intro to "C Runtime" solution

Setup the activation of main() /* C language */:

Activating main():

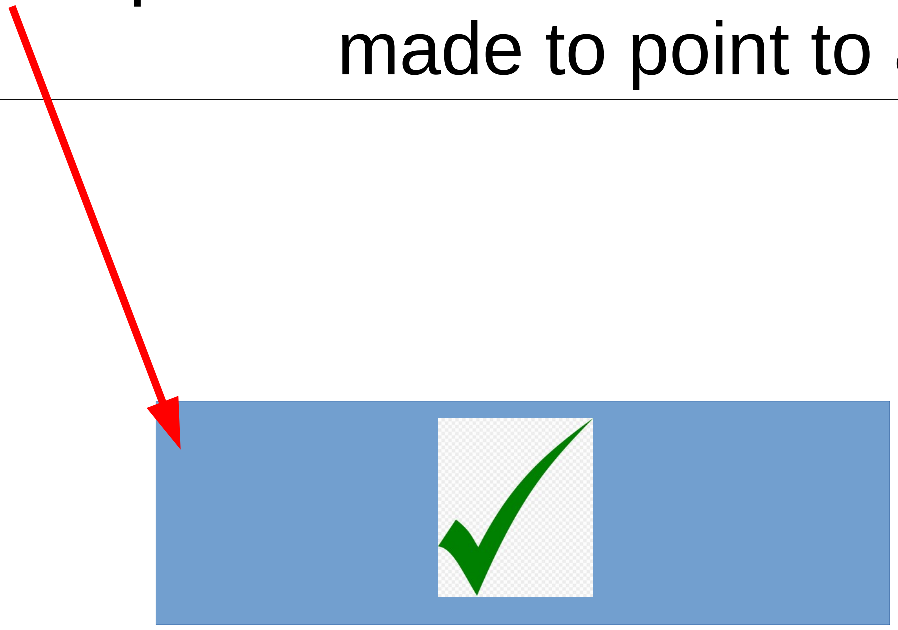
- Stack segment has been allocated
- Setup the “caller” part of the activation, i.e. imagine that the OS is a “C Program” calling a C function called main().
- Parameters are predefined; int argc, char *argv[].

myprog

Intro to "C Runtime" solution

Set the ENTRY point and note it in the EXE:

ENTRY point: The instruction from where the CPU is made to point to and begin execution.



myprog

Executing a Program

```
$ ./myprog
```

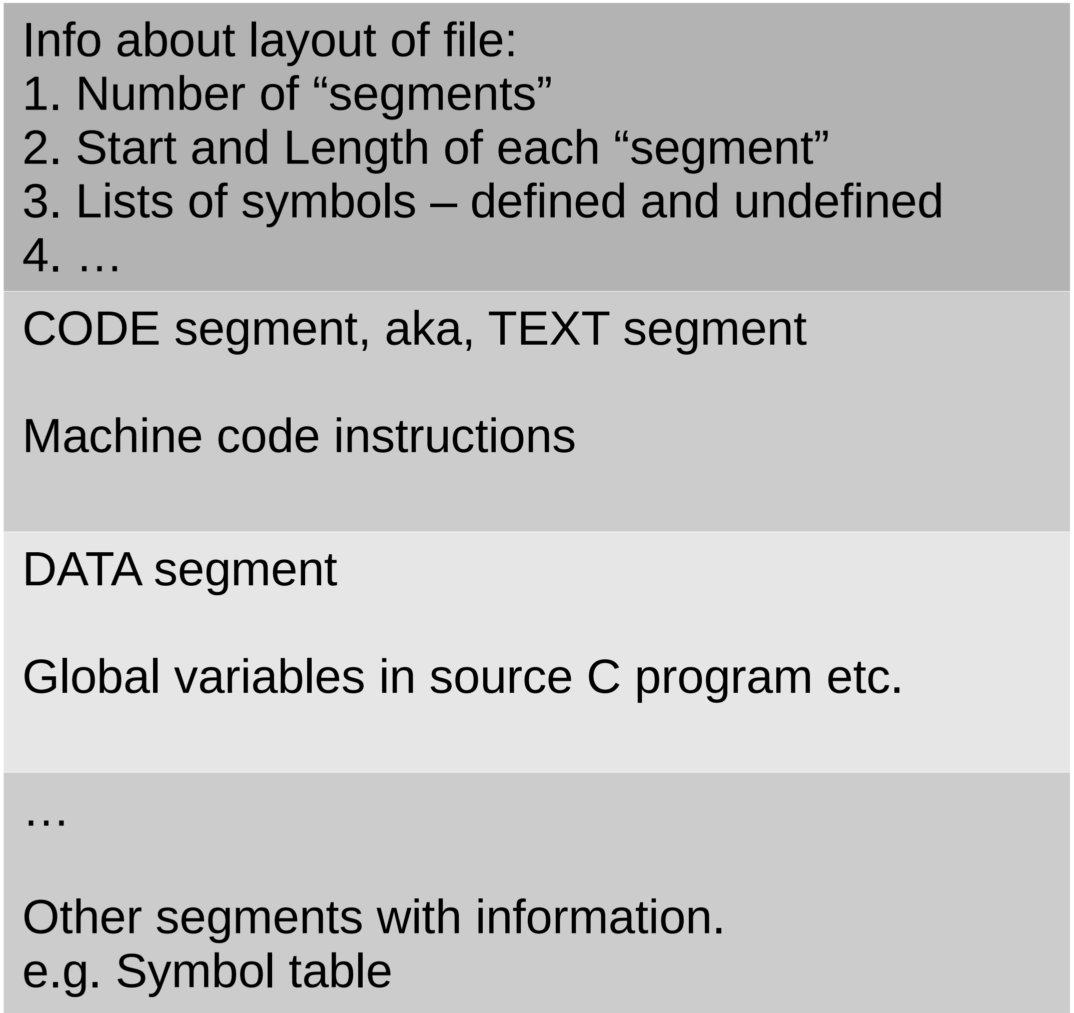
Executing a Program

```
$ ./myprog
```

ToDo for an OS

- Read info from EXE file
- Create PCB Entry
- Allocate primary memory & load
- Schedule

What must an EXE contain?



Info about layout of file:
1. Number of “segments”
2. Start and Length of each “segment”
3. Lists of symbols – defined and undefined
4. …

CODE segment, aka, TEXT segment

Machine code instructions

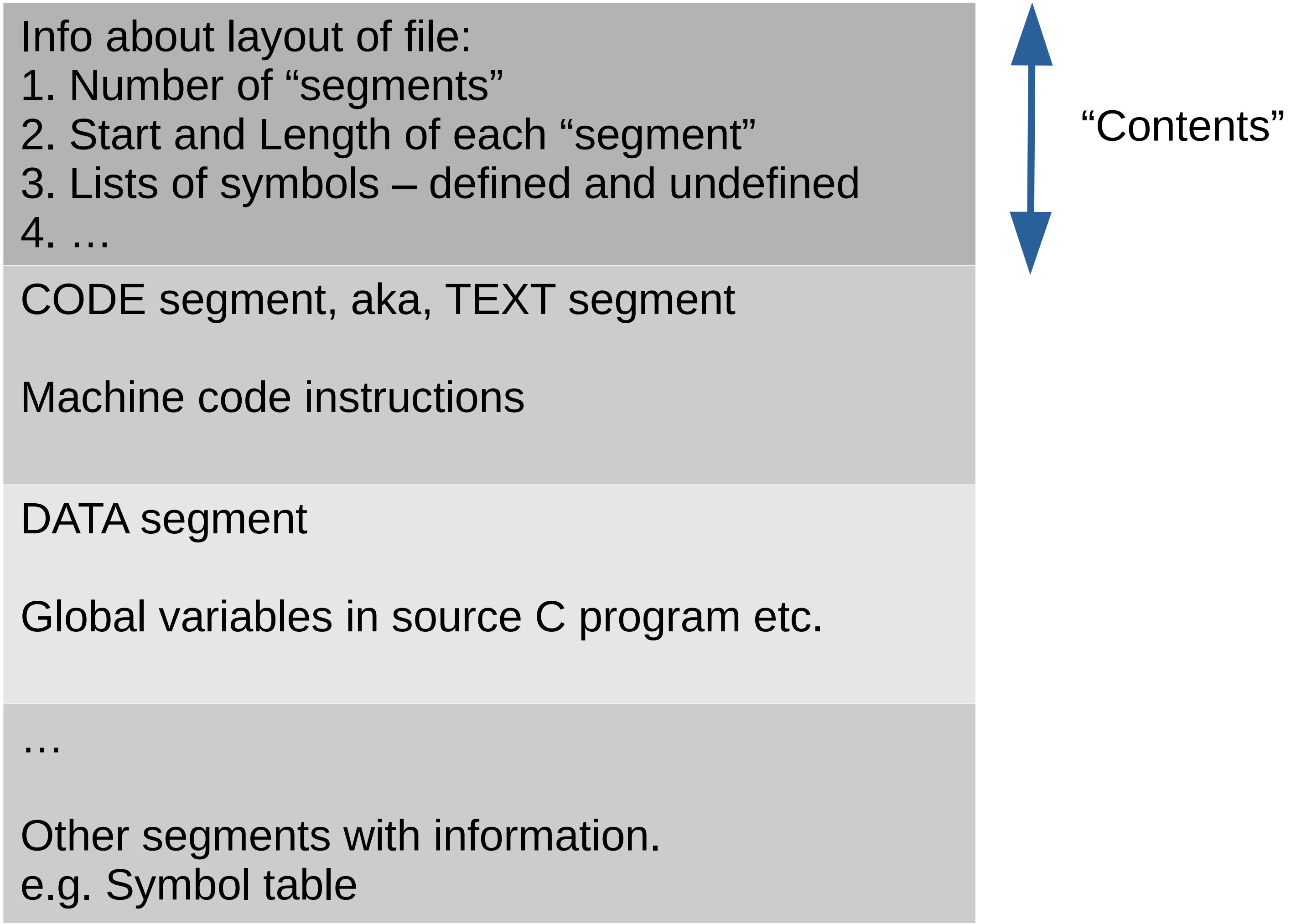
DATA segment

Global variables in source C program etc.

…

Other segments with information.
e.g. Symbol table

What must an EXE contain?



Info about layout of file:
1. Number of “segments”
2. Start and Length of each “segment”
3. Lists of symbols – defined and undefined
4. …

“Contents”

CODE segment, aka, TEXT segment

Machine code instructions

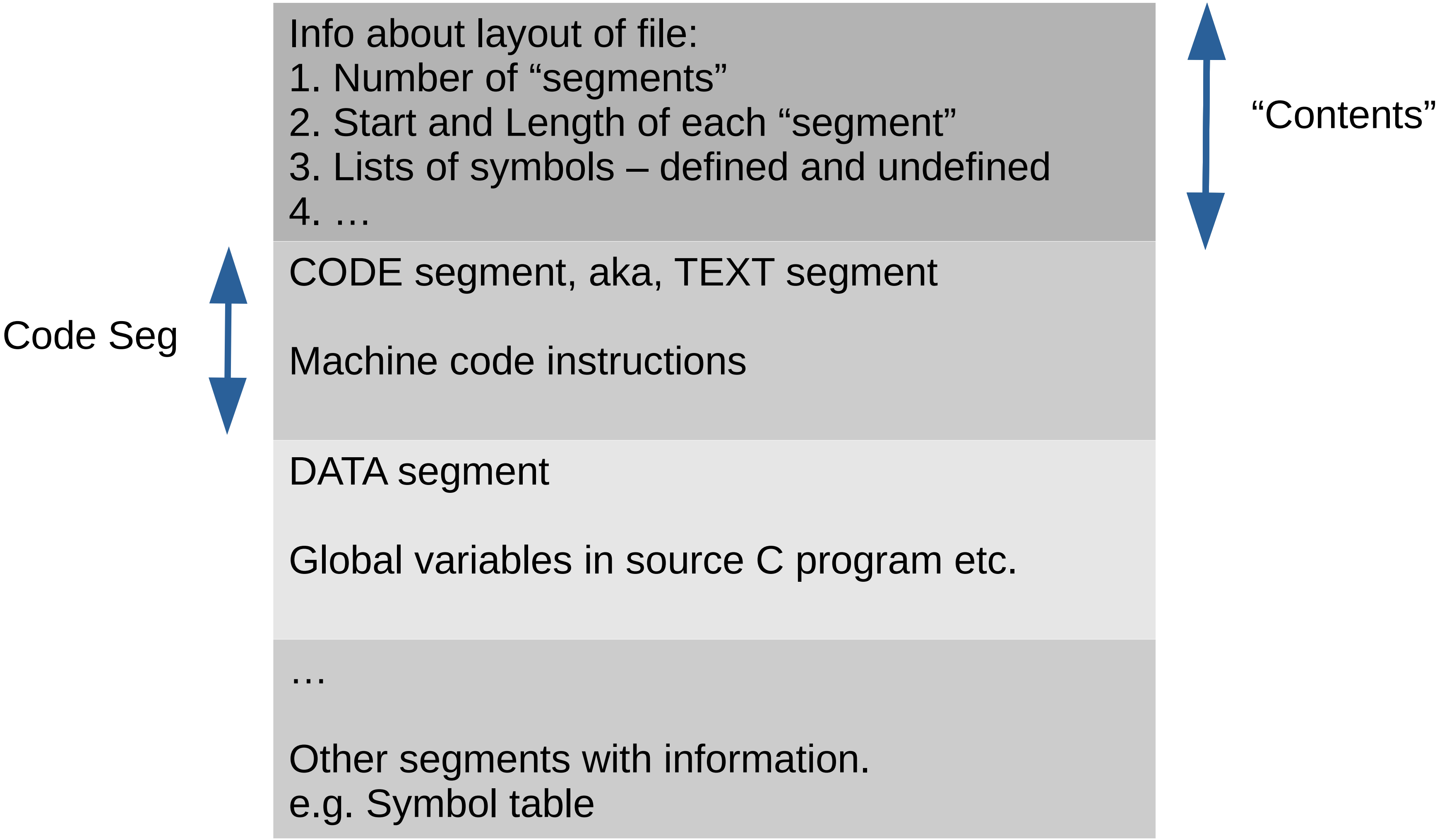
DATA segment

Global variables in source C program etc.

…

Other segments with information.
e.g. Symbol table

What must an EXE contain?



Info about layout of file:
1. Number of “segments”
2. Start and Length of each “segment”
3. Lists of symbols – defined and undefined
4. …

“Contents”

CODE segment, aka, TEXT segment

Machine code instructions

Code Seg

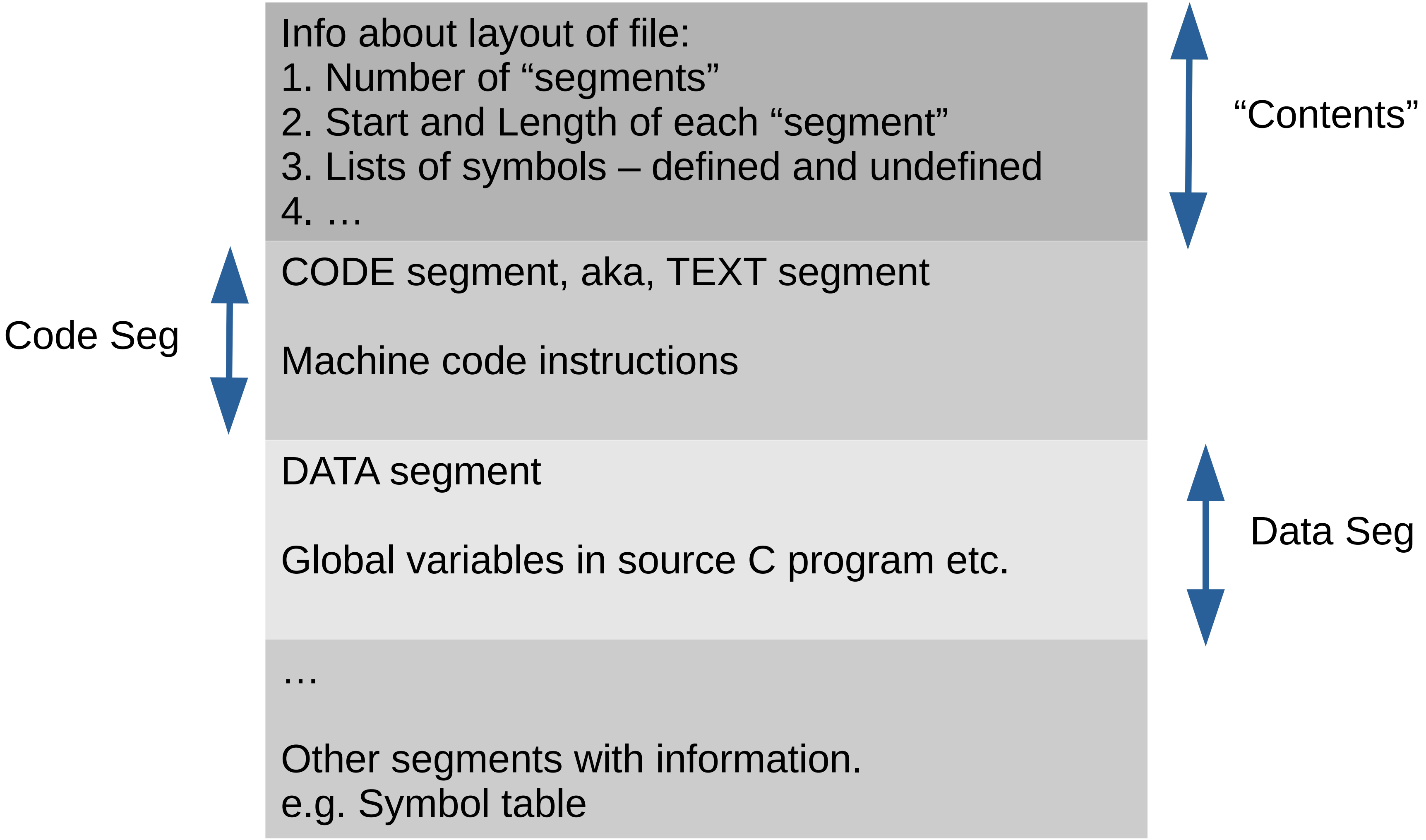
DATA segment

Global variables in source C program etc.

…

Other segments with information.
e.g. Symbol table

What must an EXE contain?



Info about layout of file:
1. Number of “segments”
2. Start and Length of each “segment”
3. Lists of symbols – defined and undefined
4. …

“Contents”

CODE segment, aka, TEXT segment

Machine code instructions

Code Seg

DATA segment

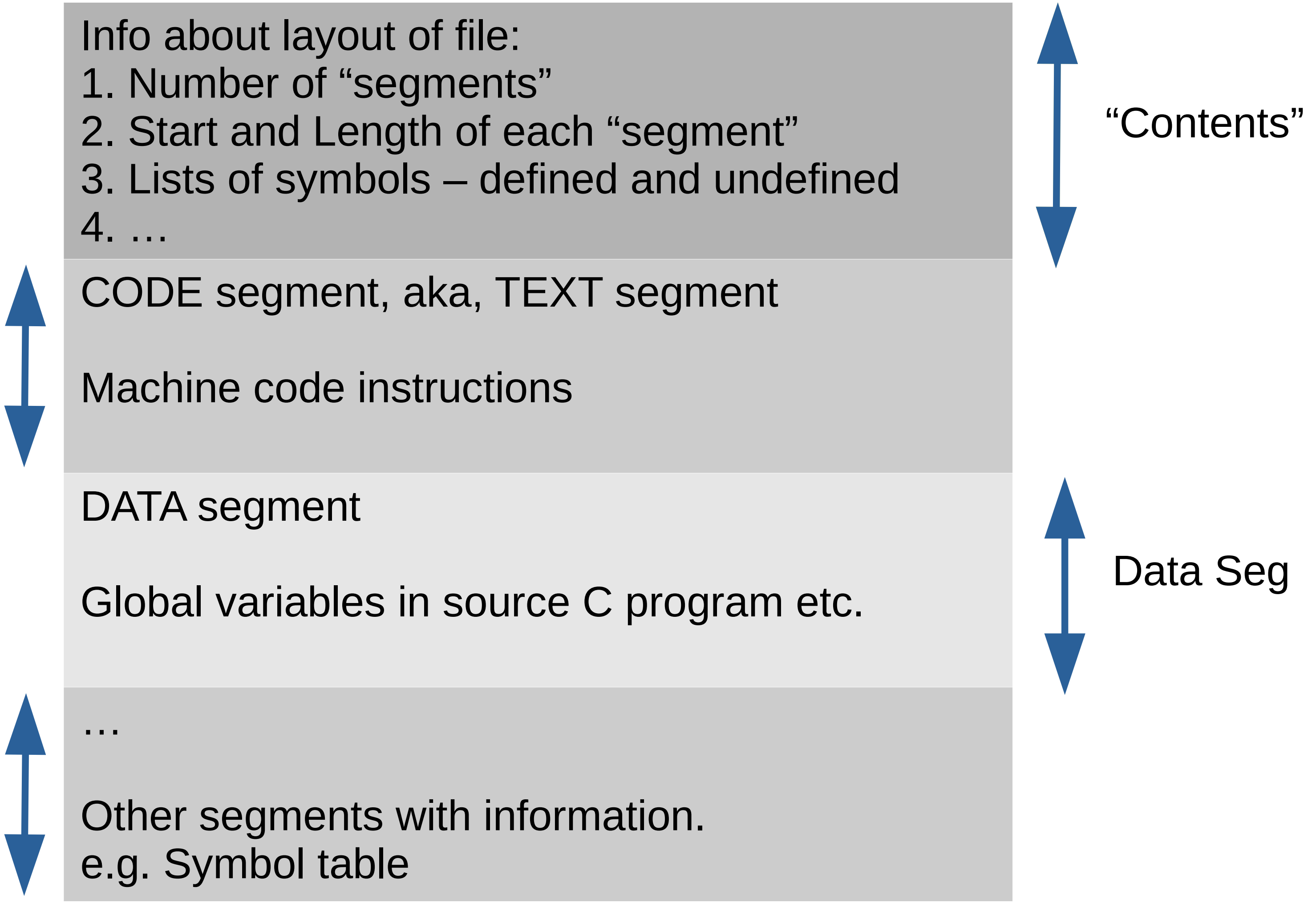
Global variables in source C program etc.

Data Seg

…

Other segments with information.
e.g. Symbol table

What must an EXE contain?



Info about layout of file:
1. Number of “segments”
2. Start and Length of each “segment”
3. Lists of symbols – defined and undefined
4. …

“Contents”

Code Seg

CODE segment, aka, TEXT segment

Machine code instructions

DATA segment

Global variables in source C program etc.

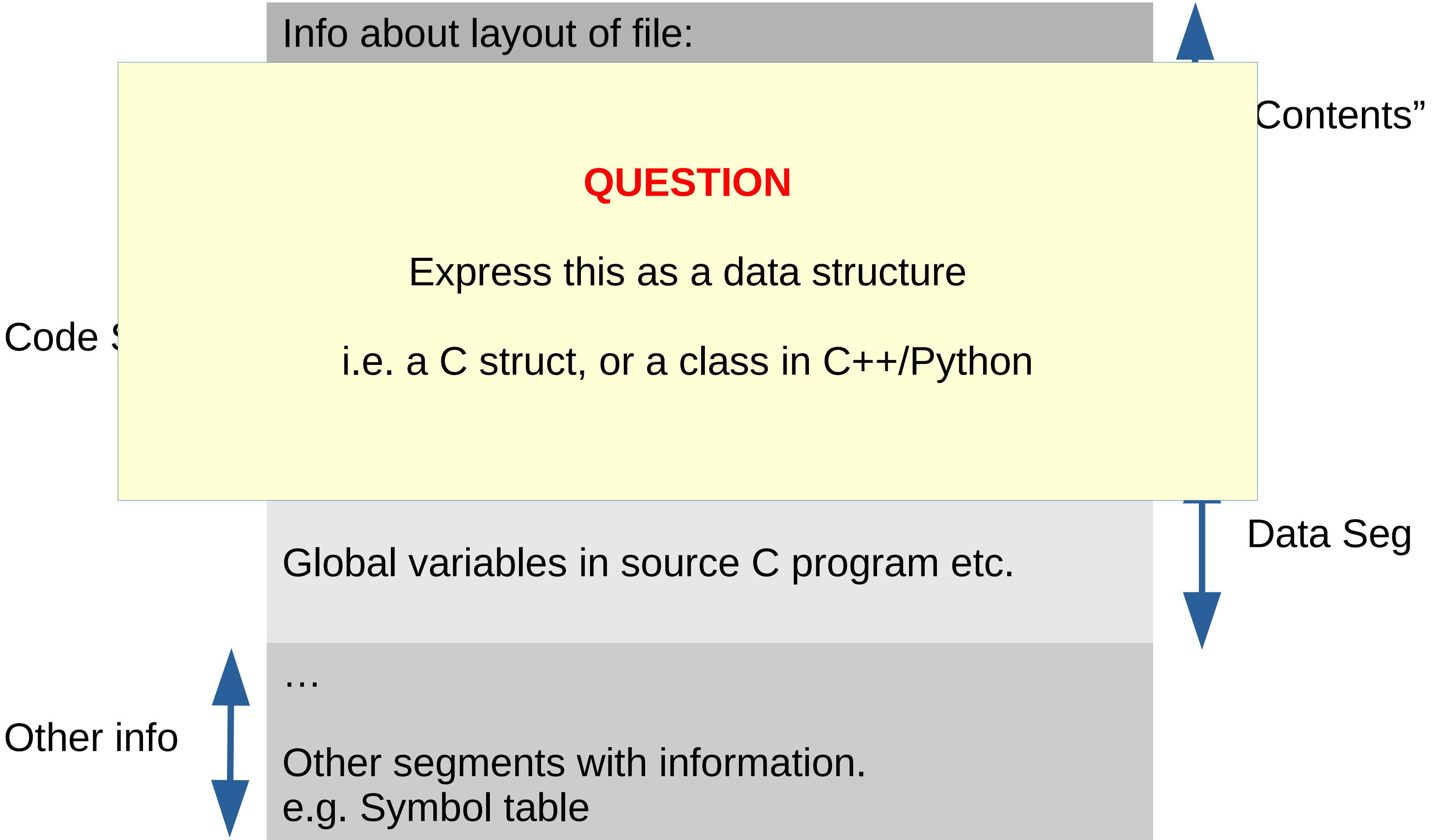
Data Seg

Other info

…

Other segments with information.
e.g. Symbol table

What must an EXE contain?



Info about layout of file:

Contents”

Code S

QUESTION

Express this as a data structure

i.e. a C struct, or a class in C++/Python

Data Seg

Global variables in source C program etc.

…

Other info

Other segments with information.
e.g. Symbol table

Executing a Program: Unix style

```
$ ./myprog
```

Executing a Program: Unix style

User

```
$ ./myprog
```

The Shell

```
{  scanf (cmd line)
   fork() …
   exec(“./myprog”)
   …
}
```

Executing a Program: Unix style

User

```
$ ./myprog
```

The Shell

```
{  scanf (cmd line)
   fork() …
   exec(“./myprog”)
   …
}
```

The Unix Kernel

```
fork () {
/* duplicate shell
    process */ }
exec (…) {
/* overlay fork’d shell
process */ }
```

The fork() Call

- Allocate PM for fork’d shell
- Duplicate PCB entry

The fork() Call

PCB

Shell entry in PCB

Shell process

- Code
- Data
- Stack

- Allocate PM for fork’d shell
- Duplicate PCB entry

The fork() Call

PCB

Shell entry in PCB

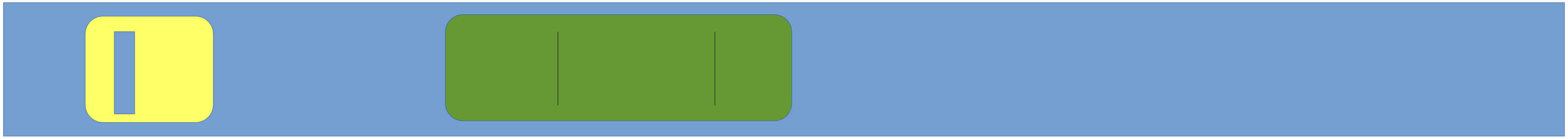
Shell process

Code

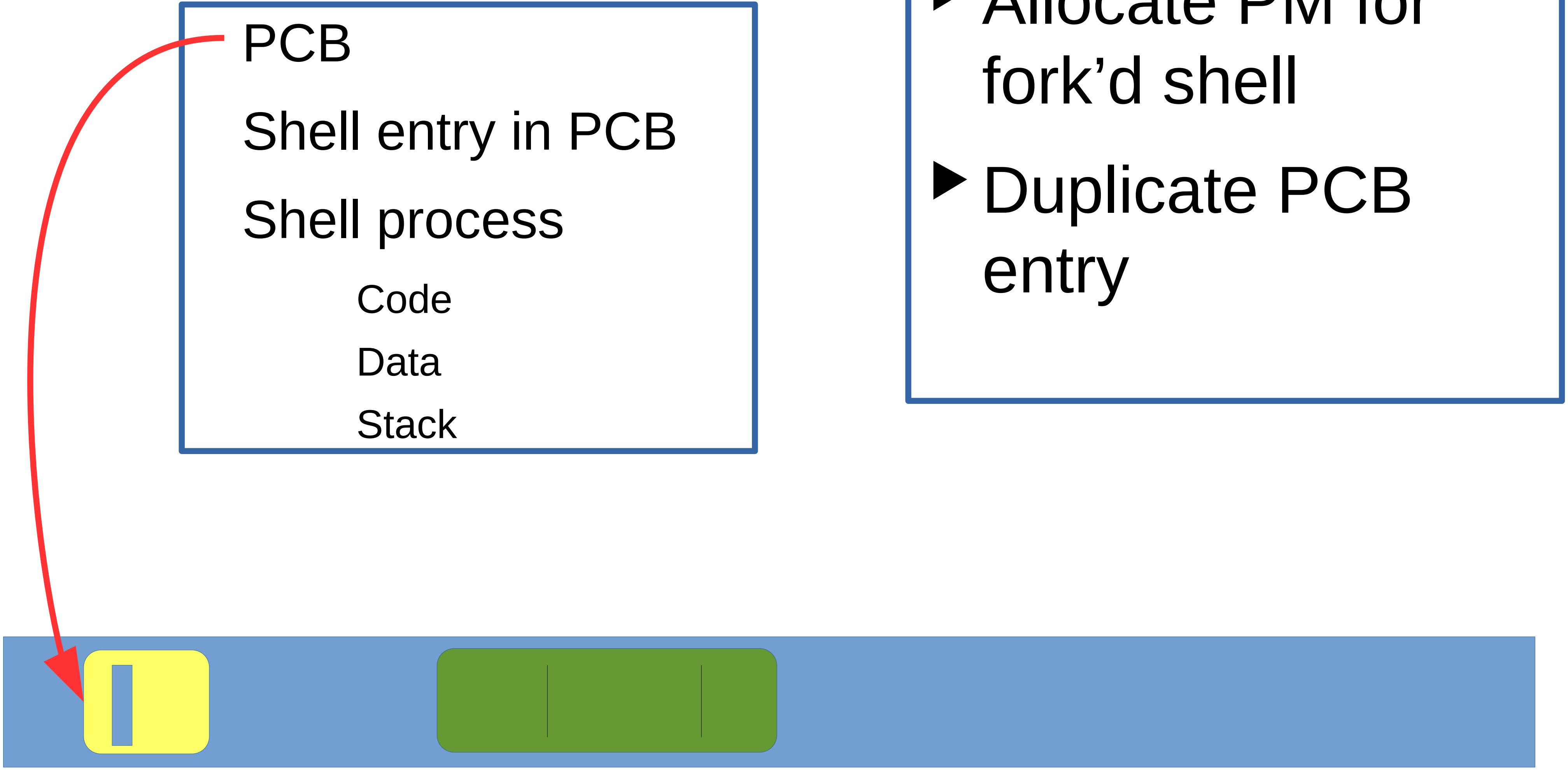
Data

Stack

- Allocate PM for fork'd shell
- Duplicate PCB entry



The fork() Call



PCB

Shell entry in PCB

Shell process

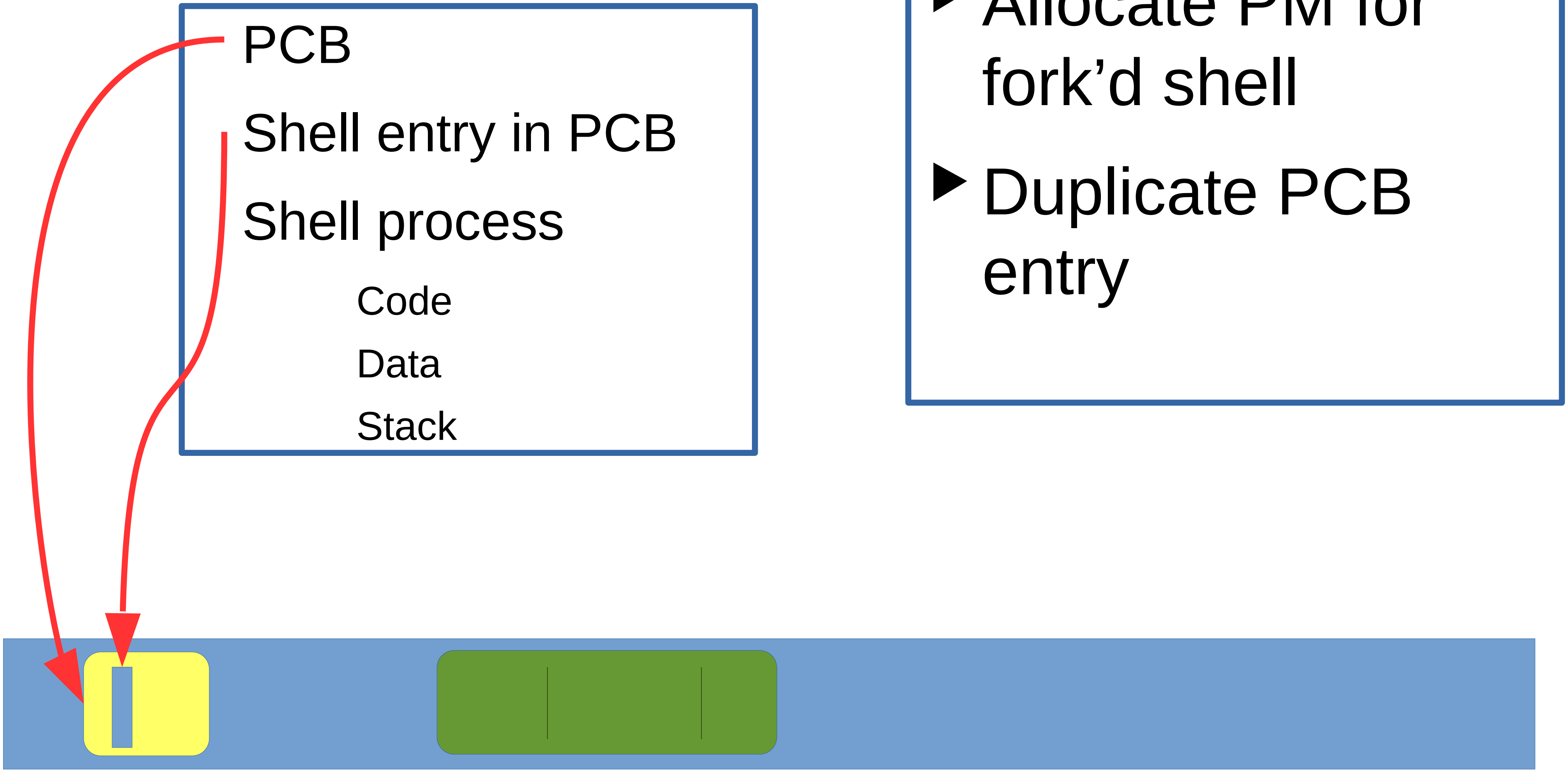
Code

Data

Stack

- Allocate PM for fork’d shell
- Duplicate PCB entry

The fork() Call



PCB

Shell entry in PCB

Shell process

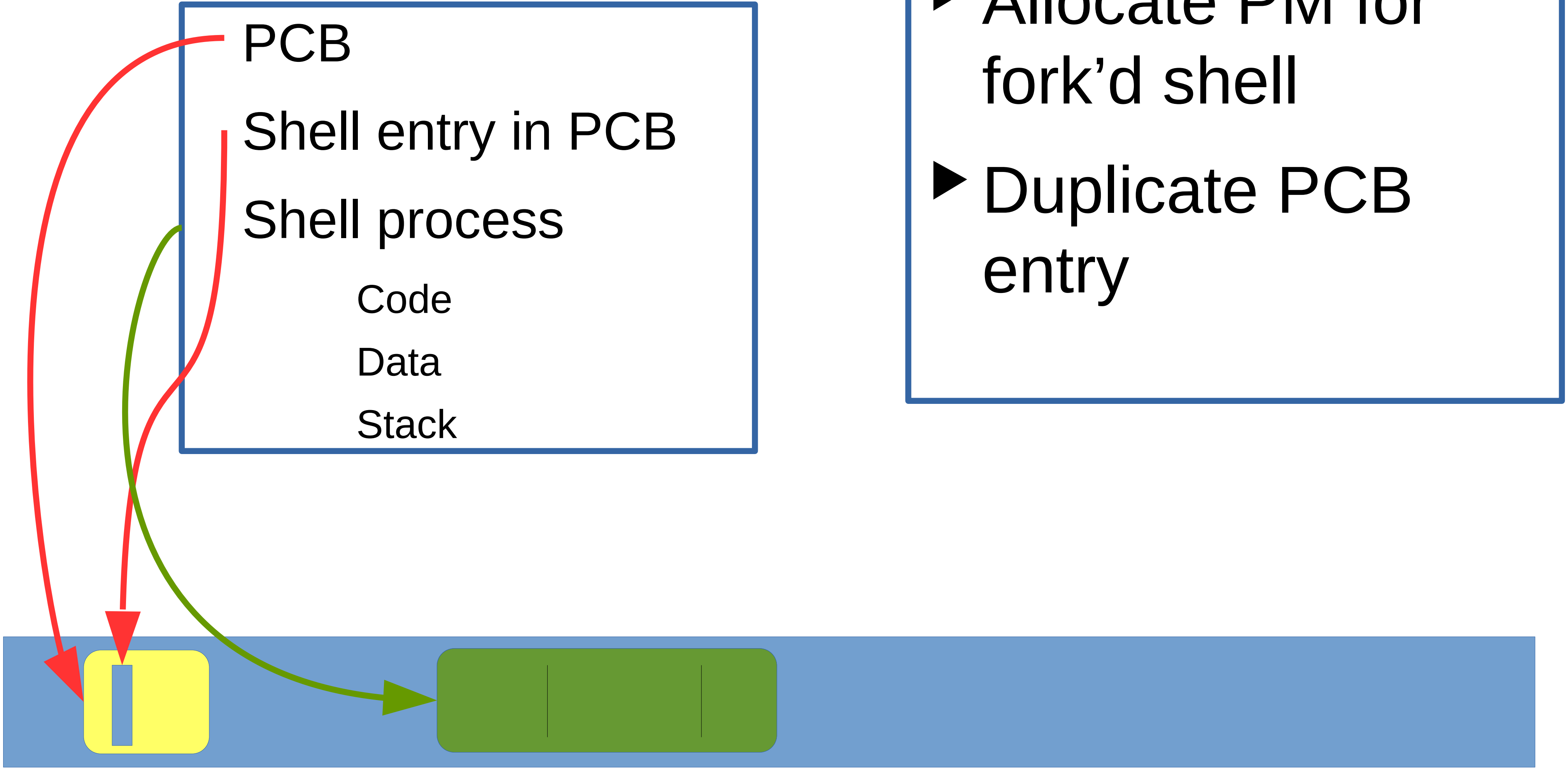
Code

Data

Stack

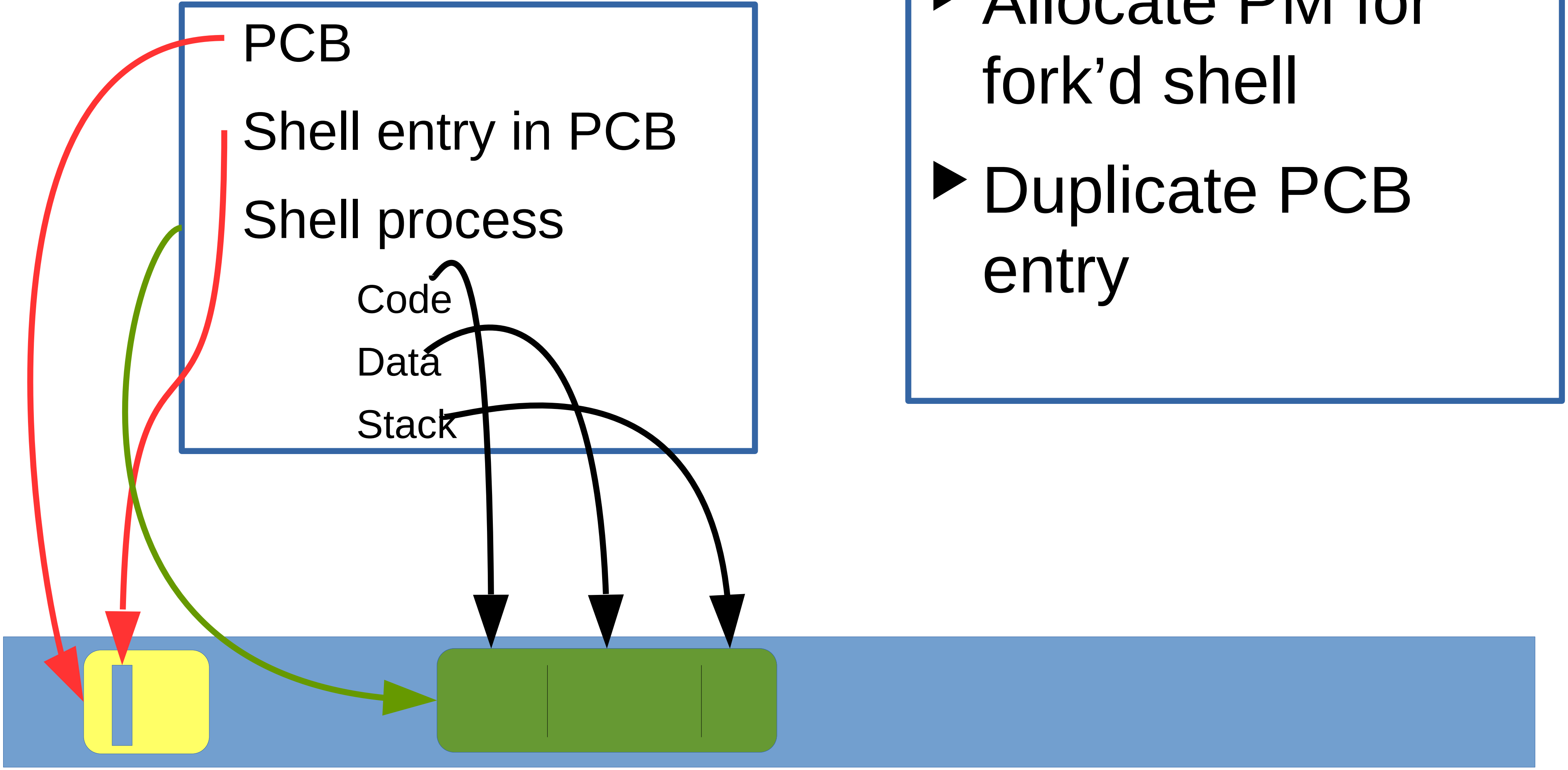
- Allocate PM for fork’d shell
- Duplicate PCB entry

The fork() Call



- Allocate PM for fork’d shell
- Duplicate PCB entry

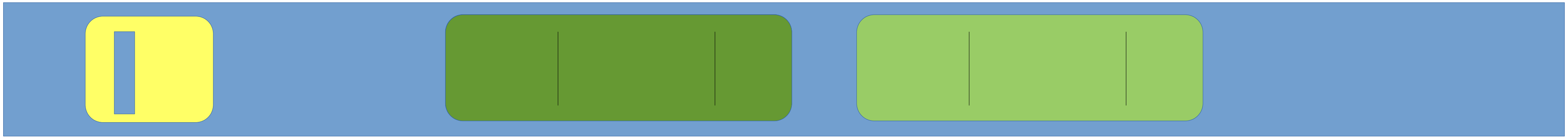
The fork() Call



- Allocate PM for fork’d shell
- Duplicate PCB entry

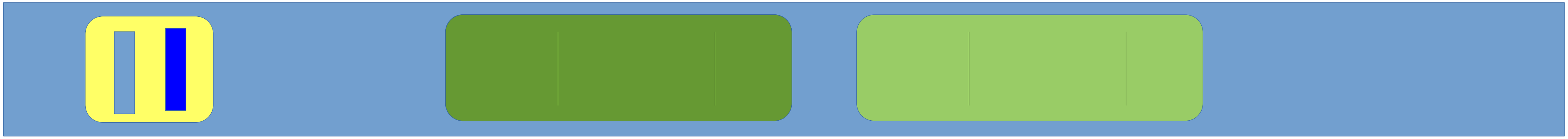
The fork() Call

- Allocate PM for fork’d shell
- Duplicate PCB entry



The fork() Call

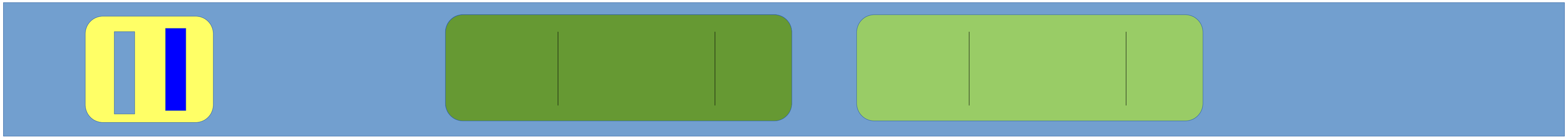
- Allocate PM for fork’d shell
- Duplicate PCB entry



The fork() Call

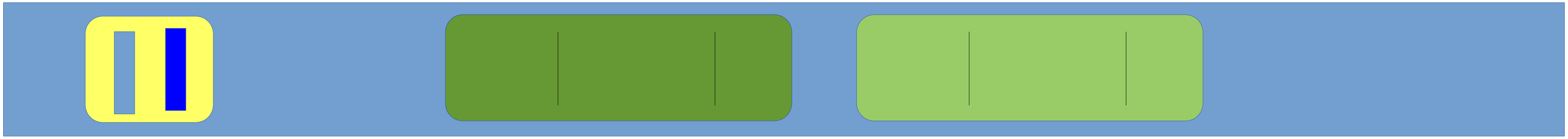
Scheduler is programmed to use entries in the PCB!

- Allocate PM for fork’d shell
- Duplicate PCB entry



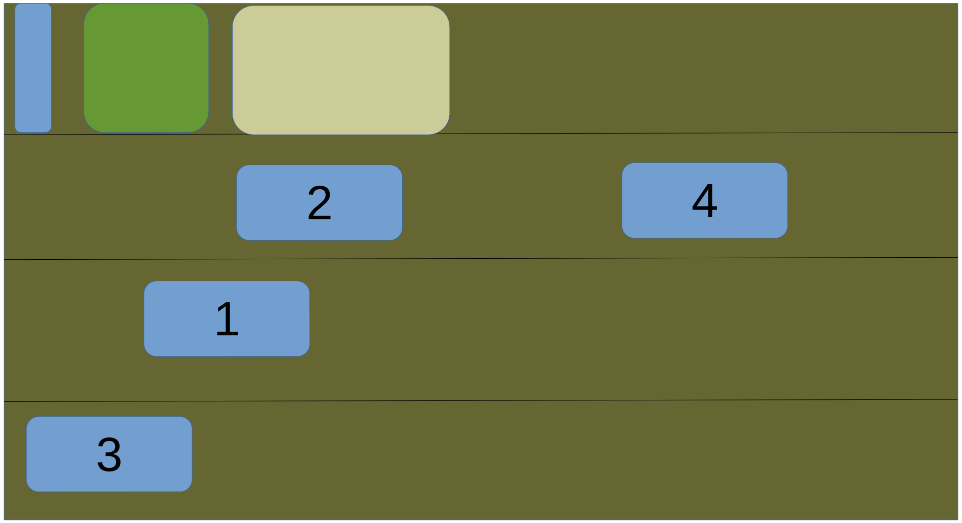
The exec() Call

- Read program EXE
- Resize segments
- Load from EXE
- Reset PCB data

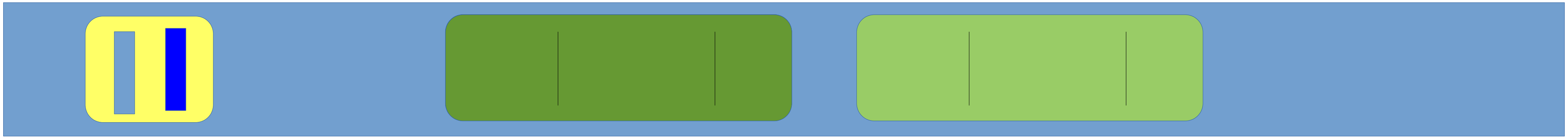


The exec() Call

/home/amv/bin/myprog

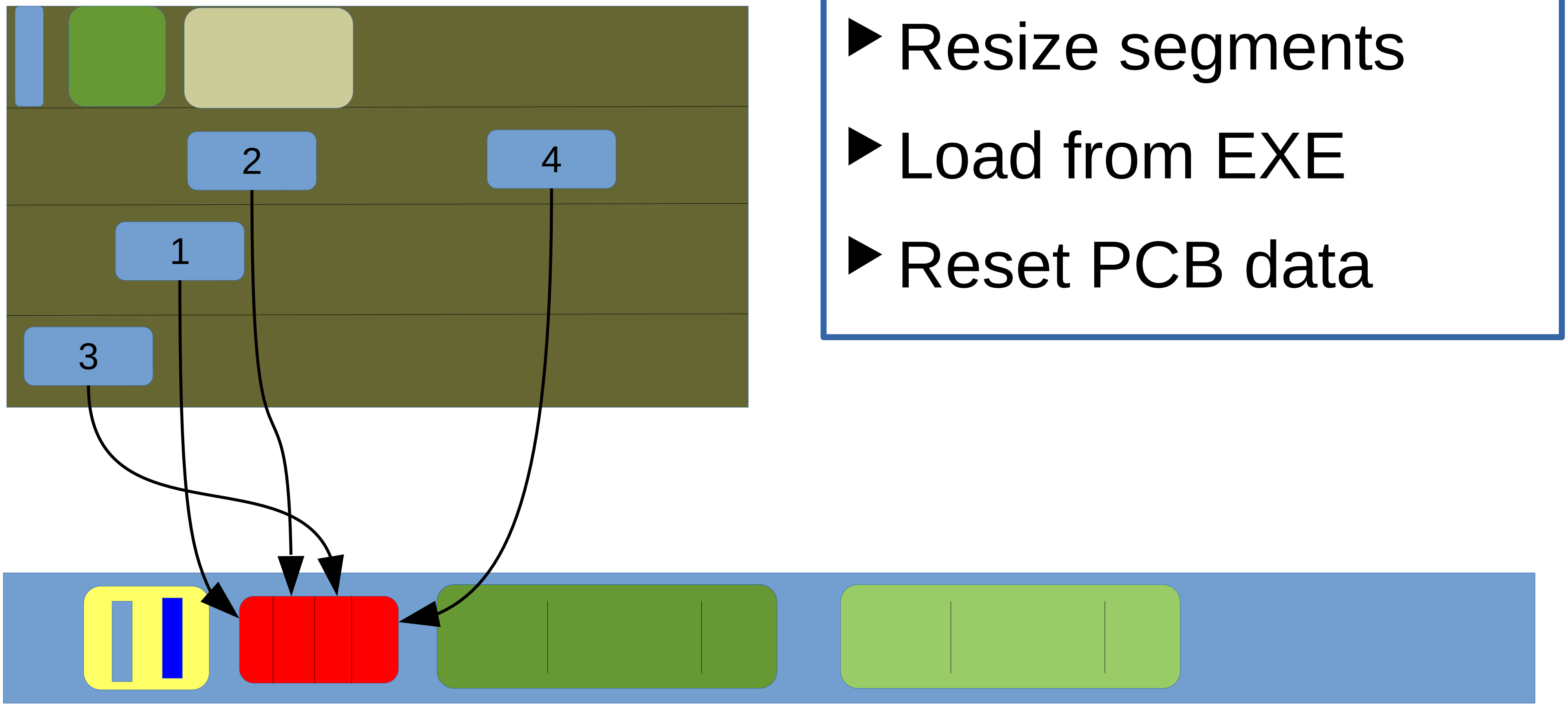


- Read program ELF
- Resize segments
- Load from ELF
- Reset PCB data



The exec() Call

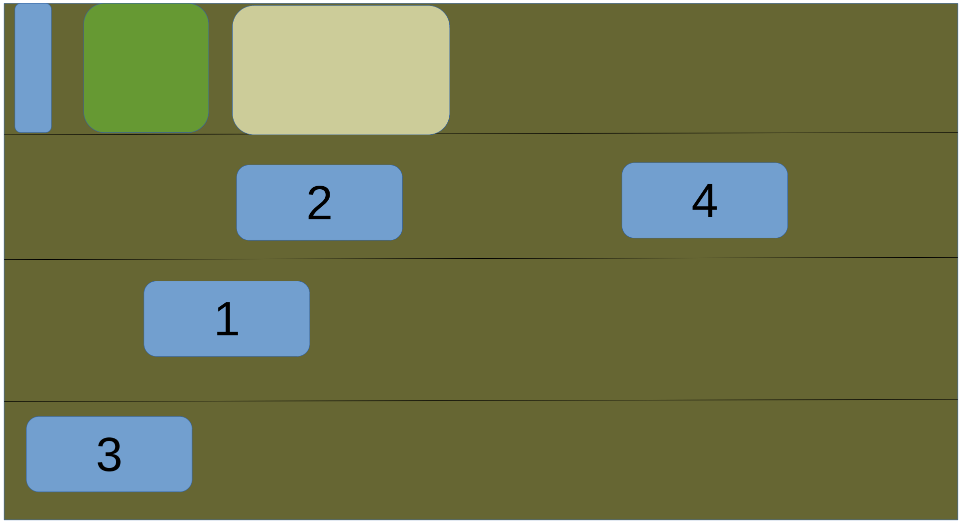
/home/amv/bin/myprog



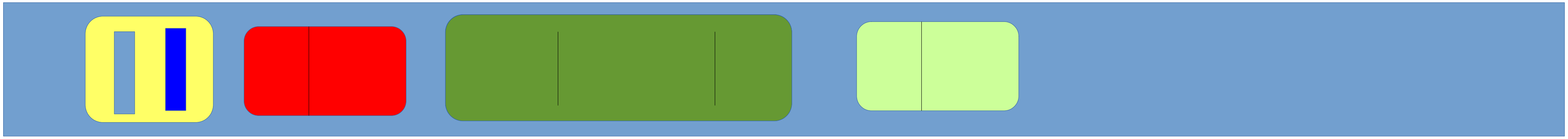
- Read program EXE
- Resize segments
- Load from EXE
- Reset PCB data

The exec() Call

/home/amv/bin/myprog

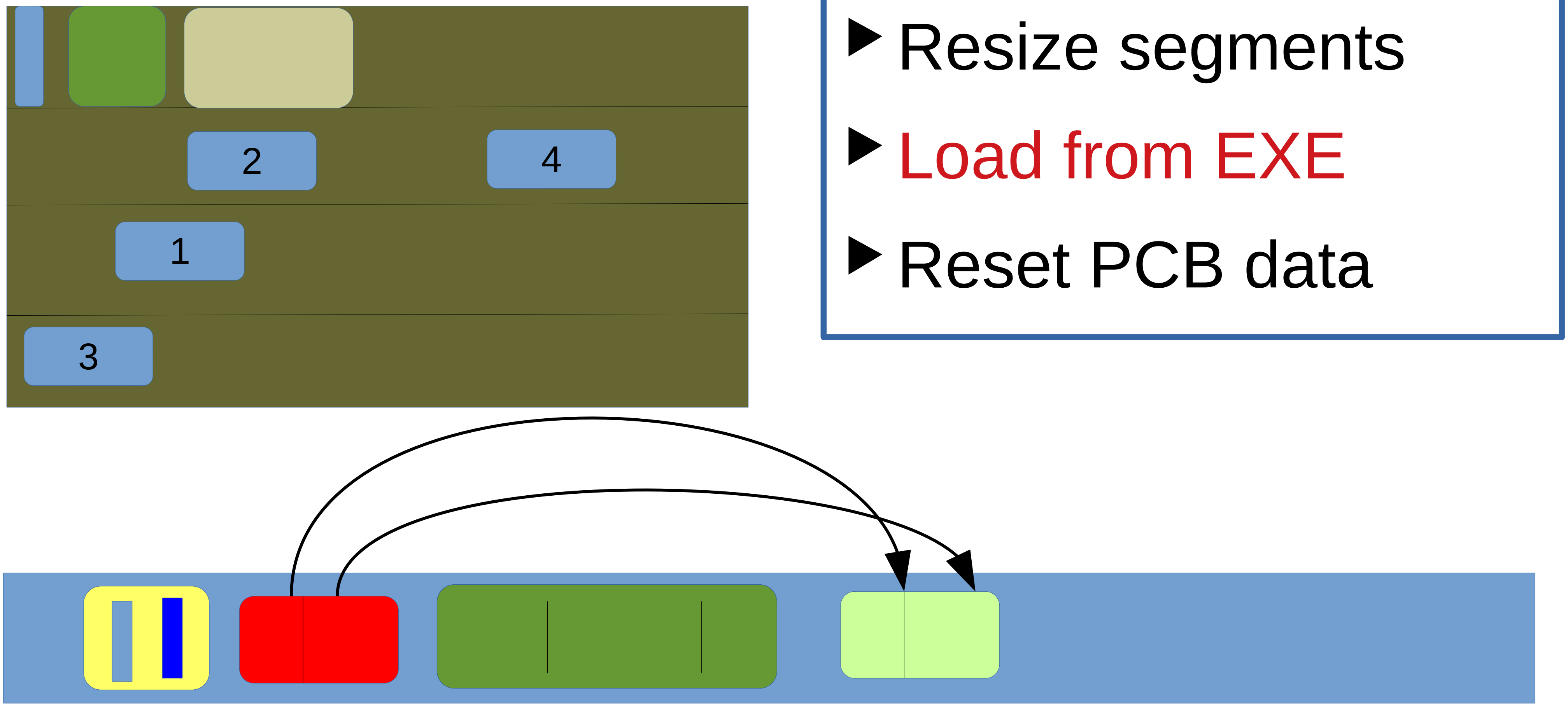


- Read program EXE
- Resize segments
- Load from EXE
- Reset PCB data



The exec() Call

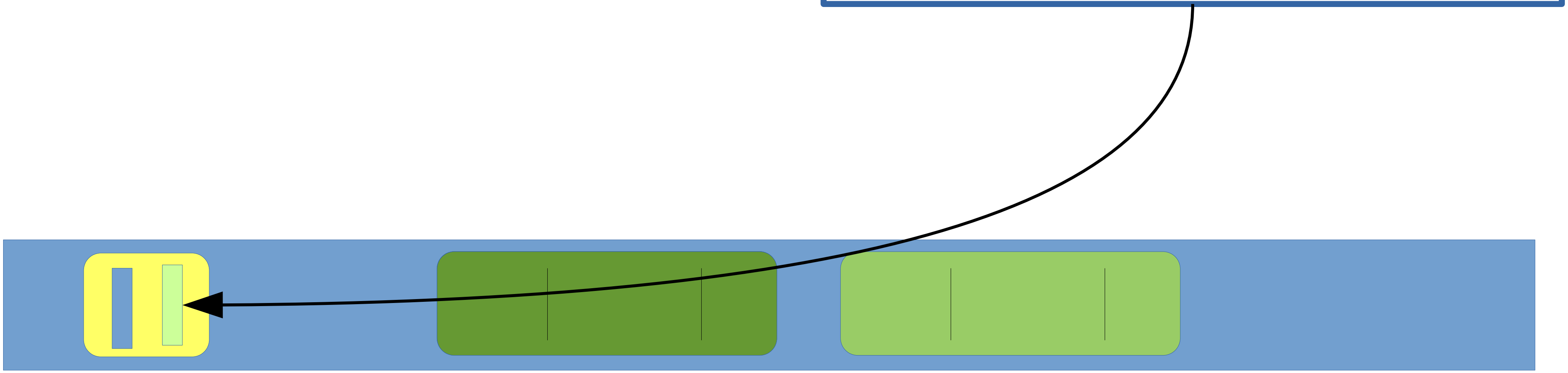
/home/amv/bin/myprog



- Read program EXE
- Resize segments
- Load from EXE
- Reset PCB data

The exec() Call

- Read program EXE
- Resize segments
- Load from EXE
- Reset PCB data



Creating Executable Files

Key idea:

Construct according to the needs of the underlying machine!

Suggested Lab Exercises

“Information gathering” questions

- What is the executable file format on your system?
- Expand its abbreviation, if any.
- View your executable file format as a layouts data structure. Find out the details of its layout, and express it as either C structures or C++ classes.

“Discover/Explore/Study” questions

- How is the ELF format able to deal with both OBJECT and EXECUTABLE files?
- Write a C program that prints the header information out of an ELF file which can be either an OBJ or an EXE.

Session Summary

- EXE Production
  - H/W is extremely detailed and precise
  - Code, Data, ... everything must be in binary
  - Gradually transform source to EXE
- EXE Execution
  - Not everything can be known while producing the EXE
  - Programming language design introduces some blanks to fill
  - More transformation steps are needed
  - The compilation system and the operating system meet each other here
- Each step in transformation
  - introduces more details and precision
  - provides an abstraction level suited for its purpose

Runtime System: Bridge between EXE production and EXE Execution

Conceptual View

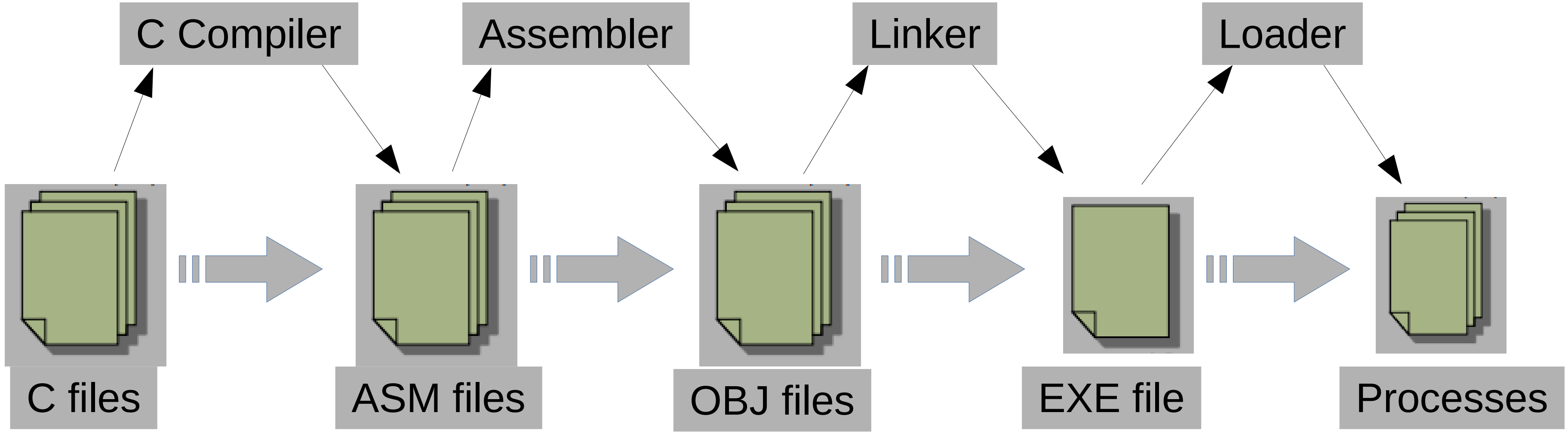
Conceptual Ideas for Runtimes

- Thinking Models
  - The C language model
  - The Assembly Language model
  - ...
  - Final Model: The Process?
- Binding
  - The Idea
  - “Gradual Transformations” as Binding refinement over time!
- Key Idea of Compilers/Interpreters + OS

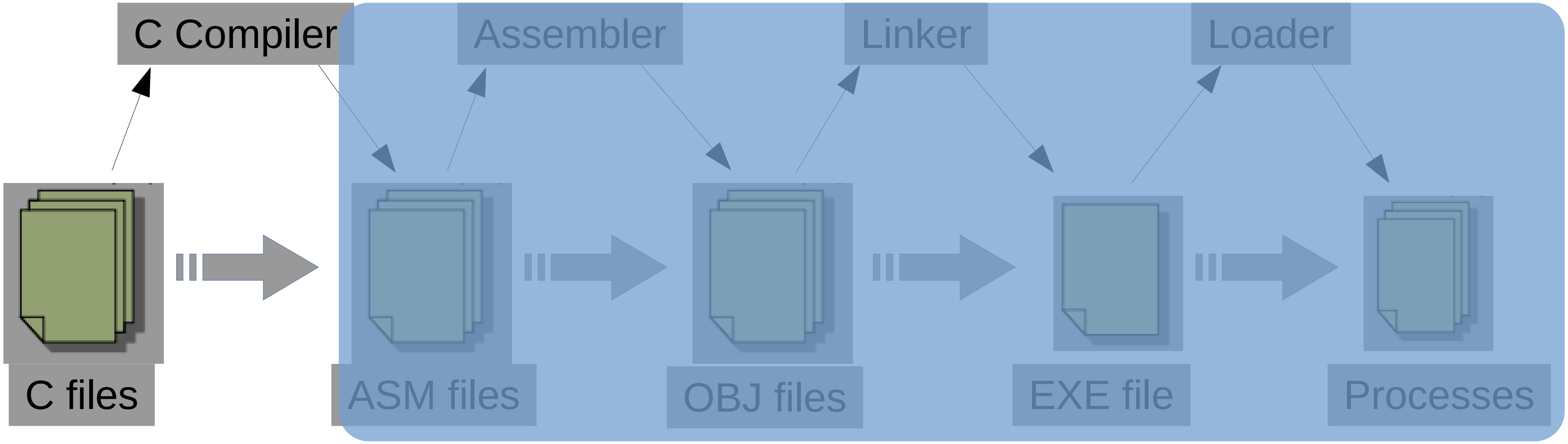
HLL: C Programming Model

- Variables: Abstraction of memory locations
- Functions: Abstraction of computation, i.e. “black boxes” that transform parameters passed to the value returned – Write-Once-Use-Many
- Function calls: LIFO order, i.e. caller is suspended until callee returns.
- Program = Set of functions that start from the function called “main()”.
- ...

Thinking Models



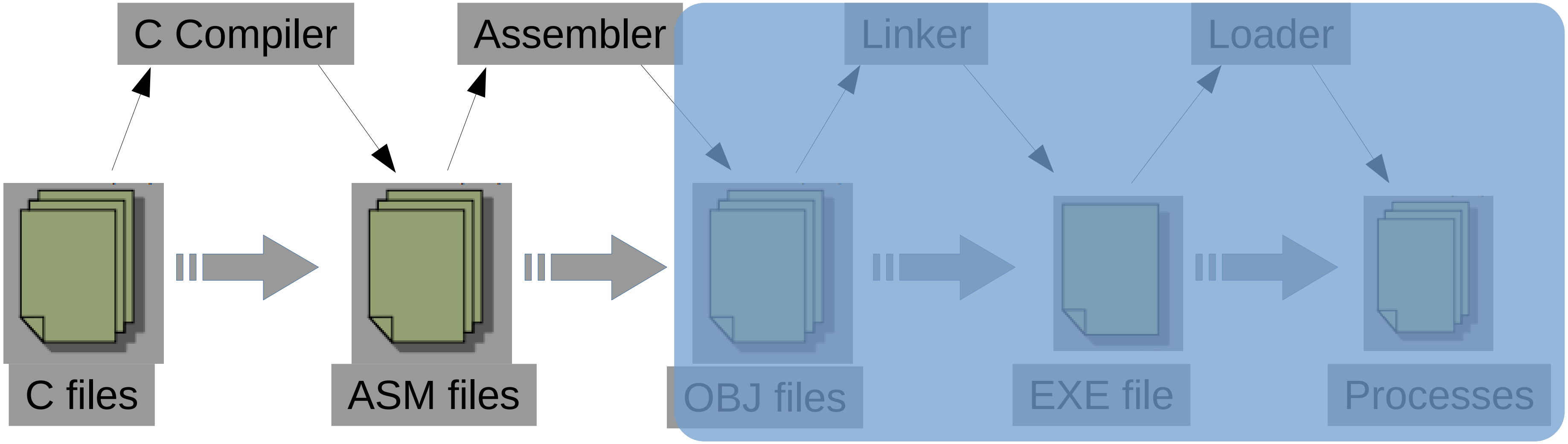
Thinking Models



Thinking Model for Source Programs:

- Logical, i.e. human level, operations, logical data types
- Entities in a program are logically labeled using symbols aka identifiers
- No concept of an “address”

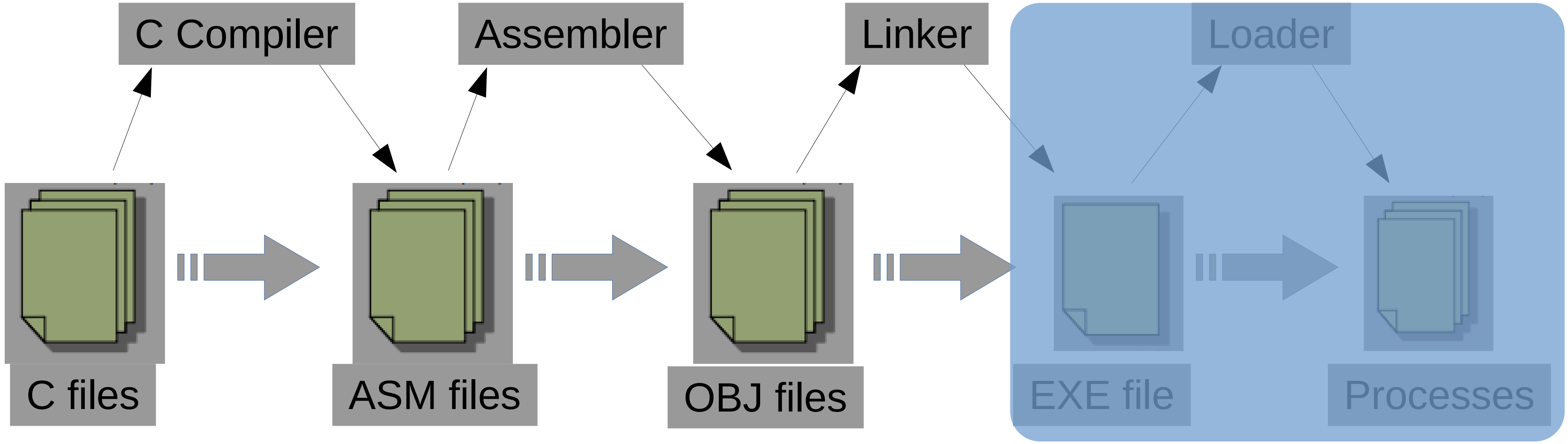
Thinking Models



Thinking Model for ASM Programs:

- Explicit mnemonics for operations, machine data types
- Locations in a program are logically labeled using symbols aka identifiers aka labels
- Concept of an "address" – finite set of positive integers that denote locations

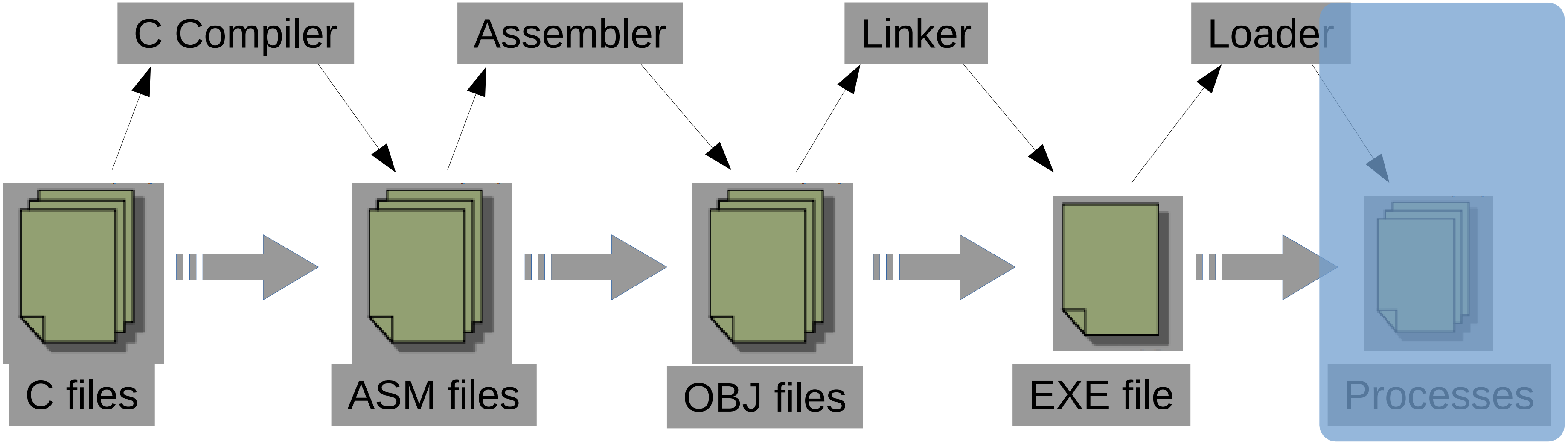
Thinking Models



Thinking Model for OBJ files:

- Explicit binary for operations, machine data types
- Locations, i.e. addresses are in binary
- Symbols whose address cannot be computed are tabulated
- Tables of symbols with addresses and symbols without addresses

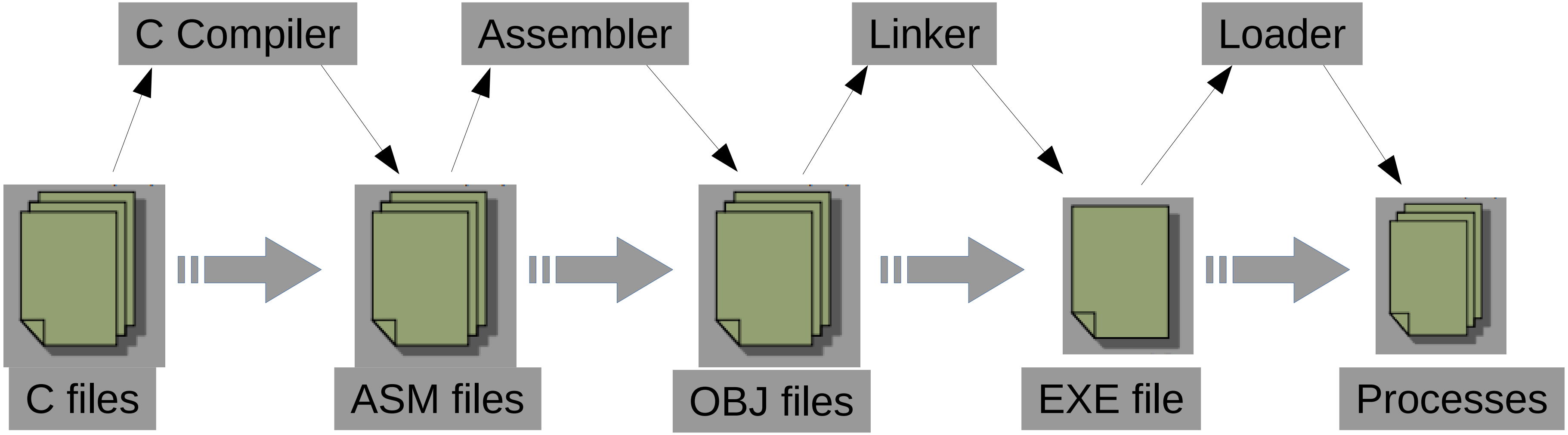
Thinking Models



Thinking Model for EXE files:

- Explicit binary for operations, machine data types
- Locations, i.e. addresses are in binary
- Symbols whose address cannot be computed are tabulated
- Tables of symbols with addresses and symbols without addresses

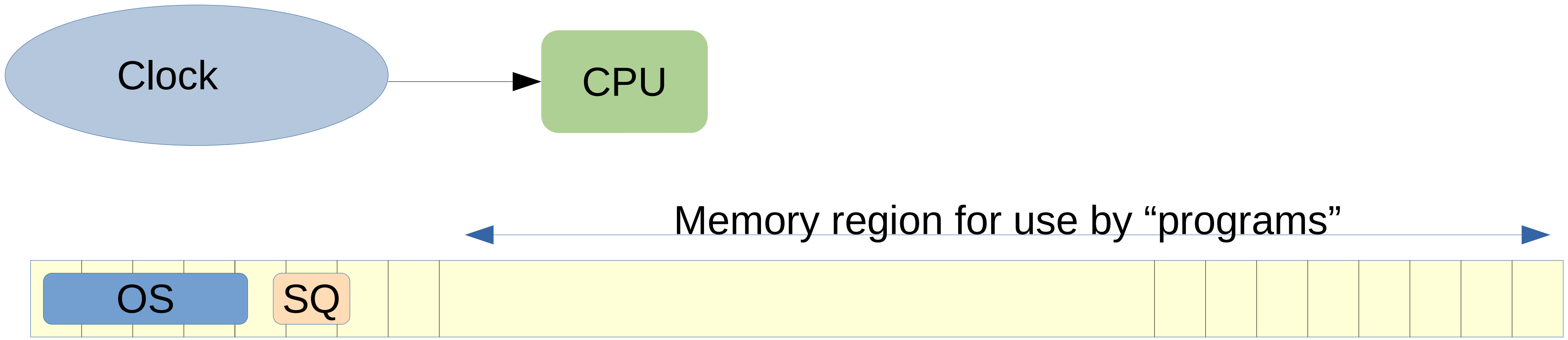
Thinking Models

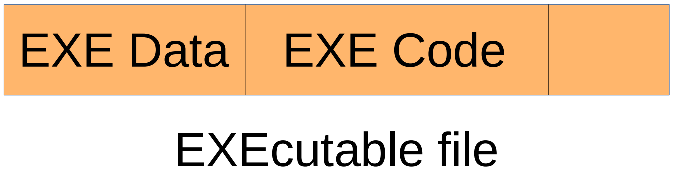


Thinking Model for PROCESSES:

- Use OS memory management algorithms to allocate memory for loading
- Many processes can be generated from one program
- Addresses in program code need to be updated when the memory is allocated

Loading a Program

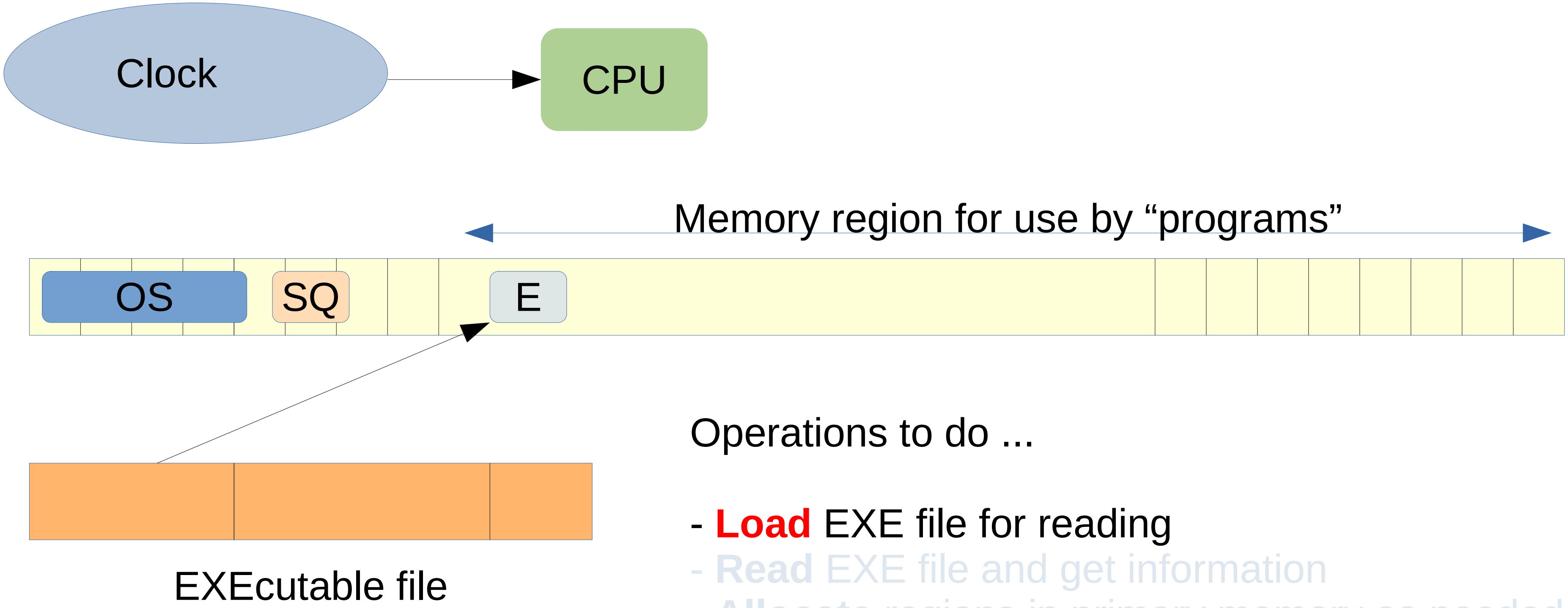




Operations to do ...

- **Load** EXE file for reading
- **Read** EXE file and get information
- **Allocate** regions in primary memory as needed
- **Initialise** memory regions as noted in EXE
- **Record** the first instruction as noted in the EXE
- **Collect** all this information into a data structure
- **Give** this data structure to the scheduler **S**

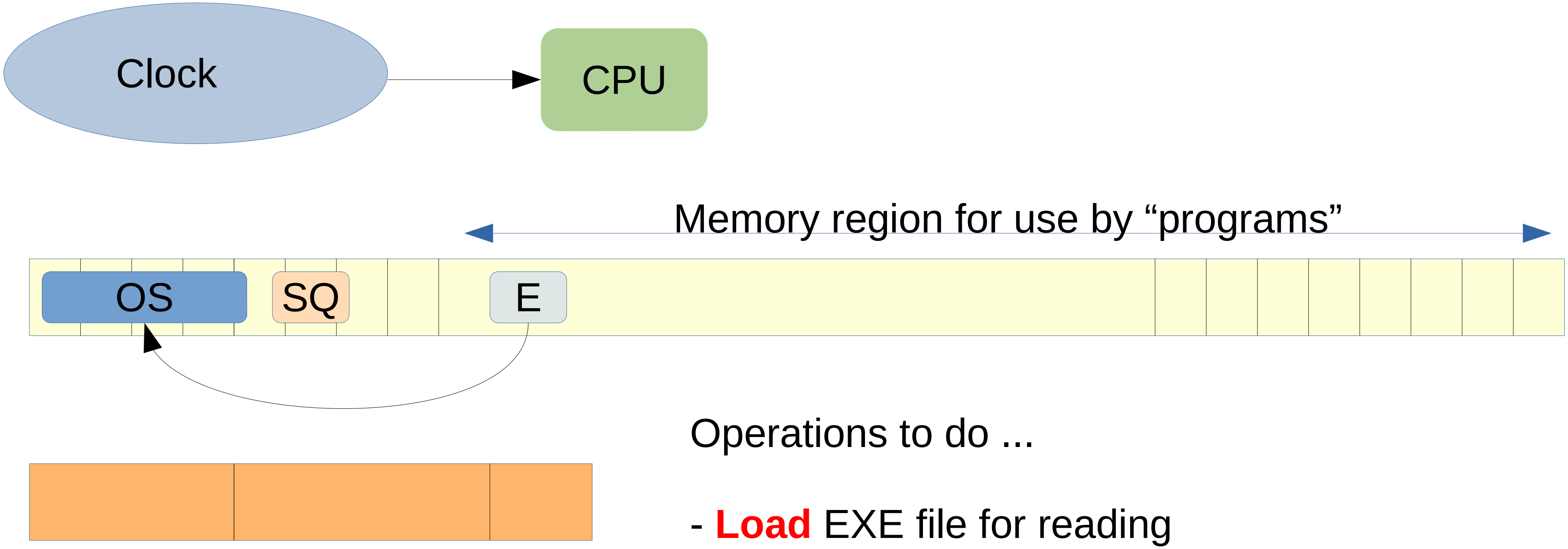
Loading a Program



Operations to do ...

- **Load** EXE file for reading
- **Read** EXE file and get information
- **Allocate** regions in primary memory as needed
- **Initialise** memory regions as noted in EXE
- **Record** the first instruction as noted in the EXE
- **Collect** all this information into a data structure
- **Give** this data structure to the scheduler **S**

Loading a Program

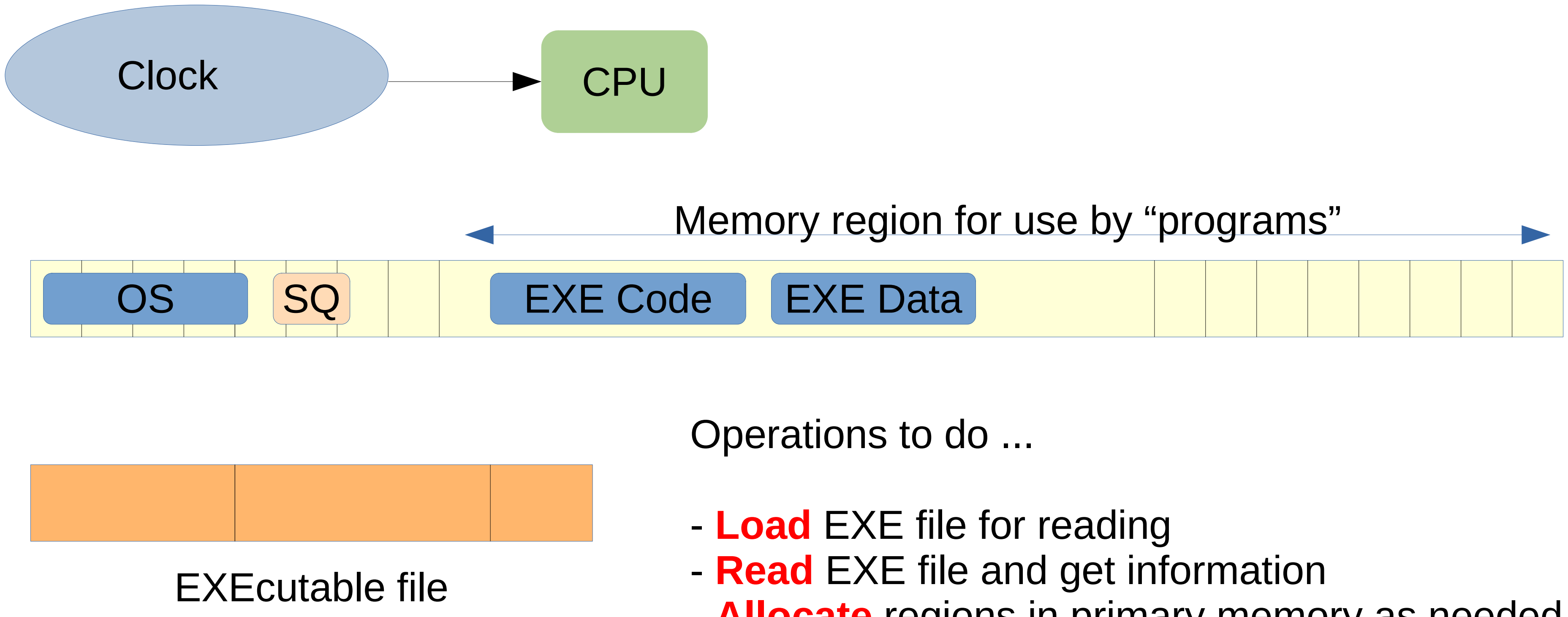


EXEcutable file

Operations to do ...

- **Load** EXE file for reading
- **Read** EXE file and get information
- **Allocate** regions in primary memory as needed
- **Initialise** memory regions as noted in EXE
- **Record** the first instruction as noted in the EXE
- **Collect** all this information into a data structure
- **Give** this data structure to the scheduler **S**

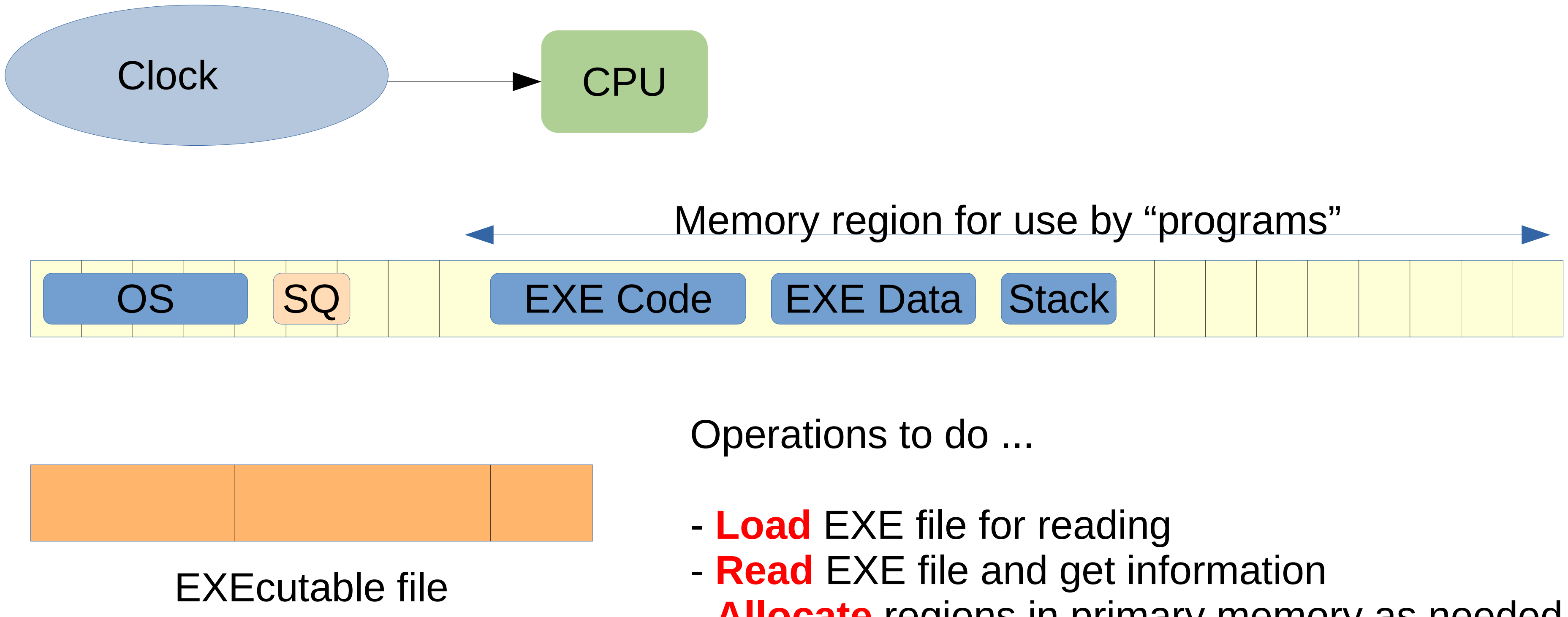
Loading a Program



Operations to do ...

- **Load** EXE file for reading
- **Read** EXE file and get information
- **Allocate** regions in primary memory as needed
- **Initialise** memory regions as noted in EXE
- **Record** the first instruction as noted in the EXE
- **Collect** all this information into a data structure
- **Give** this data structure to the scheduler **S**

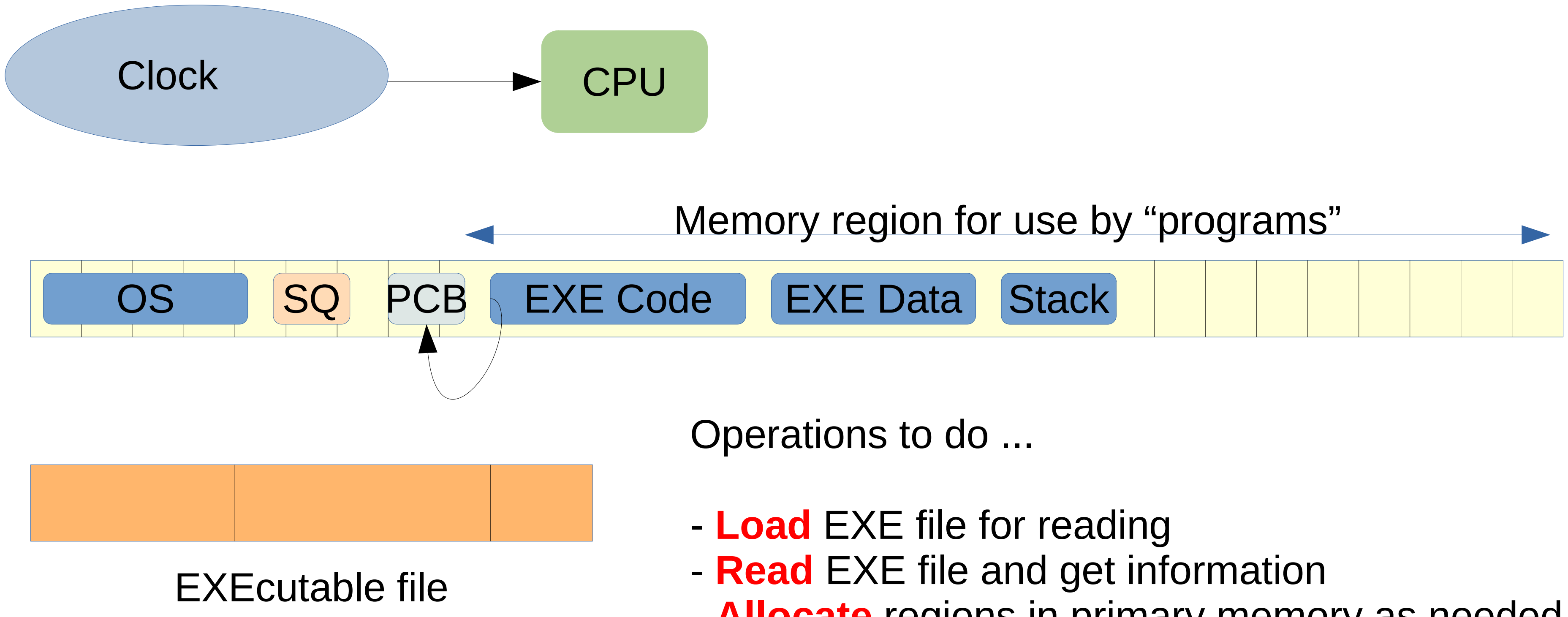
Loading a Program



Operations to do ...

- **Load** EXE file for reading
- **Read** EXE file and get information
- **Allocate** regions in primary memory as needed
- **Initialise** memory regions as noted in EXE
- **Record** the first instruction as noted in the EXE
- **Collect** all this information into a data structure
- **Give** this data structure to the scheduler **S**

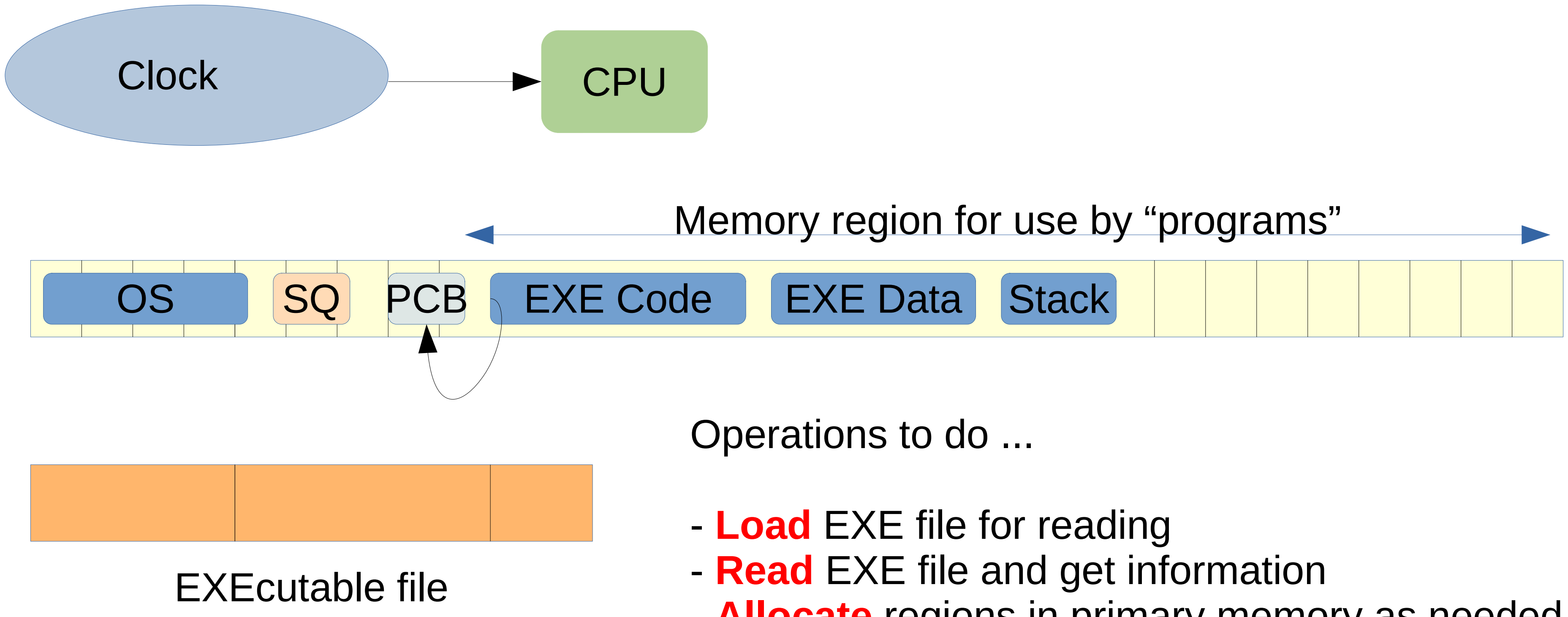
Loading a Program



Operations to do ...

- **Load** EXE file for reading
- **Read** EXE file and get information
- **Allocate** regions in primary memory as needed
- **Initialise** memory regions as noted in EXE
- **Record** the first instruction as noted in the EXE
- **Collect** all this information into a data structure
- **Give** this data structure to the scheduler **S**

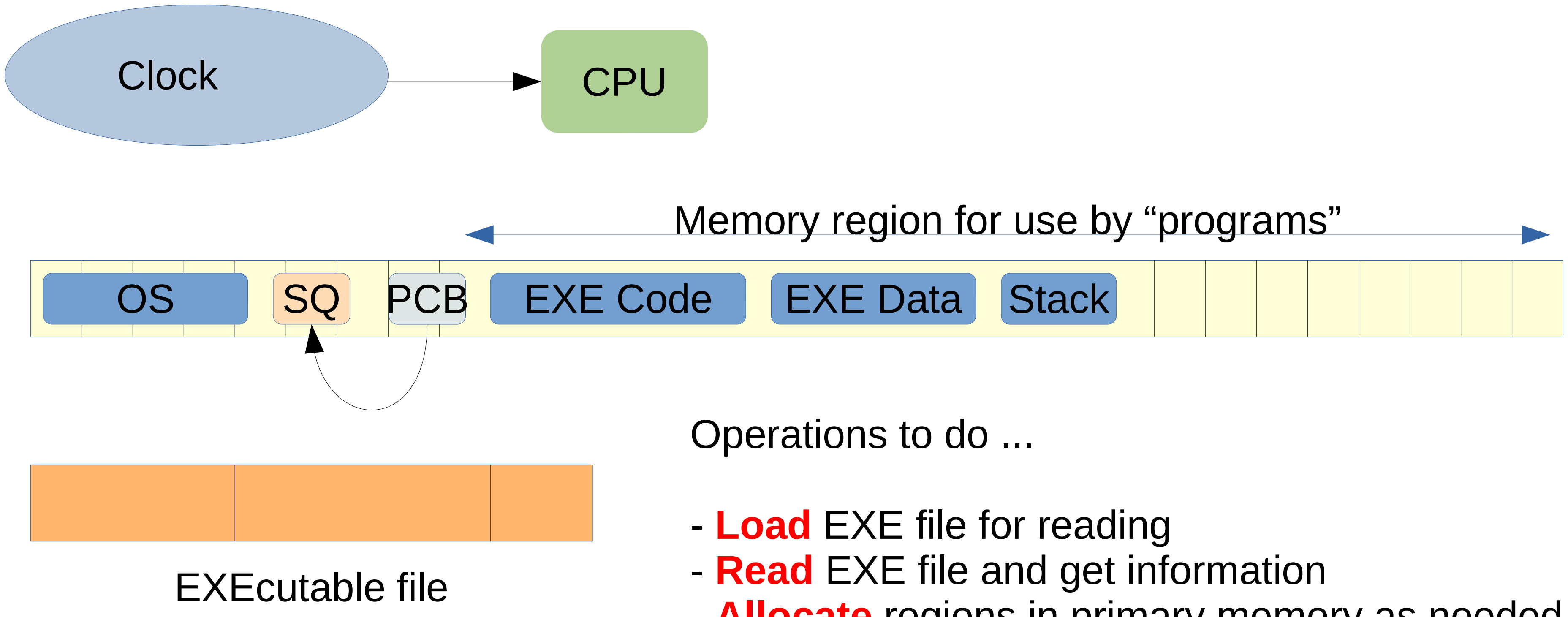
Loading a Program



Operations to do ...

- **Load** EXE file for reading
- **Read** EXE file and get information
- **Allocate** regions in primary memory as needed
- **Initialise** memory regions as noted in EXE
- **Record** the first instruction as noted in the EXE
- **Collect** all this information into a data structure
- **Give** this data structure to the scheduler **S**

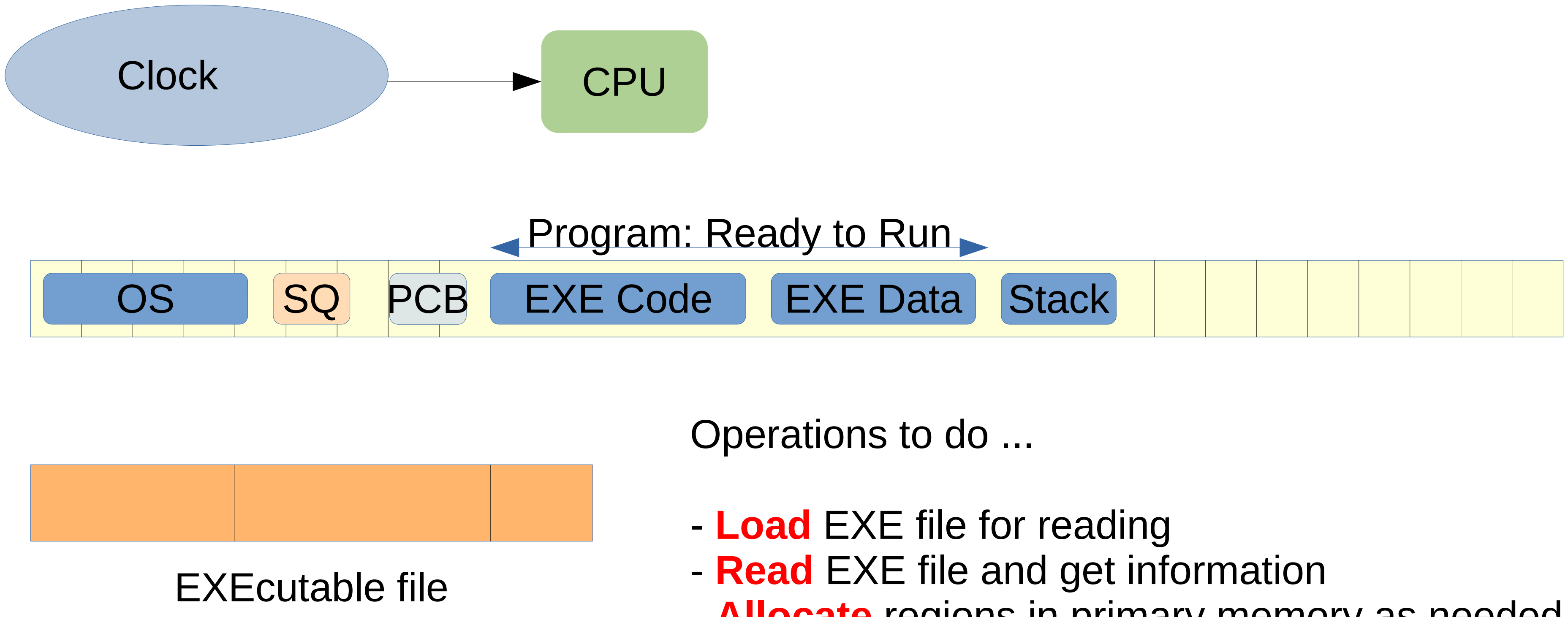
Loading a Program



Operations to do ...

- **Load** EXE file for reading
- **Read** EXE file and get information
- **Allocate** regions in primary memory as needed
- **Initialise** memory regions as noted in EXE
- **Record** the first instruction as noted in the EXE
- **Collect** all this information into a data structure
- **Give** this data structure to the scheduler **SQ**

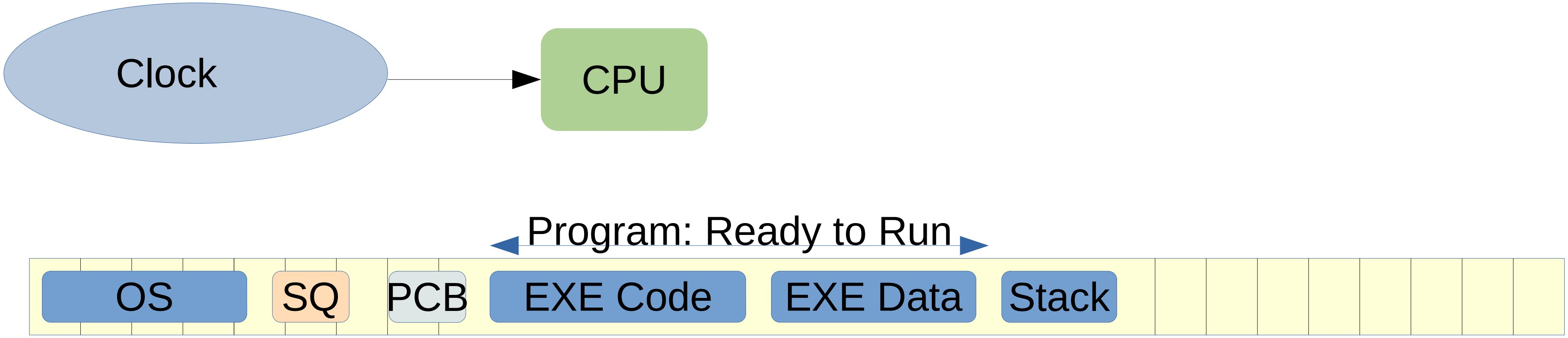
Loading a Program



Operations to do ...

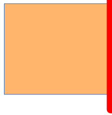
- **Load** EXE file for reading
- **Read** EXE file and get information
- **Allocate** regions in primary memory as needed
- **Initialise** memory regions as noted in EXE
- **Record** the first instruction as noted in the EXE
- **Collect** all this information into a data structure
- **Give** this data structure to the scheduler **S**

Loading a Program



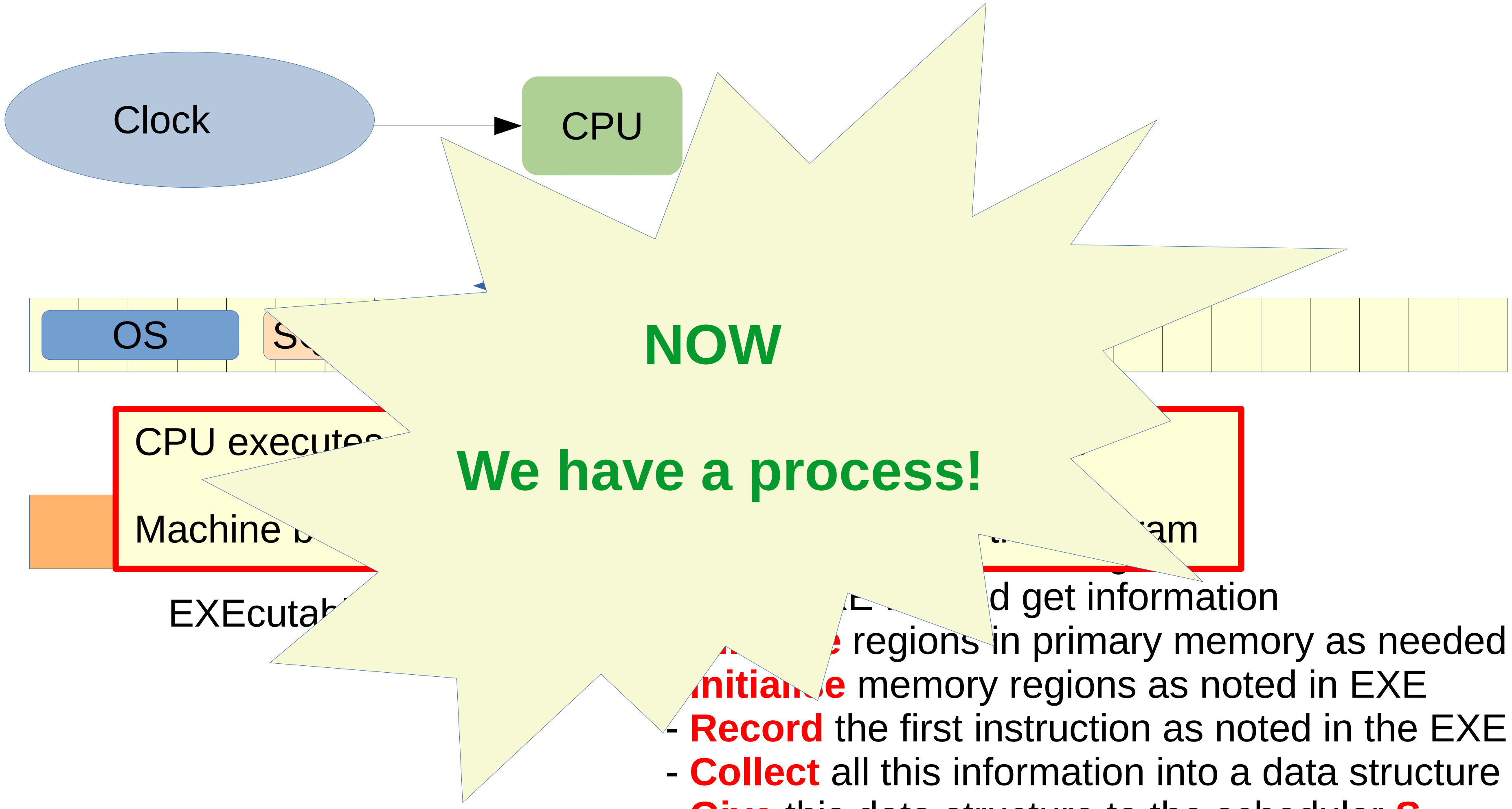
CPU executes program when the scheduler “schedules”

Machine begins to change its states according to the program



- **Read** EXE file and get information
- **Allocate** regions in primary memory as needed
- **Initialise** memory regions as noted in EXE
- **Record** the first instruction as noted in the EXE
- **Collect** all this information into a data structure
- **Give** this data structure to the scheduler **S**

Loading a Program



NOW

We have a process!

E d get information

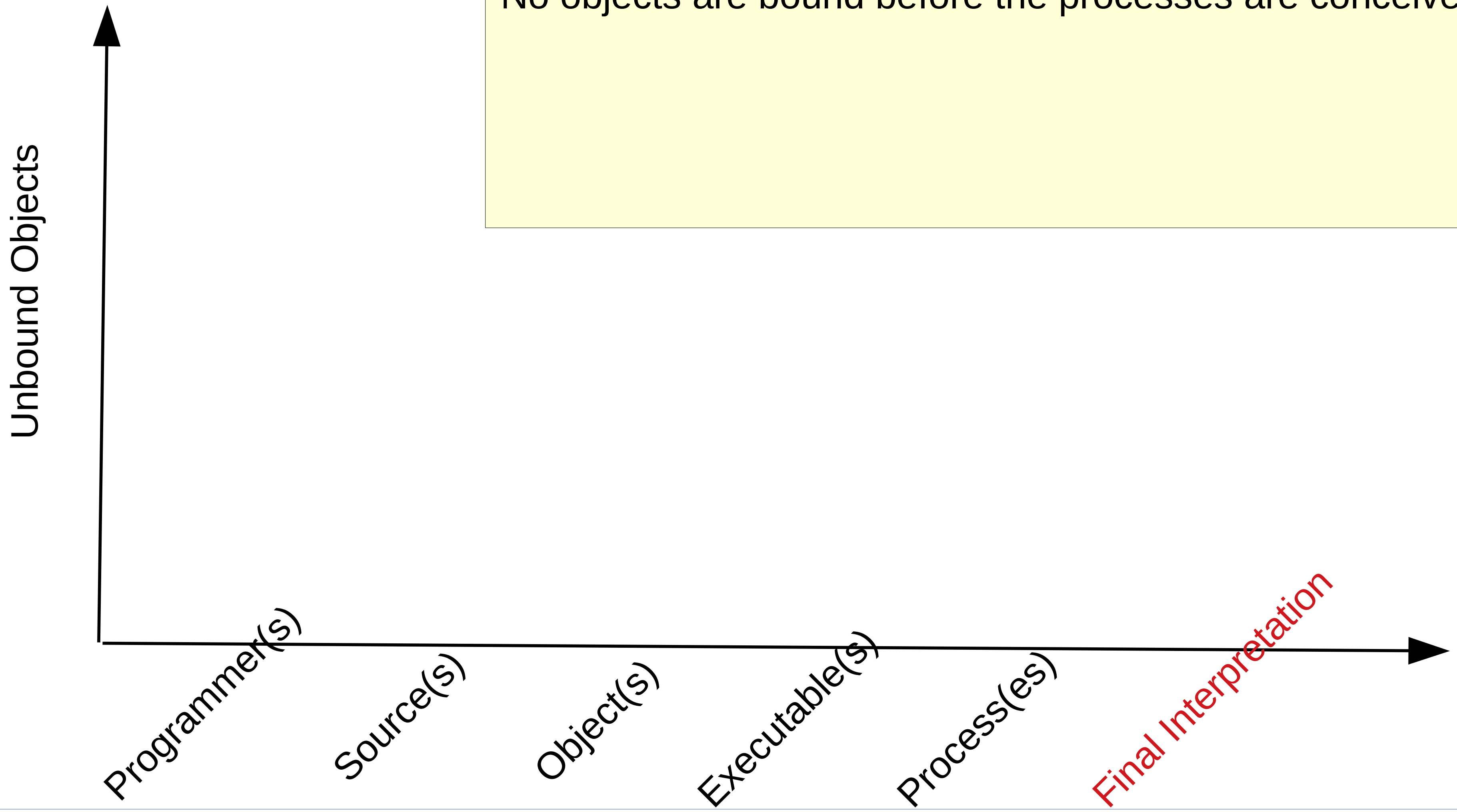
e regions in primary memory as needed

Initial e memory regions as noted in EXE

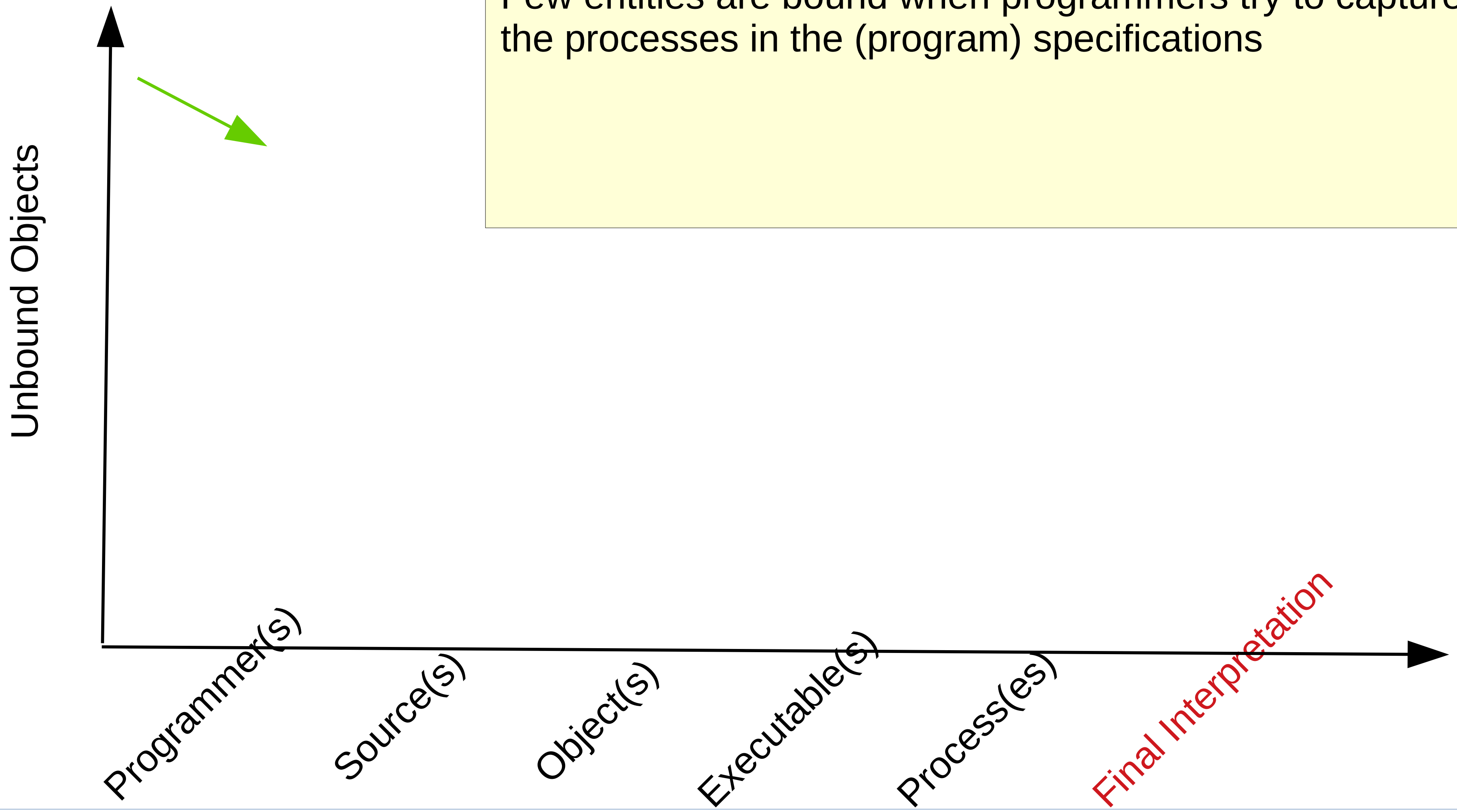
- **Record** the first instruction as noted in the EXE
- **Collect** all this information into a data structure
- **Give** this data structure to the scheduler **S**

System Software

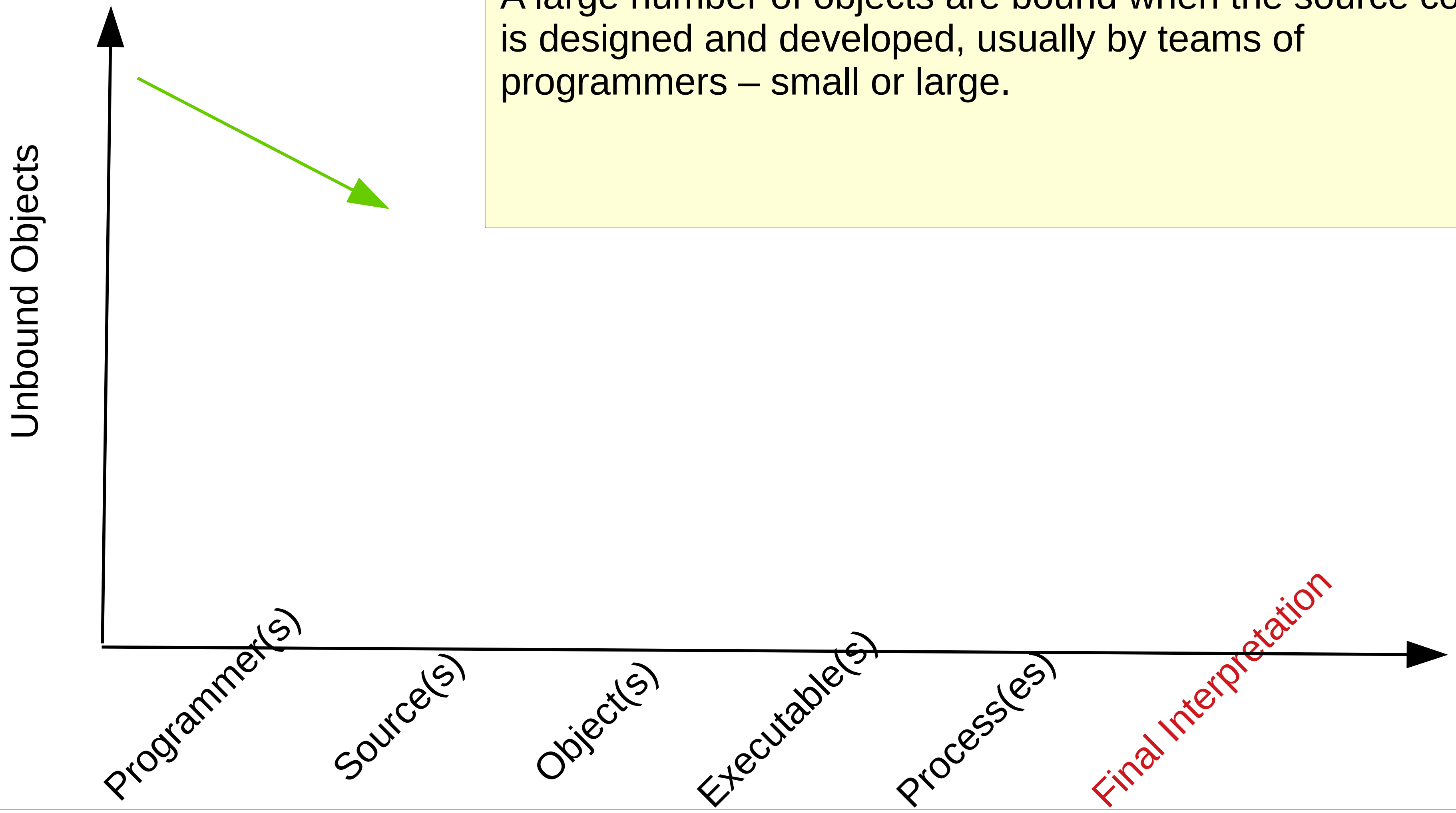
No objects are bound before the processes are conceived.



System Software

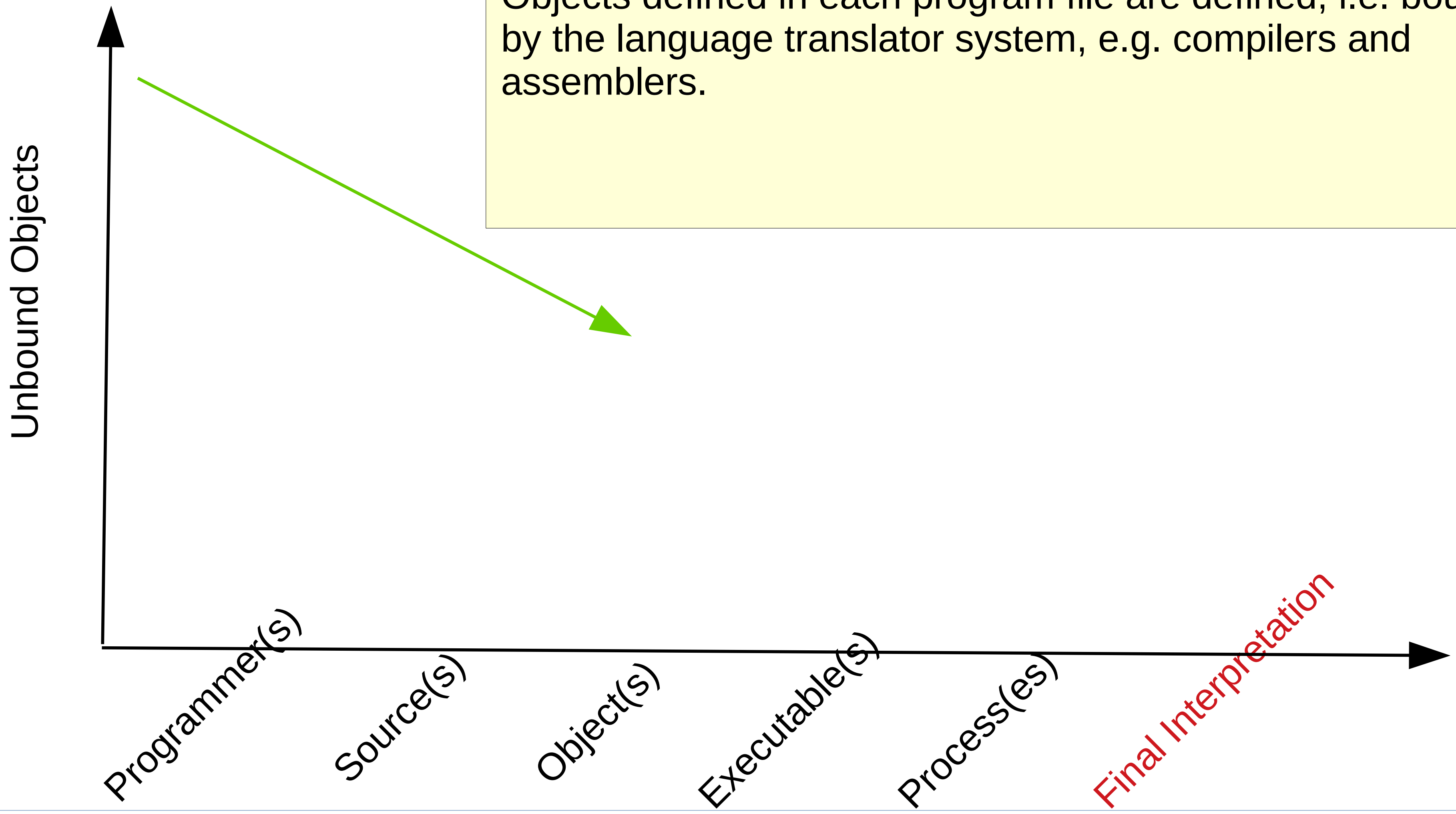


Few entities are bound when programmers try to capture the processes in the (program) specifications



A large number of objects are bound when the source code is designed and developed, usually by teams of programmers – small or large.

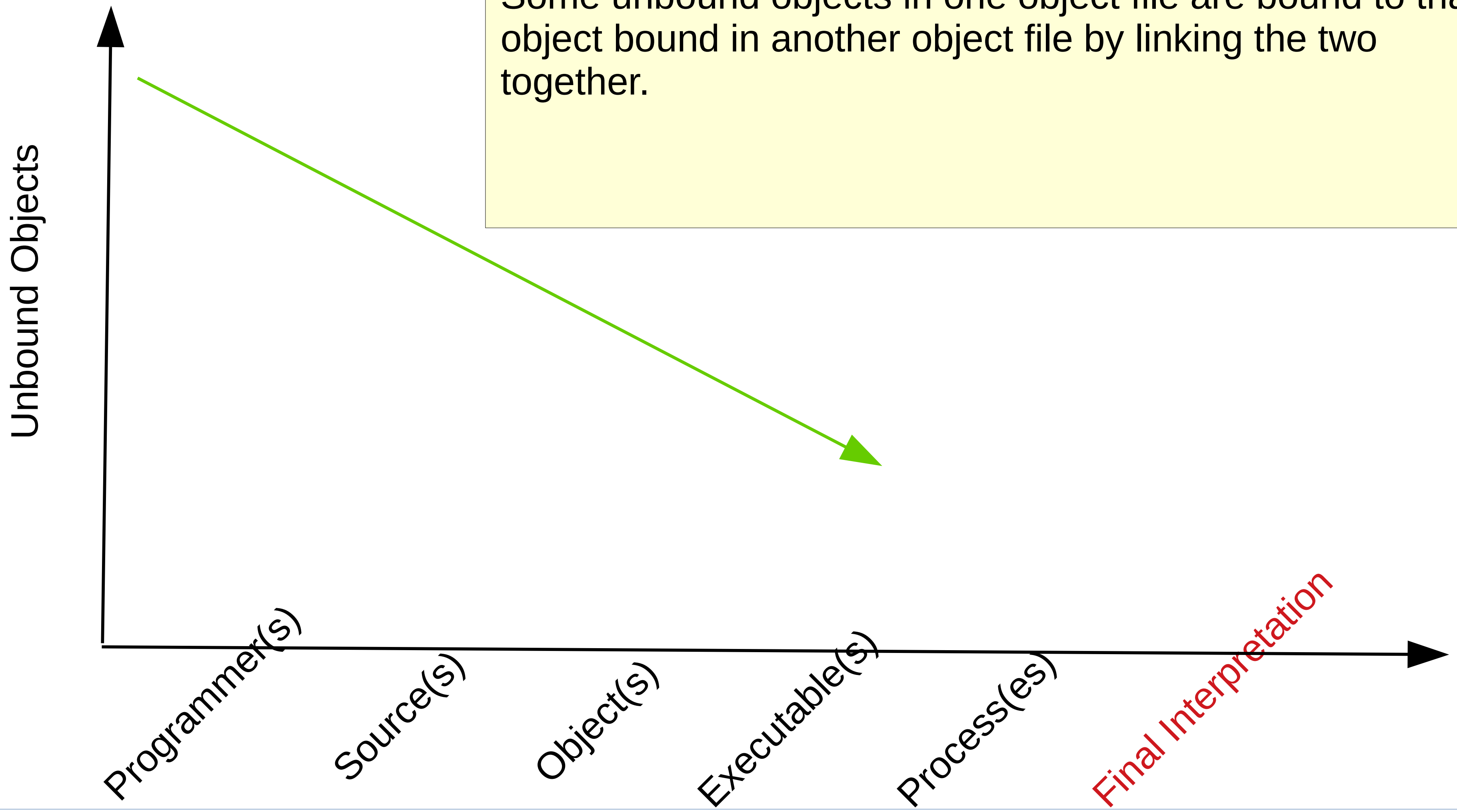
System Software



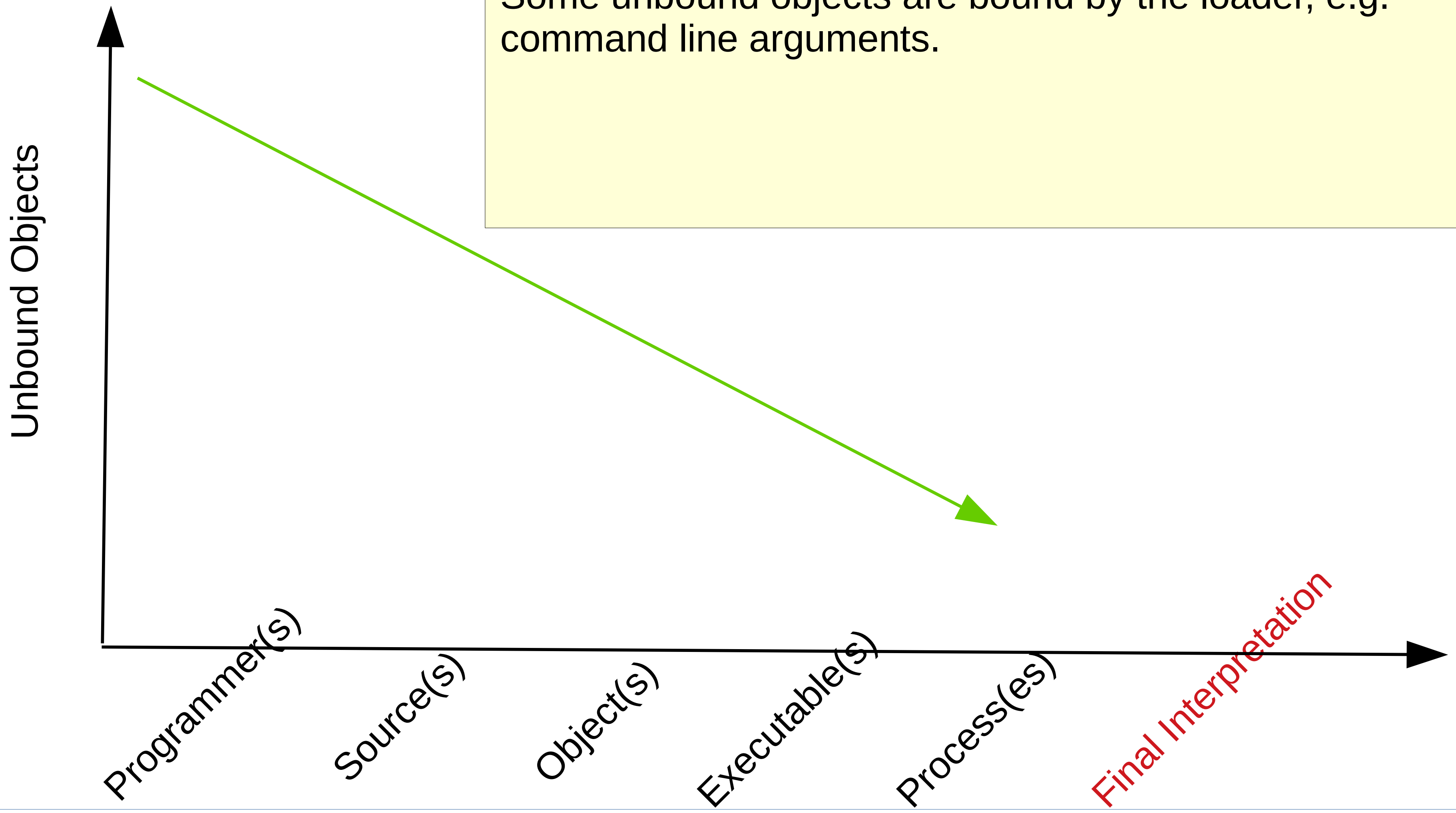
Objects defined in each program file are defined, i.e. bound, by the language translator system, e.g. compilers and assemblers.

System Software

Some unbound objects in one object file are bound to that object bound in another object file by linking the two together.



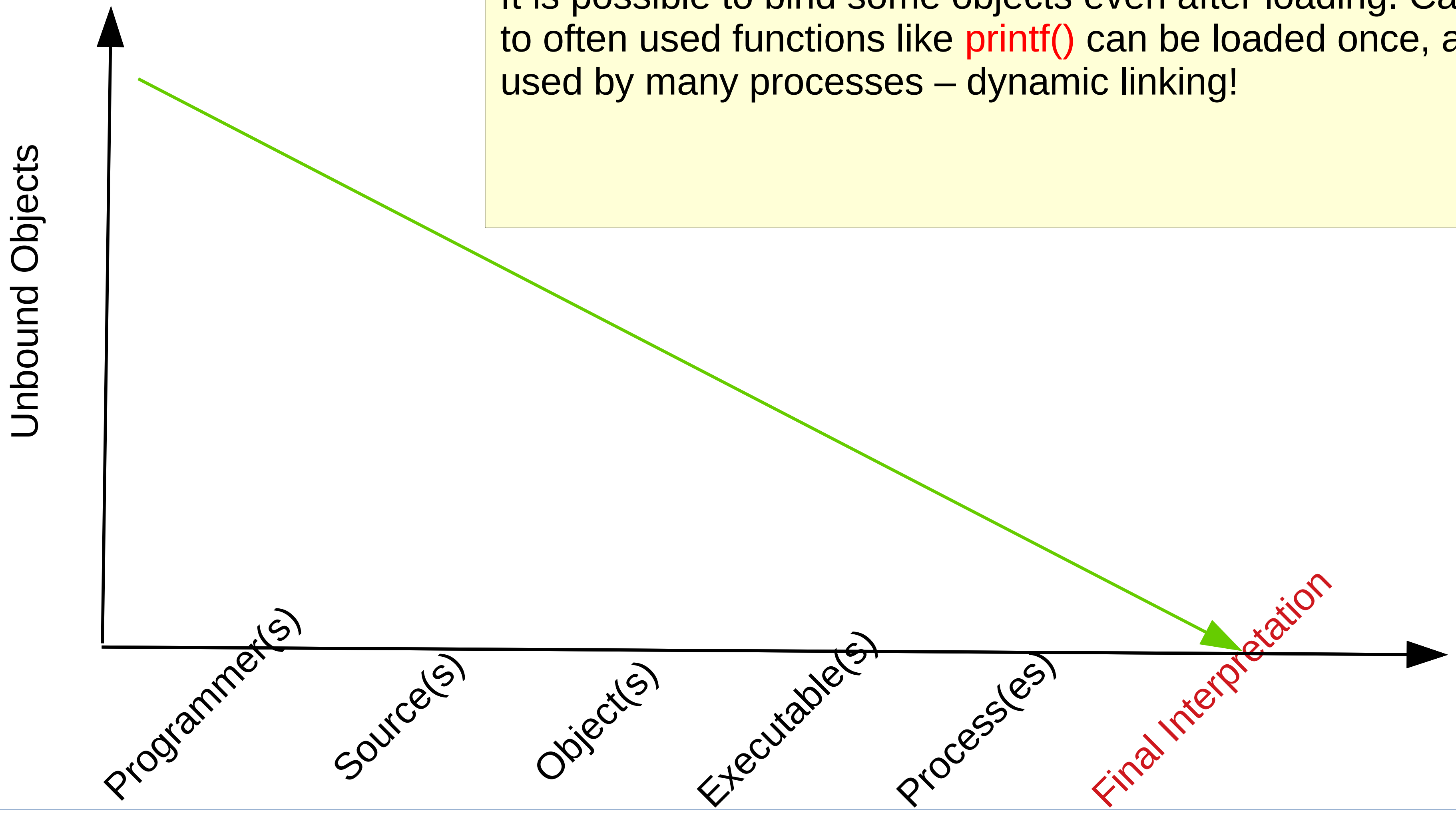
System Software



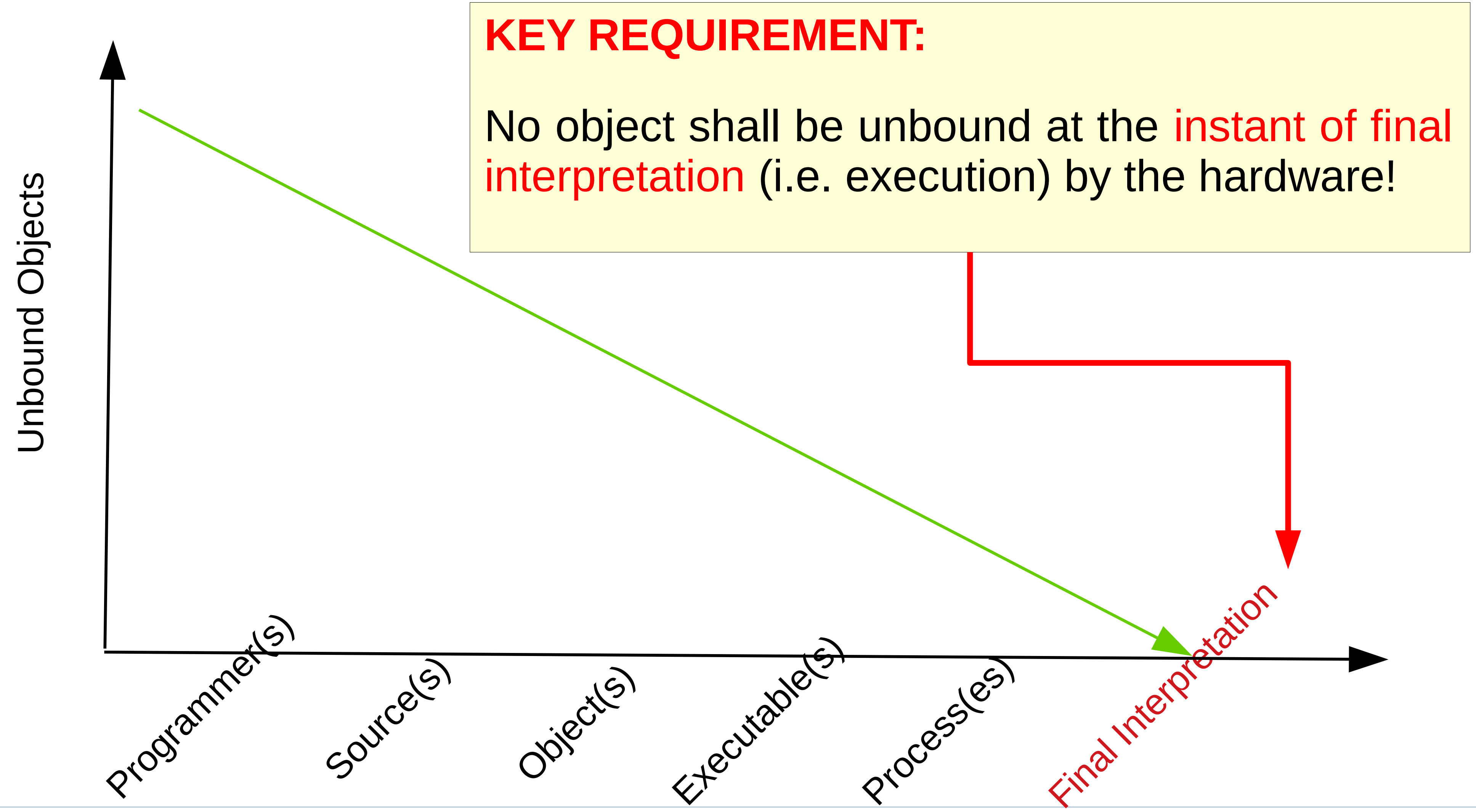
Some unbound objects are bound by the loader, e.g. command line arguments.

System Software

It is possible to bind some objects even after loading. Calls to often used functions like printf() can be loaded once, and used by many processes – dynamic linking!



System Software



KEY REQUIREMENT:

No object shall be unbound at the instant of final interpretation (i.e. execution) by the hardware!

Suggested Lab Exercises

“Discover/Explore/Study” questions

- Examine the ASM version of a C program and find out the details of construction and destruction of an activation record of a function call in that program.
- Can a process discover its own memory layout?
- Can a process find its own memory layout?
- Can you write a C program such that a given callee prints out its own activation record?

Overall Summary

- Runtime Environments: Setting up the bindings that cannot be done statically by the language processors.
- Programmer level defined by the HLL
- Machine level defined by the Hardware + OS + System Software
- HLL Transformations:
  - Static – By Compiler, without runtime context
  - Dynamic – By other tools, exploiting runtime context
- Runtime Environments: Bridge between static and dynamic

Some References

- Compiler, Assembler, Linker & Loader: A Brief Story
- C/C++ Building Process on Complex SoCs
- Process (how an OS uses the idea)
- OS Structures
- Managing Heap Memory
- PE Format: Description 1, Description 2.

Thank You & Questions