

III - Assembly - I

ICS 26Fall

Junqi Xu 2026.9.23

Recall

Concepts:

Stack

Heap

Stack Frame

Type Layout

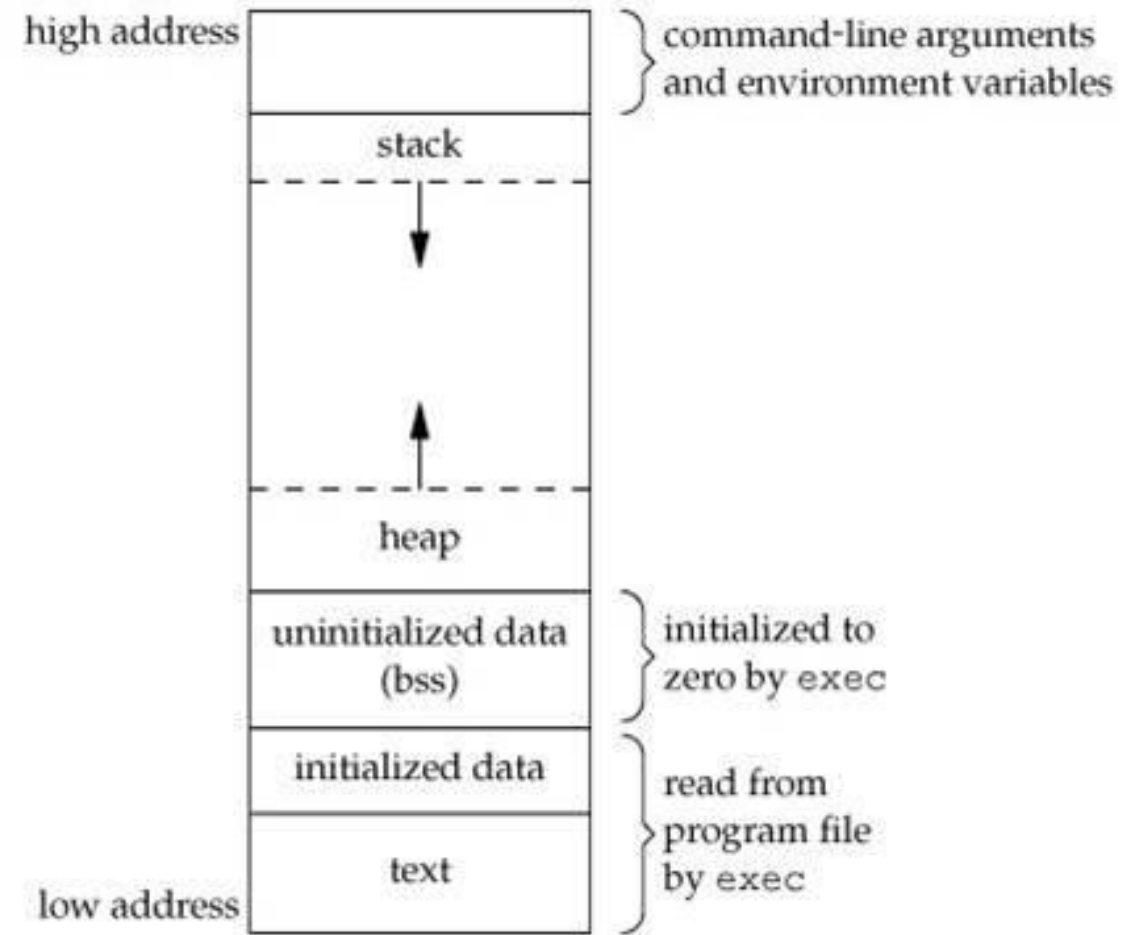


Fig.1^[1]

[1] STEVENS W R, RAGO S A. Advanced programming in the UNIX environment[M]. 3rd ed. Upper Saddle River, NJ: Addison-Wesley, 2013.

Recall

Concepts:

Stack

Heap

Stack Frame

Type Layout

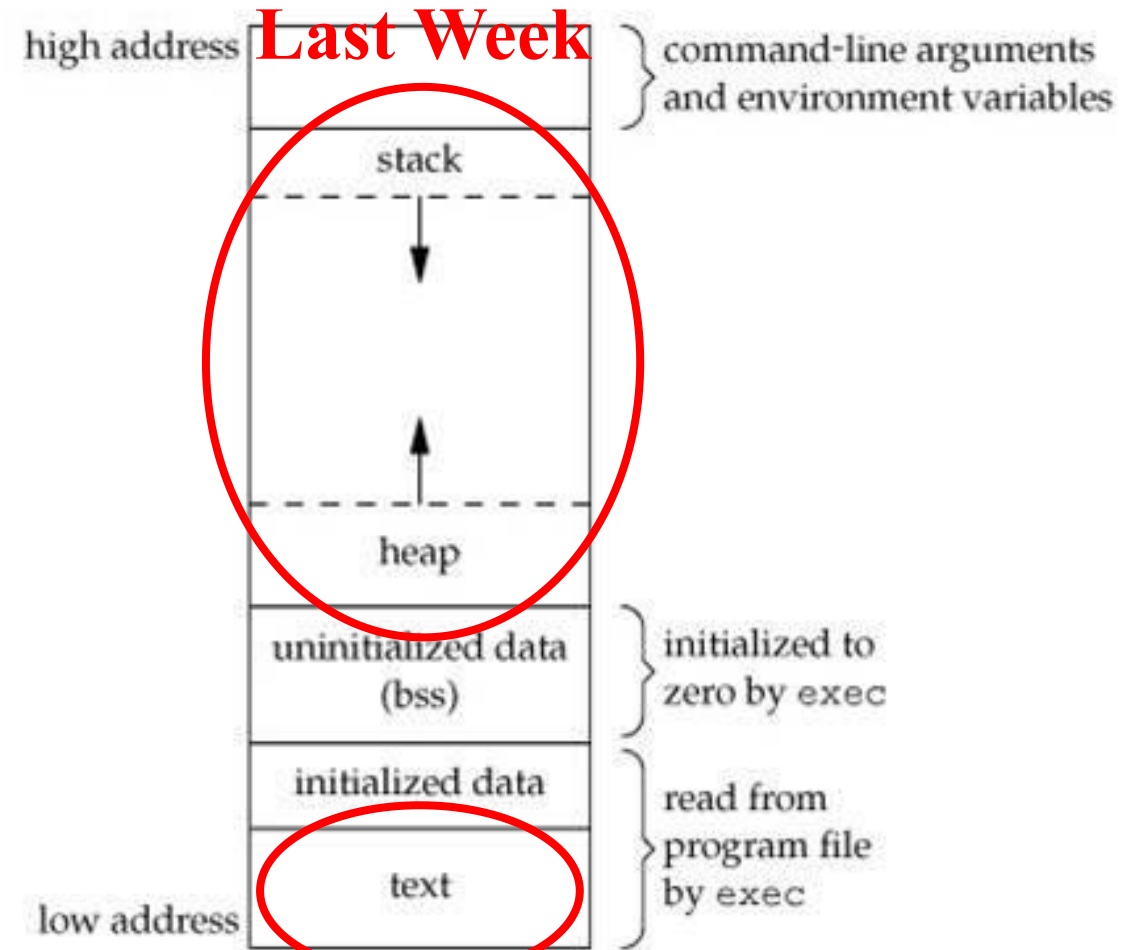


Fig. 1[1]

Today

Topics

- Intros
- PC-Relative
- Memory Addressing

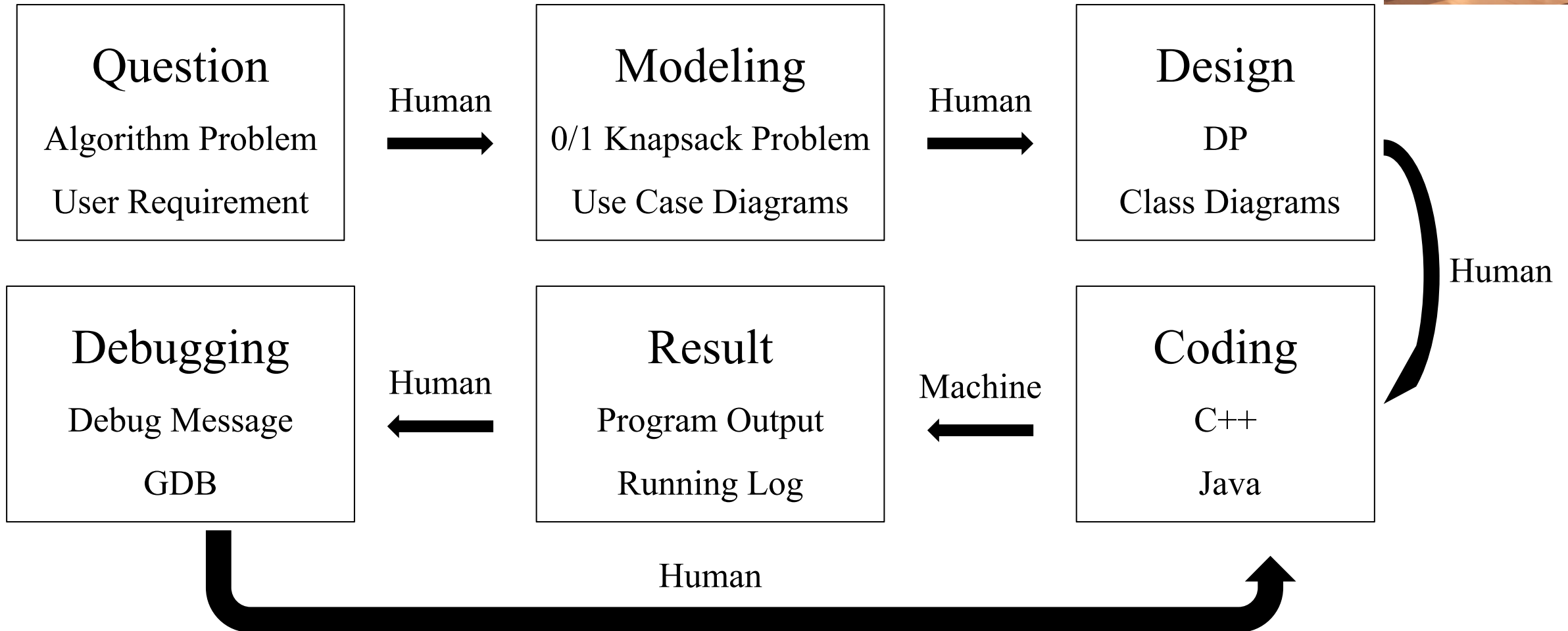
Intros

What is programming?

- Question Finding
- Modeling
- Design
- Coding
- Result Analyzing
- Debugging

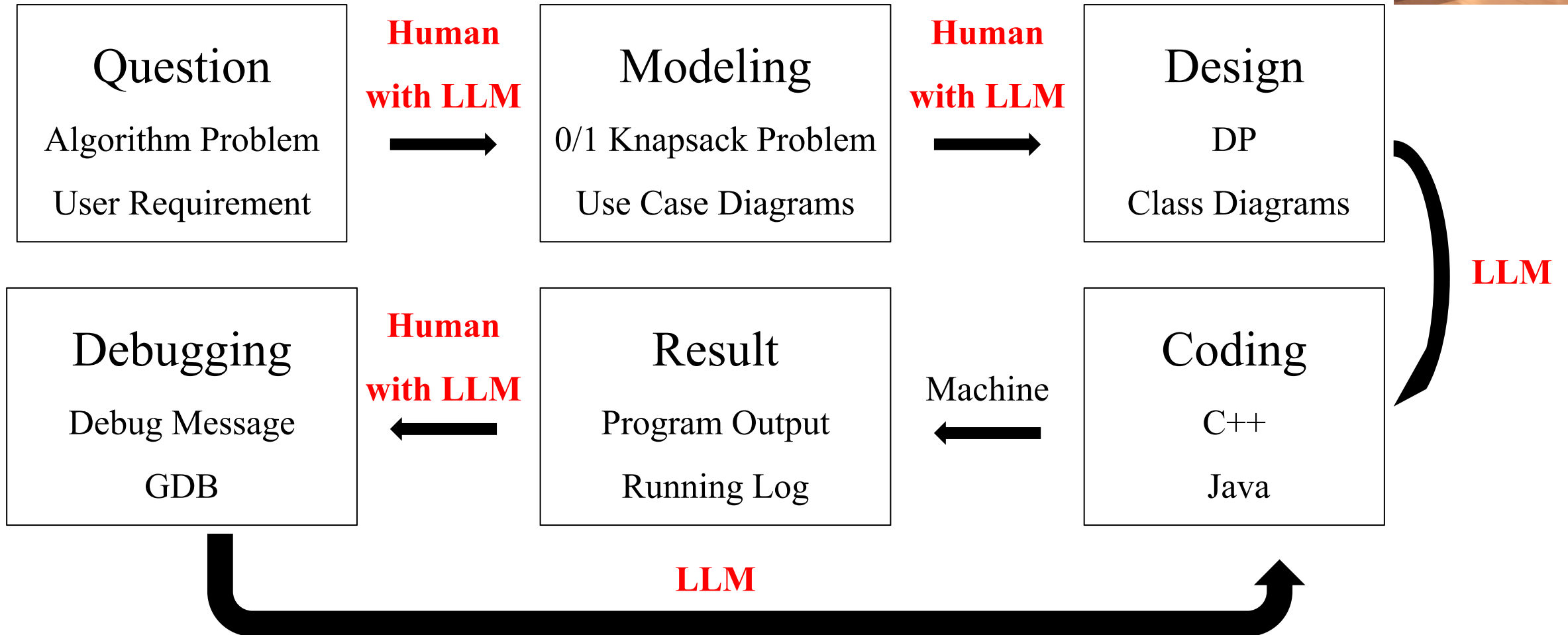
Intros

Programming (Legacy):



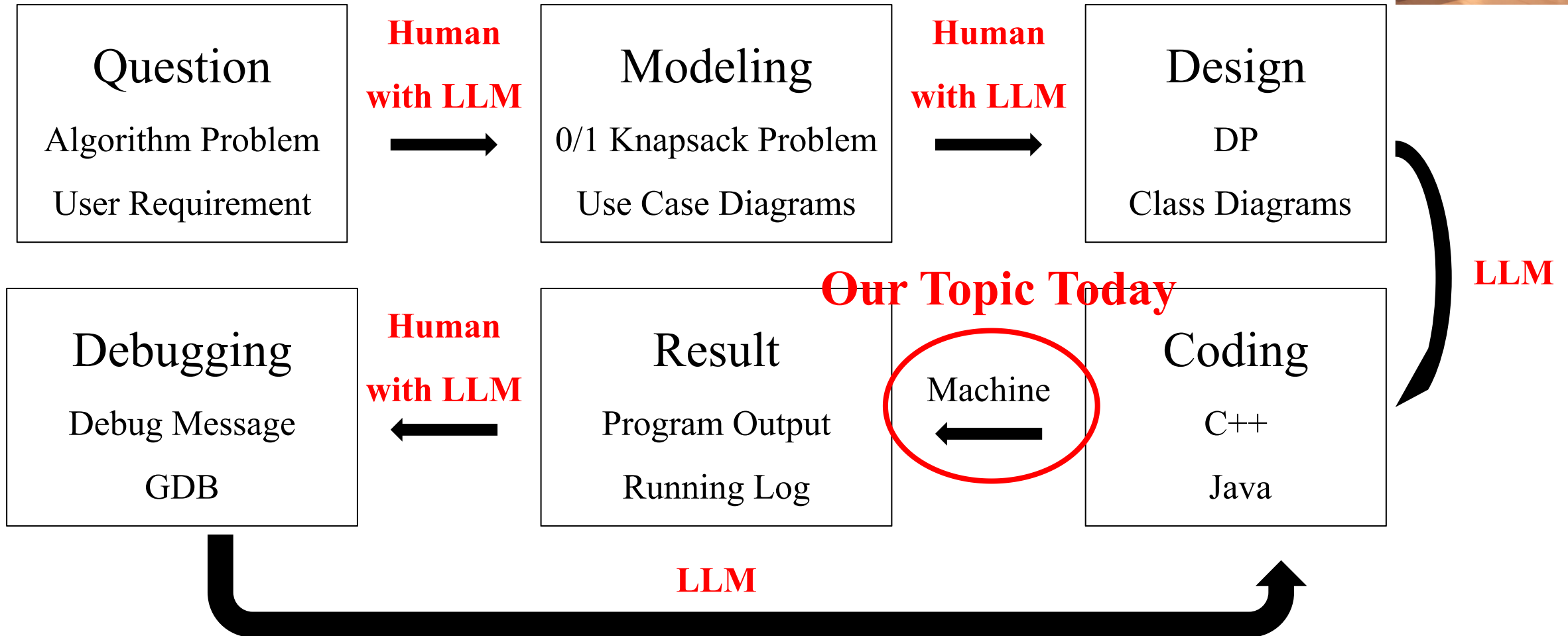
Intros

Programming (Current):



Intros

Programming (Current):



Intros

High-Level
Programming Language

C/C++

Rust

Pascal

...

Compiler

(Assembler)


Be Translated to

Assembly
Programming Language?

x86-64

Risc-V

ARM64

...

Intros

High-Level
Programming Language

C/C++
Rust
Pascal
...



Machine-Independent

Compiler

(Assembler)

Be Translated to

Assembly

Programming Language?

x86-64
Risc-V
ARM64
...



Machine-Dependent

Intros

High-Level
Programming Language

C/C++
Rust
Pascal
...

Machine-Independent

Compiler

(Assembler)

Be Translated to

Principle

Semantic Preservation

Assembly

Programming Language?

x86-64
Risc-V
ARM64
...

Machine-Dependent



Intros

Even Higher-Level
Nature Language

中文
English
...

LLM

Be Translated to
Semantic Preservation?

High-Level
Programming Language

C/C++
Rust
Pascal
...

Intros

Question:

Assumed that one compiler could translate **one** PL to **one** assembly.

Now there exists 100 PLs, 100 assemblies.

How many compilers do we need to make every PL could be translated to all assemblies? When a new PL is created, how many new compilers should be implemented?

Discuss:

How to solve this problem?

PC-Relative - PIC

Position-Independent Code

Or PIE (Position-Independent Executable)

PIC could be placed into anywhere valid!

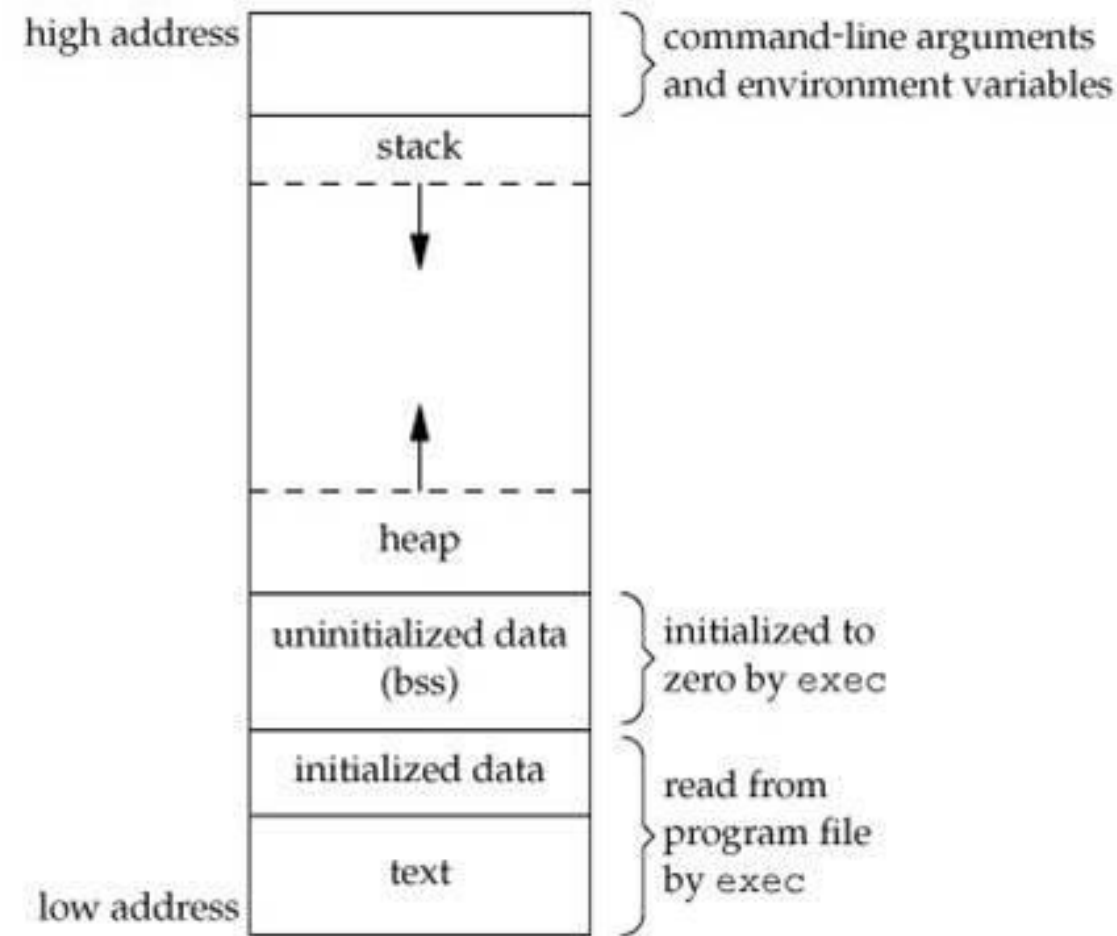


Fig.1^[1]

[1] STEVENS W R, RAGO S A. Advanced programming in the UNIX environment[M]. 3rd ed. Upper Saddle River, NJ: Addison-Wesley, 2013.

PC-Relative - PIC - Why PIC

PIC (vs. PDC)

Pros (before 1980s, when MMU was not yet popular):

PIC could be easily loaded into anywhere of memory.



Pros (after MMU and VM becoming the mainstream):

Can work with ASLR.

Multithreading-Friendly.



Cons:

Slower execution (negligible).

PC-Relative - PIC - How to make codes PIC

Relative addressing plays an essential role!

Absolute addressing:

```
jmp 0x15213
```

```
call evil
```

```
add $1, %rax
```

 Jump target. Address has to be exactly 0x15213.

Relative addressing:

```
jmp .+7
```

```
call evil
```

```
add $1, %rax
```

PC-Relative - PIC - How to make codes PIC

PC-Relative is an implementation of relative addressing.

```
jmp .+7  
call evil  
add $1, %rax
```

CPU would calculate target address by:

PC's value (in x86-64, it is stored in **%rip**)

minus

Jump operator's size (2, in this case)

plus

Relative address (+7, in this case)

Memory Addressing

```
int a[20];
```

```
...
```

```
int i = a[5];
```

How to get value of **a[5]**?

1. Get value of **a**.
2. Calculate address of **a[5]**.
3. Access **a[5]**.

Memory Addressing - Get Value of a Symbol

Case1: Local Symbol (On the Stack)

Use stack pointer (%rsp/%rbp)

```
movq 24(%rsp), %rax
```

Case2: Global Symbol (On .data Segment)

Use PC-Relative

```
movq a(%rip), %rax
```

Case3: External Symbol (On External Files)

Use GOT (external function additionally requires PLT)

```
movq a@GOTPCREL(%rip), %rax
```

Memory Addressing - PLT & GOT

PLT, Procedure Linkage Table

GOT, Global Offset Table

GOT & PLT won't be generated until **Linking**.

```
movq a@GOTPCREL(%rip), %rax
```

a@GOTPCREL is a **Relocation Entry**

It would be replaced with a relative address of GOT by **Static Linker**.

Value of external value (if it exists) would lie in GOT.

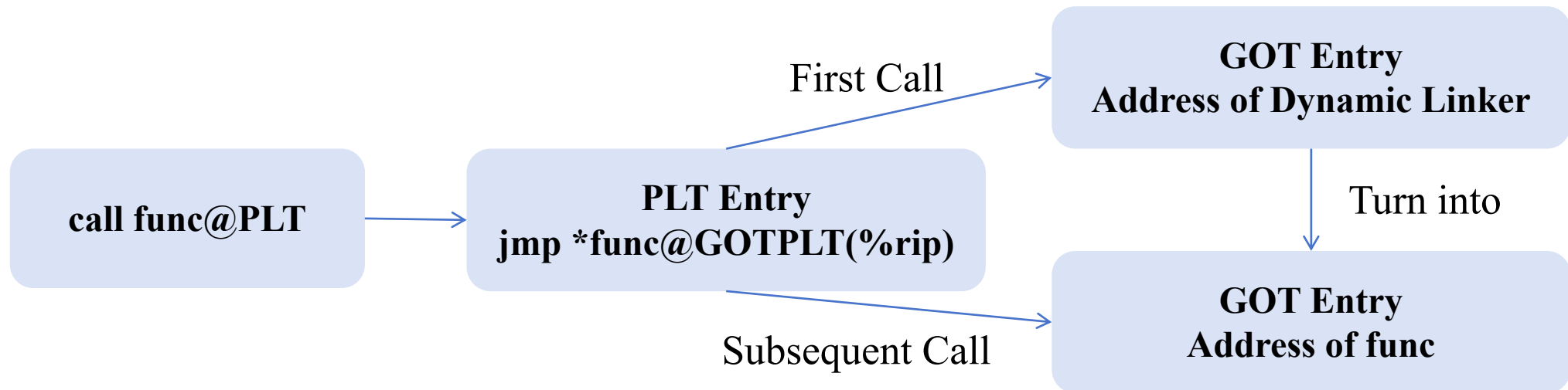
Memory Addressing - PLT & GOT

PLT is a set of **trampolines**.

It is used to implement lazy binding.

`call __isoc99_scanf@PLT`

`__isoc99_scanf@PLT` is a **Relocation Entry**



Memory Addressing - Calculate Address & Access

A.K.A. Addressing Modes

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base + displacement
Memory	(r_b, r_i)	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$Imm(r_b, r_i)$	$M[Imm + R[r_b] + R[r_i]]$	Indexed
Memory	$(, r_i, s)$	$M[R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(, r_i, s)$	$M[Imm + R[r_i] \cdot s]$	Scaled indexed
Memory	(r_b, r_i, s)	$M[R[r_b] + R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Scaled indexed

Figure 3.3 Operand forms. Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

Assumed:

`a: double[72]`

`b: double[8][8]`

`i, j: int`

`a in %rax`

`b in %rcx`

`i in %rsi`

`j in %rdi`

How to address:

`a[5]`

`a[i]`

`b[5][j]`

`b[i][5]`

`b[i][j]`

Takeaway

Concepts:

Semantic Preservation, IR

PIC, ASLR, Relative Addressing, PC-Relative

GOT, PLT, Relocation Entry