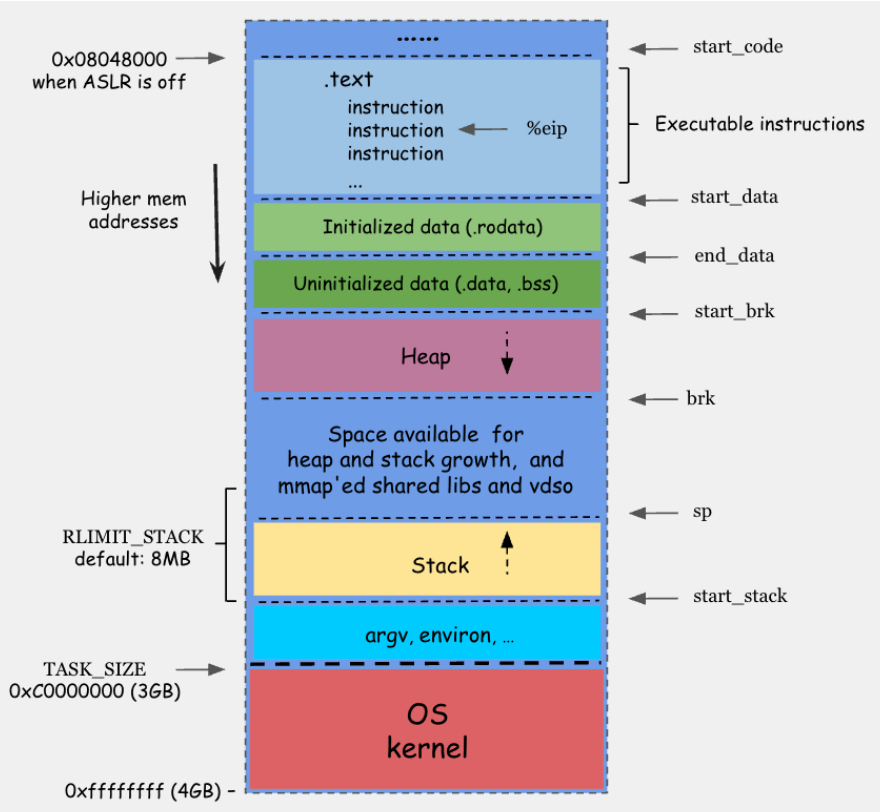
- **Q1: How does a program look in memory?**
    - All programs are given by the OS the illusion that they have exclusive access to a continuous array of memory addresses in virtual memory, known as the Virtual Address Space (VAS).
    - The VAS of each program is divided into a user space and a kernel space portion.
        - For Linux x86_32 the canonical split is the lower 3GB for the user space portion, and the upper 1GB for the kernel space portion.
        - For Linux x86_64 the canonical split is the lower 128TB for the user space portion, and the upper 128TB for the kernel space portion.
    - The user space portion of the VAS of each program is further divided into segments.
        - **Static segments** (their size is static during run time)
            - The .**text** segment: Contains the executable code (instructions) of the program. This segment is marked read-execute. [Why? Self-modifying code can't ever be a good idea.]
            - The .**rodata data** segment: Contains constant values, and is marked read-only, ensuring immutable data.
            - The .**data data** segment: Contains global and static variables that are initialized, and is read-write, so that values can be modified at runtime.
            - The .**bss data** segment: Contains global and static variables that are uninitialized or set to 0, and is also read-write.
        - **Dynamic segments** (their size is not static during run time)
            - **The heap**: Grows at run-time towards higher addresses and contains variables dynamically allocated—e.g., by the malloc user-space library function. No explicit, fixed max-size.
            - **The stack**: It is a Last-In-First-Out (LIFO) memory region that grows dynamically during execution towards lower addresses (and shrinks conversely). It is used for bookkeeping during function calls—that is, to temporarily save local variables, arguments, and return addresses.
            - **Other dynamic segments**
                - **Shared libraries**: User-space libraries that are necessary to many user programs are mapped in the address spaces of multiple processes from a single, shared instance in physical memory. (Will explain.)
                - **VDSO**: A tiny subset of frequently-used syscalls (e.g., gettimeofday()) is mmap'ed by the OS kernel to the address space of all processes to allow fast use without the cost of a full syscall invocation.
                - **Large dynamic allocations:** In modern systems, large dynamic allocations may appear in anonymous mapping segments. That is, outside the heap segment ending in program break (brk ptr).

## Layout of a program's virtual address space (Linux x86_32)




## Sample C program, and its segments in memory

```c
char uninitialized_global[1000];
const char *message = "Hello, World!\n";

void foo() {
  unsigned long sp;
  __asm__ (
    "mov %0, sp"
    : "=r" (sp)
  );
  printf("@foo / Current stack pointer (sp): %lx\n", sp);
}

void main() {
  unsigned long sp;
  printf("@.bss variable: %p\n", &uninitialized_global);
  printf("@.rodata variable: %p\n", &message);
  __asm__ (
    "mov %0, sp"
    : "=r" (sp)
  );
  printf("@main / Current sp: %lx\n", sp);
  char *heap = (char *)malloc(50 * sizeof(char));
  printf("@heap memory starts at: %p\n", heap);
  foo();
}
```

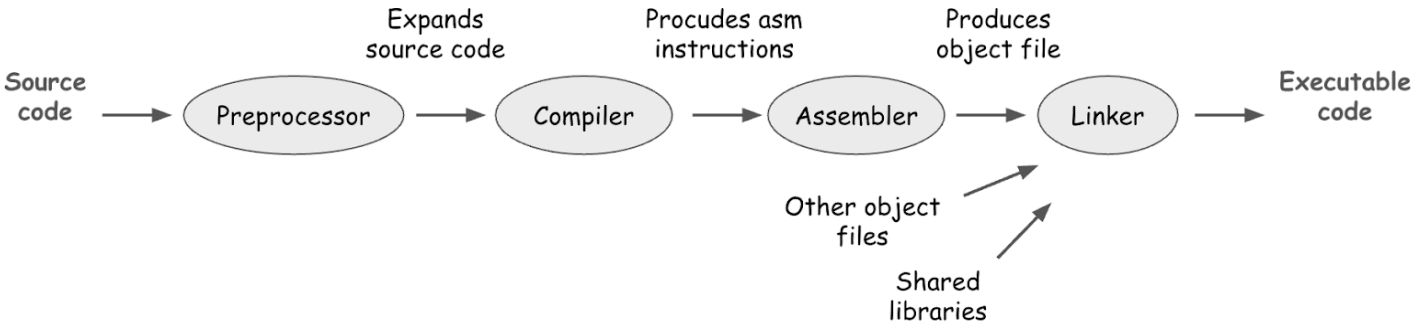
```
➜ cat /proc/1245382/maps
aaaadcda0000-aaaadcda1000 r-xp  …  /os/sample
aaaadcdb0000-aaaadcdb1000 r--p  …  /os/sample
aaaadcdb1000-aaaadcdb2000 rw-p  …  /os/sample
aaaaed20d000-aaaaed22e000 rw-p  …  [heap]
ffffbe610000-ffffbe798000 r-xp  …  /usr/lib/libc.so.6
ffffbe798000-ffffbe7a7000 ---p  …  /usr/lib/libc.so.6
ffffbe7a7000-ffffbe7ab000 r--p  …  /usr/lib/libc.so.6
ffffbe7ab000-ffffbe7ad000 rw-p  …  /usr/lib/libc.so.6
ffffbe7ad000-ffffbe7b9000 rw-p  …
ffffbe7d9000-ffffbe804000 r-xp  …  /usr/lib/ld-linux-aarch64.so.1
ffffbe80e000-ffffbe810000 rw-p
ffffbe810000-ffffbe812000 r--p  …  [vvar]
ffffbe812000-ffffbe813000 r-xp  …  [vdso]
ffffbe813000-ffffbe815000 r--p  …  /usr/lib/ld-linux-aarch64.so.1
ffffbe815000-ffffbe817000 rw-p  …
ffffff6461000-ffffff6482000  rw-p  …  [stack]

➜ ./sample
@.bss variable: 0xaaaadcdb1020
@.rodata variable: 0xaaaadcdb1010
@main / Current sp: ffffff64810e0
@heap memory starts at: 0xaaaaed20d6b0
@foo / Current sp: ffffff64810c0
```

- **Q2: Who (what) sets up a program in memory?**
  - The **loader** is a crucial component of the OS kernel responsible for loading a binary program (i.e., an **executable**) from storage into memory.
  - Given an executable in storage, the loader reads and interprets it, and then sets up the appropriate segments in the VAS of the process* that will execute it.
    - The **static segments** of the process (i.e., the .text, .data, and .rodata segments) are initialized by copying them from the target executable.
    - The **default dynamic segments** of the process (i.e., the heap and the stack) are initialized by the loader, used at runtime, and if needed the OS kernel intervenes to manage them (e.g., expand them as needed).
      - **Heap**: It's a core programming abstraction used at runtime for allocations of dynamic variables (e.g., using malloc()).
        - It is implemented and managed by user-space libraries.
        - The kernel intervenes if user-space libraries wish to increase the heap size of a process.
        - The canonical heap area (i.e., from start_brk to brk) is used for small/moderate allocations (<128 KB), because the **sbrk**() syscall can only increase the program break, and is, thus, not flexible due to internal fragmentation.
        - For large allocations, **mmap()** is used.
      - **Stack**: It's a fundamental programming abstraction, and every program has direct hardware support to manage its own stack.
        - The OS kernel expands the stack of a process as needed.
        - The OS kernel faults with a stack overflow if a process's stack exceeds a fixed stack limit (e.g., the default in Linux is 8MB).
  - Once the executable is loaded and its segments are properly set in memory, the loader transfers control to the executable's entry point (e.g., **_start**).
  - If the executable is **statically linked**, the loader's job is complete after loading the program into memory and transferring control to its entry point.
  - If the executable is **dynamically linked**, after the loader has set up the initial memory layout, it calls the **dynamic linker.**
    - The dynamic linker is a user-space program specified in the .interp section of the executable.
    - The dynamic linker (e.g., ld.so or ld-linux.so) resolves links at runtime—e.g., if shared libraries are used. (More on this soon.)
  - The core logic of the Linux loader is here.

- **Q3: What is the interface between the loader and executable files?**
  - The loader needs to know how to parse and interpret executable files.
  - We need a specification to serve as the contract (interface) between executables and the loader.
  - The Executable and Linkable Format (**ELF**) is a common standard file format for executable files, object code, and shared libraries.
  - **The ELF Format**
    - ELF Header
      - Metadata like architecture, entry point, and file type
      - Offsets to the section header table and program header table
    - Program Header Table (PHT) for segments
      - Defines segments, which are runtime memory mappings
      - Used by the loader to map the executable into the process's virtual address space.
      - Each entry in the table describes a segment (e.g., code, data, stack).
      - Segments are used at runtime and determine how the binary is mapped into the process's virtual memory.
    - Section Header Table (SHT) for sections
      - Defines sections, which store file components for linking and debugging.
      - Sections are not used at runtime but at compile/link time.

- **Q4: What are the steps involved in translating a program written in a high-level language to an ELF file?**
  - The compiler is responsible for turning a program written at a high-level language, with expressions such as "x+= 1" into architecture-specific asm instructions such as "add $1, %eax".
  - Since pure executable instructions are not enough context to inform the loader how to load an executable in memory exactly, there are a few tools and steps involved for going from instructions to an executable (e.g., an ELF file).




  - **Preprocessor**
    - Input: C source code (.c) → Output: Preprocessed C code (.i)
    - Task: Expands macros, includes headers, and processes preprocessor directives.

- **Compiler**
  - Input: Preprocessed C code (.i) → Output: Assembly code (.s)
  - Task: Translates high-level code into architecture-specific assembly instructions.
  - We need to know a few important **architecture-specific conventions**
    - What registers does the compiler use for **argument passing** during function calls? How many, and in what order?
    - What happens to the stack when you call/return to/from a function?
    - What registers are **caller-saved** (the callee function may overwrite them) vs. **callee-saved** (the callee must save and restore these registers to preserve important values across calls)?

```c
extern void foo(int);

int foo2(int a){
   a += 1;
   return a;
}

void main(void){
   foo(1);
   foo2(2);
}
```

gcc -S hello.c →

➜ hello.s (aarch64)

```
foo2:
   sub   sp, sp, #16    // Allocates stack space in the new stack frame (with 16-bytes alignment)
   str   w0, [sp, 12]   // Stores the input argument (int) at offset 12 from sp
   ldr   w0, [sp, 12]   // Loads the int value into the w0 register
   add   w0, w0, 1      // Performs calculation
   str   w0, [sp, 12]   // Stores the result of the calculation back to the current stack frame
   ldr   w0, [sp, 12]   // Loads the result of the calculation to the w0 register (return value)
   add   sp, sp, 16     // Restores the previous stack pointer
   ret                  // Returns to the caller (address from stack)

main:
   stp   x29, x30, [sp, -16]! // Saves the caller's stack frame base pointer (x29) and the
                              // return address (link register: x30) on the stack, and adjusts the stack pointer
   mov   x29, sp              // Sets up a new stack frame (base pointer now points to the top of the stack)
   mov   w0, 1                // Puts 1 (argument for foo) in w0 (lower 32 bits of the x0 register)
   bl    foo                  // Calls foo(1)
   mov   w0, 2                // Puts 2 (argument for foo2) in w0
   bl    foo2                 // Calls foo2(2)
   ldp   x29, x30, [sp], 16   // Restores the caller's stack frame base pointer and return address
   ret                        // Returns to the caller (address from link register)
```

```c
extern void foo(int);

int foo2(int a){
   a += 1;
   return a;
}

void main(void){
   foo(1);
   foo2(2);
}
```

gcc -S hello.c →

➜ hello.s (x86_64)

```
foo2:
   push  rbp                    ; Saves the caller's stack frame base pointer on the stack
   mov   rbp, rsp               ; Sets up a new stack frame (base pointer now points to the top of the stack)
   mov   dword ptr [rbp - 4], edi  ; Stores the input argument in the current stack frame
   mov   eax, dword ptr [rbp - 4]  ; Loads the value from the current stack frame into the eax register
   add   eax, 1                 ; Performs calculation
   mov   dword ptr [rbp - 4], eax  ; Stores the result of the calculation back to current stack frame
   mov   eax, dword ptr [rbp - 4]  ; Loads the result of the calculation from the stack to the eax register
   pop   rbp                    ; Restores the caller's stack frame base pointer
   ret                          ; Returns to the caller (address from stack)

main:
   push rbp          ; Saves the caller's stack frame base pointer on the stack
   mov  rbp, rsp     ; Sets up a new stack frame (base pointer now points to the top of the stack)
   mov  edi, 1       ; Puts 1 (argument for foo) in the edi register
   call foo@PLT      ; Calls the function foo(1), through the Procedure Linkage Table (PLT).
   mov  edi, 2       ; Puts 2 (argument for foo2) in the edi register
   call foo2         ; Calls foo2(2)
   pop  rbp          ; Restores the caller's stack frame base pointer
   ret               ; Returns to the caller (address from stack)
```

- **Assembler**
  - Input: Assembly code (.s) → Output: Object file (.o)
  - For the assembler, code and data are two big char arrays.
  - Tasks
    - Converts assembly instructions into machine code.
    - For every "interesting" object (i.e., functions, variables, and labels) in the aforementioned char arrays, stores in the symbol table an entry with its name and offset relative to the respective section's start.
  - Since the assembler does not know the final memory addresses where code or data will reside in the process's address space, it assumes that each section (e.g., .text, .data) starts at address zero.
  - The linker later merges these arrays from different object files, resolves symbol references, and calculates final memory addresses.

```
➜ gcc -c hello.s
➜ readelf -r hello.o
```

# The per-section table of required relocations

**Relocation section '.rela.text' at offset 0x258 contains 2 entries:**

| Offset | Info | Type | Sym. Value | Sym. Name + Addend |
|---|---|---|---|---|
| 00000000002c | 000c0000011b | R_AARCH64_CALL26 | 0000000000000000 | foo + 0 |
| 000000000034 | 000a0000011b | R_AARCH64_CALL26 | 0000000000000000 | foo2 + 0 |

**Relocation section '.rela.eh_frame' at offset 0x288 contains 2 entries:**

| Offset | Info | Type | Sym. Value | Sym. Name + Addend |
|---|---|---|---|---|
| 00000000001c | 000200000105 | R_AARCH64_PREL32 | 0000000000000000 | .text + 0 |
| 000000000034 | 000200000105 | R_AARCH64_PREL32 | 0000000000000000 | .text + 20 |

```
➜ readelf -s hello.o
```

# The table of all global, exported symbols with offset and section number

**Symbol table '.symtab' contains 13 entries:**

| Num: | Value | Size | Type | Bind | Vis | Ndx | Name |
|---|---|---|---|---|---|---|---|
| 0: | 0000000000000000 | 0 | NOTYPE | LOCAL | DEFAULT | UND | |
| 1: | 0000000000000000 | 0 | FILE | LOCAL | DEFAULT | ABS | hello.c |
| 2: | 0000000000000000 | 0 | SECTION | LOCAL | DEFAULT | 1 | .text |
| 3: | 0000000000000000 | 0 | SECTION | LOCAL | DEFAULT | 3 | .data |
| 4: | 0000000000000000 | 0 | SECTION | LOCAL | DEFAULT | 4 | .bss |
| 5: | 0000000000000000 | 0 | NOTYPE | LOCAL | DEFAULT | 1 | $x |
| 6: | 0000000000000000 | 0 | SECTION | LOCAL | DEFAULT | 6 | .note.GNU-stack |
| 7: | 0000000000000014 | 0 | NOTYPE | LOCAL | DEFAULT | 7 | $d |
| 8: | 0000000000000000 | 0 | SECTION | LOCAL | DEFAULT | 7 | .eh_frame |
| 9: | 0000000000000000 | 0 | SECTION | LOCAL | DEFAULT | 5 | .comment |
| 10: | 0000000000000000 | 32 | FUNC | GLOBAL | DEFAULT | 1 | foo2 |
| 11: | 0000000000000020 | 36 | FUNC | GLOBAL | DEFAULT | 1 | main |
| 12: | 0000000000000000 | 0 | NOTYPE | GLOBAL | DEFAULT | UND | foo |

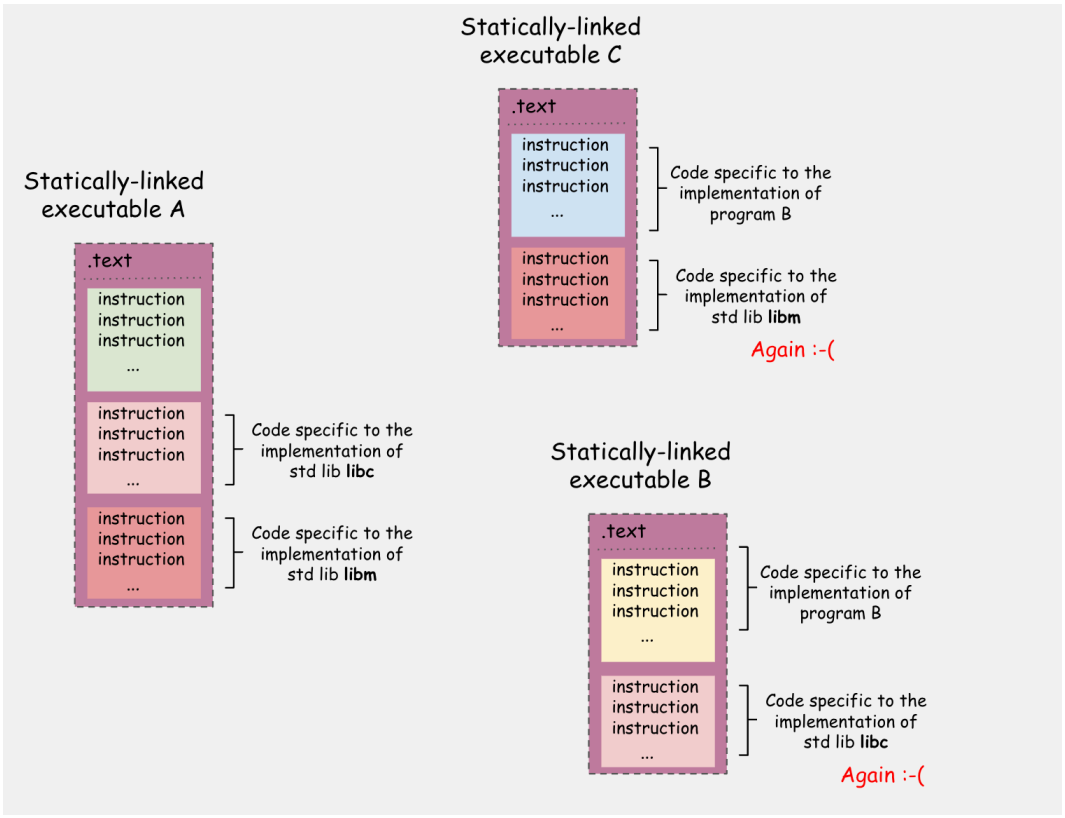
```
➜ nm hello.o
```

# If don't need full functionality of readelf, use nm

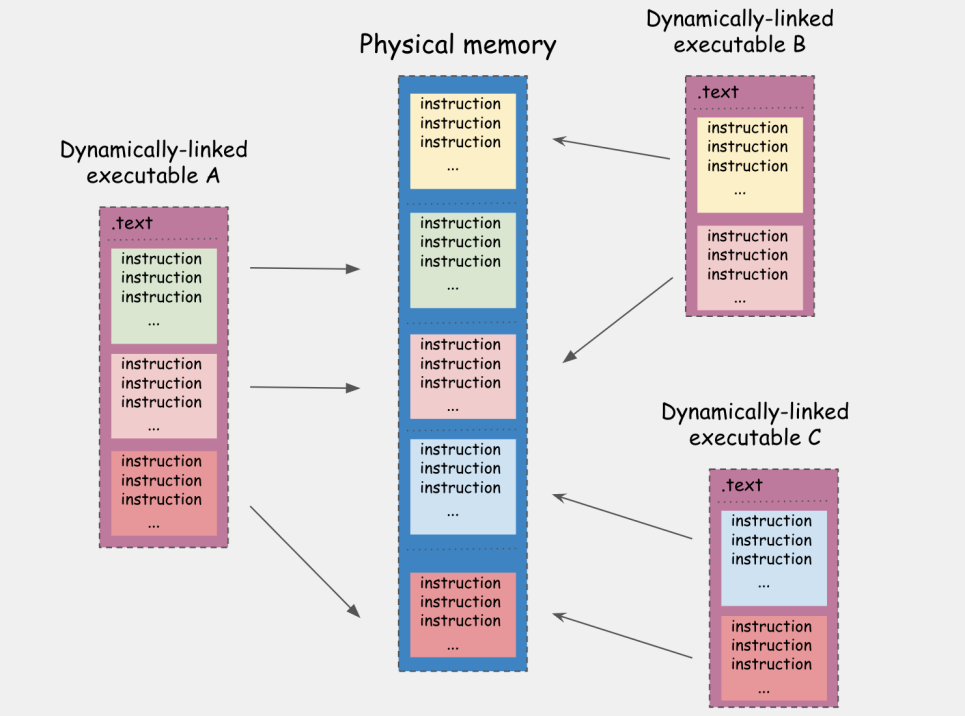
```
                 U foo
0000000000000000 T foo2
0000000000000020 T main
```

- **Linker**
    - Input: Object files (.o) and libraries → Output: Executable or library (a.out, .so)
    - The linker combines multiple object files into a final executable or library.
    - Tasks
        - **Symbol Resolution:** Object files contain undefined symbols (e.g., the call to foo, marked UND); The linker finds their definitions in other object files.
        - **Relocation:** Object files have relative addresses for functions/vars (printf is just a placeholder); The linker calculates actual memory addresses and updates references in the binary.
            - For Position Independent Code (**PIC**) and Position Independent Executables (**PIE**), the addresses are relative to where the program will be loaded at runtime.
            - For non-PIE executables, the linker assigns absolute addresses based on a fixed memory layout.
        - **Section Merging:** Combines sections (.text, .data, .bss, .rodata) from all object files and creates a new layout in the final executable.
        - **Statically linked** output files
            - Executable (a.out): Fully linked, ready to run.
            - Library code directly into the executable: Increases executable size but avoids runtime dependencies.
        - **Dynamically linked** output files
            - Shared Library (.so): The OS uses the dynamic linker to resolve symbols and load libraries at runtime, which reduces executable size and memory usage.

- **Q5: What are the differences between static and dynamic linking?**
  - Statically-linked executables can become too large and there is replication of code for common functionality (e.g., for standard libraries such as libc or libm).
  - Large in storage is OK this century. The problem is the wasted memory.




- **Desirable design:** Make standard libraries shared across executables.
- **How?** Dynamic linking
  - Recall the page table index: Every segment in the VAS of a process maps to a region in physical memory that holds the actual contents.
  - Make replicated sections point to the same regions in physical memory.
  - **Shared libraries** (e.g., .so files) contain reusable code that multiple programs can access at runtime.

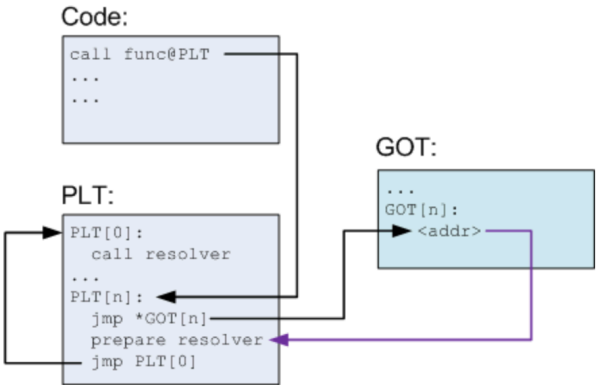



- Good idea. Implementation is another drama…
  - Shared libraries in Linux typically have the .so (shared object) extension.
  - These libraries contain reusable code that can be dynamically loaded at runtime by multiple programs simultaneously, which saves memory at runtime and disk space in storage.
- **Who finds the symbols at runtime?** In dynamically-linked executables, symbols are resolved dynamically by the dynamic linker.
  - The dynamic linker is specified in the .interpret section of the dynamically-linked executable.
  - In Linux, it is at /lib/ld-linux.so.2, or at /lib64/ld-linux-x86-64.so.2.

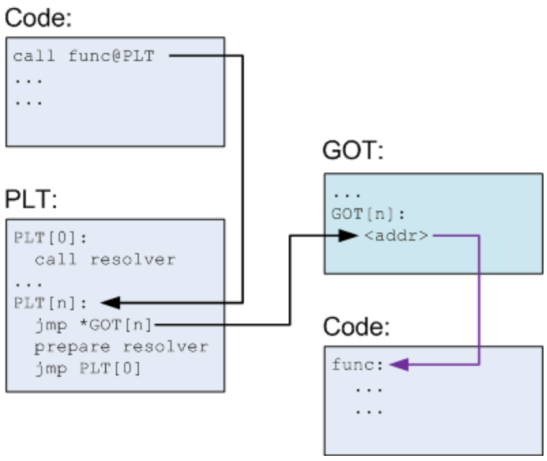
```
➜  ldd /bin/ls
linux-vdso.so.1 (0x0000ffff93223000)
libselinux.so.1 => /lib/aarch64-linux-gnu/libselinux.so.1 (0x0000ffff93160000)
libc.so.6 => /lib/aarch64-linux-gnu/libc.so.6 (0x0000ffff92fb0000)
/lib/ld-linux-aarch64.so.1 (0x0000ffff931ea000)
libpcre2-8.so.0 => /lib/aarch64-linux-gnu/libpcre2-8.so.0 (0x0000ffff92f10000)
```

  - The dynamic linker follows a specific order to locate required shared libraries.
    - 1. LD_PRELOAD environment variable: Forces the dynamic linker to load specified libraries before any others.
    - 2. LD_LIBRARY_PATH environment variable: Specifies directories to search before paths embedded in the binary..
    - 3. rpath: A path embedded in the binary at link time, used if runpath is not set.
    - 4. runpath: Another embedded path, searched after LD_LIBRARY_PATH.
    - 5. System cache (/etc/ld.so.cache): A lookup table maintained by ldconfig.
    - 6. Default system directories, such as /lib and /usr/lib.

- The dynamic linker uses a combination of two special tables that are laid out in each dynamically-linked executable.
    - The procedure linkage table (**PLT**) provides a mechanism for resolving external function calls dynamically.
    - The global offset table (**GOT**) stores the actual memory addresses of these external symbols.




    - When a program first calls an external function, the PLT stub triggers the dynamic linker to resolve the function's address, which is then cached in the GOT, allowing subsequent calls to use the resolved address directly and efficiently.




- Position Independent Code (PIC) allows executable code to run from any memory address by using relative addressing instead of absolute memory references. (See this amazing post on with lots of details.)