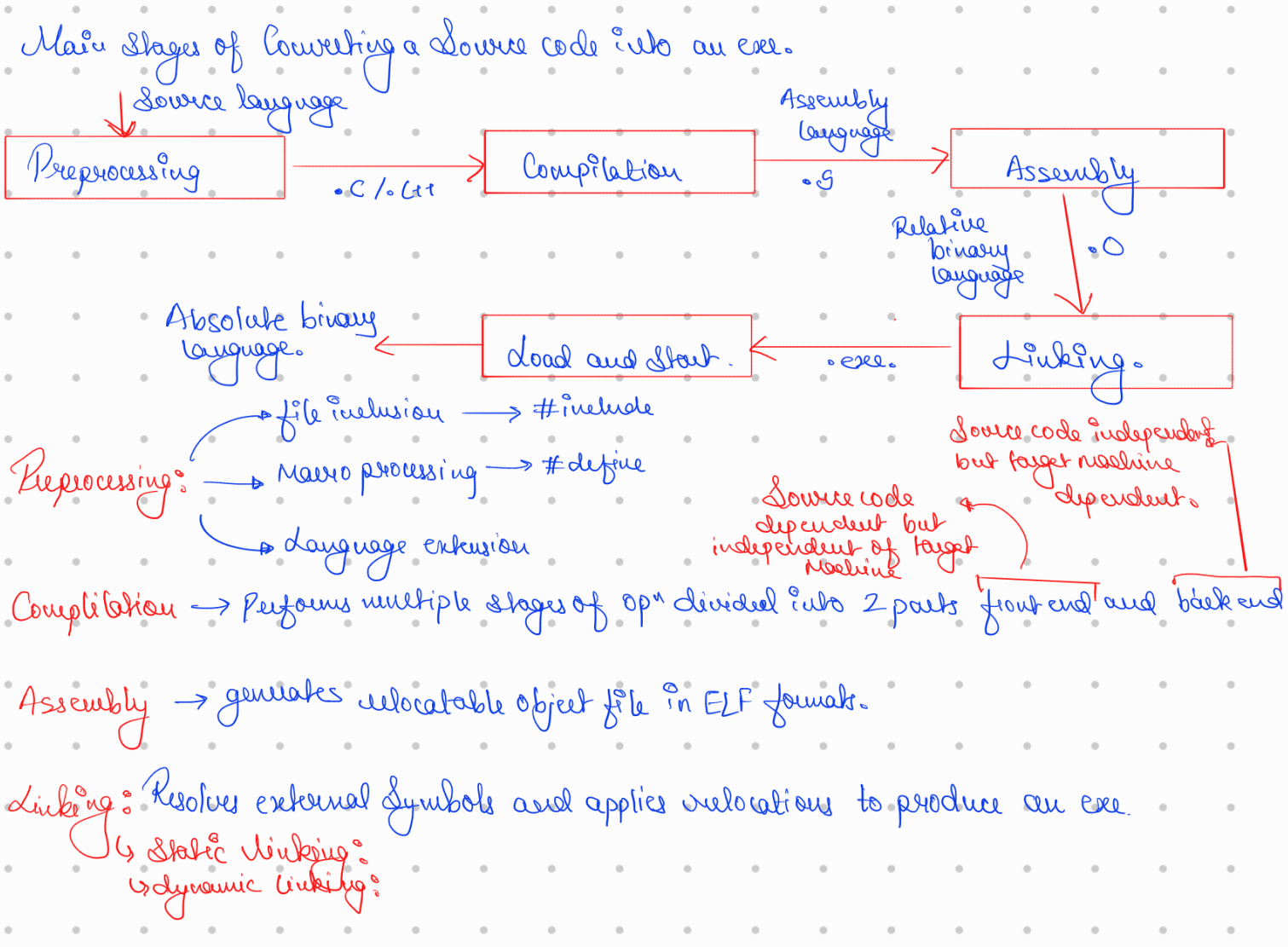
Q1: Roles — Compiler, Assembler, Loader, Linker

- Compiler: Translates high-level source (e.g., C) into target machine form through front-end analysis (lexing, parsing, semantic checks), optimization, and code generation, typically producing assembly or intermediate representation that follows the platform's calling conventions and ABI; it is part of a larger compilation system that also includes preprocessing and later stages (assembly and linking).[1] [2]
- Assembler: Converts assembly language emitted by the compiler into relocatable object files (ELF relocatable objects on Unix-like systems), encoding instructions, data, symbol tables, and relocation entries for later linking.[2] [1]
- Linker: Resolves symbol references across one or more object files and libraries, applies relocation to fix up addresses, and produces an executable (or shared object) with proper sections and segments ready for loading. It supports static and dynamic linking, symbol resolution rules, GOT/PLT for PIC, and library interpositioning.[1] [2]
- Loader: Part of the OS program loader that maps the executable's segments into memory (e.g., ELF program headers), initializes .bss, resolves dynamic dependencies (via the dynamic linker), sets up the initial stack, and transfers control to the program entry point.[2] [1]

Resources: CS:APP Chapter 1 §1.2–1.4 and Chapter 7; Lecture-22-08 (ELF, loader).[1] [2]

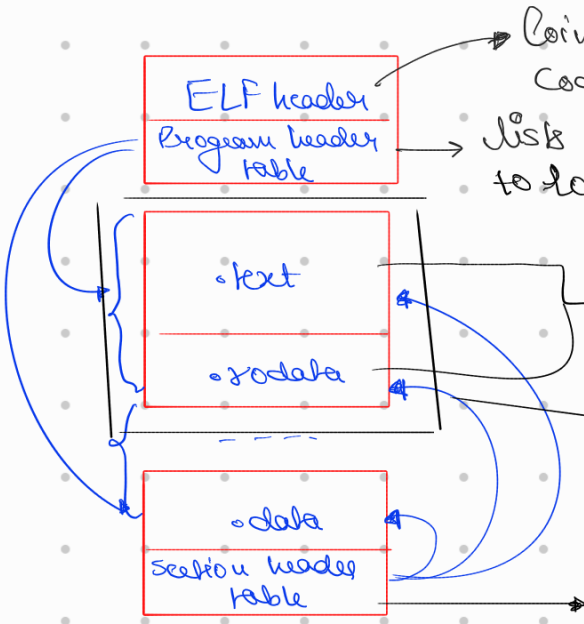


loader: loads the program into memory / basically allocates actual memory.
↳ places segments into memory and expands .bss file.

ELF format (Executable and linkable)

↳ It is basically a standard Linux/unix binary format that stores code and data in a structured way. so that assembler, linker and loader can function properly.
using the same header/section/segment conventions it can describe

① relocatable objects (.o)
② shared objects (.so)
③ executables. (.exe)



Contains entry point of the executable code

lists all the segments of the file. used to load and execute the program.

Sections: Pieces of code and data that gets linked together by compiler.

Segment: Contains one or more Sections

Used by linker

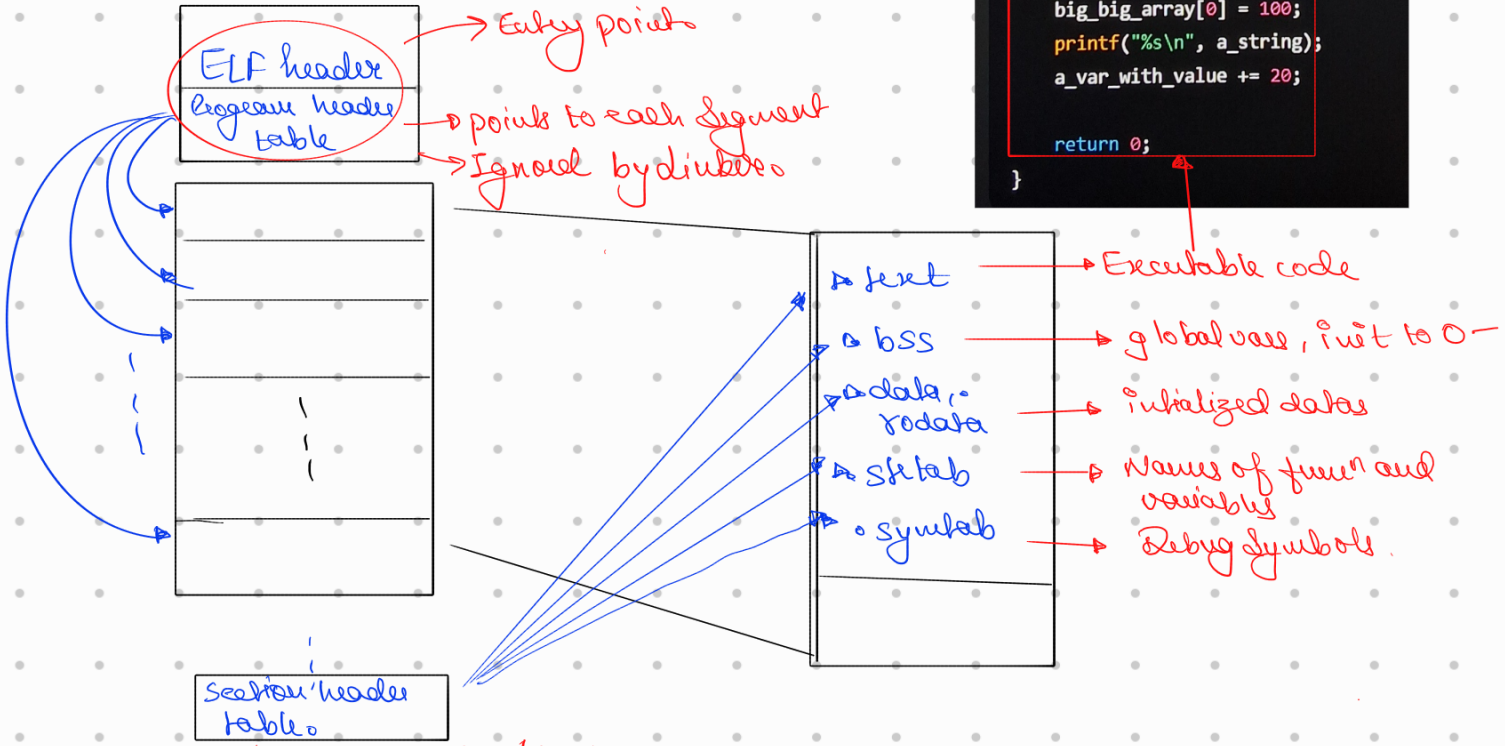
.rodata (constants)

.data (variable)

.bss

```c
#include <stdio.h>

int big_big_array[10 * 1024 * 1024];
char *a_string = "Hello, World!";
int a_var_with_value = 0x100;

int main(void) {
    big_big_array[0] = 100;
    printf("%s\n", a_string);
    a_var_with_value += 20;

    return 0;
}
```

we can basically define it as.



Entry point

points to each segment

Ignored by linker.

Executable code

global var, init to 0

initialized datas

Names of func and variables

Debug symbols

↳ ignored by OS./loader

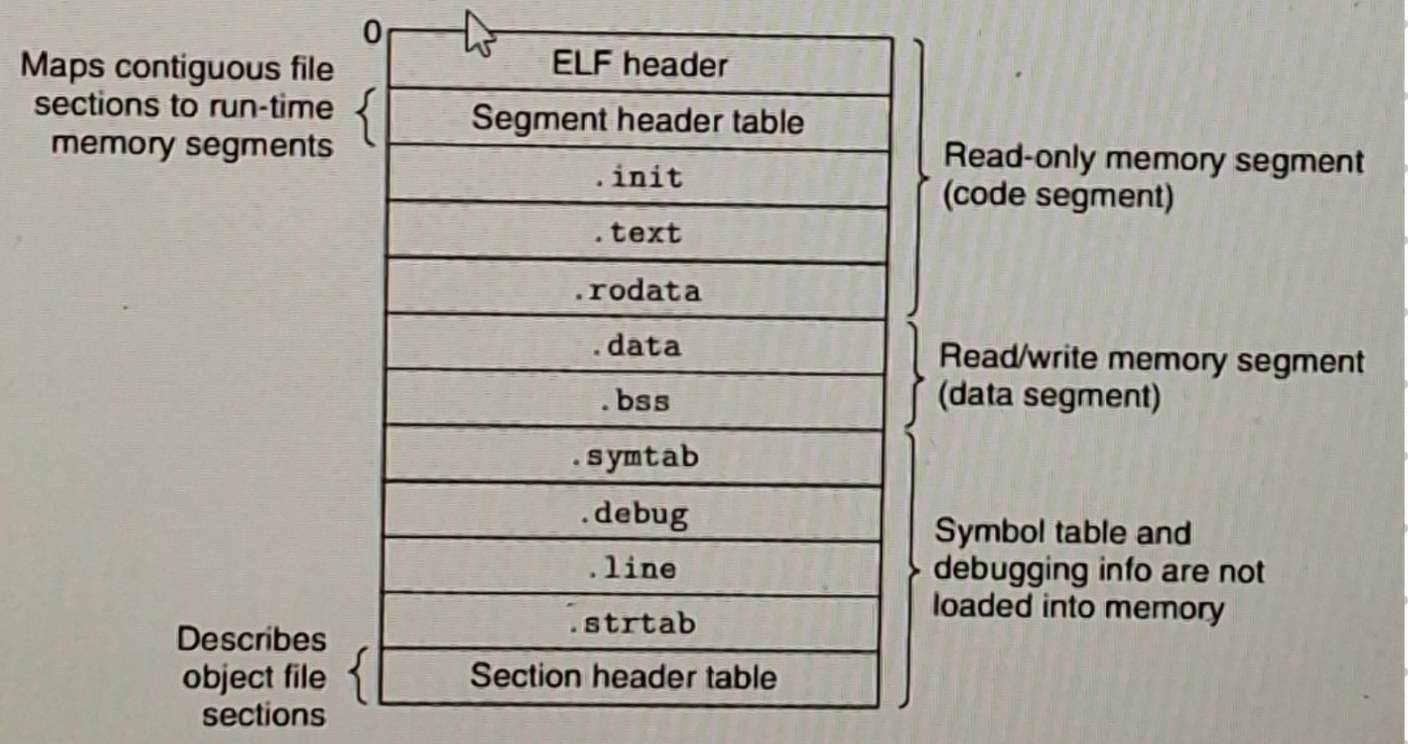


Figure 7.13 Typical ELF executable object file.

Q2: Compile-time vs Runtime Errors (with examples)

- Compile-time errors: Detected by the compiler or assembler before execution; include syntax errors, type mismatches, undefined identifiers, or invalid inline assembly syntax. Example: Missing semicolon or calling a function with wrong parameter types causes compilation to fail.[1]
- Runtime errors: Occur during execution; include segmentation faults, divide-by-zero traps, illegal instruction, null-pointer dereference, out-of-bounds access, and assertion failures. Example: Dereferencing a freed pointer triggers a fault handled as an exception/signal on Unix (SIGSEGV).[1]

Resources: CS:APP Chapter 1 (translation pipeline), Chapter 8 (exceptions, signals), Chapter 3 §3.10 (buffer overflows).[1]

Compile time error.
↳ found during parsing, Syntax, Semantic analysis or code optimization phase.
↳ binary cannot be produced until Corrected.
↳ ex: Syntax error, type mismatch etc.
↳ categories
① Syntax error : missing semicolon etc
② Type / Semantic error : incompatible types
③ Name / Symbol issue at run time : undefined reference to a function
④ Configuration / ABI mismatch : wrong object format Enforced by linker / loader.

c — Copy code

```c
#include <stdio.h>

void print_number(int x) {
    printf("Number: %d\n", x);
}

int main(void) {
    char *str = "Hello";
    print_number(str); // ❌ Passing char* instead of int
    return 0;
}
```

- **At compile-time, the compiler will emit an error/warning like:**

pgsql — Copy code

```
warning: passing argument 1 of ‘print_number’ makes integer from pointer without a cast
```

Runtime error

↳ found only when the program runs with specific inputs or env.

↳ the process starts but then

↳ may fail via traps/exceptions

↳ give incorrect results.

↳ categories (typical)

① Memory faults : dereferencing NULL or freed pointers

② Arithmetic faults : division by zero

③ Resource / OS errors : failed system calls

④ Stack errors : stack overflow.

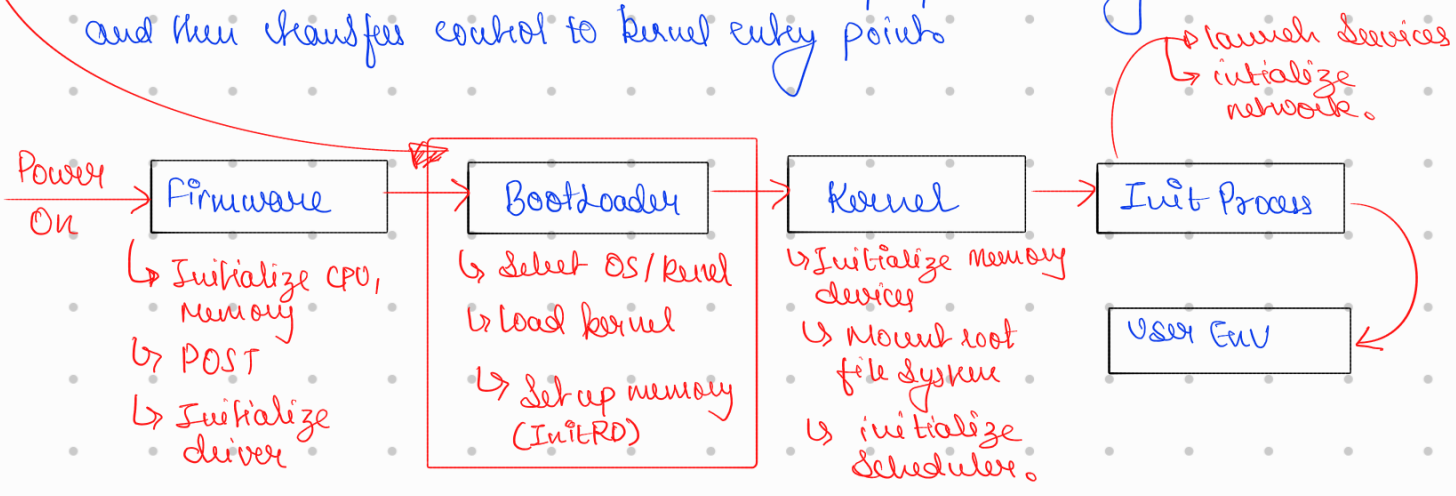
```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *ptr = malloc(0); // Could return NULL depending on system
    if (ptr == NULL) {
        printf("malloc failed, but we'll dereference anyway...\n");
    }
    *ptr = 42; // ❌ Dereferencing NULL → runtime segfault
    free(ptr);
    return 0;
}
```

Q3: Role of a Bootloader

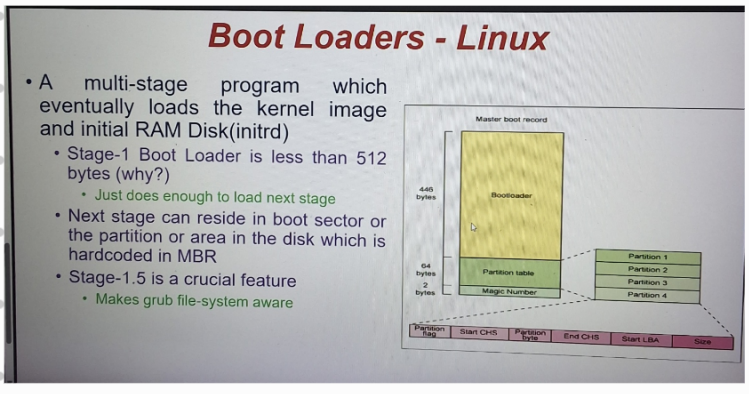
- A bootloader initializes minimal hardware, selects a bootable device, and loads the OS image and initramfs from disk into memory. In multi-stage designs (e.g., GRUB), stage-1 fits in MBR constraints (~512 bytes), stage-1.5 adds filesystem awareness, and stage-2 provides menu/config parsing; the bootloader then places the kernel and initrd, sets kernel parameters, and transfers control to the kernel's entry code.[2]
- It bridges firmware (BIOS/UEFI) and OS by setting up CPU mode, memory maps, and passing command line or device tree.[2]

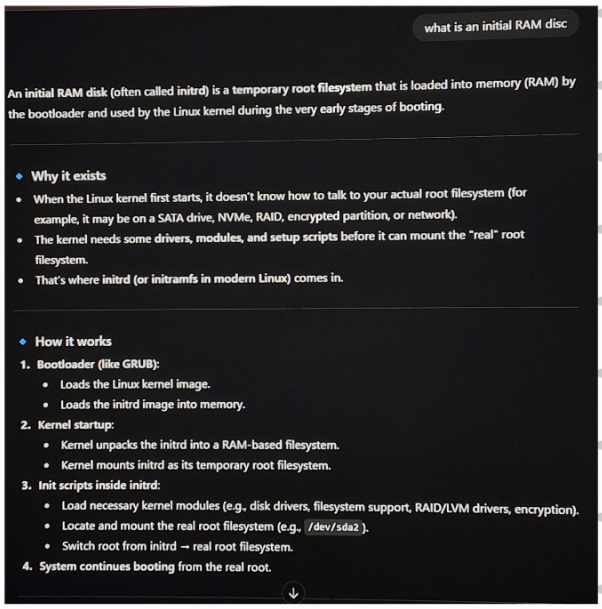
Resources: Lecture-22-08 (Boot Sequence, GRUB stages, kernel placement).[2]

Boot loader.
↳ Its job is to bridge the firmware and the OS
↳ It is the first software loaded by the firmware.
↳ It selects and loads the kernel, Initrd, prepares memory and CPU states and then transfers control to kernel entry points.



Q what boot loader sets up
↳ load kernel image to a suitable physical address and places the initial Ram Disk (InitRD) in memory.
↳ Boot Loader V/s program loader
↳ runs before any OS exists
↳ maps ELF Segments
↳ expands .bss
↳ relocates
↳ loads shared libraries.





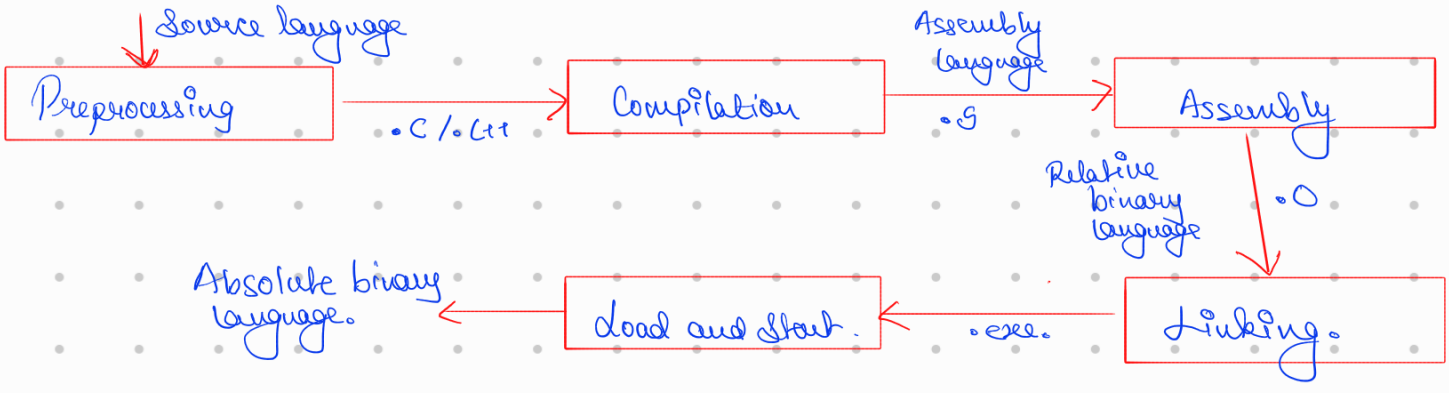
Q4: Interpreted vs Compiled Language (one key difference)

- Key difference: Execution model and timing of translation. Compiled languages (like C) are translated ahead-of-time into machine code and executed natively; interpreted languages (like Python) translate/execute at runtime via a VM/interpreter (and often a bytecode step), enabling portability and dynamic features at the cost of overhead.[1]
- Consequence: Compiled programs typically have faster steady-state performance; interpreted environments enable rapid development and introspection but rely on a runtime to execute code.[1]

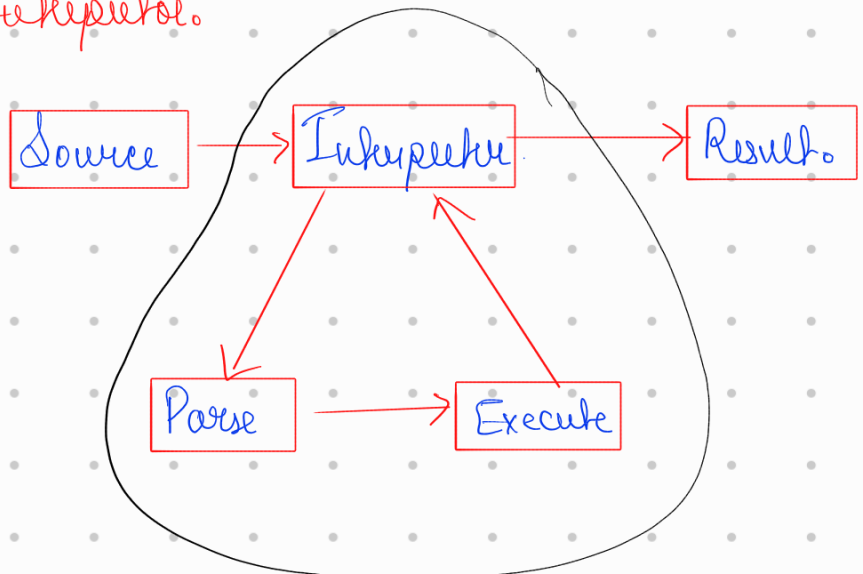
Resources: CS:APP Chapter 1 §1.2 (program translation forms) and §1.4 (execution).[1]

↳ flow of execution.

Compiler.



Interpreter.



↳ executed for every statement of the code.

Typical Headeoffs

① Performance
Compiler > interpreter.
↳ less translation overhead
↳ instructions are native

② Portability and flexibility
interpreter > Compiler
↳ dynamic

Given:

```
int g = 10; // global
int main() {
int x = 5;
char *p = malloc(100);
}
```

- g: Stored in the data segment (specifically .data since it has a non-zero initializer).[2] [1]
- x: Automatic local on the stack (lives in the current stack frame of main).[1]
- p: The pointer variable p itself is on the stack; the dynamically allocated 100 bytes are on the heap (managed by the allocator via brk/mmap).[1]

Resources: CS:APP Chapter 9 §9.8–9.9 (memory mapping, malloc), Lecture-22-08 (sections .text/.data/.bss, process memory layout).[2] [1]

Where can variable reside.

① Register (register keyword)
↳ nowadays register keyword is not used as CPU optimally uses register to store largely used datas.

② Stack (automatic storage) (auto keyword)
↳ local non static variables
↳ Scope is current block
↳ Lifetime: current block.

③ Static Storage (static keyword)
↳ initialized globals/statics → .data (data)
↳ zero initialized " " → .bss
↳ constants → .rodata.
↳ Lifetime: Program.

④ Heap
↳ dynamically allocated memory / memory allocated at runtime
↳ genally malloc/calloc
↳ Lifetime is till they are freed.

⑤ Shared libraries global.
↳ DSO/data or DSO/bss.

Storage classes, qualifiers, and effects

- auto: Default for block-scope variables; automatic storage on the stack; scope is the block; lifetime is the block's execution; rarely written explicitly in modern C. [attached_file:2 p.275]
- register: Hint to keep an automatic variable in a CPU register; compilers may ignore; same scope/lifetime as automatic; taking its address is disallowed by the language if truly in a register. [attached_file:2 p.287]
- static (at file scope): Gives the variable internal linkage (visible only in that translation unit) and static storage duration (program-long lifetime); placed in .data/.bss/.rodata depending on initialization/constness. [attached_file:4 p.6][attached_file:2 p.46]
- static (at block scope): Changes an otherwise local variable to static storage duration; a single instance persists across calls (initialized once); scope remains the block/function; typically in .data/.bss. [attached_file:2 p.46]
- extern: Declares but does not define; refers to a definition elsewhere; storage duration is static; linkage is external; placement per defining translation unit (.data/.bss/.rodata). [attached_file:2 p.715]
- const: Qualifier that forbids modification through that identifier; does not by itself change storage duration; many compilers place file-scope const objects in read-only segments (.rodata); lifetime follows storage class. [attached_file:4 p.7]
- volatile: Prevents optimization that would remove or reorder accesses; required for MMIO, signal-shared flags; does not change storage duration or placement. [attached_file:1 p.84]
- restrict (C99): Alias qualifier for pointers, enabling optimizations; no effect on placement, scope, or lifetime. [attached_file:2 p.538]

Q6: Static vs Dynamic Linking (with scenario)

- Static linking: The linker copies library code into the executable at link time, producing a self-contained binary; larger file size, no runtime dependency on shared libs, but no benefit from system library updates and potential duplication across processes.[1]
- Dynamic linking: The executable contains references resolved at load/run time by the dynamic linker; code is shared in memory across processes, smaller executables, and security/bug fixes in shared libraries apply without rebuilding the app. Uses PLT/GOT and PIC.[1]
- Prefer dynamic linking: For system tools using libc and other common libraries to reduce memory footprint and receive security updates transparently.[1]

Resources: CS:APP Chapter 7 §7.10–7.13 (dynamic linking, PIC, interpositioning).[1]

* Static linking → CSAPP (Chapter 7: 7.2, Page 708)
  - ↳ The linker pulls the required code from the static archives and writes it into the executable itself.
  - ↳ Symbol resolution and relocation are done at link time
  - ↳ Resulting files are larger but independent from external files at run time
  - ↳ Causes duplication.
  - ↳ Slightly faster calls and startup.

* Dynamic linking → CSAP (Chapter 7: 7.10, Page 734)
  - ↳ Executable contains relocation records and references
  - ↳ maps the shared objects (.so) at the time of loading the program
  - ↳ resolves the symbols lazily on first call.
  - ↳ has a smaller exe. Supports hot fixes
  - ↳ Requires loader
  - ↳ greater startup overhead
  - ↳ ABI mismatches can break if libraries change incompatibly.

Dynamic linking is preferred when System libraries frequently updates or Code Sharing matters

Static linking is preffered in minimal, single binary environment where reproduceable behaviour without external libraries is required

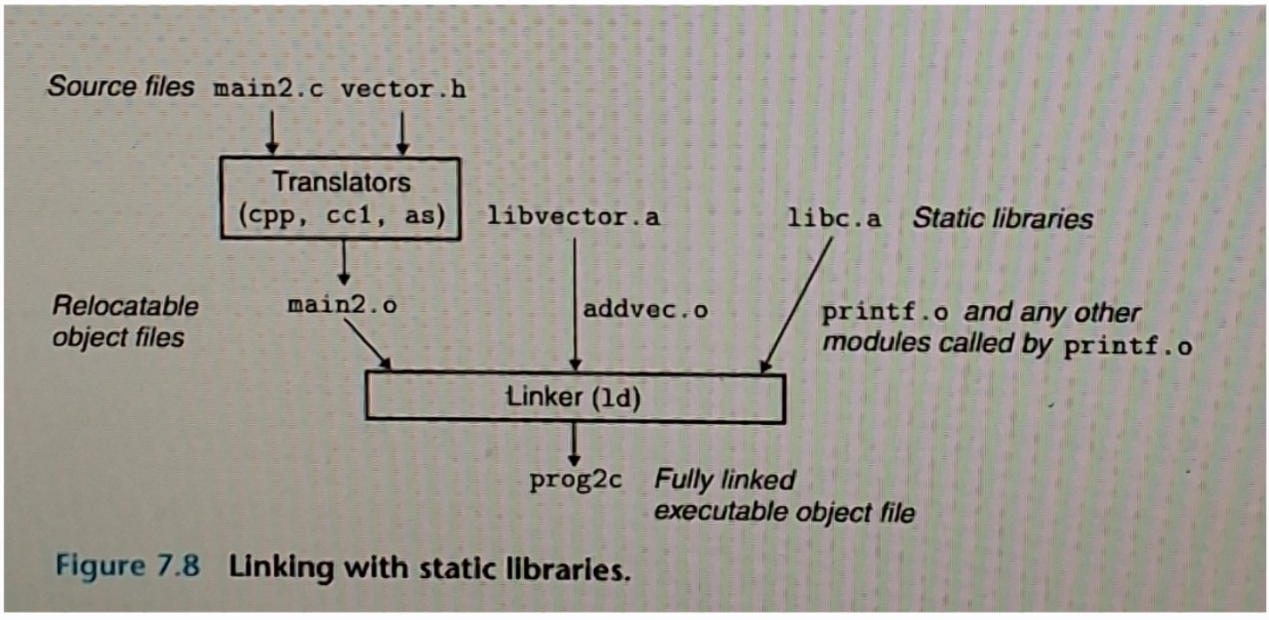
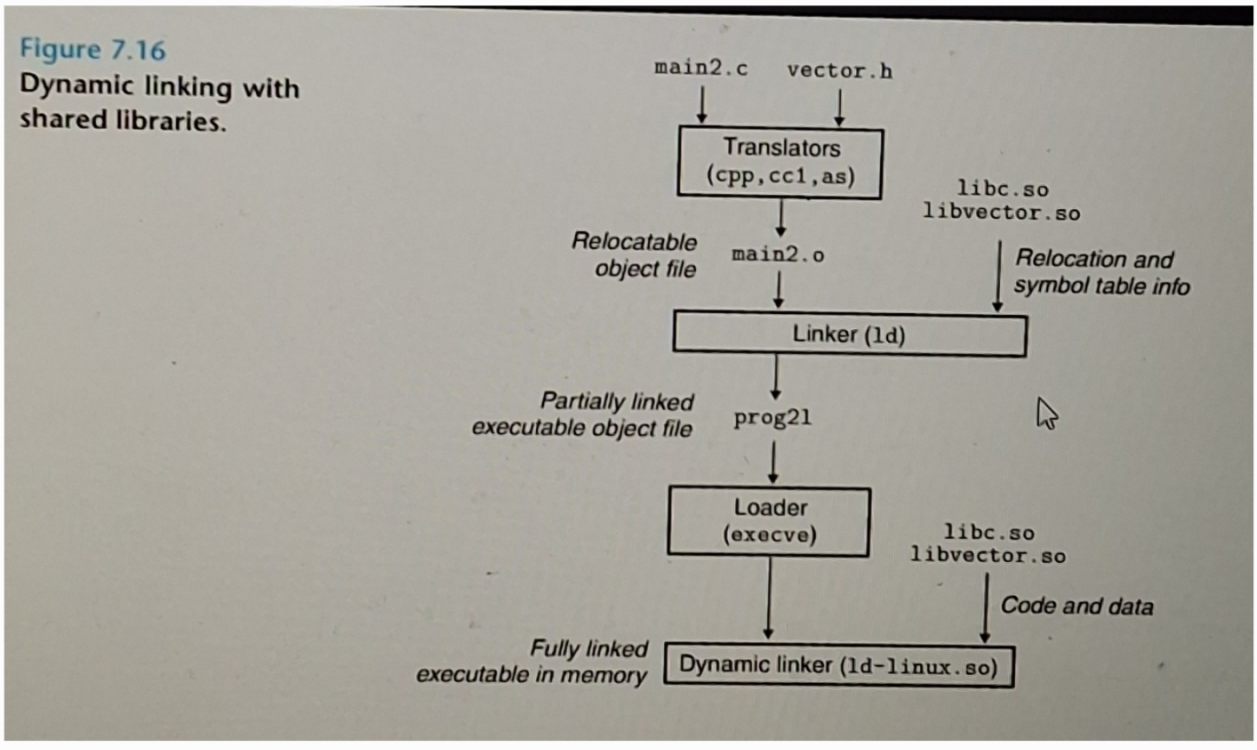


Figure 7.8 **Linking with static libraries.**



Real life examples of where what type of linking is preffered.

Static

↳ Embedded Systems.
  ↳ firmware running of MC
↳ Bootloader or kernels.
↳ where a certain version of a library is required.

Dynamic

↳ A large GUI framework.
  ↳ Instead of embedding large code every file share the code.
↳ where multiple programs may share a memory.

Q7: System Call Interface and Privileged Operations

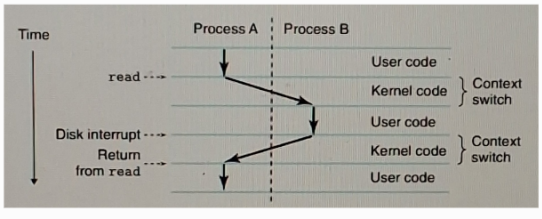
- System call interface: Provides a controlled gateway from user mode to kernel mode for privileged services (I/O, process control, memory management, networking). User programs issue syscalls (e.g., via syscall instruction) passing arguments; the kernel validates, executes in privileged mode, and returns a result or errno.[1]
- Why not direct privileged I/O: Hardware protection rings (user vs kernel) prevent untrusted code from manipulating devices or memory mappings directly, ensuring isolation, safety, and fairness; exceptional control flow and mode bits enforce this boundary.[1]

Resources: CS:APP Chapter 8 (exceptions, processes, user/kernel modes), Chapter 10 (I/O).[1]

System call Interface (SCI)

↳ A System call is the boundary b/w user space (where our code runs) and kernel space (where OS core runs).

↳ Applications cannot directly access hardware as that is Unsafe, instead they use these services by making a System call to the kernel



This is an interface b/w the software and hardware

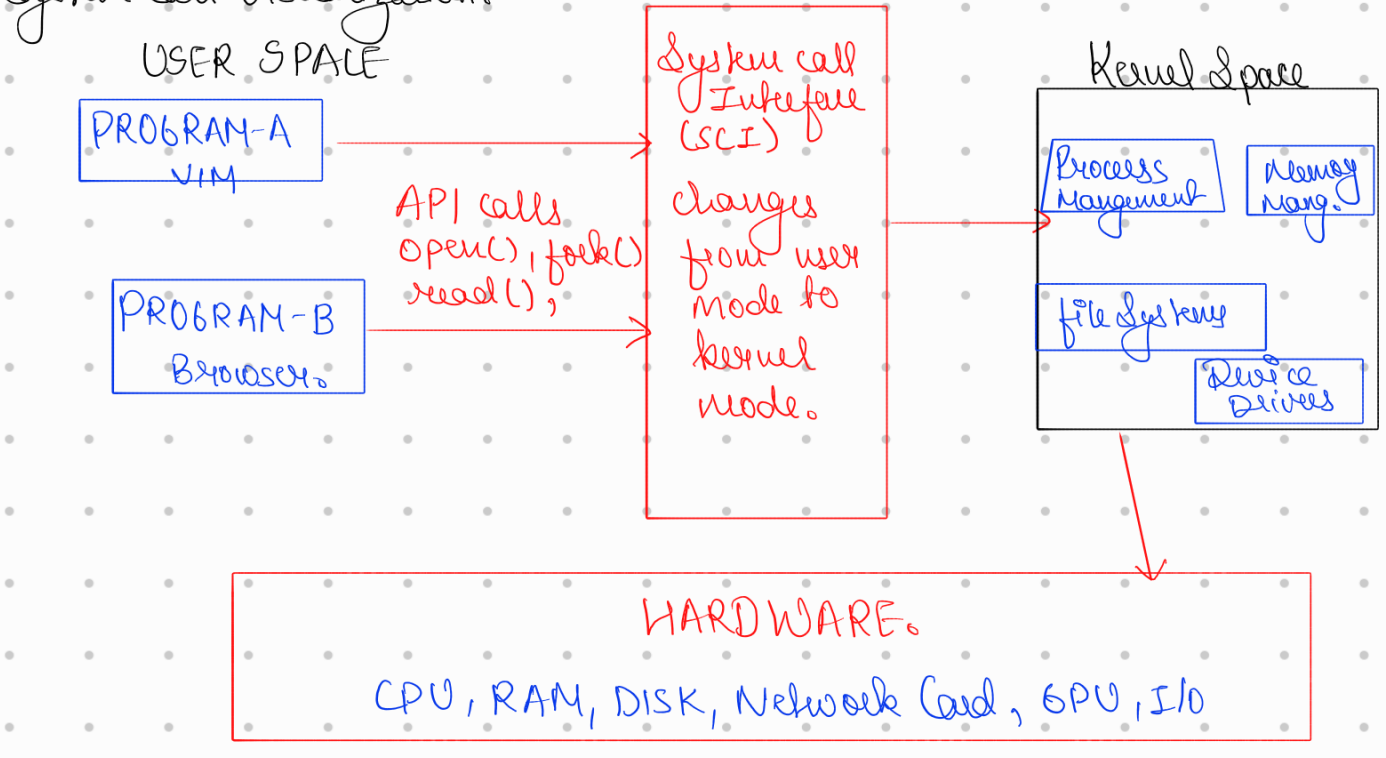
↳ Its responsibilities are

① Resource management

② Hardware abstraction (for apps that don't care you are intel or ARM)

③ Security and Isolation.

System call visualization.



Example: Writing to a File

1. User program calls:

```c
write(fd, buf, count);
```

2. The C library (glibc) translates that into a system call number (e.g., `SYS_write`).
3. CPU switches from user mode → kernel mode via a trap instruction (like `int 0x80` or `syscall`).
4. Kernel executes the write handler:
   - Finds the file in its table.
   - Writes data to disk (through driver).
5. Control returns back to user mode with success/failure code.

✅ So in essence:

- **Kernel** = the OS core that manages resources and hardware.
- **System Call Interface** = the *controlled door* that user programs must pass through to ask the kernel for services.

↳ Privileged Instructions are only executable in kernel mode. if a user tries them directly the CPU raises an exception and prevents the execution
↳ This ensures hardware protection against malicious code. How?

↳ If a System can directly perform I/O then following this are possible
① One program can write another programs memory
② disk sectors can be corrupted
③ whole OS can be deleted ultimately killing everything.
④ bypassing of Secret/important files.

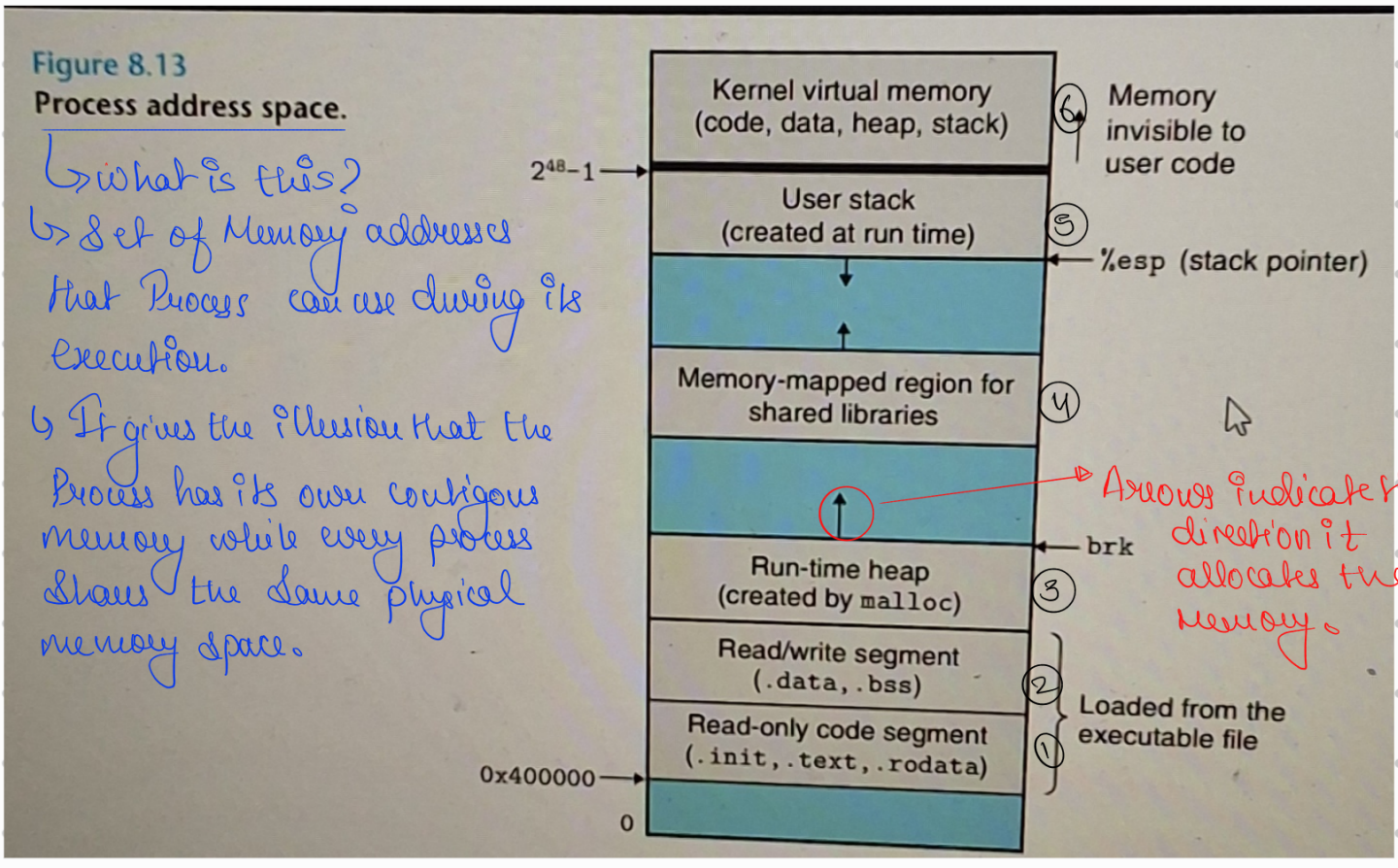
↳ no application requires hardware details like sectors, NIC registers and hence kernel provides uniform System Calls (API) and performs such tasks internally

Q8: Typical Process Address Space Layout (diagram description)

A typical 64-bit Unix process (high to low addresses):

- Kernel space (inaccessible in user mode)
- User space top
- Stack: grows down; holds frames, return addresses, locals
- Memory-mapped region: shared libs, mmap'd files, anonymous mappings
- Heap: grows up from end of .bss/.data; managed by malloc
- BSS: zero-initialized globals/static
- Data: initialized globals/static
- Text (.text): executable code (read/execute) [2] [1]

Resources: CS:APP Chapter 9 §9.8 (memory mapping, address spaces), Lecture-22-08 (ELF layout, loader, stack/heap). [2] [1]



Components of PAS

① Read only code Segment.
- ↳ Contains machine inst^{n} loaded from executables.
- ↳ no modification allowed during execution.
- ↳ Sub comp
  - ① init → Initialization Code
  - ② .text → actual program inst^{n}
  - ③ .rodata → read only constants. (Constant tables, String literals.

② Read write Segments.
- ↳ Contains global / static variables.
- ↳ These are updatable at runtime
- ↳ Sub Comp
  - ① .data → initialized globals / statics (int x = 5)
  - ② .bss → Uninitialized or zero initialized globals (int x;)

③ Run time heap.
- ↳ Region of Memory for dynamic allocation. (Malloc, calloc, new)
- ↳ grows upwards towards higher memory addresses.
- ↳ brk (program break) defines current end of heap

④ Memory mapped Region for shared libraries.
- ↳ used to load shared files / memory mapped files.
- ↳ grows upward.
- ↳ Enables loading of shared libraries to avoid duplication.

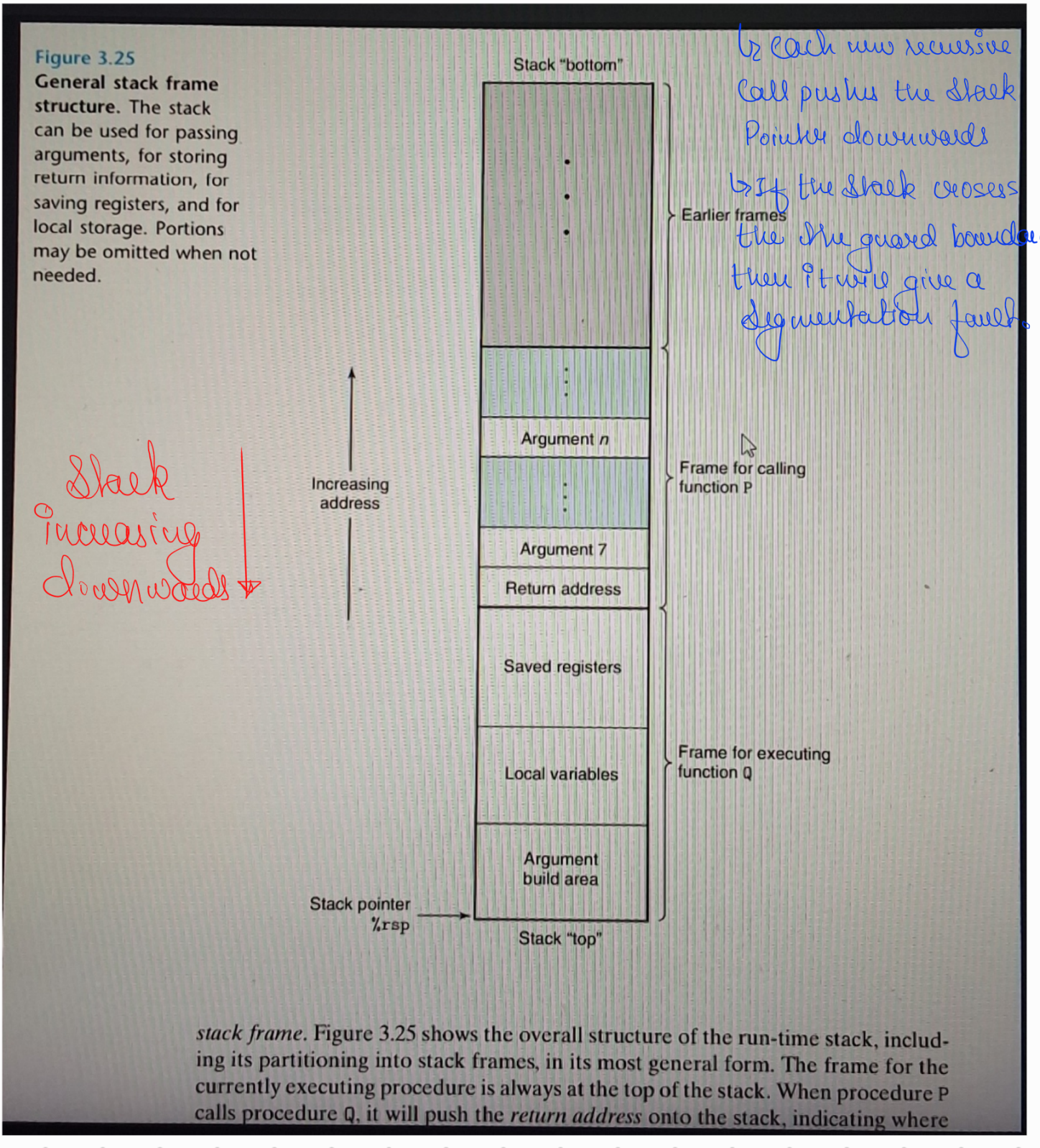
⑤ User Space → Decreasing address.
- ↳ created at runtime when the process starts
- ↳ used for func^{n} calls and flow control.
- ↳ stack pointer (%esp) is used to track top of stack.
- ↳ Each thread in a process has its own stack.

⑥ kernel Virtual memory.
- ↳ Contains kernel code, data, heap etc.
- ↳ invisible to user code.
- ↳ used for System calls, interrupt handling and device drivers. execution.

Q9: Recursion Exceeding Stack Size — What Error and Why?

- Error: Stack overflow leads to a protection fault (page fault) when the stack exceeds its guard pages/allocated region, typically delivered as a segmentation fault (SIGSEGV) to the process. The fault arises when the program attempts to access unmapped memory while growing the stack with deeper frames.[1]
- Root cause: Each recursive call consumes additional frame space; without base case or with large frames, the stack crosses guard boundaries.[1]

Resources: CS:APP Chapter 3 §3.7 (run-time stack), Chapter 8 (exceptions/signals), Chapter 9 (VM and page faults).[1]

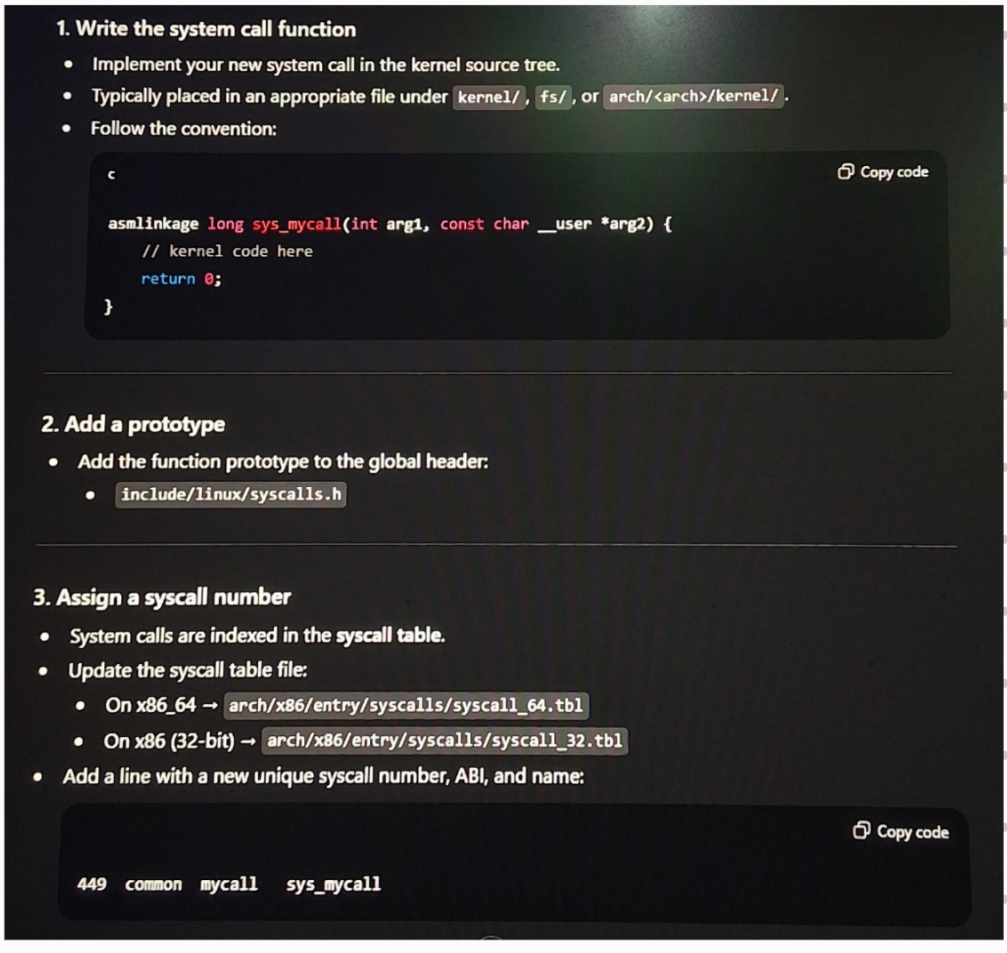


Q10: Steps to Add a New Linux System Call (high level)

- Kernel changes:
  - Implement the syscall handler function in the appropriate kernel subsystem directory, following kernel coding conventions and argument checking.[1]
  - Assign a syscall number by updating the syscall table for the target architecture (e.g., arch/x86/entry/syscalls/syscall_64.tbl) and add the prototype to the corresponding header (e.g., include/linux/syscalls.h).[1]
  - Wire up any required capability checks and documentation (man pages in section 2 usually come later).[1]
- Build and install:
  - Rebuild the kernel, install modules, update bootloader if necessary, and reboot into the new kernel.[2]
- User space:
  - Add a libc wrapper (optional until glibc adopts it). In the interim, use syscall(2) with the new number from user programs for testing.[1]

Resources: CS:APP Chapter 8 (syscall path, user/kernel boundary), Lecture-22-08 (boot/loader context for new kernels).[2] [1]

Steps to add a New Linux System Call.



4. Update user-space interface (optional)

- Define a wrapper function (e.g., in glibc) so user programs can call your syscall via a library API.
- Or call it directly using `syscall(SYS_mycall, ...)`.

---

5. Recompile the kernel

- Rebuild the kernel and modules:

```bash
make -j$(nproc)
make modules_install
make install
```

- Update the bootloader if necessary and reboot into the new kernel.

---

6. Test the new system call

- From user space:

```c
#include <unistd.h>
#include <sys/syscall.h>
#include <stdio.h>

#define __NR_mycall 449

int main() {
    long ret = syscall(__NR_mycall, 42, "Hello");
    printf("Return value: %ld\n", ret);
    return 0;
}
```