Operating System Concepts

Lecture 3: System Calls, Linking & Loading

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

Recap

- Definition: a trap or exception is a software-generated interrupt caused by an error (e.g., division by zero) or a request for OS services (**system call**)

0: 0x00080000	Illegal address
1: 0x00100000	Memory violation
2: 0x00100480	Illegal instruction
3: 0x00123010	System call

Trap Vector

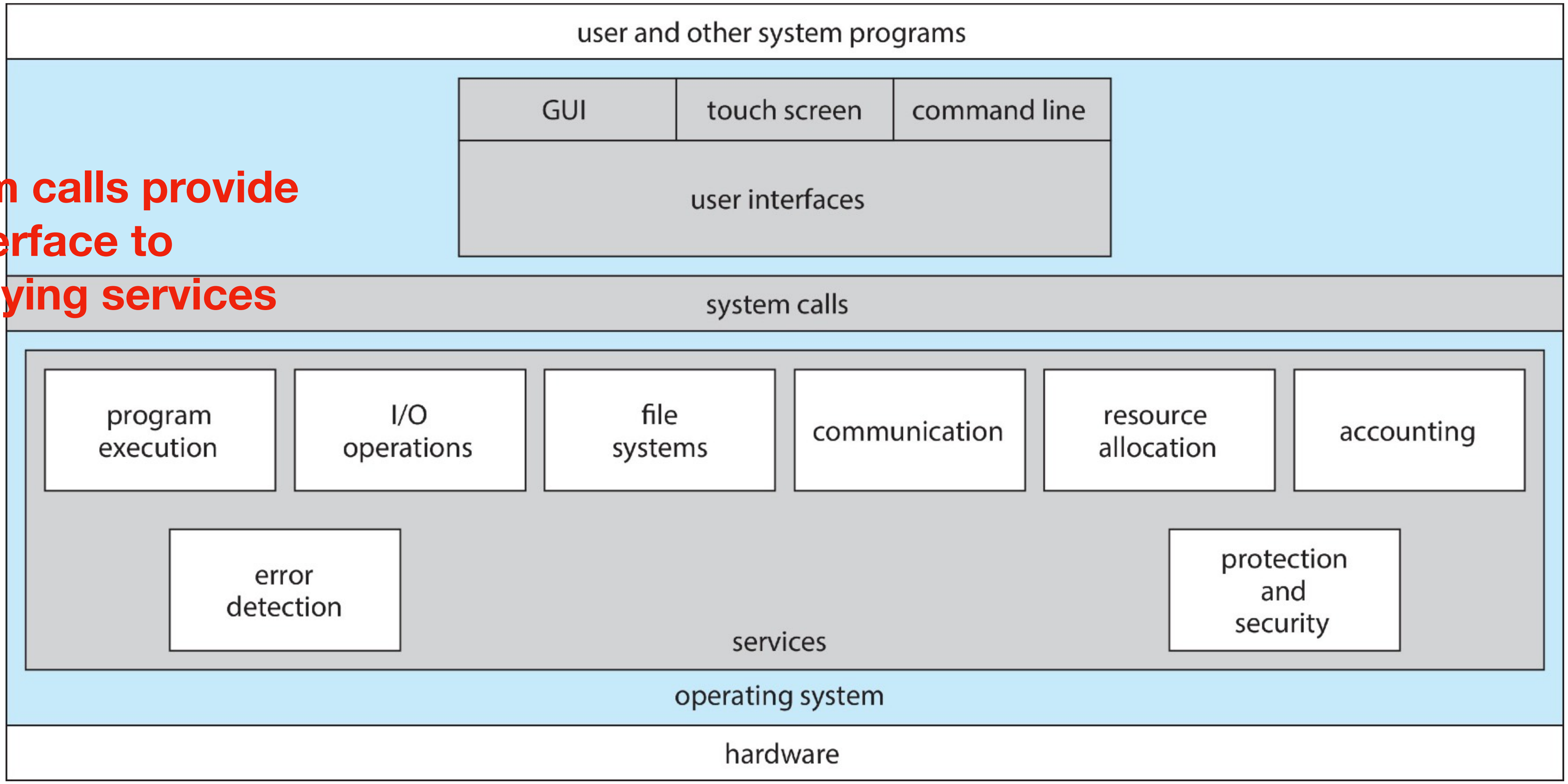
Today's class

- OS services
  - User interface
  - System calls: interface to OS services
  - Protection
- Basics of compiling, linking, and loading
- OS structure (time permitting)
  - Examples

OS Services

Operating System Services

system calls provide
an interface to
underlying services



- **Services:** user interface; program execution (loading and error handling); I/O, file system, and interprocess communication services, resource allocation (scheduling), logging, protection and security, etc.
- **Goals**:
  - making the programming task easier and increasing user convenience
  - ensuring the efficient operation of the system (i.e., resource allocation, logging, protection and security)

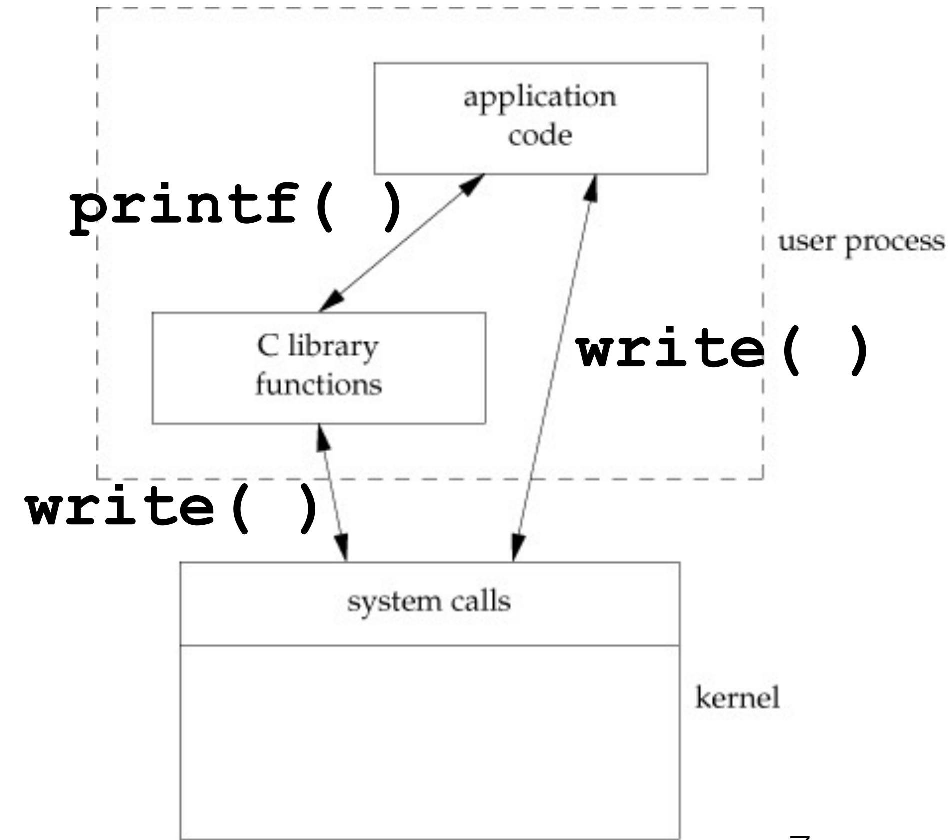
- ensuring the efficient operation of the system (i.e., resource allocation, logging, protection and security)
- making the programming task easier and increasing user convenience

System calls… what are they?

- System calls are services provided by the kernel to application programs
  - a typical OS exports a few hundred (~300-400) system calls
  - see Section 2 of the Linux manual page for the list of Linux system calls: https://man7.org/linux/man-pages/man2/syscalls.2.html
- System calls are necessary to access devices and files, request memory, set access permissions, stop and start processes, communicate with processes, set a timer, etc.

System calls… what are they?

- Each system call is typically defined as a function in the C library (e.g. glibc)
  - in UNIX-based systems, most system call wrapper functions are declared in `unistd.h`
- System calls can be invoked
  - via `syscall()` by providing their assembly language interface number, e.g. `syscall(SYS_write, …);` this is called a **raw system call**
  - indirectly via wrapper functions, e.g. `write(…)`, which may perform some pre/post-processing and error handling
- Many higher-level functions in the standard C library invoke system calls too
  - e.g. `printf(…)` from `stdio.h` calls `write(…)` from `unistd.h` after performing string formatting and type conversion



UNIX system calls

- Process control
  - `fork( ), exec( ), wait( ), exit( ), ...`
- Memory management
  - `brk( ), sbrk( ), ...`
- File/device management
  - `open( ), close( ), read( ), write( ), stat( ), lseek( ), link( ), ioctl( ), ...`
- Information maintenance
  - `getpid( ), sleep( ), time( ), …`
- Protection
  - `chmod( ), chown( ), ...`
- Communications
  - `pipe( ), shm_open( ), mmap( ), socket( ), accept( ), send( ), recv( ), ...`

Linux system calls can be found at:
http://man7.org/linux/man-pages/man2/syscalls.2.html

Tracing system calls executed when you run an application

Use the `strace` command in Linux (see the man page of `strace`)

```
oardakan@ug01:~>strace pwd
execve("/bin/pwd", ["pwd"], [/* 40 vars */]) = 0
brk(NULL)                               = 0x10d9000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=349772, ...}) = 0
mmap(NULL, 349772, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f270431d000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1868984, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f270431c000
mmap(NULL, 3971488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f2703d84000
mprotect(0x7f2703f44000, 2097152, PROT_NONE) = 0
mmap(0x7f2704144000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1c0000) = 0x7f2704144000
mmap(0x7f270414a000, 14752, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f270414a000
close(3)                                = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f270431b000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f270431a000
arch_prctl(ARCH_SET_FS, 0x7f270431b700) = 0
mprotect(0x7f2704144000, 16384, PROT_READ) = 0
mprotect(0x606000, 4096, PROT_READ)     = 0
mprotect(0x7f2704373000, 4096, PROT_READ) = 0
munmap(0x7f270431d000, 349772)          = 0
brk(NULL)                               = 0x10d9000
brk(0x10fa000)                          = 0x10fa000
open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=2981280, ...}) = 0
mmap(NULL, 2981280, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f2703aac000
close(3)                                = 0
getcwd("/cshome/oardakan", 4096)        = 17
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 4), ...}) = 0
write(1, "/cshome/oardakan\n", 17/cshome/oardakan
)       = 17
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

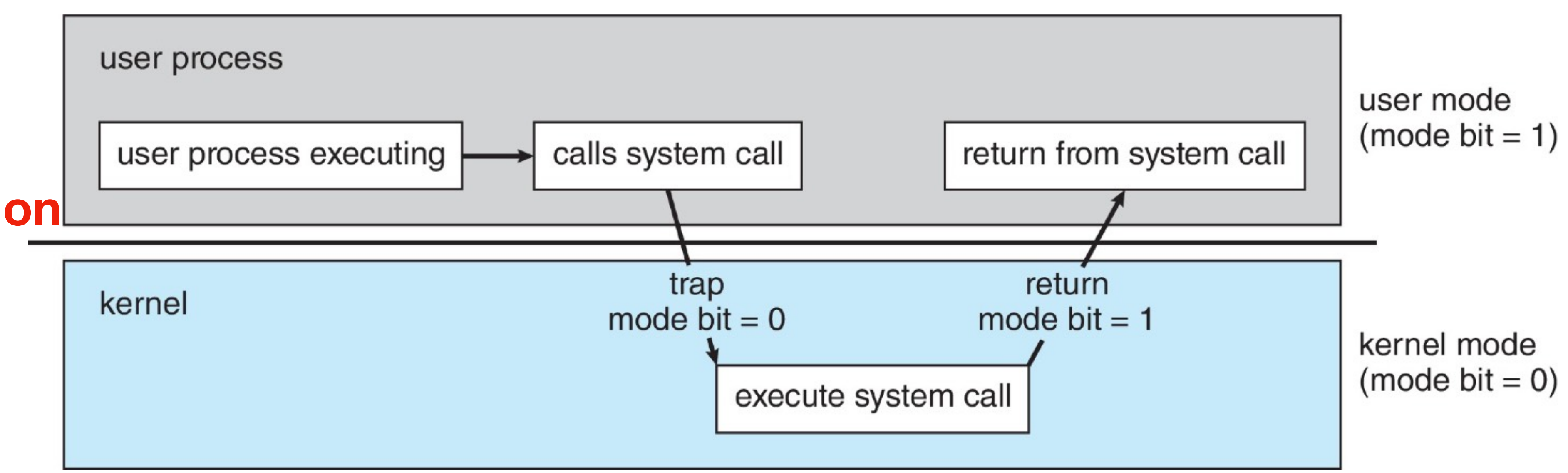
POSIX Application Programming Interface (API)

- Application programmers usually use functions from an API rather than invoking system calls directly
  - **Potability**: UNIX-based operating systems could have different declarations and implementations for their system calls, but they all support the same **POSIX API**; hence if you use the POSIX API you can expect that your program compiles and runs on any system that supports it
  - **Ease of use**: system calls are often more detailed and more difficult to use than their corresponding API functions
- **POSIX** (Portable Operating System Interface) is a family of standards implemented primarily for UNIX-based operating systems
  - POSIX compliant systems like Linux and macOS must implement the POSIX core standard (POSIX.1)
    - Linux man page says if a system call is specified in POSIX standards
  - POSIX thread library (aka pthreads) is defined in an extension of this API known as POSIX.1c
- **Win32 API** is the standard API for Windows operating systems

Protection bit — recap

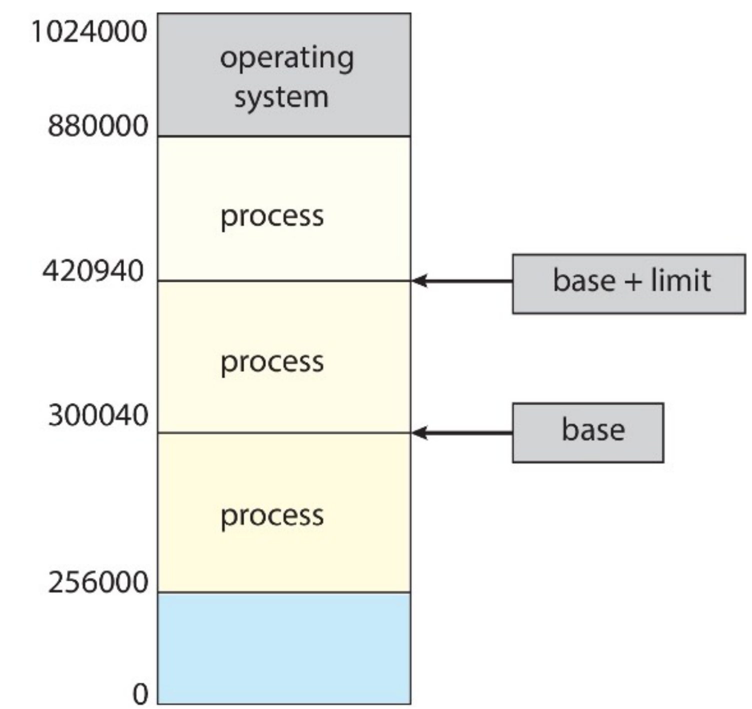
- Hardware has a status bit that indicates the current mode (user or kernel)
  - there could be more than two operation modes
    e.g., ARMv8 systems have seven modes
  - user programs run in the user mode
  - kernel code runs in the kernel mode with **full privileges** of hardware
- Operation modes provide the means for protecting OS from errant users
  - code could be buggy or malicious
- Examples of privileged instructions are I/O control, timer management, interrupt management; they can only run in the kernel mode
- Invoking a system call allows user program to run privileged instructions

crossing protection boundary



Memory protection

- Hardware must provide support so that the OS can
  - protect user programs from each other
  - protect the OS from user programs
- The simplest technique is to use base and limit registers
  - base (**relocation**) and limit registers are loaded by the OS before context switching
  - base register holds the smallest legal physical memory address of the process
  - limit register holds the size of the memory allocated to a process
  - CPU checks each reference in user mode (instruction and data addresses) to ensure that it falls between base and base+limit



Timer interrupt - Another form of protection

What if the process running on the CPU does not wait for I/O or signals?

- Kernel protects CPU from being hogged using timer interrupts that occur at regular intervals (e.g., every 100 microseconds)
  - frequency is set by the kernel
  - it’s yet another protection mechanism
- At each timer interrupt, the kernel chooses a new process to execute on CPU (scheduling)
- Interrupts can be temporarily deferred (it is crucial for implementing **mutual exclusion**) but user programs do not run with enough privilege to defer timer interrupts

0: 0x2ff080000	keyboard
1: 0x2ff100000	mouse
2: 0x2ff100480	timer
3: 0x2ff123010	Disk 1

Interrupt Vector

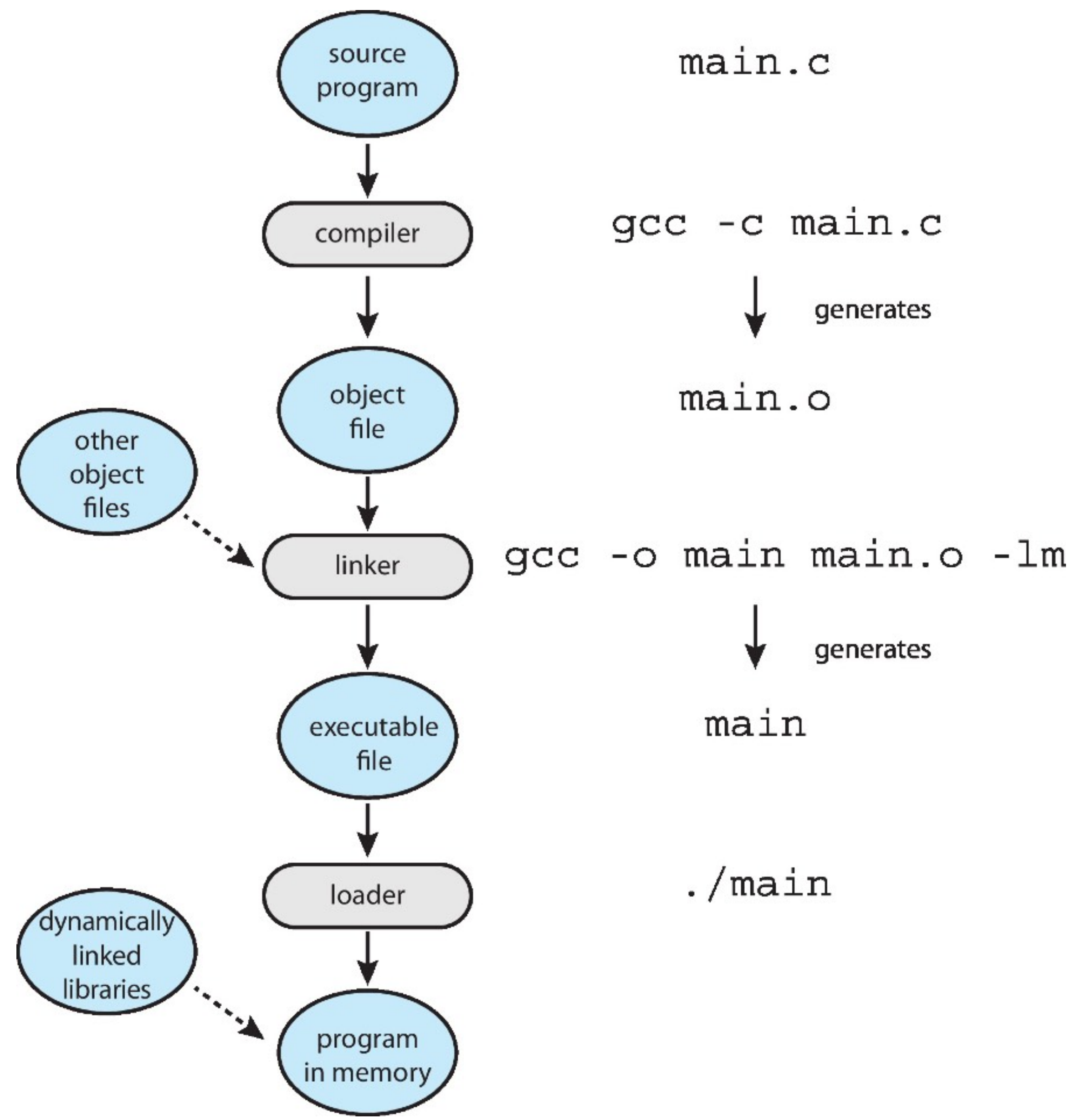
Summary

OS Services	**Hardware Support**
Interrupts	interrupt controller, interrupt request lines, interrupt vector
System calls	trap vector
I/O	interrupt and memory mapping
Protection	operation modes, privileged instructions, base and limit registers, timer interrupt
Scheduling & accounting	timer
Synchronization	atomic operations
Virtual memory	memory management unit (MMU), translation look-aside buffer (TLB)

Running a user program

From Program to Process

- Compiling is the process of converting a source file (ASCII code) into an object file
  - object files miss certain information, such as functions declared in other files and libraries
- Linking is the process of combing relocatable object files and specific libraries into a single binary executable file
- Loading is the process of bringing this executable file into memory
  - the loader is executed when you enter the name of the executable file on the command line; this is done using the `execve( )` system call
  - the loader sets up the process memory to contain code and data from executable
  - a library can be conditionally linked and loaded if it is required during the run time; this can be done using dynamically linked libraries (DLLs)



Executable and Linkable Format (ELF)

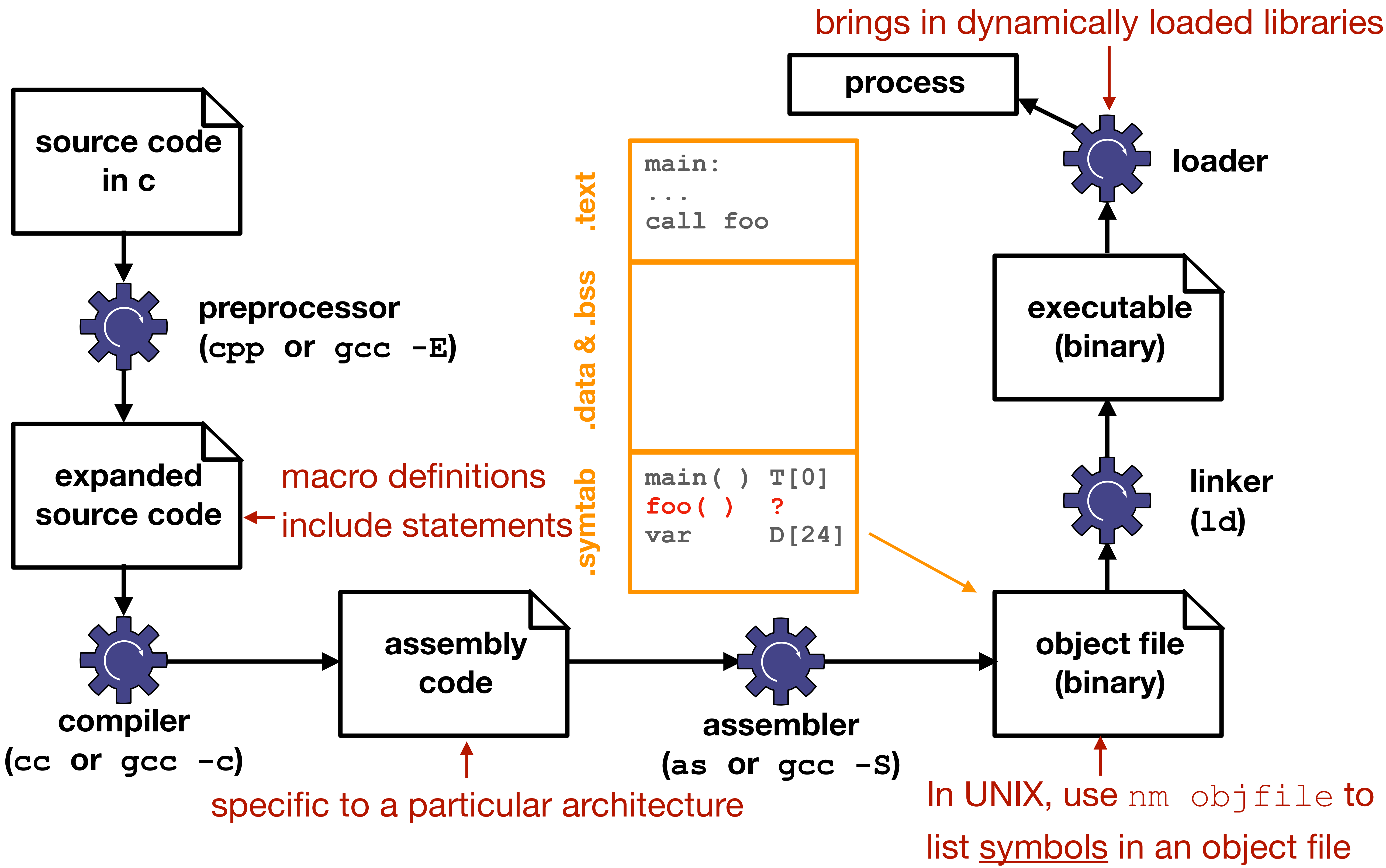
- Executable files must have a standard format, so that OS knows how to load and start them
- ELF is a common standard across modern UNIX operating systems, such as Linux
  - ELF header starts with a 4-byte magic string: 0x7F followed by 0x45 0x4C 0x46 (ELF in ASCII)
- **ELF relocatable file** contains the compiled machine code and a symbol table containing metadata about functions and variables referenced in the program
  - the symbol table is necessary for linking the relocatable object file with other object files
- **ELF executable file** contains the address of the first instruction of the program (program's **entry point**: _start function)
  - this file can be loaded into memory

Executable and Linkable Format (ELF)

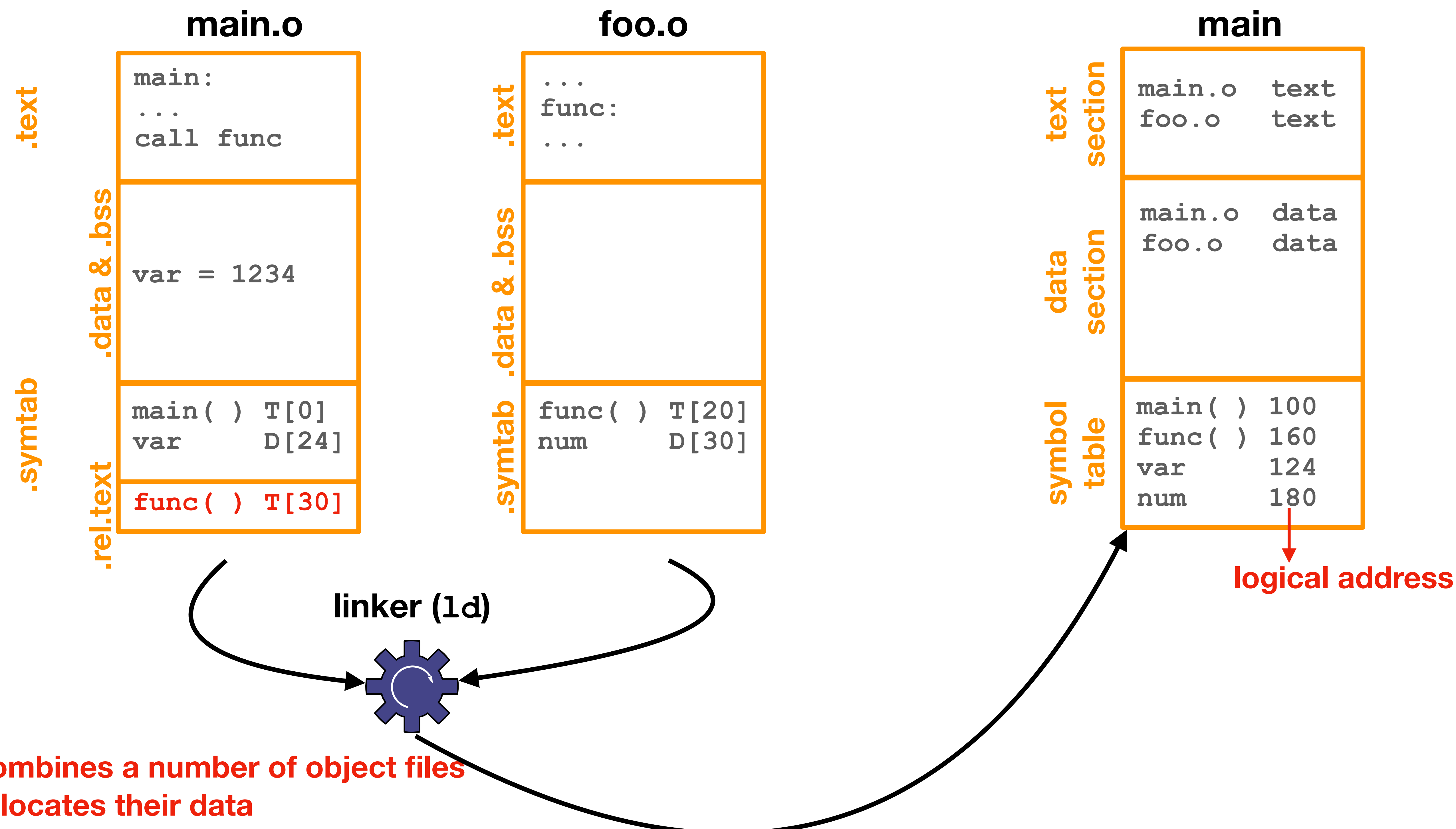
ELF file is divided into multiple sections

- **.text** contains the machine code
  - In UNIX-based systems, see the content of this section using `objdump -drS objfile`
- **.data** contains initialized global variables
- **.bss** (block storage start) contains uninitialized global variables
  - occupies no space in the object file actually
  - there is no point storing more zeroes on your disk
- **.rodata** contains read-only data such as constant strings
  - e.g., the format strings in printf statements
- **.symtab** contains the symbol table
  - information about functions and global variables defined and referenced in the program
- **.rel.text** and **.rel.data** contain relocation information for functions and global variables that are referenced but not defined (external references)
  - linker modifies this section when combining the object files, resolving external references

Compiler, Linker and Loader in action



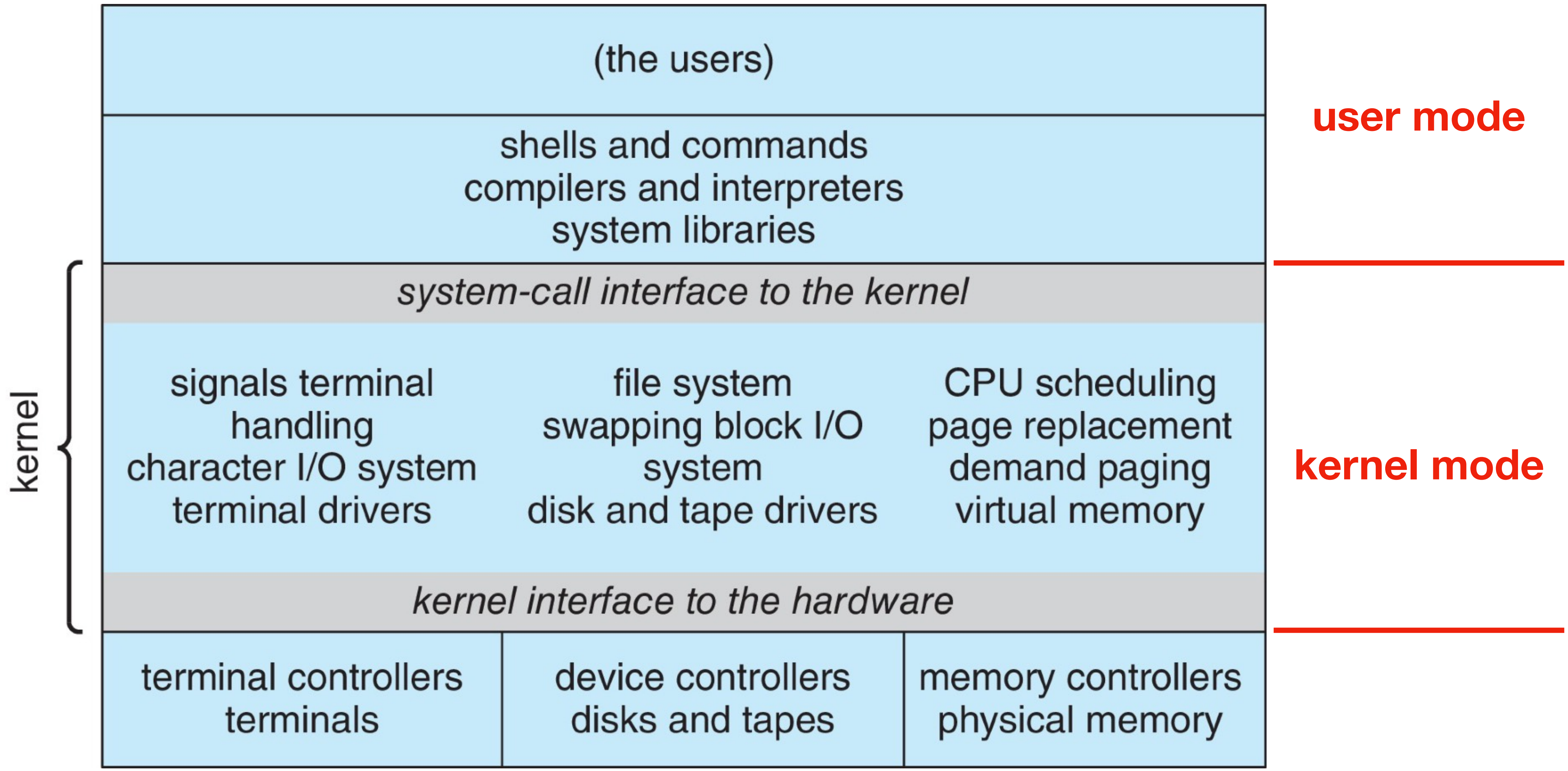
Linking



1. combines a number of object files
2. relocates their data
3. ties up symbol references (symbol resolution)

OS Structure

UNIX system has a *monolithic* structure

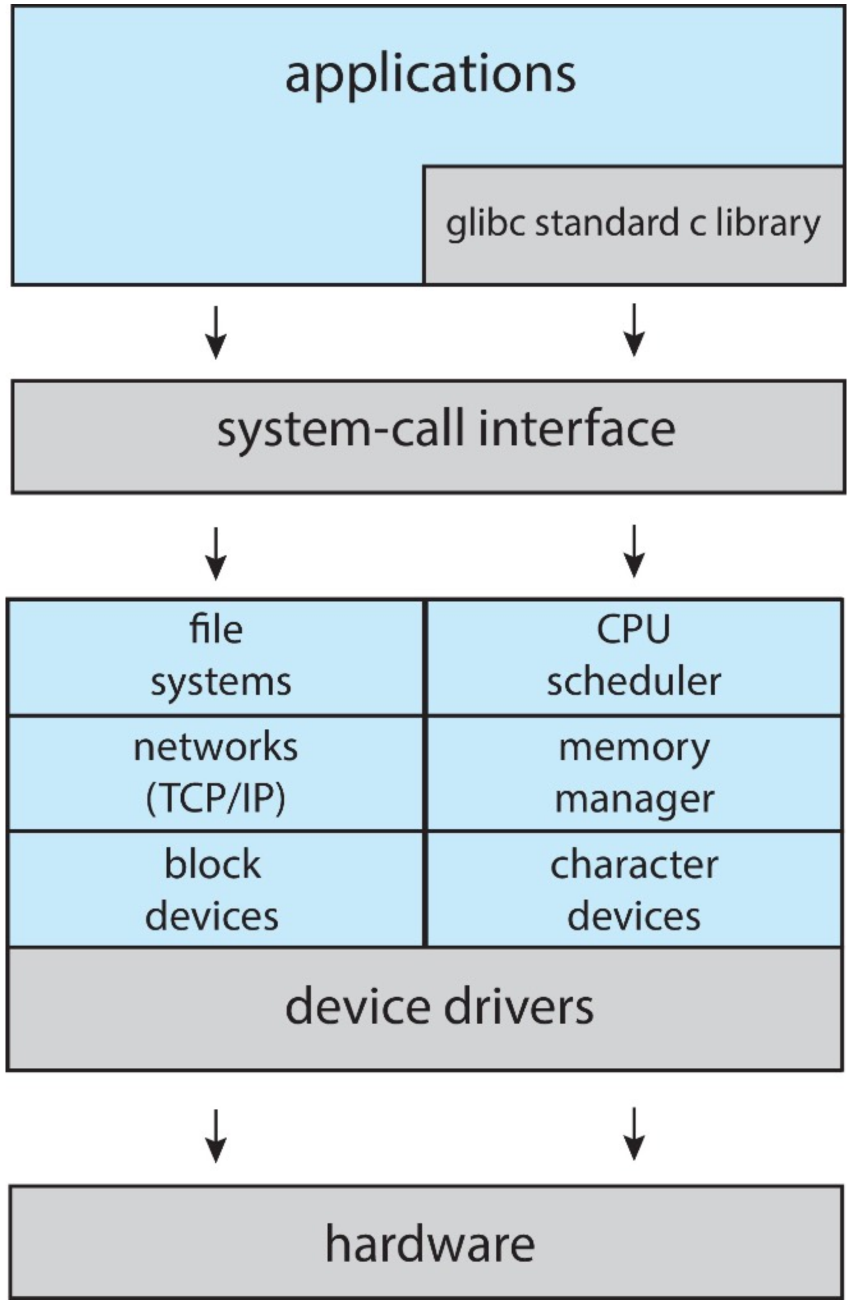


All functionality of the kernel placed into a single static binary file that runs in a single address space

- faster communication with the kernel
- little overhead in the system call interface

Linux has a monolithic structure too

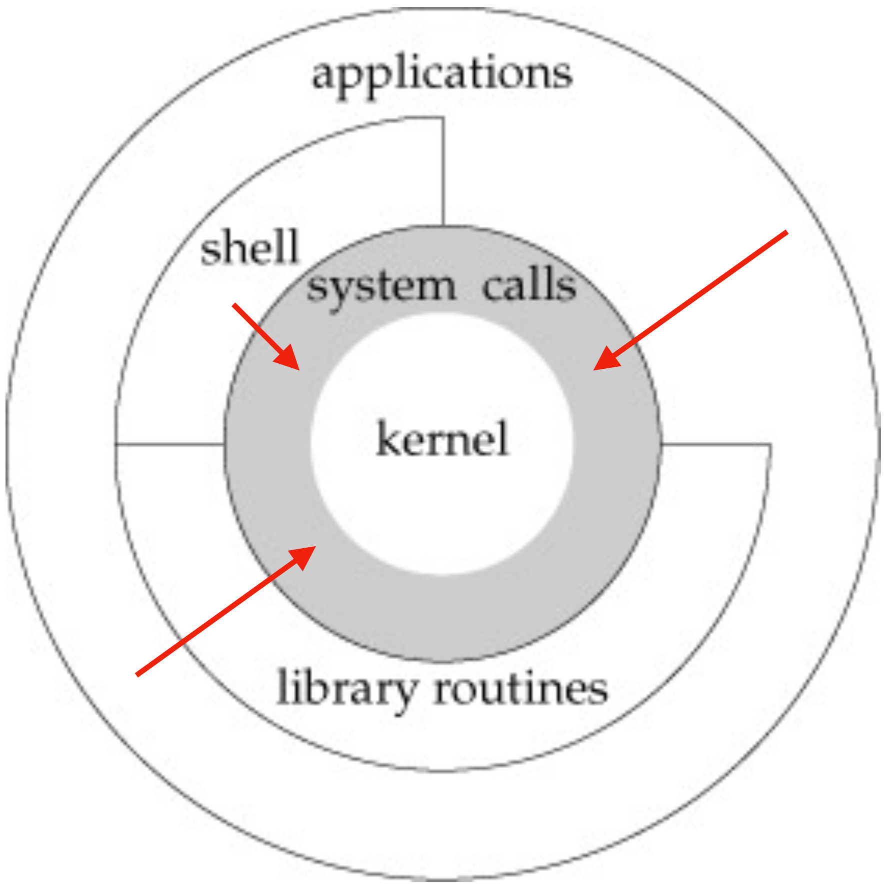
- core (~ 30% of Linux source code) + dynamically loaded kernel modules
    - examples are device drivers, file systems, network protocols
- modules can be loaded at boot time or during run time
    - adding new modules does not require recompiling the kernel
    - each module talks to other modules through known interfaces



Linux has a monolithic structure too

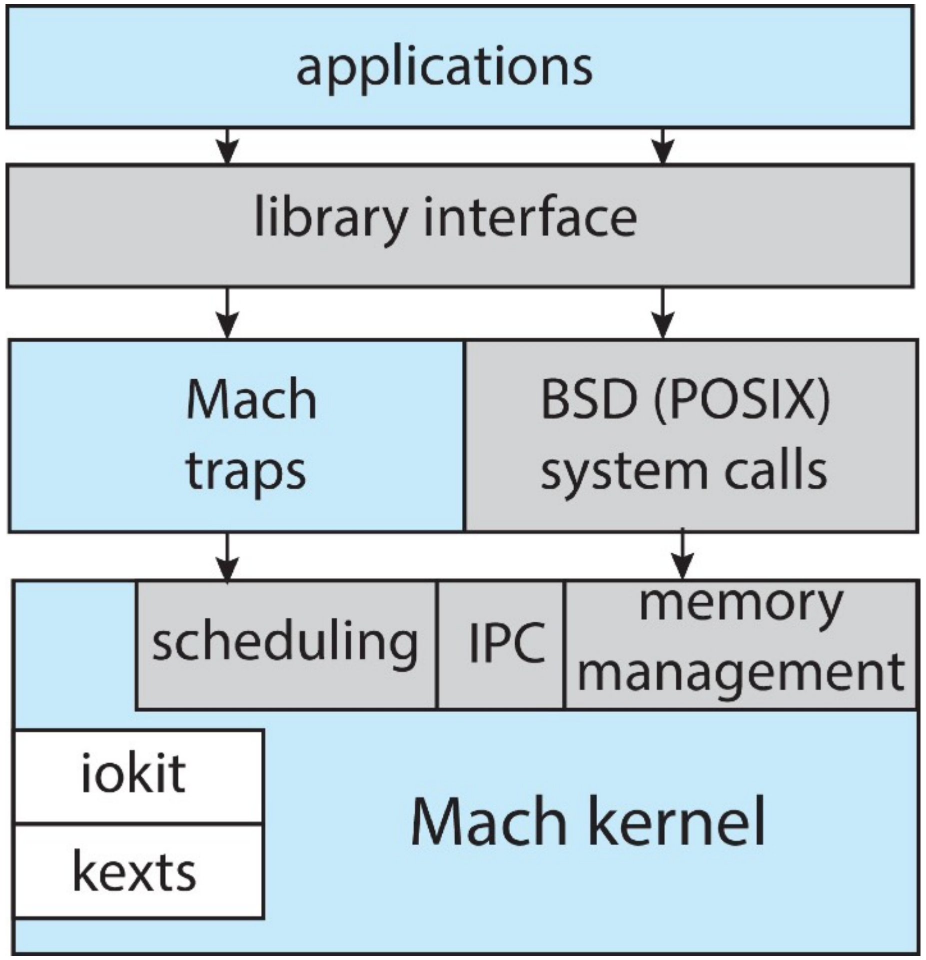
- Applications use the GNU version of the standard C library (**glibc**) which provides the functionality required by POSIX
  - glibc provides a wrapper around system calls
  - glibc is the system-call interface to the kernel
- Linux kernel source: https://www.kernel.org/
  - Directory structure:

```
include: public headers
kernel: core kernel components (e.g., scheduler)
arch: hardware-dependent code
fs: file systems
mm: memory management
ipc: interprocess communication
drivers: device drivers
usr: user-space code
lib: common libraries
```



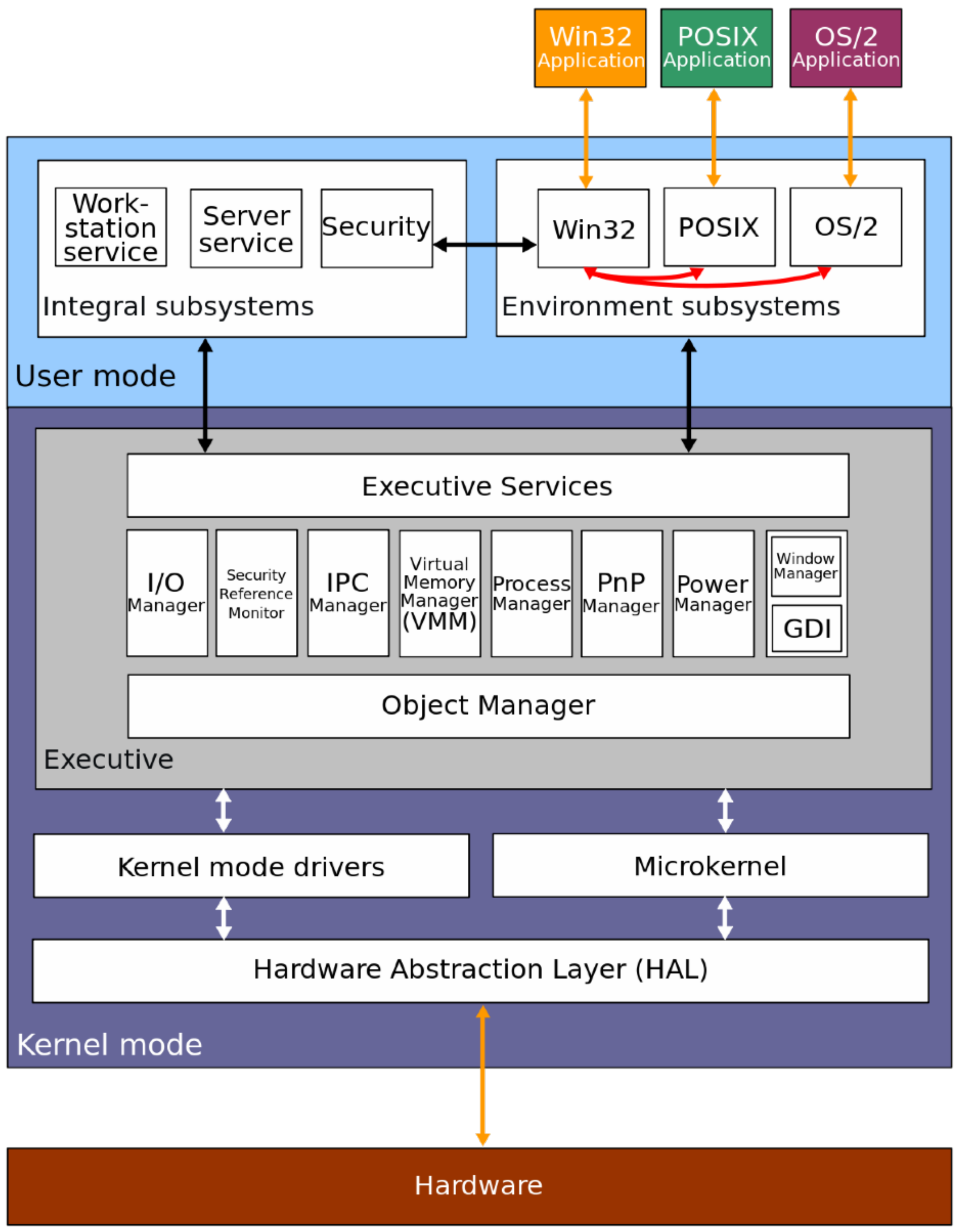
Darwin, macOS kernel, has a hybrid structure

- Darwin is Mach **microkernel** + BSD (threads, command line interface, networking, file system) + user-level services (GUI)
- Layered structure
  - <u>advantages:</u> modularity, simplicity, portability, ease of design/debugging
  - <u>disadvantage:</u> communication overhead between layers, extra copying, book-keeping
- Microkernel structure
  - a small kernel providing
    - interprocess communication (message passing usually through **ports**)
    - basic functionality (e.g., scheduling and virtual memory management)
  - other OS functionality (device drivers, file system, networking, user interface, etc.) implemented as user-level programs



Windows NT

- Layered architecture with several modules
- Windows executive provides core OS services (Ntoskrnl.exe)
- Kernel itself resides between HAL and the executive layer
  - responsible for thread scheduling and interrupt handling
- Windows native APIs are undocumented
  - POSIX, OS/2 and Win32 APIs are documented and can be used by applications
- The hardware abstraction layer (HAL.dll) provides portability across a variety of hardware platforms
  - device drivers call functions in HAL to interface with the hardware



Homework

- Familiarize yourself with Linux system calls
- Read the Linux Journal article about linking and loading