Operating System Concepts

Lecture 3: System Calls, Linking and Loading

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

Today's class

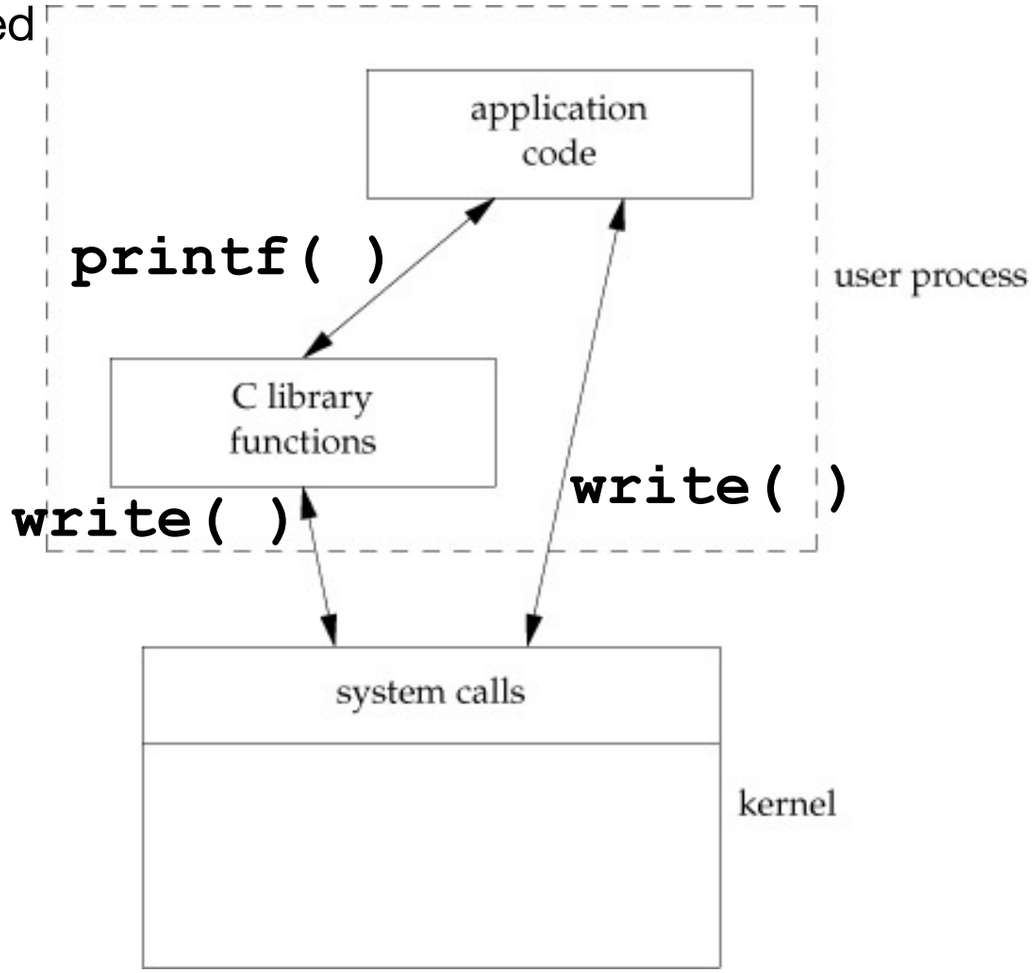
- Hardware support
  - System calls
  - Protection
- Basics of compiling, linking, and loading
- OS structure
  - Examples

System calls… what are they?

- Definition: services provided by the kernel to application programs
  - There are instructions that only the OS can run
  - A typical OS exports a few hundred (~300-400) system calls

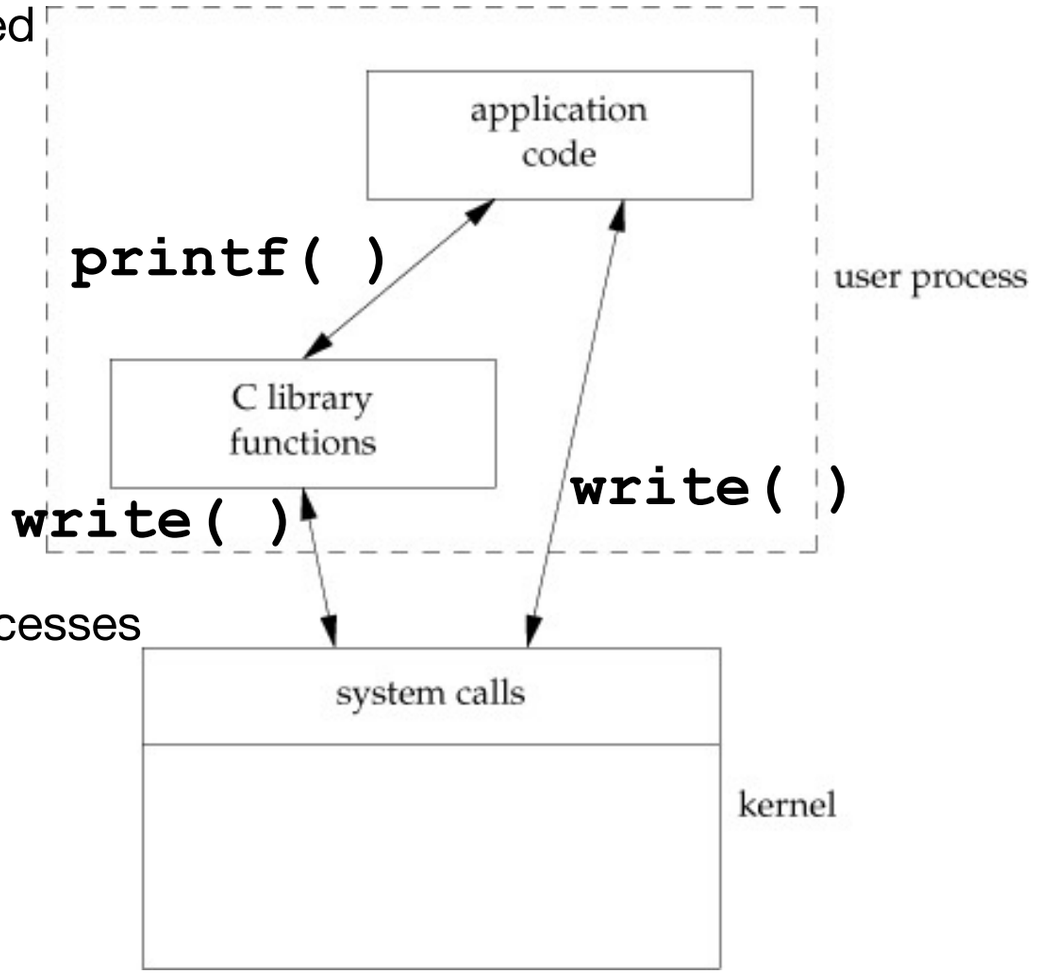
System calls… what are they?

- Definition: services provided by the kernel to application programs
  - There are instructions that only the OS can run
  - A typical OS exports a few hundred (~300-400) system calls
- Each system call is typically defined as a function in the standard C library
  - In UNIX-based systems most system calls are declared in `unistd.h` and some are declared in `fcntl.h`



System calls… what are they?

- Definition: services provided by the kernel to application programs
  - There are instructions that only the OS can run
  - A typical OS exports a few hundred (~300-400) system calls
- Each system call is typically defined as a function in the standard C library
  - In UNIX-based systems most system calls are declared in `unistd.h` and some are declared in `fcntl.h`
- System calls are necessary to
  - Access devices and files
  - Request memory
  - Set access permissions
  - Stop and start processes, and communicate with processes
  - Set a timer



UNIX system calls

- Process management
  - `getpid( ), fork( ), exec( ), wait( ), exit( ), ...`

UNIX system calls

- Process management
  - `getpid( ), fork( ), exec( ), wait( ), exit( ), ...`
- File manipulation
  - `open( ), close( ), read( ), write( ), stat( ), lseek( ), link( ), ...`

UNIX system calls

- Process management
  - `getpid( ), fork( ), exec( ), wait( ), exit( ), ...`
- File manipulation
  - `open( ), close( ), read( ), write( ), stat( ), lseek( ), link( ), ...`
- Protection
  - `chmod( ), chown( ), ...`

UNIX system calls

- Process management
  - `getpid( ), fork( ), exec( ), wait( ), exit( ), ...`
- File manipulation
  - `open( ), close( ), read( ), write( ), stat( ), lseek( ), link( ), ...`
- Protection
  - `chmod( ), chown( ), ...`
- Communications
  - `pipe( ), shm_open( ), mmap( ), socket( ), accept( ), send( ), recv( ), ...`

UNIX system calls

- Process management
  - `getpid( ), fork( ), exec( ), wait( ), exit( ), ...`
- File manipulation
  - `open( ), close( ), read( ), write( ), stat( ), lseek( ), link( ), ...`
- Protection
  - `chmod( ), chown( ), ...`
- Communications
  - `pipe( ), shm_open( ), mmap( ), socket( ), accept( ), send( ), recv( ), ...`

See Linux system calls here:
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 `strace` man page)

```
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

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 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 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)

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)
  - POSIX thread library (aka pthreads) is defined in an extension of this API known as POSIX.1c

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)
  - 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

- Hardware has a status bit that indicates the current operation mode
  - there could be more than two operation modes
    e.g., ARMv8 systems have seven modes
  - user applications run in the **user mode**
  - kernel code runs in the **kernel model** with the **full privileges** of the hardware
  - examples of privileged instructions are I/O control, timer management, interrupt management; they can only run in the kernel mode

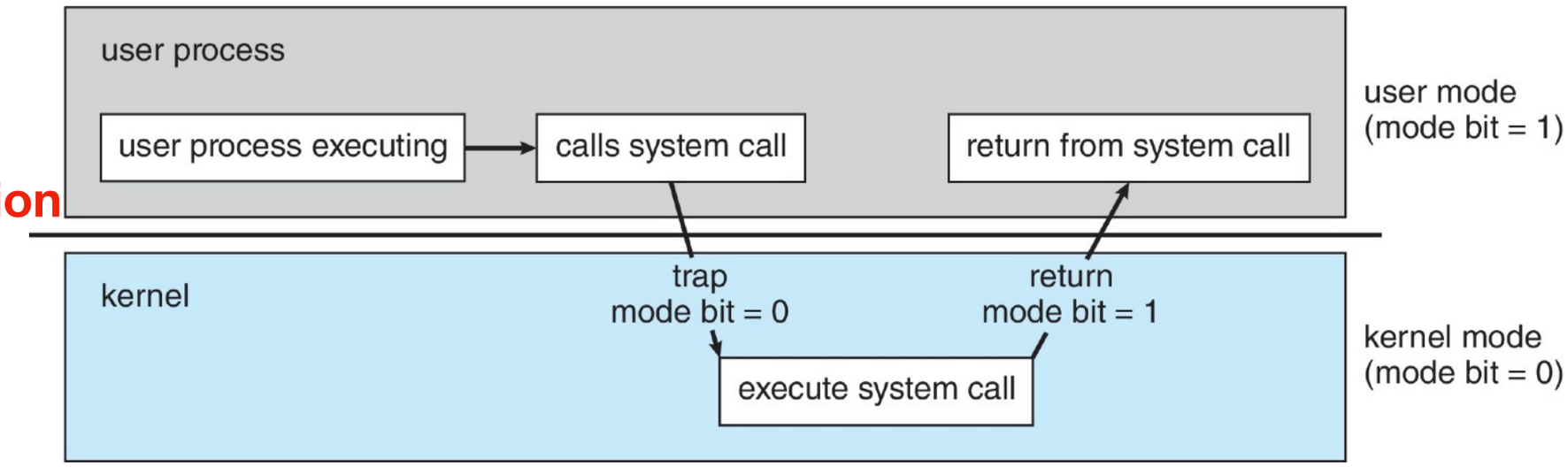
Protection

- Hardware has a status bit that indicates the current operation mode
  - there could be more than two operation modes
    e.g., ARMv8 systems have seven modes
  - user applications run in the **user mode**
  - kernel code runs in the **kernel model** with the **full privileges** of the hardware
  - 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 a user program to run privileged instructions

Protection

- Hardware has a status bit that indicates the current operation mode
  - there could be more than two operation modes
    e.g., ARMv8 systems have seven modes
  - user applications run in the **user mode**
  - kernel code runs in the **kernel model** with the **full privileges** of the hardware
  - 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 a user program to run privileged instructions
- operation modes provide the means for protecting OS from errant users
  - the code could be buggy or malicious

crossing protection boundary

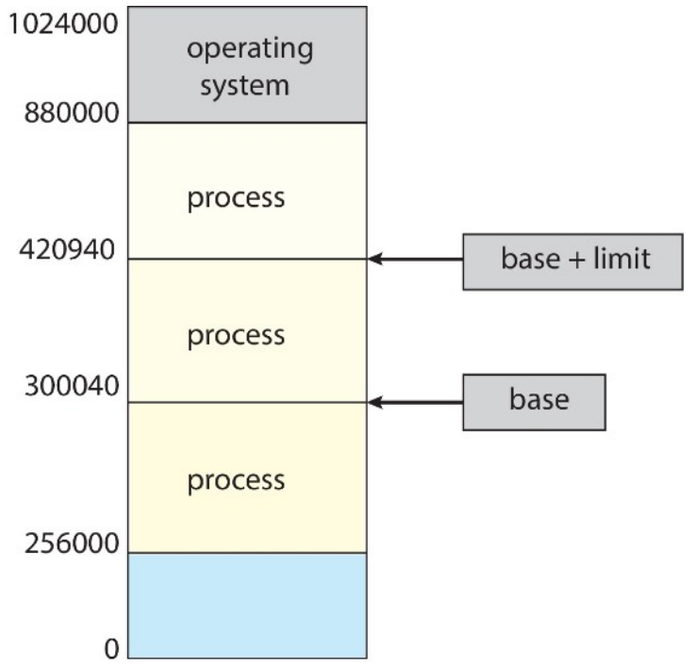


Memory protection

- hardware must provide support so that the OS can
  - protect user programs from each other
  - protect the OS from user programs

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 and limit registers are loaded by the OS before creating a process
  - 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

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

Timer interrupt

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

- the kernel must periodically get back control

Timer interrupt

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

- the kernel must periodically get back control
- CPU is protected from being hogged using timer interrupts that occur at regular intervals (e.g., every 100 microseconds)
  - frequency is set by the kernel
  - yet another protection mechanism

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

Interrupt Vector

Timer interrupt

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

- the kernel must periodically get back control
- CPU is protected from being hogged using timer interrupts that occur at regular intervals (e.g., every 100 microseconds)
  - frequency is set by the kernel
  - yet another protection mechanism
- at each timer interrupt, the CPU chooses a new process to execute

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

Interrupt Vector

Timer interrupt

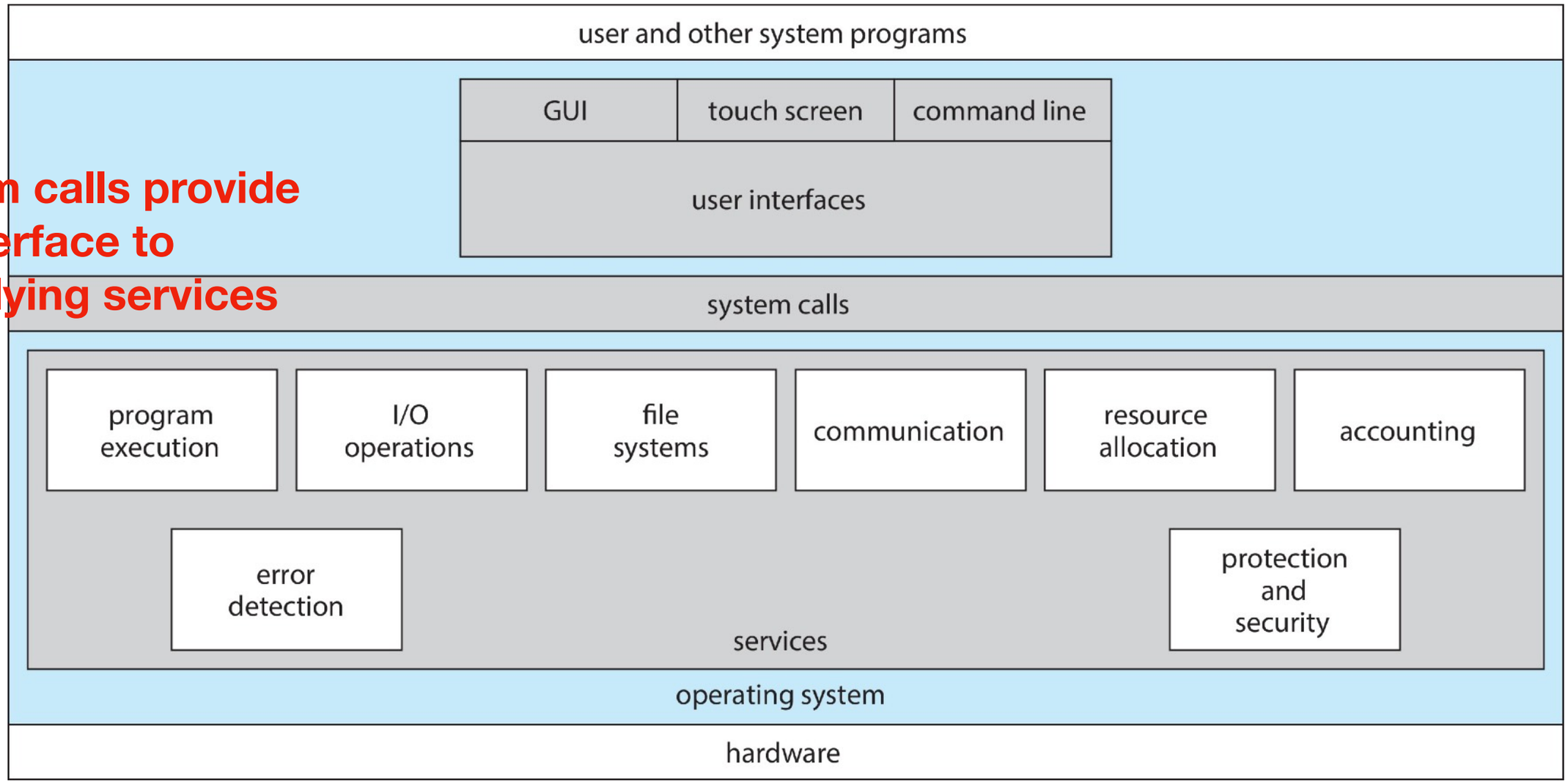
What if the process running on the CPU does not wait for I/O or signals?

- the kernel must periodically get back control
- CPU is protected from being hogged using timer interrupts that occur at regular intervals (e.g., every 100 microseconds)
  - frequency is set by the kernel
  - yet another protection mechanism
- at each timer interrupt, the CPU chooses a new process to execute
- interrupts can be temporarily deferred
  - it is crucial for implementing mutual exclusion

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

Interrupt Vector

Operating System services



system calls provide an interface to underlying services

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

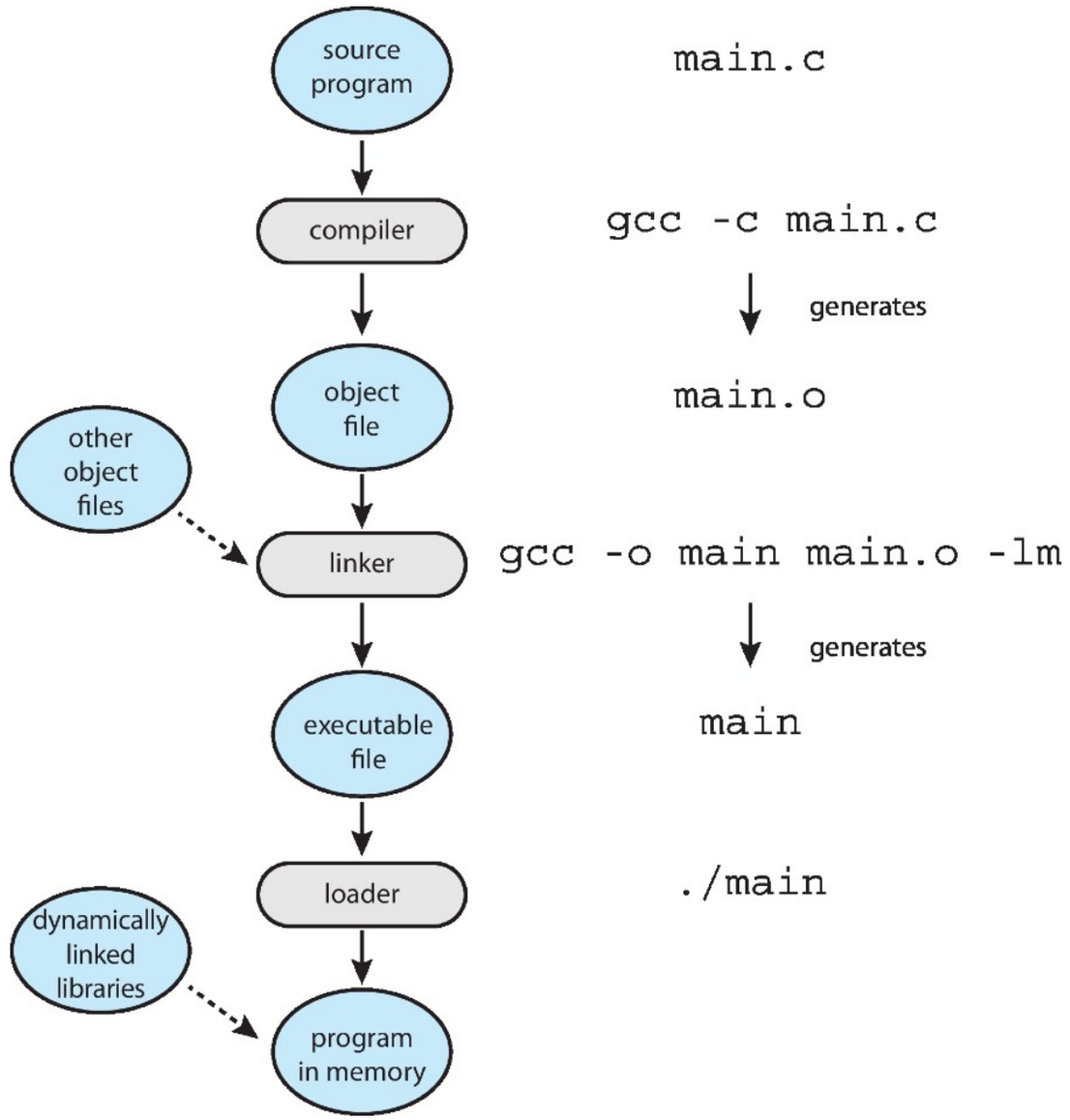
Summary

OS Services	Hardware Support
Interrupts	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
Scheduling & accounting	timer
Synchronization	atomic operations
Virtual memory	MMU, translation look-aside buffer (TLB)

Running a user program

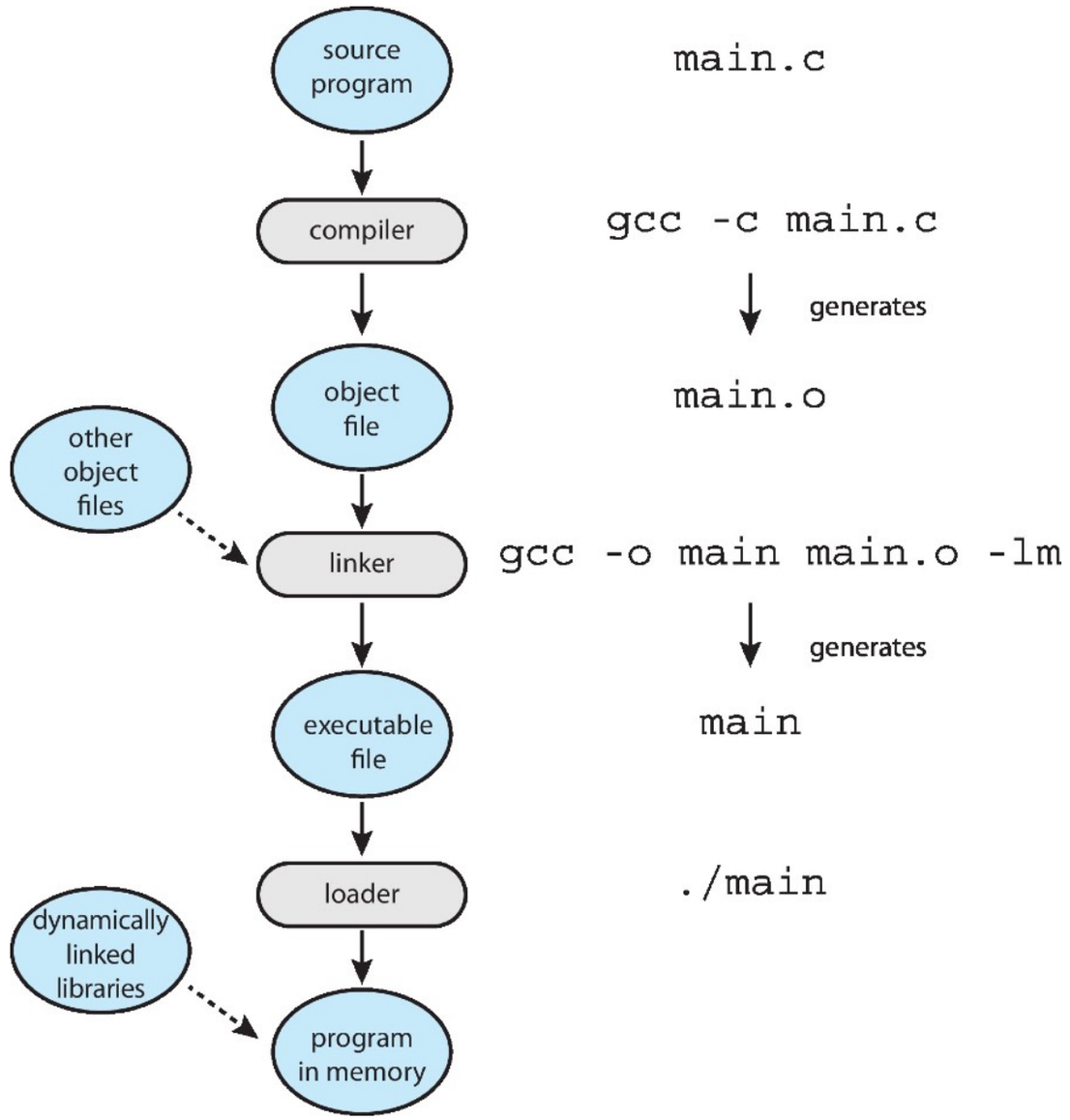
Program to Process

- Compiling is the process of converting a source file (ASCII code) into an object file
  - object files miss certain information, e.g., functions called from other files and libraries



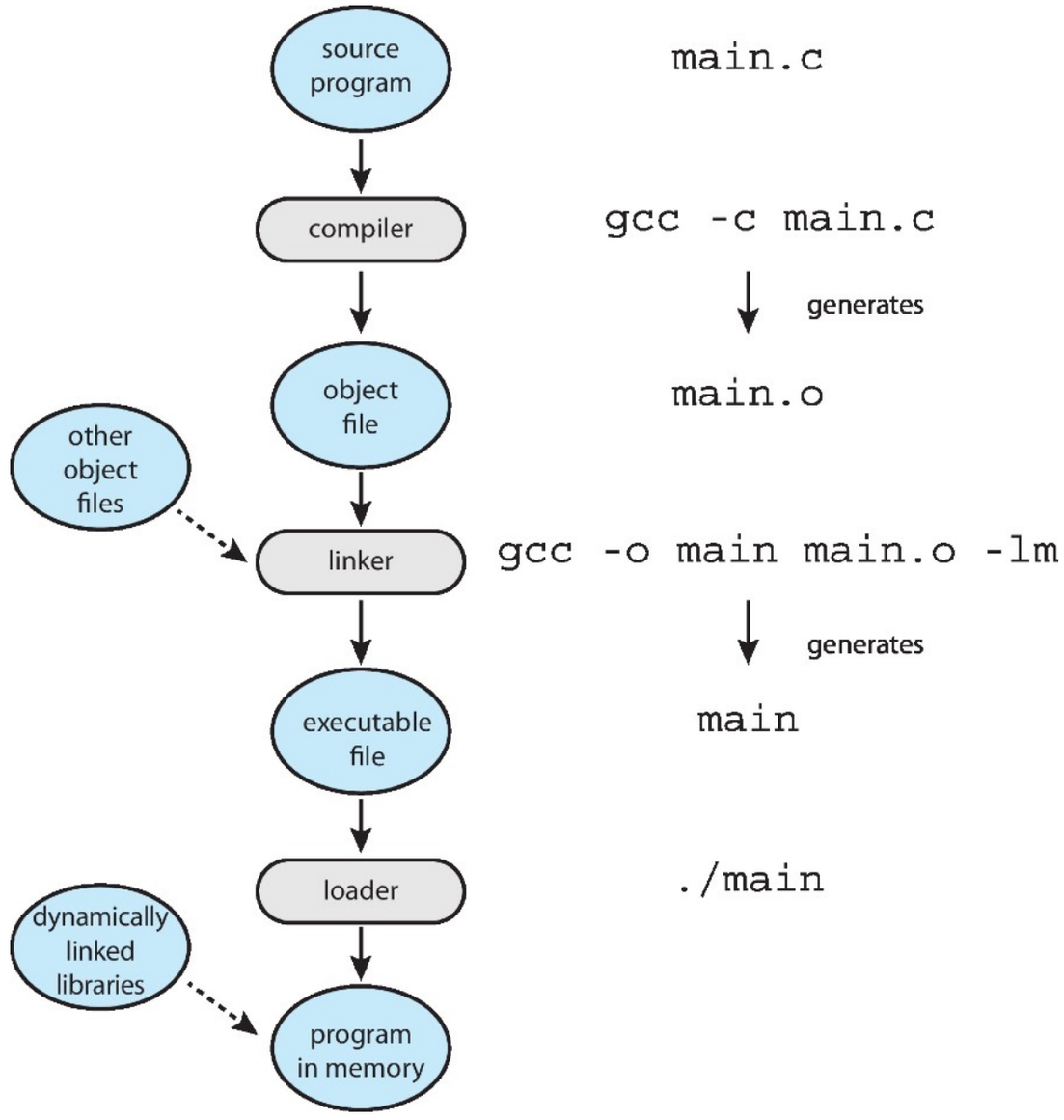
Program to Process

- Compiling is the process of converting a source file (ASCII code) into an object file
  - object files miss certain information, e.g., functions called from other files and libraries
- Linking is the process of combing relocatable object files and specific libraries into a single binary executable



Program to Process

- Compiling is the process of converting a source file (ASCII code) into an object file
  - object files miss certain information, e.g., functions called from other files and libraries
- Linking is the process of combing relocatable object files and specific libraries into a single binary executable
- 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 `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 the OS knows how to load and start them

Executable and Linkable Format (ELF)

- Executable files must have a standard format, so that the OS knows how to load and start them
- ELF is a common standard across modern UNIX operating systems, such as Linux

Executable and Linkable Format (ELF)

- Executable files must have a standard format, so that the OS knows how to load and start them
- ELF is a common standard across modern UNIX operating systems, such as Linux
- **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

Executable and Linkable Format (ELF)

- Executable files must have a standard format, so that the OS knows how to load and start them
- ELF is a common standard across modern UNIX operating systems, such as Linux
- **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)

- Executable files must have a standard format, so that the OS knows how to load and start them
- ELF is a common standard across modern UNIX operating systems, such as Linux
- **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
- the ELF header starts with a 4-byte magic string:
  - 0x7F followed by 0x45 0x4C 0x46 (ELF in ASCII)

Executable and Linkable Format (ELF)

An ELF file is divided into sections

Executable and Linkable Format (ELF)

An ELF file is divided into sections

- **.text** contains the machine code
  - In UNIX-based systems, see the content of this section using `objdump -drS objfile`

Executable and Linkable Format (ELF)

An ELF file is divided into 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

Executable and Linkable Format (ELF)

An ELF file is divided into 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

Executable and Linkable Format (ELF)

An ELF file is divided into 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

Executable and Linkable Format (ELF)

An ELF file is divided into sections

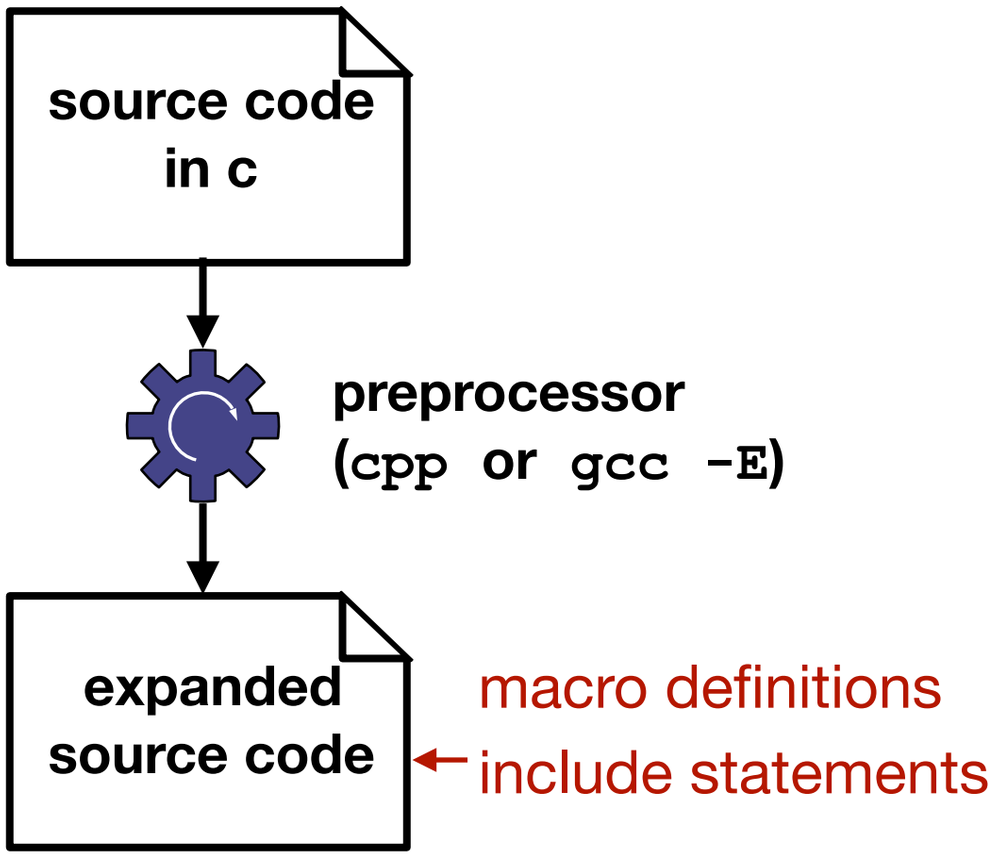
- **.text** contains the machine code
  - In UNIX-based systems, see the content of this section using `objdump -drS objfile`
- **.data** contains initialized global variables
- **.bss** (block storage start) contains uninitialized global variables
  - occupies no space in the object file actually
  - 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

Executable and Linkable Format (ELF)

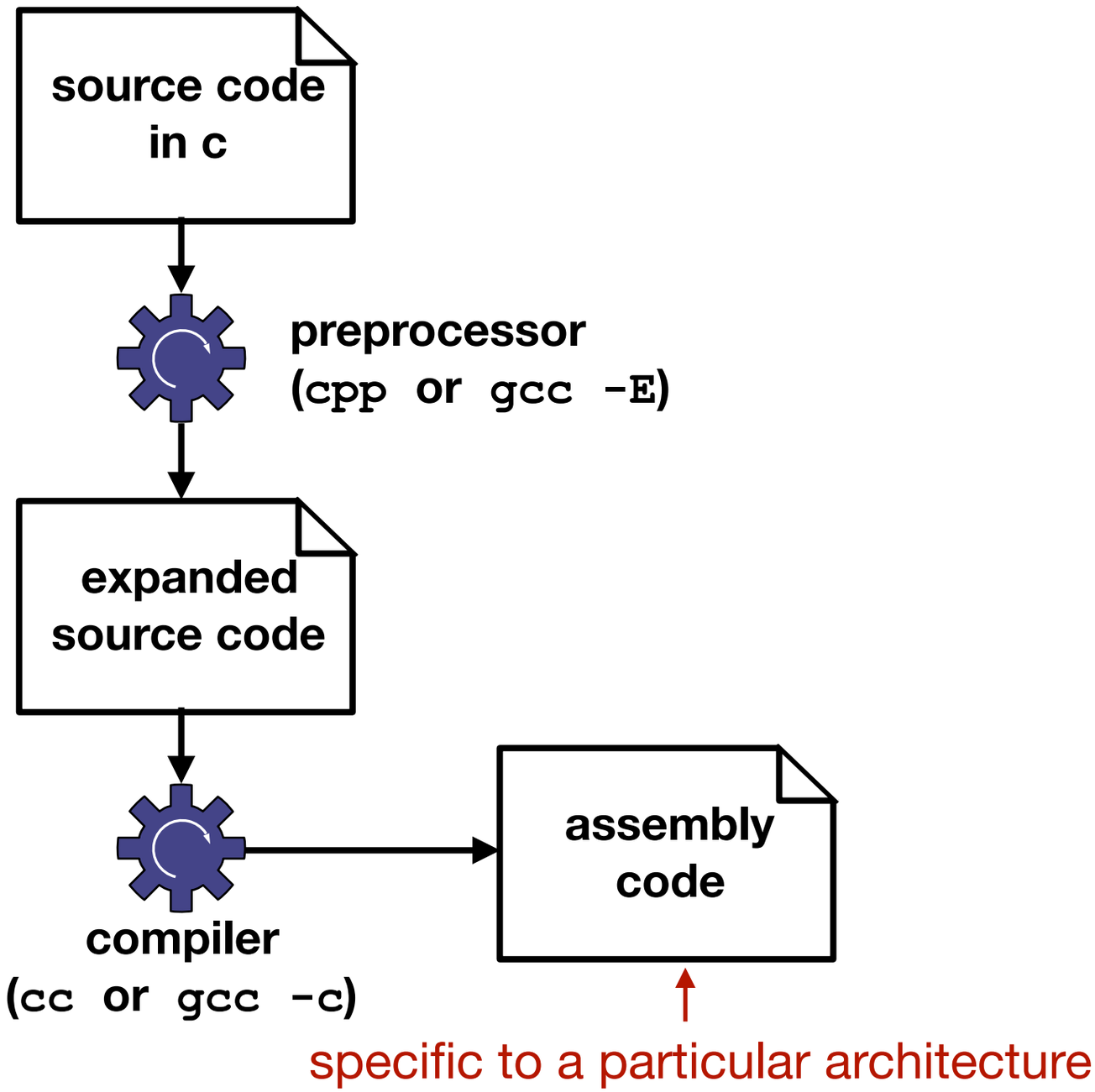
An ELF file is divided into sections

- **.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

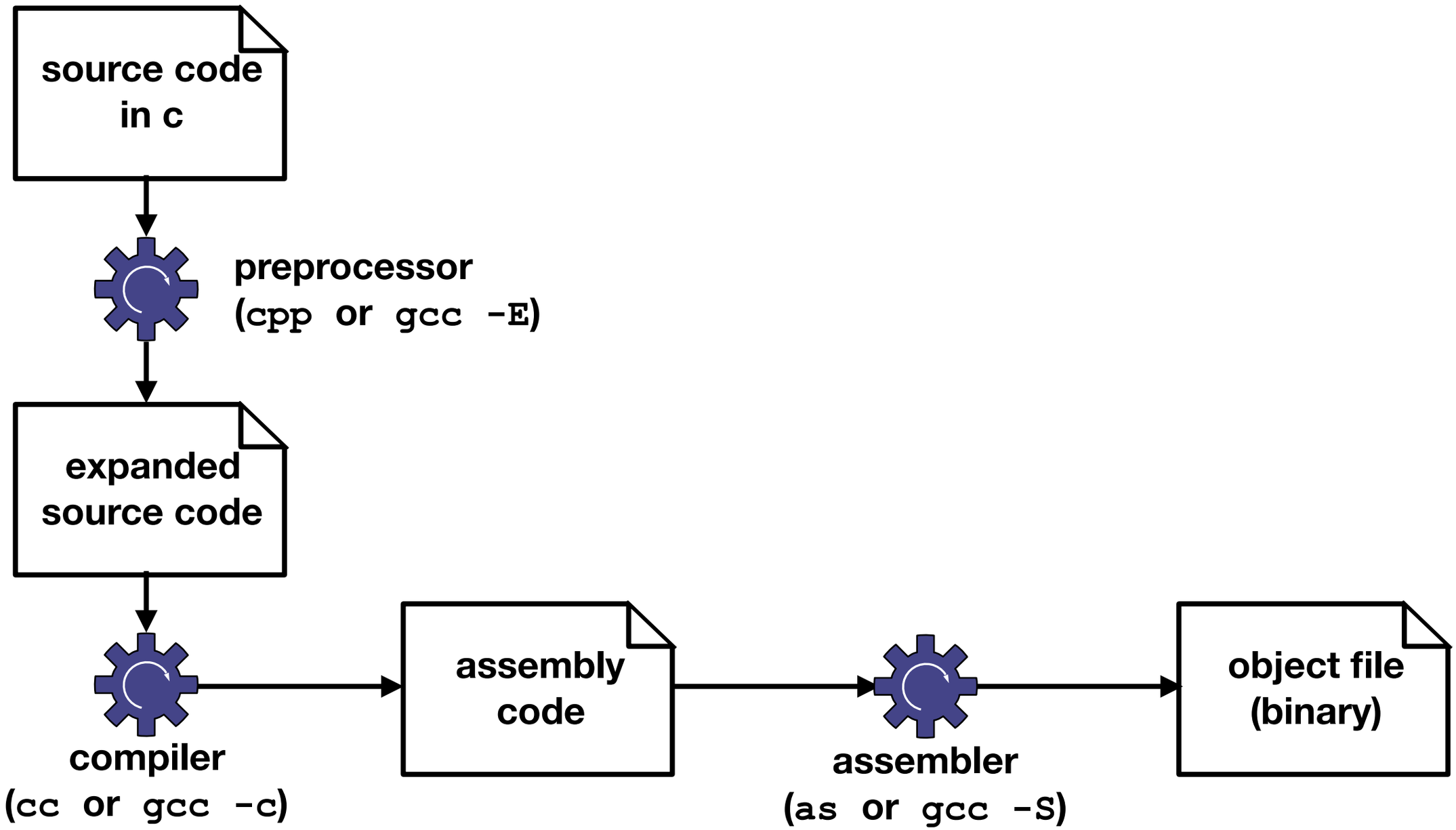
Compiler, Linker and Loader in action



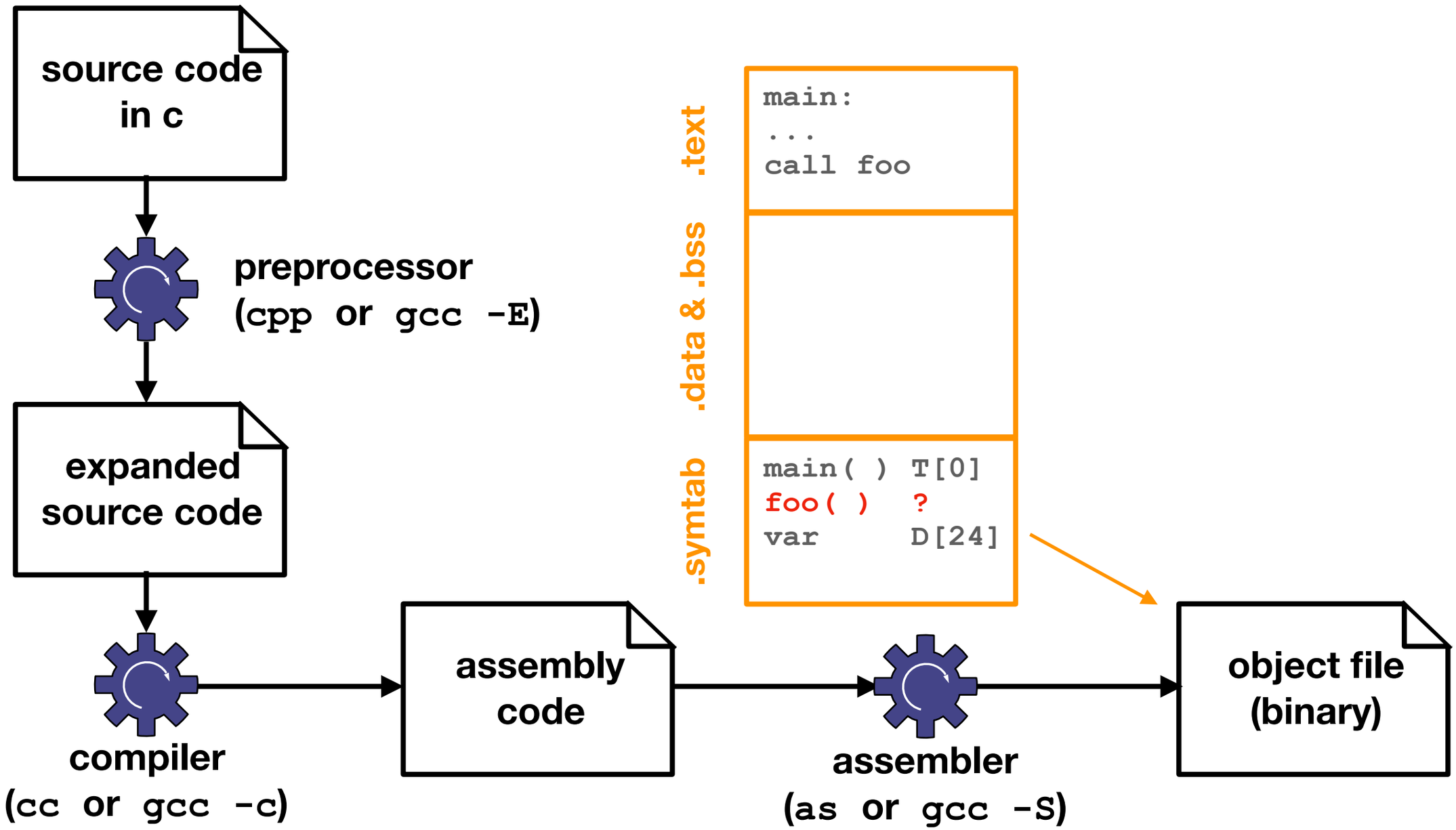
Compiler, Linker and Loader in action



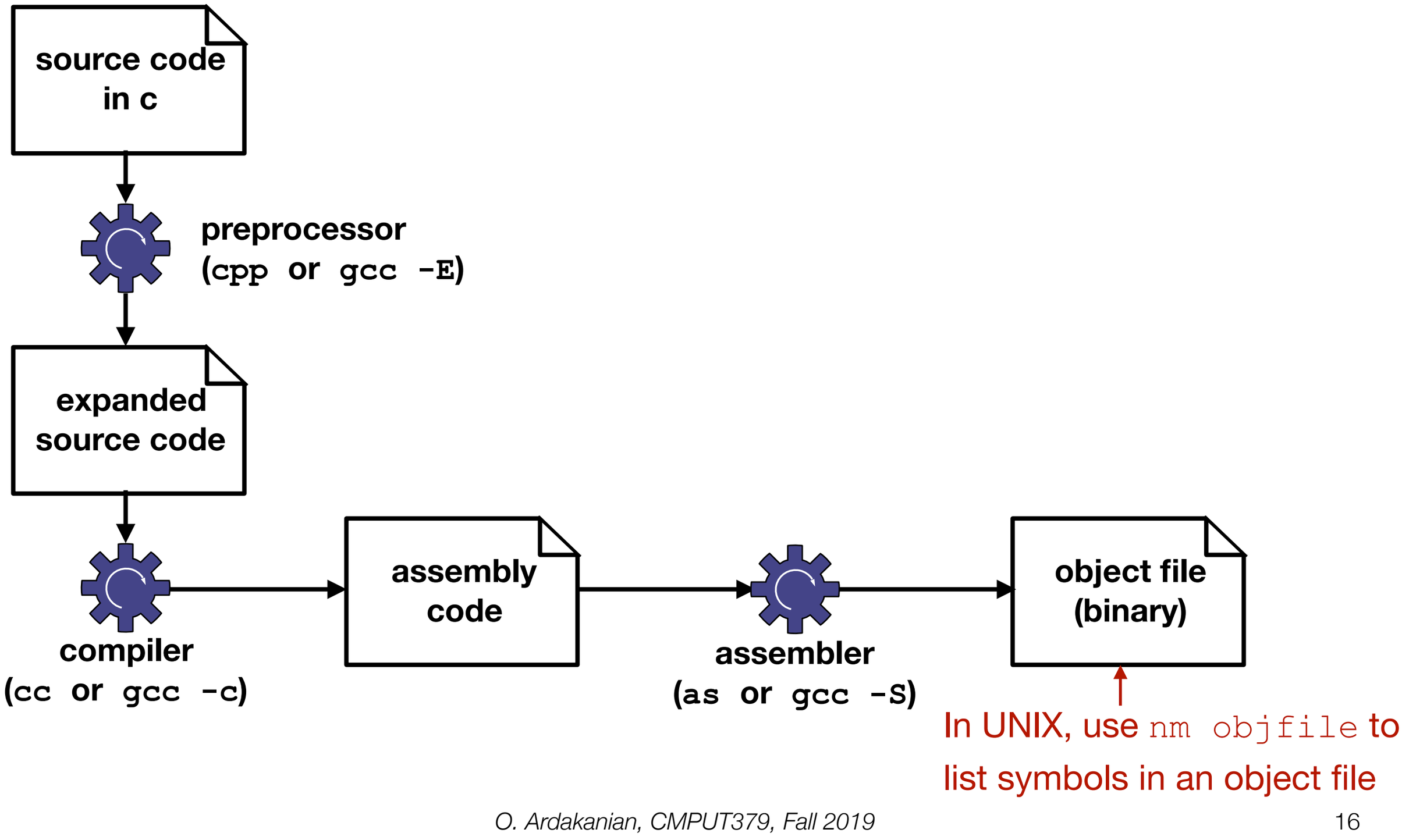
Compiler, Linker and Loader in action



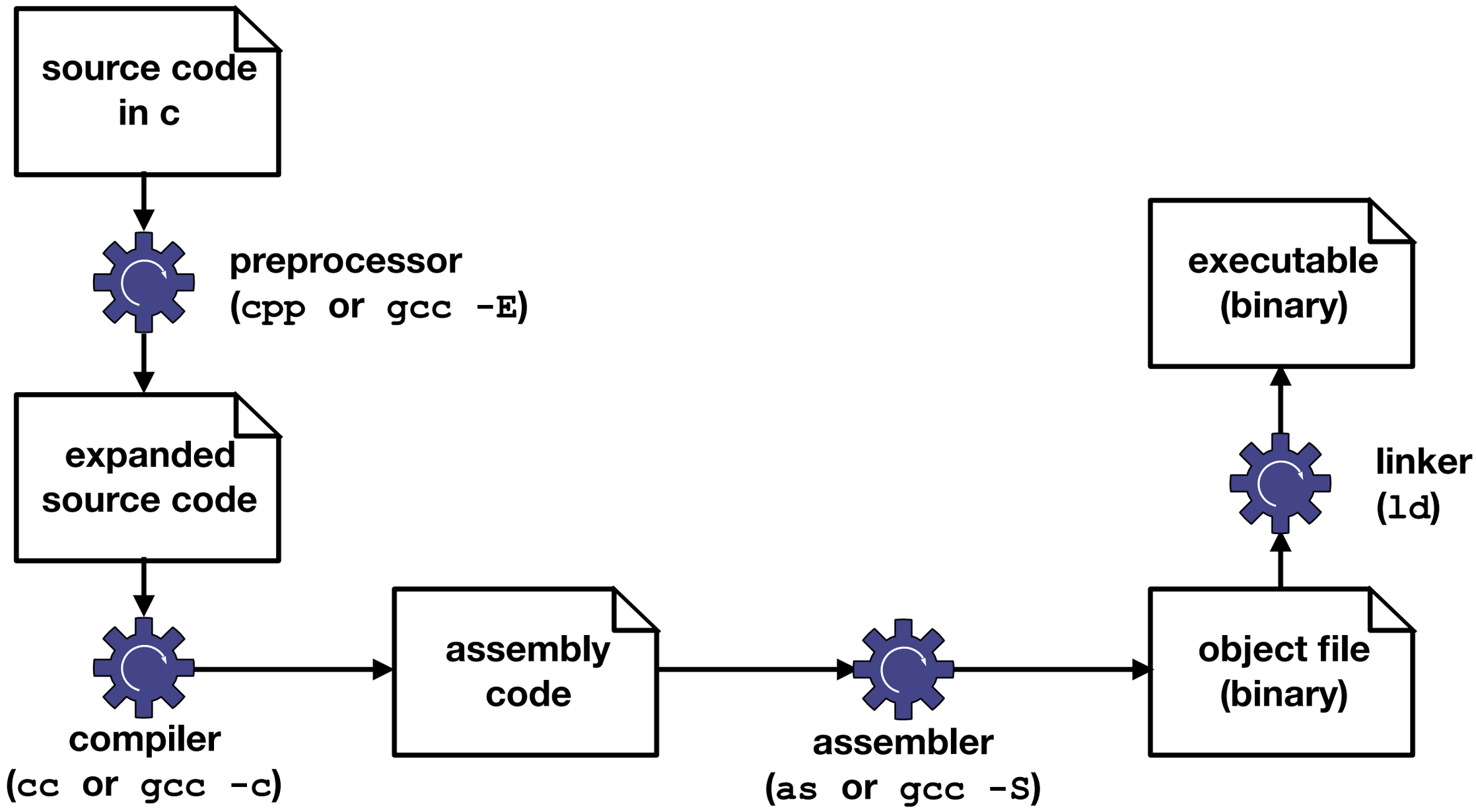
Compiler, Linker and Loader in action



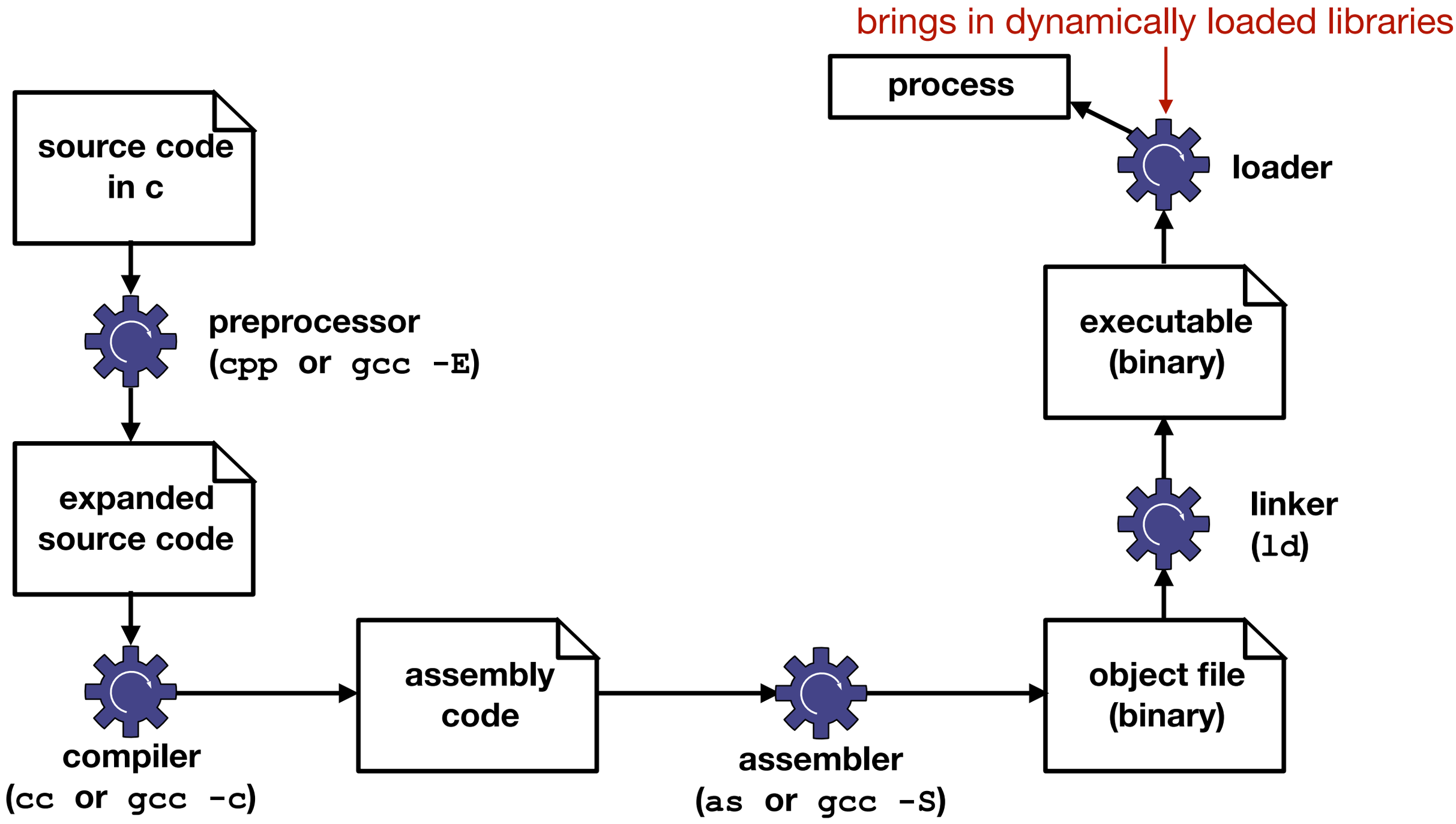
Compiler, Linker and Loader in action



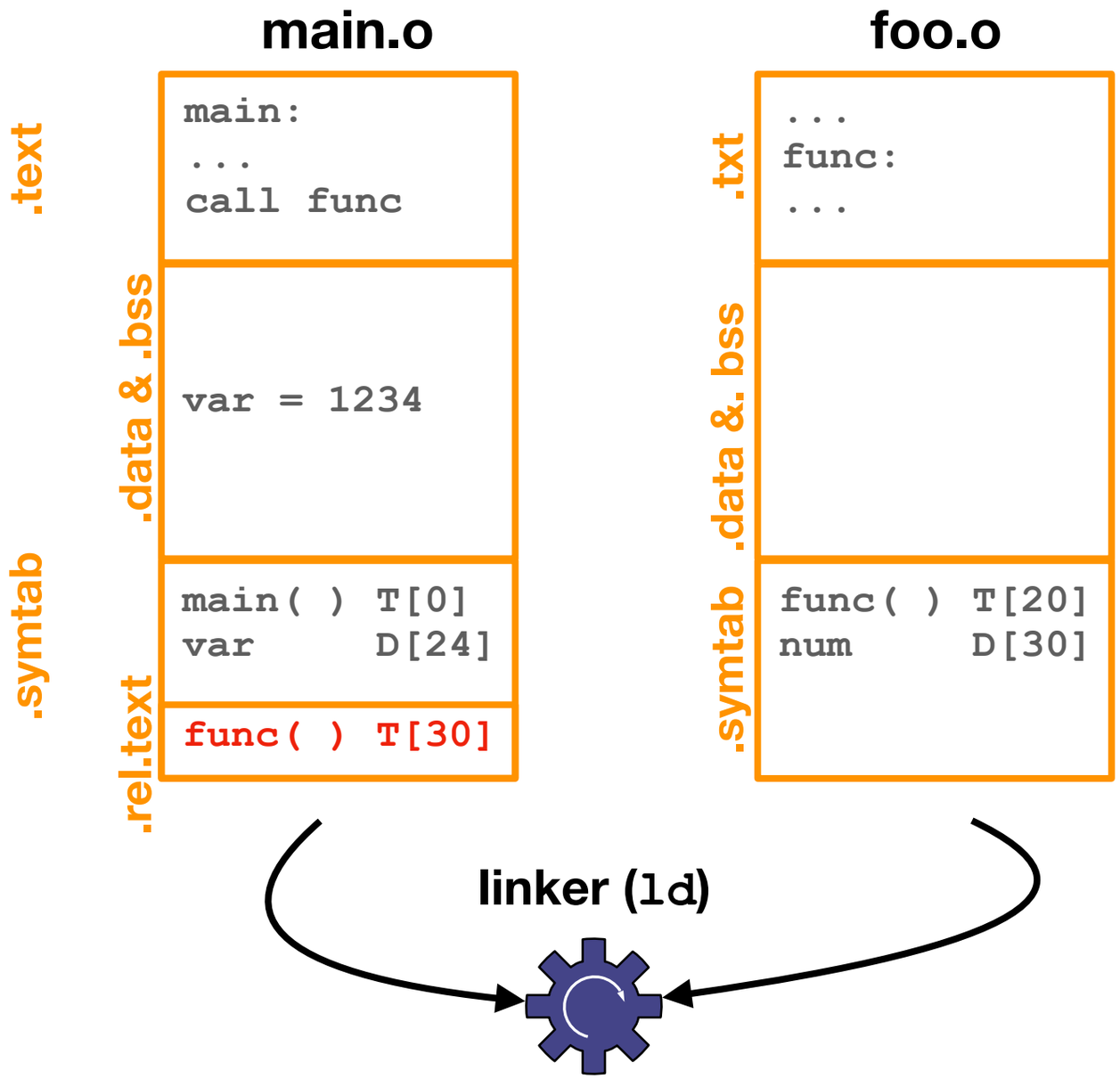
Compiler, Linker and Loader in action



Compiler, Linker and Loader in action

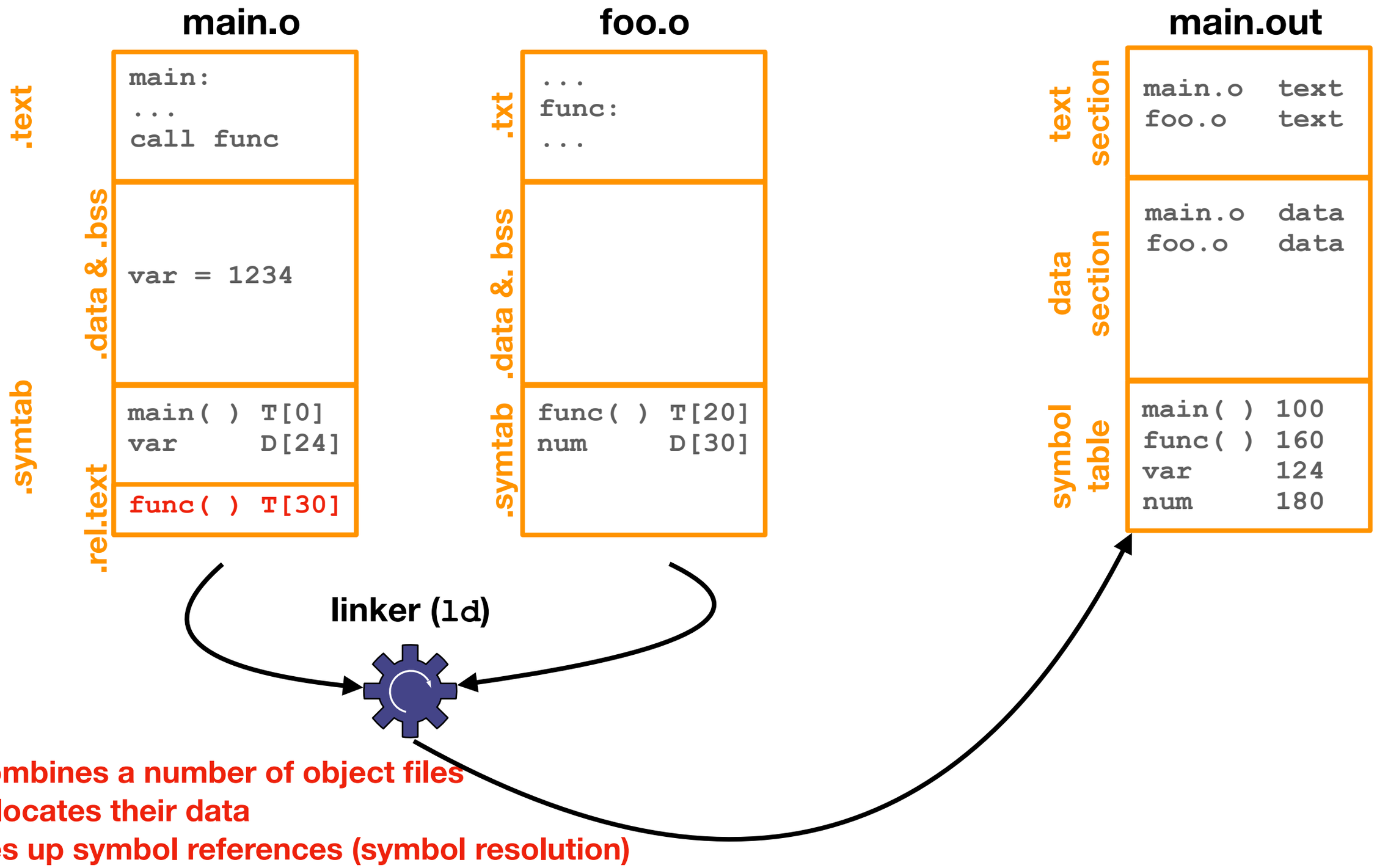


Linking



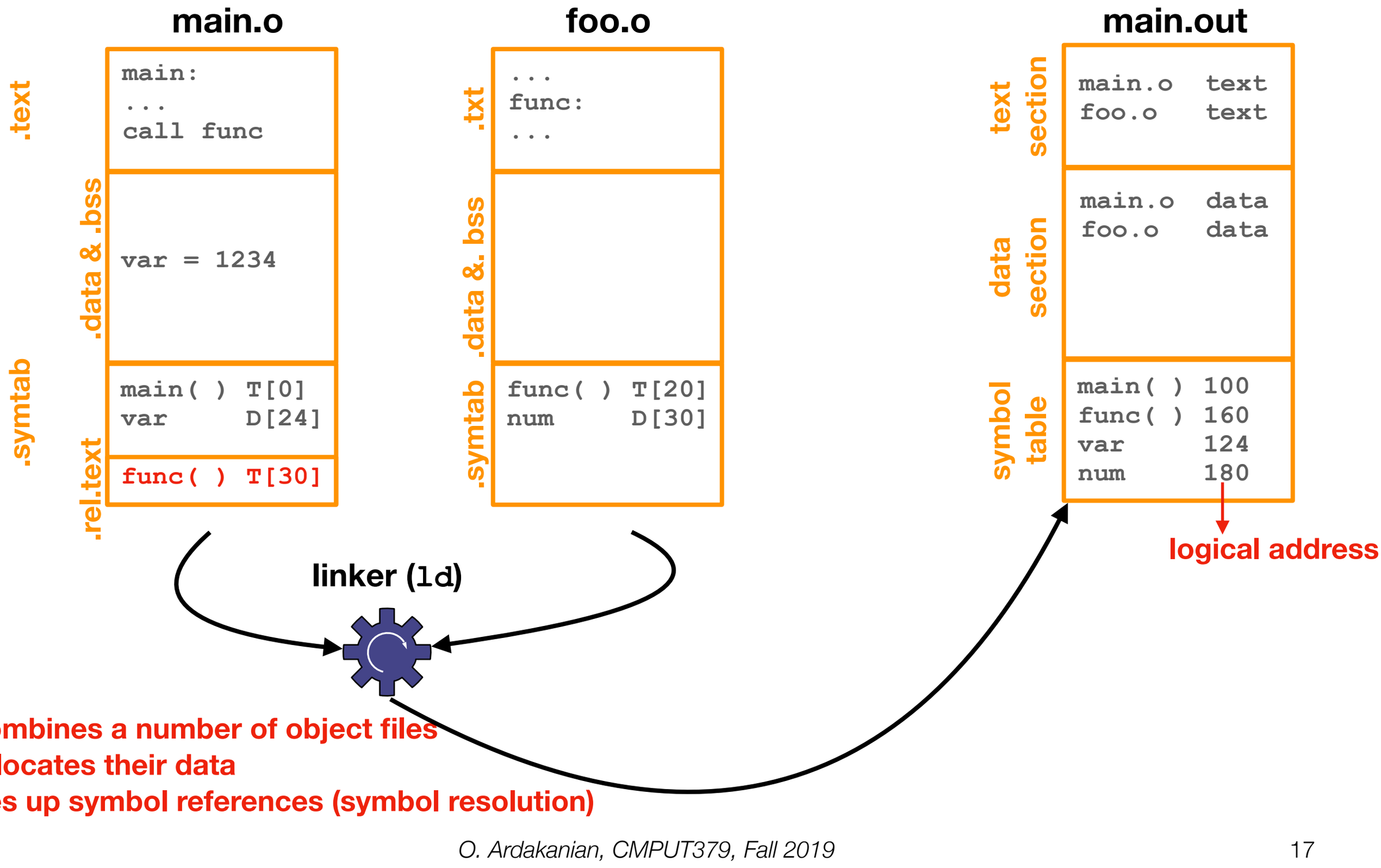
1. combines a number of object files
2. relocates their data
3. ties up symbol references (symbol resolution)

Linking



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

Linking



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

Homework

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