DCS216 Operating Systems

Lecture 03
Operating Systems Structures (1)

Mar 4th, 2024

Instructor: Xiaoxi Zhang

Sun Yat-sen University

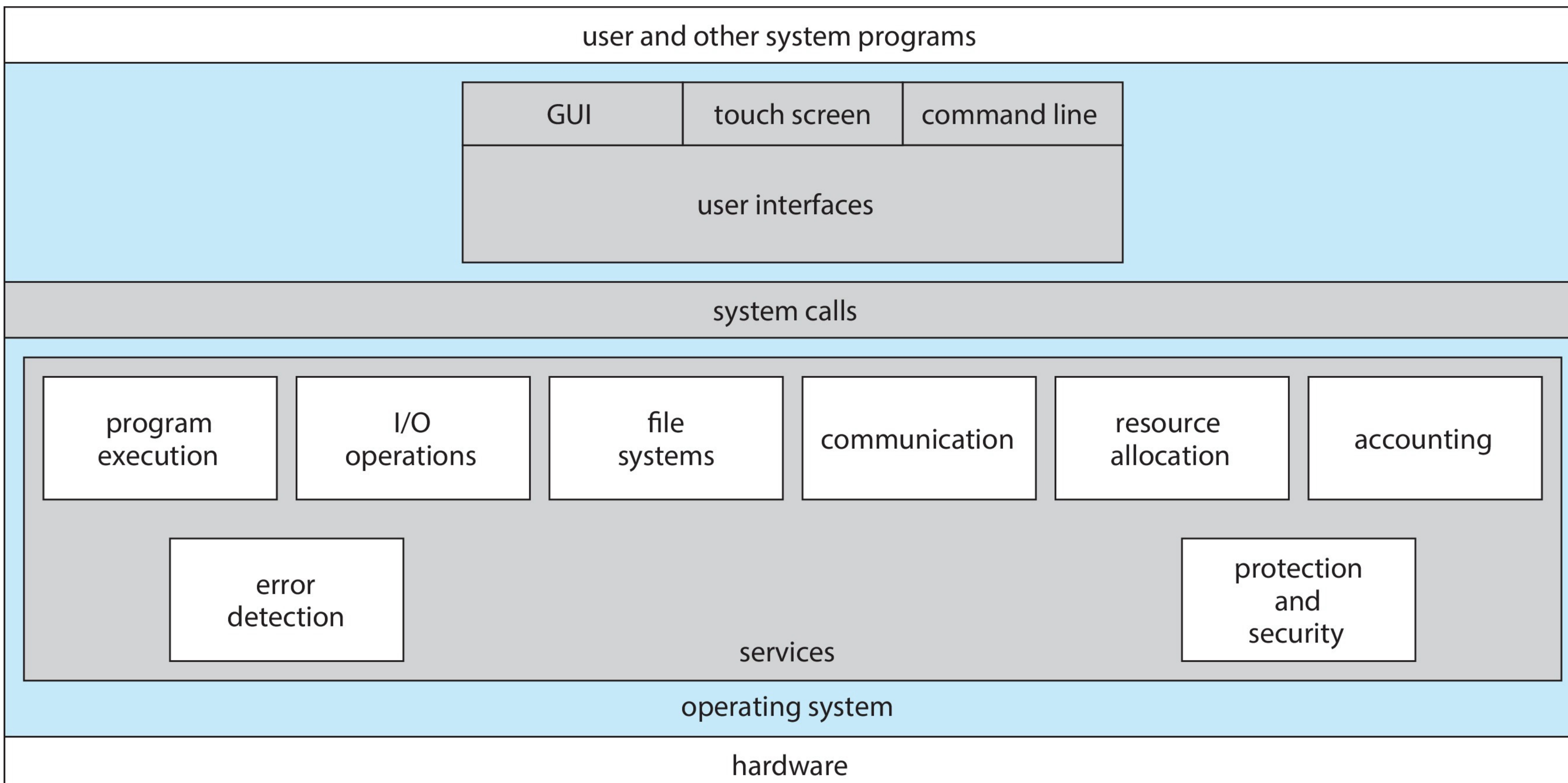
Content

- Operating System Services
- User Interface
- System Calls and APIs
- Linkers and Loaders
- Operating System Design and Implementation
- Operating System Structure
- Building and Booting an Operating System
- Operating System Debugging

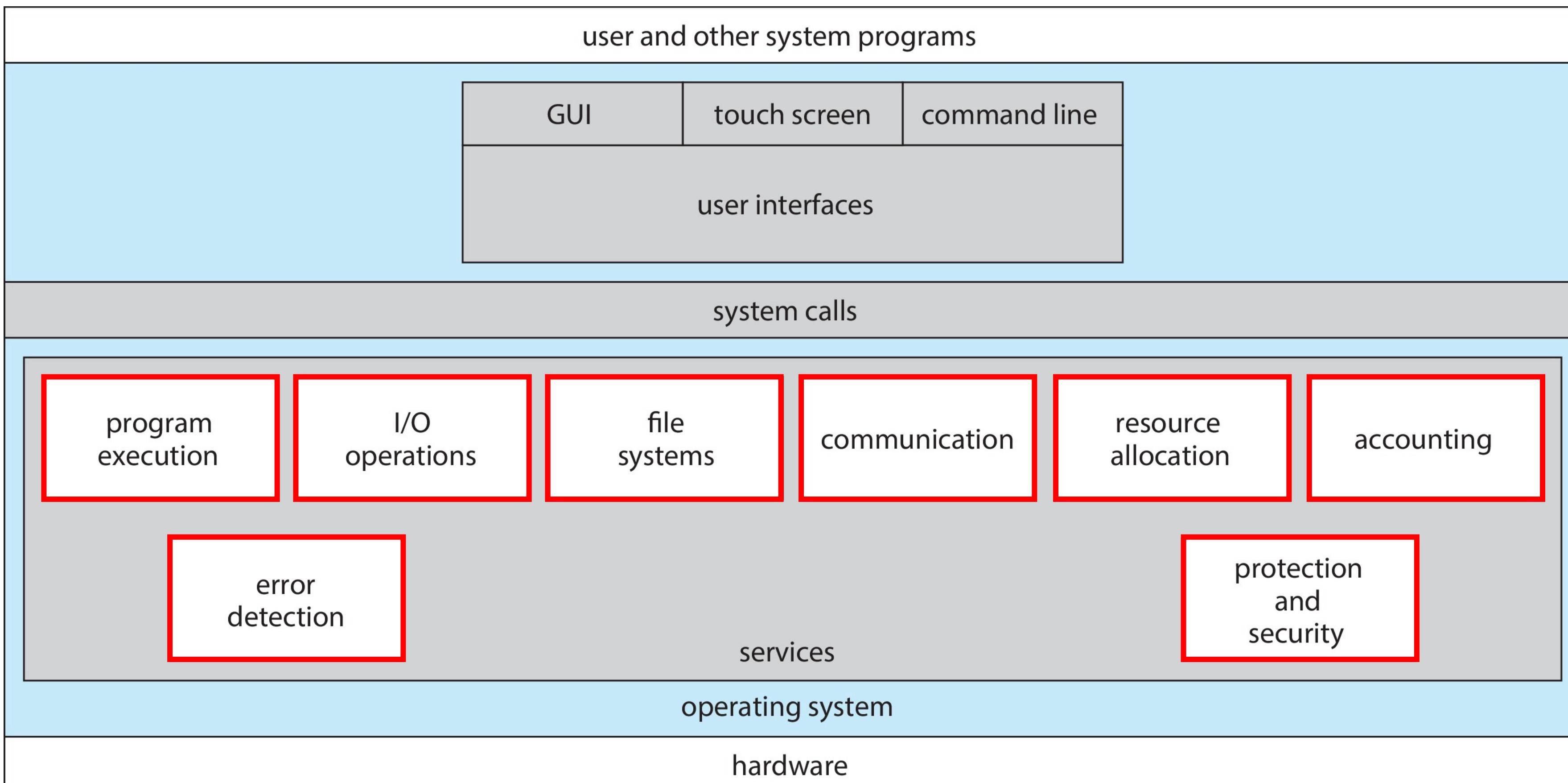
An Operating System provides the **environment** within which programs are **executed**.

操作系统提供程序**运行**所需的**环境**。

Operating System Services



- **Operating System Services**
  - A View of Common Operating System Services
  - These services are provided for the convenience of the programmer



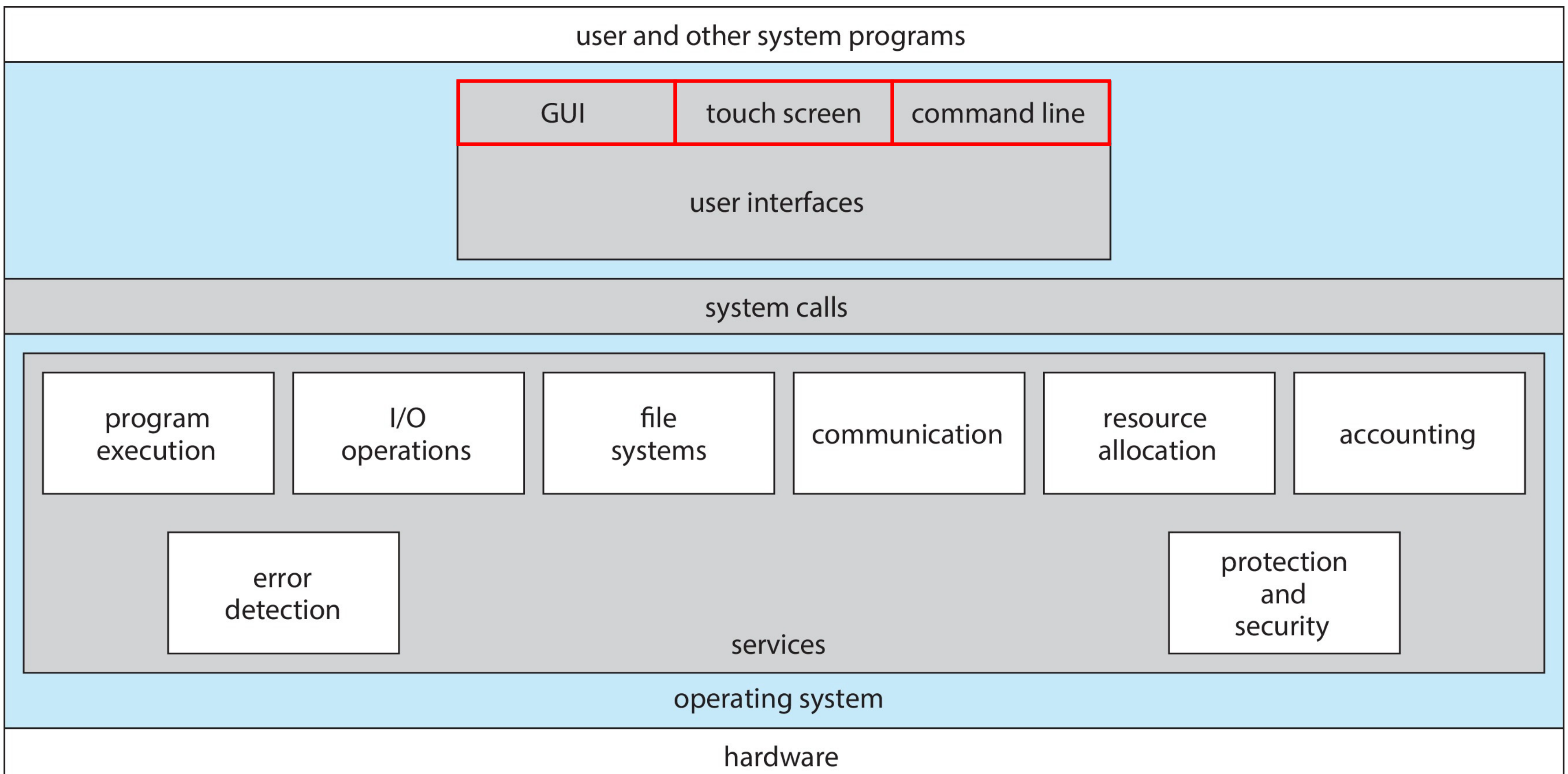
Operating System Services

- **Operating System Services (helpful to users)**
  - **User Interface** – Almost all OSes have a user interface (UI)
  - **Program Execution** – The system must be able to load a program into memory and to run that program, end execution
  - **I/O Operations** – A running program may require I/O, which may involve a file or an I/O device
  - **File-system Manipulation** – Programs need to read, write, create or delete files and directories
  - **Communications** – Processes may exchange information, on the same computer or between computers over a network
  - **Error Detection** – OS needs to be constantly aware of possible errors
    - May occur in the CPU and memory, I/O devices, or in user program
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - Debugging facilities can greatly enhance the user's ability to efficiently use the system

Operating System Services

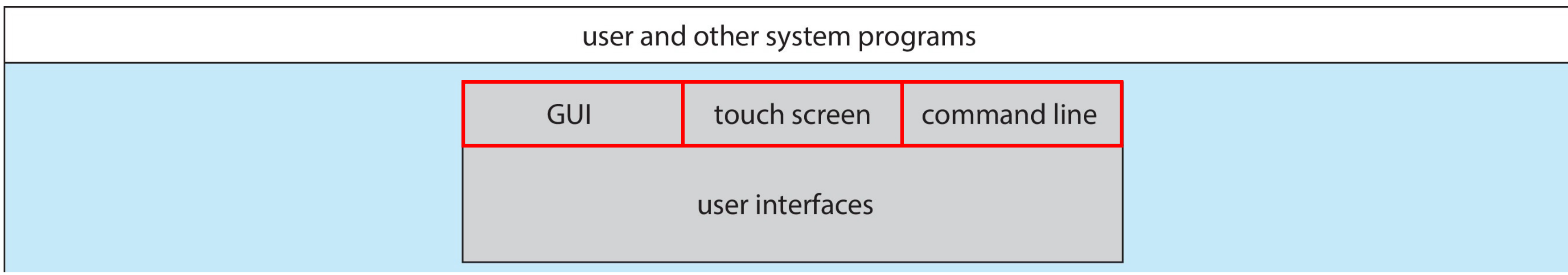
- **Operating System Services (of the OS itself)**
  - **Resource Allocation** – When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources: CPU, Memory, File Storage, I/O Devices
  - **Logging** – To keep track of which users how much and what kinds of computer resources
  - **Protection and security** – The owners of information stored in a multi-user or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

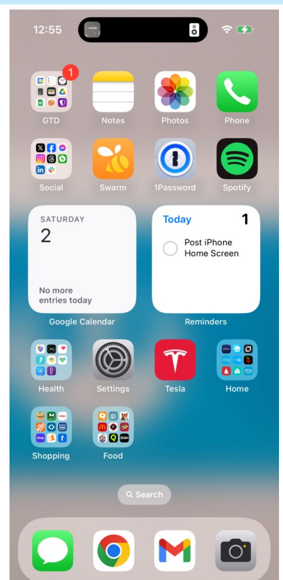
- **User Interface**
  - Command Line Interface (**`CLI`**)
  - Graphical User Interface (**`GUI`**)
  - Touch-screen Interface

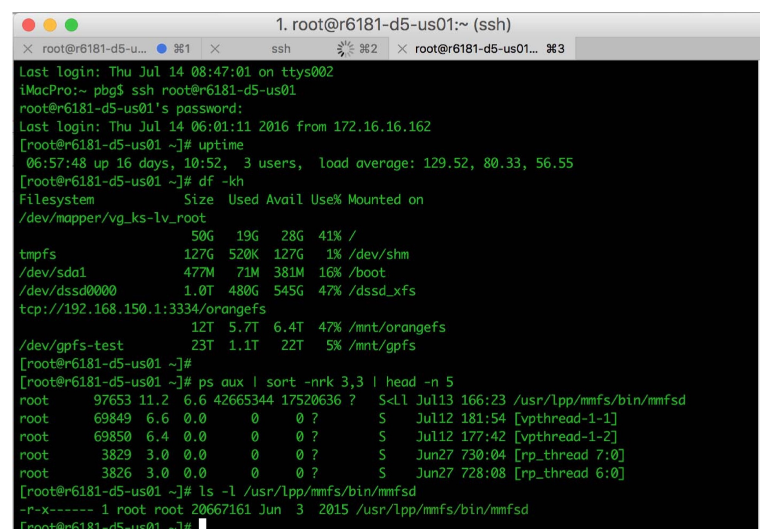


User Interface

- Command Line Interface (**CLI**)
- Graphical User Interface (**GUI**)
- Touch-screen Interface

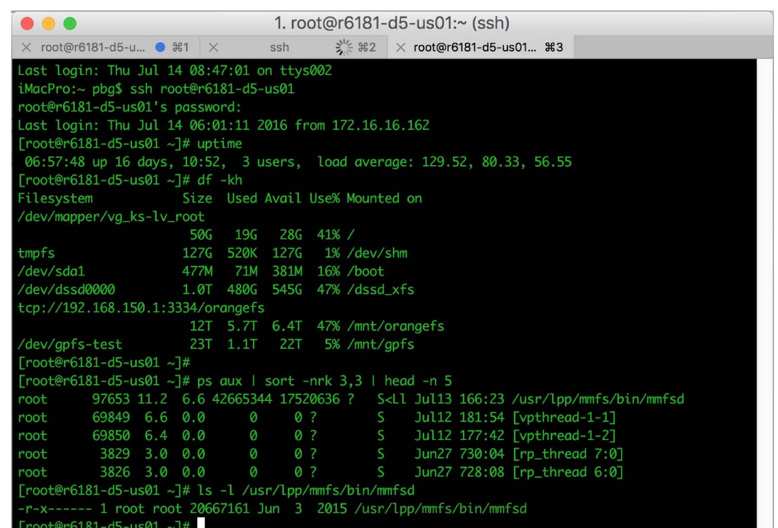






Command Line Interface (CLI)

- CLI allows direct command entry
- Primarily fetches a command from user and executes it
- Command Line Interpreter itself contains build-in commands, or it can execute **ANY** command by specifying correct paths.
- Examples:
  - UNIX Shells (bash, zsh, csh, ksh ...)
  - Windows CMD.exe
- (some) Experienced programmers **live** in the CLI
  - tmux
  - emacs
  - vim
  - ssh
  - w3m
  - ...

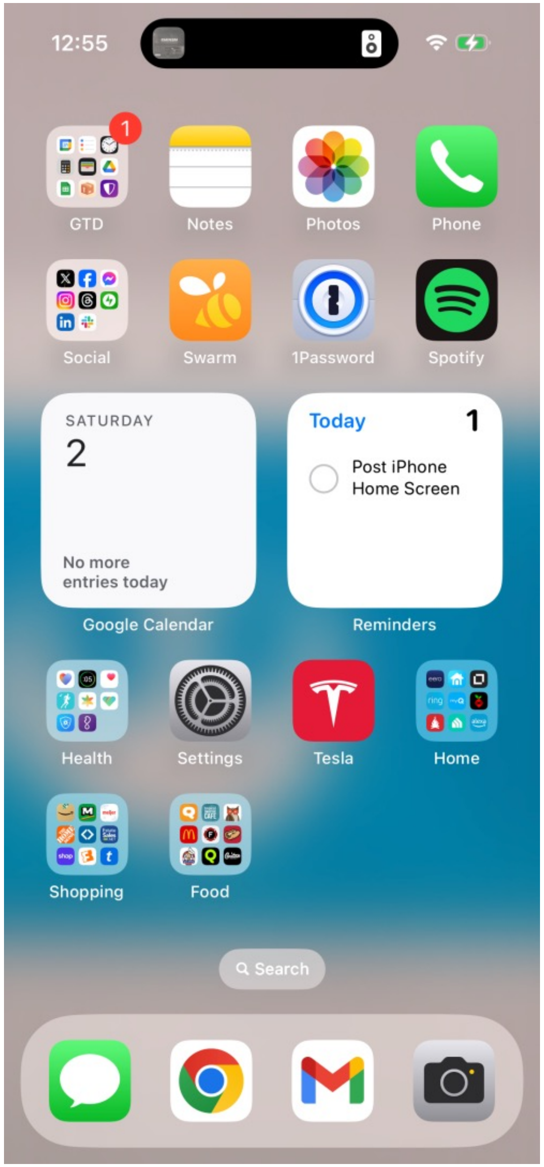


Graphical User Interface (GUI)

- User-friendly desktop metaphor interface
  - Usually mouse, keyboard, and monitor
  - Icons represent files, programs, actions, etc.
  - Various mouse buttons over objects cause various actions
  - Invented by Xerox PARC
- Many systems now include both **CLI** and **GUI** interface:
  - Microsoft Windows is GUI with CLI "CMD.exe"
  - Apple macOS is "Aqua" GUI with UNIX kernel underneath (with shells)
  - UNIX and Linux have CLI with optional GUI (GNOME, KDE, etc.)

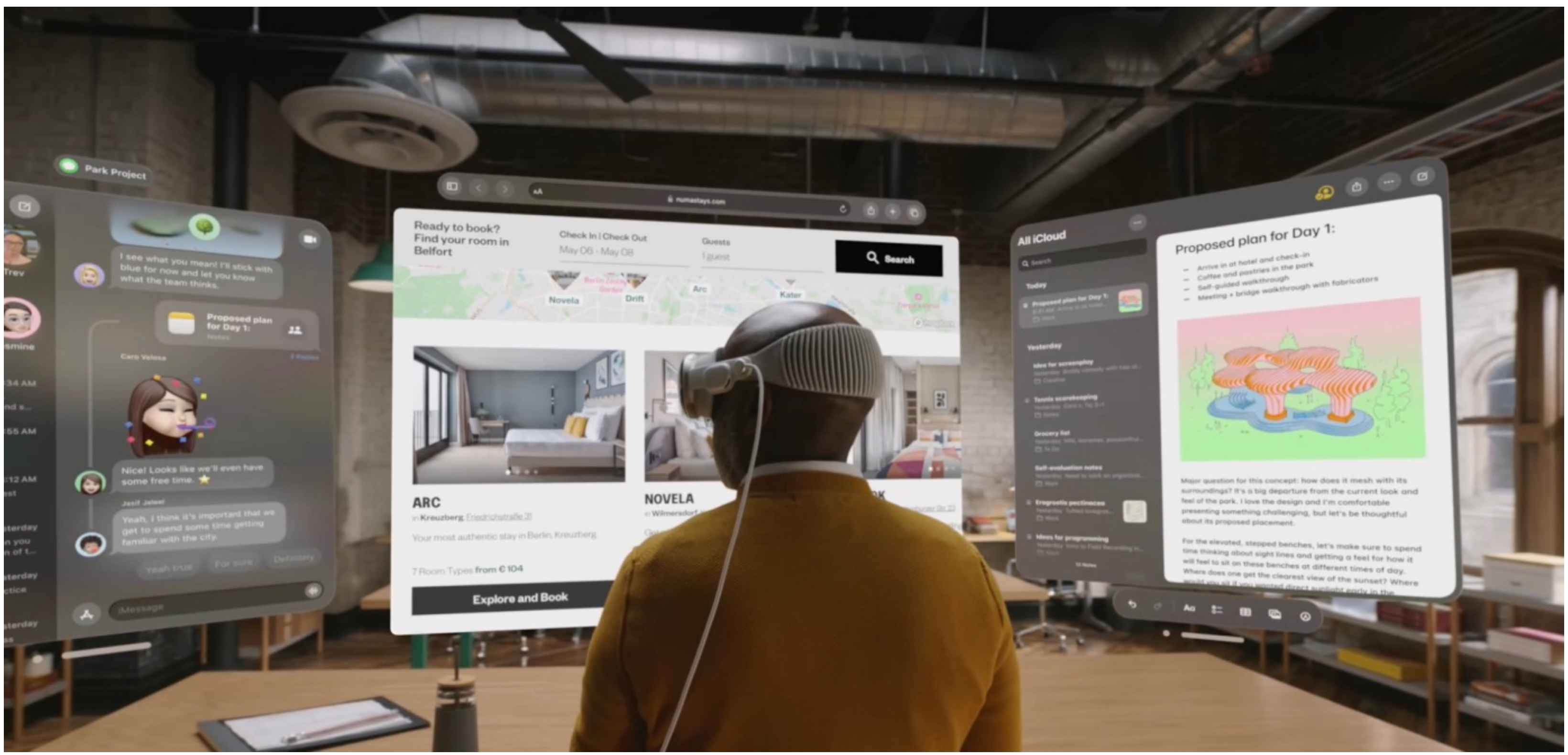
Touchscreen Interfaces

- Touchscreen devices require new interfaces
  - **Mouse** not possible or extremely inconvenient
  - **Physical keyboard** extremely inconvenient
    - Use **Virtual keyboard** (on-screen) instead
  - Actions and selection based on **gestures**
- Voice commands (Siri)



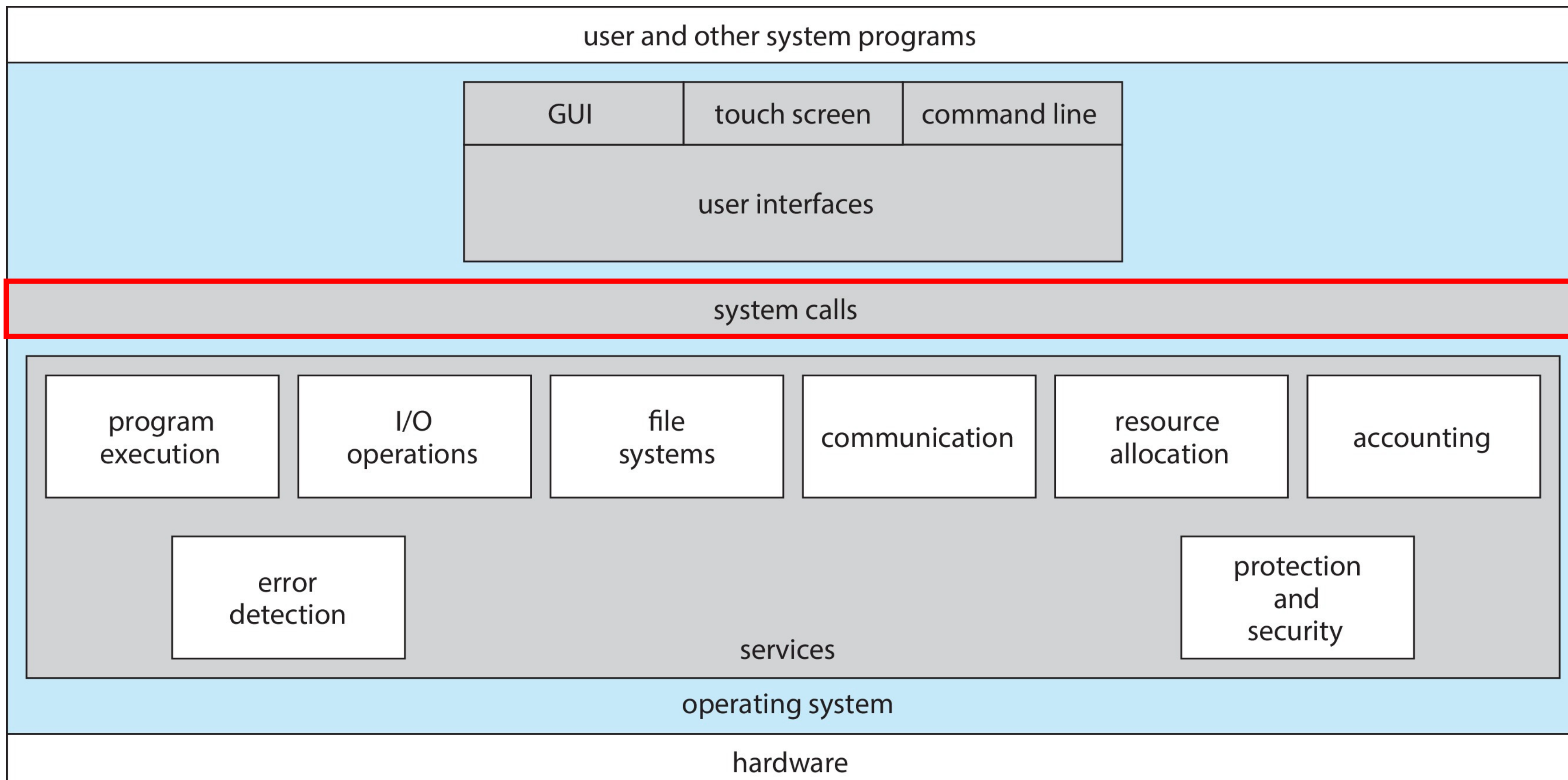
User Interface

- **Spatial Computing (VR)**
  - New trend (*Apple Vision Pro*, *Meta Oculus*)
  - Eye tracking, hand gesture as Human Computer Interaction (HCI)
  - Hoax? or the Future?



System Calls

- System Calls provide **programming interface** to the services provided by the OS



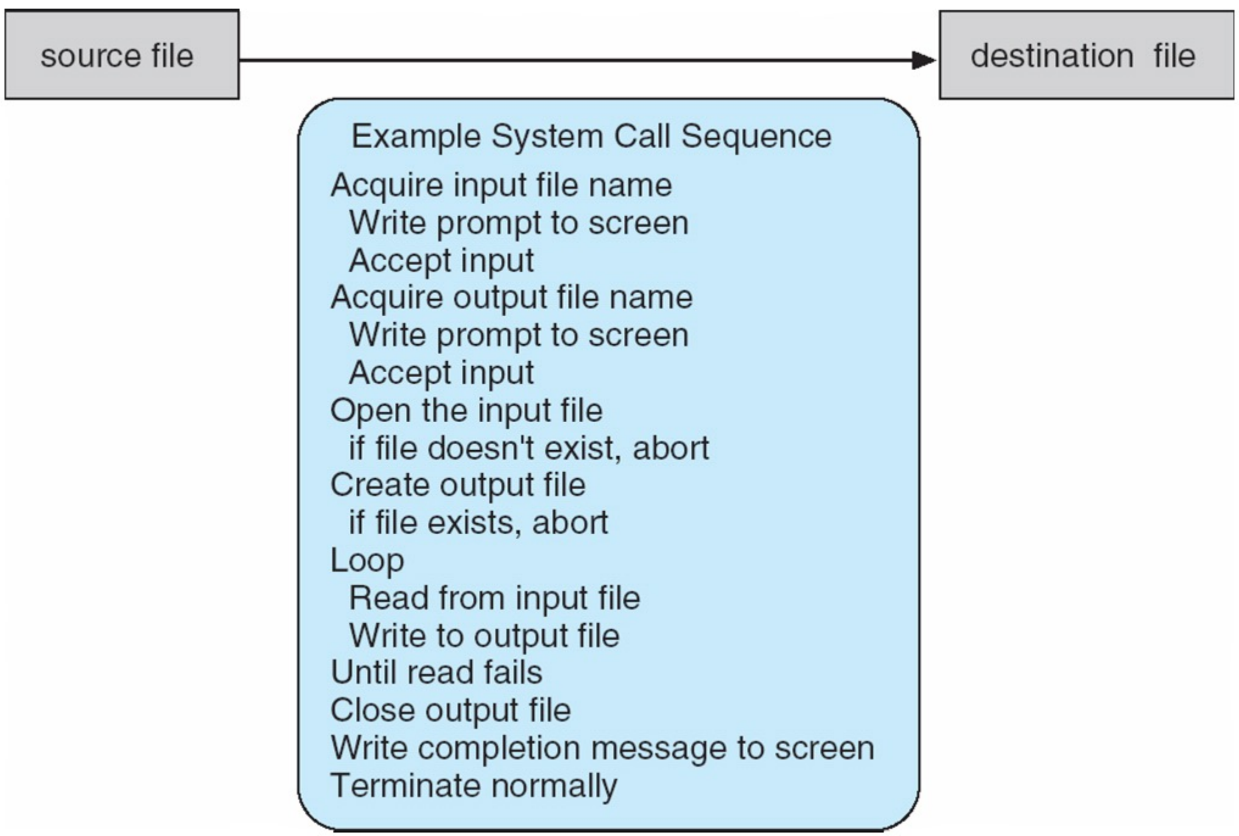
System Calls

- System Calls provide **programming interface** to the services provided by the OS
  - Typically written in a high-level language (C or C++)
  - Mostly accessed by programs via a high-level **Application Programming Interface** (**API**) rather than direct system call use
  - **Recall** that System Call is one of three types of interrupts. The other two are **Hardware Interrupt** and **Exceptions** (Traps).
  - When a user program wants to perform privileged instructions or access hardware, etc., it must do so via **System Calls.**
- Three most common APIs:
  - **Win32** API for Windows
  - **POSIX** API for POSIX-based systems
    - **POSIX: P**ortable **O**perating **S**ystem **I**nterface (IEEE STD 1003, ISO/IEC 9945)
    - including virtually all versions of UNIX, Linux and macOS
    - defines both the system- and user-level API
  - **Java API** for the Java Virtual Machine (JVM)

System Calls Example

- copy the input file (`in.txt`) to the output file (`out.txt`)

```
$ cp in.txt out.txt
```



System Calls Example

- copy the input file (`in.txt`) to the output file (`out.txt`)

```
$ strace cp in.txt out.txt
execve("/usr/bin/cp", ["cp","in.txt","out.txt"],0x7fbd134460 ..) = 0
geteuid()                                        = 1000
openat(AT_FDCWD, "in.txt", O_RDONLY)             = 3
openat(AT_FDCWD, "out.txt", O_WRONLY|O_TRUNC) = 4
fadvise64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
read(3, "hello world\n", 131072)                 = 12
write(4, "hello world\n", 12)                    = 12
read(3, "", 131072)                              = 0
close(4)                                         = 0
close(3)                                         = 0
lseek(0, 0, SEEK_CUR)                            = -1 ESPIPE (Illegal seek)
close(0)                                         = 0
close(1)                                         = 0
close(2)                                         = 0
exit_group(0)                                    = ?
```

`strace` is system program that traces system calls invoked by a program.
In this example, `strace` traces syscalls invoked by `cp in.txt out.txt`.
(The above output is redacted to show only relevant syscalls.)

System Calls vs API

- Even simple programs (e.g., cp) make heavy use of system calls.
- Most programmers never see this level of detail.
- Typically, application developers design programs according to an application programming interface (**API**), rather than **directly** invoking the system call.

System Calls vs API

- Typically, application developers design programs according to **an application programming interface (API)**, rather than directly invoking the system call.

```c
/* write_api.c */
#include <unistd.h>
#include <stdio.h>
int main() {
    const char *msg = "Hello, world!\n";
    size_t len = 14;
    int fd = STDOUT_FILENO;  /* 1 */




    write(fd, msg, len);




    return 0;
}
```

```
$ make write_api
$ ./write_api
Hello, world!
```

System Calls vs API

- Typically, application developers design programs according to an application programming interface (API), rather than **directly invoking the system call**.

```c
/* write_direct.c */
#include <unistd.h>
#include <stdio.h>
int main() {
    const char *msg = "Hello, world!\n";
    size_t len = 14;
    int fd = STDOUT_FILENO;  /* 1 */
    asm volatile (
            "movq $1, %%rax\n"
            "movq %0, %%rdi\n"
            "movq %1, %%rsi\n"
            "movq %2, %%rdx\n"
            "syscall\n"
            :
            : "g"((long)fd), "g"((long)msg), "g"((long)len)
            : "rax", "rdi", "rsi", "rdx", "r10", "r8", "r9", "memory"
            );
    return 0;
}
```

```
$ make write_direct
$ ./write_direct
Hello, world!
```

System Calls

System Calls vs API

- Even simple programs (e.g., cp) make heavy use of system calls.
- Most programmers never see this level of detail.
- Typically, application developers design programs according to an application programming interface (**API**), rather than **directly** invoking the system call (via Assembly Code).
- In most cases, the API and its corresponding system call share the same name, e.g., `open()`, `read()`, `write()`, `close()`, etc.
  - Hence, in most cases, we refer to those **APIs** as **System Calls** as well.

System Calls

Example of Standard API

- As an example of a standard API, consider `write()` function that is available in UNIX and Linux systems.
- The API for `write()` can be obtained by `` `man 2 write` ``

ubuntu@render: ~ — ssh render — 80×24

```
WRITE(2)                   Linux Programmer's Manual                  WRITE(2)

NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write() writes up to count bytes from the buffer starting at buf to the
       file referred to by the file descriptor fd.

       The number of bytes written may be less than  count  if,  for  example,
       there  is  insufficient space on the underlying physical medium, or the
       RLIMIT_FSIZE resource limit is encountered (see setrlimit(2)),  or  the
       call was interrupted by a signal handler after having written less than
       count bytes.  (See also pipe(7).)

       For a seekable file (i.e., one to which lseek(2) may  be  applied,  for
       example,  a  regular  file) writing takes place at the file offset, and
       the file offset is incremented by the number of bytes actually written.
 Manual page write(2) line 1 (press h for help or q to quit)
```

Example of Standard API

- As an example of a standard API, consider `write()` function that is available in UNIX and Linux systems.
- The API for `write()` can be obtained by \``man 2 write`\`
  - Why not \``man write`\`? Because man page is divided into several sections, only **section 2** is for system calls. \``man write`\` will display the manpage of the utility program write (located in \``/usr/bin/write`\`)

```
DESCRIPTION
     The man utility finds and displays online manual documentation pages.  If
mansect is provided, man restricts the search to the specific section of the
manual.

     The sections of the manual are:
           1.   General Commands Manual
           2.   System Calls Manual
           3.   Library Functions Manual
           4.   Kernel Interfaces Manual
           5.   File Formats Manual
           6.   Games Manual
           7.   Miscellaneous Information Manual
           8.   System Manager's Manual
           9.   Kernel Developer's Manual
```

Example of Standard API

```c
#include <unistd.h>

ssize_t     write(int fd, const void *buf, size_t count);
```

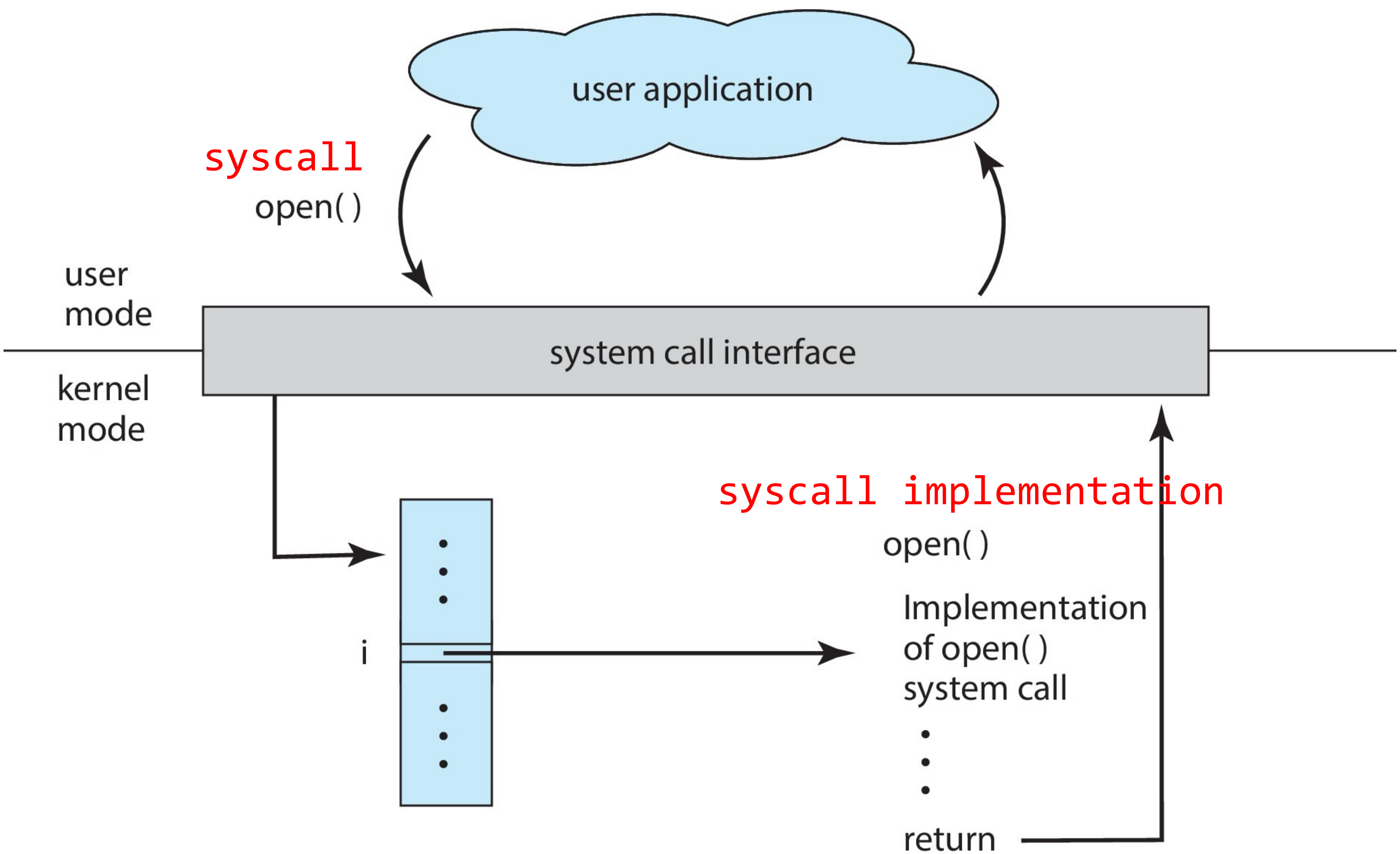
return type — function name — parameters

A program that uses `write()` function must include the *<unistd.h>* header file, as this header file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `write()` are as follows:

- `int fd` : *the file descriptor to be written*
- `const void *buf` : *a buffer from which the data will be read*
- `size_t count` : *the maximum number of bytes to be read from buffer*

On a successful write, the number of bytes written is returned. If an error occurs, `write()` returns `-1`.

API – System Calls – OS Relationship



System Call Implementation

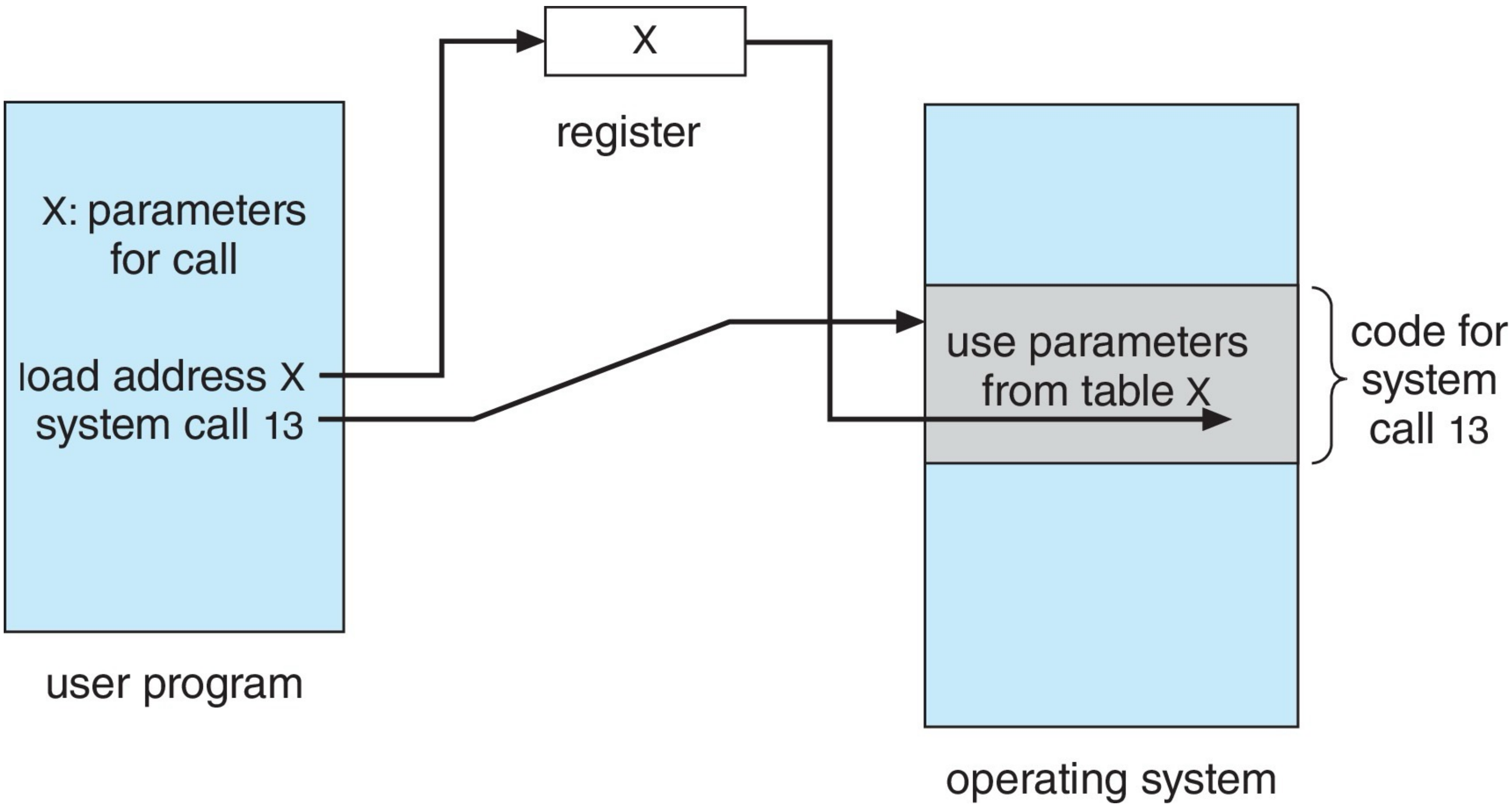
- Typically, a number is associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller don't need to know how the system call is implemented
  - Just needs to obey the API and understand what OS will do as a result
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

System Call Parameter Passing

- Often times, more information is required than simply identity of the desired system call
- Three general methods used to pass parameters to the OS:
  - Simplest: pass the parameters in registers
    - In some cases, there may be more parameters than registers
  - Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach is taken by Linux and Solaris
  - Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system
  - Block and stack methods do not limit the number of parameters being passed.

System Call Parameter Passing

- Passing via Table



System Calls

- **Types of System Calls**
  - Process Control
  - File Management
  - Device Management
  - Information Maintenance
  - Communications
  - Protection

System Calls

- **Types of System Calls**
  - Process Control
    - `fork(), exec(), exit(), wait() ...`
  - File Management
    - `open(), close(), read(), write(), stat(), mkdir() ...`
  - Device Management
    - `ioctl(), read(), write() ...`
  - Information Maintenance
    - `getpid(), alarm(), sleep(), time() ...`
  - Communications
    - `pipe(), shm_open(), mmap() ...`
  - Protection
    - `chmod(), unmask(), chown() ...`

Examples of Different Types of System Calls

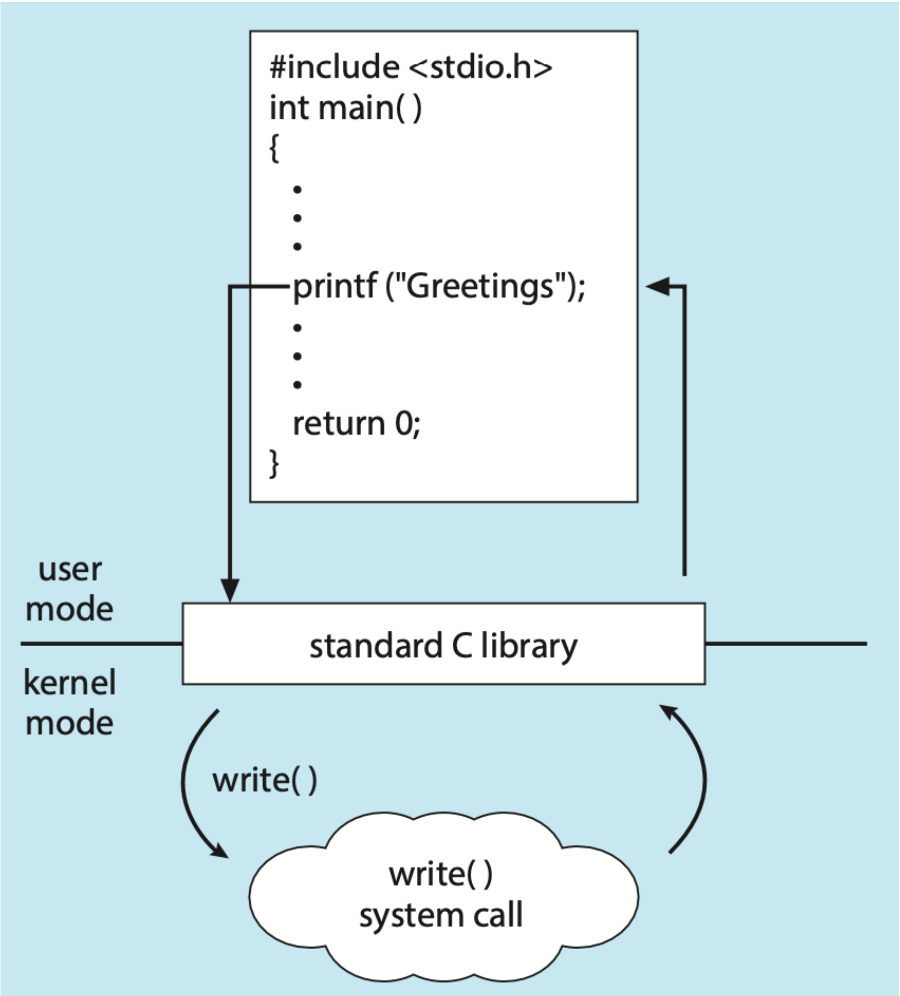
EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	**Unix**
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communications	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

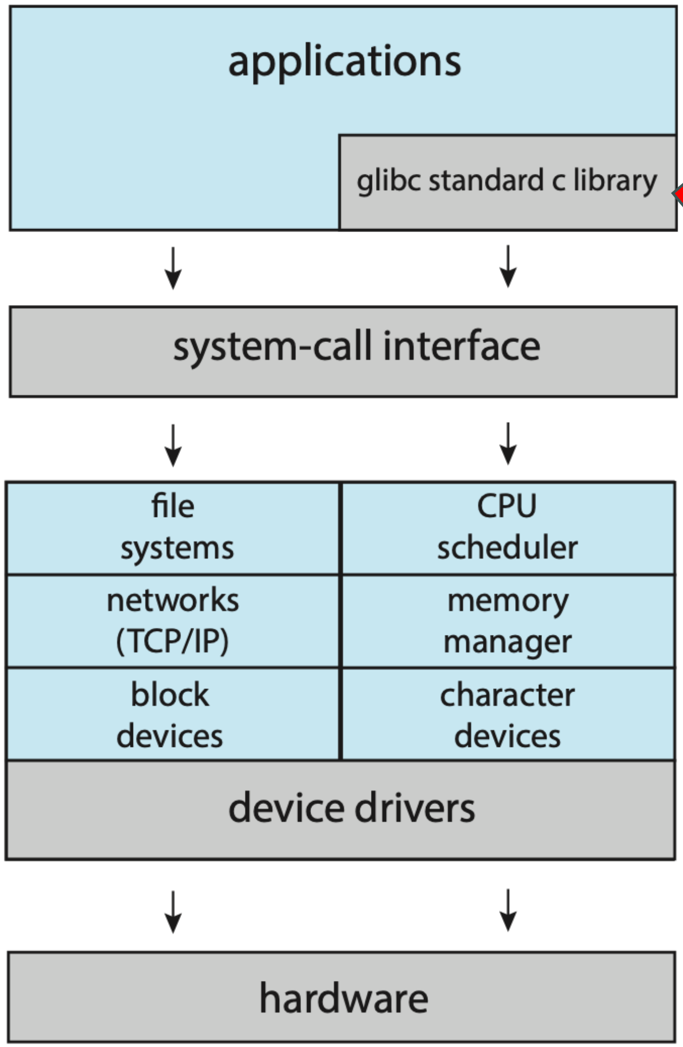
Standard C Library Example

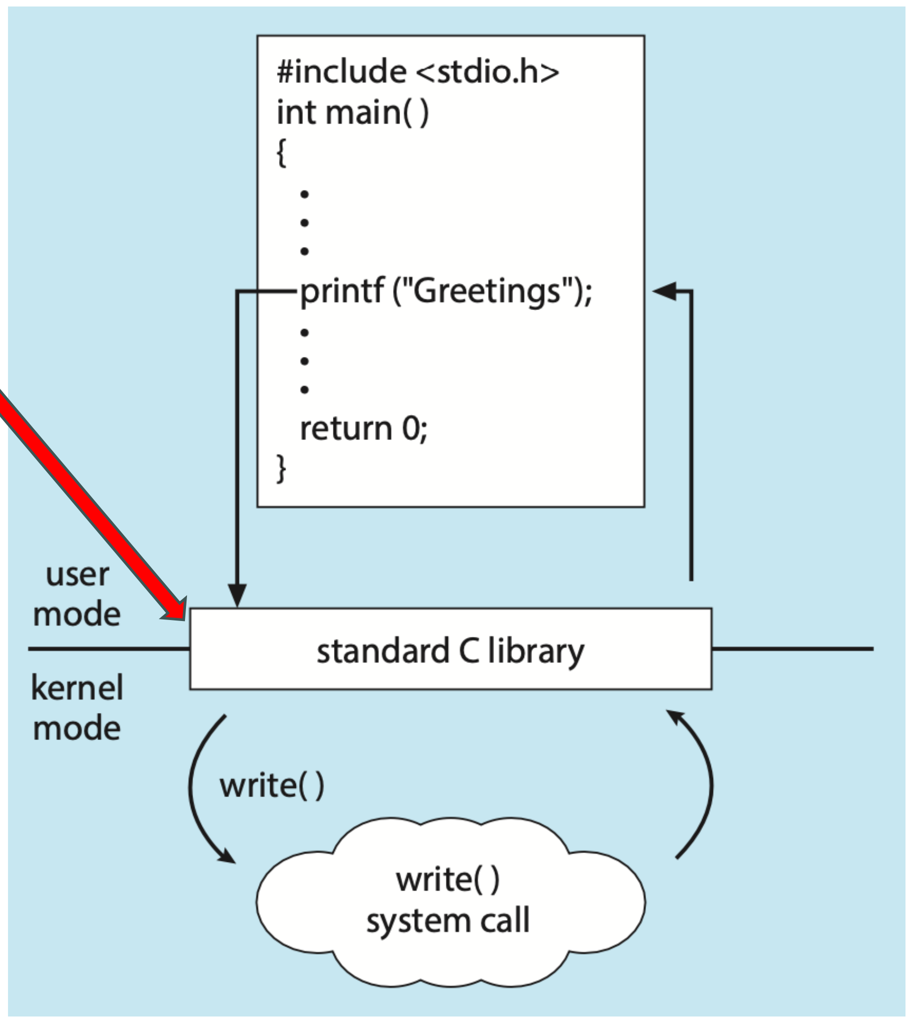
- C program invoking `printf()` library call, which calls `write()` system call



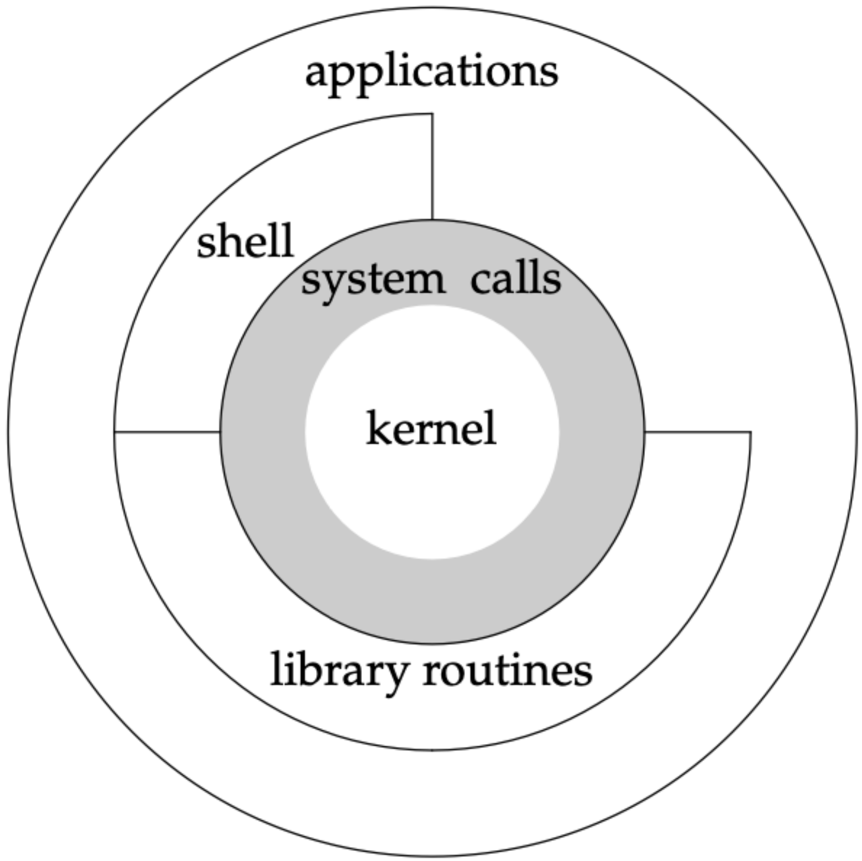
Standard Library vs System Calls

- Programs can either invoke the **Standard C Library**, or **Syscalls**.





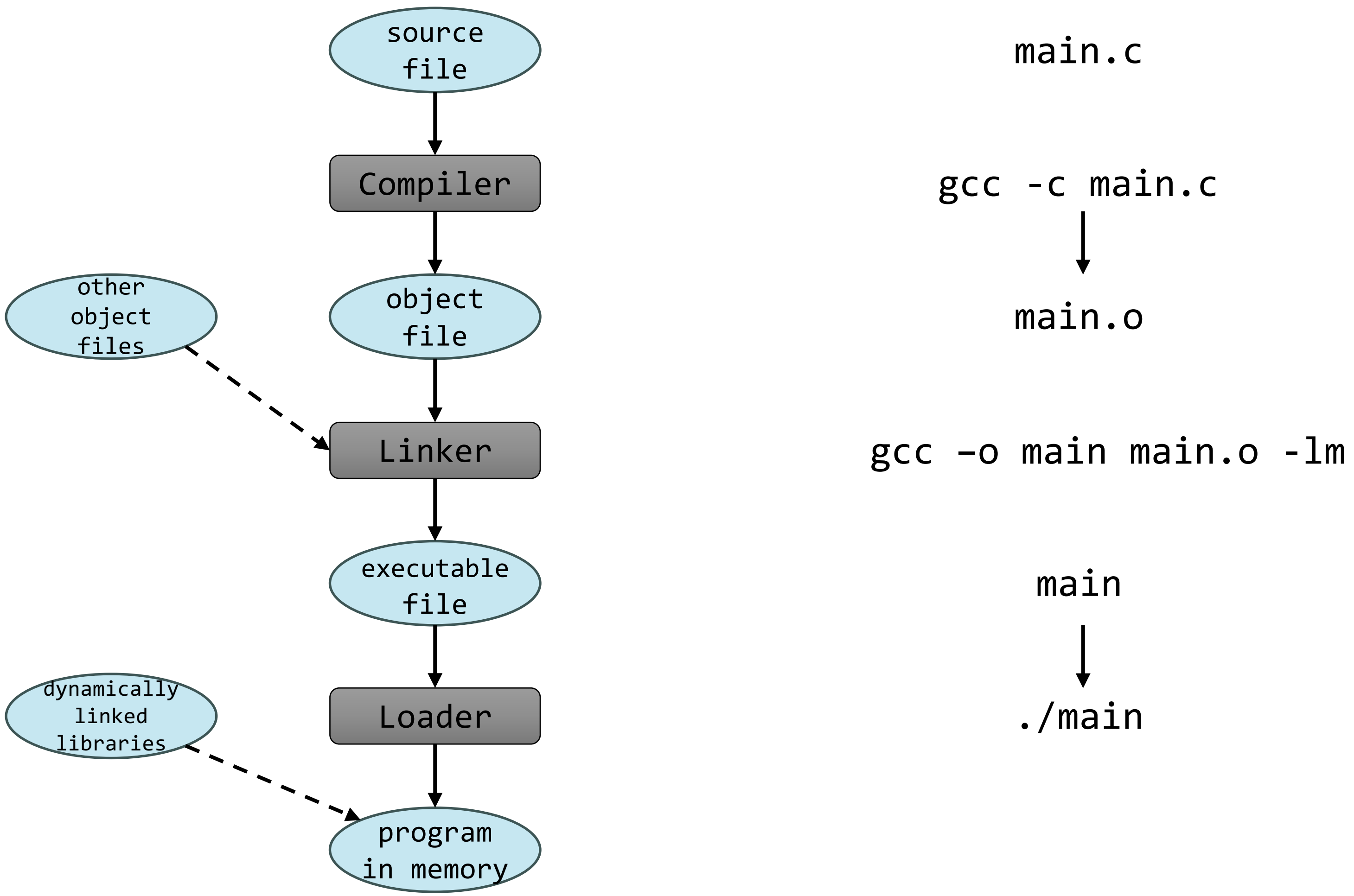
Standard Library vs System Calls



Linkers and Loaders

- Source code compiled into object files designed to be loaded into any physical memory location – relocatable object file
- Linker(链接器) combines these into single binary executable file
- Program resides on secondary storage as binary executable
- Must be brought into memory by loader(加载器) to be executed
  - Relocation assigns final addresses to program parts and adjust code and data in program to match those addresses
- Modern general purpose systems don't link libraries into executables
  - Rather, dynamically linked libraries (in Windows, DLLs) are loaded as needed, shared by all that uses the same library
- Object, executable files have standard formats, so operating system knows how to load and start them

Linkers and Loaders



Linkers and Loaders


```
$ cat main.c
```

```
$ cat main.c
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

Linkers and Loaders


```
$ cat main.c
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
$ gcc -c main.c -o main.o
```

Linkers and Loaders


```
$ cat main.c
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
$ gcc -c main.c -o main.o
$ objdump -d main.o

main.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # f <main+0xf>
   f:   48 89 c7                mov    %rax,%rdi
  12:   e8 00 00 00 00          call   17 <main+0x17>
  17:   b8 00 00 00 00          mov    $0x0,%eax
  1c:   5d                      pop    %rbp
  1d:   c3                      ret
```

Placeholder Address

Linkers and Loaders


```
$ cat main.c
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
$ gcc -c main.c -o main.o
$ gcc -o main main.o
```

Linkers and Loaders

```
$ cat main.c
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
$ gcc -c main.c -o main.o
$ gcc -o main main.o
$ objdump -d main
main:     file format elf64-x86-64

Disassembly of section .text:
......                                    Final Address after Linking
0000000000001149 <main>:
    1149:       f3 0f 1e fa             endbr64
    114d:       55                      push   %rbp
    114e:       48 89 e5                mov    %rsp,%rbp
    1151:       48 8d 05 ac 0e 00 00    lea    0xeac(%rip),%rax
    1158:       48 89 c7                mov    %rax,%rdi
    115b:       e8 f0 fe ff ff          call   1050 <puts@plt>
    1160:       b8 00 00 00 00          mov    $0x0,%eax
    1165:       5d                      pop    %rbp
    1166:       c3                      ret
......
```

Linkers and Loaders


```
$ cat main.c
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
$ gcc -c main.c -o main.o
$ gcc -o main main.o
$ ./main
```

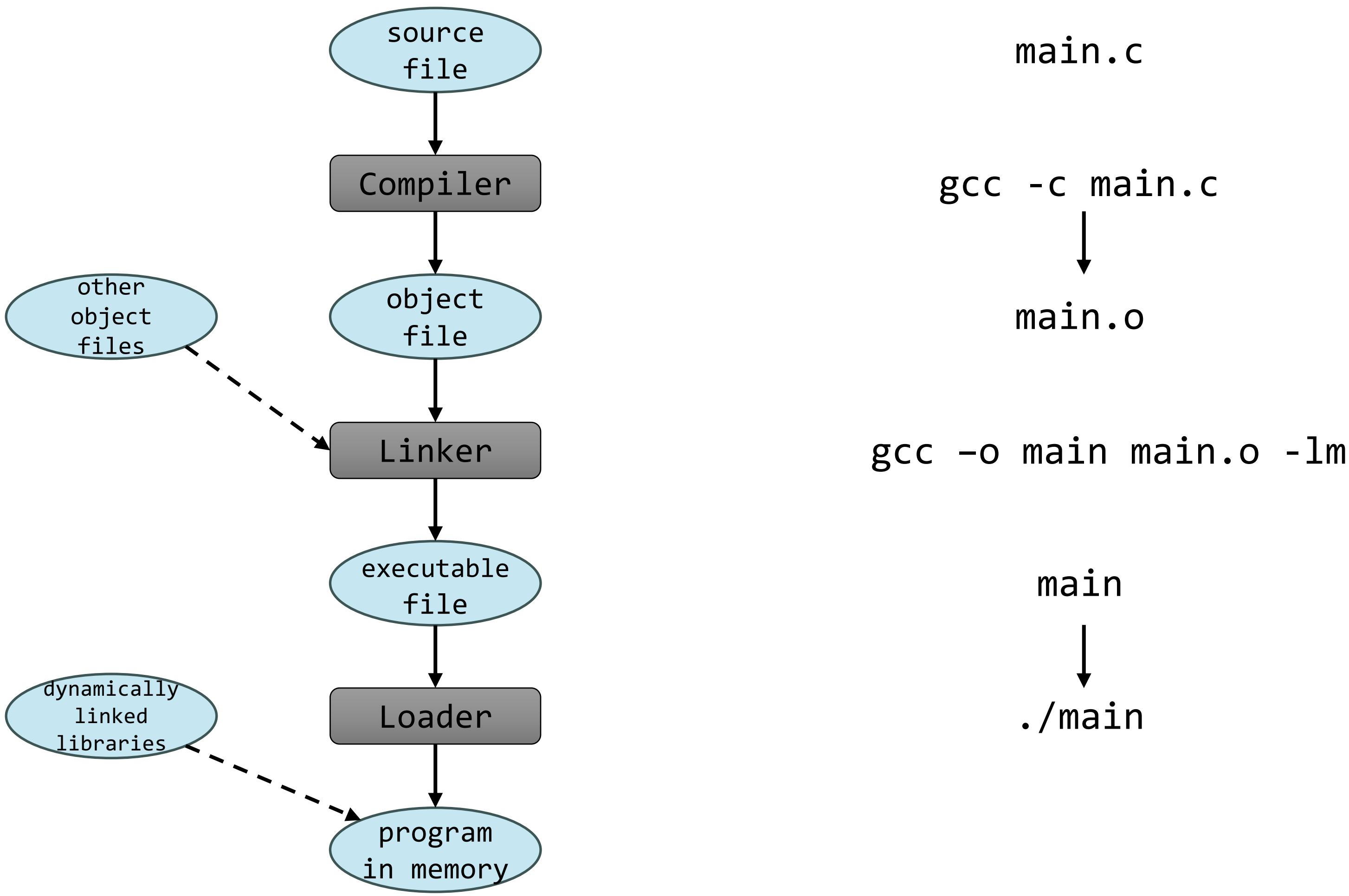
Linkers and Loaders


```
$ cat main.c
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
$ gcc -c main.c -o main.o
$ gcc -o main main.o
$ ./main
Hello, world!
```

Linkers and Loaders


```
$ cat main.c
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
$ gcc -c main.c -o main.o
$ gcc -o main main.o
$ ./main
Hello, world!
$ strace ./main
execve("./main", ["./main"], 0x7ffe2d88f8f0 /* 30 vars */) = 0
brk(NULL)                                       = 0x5598cb9d2000
...
write(1, "Hello, world!\n", 14Hello, world!)          = 14
exit_group(0)                                   = ?
+++ exited with 0 +++
```

Linkers and Loaders



Thank you!