Resource Virtualization with Containers

Tanu Malik

School of Computing, DePaul University

Visiting Faculty, CSE, IIT, Delhi

COV882

- Course website: https://dice.cs.depaul.edu/courses/882/index.html
- Course mailing list: 2201-cov882@courses.iitd.ac.in
- Team: 2201-COV882 SPECIAL MODULE IN SOFTWARE SY.
- Baadal for HW (request VM)
  - Ubuntu 20.04 1 CPU 2GM memory 80 GB HDD
- Discord for discussions: https://discord.gg/sydFh5rq
- Moodle for submissions
- Send email: tmalik@cse.iitd.ac.in or tanu.malik@depaul.edu

Resource Virtualization with Containers

- Resource
- Virtual-ization
- Containers

Process of virtualization in which we establish views on cpus, disks, network

Why is this course important?

- Applications in cloud computing
- Applications in software engineering
- Applications in systems and security
- Applications in reproducibility
- Applications in data storage

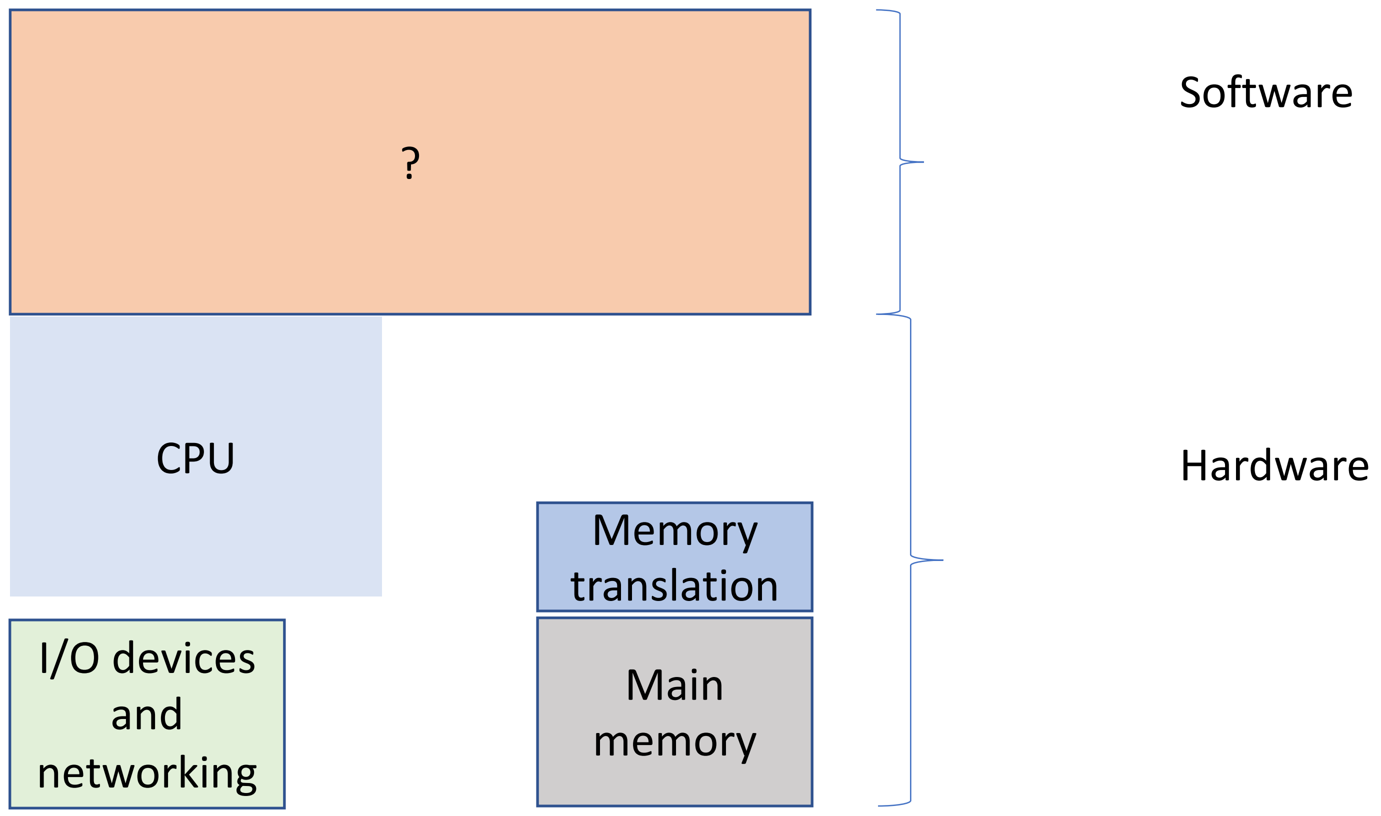
It is a multi-billion dollar industry

Course Objectives

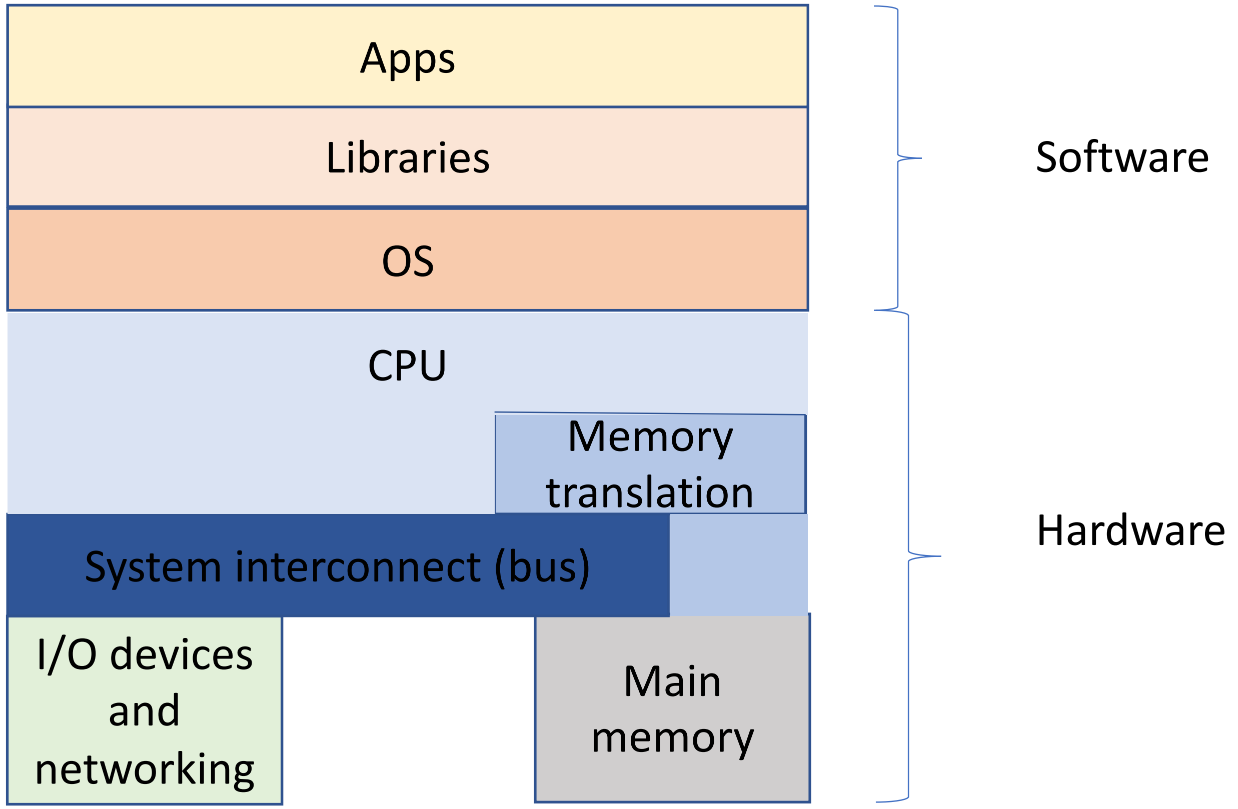
- Understand and explore system concepts necessary to understand containers
- Develop and establish different types of bare bones containers
- Explore containers in the commercial and research world

Virtualization Vs Abstraction

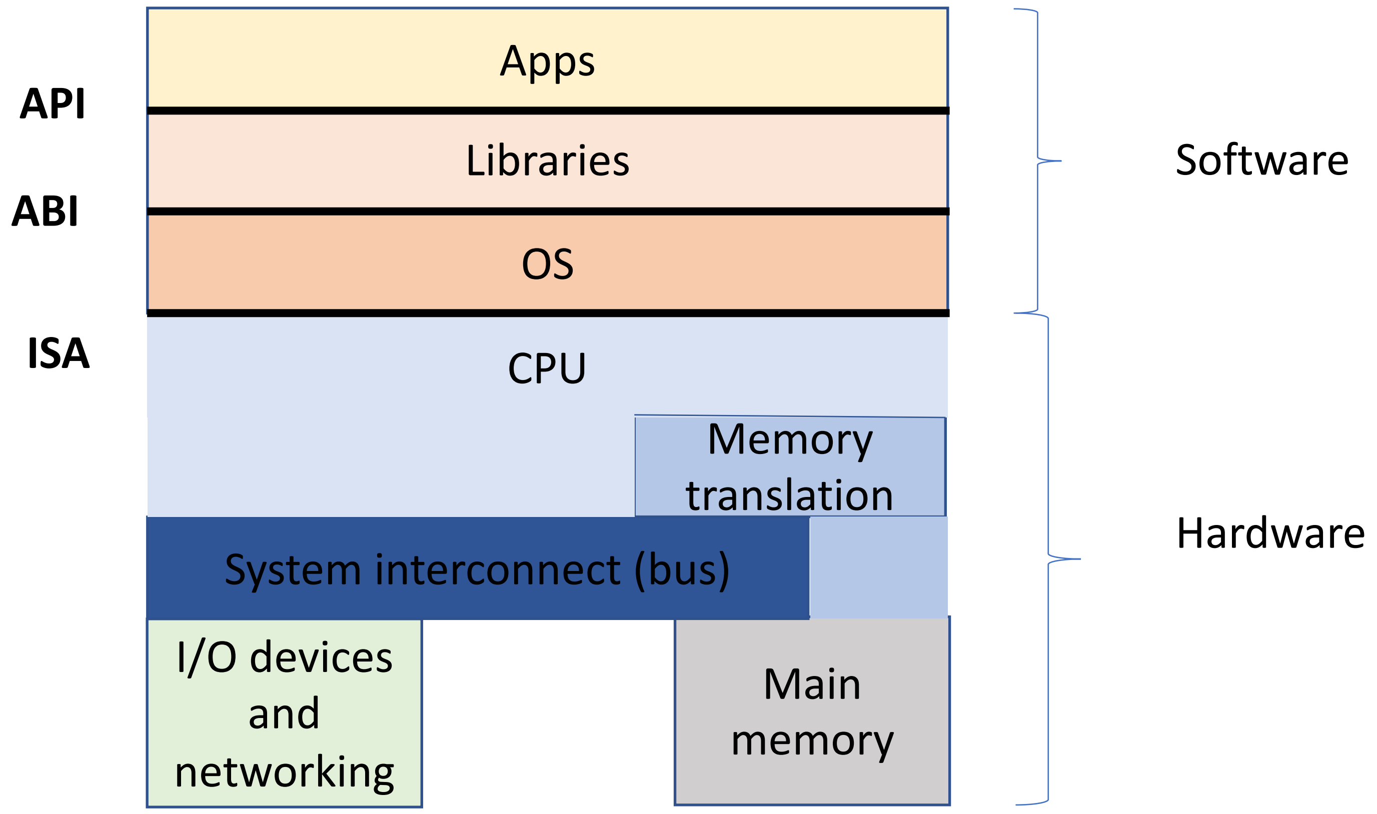
A Computer System



A Computer System



Well-defined abstraction interfaces



API Vs ABI Vs ISA

- API is used by high-level language (HLL) programmers to invoke some library or OS features.
  - An API enables compliant applications to be ported easily (via recompilation) to any system that supports the same API.
- The ABI is a compiled version of the API.
  - It is at the machine language level. With ABI, system functionalities are accessed through OS system calls.
  - A source code compiled to a specific ABI can run unchanged only on a system with the same OS and ISA.
- ISA defines a set of storage resources (e.g., registers and memory) and a set of instructions that allows manipulating data held at storage resources.
  - ISA lies at the boundary between hardware and software.

Abstraction

- Abstraction means hide the details of the layer below and only expose a logical view.

- Quiz?

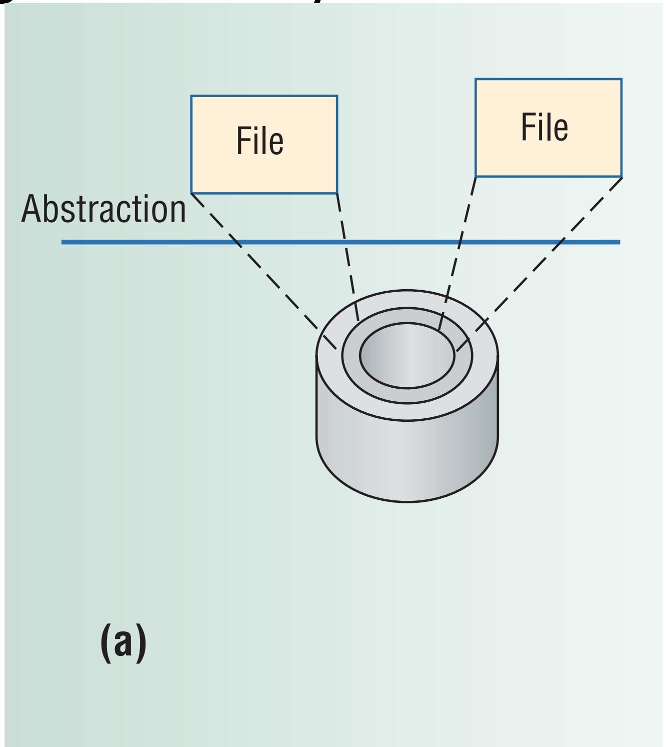
Abstraction

- Abstraction means hide the details of the layer below and only expose a logical view.

- Quiz: Have you as a programmer ever used the concept of abstraction for any other part of the computer system?

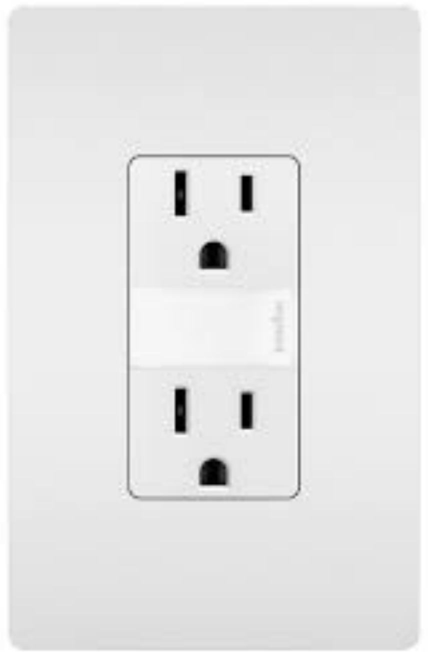
Abstraction

- The details of a hard disk are abstracted by the OS
  - Disk storage appears to applications as a set of variable-size files.
- Disk storage is locations and sizes of cylinders, sectors, and tracks or bandwidth allocations at disk controllers.
  - Programmers are not aware of it. They can simply create, read, and write files without knowledge of the way the hard disk is constructed or organized.

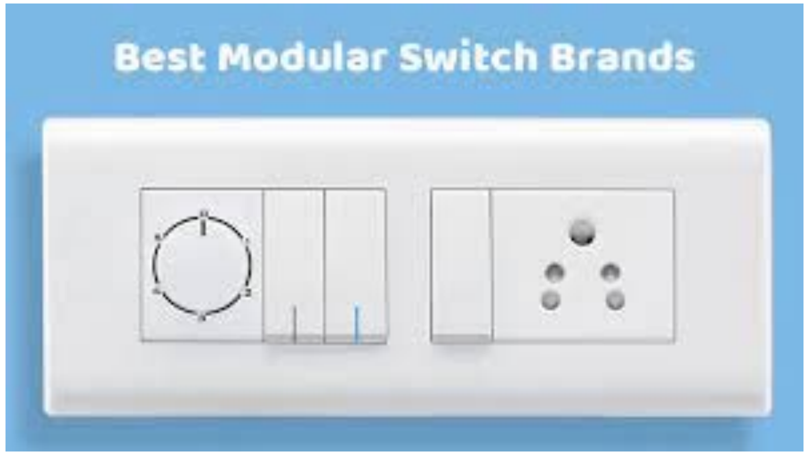


What is the disadvantage of abstraction via interfaces?

Will this socket enable access to electricity in India?



Will this socket enable access to electricity in the US?



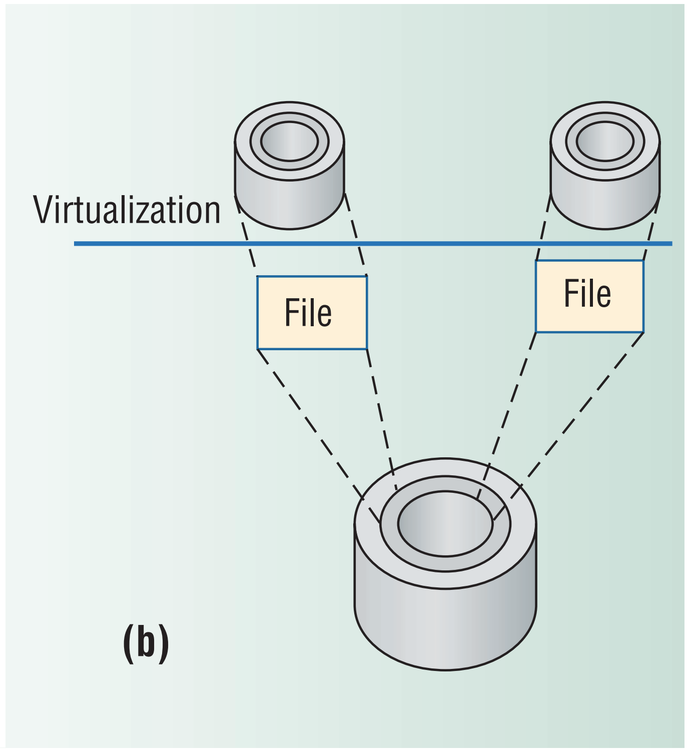
Solution? Adaptor

- Any problems with this solution?

Virtualization

- Virtualization does not necessarily aim to simplify or hide details.

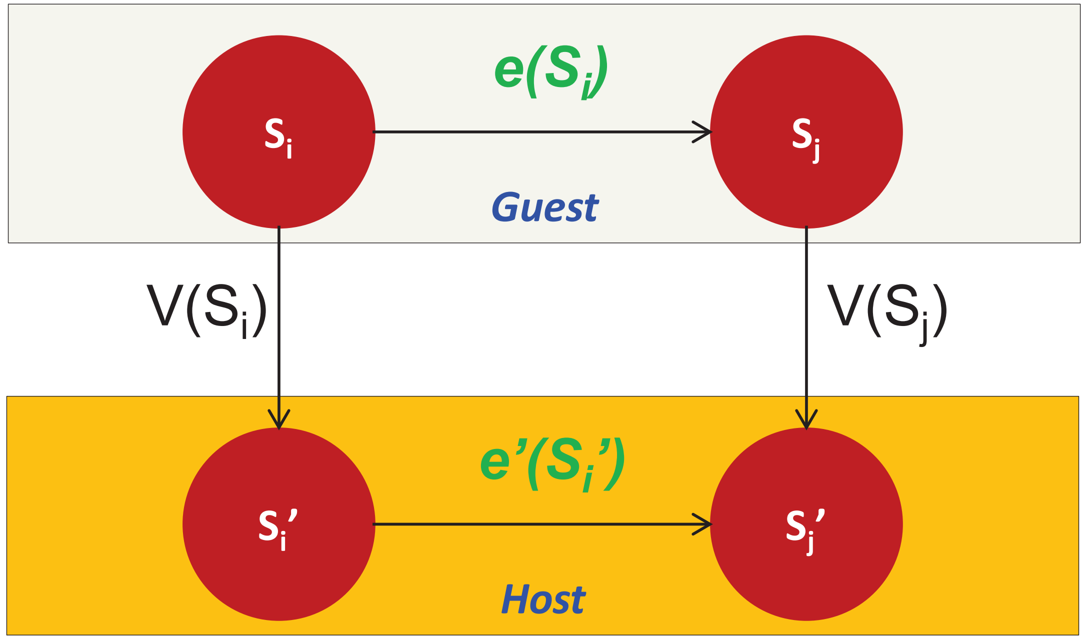
Virtualization



- Example
- An Intel CPU with Linux installed and a AMD CPU with Linux installed on it.
  - Compile source code into machine code in each machine but cannot take the machine code from one machine and simply run on another machine.

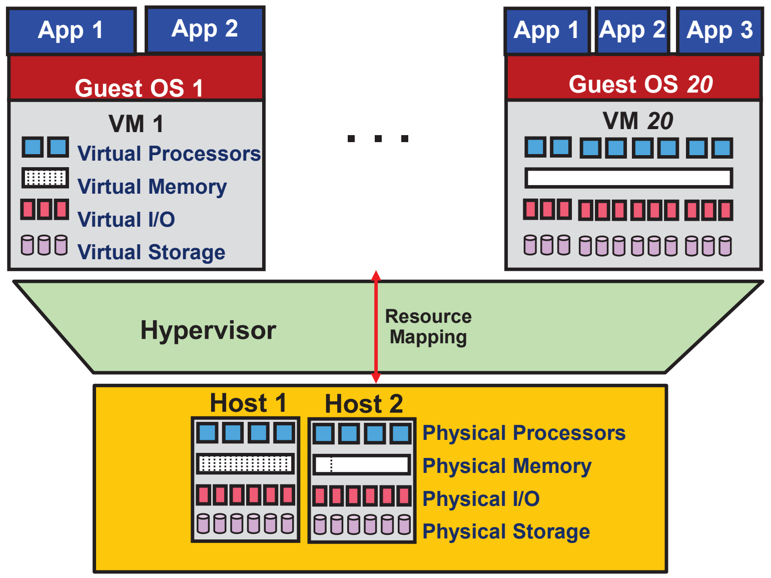
Virtualization Isomorphism

- Construction of an isomorphism that maps a virtual guest resource/system to a real host system/resource



Hypervisor is an implementation of this isomorphism

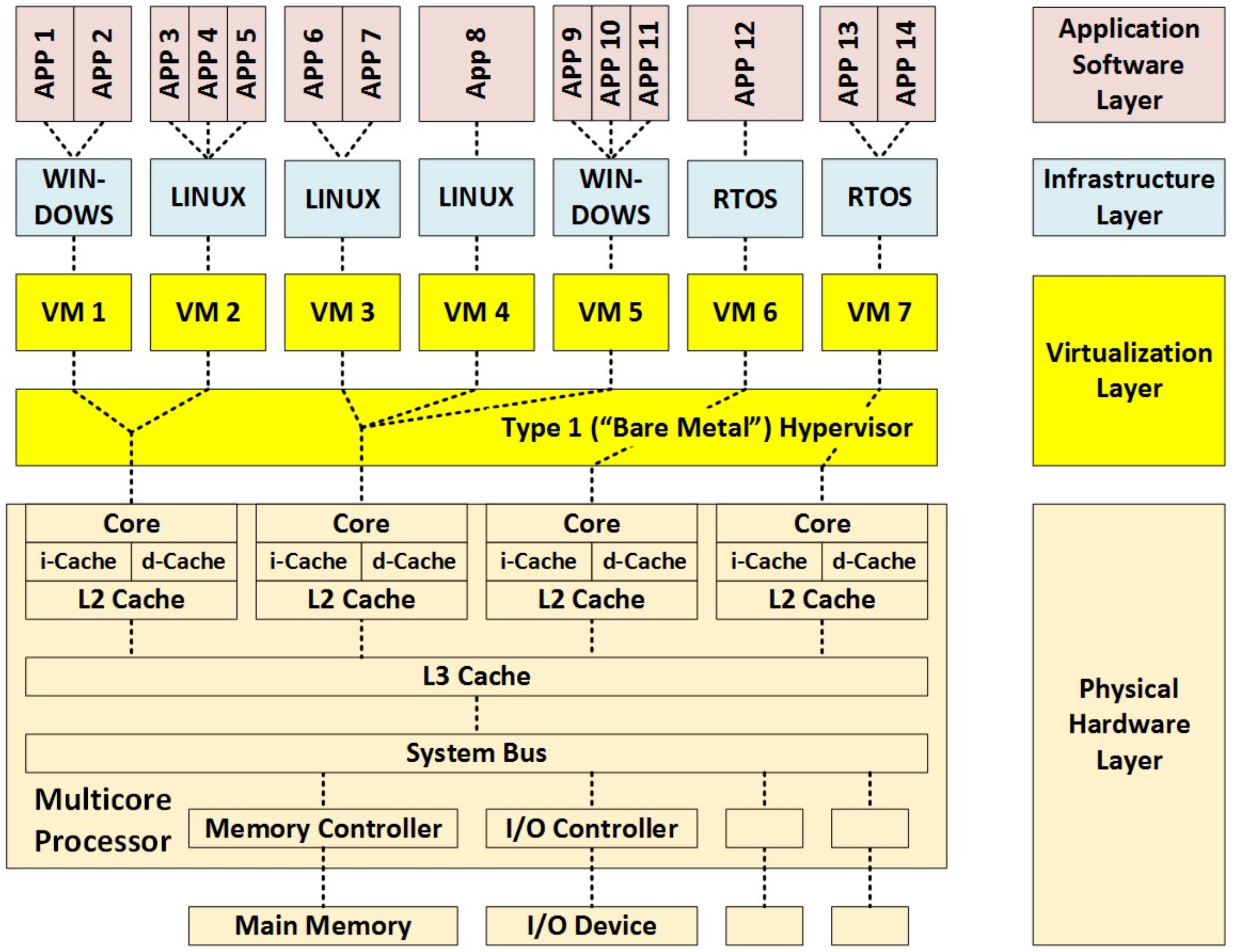
- A full set of hardware resources, including processors, memory, and I/O devices will be virtualized to provide the VM.



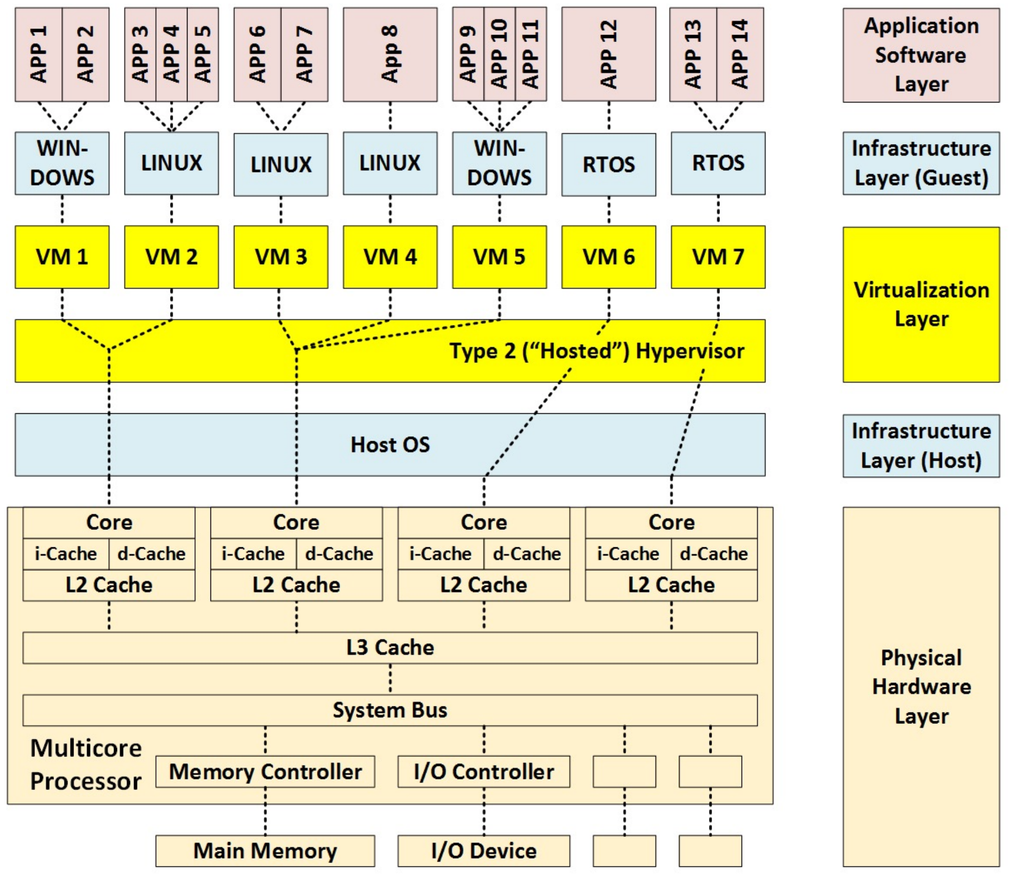
Guest machine is a GuestOS

Physical machine is a host

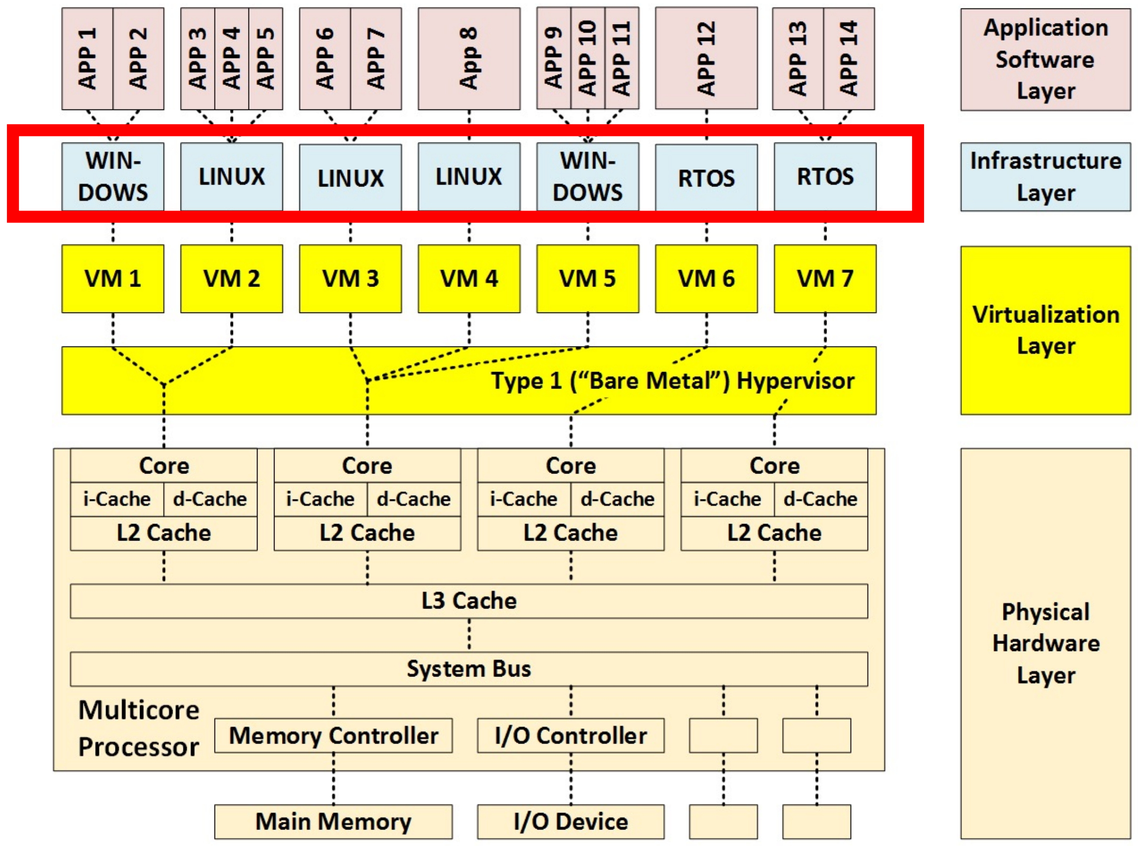
System Virtual Machine

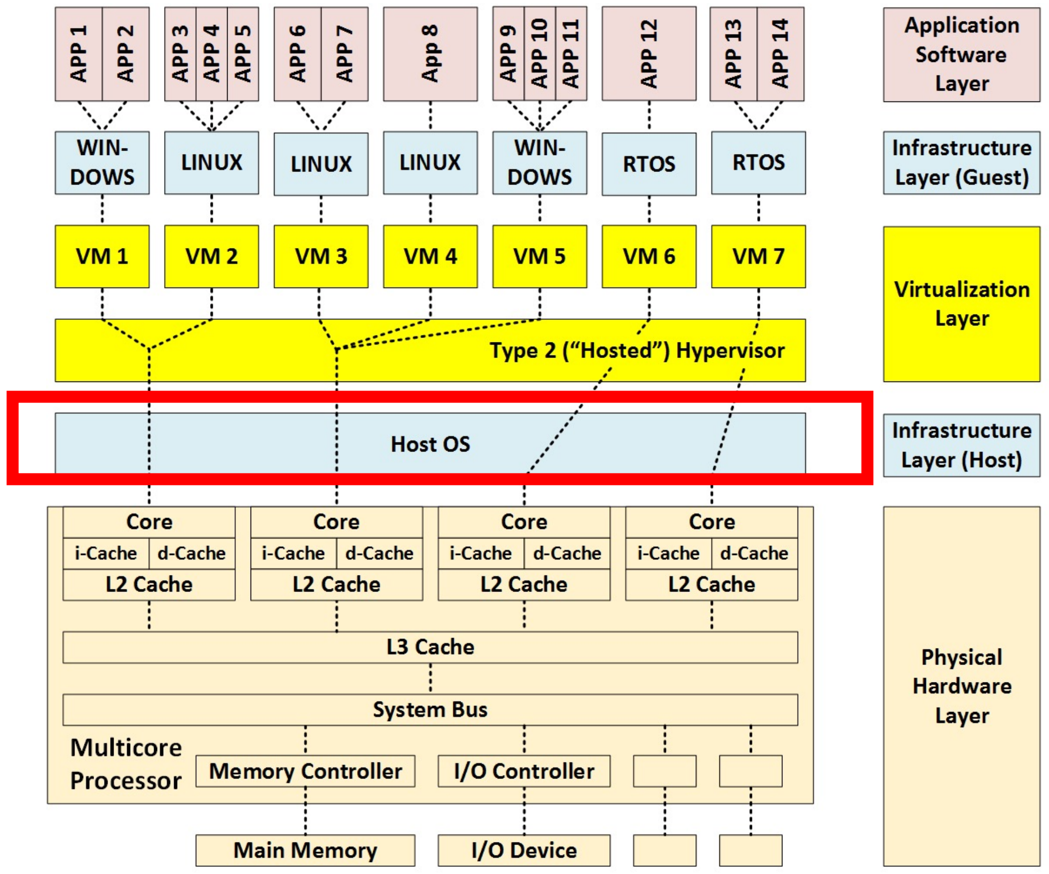


Process Virtual Machine

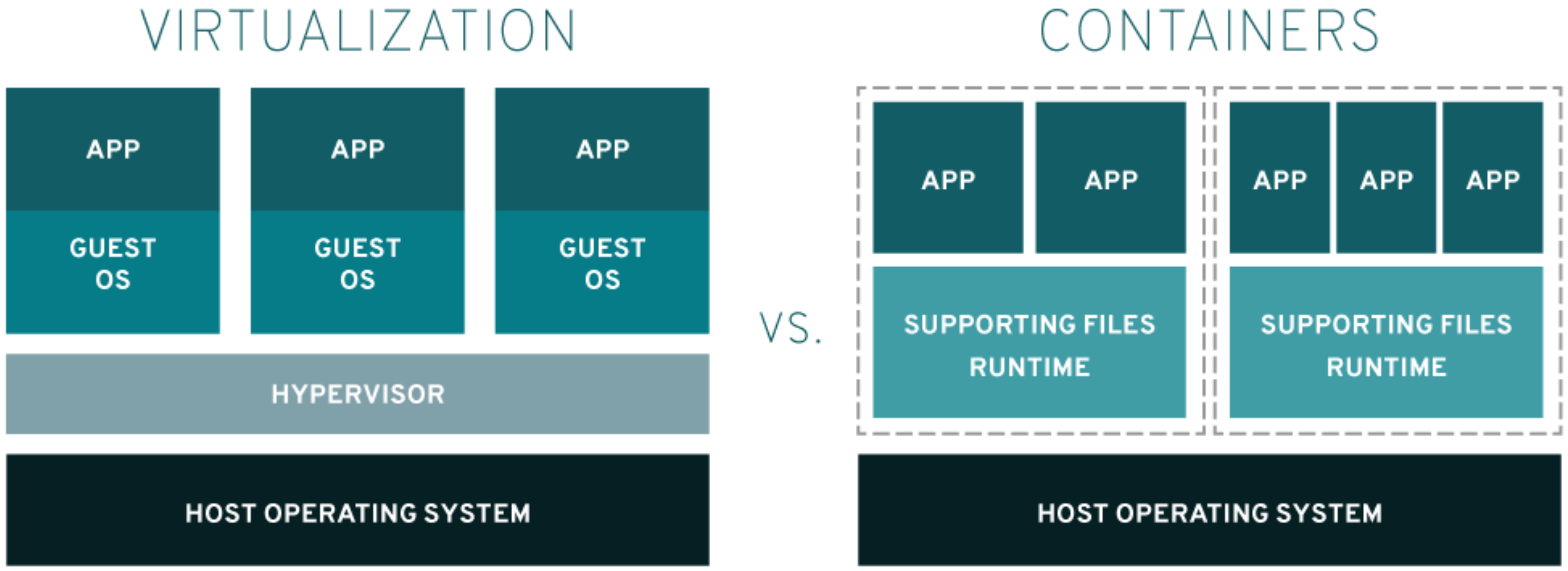


Host OS is integral





Containers: Virtualize OS behavior



What is OS behavior?

Operating System

- Definition 1: A package consisting of the central software for managing a computer’s resources and all of the accompanying standard software tools, such as command-line interpreters, graphical user interfaces, file utilities, and editors.

- Definition 2: The central software that manages and allocates computer resources (i.e., the CPU, RAM, and devices).
  - We often use the term kernel is often as a synonym for the second meaning.

The Linux Kernel

- Textbook: The Linux Programming Interface

- https://github.com/oliverralbertini/the-linux-programming-interface

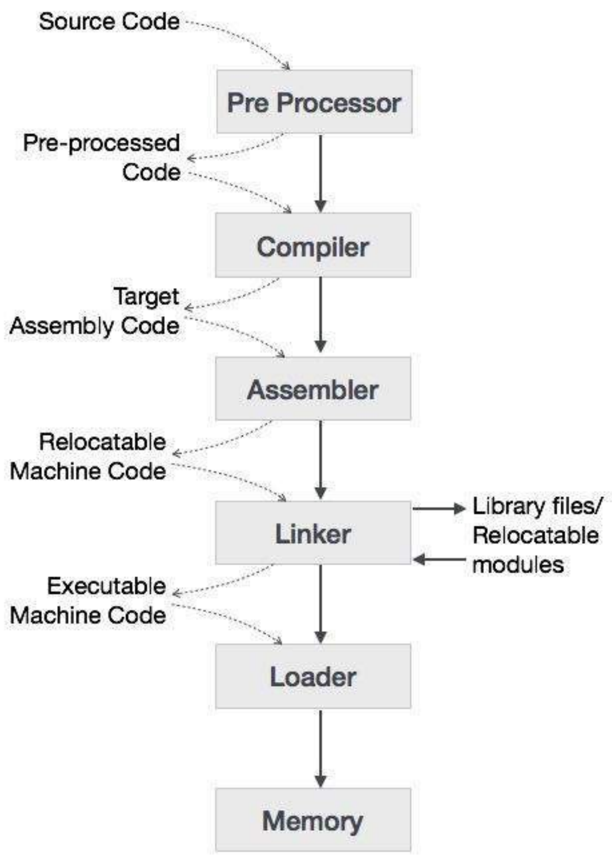
- Introduction to the ubuntu machine

- Man pages

3 facts about the kernel

1. Kernel means access to resources

OS/Kernel access



3 facts about the kernel

1. Kernel means access to resources

2. Distinction between kernel and userspace

```
int i = 0;
i = i + 1;
printf("Hello World");
```

- At some point of the program kernel became the owner of this process to print this for you.
- Kernel never relinquishes control over resources.

3 facts about the kernel

1. Kernel means access to resources

2. Distinction between kernel and userspace

3. Kernel is not a process but the process manager.
   - init
   - pstree

System Calls

- A system call is a common, controlled entry point into the kernel, allowing a process to request that the kernel perform some action on the process's behalf.

- By <u>common</u> we imply all processes use this method.
- By <u>controlled</u> we imply that sharing has to be done safely.

- Examples of system calls: creating a new process, performing I/O, and creating a pipe for interprocess communication.

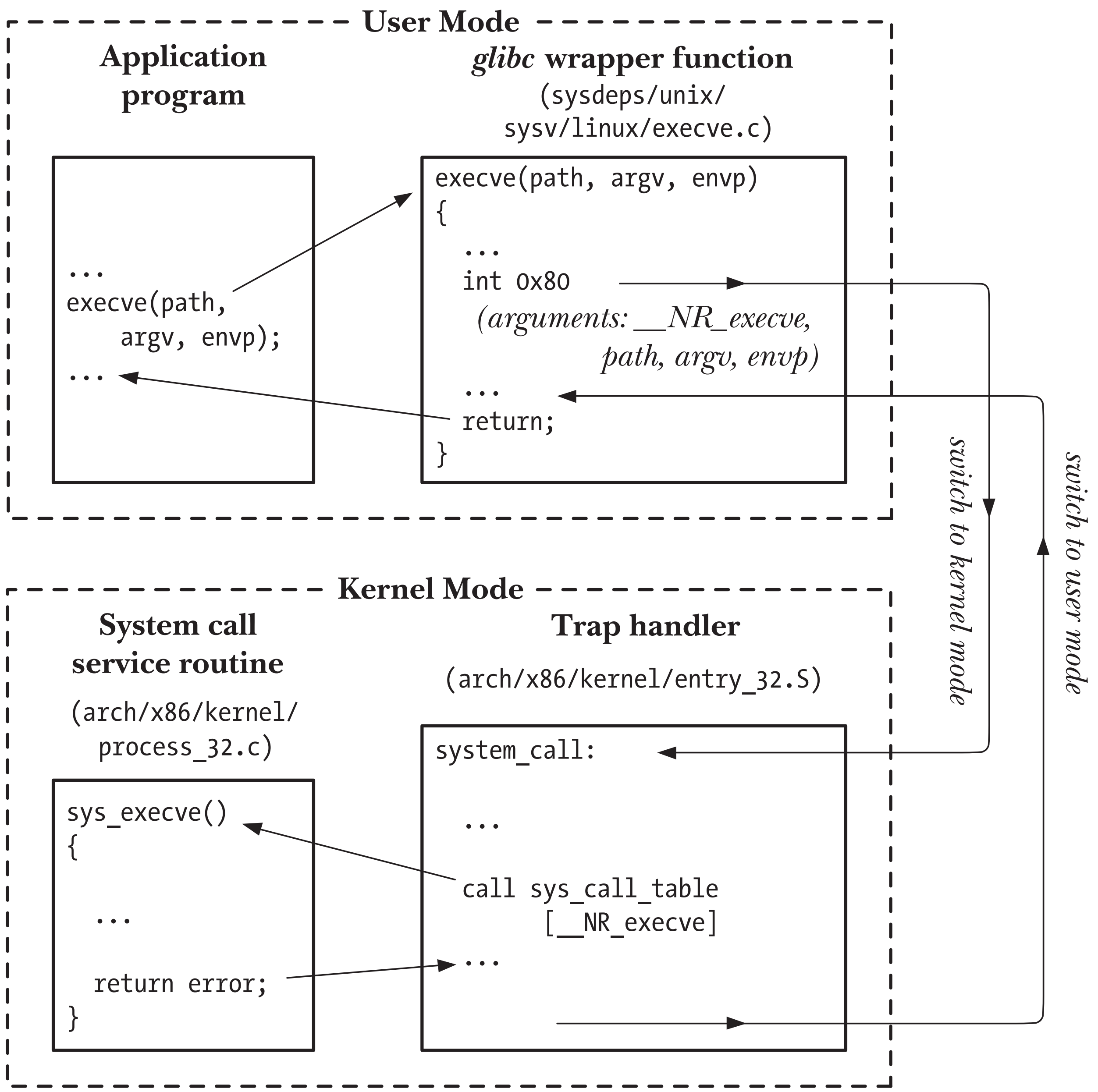
3 facts about system calls

1. A system call changes **the process state** from user mode to kernel mode, so that the CPU can access protected kernel memory.

2. System calls are akin to calling C functions except these functions perform some standard steps

   - Each system call may have a set of arguments that specify information to be transferred from user space (i.e., the process's virtual address space) to kernel space and vice versa.

3. Set of system calls is fixed. Each system call is identified by a unique number.
   - Note: there are a total of 329 calls. See here for a listing
   - http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/


User Mode
Application program
glibc wrapper function
(sysdeps/unix/sysv/linux/execve.c)
...
execve(path, argv, envp);
...
execve(path, argv, envp)
{
...
int 0x80
(arguments: __NR_execve, path, argv, envp)
...
return;
}
switch to kernel mode
switch to user mode
Kernel Mode
System call service routine
(arch/x86/kernel/process_32.c)
Trap handler
(arch/x86/kernel/entry_32.S)
sys_execve()
{
...
return error;
}
system_call:
...
call sys_call_table [__NR_execve]
...

Demo

- #include <fcntl.h>
 int main() {
 int fd = open("HelloWorld.txt", 0); }

- $ strace ./openHelloWorld```

- $ gdb openHelloWorld
- ``` b openHelloWorld.c:4```
- ``` r```
- ```s   ```
- ```disassemble```

Overview of System Calls

- I/O system calls
- Process lifecycle system calls
- Networking system calls
- IPC system calls
- Security system calls

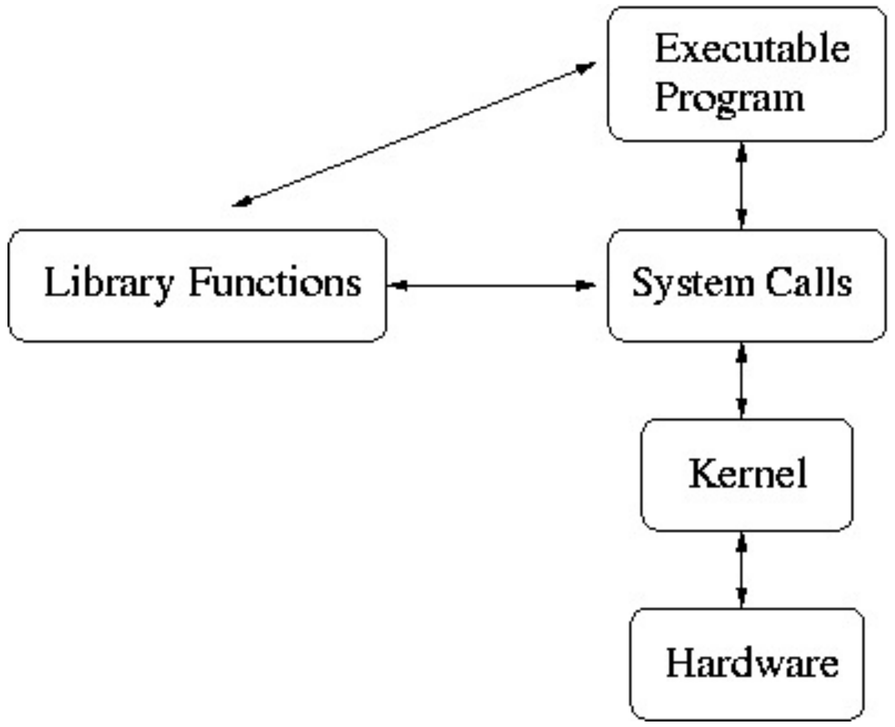
Examples of System Calls

- getuid() //get the user ID
- read() // read a buffer worth of data
- execve() //execute a program

- Don’t mix system calls with standard library functions
    - Is printf() a system call?
    - Is rand() a system call?

System calls in a program

- Some library functions have embedded system calls. For instance, the library routines scanf and printf make use of the system calls read and write.



System Calls Vs Library Routines

- System calls look very much like a library routine
- How do we know what is and what isn’t a system call?
- All you have to remember which is which
  - 2: System Call
  - 3: Library Call
- Read documentation about system calls:
  - % man read
  - http://man7.org/linux/man-pages/man2/read.2.html
- Read documentation about library functions:
  - % man fread
  - http://man7.org/linux/man-pages/man3/fread.3.html

Unix: Everything is a file

- Files are central to the UNIX philosophy.
- In the most basic form, a UNIX file is a sequence of bytes
  - *B0,B1,....,Bk ,....,Bm-1*
- All I/O devices (e.g.,network,disks, terminals) are modeled as files
  - A simple and elegant low-level interface
  - For example:
    - /dev/sda   // Hard disk
    - /dev/tty   // terminal for the current process
    - /proc/cpuinfo CPU as deduced by the kernel

File descriptiors: Kernel identification of a file

- All system calls for performing I/O refer to open files using a *file descriptor*, a (usually small) nonnegative integer.
- FDs refer to open files, where file is
  - Pipe
  - FIFOs
  - Sockets
  - Terminals
  - Regular files
  - Processe control block
- Standard file descriptors

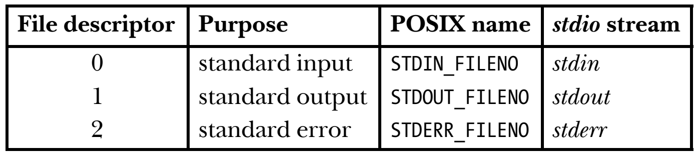


Table 4-1: Standard file descriptors

File descriptor	Purpose	POSIX name	*stdio* stream
0	standard input	`STDIN_FILENO`	*stdin*
1	standard output	`STDOUT_FILENO`	*stdout*
2	standard error	`STDERR_FILENO`	*stderr*

Unix I/O

- fd = open(file, how, ...)   // open a file for reading, writing or both
- s = close(fd)   // close an open file
- n = read(fd, buffer, nbytes) //read data from a file into a buffer
- n = write(fd, buffer, nbytes) //write data from a buffer into a file
- position = lseek(fd, offset, whence) //move the file pointer
- fd = creat(file,how)
- s = stat(name, &buf) // get a file’s status information from filesystem
- link( ), unlink( ) // aliasing and removing files

Kernel View of Files

- It seems there is a one-to-one correspondence between a file descriptor and an open file. In practice not true.
- Multiple descriptors referring to the same open file.
  - These file descriptors may be open in the same process or in different processes.
  - These may have different file status flags, and may even have different offsets within the file.

Kernel Data structure

- 3 important data structures kernel maintains for open files
  - the per-process file descriptor table;
  - the system-wide table of open file descriptions; and
  - the file system i-node table.

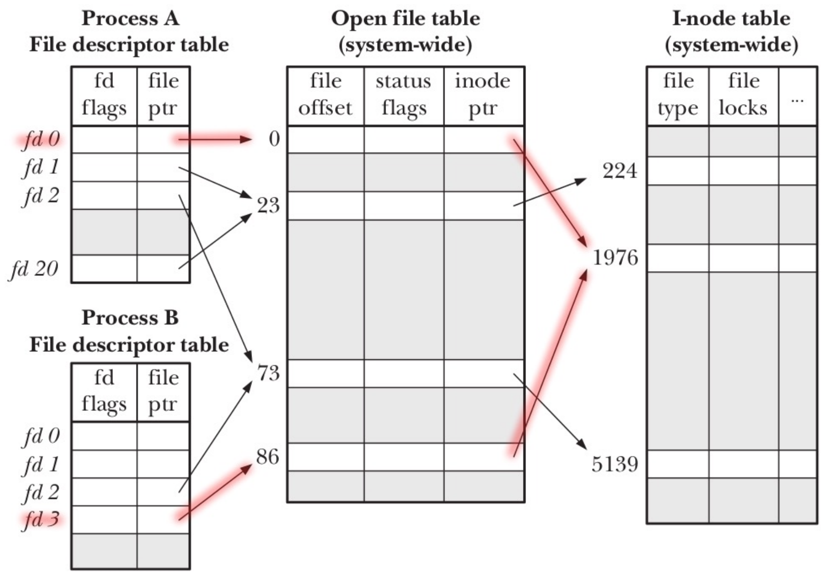


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

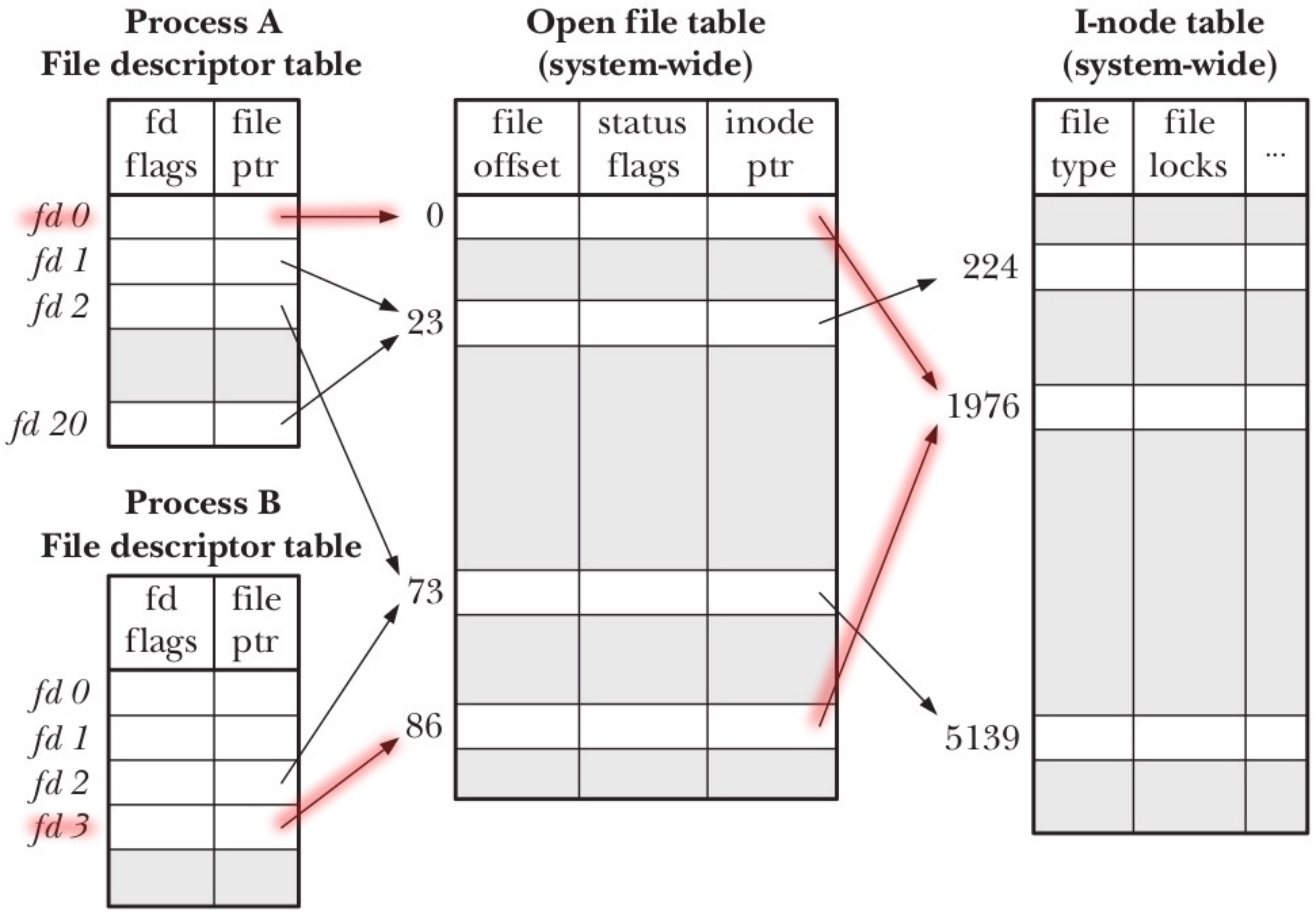


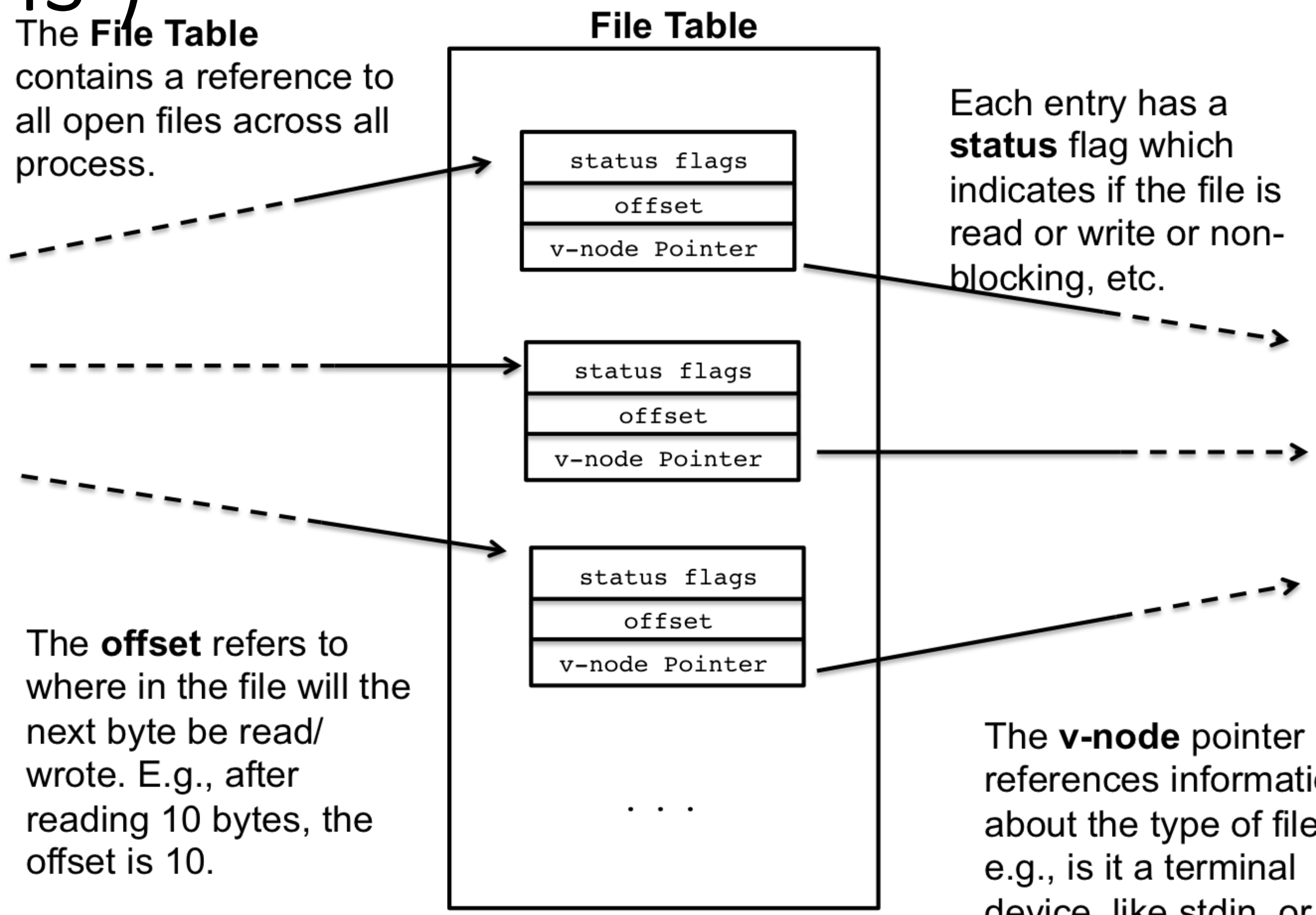
Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Open file table

- System-wide table, one entry for each open file on system:
- File offset
- File access mode (R / W / R-W, from *open()*)
- File status flags (from *open()*)
- Signal-driven I/O settings
- Reference to i-node object for file
  struct file in include/linux/fs.h
- Following terms are commonly treated as synonyms:
  - **open file description (OFD)** (POSIX)
  - **open file table entry** or **open file handle**
  - (These two are ambiguous; POSIX terminology is preferable)

Open File Table (table of open file descriptions)

The **File Table** contains a reference to all open files across all process.



Each entry has a **status** flag which indicates if the file is read or write or non-blocking, etc.

The **offset** refers to where in the file will the next byte be read/ wrote. E.g., after reading 10 bytes, the offset is 10.

The **v-node** pointer references information about the type of file, e.g., is it a terminal device, like stdin, or a file that exists on disc.

I-node Table

- System-wide table drawn from file i-node information in filesystem:
- File type (regular file, FIFO, socket, . . .)
- File permissions
- Other file properties (size, timestamps, . . .)
- *struct inode* in include/linux/fs.h

Why does it matter?

- Two different FDs referring to same OFD share file offset
  - (File offset == location for next *read()*/*write()*)
  - Changes (*read()*, *write()*, *lseek()*) via one FD visible via other FD
  - Applies to both intraprocess & interprocess sharing of OFD
- Similar scope rules for status flags (O_APPEND, O_SYNC, . . .)
  - Changes via one FD are visible via other FD
- Conversely, changes to FD flags (held in FD table) are private to each process and FD

File: dup

```
#include <unistd.h>
int dup(int oldfd)
```

- Takes *oldfd*, an open file descriptor, and returns a new descriptor that refers to the same open file description.
- The new descriptor is guaranteed to be the lowest unused file descriptor.
- `Returns`

  (new) file descriptor on success, or –1 on error
- newfd = dup(1);

Example

```
main() {
  int fd1, fd2;
  fd1 = open("file1", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  fd2 = open("file1", O_WRONLY);
}
```

Example

```
main() {
  int fd1, fd2;

  fd1 = open("file1", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  fd2 = open("file1", O_WRONLY);

  write(fd1, "The Brown Dog\n", strlen("The Brown Dog\n"));
  write(fd2, "Jumped over the moon\n", strlen("Jumped over the moon\n"));

  close(fd1);
  close(fd2);
}
```

Example

```
#include <fcntl.h>
#include <stdio.h>
main() {
  int fd1, fd2;

  fd1 = open("file2", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  fd2 = dup(fd1);

  write(fd1, "The Brown Dog\n", strlen("The Brown Dog\n"));
  write(fd2, "Jumped over the moon\n", strlen("Jumped over the moon\n"));

  close(fd1);
  close(fd2);
}
```