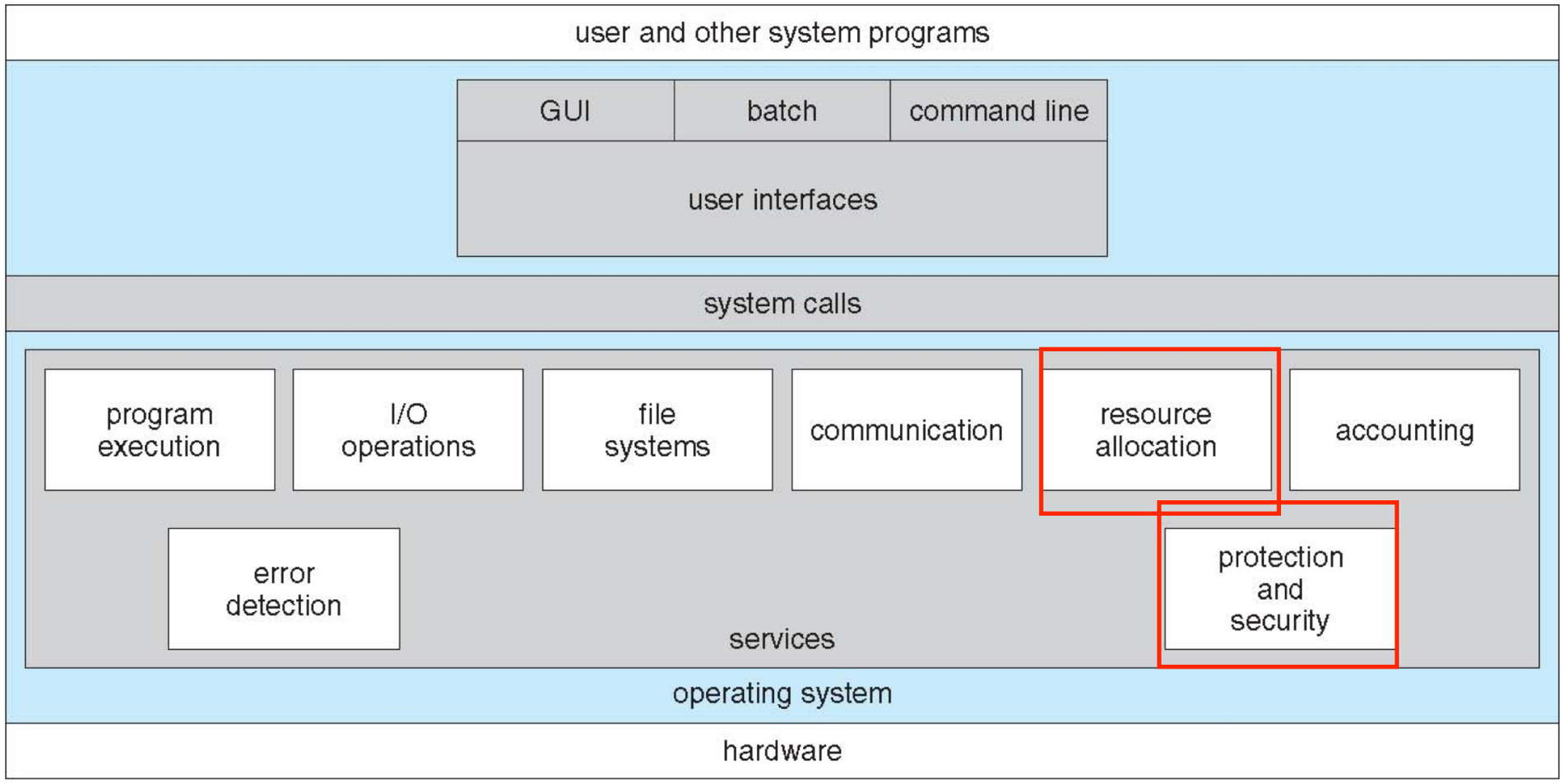
Operating System Services & Structures

Yajin Zhou (http://yajin.org)

Zhejiang University

Acknowledgement: some pages are based on the slides from Zhi Wang(fsu) and Yubing Xia(SJTU).

A View of Operating System Services



Operating System Services (User/Programmer-Visible)

- **User interface**
  - most operating systems have a user interface (UI).
  - e.g., command-Line (CLI), graphics user interface (GUI), or batch
- **Program execution: from program to process**
  - load and execute an program in the memory
  - end execution, either normally or abnormally
- **I/O operations**
  - a running program may require I/O such as file or I/O device
- **File-system manipulation**
  - read, write, create and delete files and directories
  - search or list files and directories
  - permission management

```
  13 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/1
  14 root      rt   0       0      0      0 S   0.0  0.0   0:00.13 watchdog/1
  15 root      rt   0       0      0      0 S   0.0  0.0   0:00.01 migration/1
  16 root      20   0       0      0      0 S   0.0  0.0   0:00.11 ksoftirqd/1
  18 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/1:+
  19 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kdevtmpfs
  20 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 netns
os@os:~$
os@os:~$
os@os:~$
os@os:~$
os@os:~$
os@os:~$
os@os:~$
os@os:~$
os@os:~$
os@os:~$
os@os:~$
os@os:~$
os@os:~$
os@os:~$ ls
Desktop    Downloads  Pictures  Templates  examples.desktop
Documents  Music      Public    Videos     os2018fall
os@os:~$ pwd
/home/os
os@os:~$
```

Operating System Services (User-Visible)

- **Communications**
  - processes exchange information, on the same system or over a network
  - via shared memory or through message passing

- **Error detection**
  - OS needs to be constantly aware of possible errors
  - errors in CPU, memory, I/O devices, programs
  - it should take appropriate actions to ensure correctness and consistency

Operating System Services (System View)

- **Resource allocation**
  - allocate resources for multiple users or multiple jobs running concurrently
  - many types of resources: CPU, memory, file, I/O devices
- **Accounting/Logging**
  - to keep track of which users use how much and what kinds of resources
- **Protection and security**
  - protection provides a mechanism to control access to system resources
    - access control: control access to resources
    - isolation: processes should not interfere with each other
  - security authenticates users and prevent invalid access to I/O devices
    - a chain is only as strong as its weakest link
  - protection is the **mechanism**, security towards the **policy**

```
top - 01:25:31 up 14:16,  3 users,  load average: 0.00, 0.00, 0.00
Tasks:  98 total,   1 running,  97 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.2 us,  0.0 sy,  0.0 ni, 99.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1003772 total,   217452 free,    69248 used,   717072 buff/cache
KiB Swap:  1046524 total,  1046012 free,      512 used.   749832 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
26713 root      20   0       0      0      0 S   0.3  0.0   0:00.01 kworker/u6+
    1 root      20   0  119600   5076   3336 S   0.0  0.5   0:03.39 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.01 kthreadd
    4 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:+
    6 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 mm_percpu_+
    7 root      20   0       0      0      0 S   0.0  0.0   0:00.04 ksoftirqd/0
    8 root      20   0       0      0      0 S   0.0  0.0   0:01.02 rcu_sched
    9 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
   10 root      rt   0       0      0      0 S   0.0  0.0   0:00.01 migration/0
   11 root      rt   0       0      0      0 S   0.0  0.0   0:00.13 watchdog/0
   12 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/0
   13 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/1
   14 root      rt   0       0      0      0 S   0.0  0.0   0:00.13 watchdog/1
   15 root      rt   0       0      0      0 S   0.0  0.0   0:00.01 migration/1
   16 root      20   0       0      0      0 S   0.0  0.0   0:00.11 ksoftirqd/1
   18 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/1:+
   19 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kdevtmpfs
   20 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 netns
```

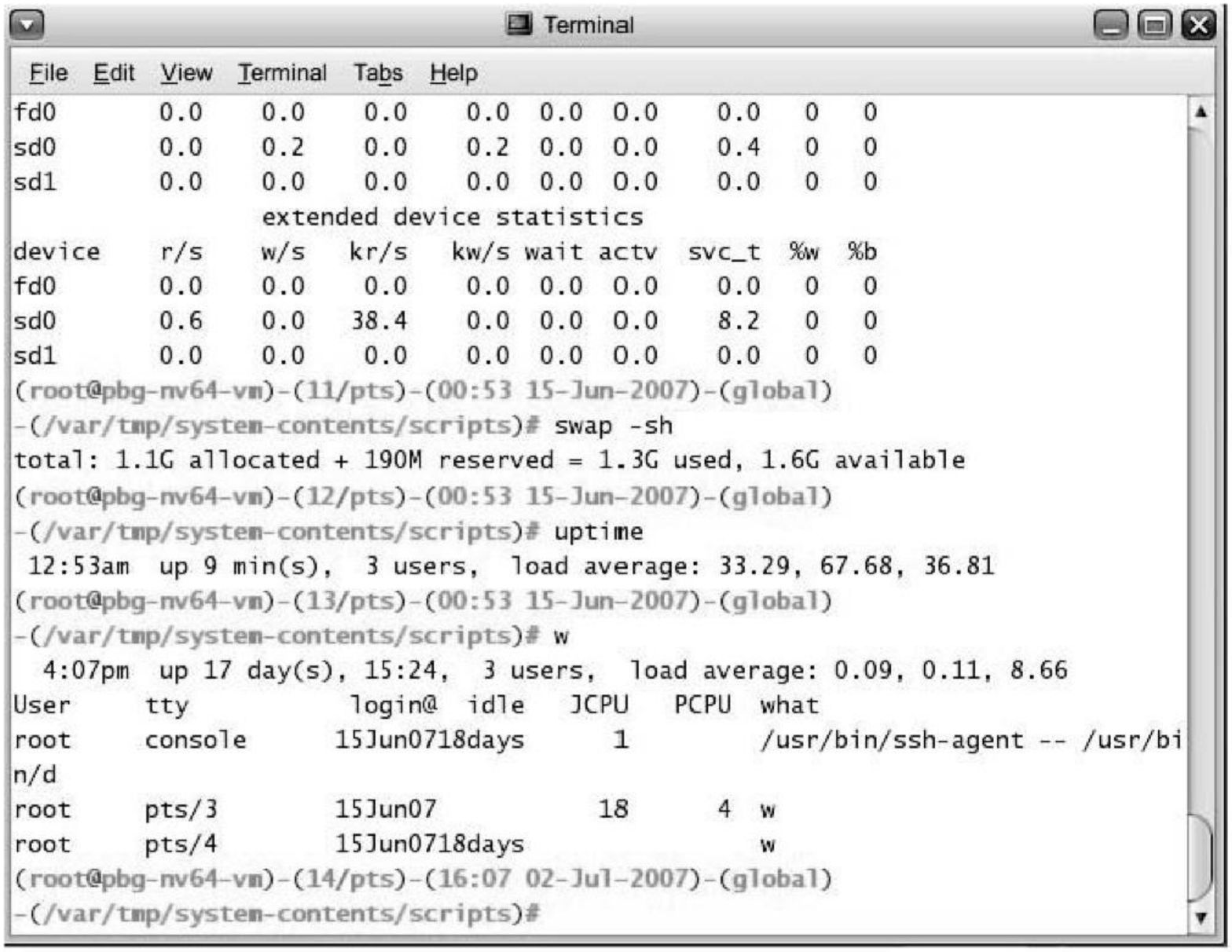
User Operating System Interface - CLI

- CLI (or command interpreter) allows direct command entry
  - a loop between fetching a command from user and executing it
- Commands are either **built-in** or **just names of programs**
  - itself contains the code to execute the command
  - implements most commands through system programs
- if the latter, adding new features doesn’t require shell modification

User Operating System Interface - GUI

- User-friendly desktop metaphor interface
  - users use mouse, keyboard, and monitor to interactive with the system
  - icons represent files, programs, actions, etc
  - mouse buttons over objects in the interface cause various actions
    - open file or directory (aka. folder), execute program, list attributes
    - invented at Xerox PARC
- Many systems include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI “command” shell
  - Apple Mac OS X as “Aqua” GUI
  - Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)
  - Linux: GNOME/KLDE GUI, and shell

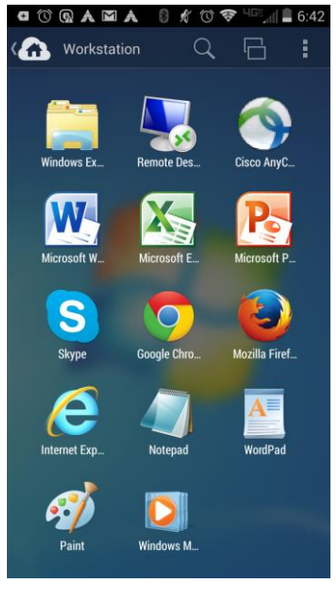
Bourne Shell Command Interpreter

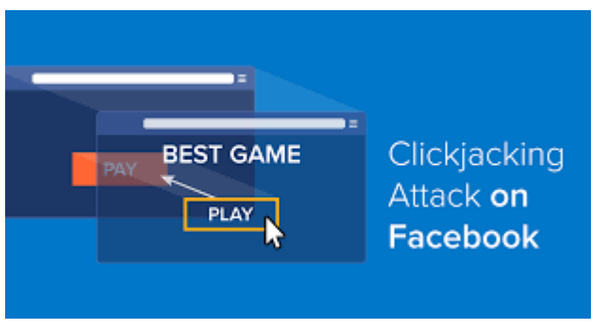


The Mac OS X GUI

Touchscreen Interfaces

- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
  - Security issues: clickjacking





Figures: https://www.brainpulse.com/articles/accessibility-clickjacking-android-device.php https://www.pcworld.com/article/2364268/parallels-access-2-0-review-remote-desktop-control-from-your-android-phone-or-tablet.html

Voice Commands

- **Voice commands**
  - **Security issues**: users' voices can be recorded, manipulated, and replayed to the assistants
  - Privacy issues

Some also called for more concrete evidence on the accusation, which has yet to be offered. Another focus of the issue is that the Alipay app is frequently requesting access to the smartphone's camera and microphone even when it's idle. Less critical comments said that even if Alipay didn't violate user-privacy by sneaking photos, it may occupy more storage and eventually ruin Android users' experiences.

Figures: https://alltechasia.com/alipay-dismisses-accusation-it-violated-user-privacy-by-snapping-photos/

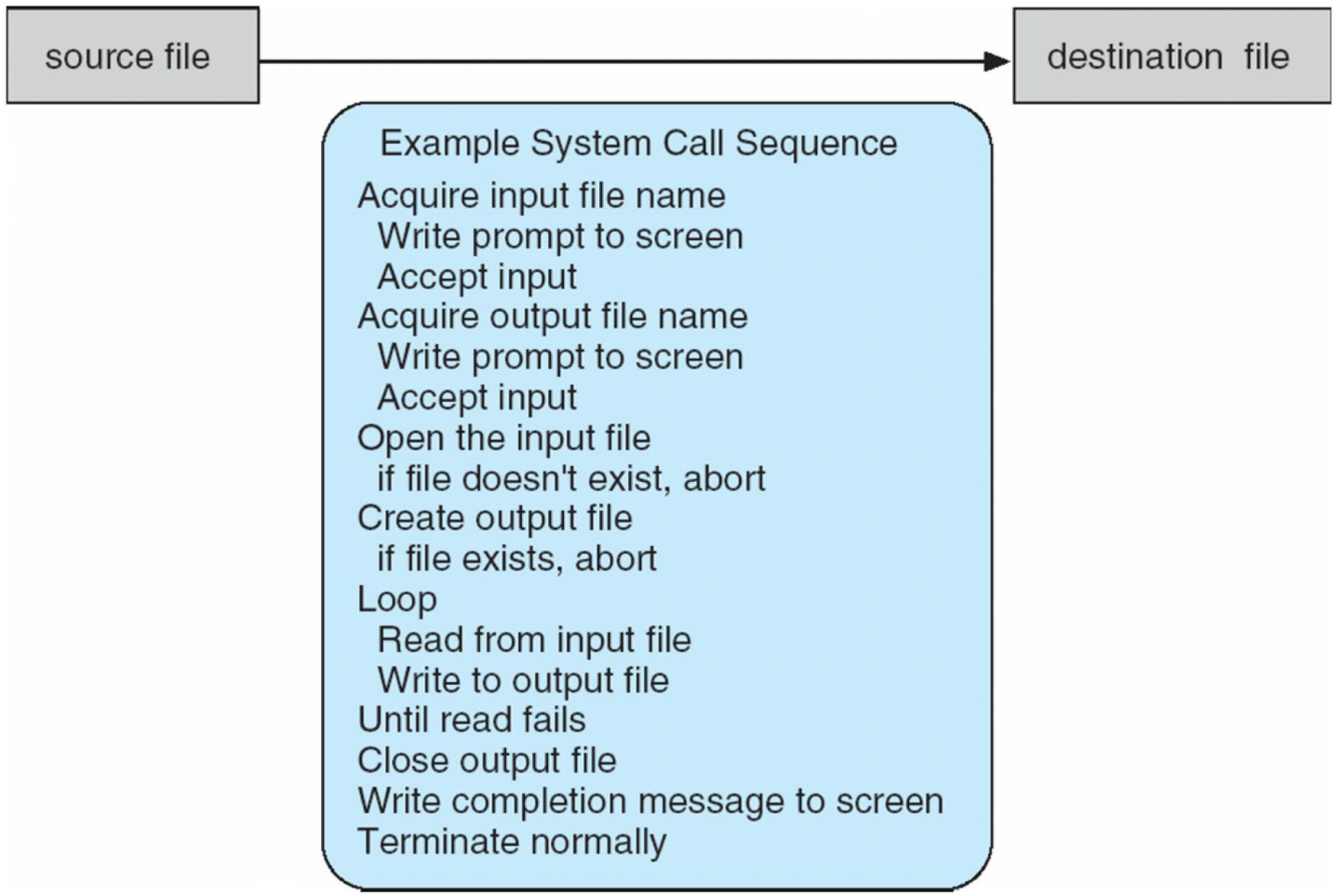
Figures: https://www.nytimes.com/2017/02/01/technology/personaltech/stop-hijacking-home-devices.html

System Calls

- System call is **a programming interface to access the OS services**
  - Typically written in a high-level language (C or C++)
  - Certain low level tasks are in assembly languages

Example of System Calls

- cp in.txt out.txt



Application Programming Interface

- Mostly accessed by programs via a high-level Application Programming Interface (API) rather than direct system call use
  - three most common APIs:
    - **Win32** API for Windows
    - **POSIX** API for POSIX-based systems (UNIX/Linux, Mac OS X)
    - **Java** API for the Java virtual machine (JVM)
  - why use APIs rather than system calls?
    - portability

Example of Standard API

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

return value — function name — parameters

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

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

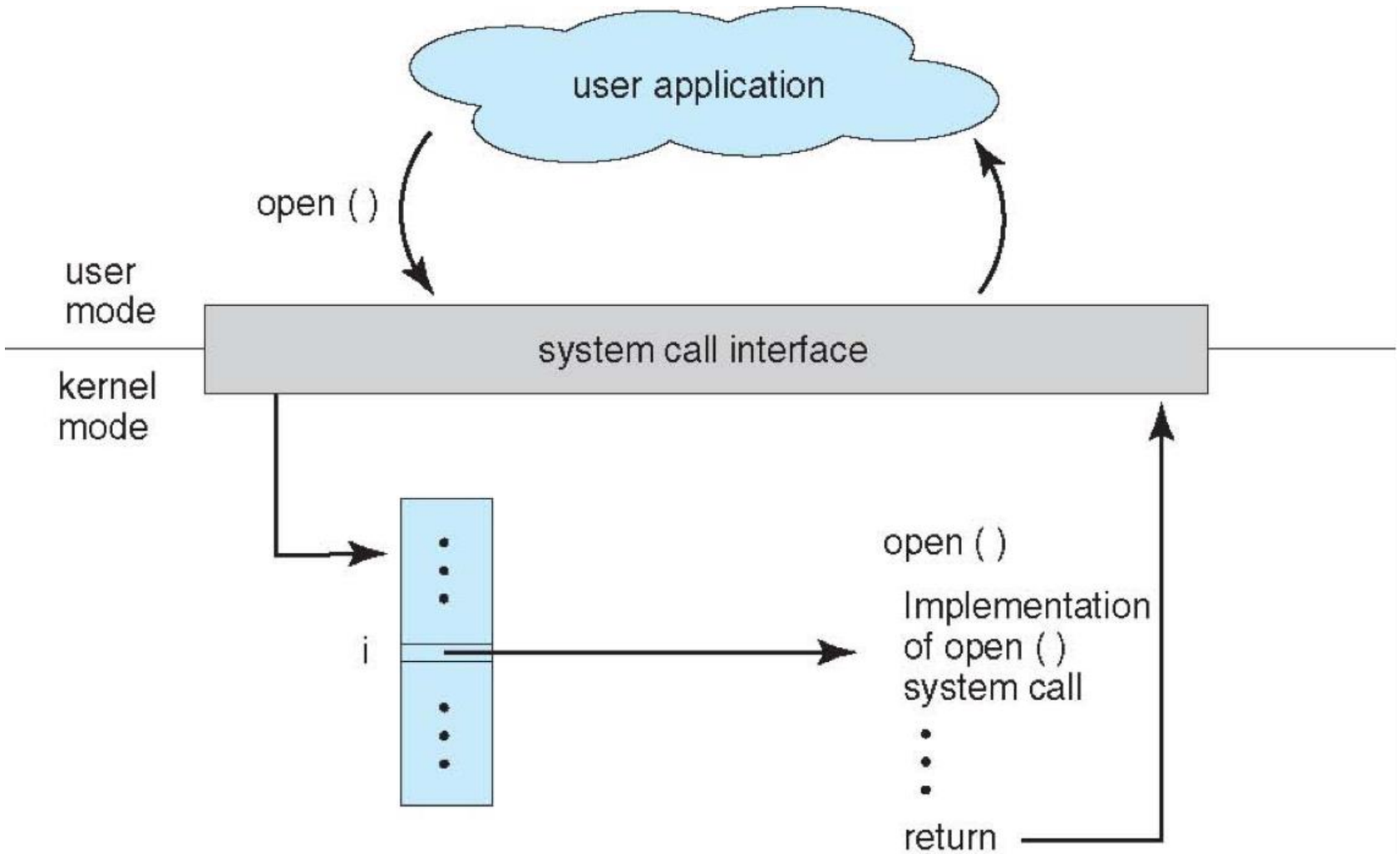
System Calls Implementation

- Typically, a **number** is associated with each system call
  - system-call interface maintains a table indexed by these numbers
  - e.g., Linux has around 340 system call (x86: 349, arm: 345)

System Calls Implementation

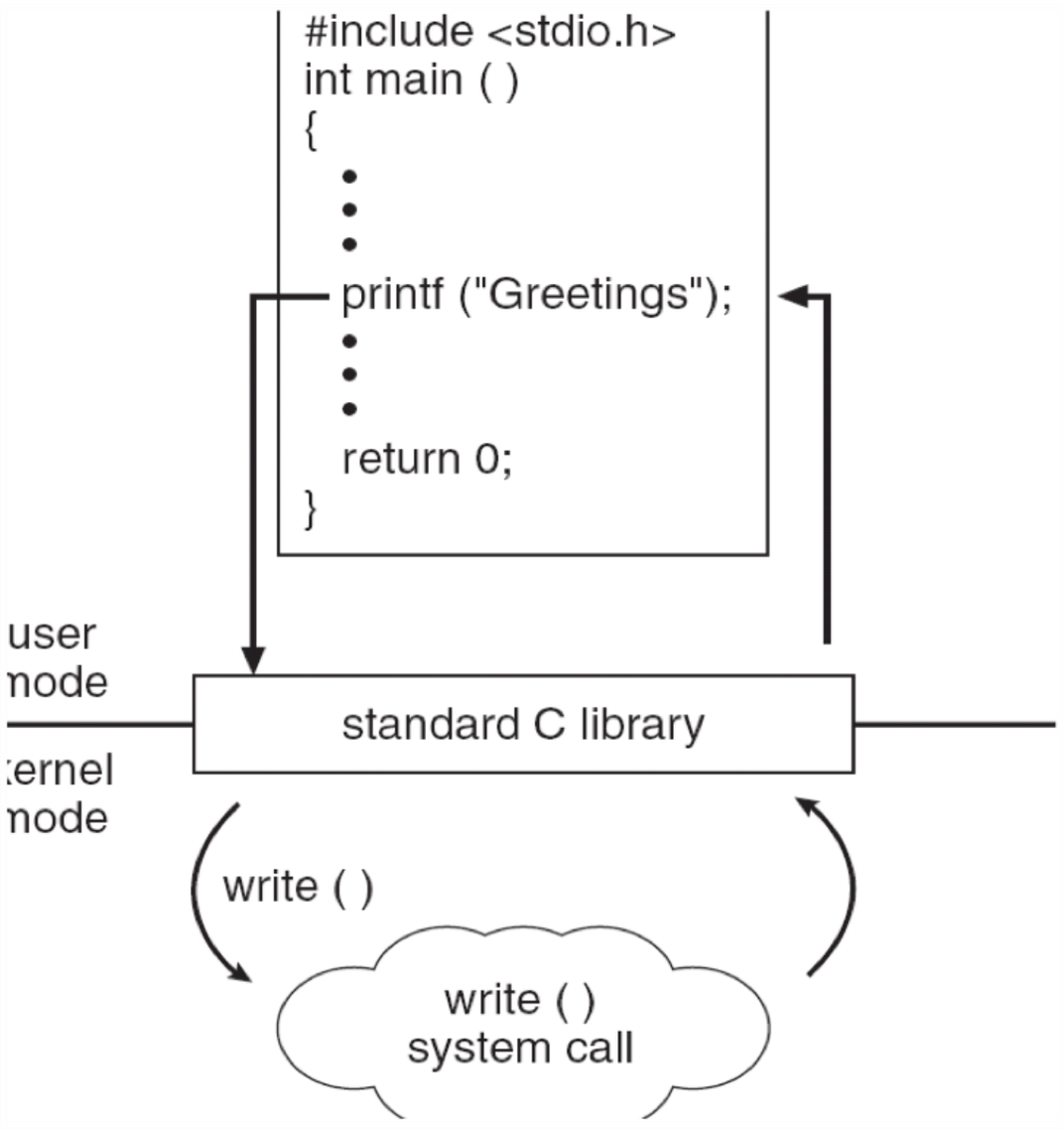
- Kernel invokes intended system call and returns results
- User program needs to know nothing about syscall details
  - it just needs to use API (e.g., in libc) and understand what the API will do
  - most details of OS interface hidden from programmers by the API

API – System Call – OS Relationship



Standard C Library Example

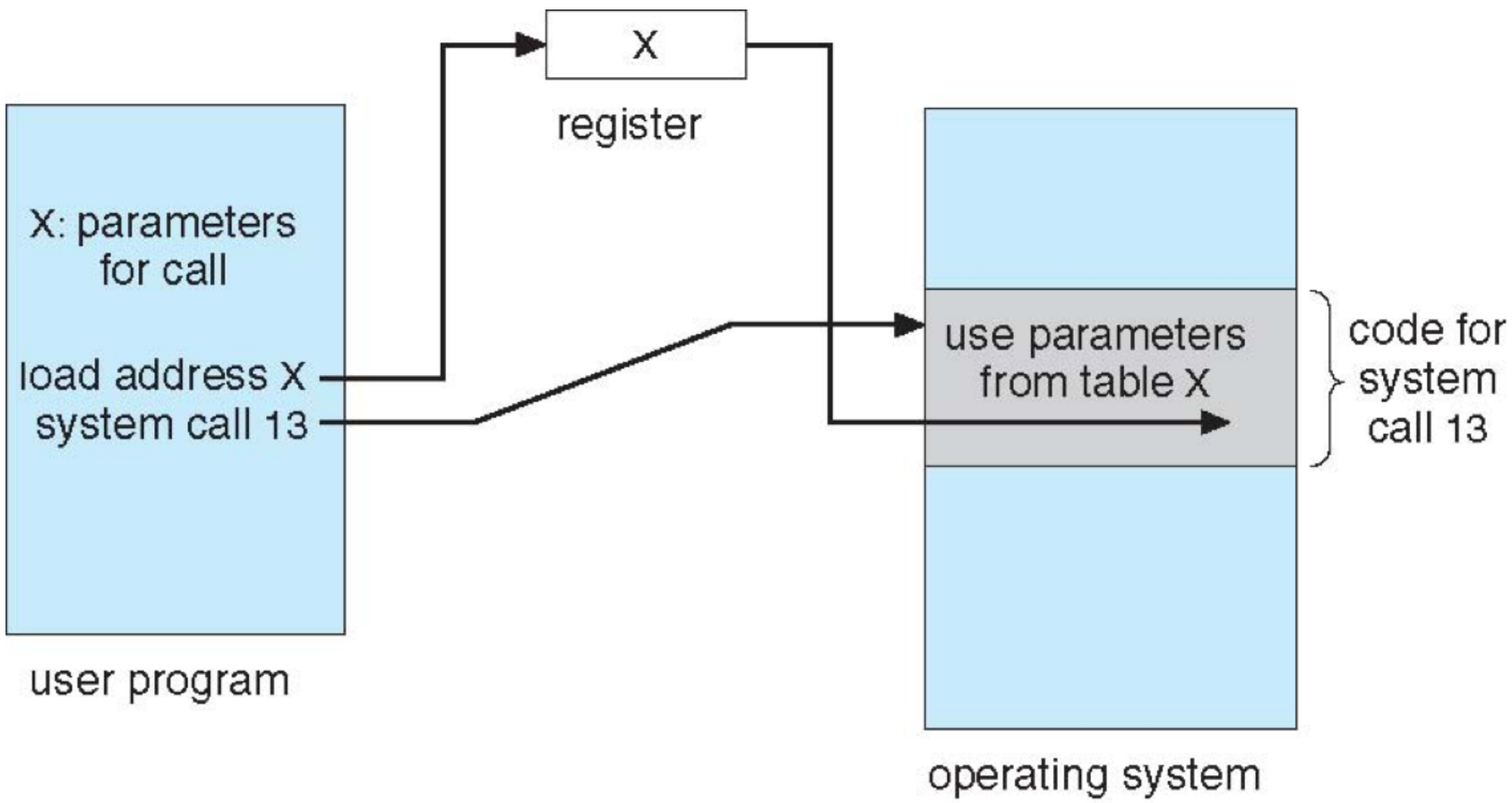
- C program invoking printf() library call, which calls write() system call



System Call Parameter Passing

- Parameters are required besides the **system call number**
  - exact type and amount of information vary according to OS and call
- Three general methods to pass parameters to the OS
  - **Register**:
    - pass the parameters in registers
    - simple, but there may be more parameters than registers
  - **Block**:
    - parameters stored in a memory block (or table)
    - address of the block passed as a parameter in a register
    - taken by Linux and Solaris
  - **Stack**:
    - parameters placed, or pushed, onto the stack by the program
    - popped off the stack by the operating system
- Block and stack methods don’t limit number of parameters being passed

Parameter Passing via Block/Table



Execve System Call on Linux/x86

- Store syscall number in eax
- Save arg 1 in ebx, arg 2 in ecx, arg 3 in edx
- Execute int 0x80 (or sysenter)
- Syscall runs and returns the result in eax

execve (“/bin/sh”, 0, 0)

eax: 0x0b
ebx: addr of “/bin/sh”
ecx: 0

Execve System Call on Linux/ARM

```
int execv(const char* name, char* const* argv) {
  return execve(name, argv, environ);
}
```

```
ENTRY(execve)
    mov     ip, r7
    ldr     r7, =__NR_execve
    swi     #0
    mov     r7, ip
    cmn     r0, #(MAX_ERRNO + 1)
    bxls    lr
    neg     r0, r0
    b       __set_errno
END(execve)
```

```
#define __NR_OABI_SYSCALL_BASE  0x900000

#if defined(__thumb__) || defined(__ARM_EABI__)
#define __NR_SYSCALL_BASE       0
#else
#define __NR_SYSCALL_BASE       __NR_OABI_SYSCALL_BASE
#endif

#define __NR_execve                     (__NR_SYSCALL_BASE+ 11)
```

Types of System Calls

- Process control
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - Debugger for determining bugs, single step execution
  - Locks for managing access to shared data between processes

Types of System Calls

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
  - can be combined with file management system call

Types of System Calls

- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages: **message passing model** to **host name** or **process name**
    - From **client** to **server**
  - **Shared-memory model** create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices

Types of System Calls

- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

Case Study: ioctl

NAME top

ioctl - control device

SYNOPSIS top

```
#include <sys/ioctl.h>

int ioctl(int fd, unsigned long request, ...);
```

DESCRIPTION top

The **ioctl**() system call manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g., terminals) may be controlled with **ioctl**() requests. The argument *fd* must be an open file descriptor.

The second argument is a device-dependent request code. The third argument is an untyped pointer to memory. It's traditionally **char** **argp* (from the days before **void** * was valid C), and will be so named for this discussion.

An **ioctl**() *request* has encoded in it whether the argument is an *in* parameter or *out* parameter, and the size of the argument *argp* in bytes. Macros and defines used in specifying an **ioctl**() *request* are located in the file *<sys/ioctl.h>*.

RETURN VALUE top

Usually, on success zero is returned. A few **ioctl**() requests use the return value as an output parameter and return a nonnegative value on success. On error, -1 is returned, and *errno* is set appropriately.

Case Study: ioctl

`ioctl` calls minimize the complexity of the kernel's system call interface. However, by providing a place for developers to "stash" bits and pieces of kernel programming interfaces, `ioctl` calls complicate the overall user-to-kernel API. A kernel that provides several hundred system calls may provide several thousand ioctl calls.

Though the interface to `ioctl` calls appears somewhat different from conventional system calls, there is in practice little difference between an `ioctl` call and a system call; an `ioctl` call is simply a system call with a different dispatching mechanism. Many of the arguments against expanding the kernel system call interface could therefore be applied to `ioctl` interfaces.

Security [edit]

The user-to-kernel interfaces of mainstream operating systems are often audited heavily for code flaws and security vulnerabilities prior to release. These audits typically focus on the well-documented system call interfaces; for instance, auditors might ensure that sensitive security calls such as changing user IDs are only available to administrative users.

`ioctl` interfaces are more complicated, more diverse, and thus harder to audit than system calls. Furthermore, because `ioctl` calls can be provided by third-party developers, often after the core operating system has been released, `ioctl` call implementations may receive less scrutiny and thus harbor more vulnerabilities. Finally, many `ioctl` calls, particularly for third-party device drivers, are undocumented.

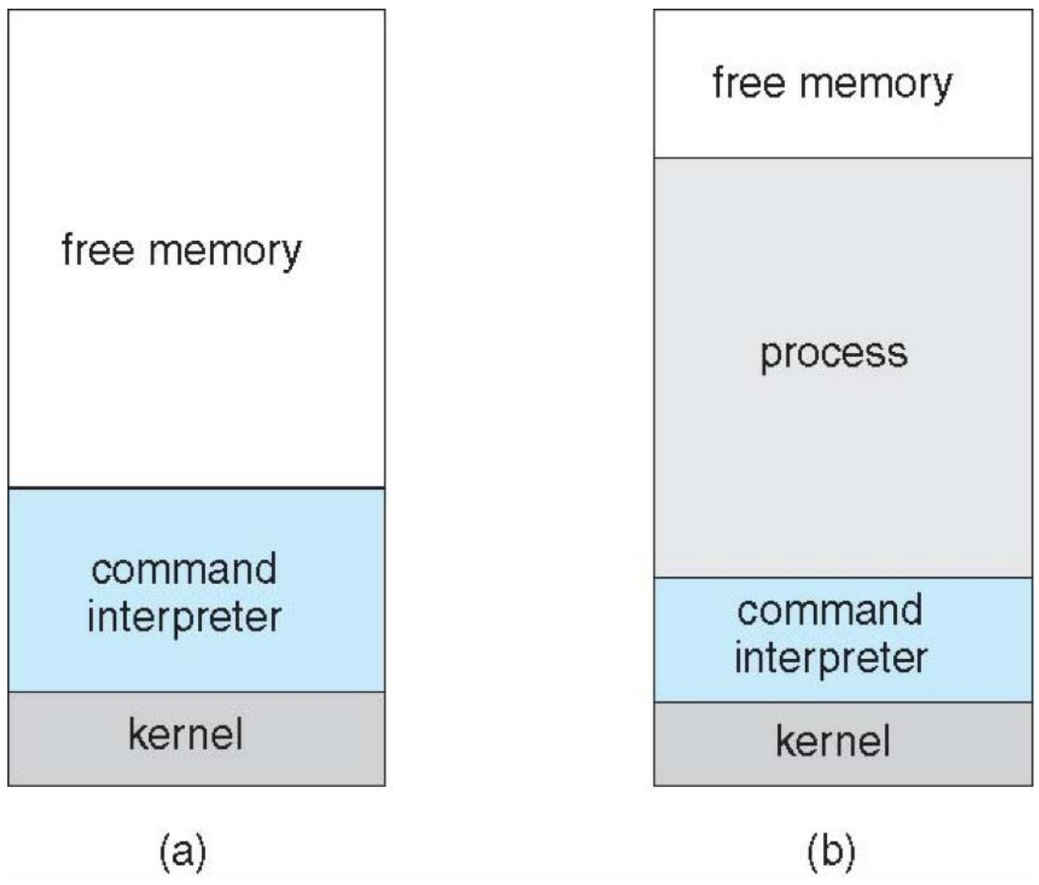
Windows and Unix System Calls

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

Example: MS-DOS

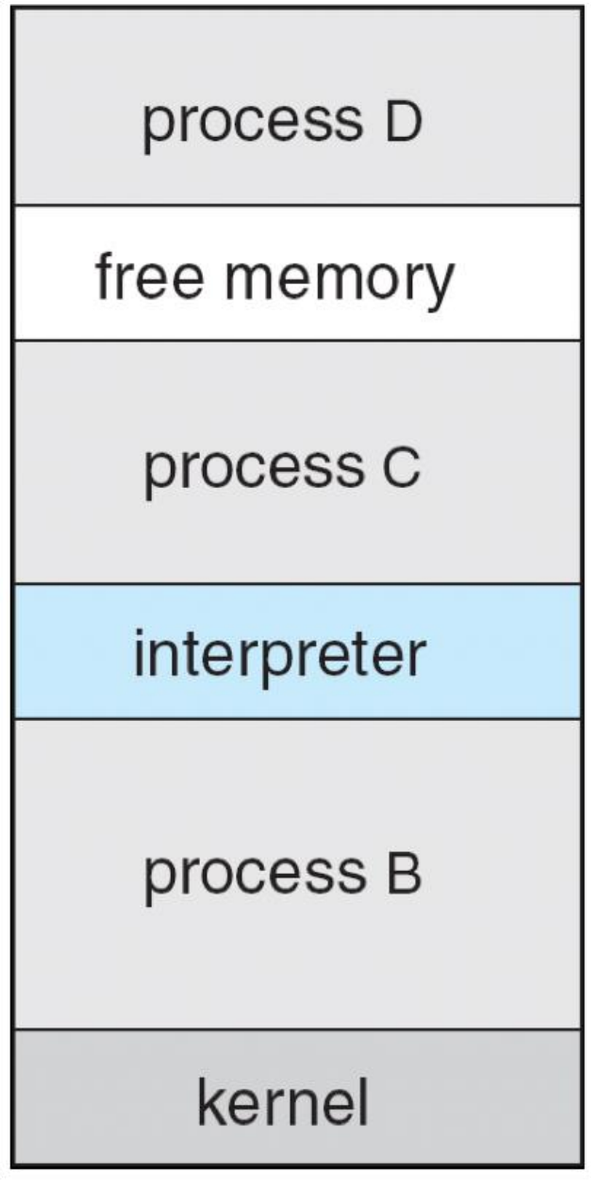
- Single-tasking
- Shell invoked when system booted
- Simple method to run program
  - no process created
  - single memory space
  - loads program into memory, overwriting all but the kernel
  - program exit -> shell reloaded

MS-DOS Execution

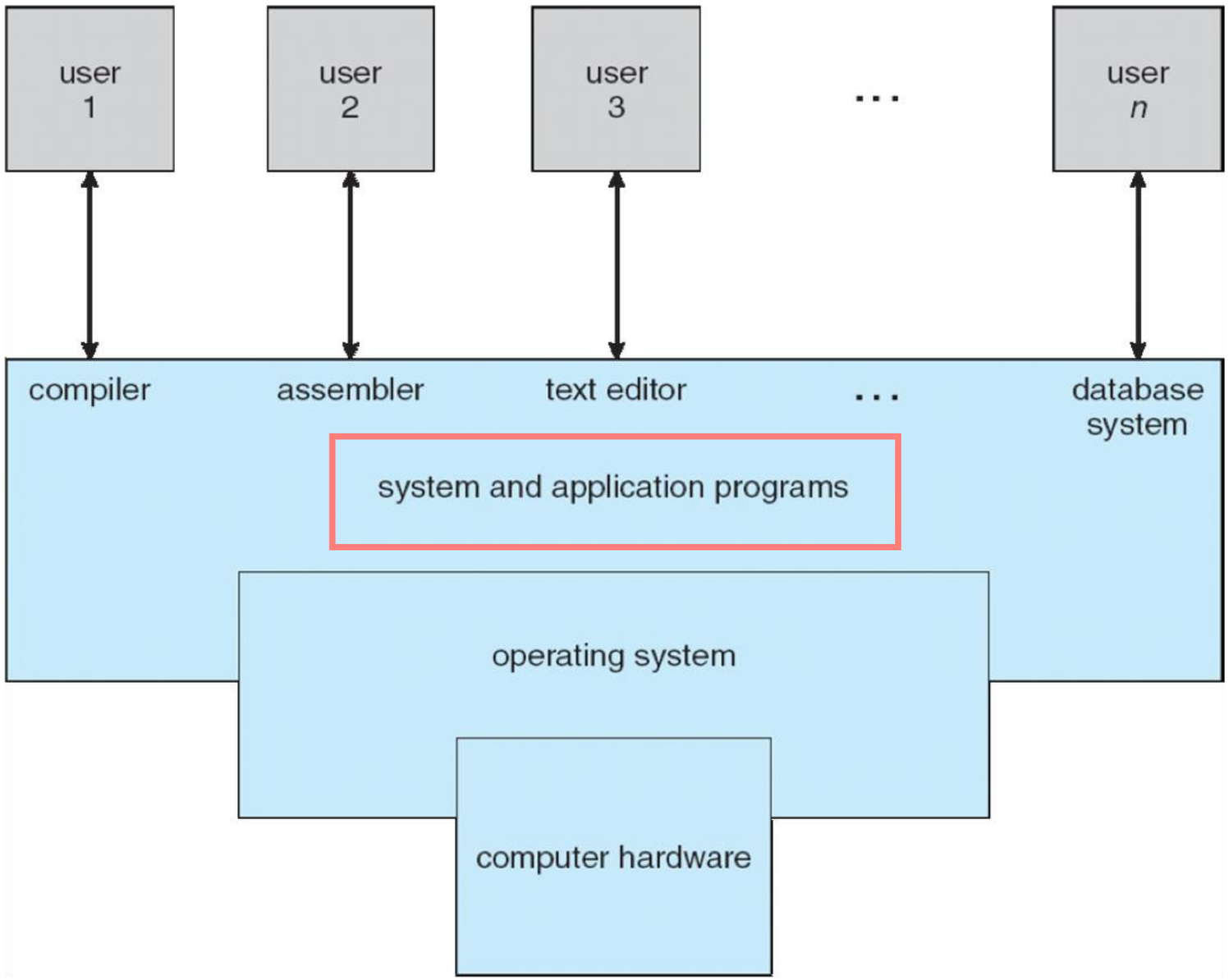


Example: FreeBSD

- A variant of Unix, it supports multitasking
- Upon user login, the OS invokes **user's choice** of shell
- Shell executes **fork**() system call to create process, then calls **exec**() to load program into process
  - shell waits for process to terminate or continues with user commands
- Process exits with:
  - code = 0 – no error
  - code > 0 – error code



System Services (Programs)



System Services

- System programs provide a convenient environment for program development and execution. They can be divided into:
  - File manipulation
    - create/delete/copy files/directories …
  - Status information sometimes stored in a file modification

```
os@os:~$ df -lh
Filesystem      Size  Used Avail Use% Mounted on
udev            465M     0  465M   0% /dev
tmpfs            99M  8.4M   90M   9% /run
/dev/sda1        62G  4.2G   55G   8% /
tmpfs           491M     0  491M   0% /dev/shm
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           491M     0  491M   0% /sys/fs/cgroup
tmpfs            99M     0   99M   0% /run/user/1000
os@os:~$ free -lh
              total        used        free      shared  buff/cache   available
Mem:           980M         63M        220M        8.0M        695M        737M
Low:           980M        759M        220M
High:            0B          0B          0B
Swap:          1.0G        512K        1.0G
```

System Services

- Programming language support
  - c/python/Java …
- Program loading and execution
- Communications
  - between processes, hosts and etc.
- Background services
  - services, **daemon**, sub-system
- Application programs

```
os@os:~$ python
Python 2.7.12 (default, Dec  4 2017, 14:50:18)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello World!"
Hello World!
>>>
```

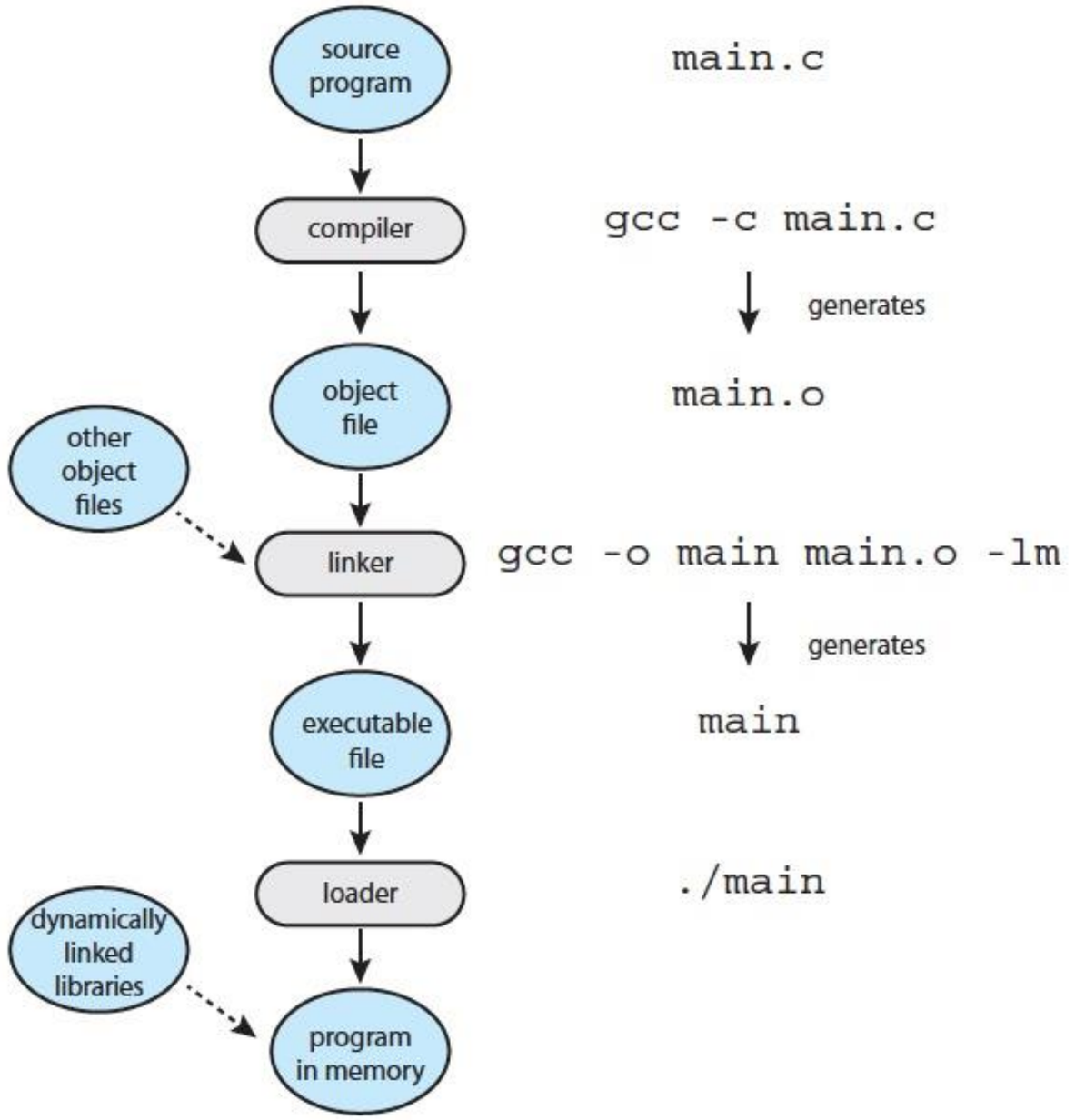
```
os@os:~$ cat /etc/crontab
# /etc/crontab: system-wide crontab
# Unlike any other crontab you don't have to run the `crontab'
# command to install the new version when you edit this file
# and files in /etc/cron.d. These files also have username fields,
# that none of the other crontabs do.

SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

# m h dom mon dow user  command
17 *    * * *   root    cd / && run-parts --report /etc/cron.hourly
25 6    * * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.daily )
47 6    * * 7   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.weekly )
52 6    1 * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.monthly )
#
```

Review

- Operating system services
  - User interface, program execution, I/O, file system manipulation…
  - Resource allocation, Logging/accounting, Protection & Security
- System call
  - User program - API - system call - OS
  - System call implementation: parameter passing
  - Types of System call

Linkers & Loaders

- linker
  - from object files to executable file
- loader
  - from program to process
  - static linking vs dynamic linking
  - lazy binding

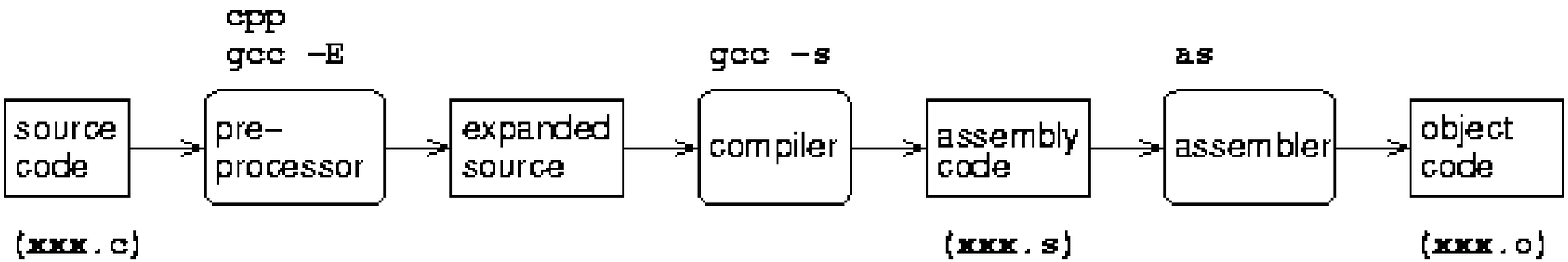


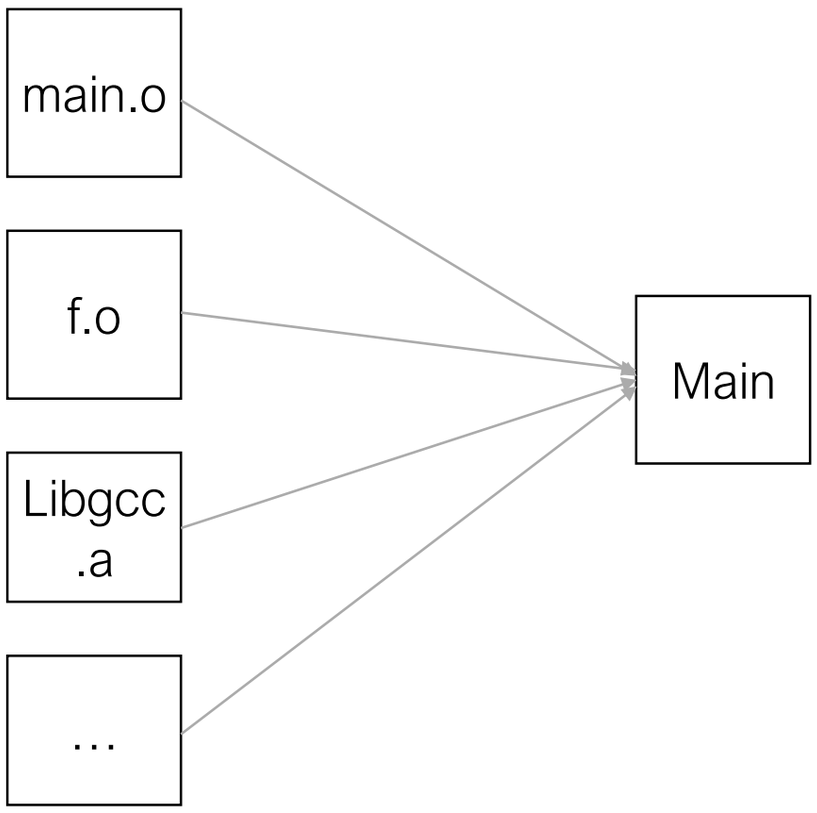
Linkers & Loaders

```c
#include <stdio.h>

extern int f  (int x);
int i = 2;
char format[] = "f (%d) = %d\n";

int main (int argc, char const *argv[])
{
  int j;
  j = f (i);
  printf (format, i, j);
  return 0;
}
```

```c
int f (int x)
{
  if (x <= 1) return x;

  return x - 1;
}
```

Linkers & Loaders





Static Linking

```
00000000 <main>:
  0:  e92d4010        push {r4, lr}
  4:  e59f4024  ldr   r4, [pc, #36]     ; 30 <main+0x30>
  8:  e79f4004  ldr   r4, [pc, r4]
  c:  e5940000        ldr    r0, [r4]
 10:  ebffffe    bl    0 <f>
 14:  e59f3018  ldr   r3, [pc, #24]     ; 34 <main+0x34>
 18:  e1a02000        mov  r2, r0
 1c:  e5941000        ldr    r1, [r4]
 20:  e79f0003  ldr   r0, [pc, r3]
 24:  ebffffe    bl    0 <printf>
 28:  e3a00000        mov  r0, #0
 2c:  e8bd8010        pop   {r4, pc}
 30:  00000020        .word       0x00000020
 34:  0000000c        .word       0x0000000c
```

main.o

```
00000000 <f>:
  0:  e3500001        cmp  r0, #1
  4:  c2400001        subgt       r0, r0, #1
  8:  e12fff1e    bx    lr
```

f.o

```
-> ls -lh libs/armeabi/main
-rwxr-xr-x  1 yajin  staff   146K Sep 27 18:53 libs/armeabi/main
```

Parameters are passed using register r0- r3, return value is in register r0.

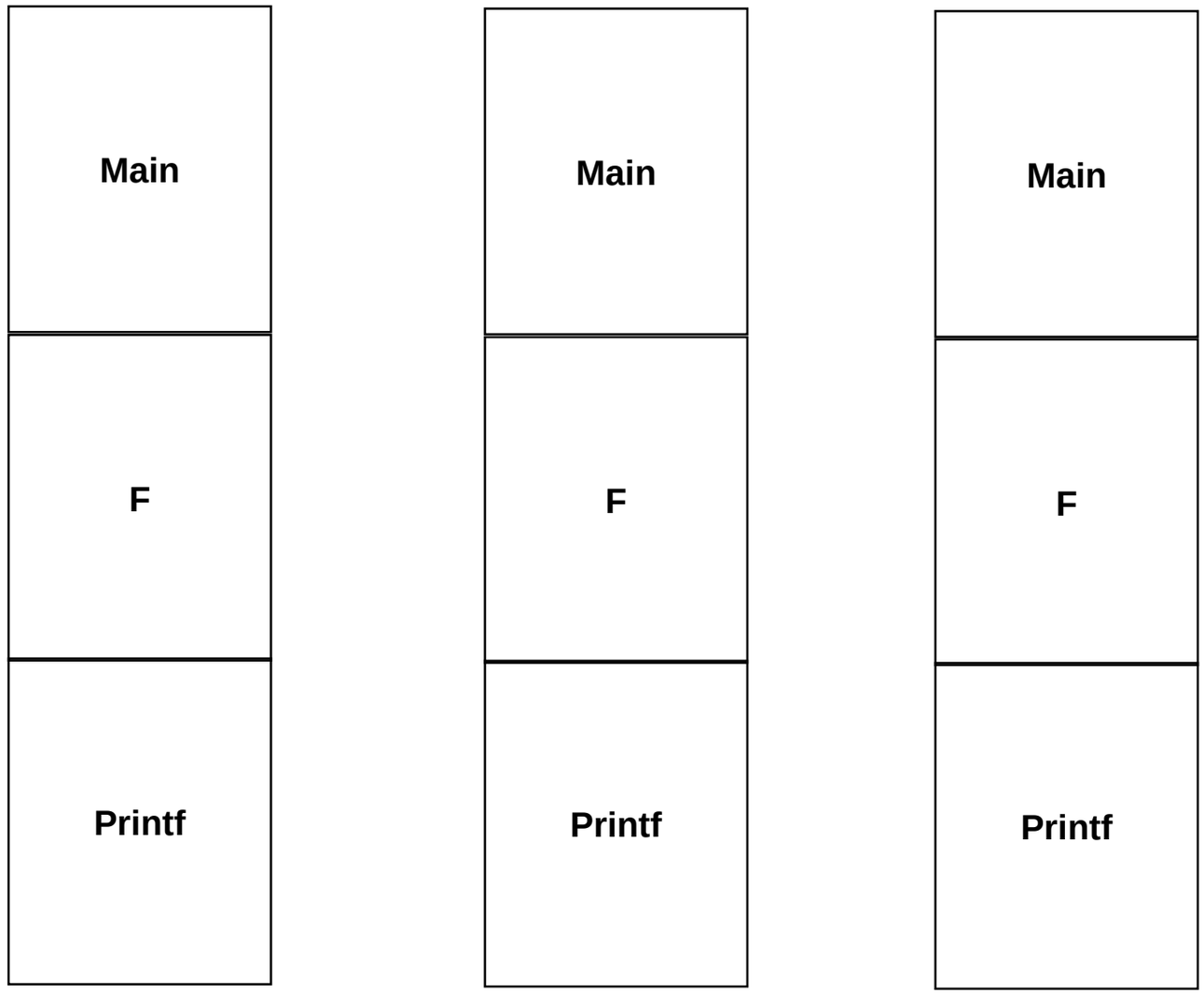
Static Linking

```
0000885c <main>:
   885c:    e92d4010    push    {r4, lr}
   8860:    e59f4024    ldr     r4, [pc, #36]   ; 888c <main+0x30>
   8864:    e79f4004    ldr     r4, [pc, r4]
   8868:    e5940000    ldr     r0, [r4]
   886c:    eb000067    bl      8a10 <f>
   8870:    e59f3018    ldr     r3, [pc, #24]   ; 8890 <main+0x34>
   8874:    e1a02000    mov     r2, r0
   8878:    e5941000    ldr     r1, [r4]
   887c:    e79f0003    ldr     r0, [pc, r3]
   8880:    fa000a3e    blx     b180 <printf>
   8884:    e3a00000    mov     r0, #0
   8888:    e8bd8010    pop     {r4, pc}
   888c:    0002366c    .word   0x0002366c
   8890:    00023658    .word   0x00023658
00008a10 <f>:
   8a10:    e3500001    cmp     r0, #1
   8a14:    c2400001    subgt   r0, r0, #1
   8a18:    e12fff1e    bx      lr
```

```
0000b180 <printf>:
   b180:    b40f        push    {r0, r1, r2, r3}
   b182:    b507        push    {r0, r1, r2, lr}
   b184:    aa04        add     r2, sp, #16
   b186:    4b08        ldr     r3, [pc, #32]   ; (b1a8 <printf+0x28>)
   b188:    f852 1b04   ldr.w   r1, [r2], #4
   b18c:    4807        ldr     r0, [pc, #28]   ; (b1ac <printf+0x2c>)
   b18e:    447b        add     r3, pc
   b190:    9201        str     r2, [sp, #4]
   b192:    581b        ldr     r3, [r3, r0]
   b194:    f103 0054   add.w   r0, r3, #84     ; 0x54
   b198:    f001 fb1a   bl      c7d0 <vfprintf>
   b19c:    b003        add     sp, #12
   b19e:    f85d eb04   ldr.w   lr, [sp], #4
   b1a2:    b004        add     sp, #16
   b1a4:    4770        bx      lr
   b1a6:    bf00        nop
   b1a8:    00020e62    .word   0x00020e62
   b1ac:    ffffff0c    .word   0xffffff0c
```

PC relative addressing so that the code can be loaded into arbitrary addresses in the memory.

Static Linking: in memory



Dynamic Linking

```
00000000 <main>:
   0:  e92d4038        push {r3, r4, r5, lr}
   4:  e59f5024  ldr   r5, [pc, #36]     ; 30 <main+0x30>
   8:  e08f5005  add   r5, pc, r5
   c:  e1a04005        mov  r4, r5
  10:  e4940008        ldr    r0, [r4], #8
  14:  ebfffffe   bl    0 <f>
  18:  e5951000        ldr    r1, [r5]
  1c:  e1a02000        mov  r2, r0
  20:  e1a00004        mov  r0, r4
  24:  ebfffffe   bl    0 <printf>
  28:  e3a00000        mov  r0, #0
  2c:  e8bd8038        pop  {r3, r4, r5, pc}
  30:  00000020        .word       0x00000020
```

main.o

```
00000000 <f>:
   0:  e3500001        cmp  r0, #1
   4:  c2400001        subgt       r0, r0, #1
   8:  e12fff1e   bx    lr
```

f.o

```
-> ls -lh libs/armeabi/main
-rwxr-xr-x  1 yajin  staff   9.4K Sep 27 19:09 libs/armeabi/main
```

Dynamic Linking

```
000004d0 <main>:
    4d0:        e92d4038    push    {r3, r4, r5, lr}
    4d4:        e59f5024    ldr     r5, [pc, #36]   ; 500
<main+0x30>
    4d8:        e08f5005    add     r5, pc, r5
    4dc:        e1a04005    mov     r4, r5
    4e0:        e4940008    ldr     r0, [r4], #8
    4e4:        eb000030    bl      5ac <f>
    4e8:        e5951000    ldr     r1, [r5]
    4ec:        e1a02000    mov     r2, r0
    4f0: e1a00004   mov     r0, r4
    4f4:        ebfffe3     bl      488 <printf@plt>
    4f8: e3a00000   mov     r0, #0
    4fc: e8bd8038   pop     {r3, r4, r5, pc}
    500:        00002b20    .word   0x00002b20

00000488 <printf@plt>:
 488:   e28fc600    add     ip, pc, #0
 48c:   e28cca02    add     ip, ip, #8192   ; 0x2000
 490:   e5bcfb58    ldr     pc, [ip, #2904]!        ; 0xb58
```

Location: (0x488 + 0x8) (PC) + 0x2000 + 0xb58 = **0x2fe8**

```
-> arm-linux-androideabi-readelf  -S main
There are 36 section headers, starting at offset 0x9a08:

  [19] .got              PROGBITS        00002fa4 001fa4 00005c 00  WA  0   0  4
```

```
-> arm-linux-androideabi-objdump -R main

00002fe8 R_ARM_JUMP_SLOT   printf
```

```
-> arm-linux-androideabi-readelf -l main
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align

  GNU_RELRO      0x001e6c 0x00002e6c 0x00002e6c 0x00194 0x00194 RW  0x4
```

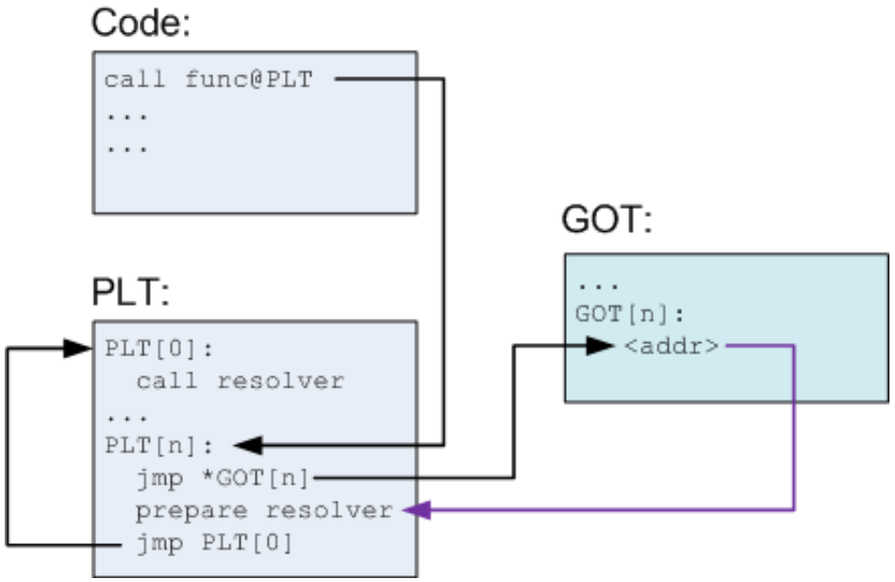
The real jump address is in the GOT section, which is readable and writable. The question is who is going to set the address (**0x2fe8)** to the real address of the function printf in libc? And who is responsible to load the correspond libraries into the memory?

Loader

- When loading a binary
  - Load the PT_LOADED segments into memory
  - Resolve the library dependencies and load the corresponding libraries into memory
    - Set the value in the GOT entry to the actual address of the function in the libraries (not necessary) - since these processes are performed when the binary is loading, it may slow the binary loading process
- Static linking vs dynamic linking

Linkers & Loaders: x86/Lazy Binding

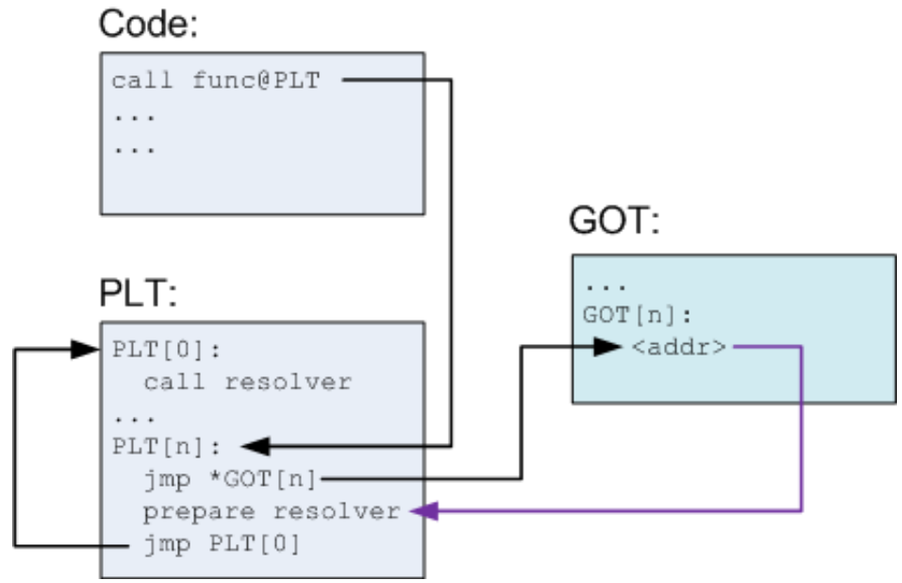
- In the code, a function func is called. The compiler translates it to a call to func@plt, which is some N-th entry in the PLT.
- The PLT consists of a special first entry, followed by a bunch of identically structured entries, one for each function needing resolution.
- Each PLT entry but the first **consists of these parts**:
  - A jump to a location which is specified in a corresponding GOT entry
  - Preparation of arguments for a "resolver" routine
  - Call to the resolver routine, which resides in the first entry of the PLT
- The first PLT entry is a call to a **resolver routine**, which is located in the dynamic loader itself. This routine resolves the actual address of the function.
- Before the function's actual address has been resolved, the Nth GOT entry just points to after the jump. This is why this arrow in the diagram is colored differently - it's not an actual jump, just a pointer.



Figures: https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/

Linkers & Loaders: Example

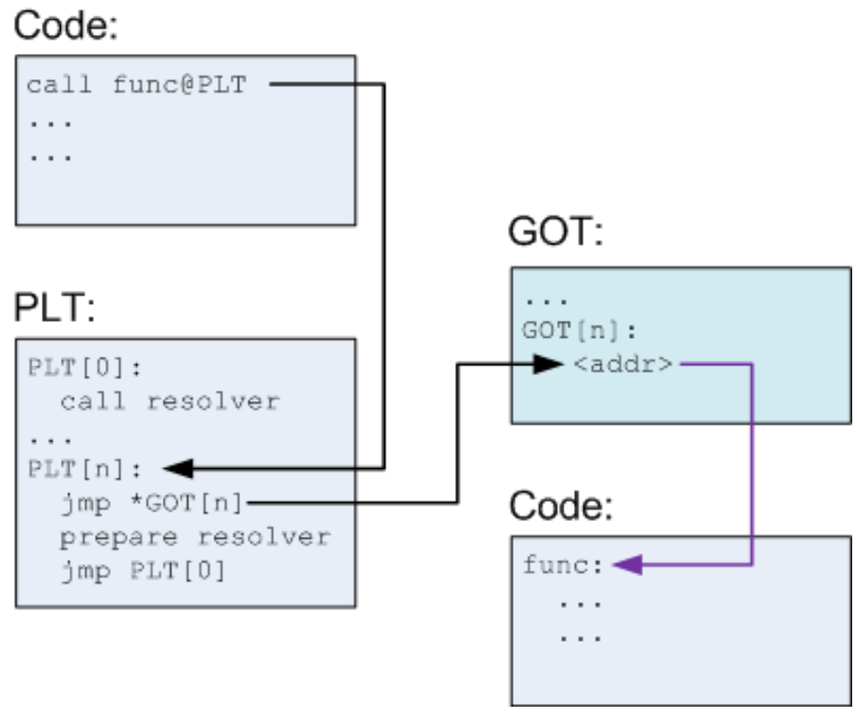
- PLT[n] is called and jumps to the address pointed to in GOT[n].
- This address points into PLT[n] itself, to the preparation of arguments for the resolver.
- The resolver is then called.
- The resolver performs resolution of the actual address of func, places its actual address into GOT[n] and calls func.



Figures: https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/

Linkers & Loaders: Example

- Note that GOT[n] now points to the actual func [7] instead of back into the PLT. So, when func is called again:
  - PLT[n] is called and jumps to the address pointed to in GOT[n].
  - GOT[n] points to func, so this just transfers control to func.

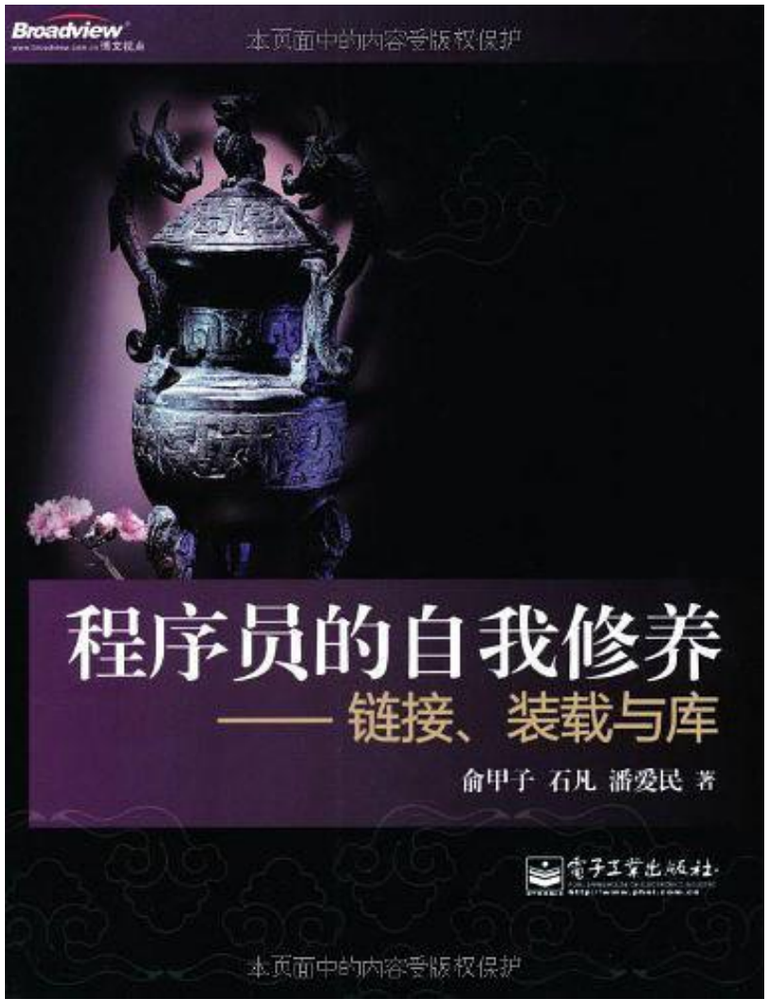


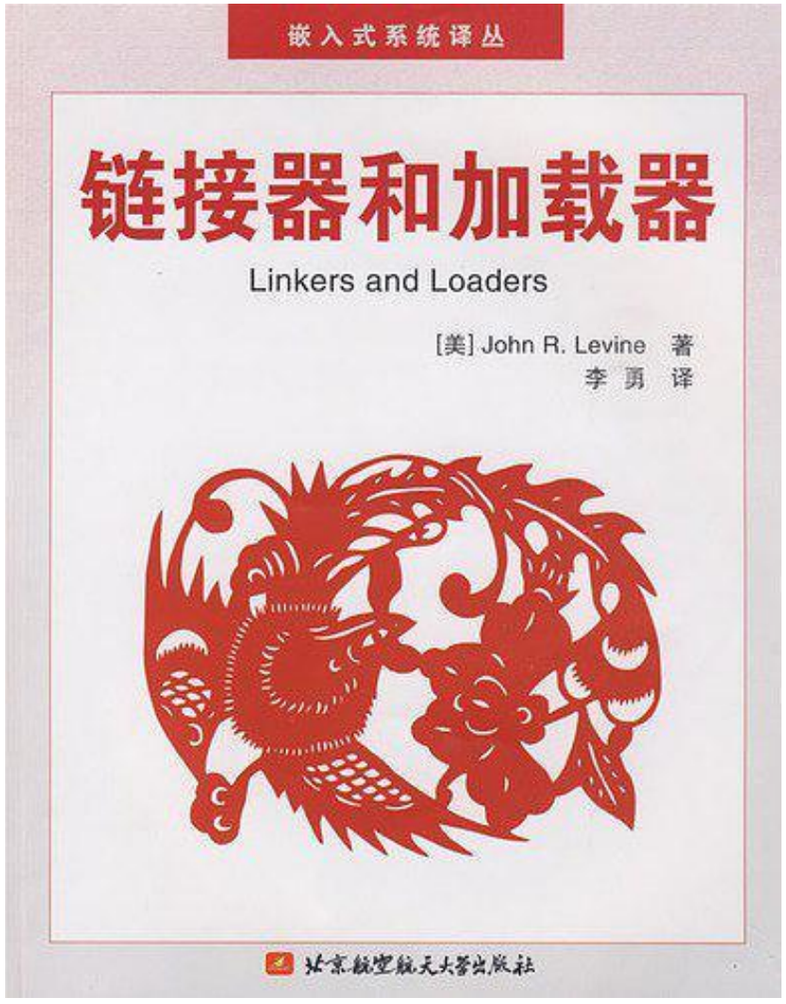
Q: The resolver is a program, then who is responsible for resolving the symbols in resolver (or is this needed for resolver)?

Further reading: https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/

Figures: https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/

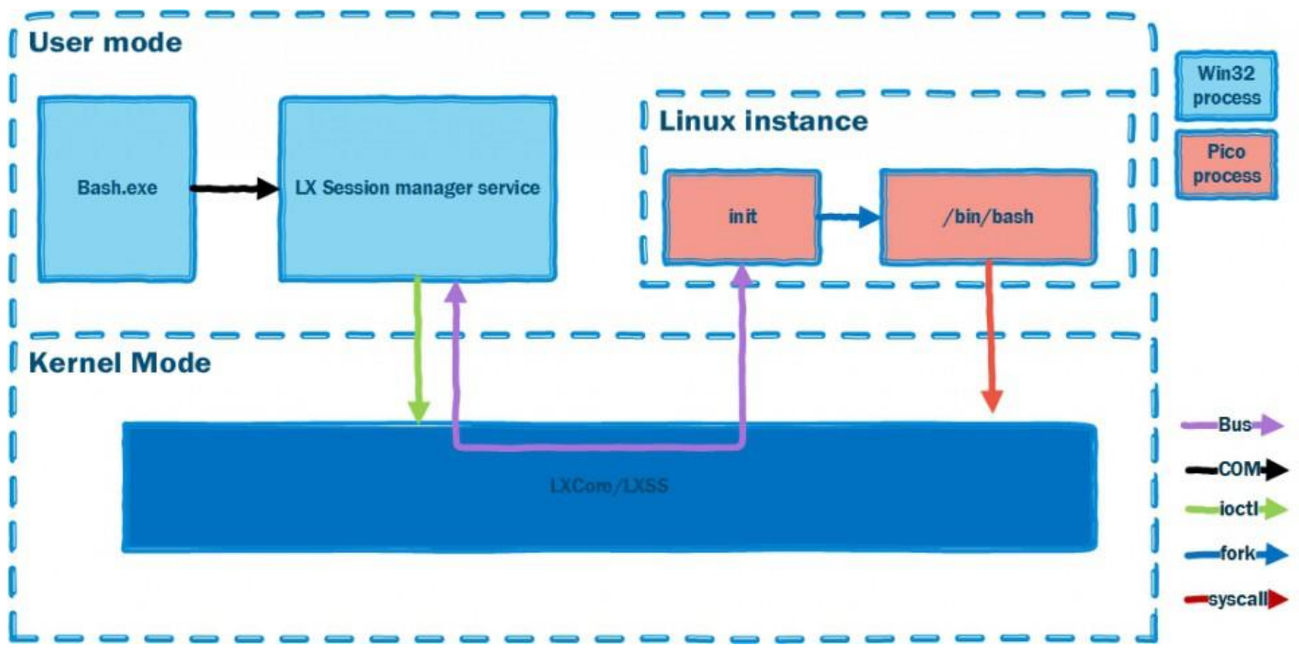
Further Reading





Why Applications Are OS Specific

- How
  - interpreted language
  - VM
  - only use standard APIs
- Still it is not a easy task
  - different binary format: ELF vs PE
  - different instruction set
  - different system call interfaces



WSL: Windows Subsystem for Linux

Figures: https://blogs.msdn.microsoft.com/wsl/2016/04/22/windows-subsystem-for-linux-overview/

Operating System Structure

Operating System Design and Implementation

- Design and Implementation of OS not “solvable”, but some approaches have proven successful
  - Internal structure of different Operating Systems can vary widely
  - Start by defining goals and specifications
  - Affected by choice of hardware, type of system

Worse is better, though it’s sad (in some cases)

The good news is that in 1995 we will have a good operating system and programming language; the bad news is that they will be Unix and C++.

Lips, Erlang

Worse Is Better Philosophy: Simple Is Better

- Simplicity-the design must be simple, both in **implementation** and **interface**. It is **more important for the interface to be simple** than the implementation.
- Correctness-the design **must be correct in all observable aspects**. Incorrectness is simply not allowed.
- Consistency-**the design must not be inconsistent**. A design is allowed to be **slightly less simple and less complete to avoid inconsistency**. Consistency is as important as correctness.
- Completeness-the design must cover as many important situations as is practical. All reasonably expected cases must be covered. **Simplicity is not allowed to overly reduce completeness**.

MIT/Stanford style of design

- Simplicity-the design must be simple, both in implementation and interface. It is **more important for the implementation to be simple** than the interface. Simplicity is the most important consideration in a design.
- Correctness-the design must be correct in all observable aspects. **It is slightly better to be simple than correct**.
- Consistency-the design **must not be overly inconsistent**. **Consistency can be sacrificed for simplicity in some cases**, but it is better to drop those parts of the design that deal with less common circumstances than to introduce either implementational complexity or inconsistency.
- Completeness-the design must cover as many important situations as is practical. All reasonably expected cases should be covered. Completeness can be sacrificed in favor of any other quality. **In fact, completeness must sacrificed whenever implementation simplicity is jeopardized**. **Consistency can be sacrificed to achieve completeness if simplicity is retained**; **especially worthless is consistency of interface.**

New Jersey approach

ading: https://www.jwz.org/doc/worse-is-better.html, http://blog.reverberate.org/2011/04/eintr-and-pc-loser-ing-is-better

Operating System Design and Implementation

- Important principle: to separate mechanism and policy
  - **mechanism**: **how** to do it
  - **policy**: **what/which** will be done

- Mechanisms determine how to do something, policies decide what/which will be done
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer)

Operating System Design and Implementation

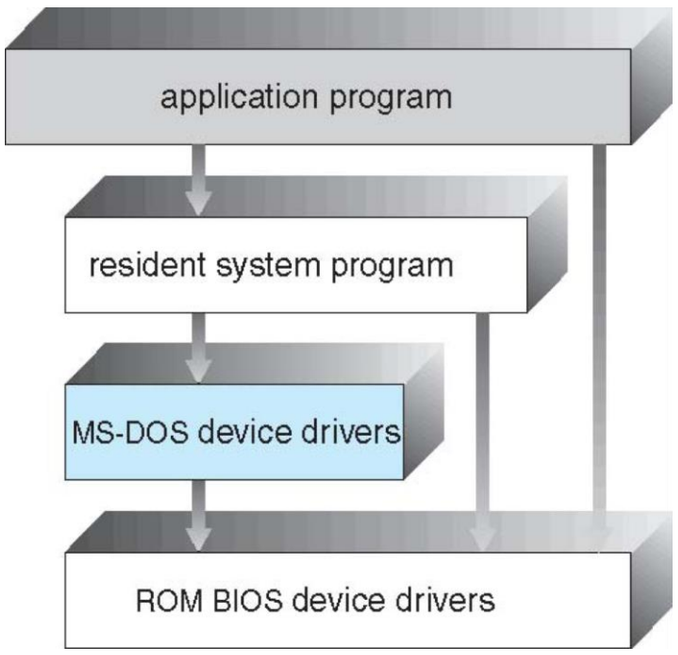
- Much variation
  - Early OSes in **assembly language**
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in **assembly**
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to port to other hardware
  - But slower

Operating System Structure

- Many structures:
  - **simple structure - MS-DOS**
  - **more complex -- UNIX**
  - **layered structure - an abstraction**
  - **microkernel system structure - L4**
  - **hybrid: Mach, Minix**
  - research system: **exokernel**

Simple Structure: MS-DOS

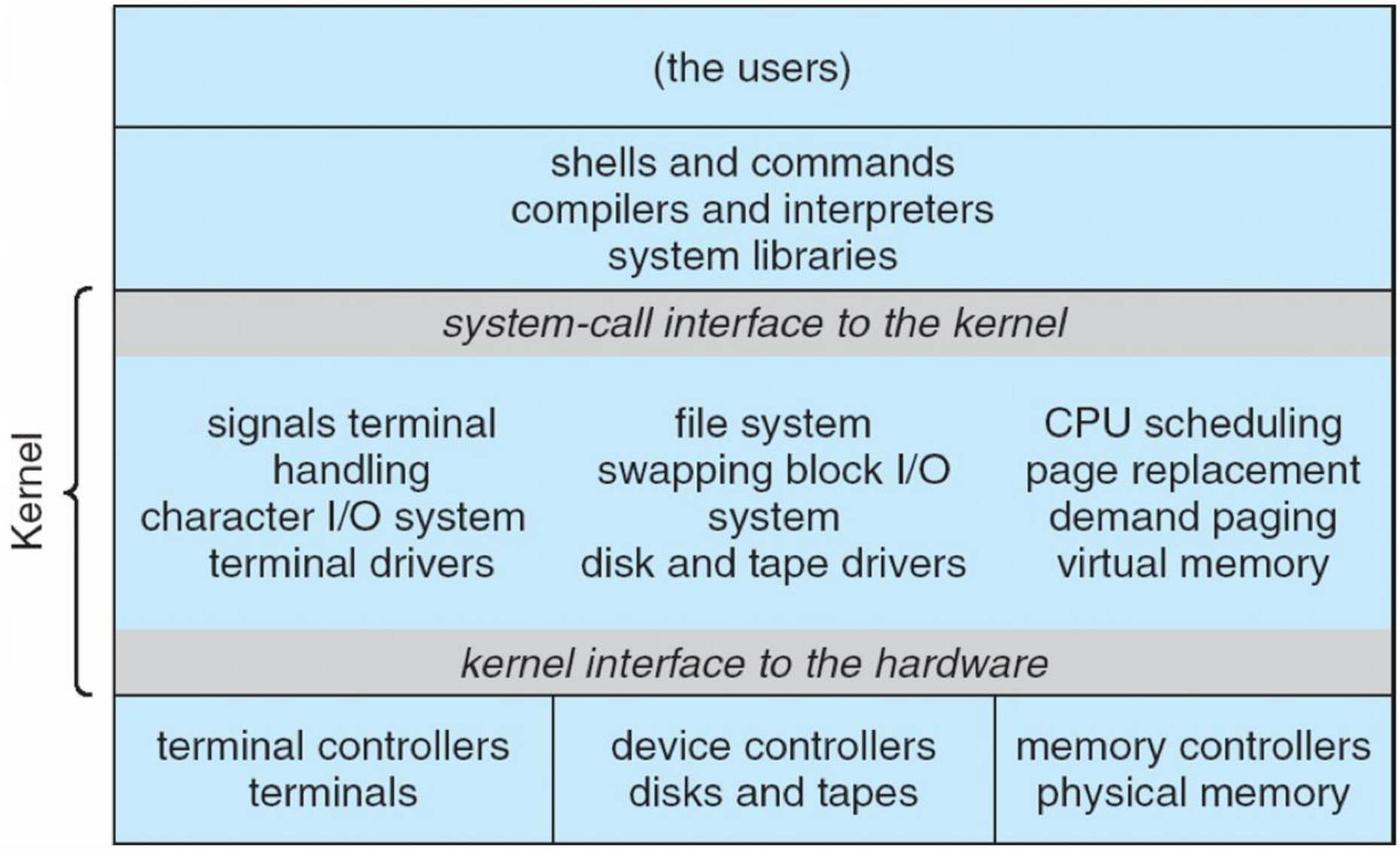
- No structure at all!: (1981~1994)
  - written to provide the most functionality in the least space
- A typical example: MS-DOS
  - Has some structures:
    - its interfaces and levels of functionality are not well separated
  - the kernel is not divided into modules



Monolithic Structure – Original UNIX

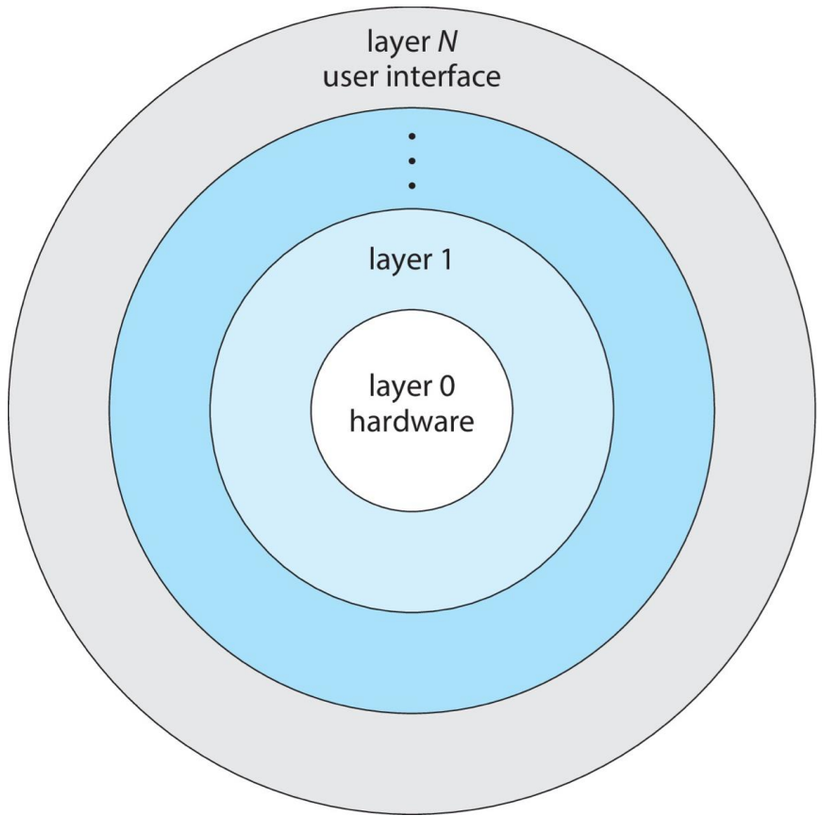
- Limited by hardware functionality, the original UNIX had limited structure
- UNIX OS consists of two separable layers
  - systems programs
  - **the kernel**: everything below the system-call interface and above physical hardware
    - a large number of functions for one level: file systems, CPU scheduling, memory management …

Traditional UNIX System Structure



Layered Approach

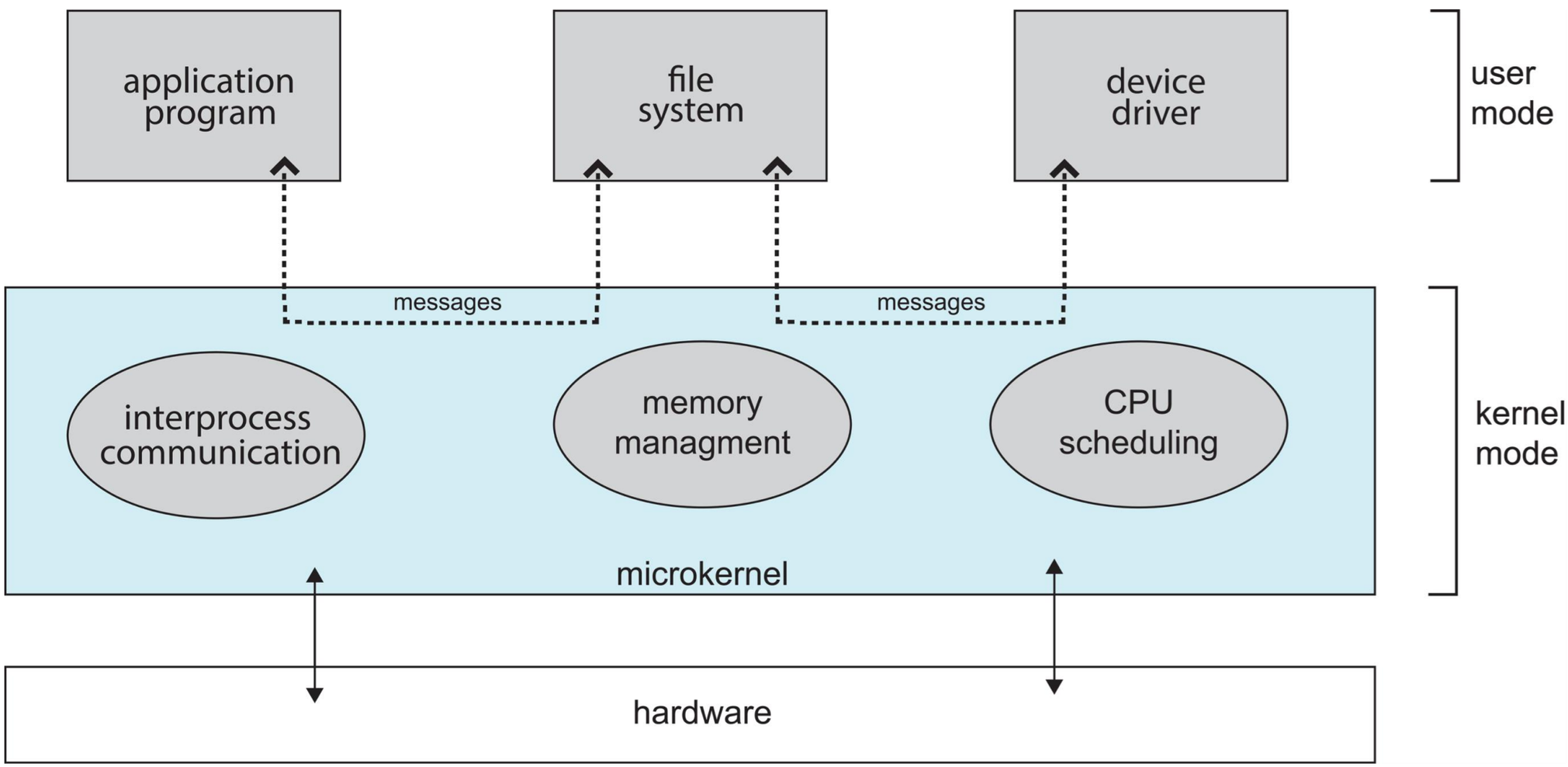
- The operating system is divided into a number of layers (levels)
  - each built on top of lower layers
  - The bottom layer (layer 0), is the hardware
  - the highest (layer N) is the user interface
- With modularity, layers are selected such that
  - each uses functions (operations) and services of only lower-level layers



Microkernel System Structure

- Microkernel moves as much from the kernel (e.g., file systems) into “user” space
- Communication between user modules uses **message passing**
- Benefits:
  - easier to extend a microkernel
  - easier to port the operating system to new architectures
  - more reliable (less code is running in kernel mode)
  - more secure
- Detriments:
  - performance overhead of user space to kernel space communication
- Examples: Minix, Mach, QNX, L4…

Microkernel System Structure

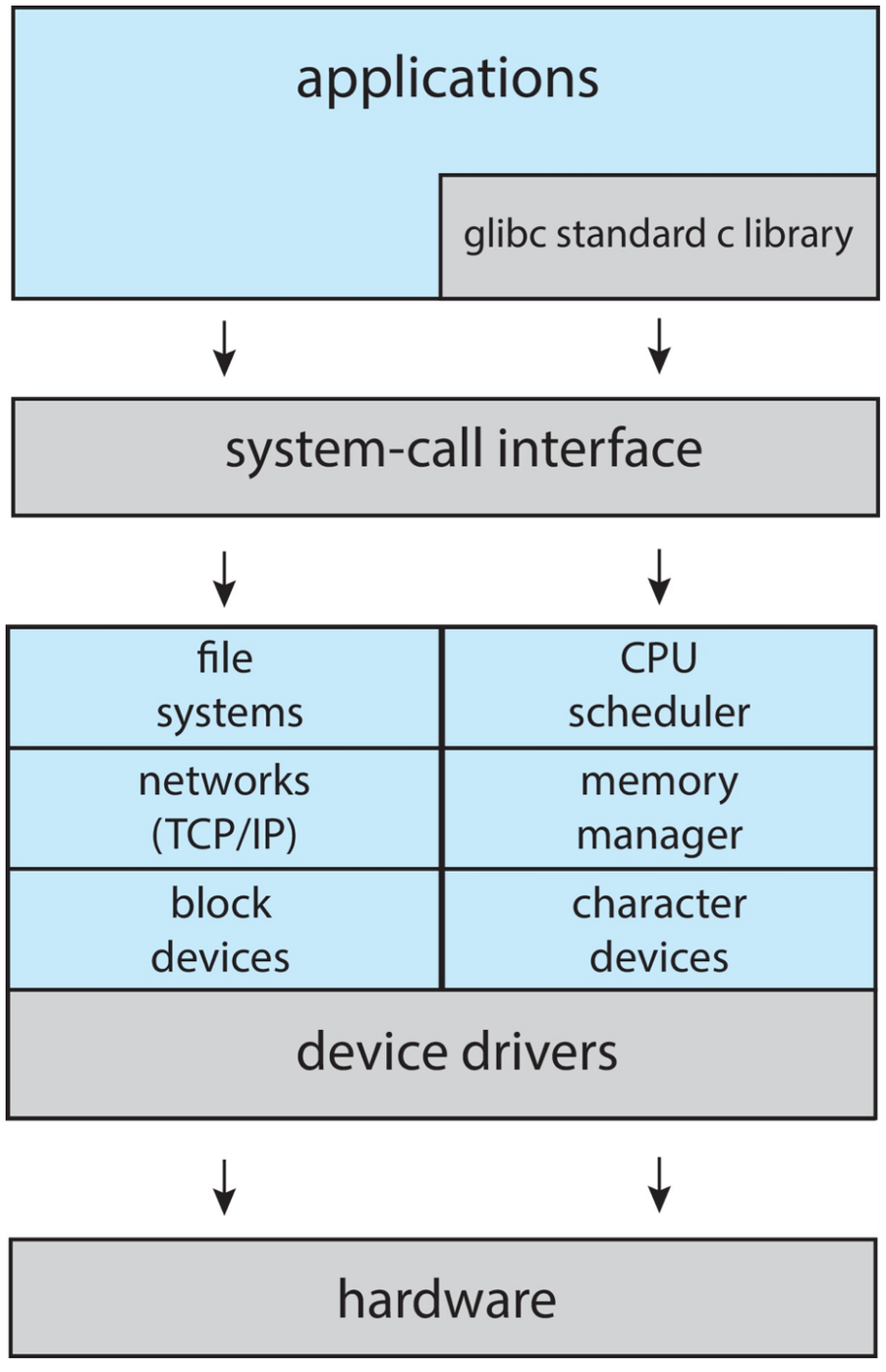


Modules

- Most modern operating systems implement **kernel modules**
  - uses **object-oriented** design pattern
  - each core component is separate, and has **clearly defined interfaces**
    - some are loadable as needed
- Overall, similar to layers but with more flexible
- Example: Linux, BSD, Solaris
  - http://www.makelinux.net/kernel_map/

Linux System Structure

- Monolithic plus modular design



Linux kernel map

functionalities

layers

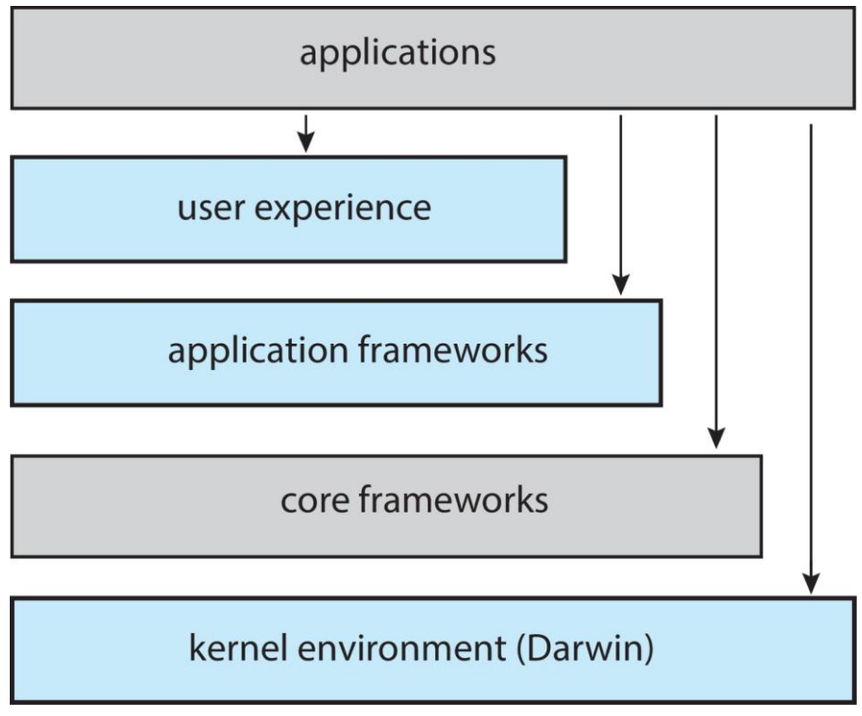
	human interface	system	processing	memory	storage	networking
user space interfaces system calls and system files	HI char devices	interfaces core	processes	memory access	files & directories access	sockets access
virtual	security	Device Model	threads	virtual memory	Virtual File System	protocol families
bridges cross-functional modules	debugging		synchronization	memory mapping	page cache swap	networking storage socket splice
logical functions implementations	HI subsystems	system run	Scheduler	logical memory	logical file systems	protocols
device control	abstract devices and HID class drivers	generic HW access	interrupts core	Page Allocator	block devices	network interface
hardware interfaces drivers, registers and interrupts	HI peripherals device drivers	device access and bus drivers	CPU specific	physical memory operations	disk controller drivers	network device drivers
electronics	user peripherals	I/O	CPU	memory	disk controllers	network controllers

Hybrid Systems

- Most modern operating systems are actually not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic, plus microkernel for different subsystem ***personalities***
- Apple Mac OS X hybrid, layered, **Aqua** UI plus **Cocoa** programming environment
  - Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)

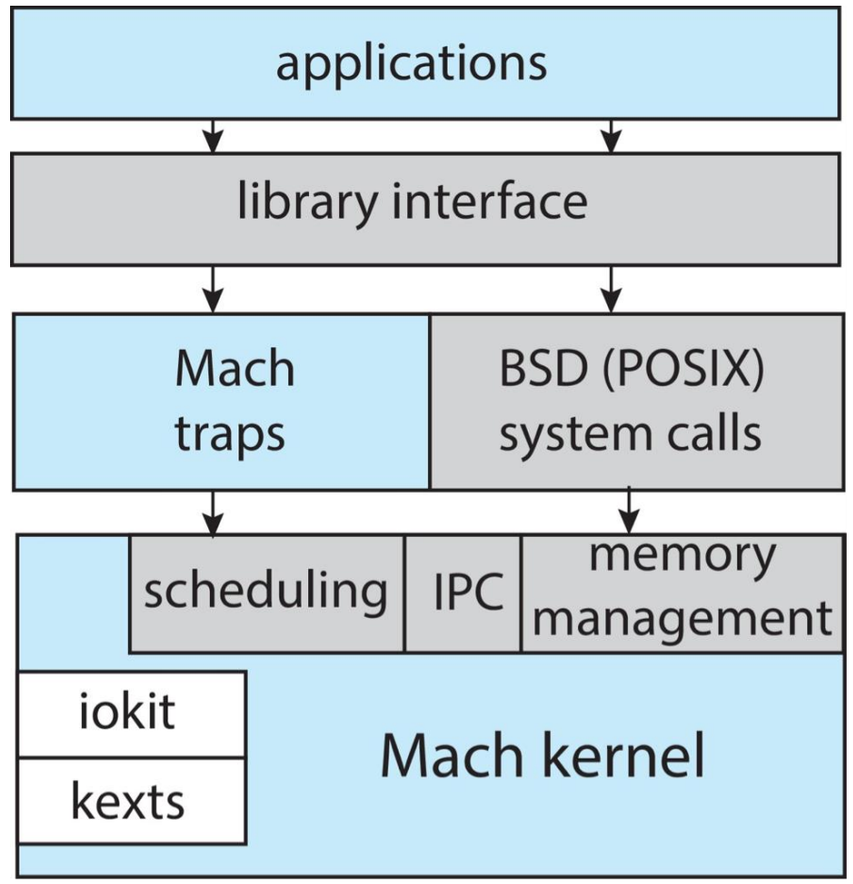
macOS and iOS Structure

- user experience: Aqua/Springboard user interface
- Application frameworks: Cocoa (Touch) provides API for Object C and Swift programing languages
- Core frameworks: defines frameworks that support graphics and media

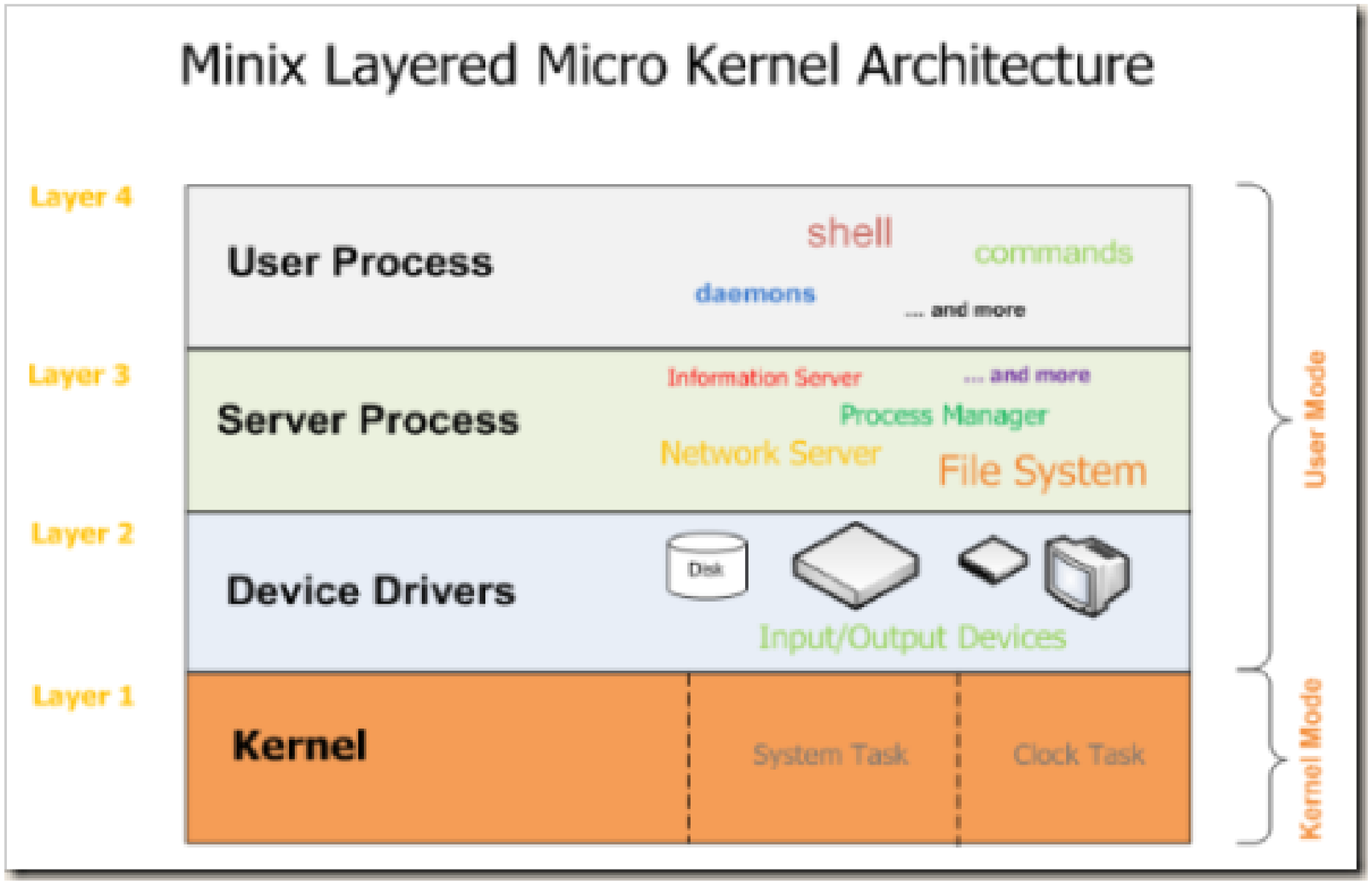


Darwin: layered + microkernel + modules

- Two system-call interfaces: Mach(trap), BSD(POSIX)
- Mach provides basic OS services: MM, scheduling, IPC.
- These services are through tasks (Mach process), threads, memory objects and ports (used for IPC)
  - fork() in BSD -> kernel abstraction (task) in Mach
- Kexts: kernel extensions



Layered Microkernel: Minix



Figures:https://imma.wordpress.com/2007/04/02/presentation-internal-structure-of-minix/

Exokernel: Motivation

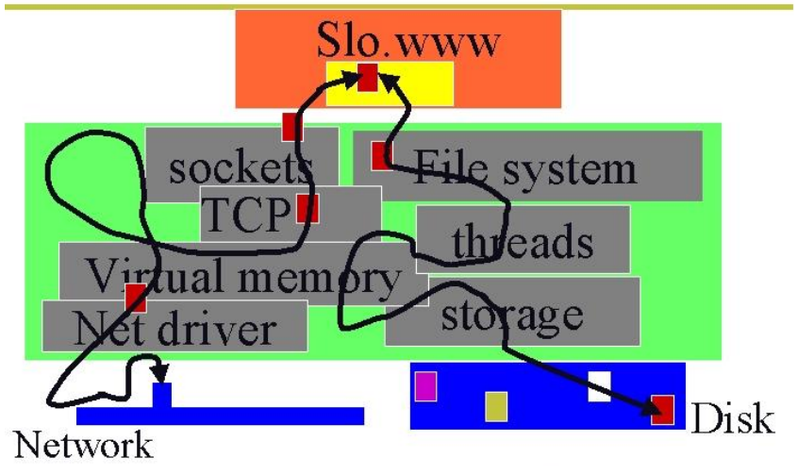
- In traditional operating systems, only **privileged servers and the kernel** can manage system resources

- Un-trusted applications are required to interact with the hardware via some **abstraction model**
  - File systems for disk storage, virtual address spaces for memory, etc.

- **But application demands vary widely**!!
  - An interface designed to accommodate every application must anticipate all possible needs

Exokernel: Motivation

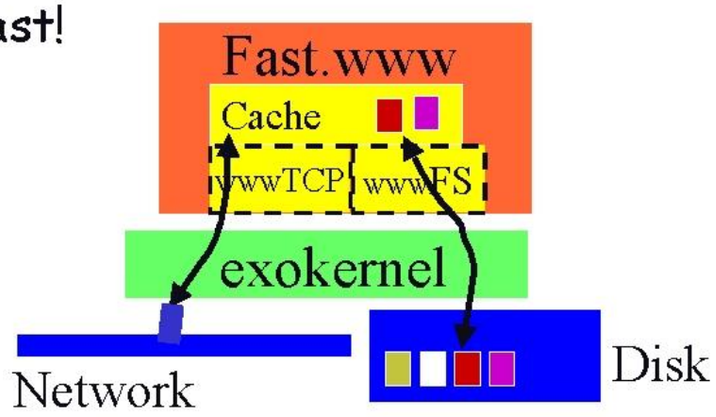
Traditional OS structure



- slow and can't fix it!

Exokernel: application control

- Application software can override OS
- Fast!



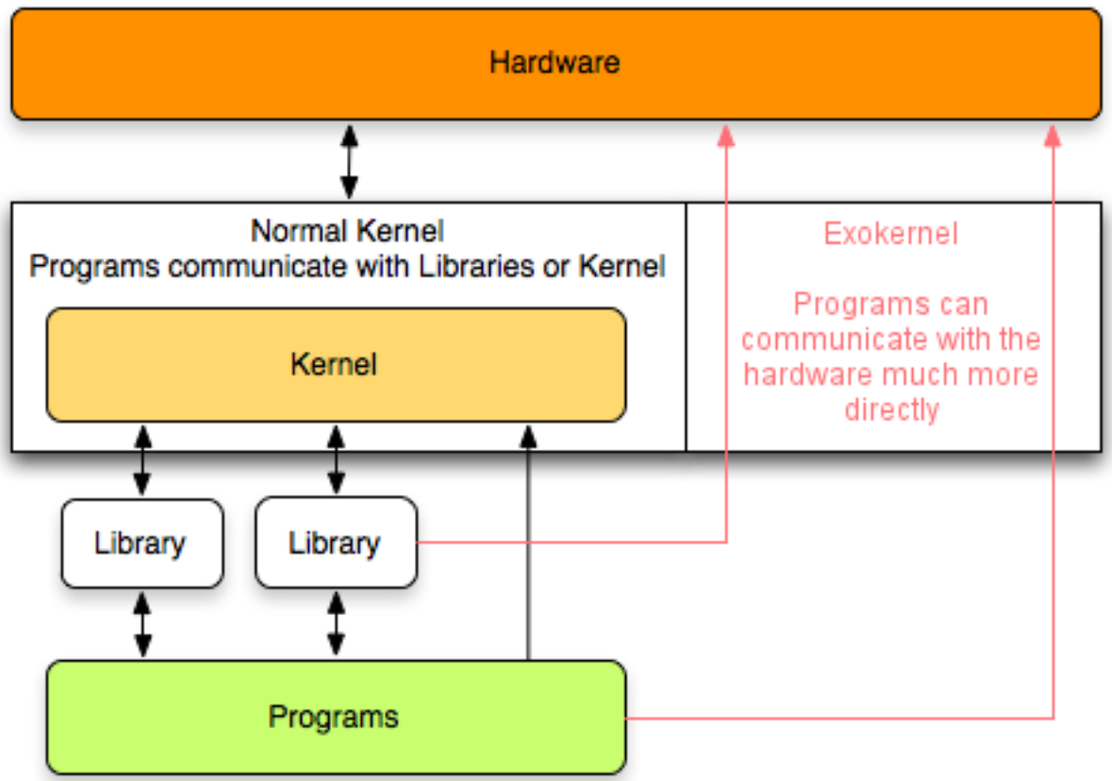
Figures: https://pdos.csail.mit.edu/archive/exo/exo-slides/

Exokernel: Motivation

- Give **un-trusted applications as much control** over physical resources **as possible**
- To force **as few abstraction as possible** on developers, enabling them to make as many decisions as possible about hardware abstractions.
    - Let the kernel allocate the basic physical resources of the machine
    - Let each program decide what to do with these resources
- Exokernel separate **protection** from **management**
    - They **protect** resources but delegate **management** to application

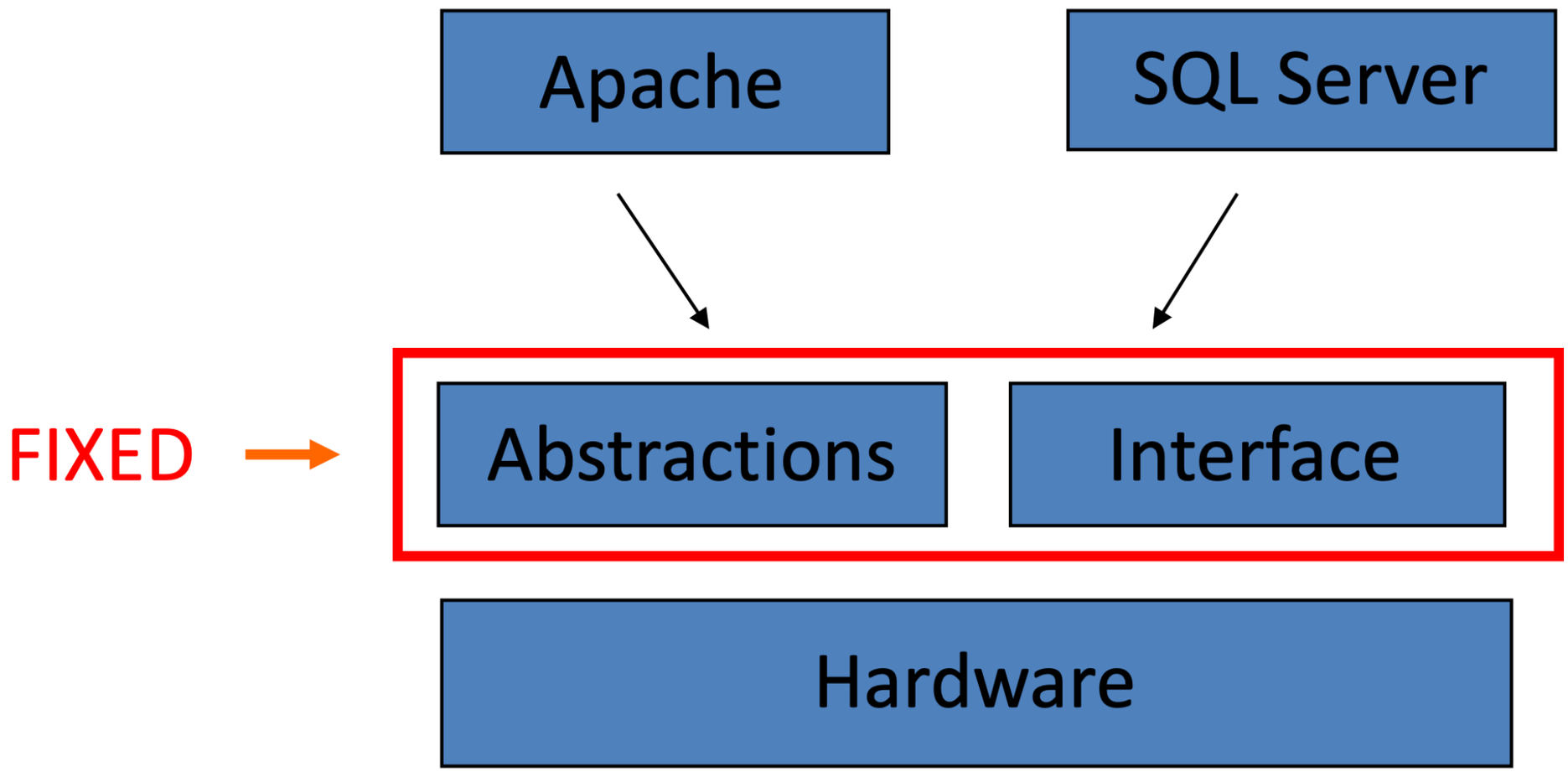
Exokernel

- Exokernel give more direct access to the hardware, thus removing most abstractions

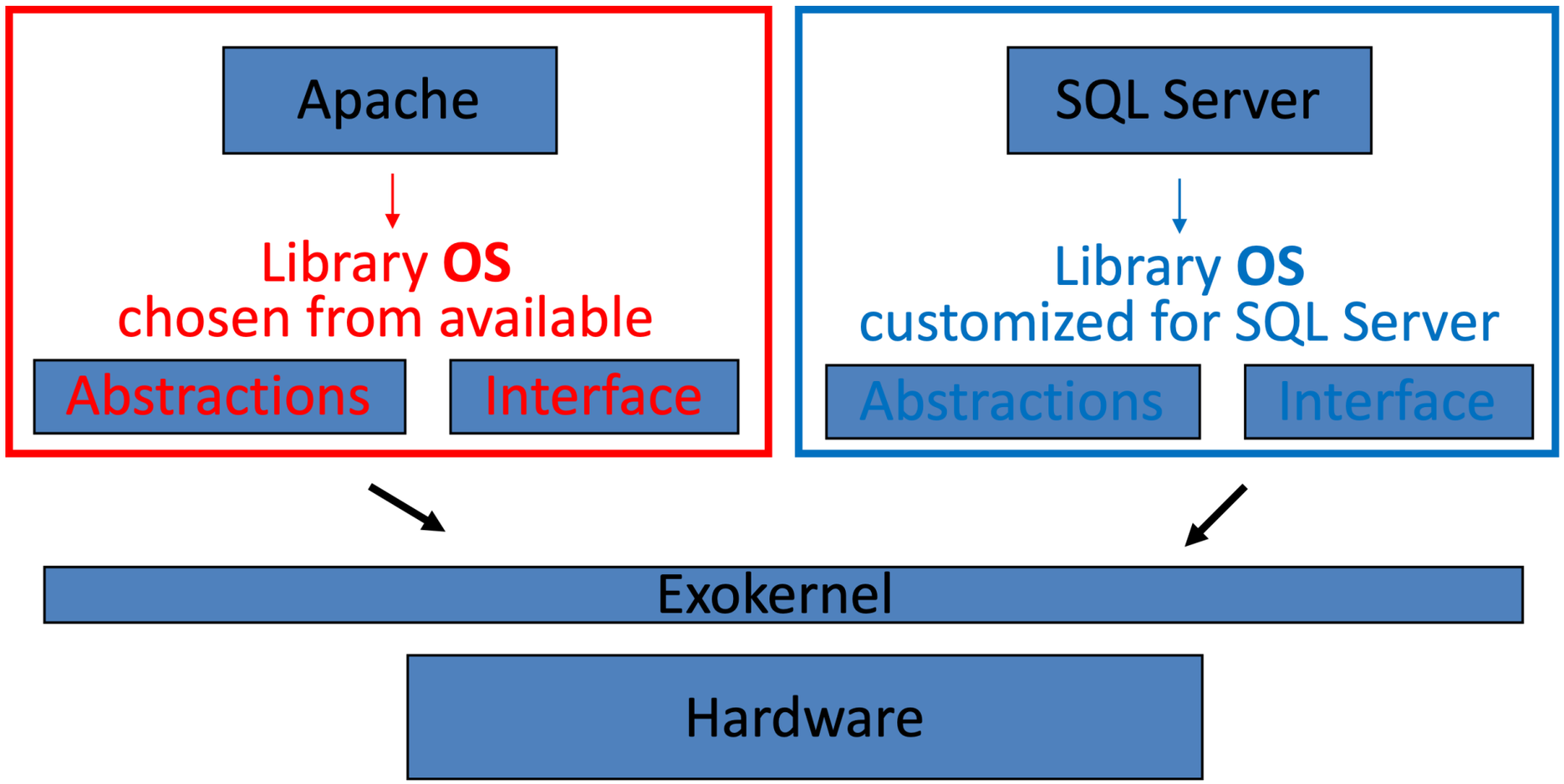


Figures:https://medium.com/@vithushaaarabhi/exokernels-an-operating-system-architecture-for-application-level-resource-management-32d0daaeeab0

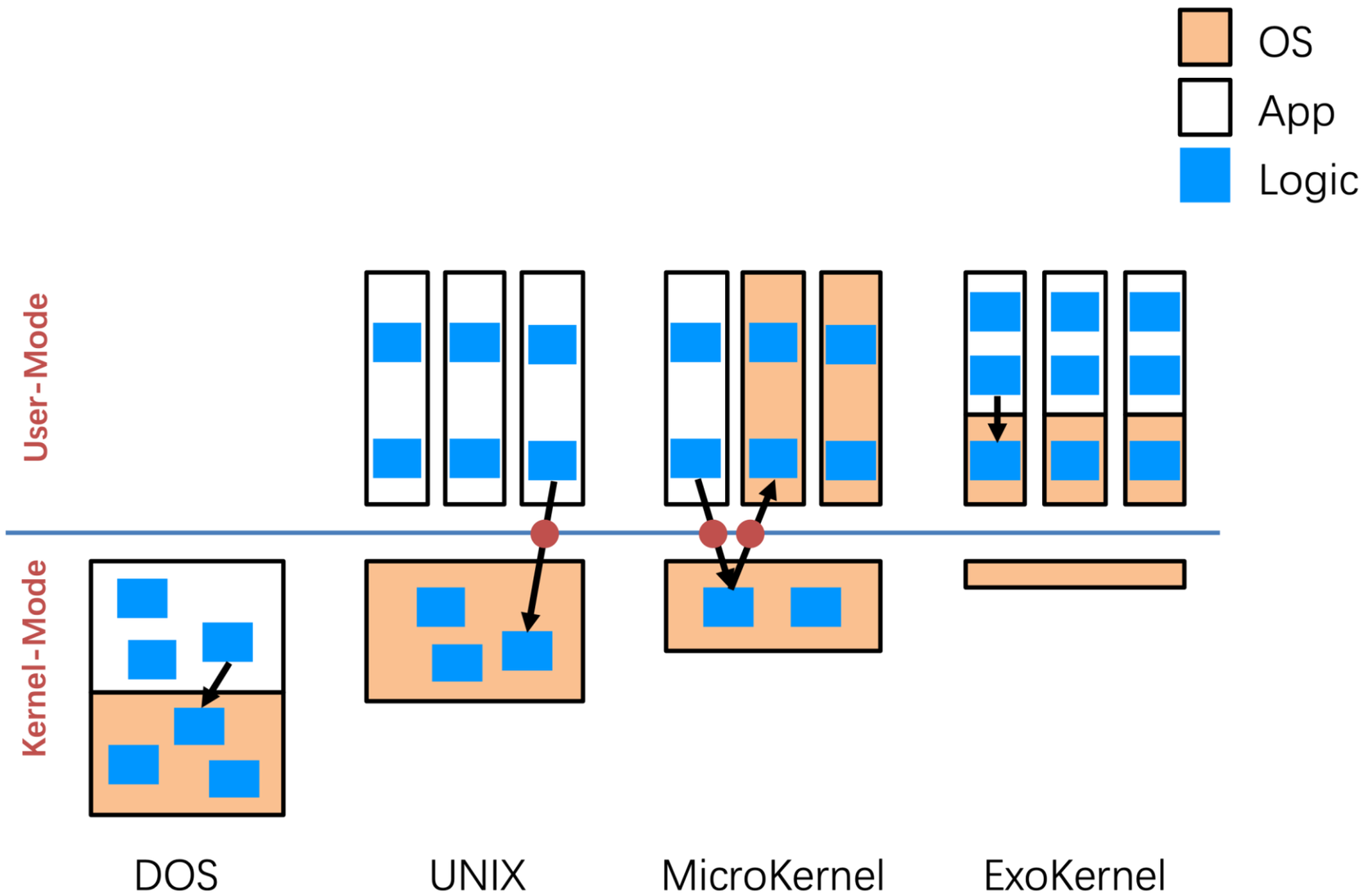
Traditional OS



Exokernel


Apache
Library OS
chosen from available
Abstractions
Interface
SQL Server
Library OS
customized for SQL Server
Abstractions
Interface
Exokernel
Hardware

Comparison


OS
App
Logic
User-Mode
Kernel-Mode
DOS
UNIX
MicroKernel
ExoKernel

Tracing

- Collects data for a specific event, such as steps involved in a system call invocation
- Tools include
  - strace – trace system calls invoked by a process
  - gdb – source-level debugger
  - perf – collection of Linux performance tools
  - tcpdump – collects network packets

Strace

```
os@os:~/os2018fall/code/1_cpu$ strace ./cpu 'A'
execve("./cpu", ["./cpu", "A"], [/* 32 vars */]) = 0
```

```
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
brk(NULL)                               = 0xedd000
brk(0xefe000)                           = 0xefe000
write(1, "A\n", 2A
)                       = 2
write(1, "A\n", 2A
)                       = 2
write(1, "A\n", 2A
)                       = 2
^C--- SIGINT {si_signo=SIGINT, si_code=SI_KERNEL} ---
strace: Process 26654 detached
```

HW2 is out!