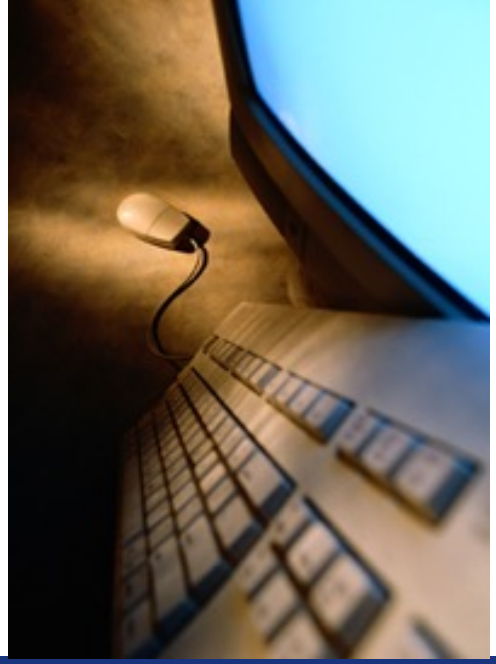
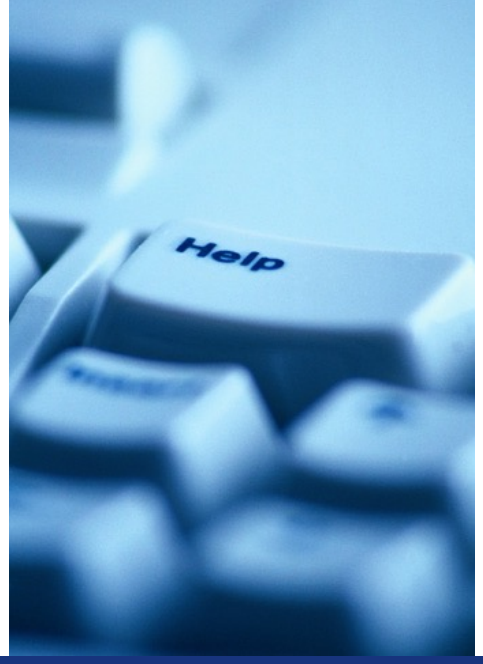
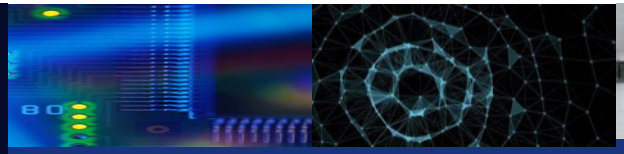



操作系统原理

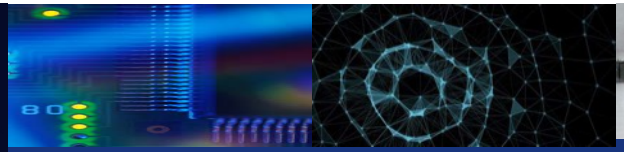
Operating Systems Principles

陈鹏飞
计算机学院
2022-02



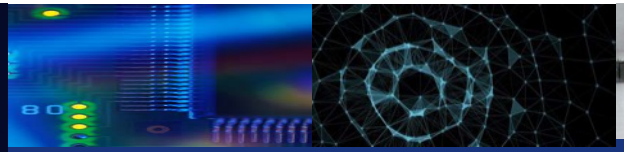
第二讲 —操作系统结构

https://www.bilibili.com/video/av55900643/



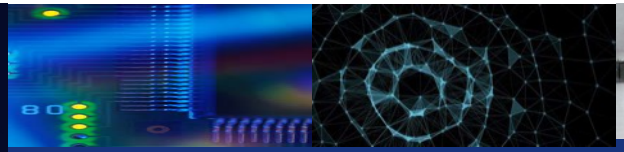

概述

- 操作系统发展
- 操作系统服务
- 操作系统设计和实现
- 操作系统结构
- 操作系统调试、生成和引导
- 操作系统实例



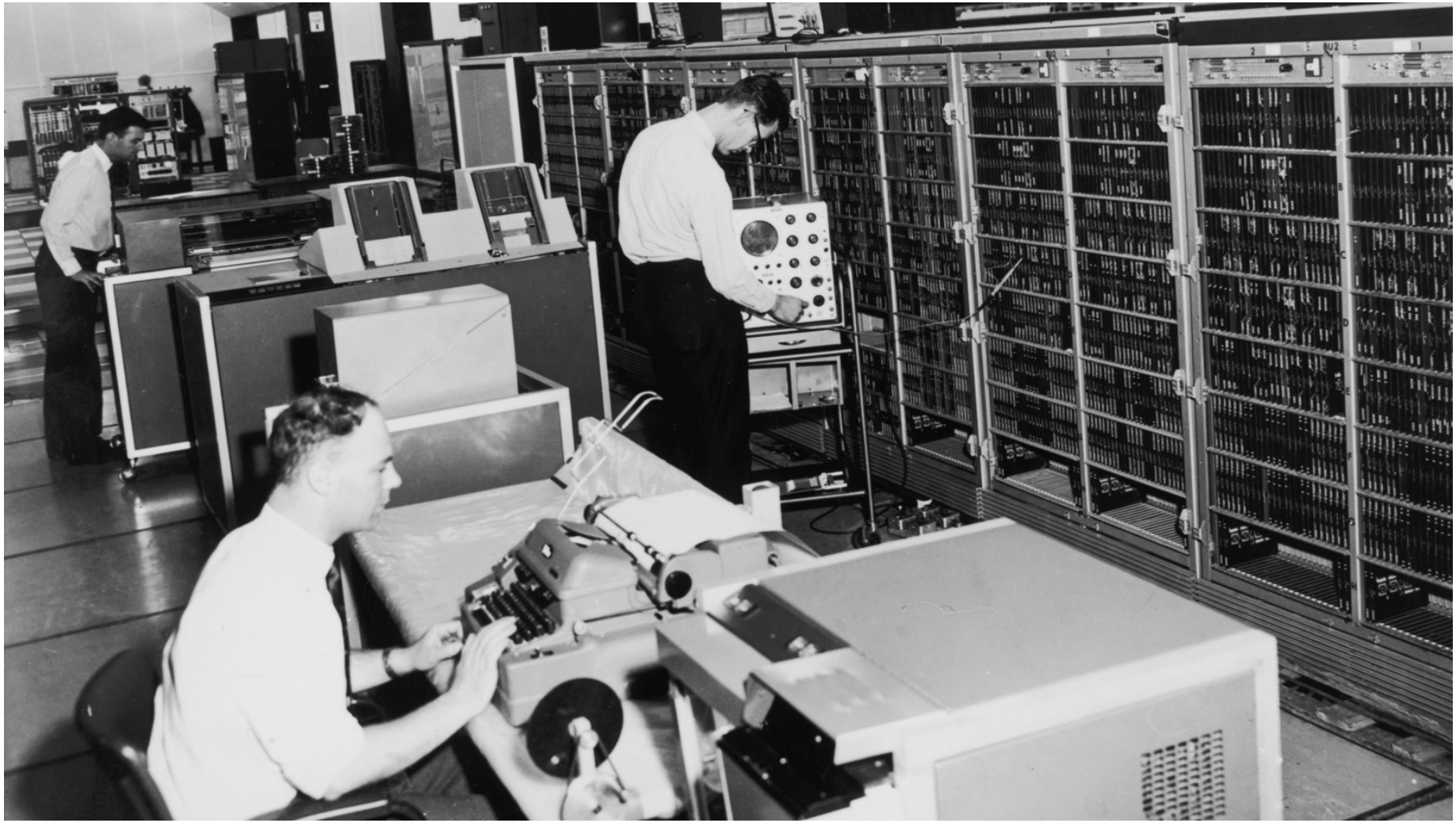
概述

- 识别操作系统提供的服务
- 说明如何使用系统调用来提供操作系统服务
- 比较和对比设计操作系统的单片、分层、微核、模块化和混合策略
- 演示引导操作系统的过程
- 应用用于监视操作系统性能的工具
- 设计和实现与Linux内核交互的内核模块



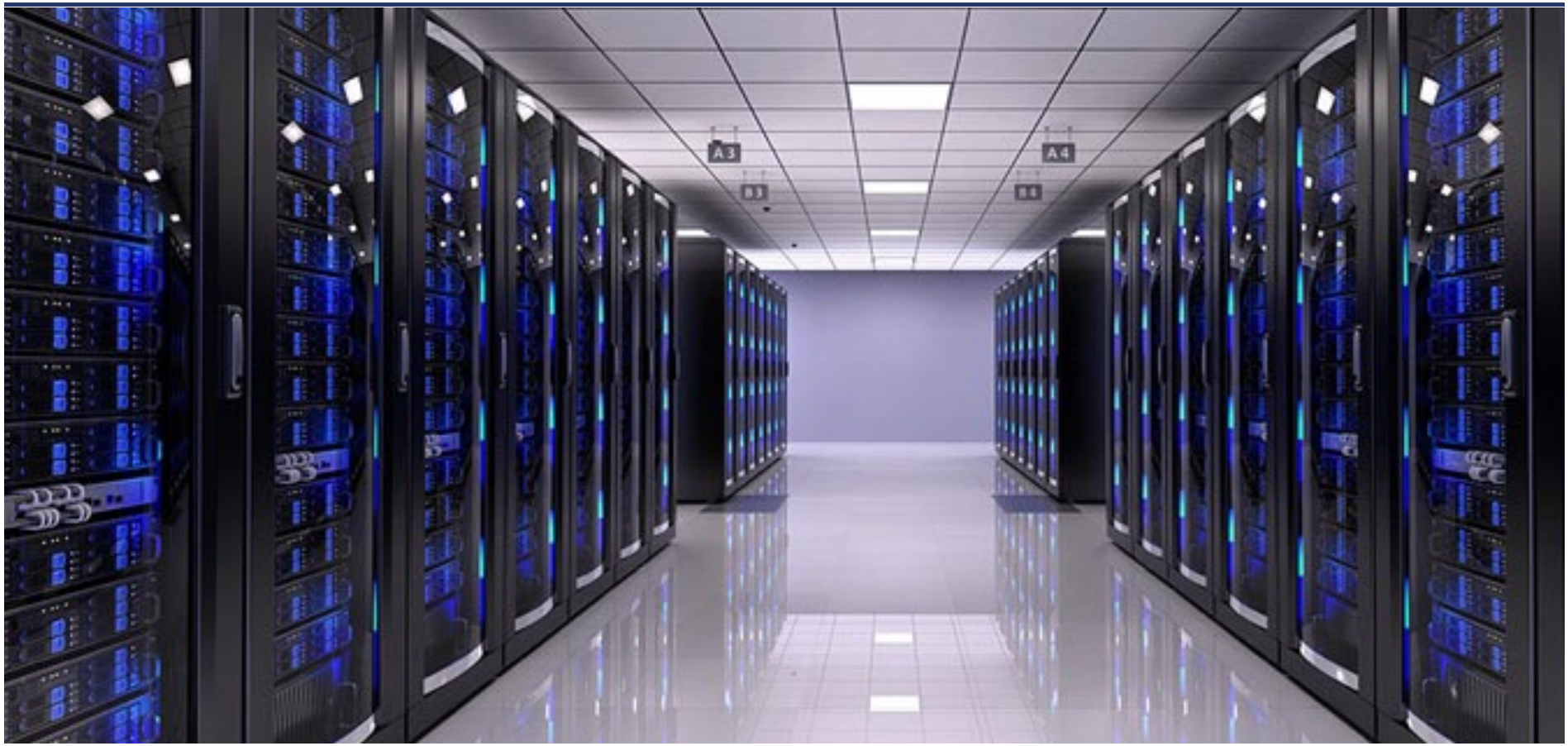
1. 操作系统发展

历史上的操作系统



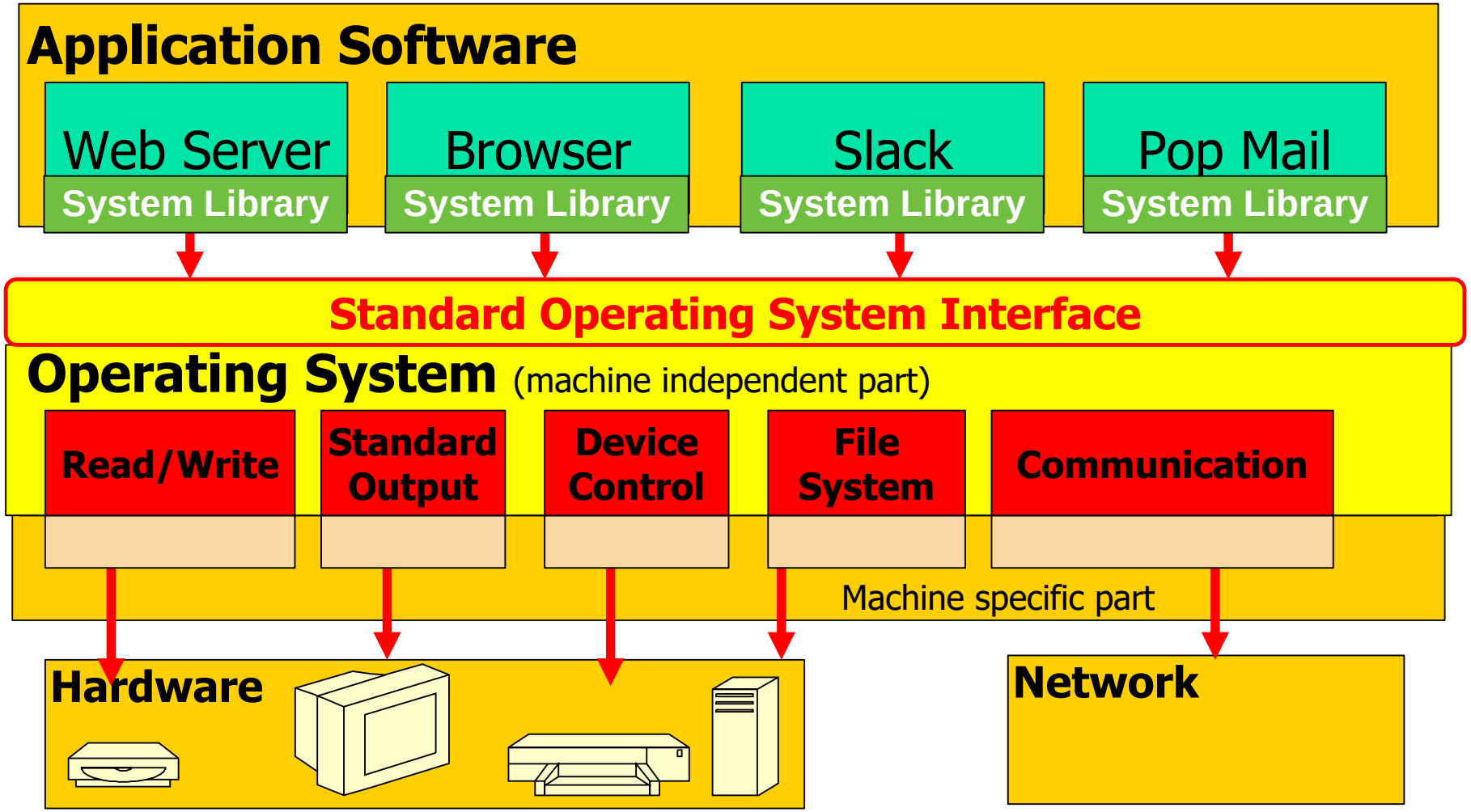
当前操作系统

面向大规模数据中心的操作系统，管理海量的资源，多大百万级的机器；



站在五千米高空看操作系统

操作系统是软硬件之间的桥梁，提供了访问硬件的接口；



操作系统的发展

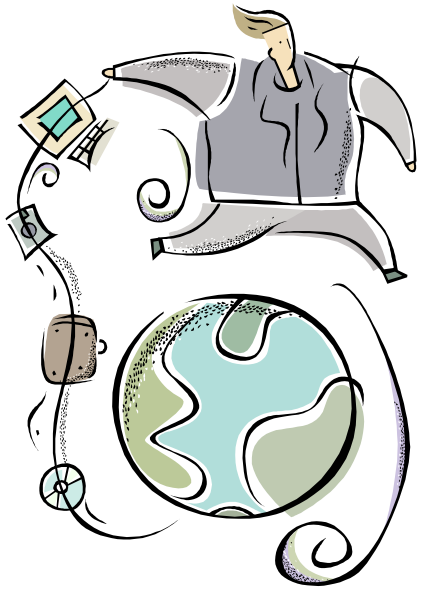
- 操作系统随着时间发展的几个原因：

hardware upgrades

new types of hardware

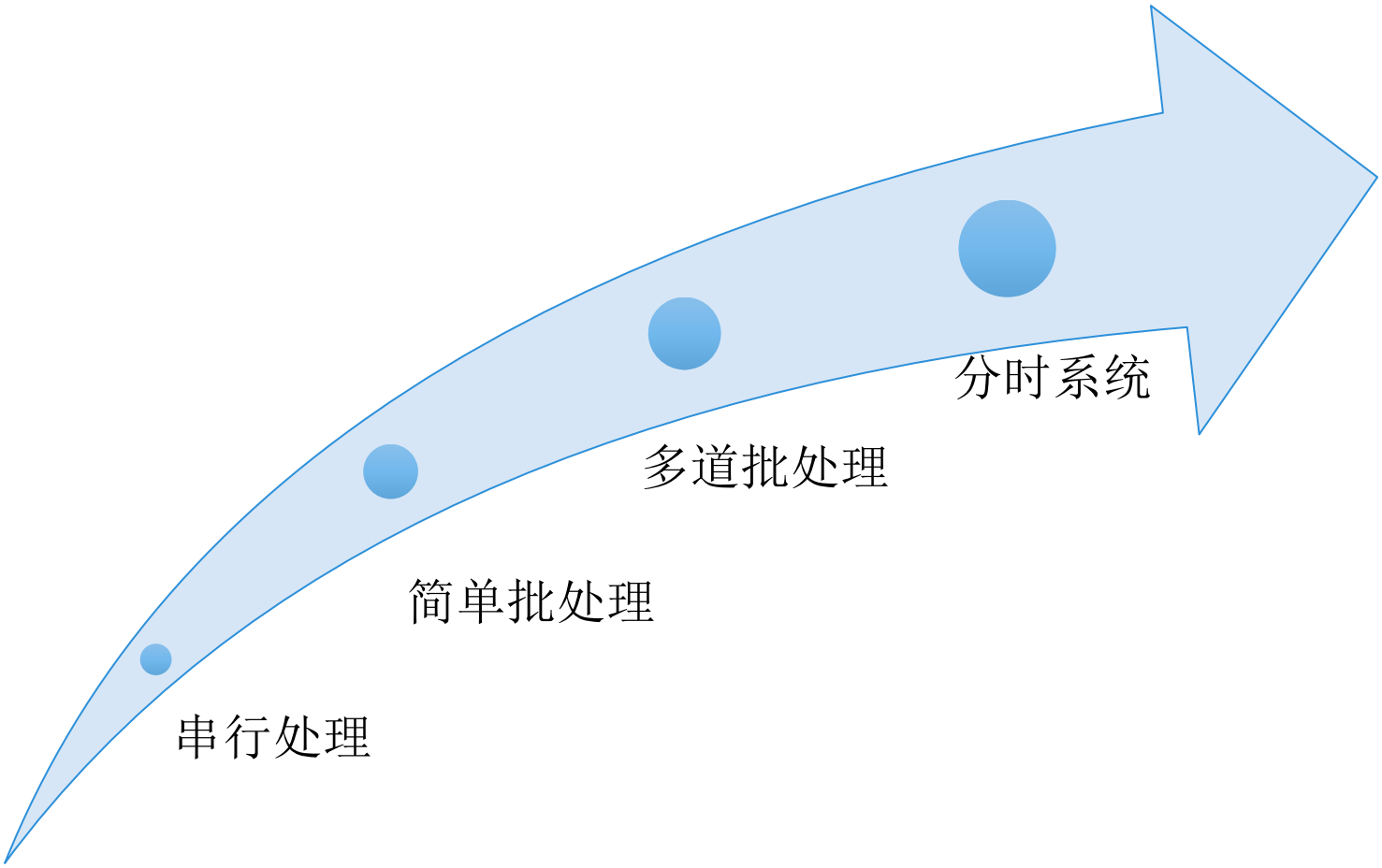
new services

Fixes



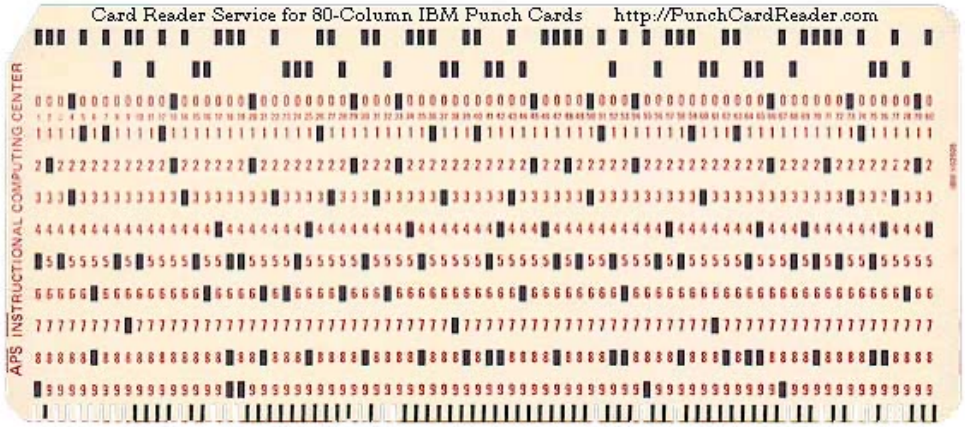
操作系统的发展

- 操作系统发展的几个阶段：

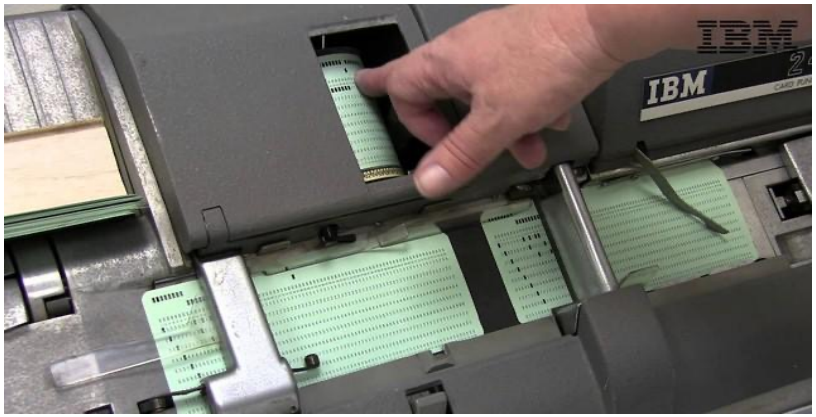


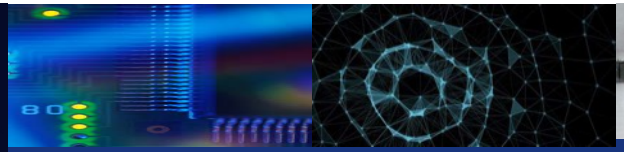
串行处理（1940’~1950’）

- 最早的计算机：
  - 没有操作系统：
    - 程序员直接跟计算机硬件打交道；
  - 计算机 有控制台可以显示灯光、控制器 、开关和一些输出设备、打印机等；
  - 用户串行地控制计算机；



- 问题：
  - 调度：
    - 大多数装置利用硬拷贝登记表来预订机器时间；
    - 时间会闲置，导致资源浪费；
  - 准备时间：
    - 加载程序的时间非常长，需要花费大量的时间；





简单批处理系统

- 早期的计算机非常昂贵
  - 因此最大化处理器的利用率非常重要；
- 监控程序
  - 用户不再直接访问处理器；
  - 操作员把这些作业按顺序组织成批，并将整个批作业放在输入设备上，供监控程序使用；
  - 每个程序完成处理后返回监控程序，同时监控程序自动加载下一个程序；

监控程序的视角

- Monitor controls the sequence of events
- *Resident Monitor* is software always in memory
- Monitor reads in job and gives control
- Job returns control to monitor

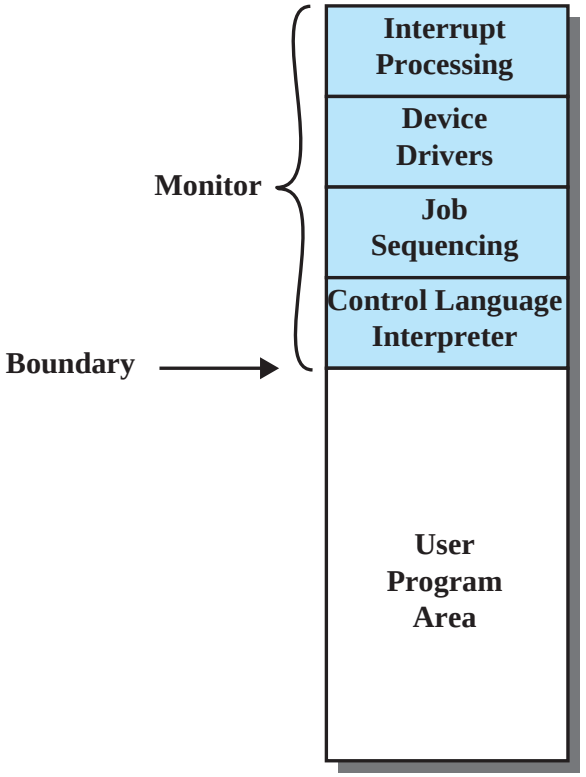
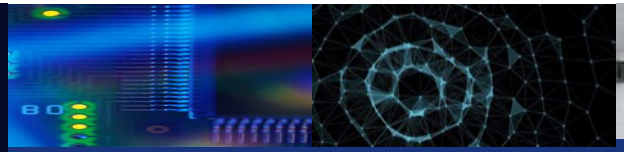


Figure 2.3 Memory Layout for a Resident Monitor



处理器的视角

- 处理器执行内存中存储的监控程序中的指令；
- 处理器执行用户程序指令，直到遇到一个结束指令或错误指令；
- 控制权交给作业意味着处理器当前取的和执行的都是用户程序的指令；
- 控制权返回给监控器意味着处理器当前从监控程序中取指令并执行指令；

作业控制语言（JCL）

Special type of programming language used to provide instructions to the monitor

what compiler to use

what data to use

```
$JOB
$FTN
•
•
•
$LOAD
$RUN
•
•
•
$END
```

FORTRAN instructions

Data

硬件考虑

Memory protection for monitor

- while the user program is executing, it must not alter the memory area containing the monitor

Timer

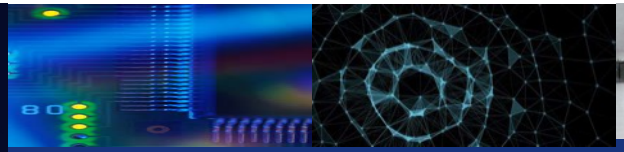
- prevents a job from monopolizing the system

Privileged instructions

- can only be executed by the monitor

Interrupts

- gives OS more flexibility in controlling user programs



操作系统模式

User Mode

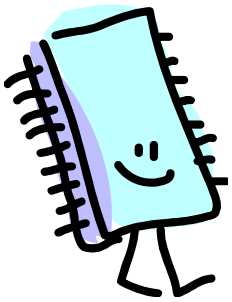
- user program executes in user mode
- certain areas of memory are protected from user access
- certain instructions may not be executed

Kernel Mode

- monitor executes in kernel mode
- privileged instructions may be executed
- protected areas of memory may be accessed

简单批处理系统的开销

- 用户程序和监控程序交替执行；
  - 牺牲了一部分主存，交给了监控程序；
  - 消耗了一部分机器时间；
- 虽然存在系统开销，但简单批处理系统还是提高了计算机的利用率；



多道批处理系统

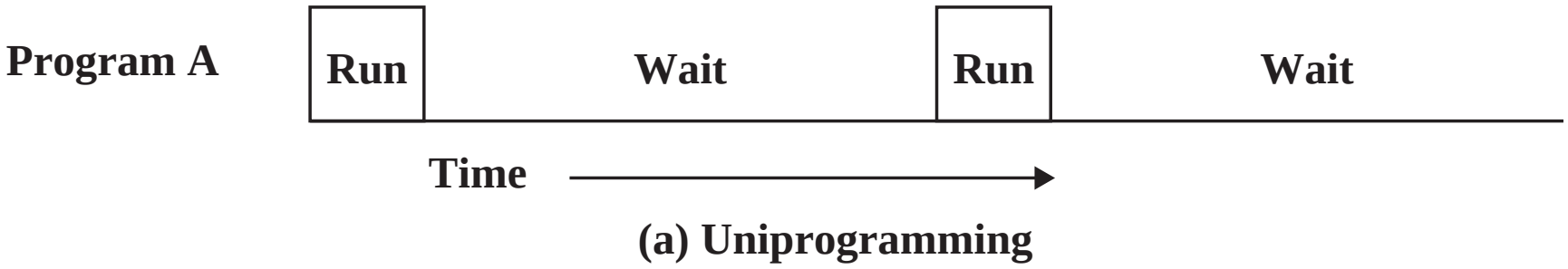
Read one record from file	15 μs
Execute 100 instructions	1 μs
Write one record to file	15 μs
TOTAL	31 μs

Percent CPU Utilization $= \frac{1}{31} = 0.032 = 3.2\%$

Figure 2.4 System Utilization Example

- **Processor is often idle**
  - even with automatic job sequencing
  - I/O devices are slow compared to processor

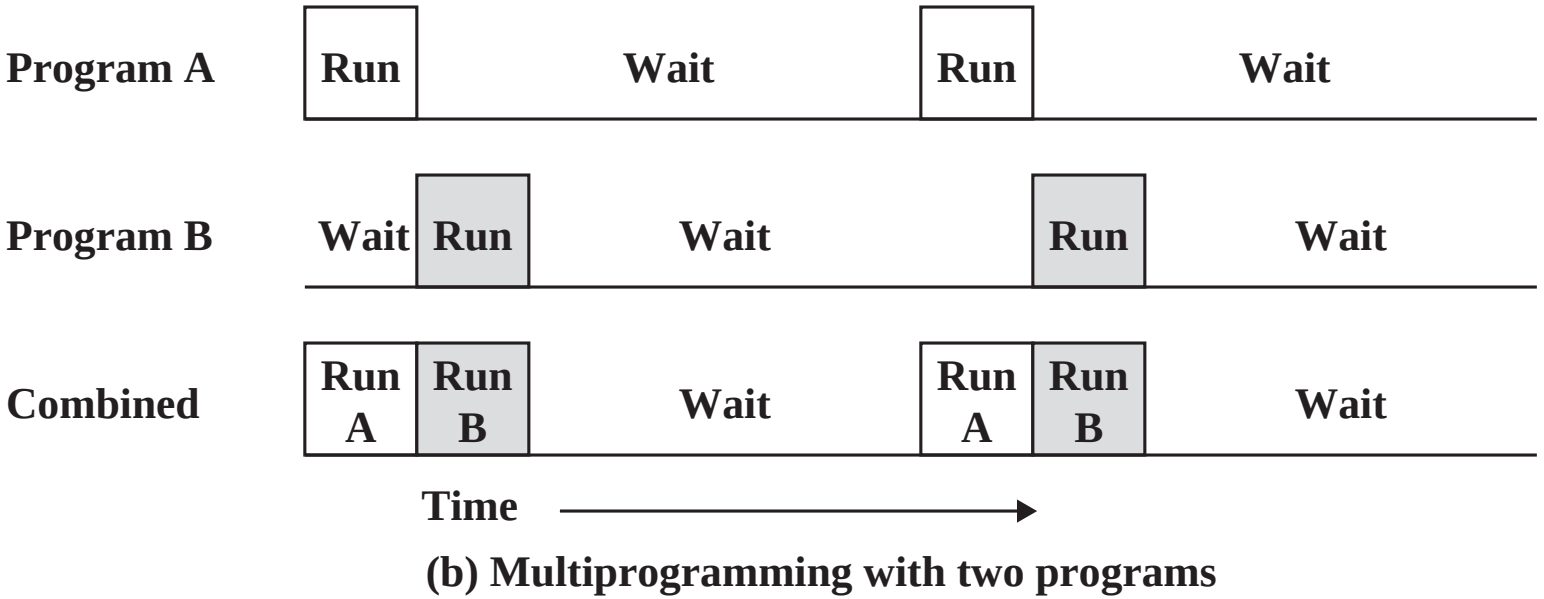
单道程序设计



(a) Uniprogramming

- 处理器花费一定的运行时间进行计算，直到遇到一个I/O指令，这时它需等到I/O指z令结束才能继续进行。

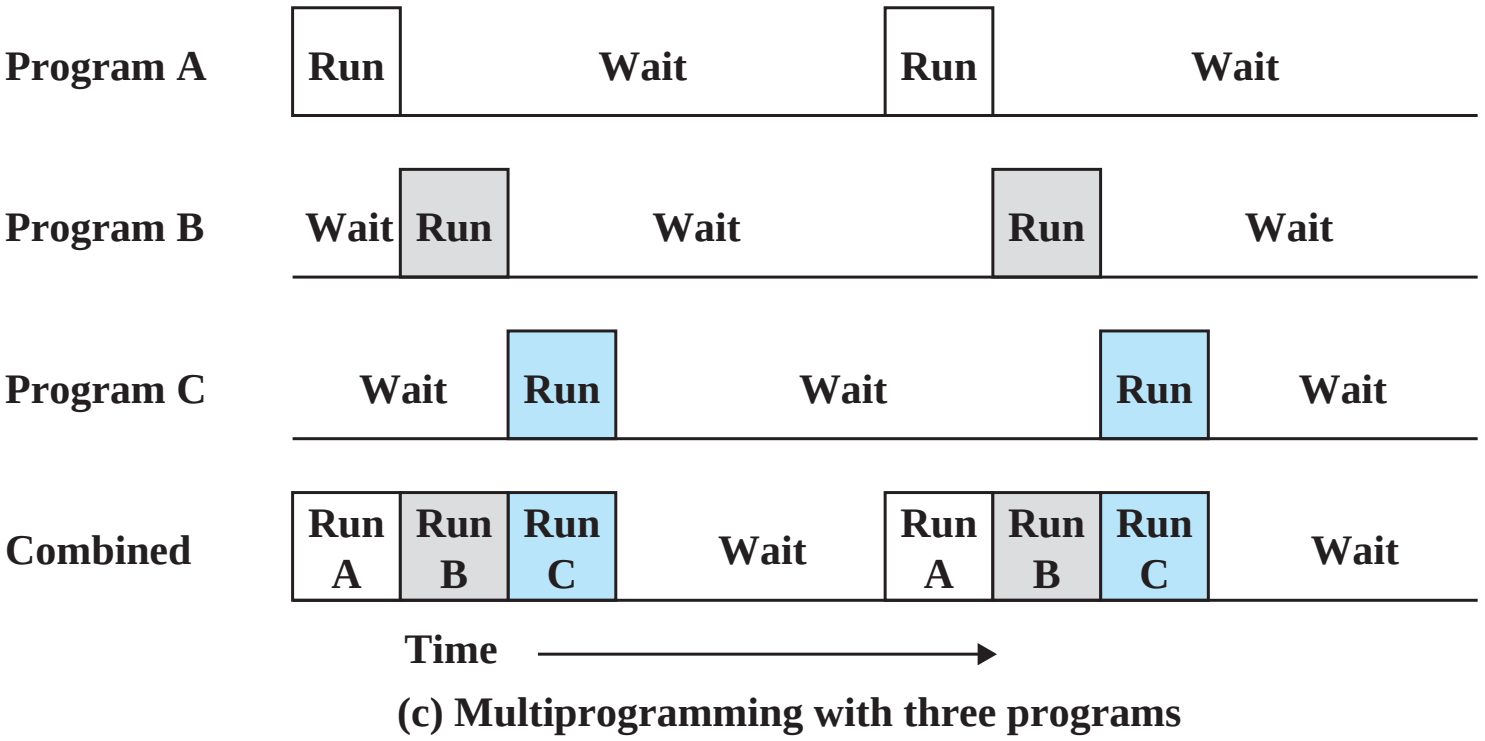
多道程序设计



(b) Multiprogramming with two programs

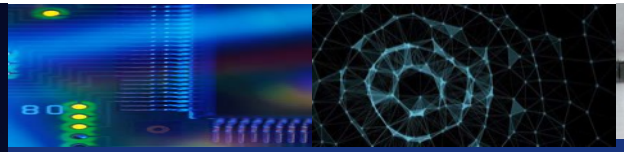
- There must be enough memory to hold the OS (resident monitor) and one user program
- When one job needs to wait for I/O, the processor can switch to the other job, which is likely not waiting for I/O

多道程序设计



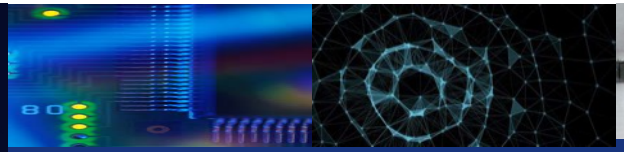
(c) Multiprogramming with three programs

- 多道程序设计
  - 也称为多任务处理；
  - 扩展存储器以保证可以保存3、4个或者更多程序，且在它们之间进行切换；



多道程序设计样例

	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min	15 min	10 min
Memory required	50 M	100 M	75 M
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes



资源利用率

	Uniprogramming	Multiprogramming
Processor use	20%	40%
Memory use	33%	67%
Disk use	33%	67%
Printer use	33%	67%
Elapsed time	30 min	15 min
Throughput	6 jobs/hr	12 jobs/hr
Mean response time	18 min	10 min

资源利用率直方图

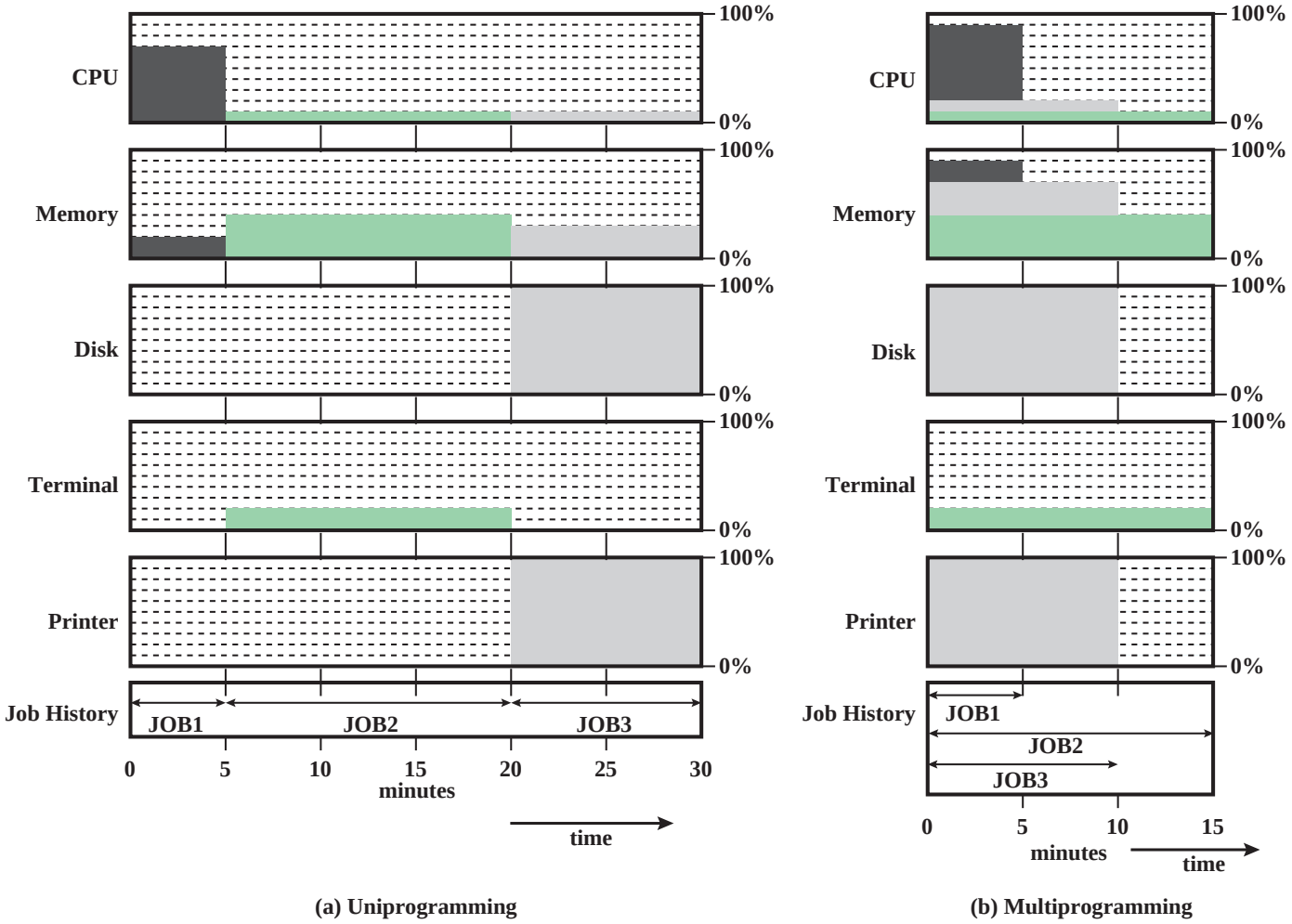
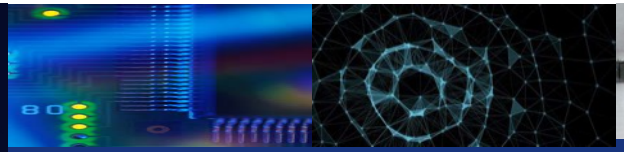
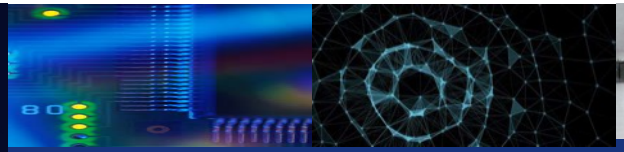


Figure 2.6 Utilization Histograms



分时系统

- 能够用于处理多个交互式作业；
- 处理器的时间在多个用户之间共享；
- 多个用户通过终端同时访问系统，由操作系统控制每个用户程序在很短的时间内交替执行；



多道批处理VS分时系统

	Batch Multiprogramming	Time Sharing
Principal objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job control language commands provided with the job	Commands entered at the terminal

Table 2.3 Batch Multiprogramming versus Time Sharing

Multics OS

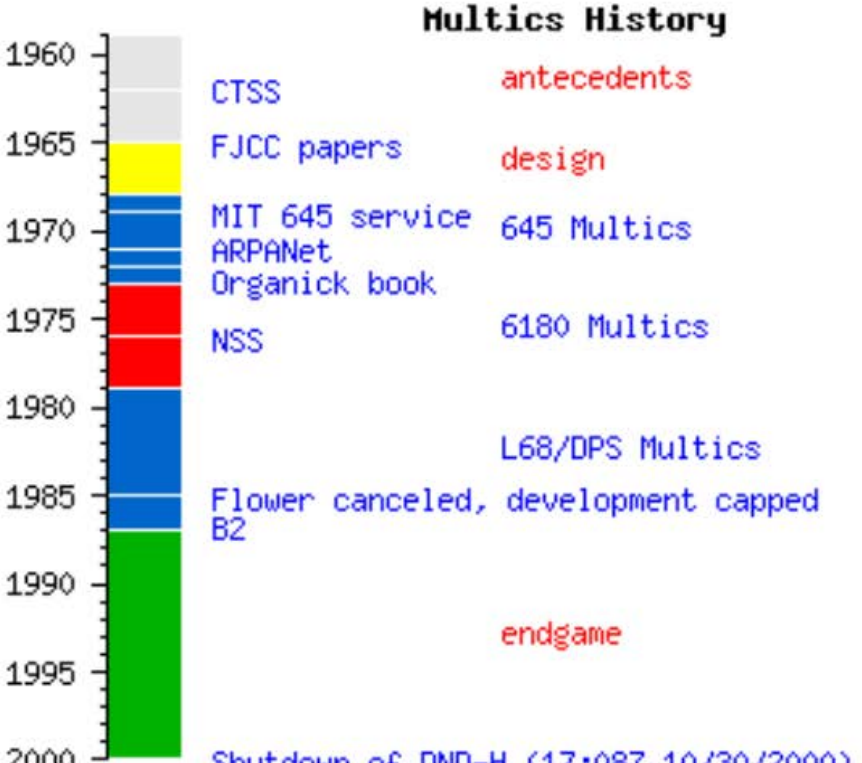


图: Multics OS–当今操作系统的鼻祖

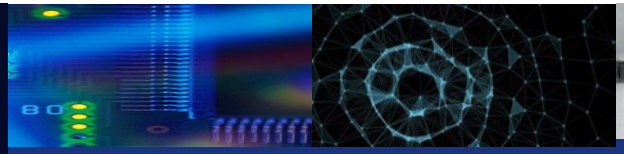
Multics OS

MULTICS (MULTiplexed Information and Computing System)，是 1964 年由 MIT, 贝尔实验室及美国通用电气公司所共同参与研发的，是一套安装在大型主机上多人多任务的操作系统

MULTICS 以 Compatible Time-Sharing System （CTSS）做基础，建置在美国通用电力公司的大型机 GE-645 。目的是连接 **1000 部终端机**，支持 **300 位用户**同时上线

PL/I　　分层文件目录　　分时调度　　二维虚拟内存

图: Multics OS 简介



兼容分时系统

CTSS

- One of the first time-sharing operating systems
- Developed at MIT by a group known as Project MAC
- Ran on a computer with 32,000 36-bit words of main memory, with the resident monitor consuming 5000 of that
- To simplify both the monitor and memory management a program was always loaded to start at the location of the 5000th word

Time Slicing

- System clock generates interrupts at a rate of approximately one every 0.2 seconds
- At each interrupt OS regained control and could assign processor to another user
- At regular time intervals the current user would be preempted and another user loaded in
- Old user programs and data were written out to disk
- Old user program code and data were restored in main memory when that program was next given a turn

CTSS

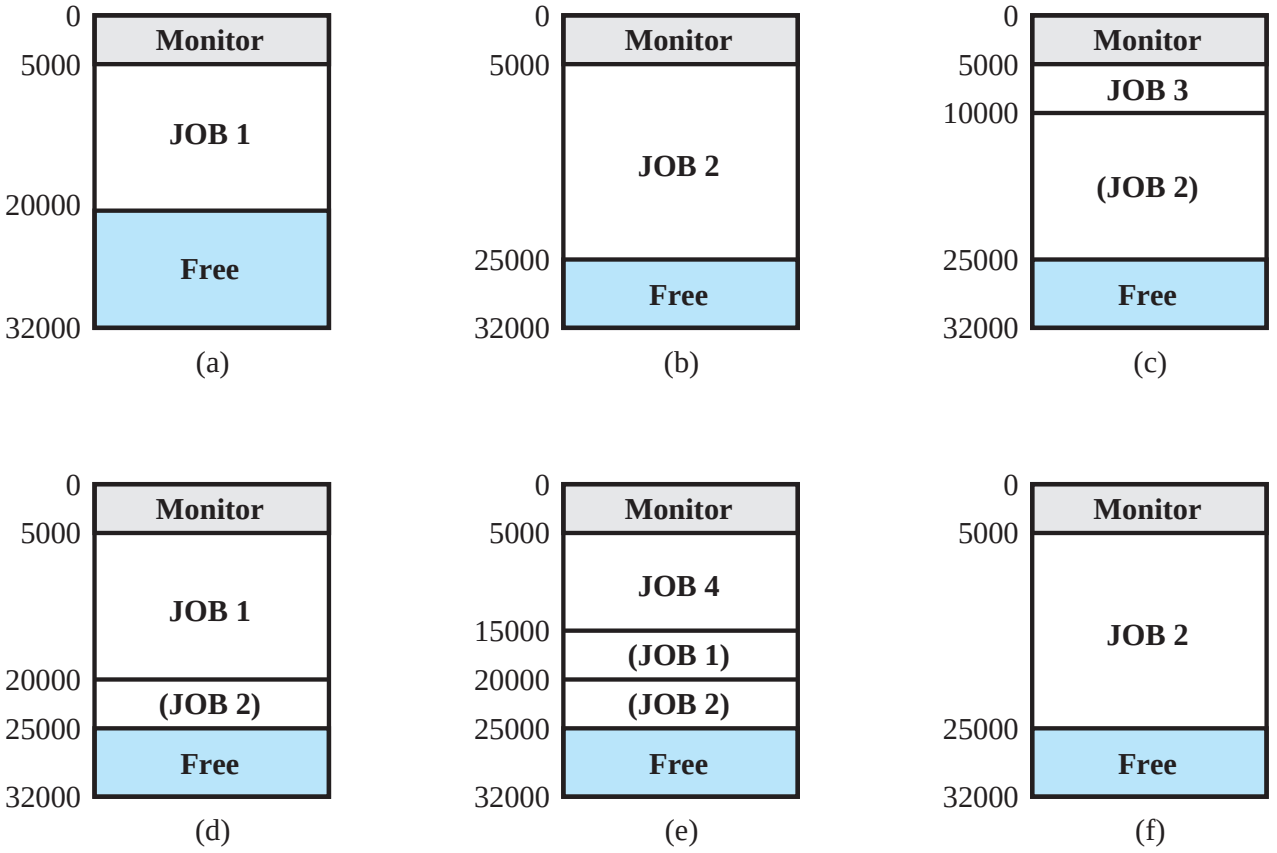
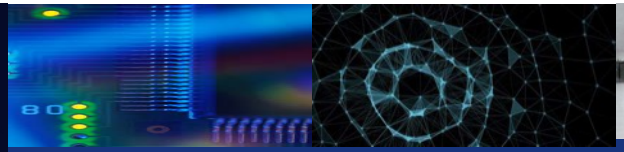
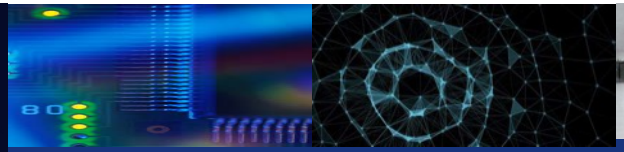


Figure 2.7 CTSS Operation



2. 操作系统服务



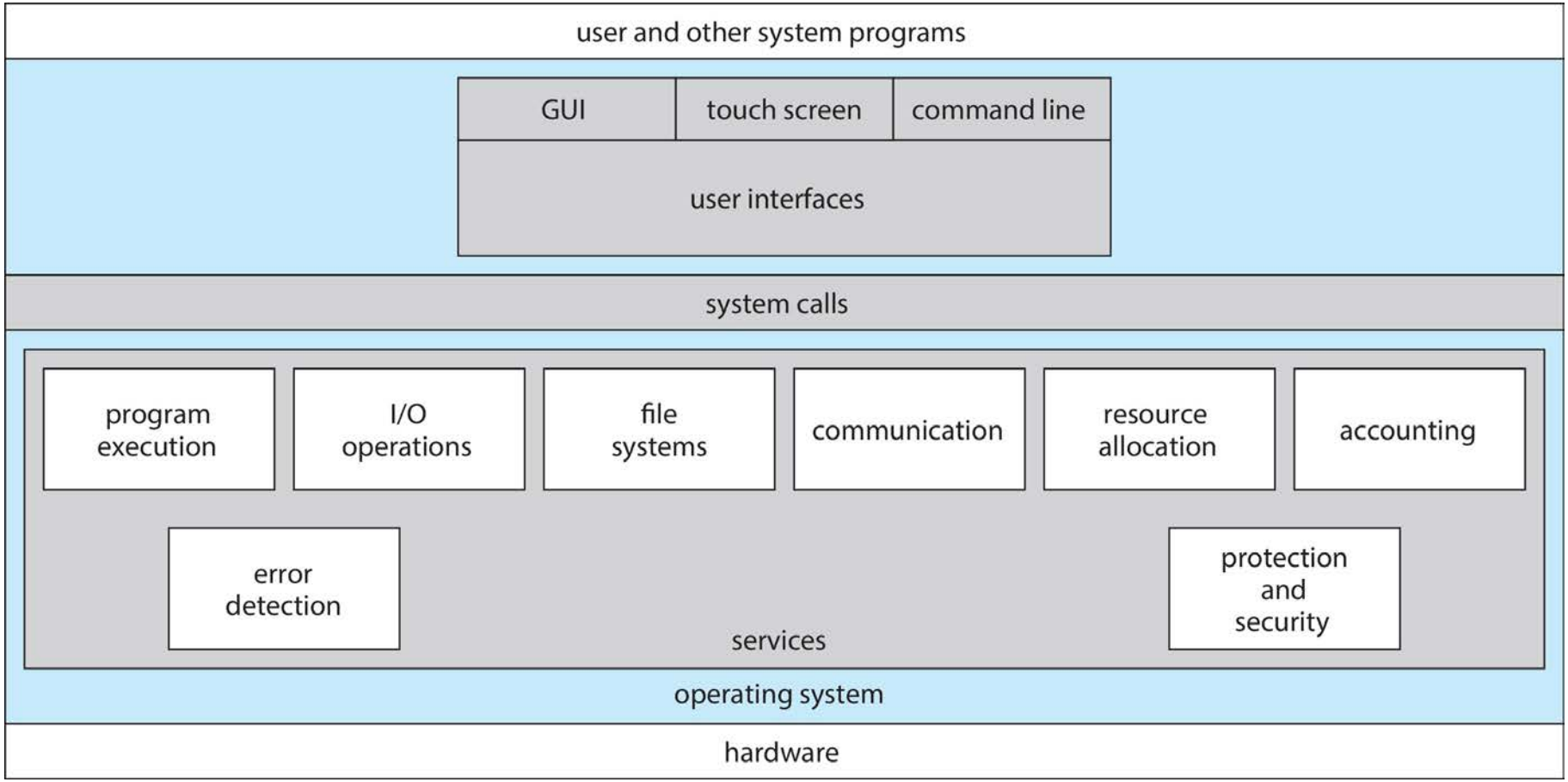
操作系统服务

- 操作新系统提供以下功能：
  - —— 程序执行的运行环境；
  - —— 程序和用户使用的各种服务；

- 从用户、程序员和操作系统设计人员三个角度看操作系统服务；
  - —— 系统提供的服务；
  - —— 用户和程序员采用的接口；
  - —— 系统组件及其相互关系；

操作系统服务

- 提供的服务随操作系统的不同而不同，但是也有共同点：



操作系统服务

- **一组操作系统服务提供有助于用户的功能：**
  - 用户界面-几乎所有操作系统都有**用户界面**（UI）。
    不同的命令行（CLI）、图形用户界面（GUI）、触摸屏、
    批处理（三者都具备）；
  - 程序执行-系统必须能够将**程序加载到内存中并运行**该程序，正常或异常结束执行（指示错误）；
  - **I/O操作**-正在运行的程序可能需要I/O，这可能涉及文件或I/O设备；
  - **文件系统操作**-用户对文件系统是特别感兴趣。程序需要读写文件和目录，创建和删除它们，搜索它们，列出文件信息，权限管理。

操作系统服务

- 一组操作系统服务提供有助于用户的功能（续）：
  - **通信** – 进程可以在同一台计算机上或通过网络在计算机之间交换信息；
    - 通信可以通过**共享内存或通过消息传递**（由操作系统移动的数据包）进行；
  - **错误检测** – 操作系统需要不断检测错误和更正错误；
    - 可能发生在CPU和内存硬件、I/O设备和用户程序；
    - 对于每种类型的错误，操作系统都应该采取适当的措施来确保正确和一致的计算；
    - 调试设施和工具可以大大提高用户和程序员有效使用系统的能力；

操作系统服务

- 另一组操作系统功能用于通过资源共享确保系统本身的高效运行
  - **资源分配**—当多个用户或多个作业同时运行时，必须为每个用户或作业分配资源
    - 许多类型的资源—CPU周期、主内存、文件存储、I/O设备、寄存器、Cache等；
  - **记账**—跟踪哪些用户使用了多少以及哪些类型的计算机资源；
  - **日志服务**—跟踪系统软硬件的行为状态；
  - **保护和安全**—存储在多用户或联网计算机系统中的信息的所有者可能希望控制该信息的使用，并发进程不应相互干扰
    - 保护包括确保对系统资源的所有访问都受到控制；
    - 外部系统的安全性要求用户身份验证，扩展到保护外部I/O设备免受无效访问尝试；

操作系统软件

- 操作系统本身作为一种软件、代码，跟其他软件工作的方式是一样的；
- 操作系统作为一个或者一组程序由处理器执行；
- 经常释放控制，必须依赖处理器才能重新获得控制；

作为资源管理器的操作系统

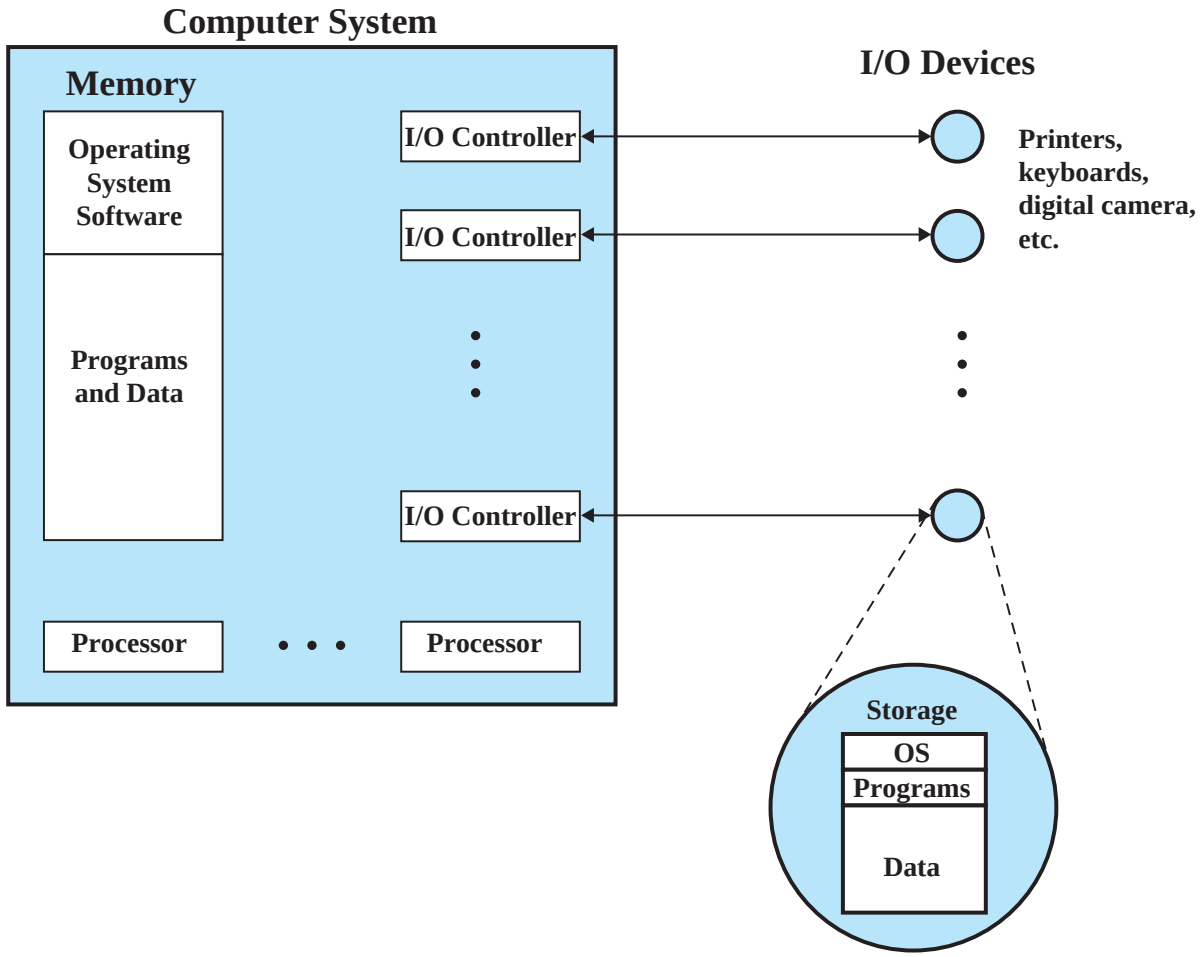


Figure 2.2 The Operating System as Resource Manager

用户操作系统的交互界面

- Shell – 命令行接口
- GUI – 图形用户接口
- Kernel–操作系统的内部

计算机硬件和软件结构

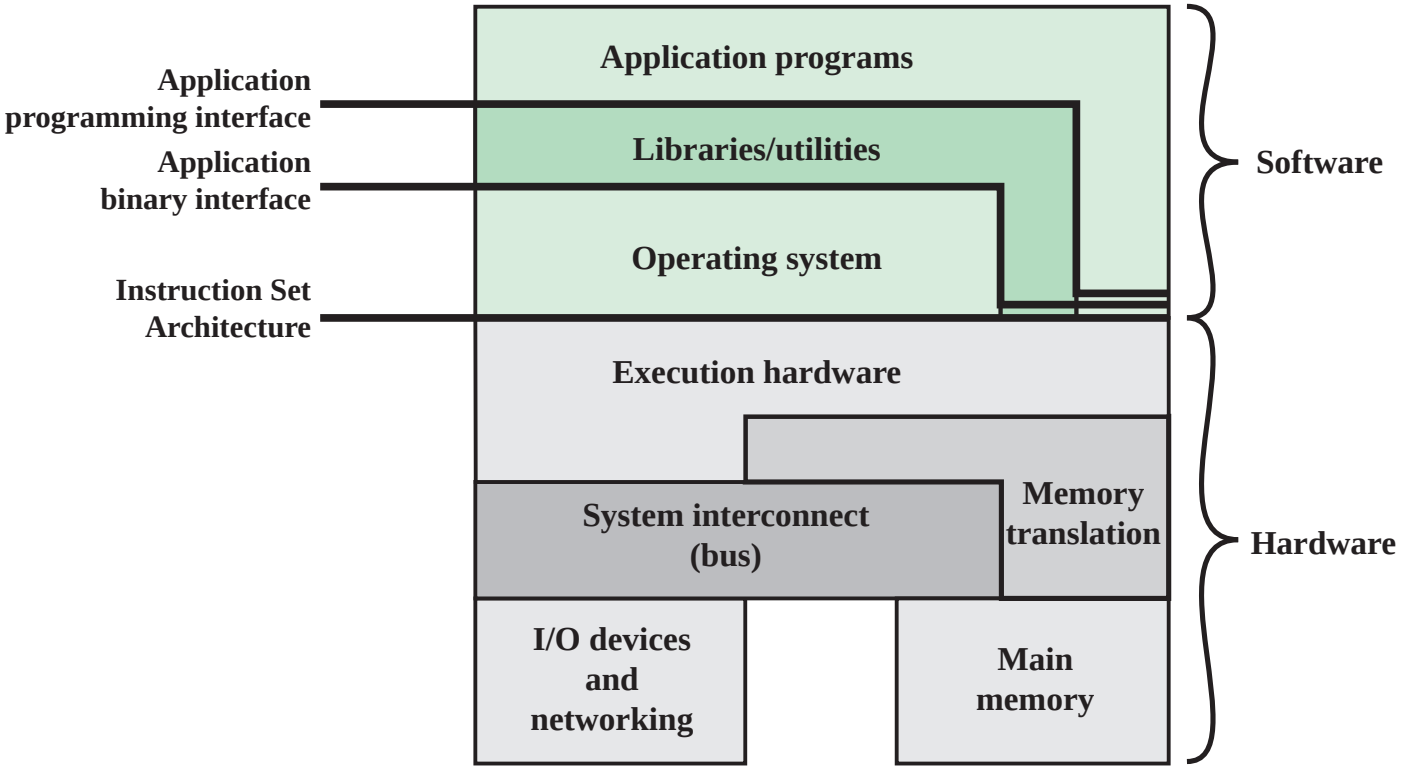
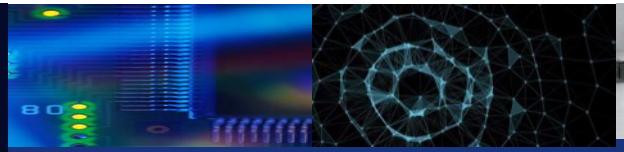


Figure 2.1 Computer Hardware and Software Structure

关键接口

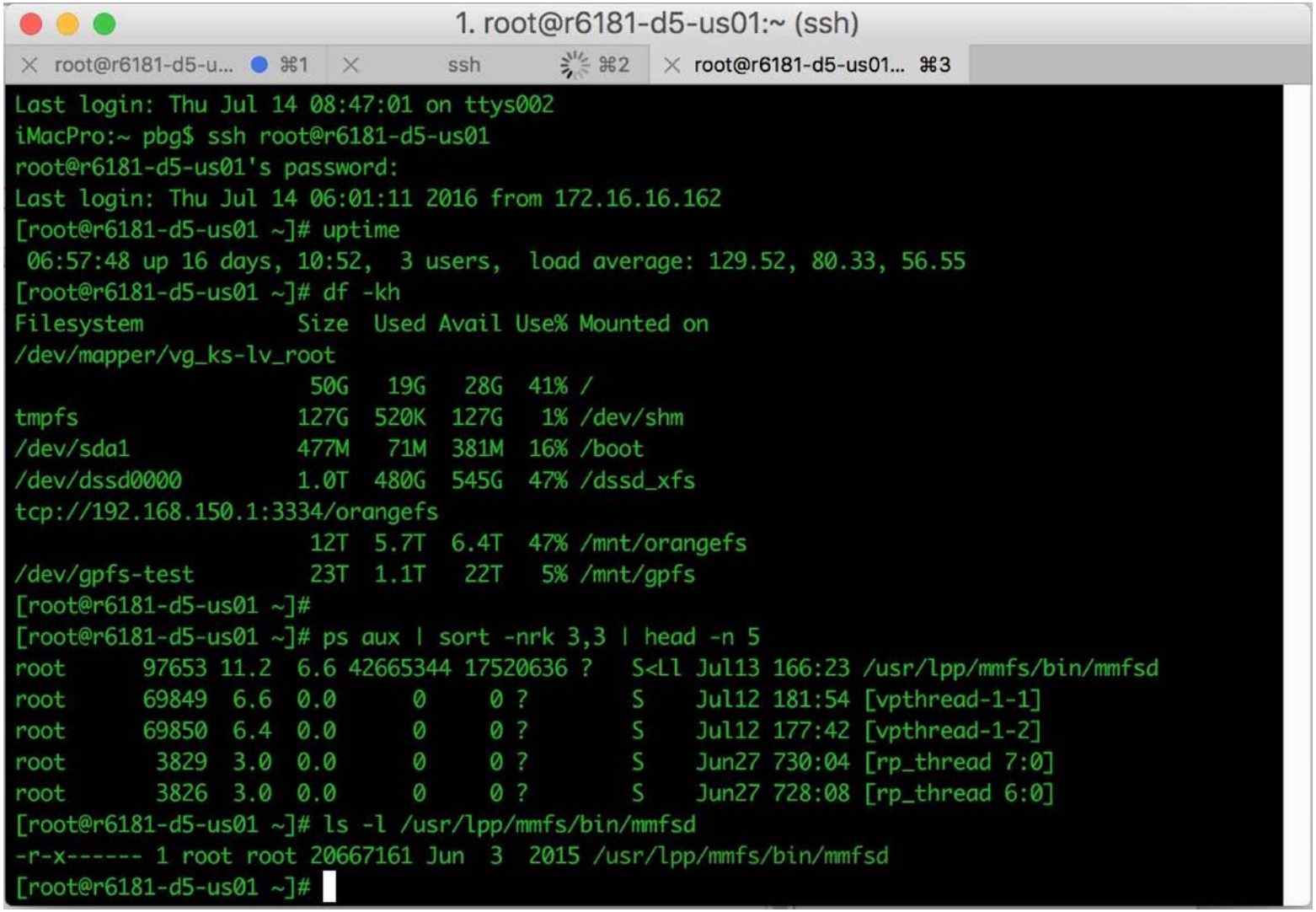
- Instruction set architecture (ISA)
- Application binary interface (ABI)
- Application programming interface (API)



命令解释程序

- CLI允许直接输入命令
- 有时在内核中实现，有时由系统程序实现
- 有时会实现多种风格 - shell
- 主要从用户获取命令并执行它
- 有时是内置命令，有时只是程序名
  - 如果是后者，则添加新功能不需要修改shell

Bourne shell



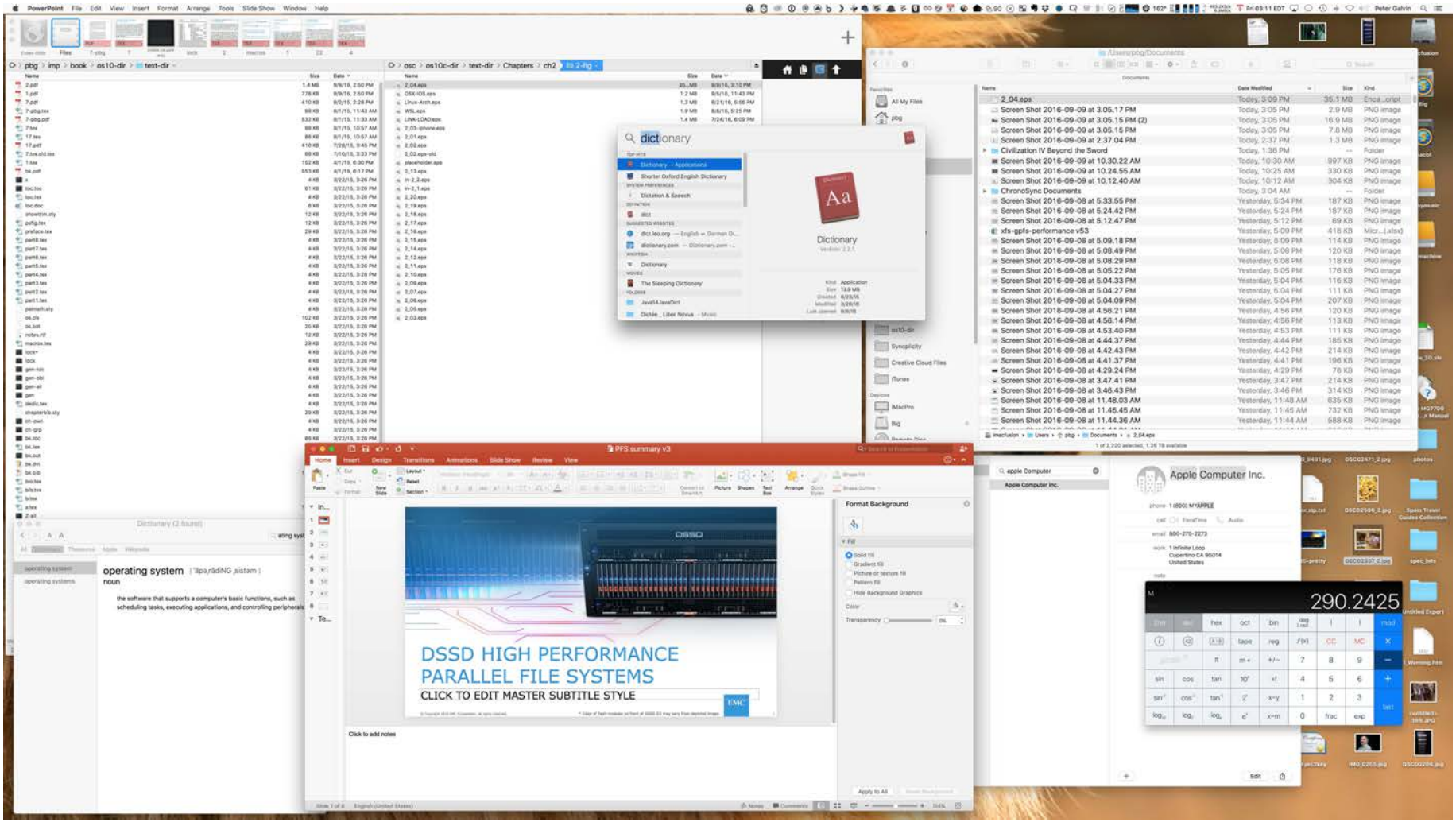
图形用户界面

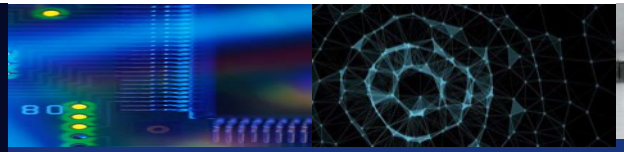
- 用户友好的桌面界面
  - 通常是鼠标、键盘和显示器
  - 图标代表文件、程序、动作等
  - 界面中对象上的各种鼠标按钮会导致各种操作（提供信息、选项、执行功能、打开目录（称为文件夹））
  - 发明于施乐帕克
- 许多系统现在同时包括CLI和GUI界面
  - Microsoft Windows是带有CLI“命令”shell的GUI
  - Apple Mac OS X是一个“Aqua”GUI界面，下面是UNIX内核，外壳可用
  - Unix和Linux具有CLI和可选GUI界面（CDE、KDE、GNOME）

MacOSXGUI

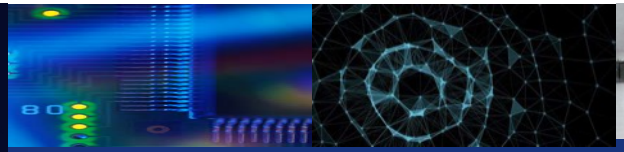
- 触摸屏设备需要新的接口
  - 鼠标不可能或不需要
  - 基于手势的动作和选择
  - 用于文本输入的虚拟键盘
- 语音命令

图形用户界面


dictionary
Dictionary
operating system
noun
DSSD HIGH PERFORMANCE
PARALLEL FILE SYSTEMS
CLICK TO EDIT MASTER SUBTITLE STYLE
Apple Computer Inc.
290.2425



3. 系统调用



系统调用

- 系统调用提供用户程序与操作系统之间的接口
- 通常用高级语言（C或C++）编写，可以通过汇编直接访问硬件；
- 主要由程序通过高级应用程序编程接口（API）访问，而不是直接使用系统调用；
- 三种最常见的API是用于Windows的Win32 API、用于基于POSIX的系统的POSIX API（包括几乎所有版本的UNIX、Linux和Mac OS X）以及用于Java虚拟机（JVM）的Java API；

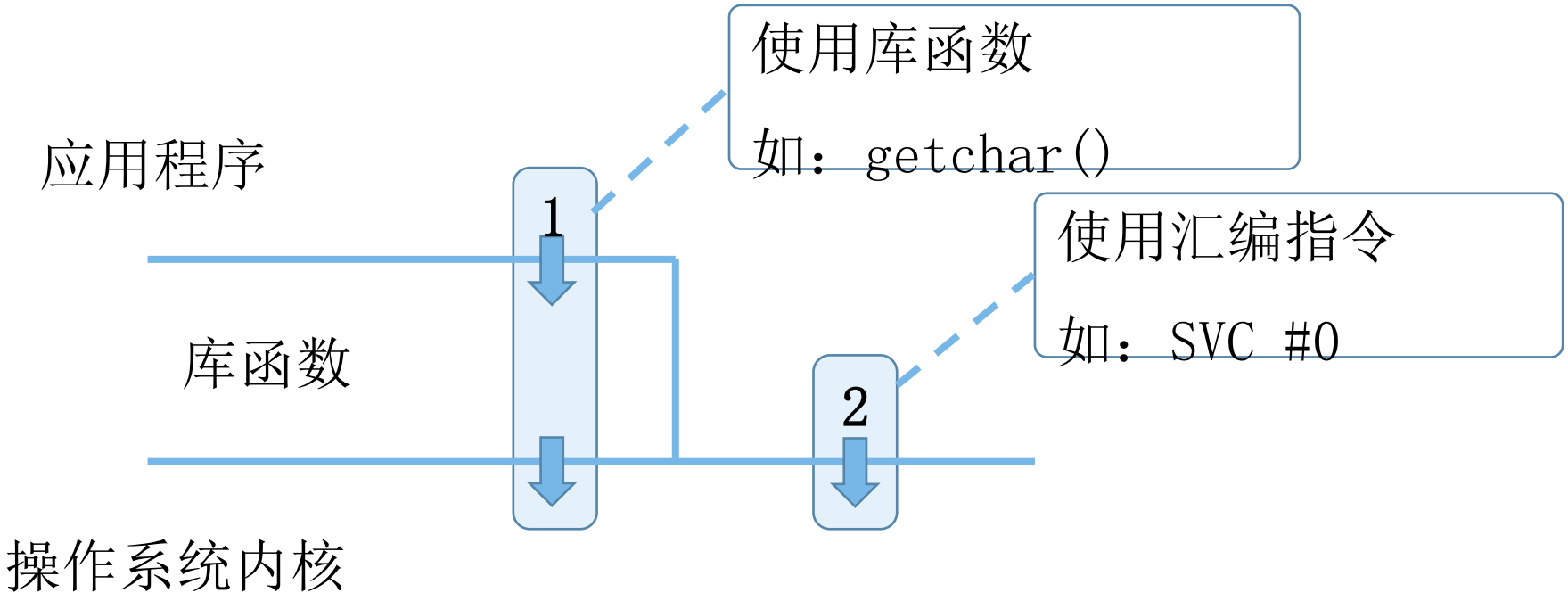
系统调用

- 将一个文件的内容复制到另一个文件的系统调用序列

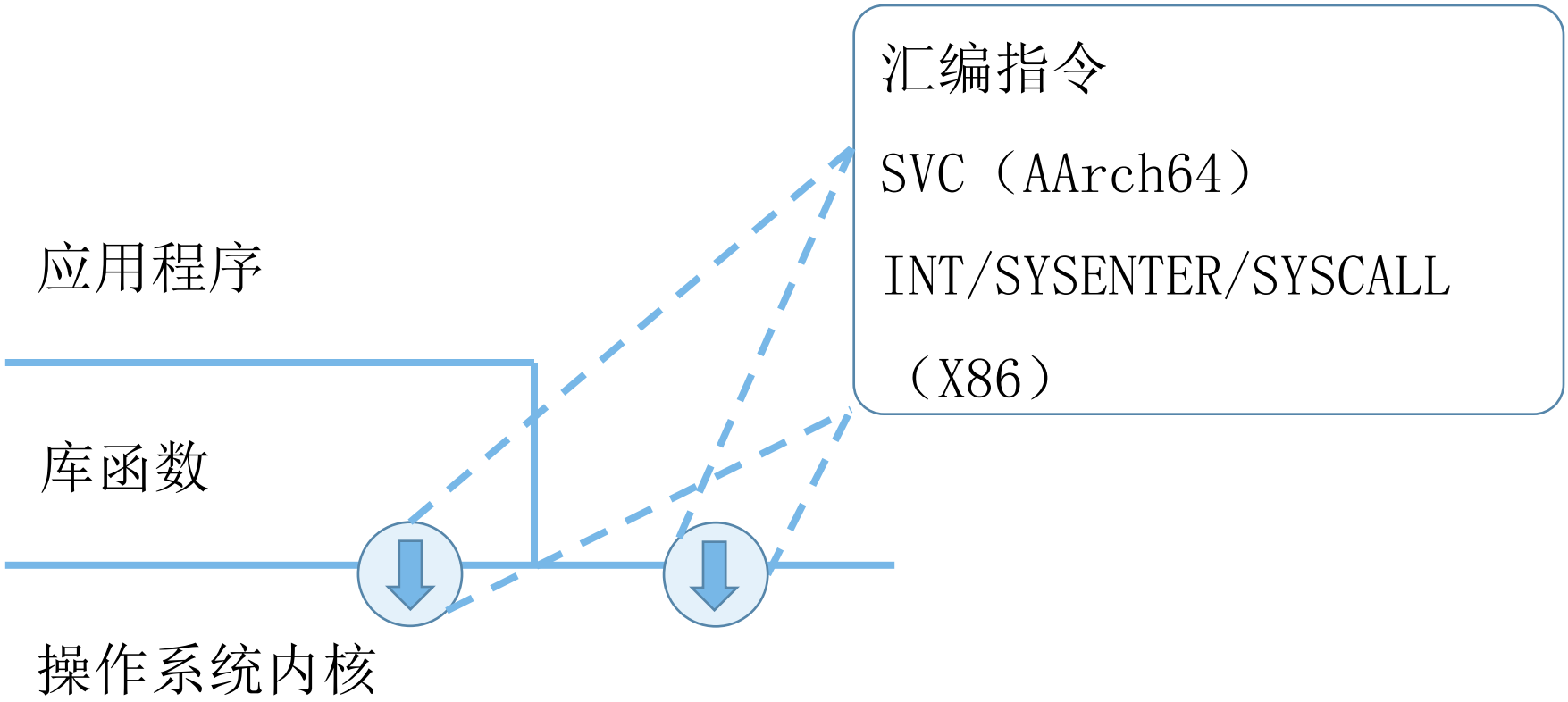
Example System Call Sequence

```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

系统调用-程序员视角



系统调用-硬件视角



标准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 into which the data will be read
- `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.

系统调用样例程序

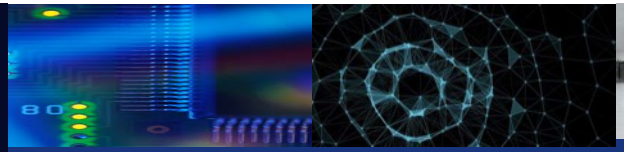
```c
#include<fcntl.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/stat.h>

int main()
{
   int n, fd_a, fd_b;

   char buf[30];

   fd_a = open("test

   n=read(fd_a,buf,2

   fd_b=open("target

   write(fd_b,buf,n)

}
```

```
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=146797, ...}) = 0
mmap(NULL, 146797, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f1d4024e000
close(3)                                = 0
openat(AT_FDCWD, "/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\0\260A\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1824496, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f1d4024c000
mmap(NULL, 1837056, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f1d4008b000
mprotect(0x7f1d400ad000, 1658880, PROT_NONE) = 0
mmap(0x7f1d400ad000, 1343488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x22000) = 0x
7f1d400ad000
mmap(0x7f1d401f5000, 311296, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x16a000) = 0x7f1d401f50
00
mmap(0x7f1d40242000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b6000) = 0x
7f1d40242000
mmap(0x7f1d40248000, 14336, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f1d40
248000
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7f1d4024d500) = 0
mprotect(0x7f1d40242000, 16384, PROT_READ) = 0
mprotect(0x55749e49e000, 4096, PROT_READ) = 0
mprotect(0x7f1d40299000, 4096, PROT_READ) = 0
munmap(0x7f1d4024e000, 146797)          = 0
openat(AT_FDCWD, "test.txt", O_RDONLY)  = 3
read(3, "hello\n", 20)                  = 6
openat(AT_FDCWD, "target", O_WRONLY|O_CREAT, 0642) = 4
write(4, "hello\n", 6)                  = 6
exit_group(0)                           = ?
+++ exited with 0 +++
```

Linux追踪系统调用

- 每当有系统调用产生时，Linux可打印发生的系统调用、系统调用的参数和系统调用的返回值

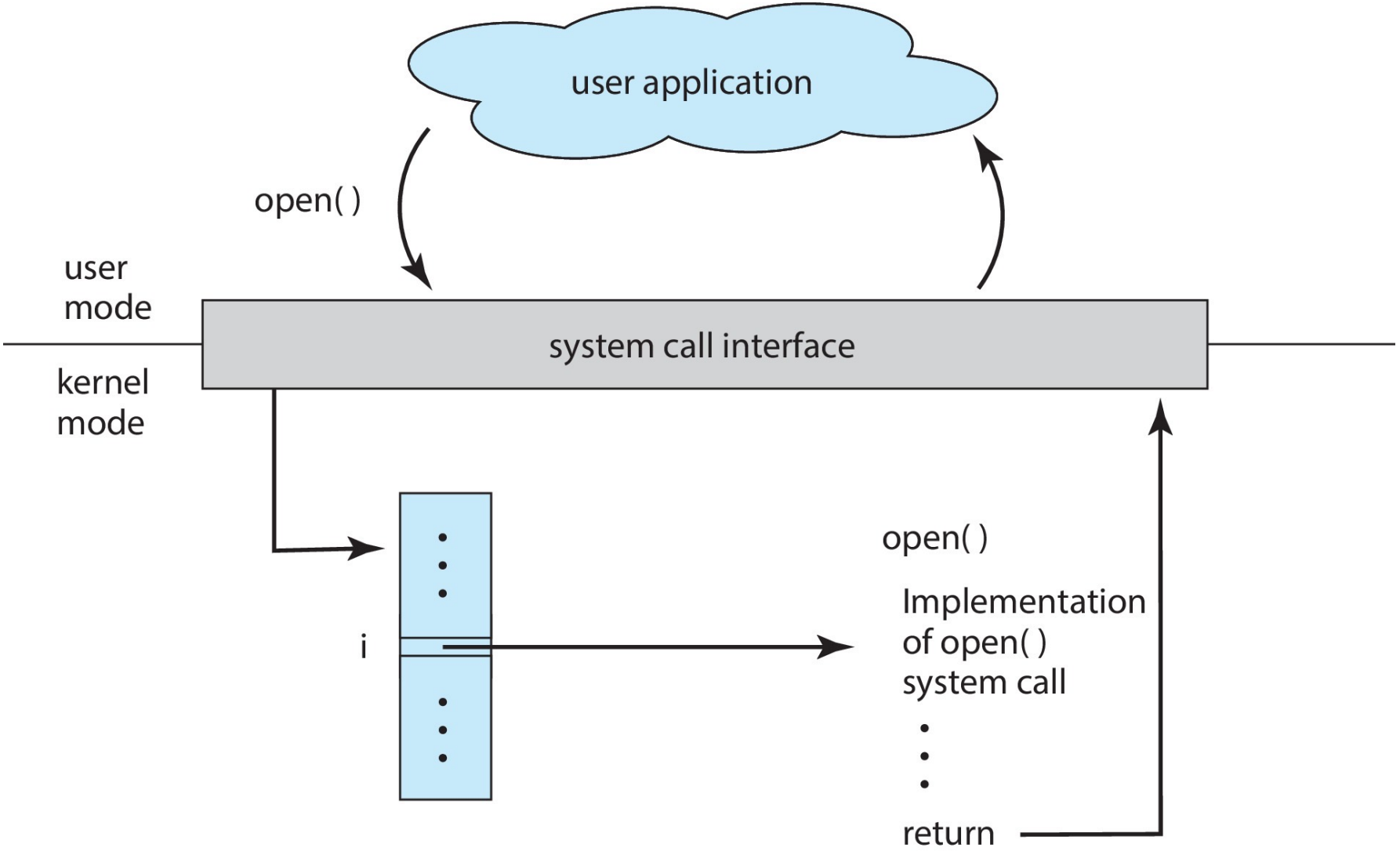
- ptrace()可追踪Linux中的系统调用情况
  - 广泛应用在各种debugger中
- 命令行中
  - strace追踪系统调用
  - ltrace追踪库函数的调用



系统调用实现

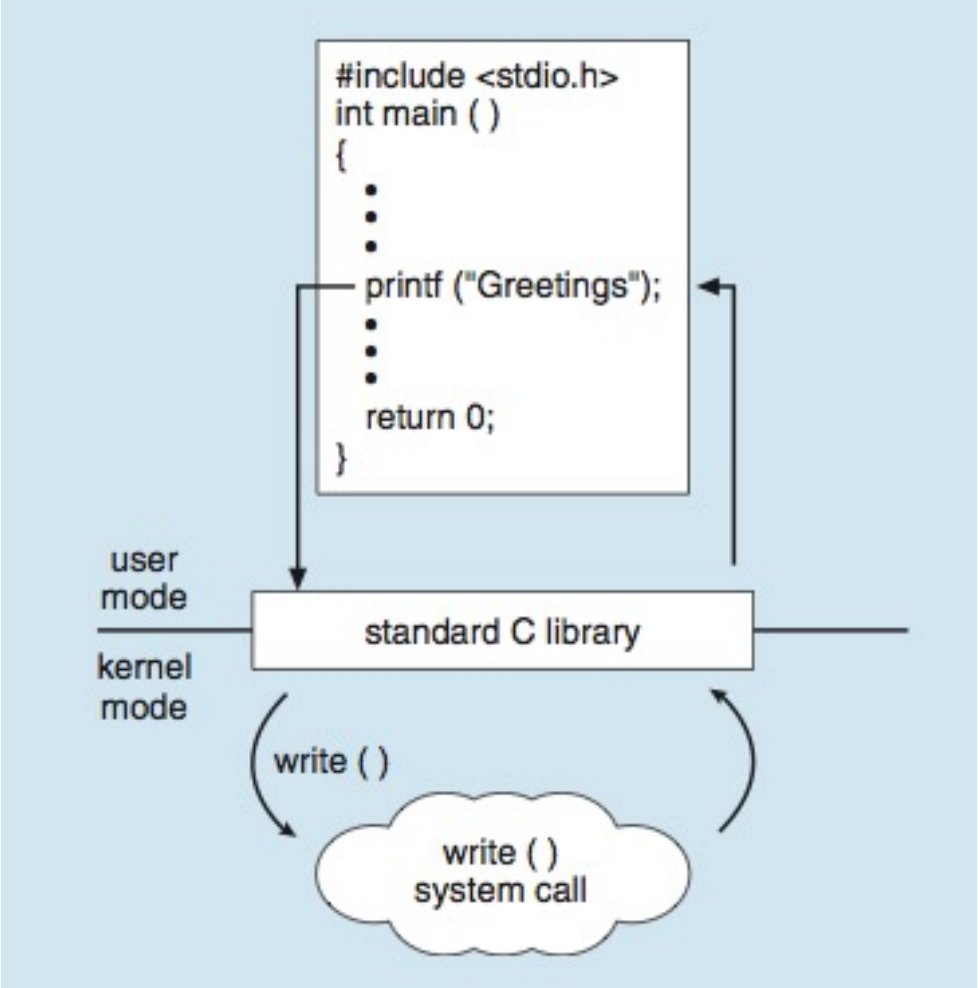
- 通常，一个号码与每个系统调用相关联
  - 系统调用接口根据这些数字维护索引表
- 系统调用接口调用操作系统内核中的预置系统调用，并返回系统调用的状态和任何返回值；
- 调用者不需要知道系统调用是如何实现的
  - 只需要遵守API并了解操作系统作为结果调用将做什么；
  - API对程序员隐藏的操作系统接口的大部分细节
    - 由运行时支持库（内置于编译器附带的库中的函数集）管理

系统调用-操作系统关系


user application
open( )
user mode
system call interface
kernel mode
i
open( )
Implementation of open( ) system call
return

系统调用-操作系统关系

使用标准C库函数与系统调用的区别



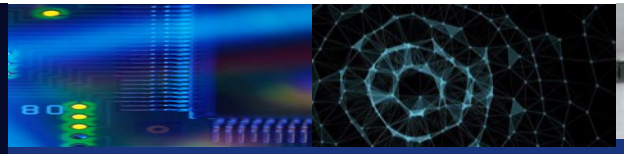
Demo

系统调用参数传递

- **通常系统调用需要更多的信息，而不仅仅是系统调用的标识**
  - 参数的确切类型和数量因操作系统和调用而异；

- **用于向操作系统传递参数的三种通用方法**
  - **最简单的方法：**在寄存器中传递参数
    - 在某些情况下，参数可能比寄存器多
  - **内存中存储在块或表中的参数**，以及作为参数在寄存器中传递的块地址
    - Linux和Solaris采用的这种方法
  - **由程序放置或推送到堆栈上并由操作系统弹出堆栈的参数**

    块和堆栈方法不限制所传递参数的数量或长度



系统调用参数传递

- 在寄存器中传递参数

```
 1 SECTION .data            ; data section
 2 msg:    db "Hello World",10     ; the string to print, 10=cr
 3 len:    equ $-msg               ; "$" means "here"
 4                                 ; len is a value, not an address
 5
 6         SECTION .text           ; code section
 7         global main             ; make label available to linker
 8 main:                           ; standard  gcc  entry point
 9
10         mov     edx,len         ; arg3, length of string to print
11         mov     ecx,msg         ; arg2, pointer to string
12         mov     ebx,1           ; arg1, where to write, screen
13         mov     eax,4           ; write sysout command to int 80 hex
14         int     0x80            ; interrupt 80 hex, call kernel
15
16         mov     ebx,0           ; exit code, 0=normal
17         mov     eax,1           ; exit command to kernel
18         int     0x80            ; interrupt 80 hex, call kernel
```

系统调用参数传递

- 由程序放置或推送到堆栈上并由操作系统弹出堆栈的参数

```
1  mov ah, 0x0e ; tty mode
2
3  mov bp, 0x8000 ; this is an address far away from 0x7c00 so that we don't ge
   t overwritten
4  mov sp, bp ; if the stack is empty then sp points to bp
5
6  push 'A'
7  push 'B'
8  push 'C'
9
10 ; to show how the stack grows downwards
11 mov al, [0x7ffe] ; 0x8000 - 2
12 int 0x10
13
14 ; however, don't try to access [0x8000] now, because it won't work
15 ; you can only access the stack top so, at this point, only 0x7ffe (look abo
   ve)
16 mov al, [0x8000]
17 int 0x10
18
19
20 ; recover our characters using the standard procedure: 'pop'
21 ; We can only pop full words so we need an auxiliary register to manipulate
```

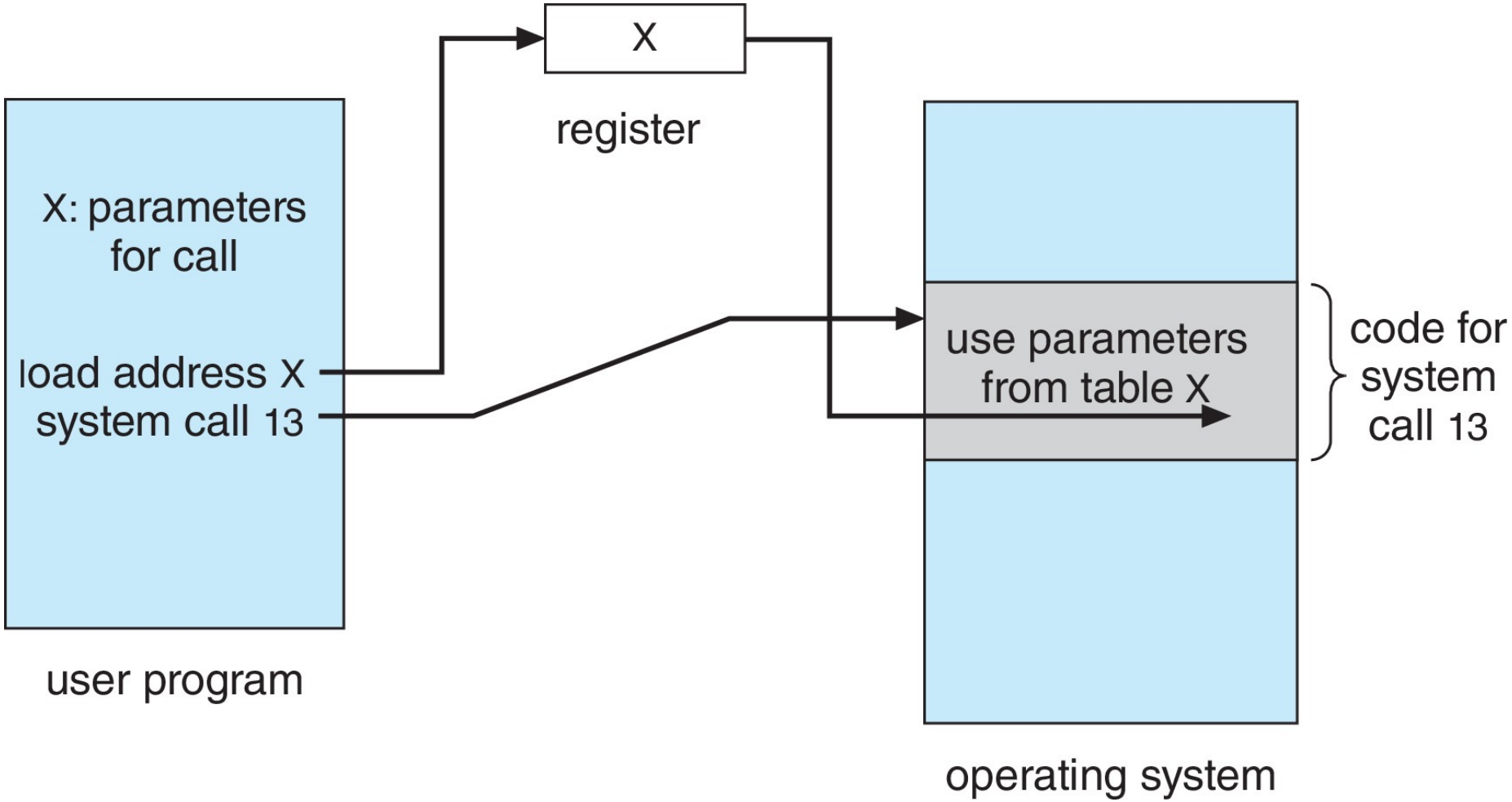
系统调用参数传递

- 内存中存储在块或表中的参数

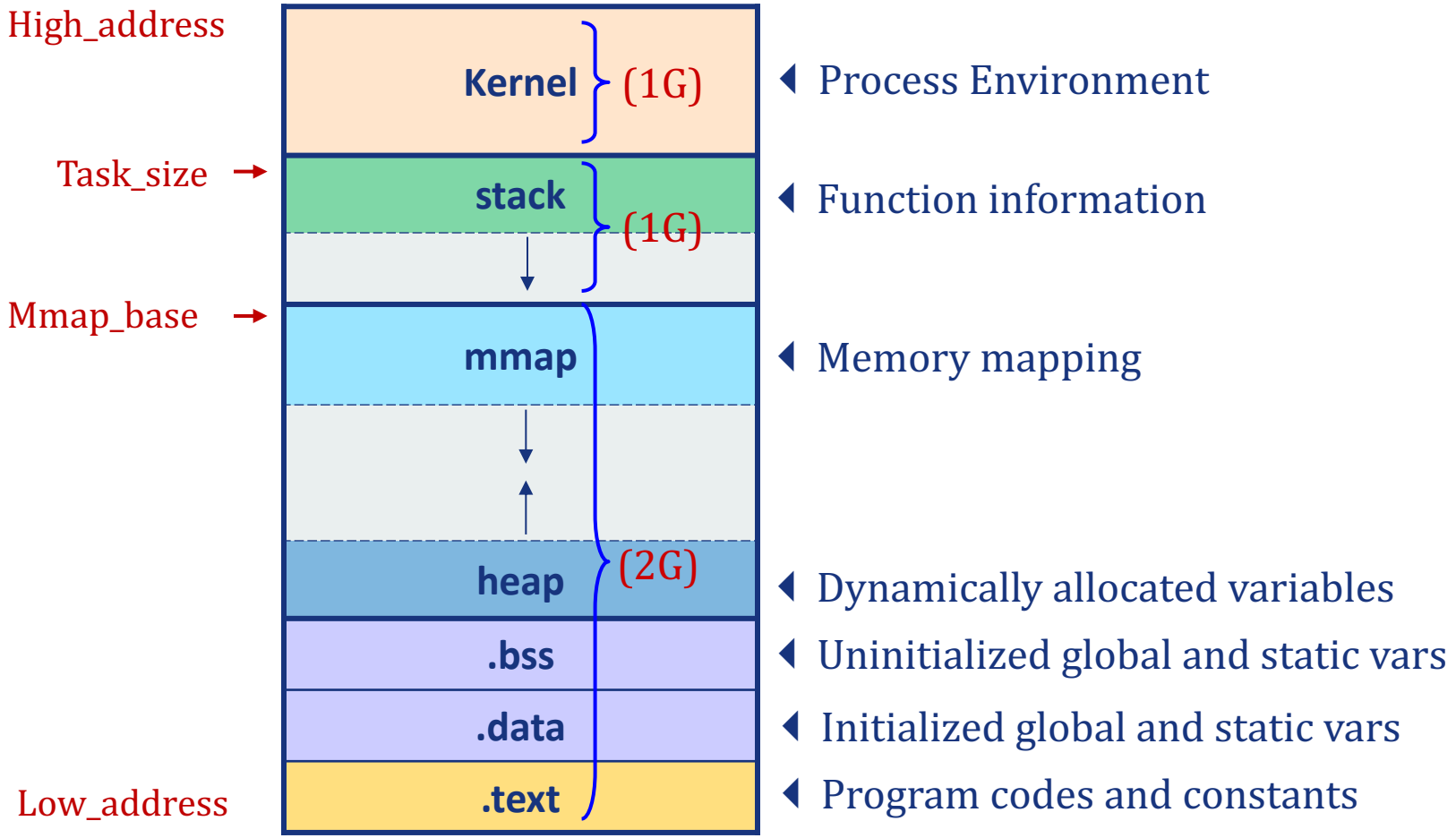
```
; Want to call a function “myFunc” that takes three
; integer parameters. First parameter is in EAX.
; Second parameter is the constant 123. Third
; parameter is in memory location “var”

   push  [var]  ; Push last parameter first
   push  123
   push  eax    ; Push first parameter last

   call _myFunc ; Call the function (assume C naming)

; On return, clean up the stack. We have 12 bytes
; (3 parameters * 4 bytes each) on the stack, and the
; stack grows down. Thus, to get rid of the parameters,
; we can simply add 12 to the stack pointer

    add   esp, 12

; The result produced by “myFunc” is now available for
; use in the register EAX. No other register values
; have changed
```

```
.486
MODEL FLAT
.CODE
PUBLIC _myFunc
_myFunc PROC
   ; *** Standard subroutine prologue ***
   push ebp     ; Save the old base pointer value.
   mov ebp, esp ; Set the new base pointer value.
   sub esp, 4   ; Make room for one 4-byte local variable.
   push edi     ; Save the values of registers that the function
   push esi     ; will modify. This function uses EDI and ESI.
                ; (no need to save EAX, EBP, or ESP)

   ; *** Subroutine Body ***
   mov eax, [ebp+8] ; Put value of parameter 1 into EAX
   mov esi, [ebp+12]; Put value of parameter 2 into ESI
   mov edi, [ebp+16]; Put value of parameter 3 into EDI

   mov [ebp-4], edi ; Put EDI into the local variable
   add [ebp-4], esi ; Add ESI into the local variable
   add eax, [ebp-4] ; Add the contents of the local variable
                    ; into EAX (final result)

   ; *** Standard subroutine epilogue ***
   pop esi      ; Recover register values
   pop edi
   mov esp, ebp ; Deallocate local variables
   pop ebp      ; Restore the caller’s base pointer value
   ret
ENDP _myFunc
END
```

系统调用参数传递

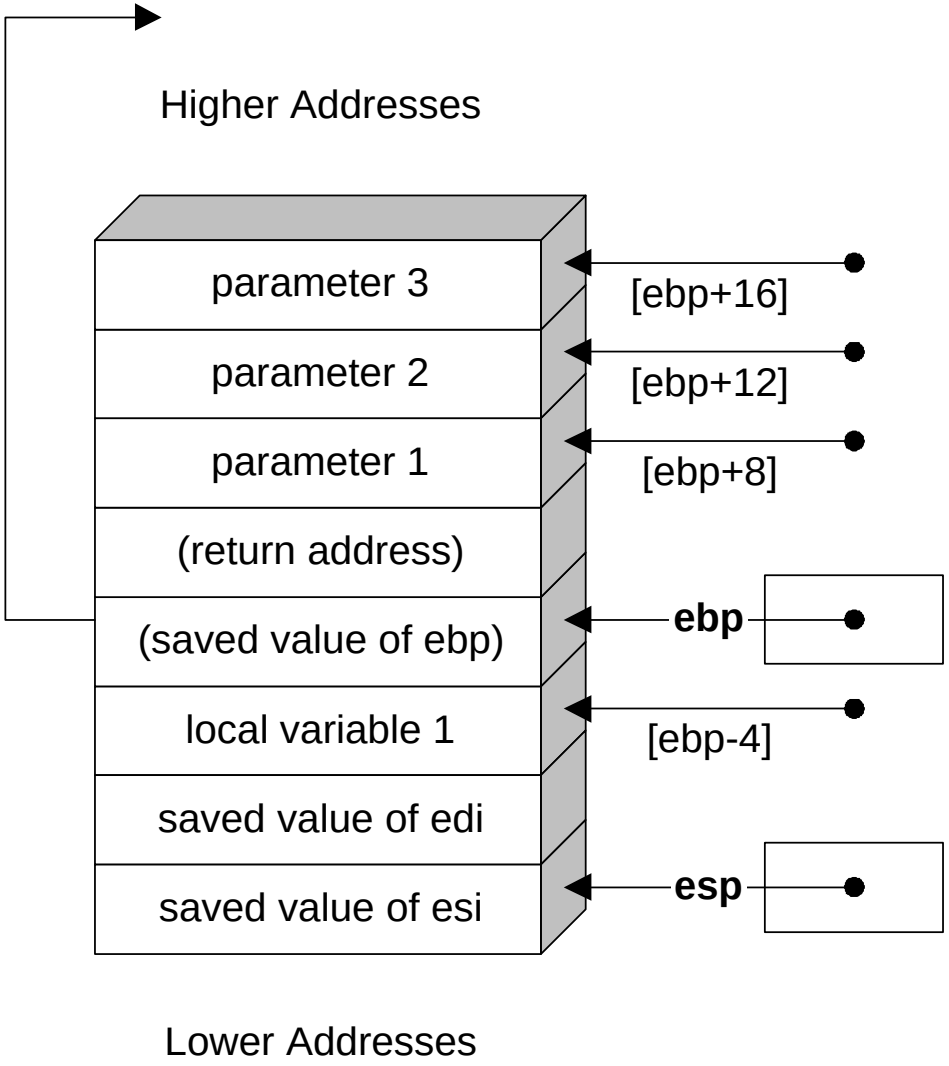

X
register
X: parameters for call
load address X
system call 13
user program
use parameters from table X
code for system call 13
operating system

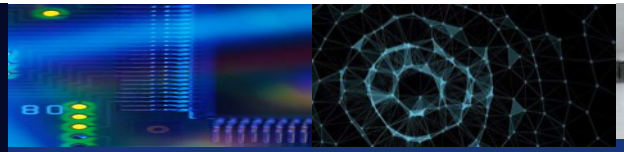
系统调用参数传递



Process Virtual Memory (PVM)

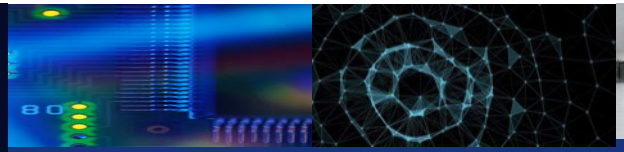
系统调用参数传递


Higher Addresses
parameter 3
parameter 2
parameter 1
(return address)
(saved value of ebp)
local variable 1
saved value of edi
saved value of esi
[ebp+16]
[ebp+12]
[ebp+8]
ebp
[ebp-4]
esp
Lower Addresses



系统调用的类型

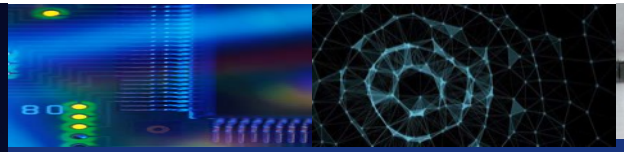
- 进程控制
- 文件管理
- 设备管理
- 信息维护
- 通信
- 安全保护



系统调用的类型

Process control

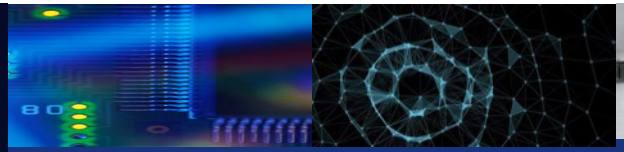
- `end(), abort()`
  - Halt a running program normally or abnormally
  - Transfer control to invoking command interpreter
  - Memory dump & & error message
    - Written to disk and examined by debugger
    - Respond to error: alert window (GUI system) or terminate the entire job (batch system)
  - Error level: normal termination (level 0)




系统调用的类型

Process control

- `end(), abort()`
- `load(), execute()`
  - Where to return?
    - Return to existing program: save mem. image
    - Both programs continue concurrently: multiprogram
- `create_process(), terminate_process()`
- `get_process_attributes(), set_process_attributes()`
  - Job's priority, maximum allowable execution time, etc



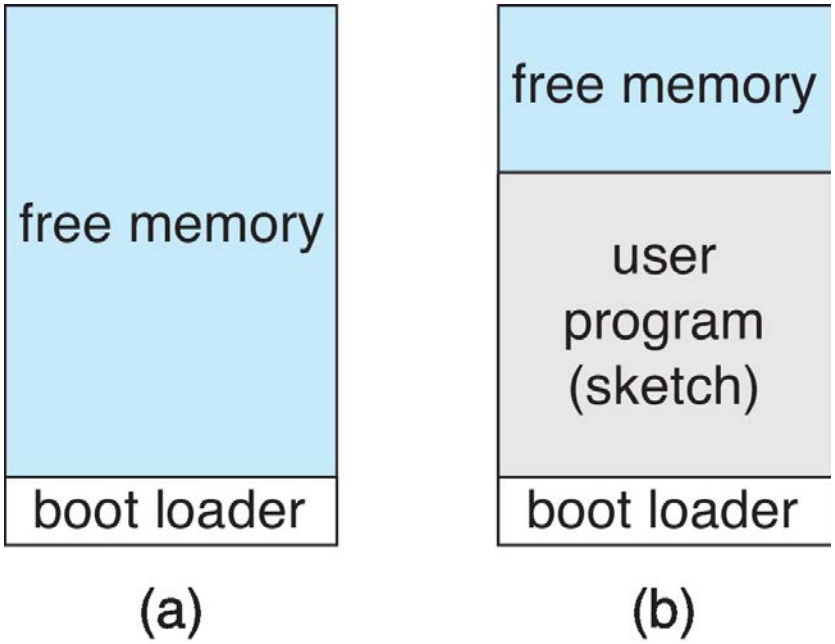
系统调用的类型

Process control

- end(), abort()
- load(), execute()
- create_process(), terminate_process()
- get_process_attributes(), set_process_attributes()
- wait_time()
- wait_event(), signal_event()
- acquire_lock(), release_lock()

MS-DOS/ Arduino进程控制案例

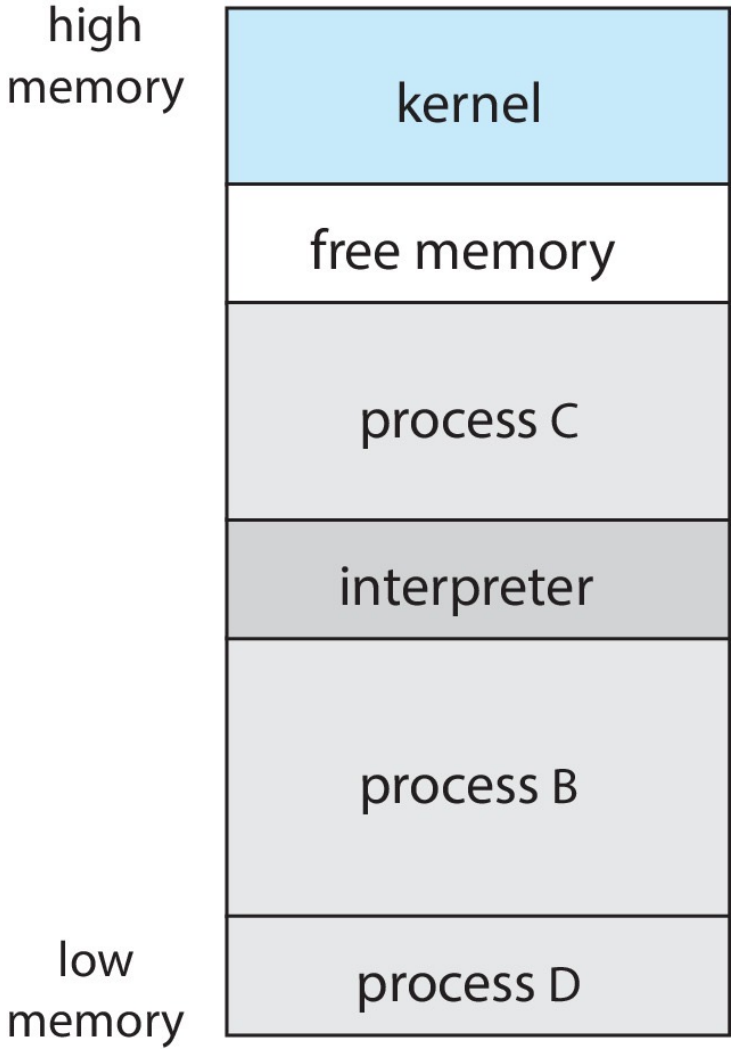
- 单一任务
- 无操作系统
- 通过USB加载到闪存中的程序（草图）
- 单个内存空间
- 引导加载程序加载程序
- 程序退出->外壳重新加载

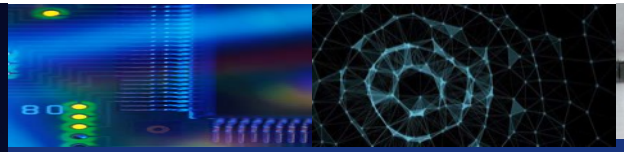


在系统启动时运行一个程序

示例：FreeBSD

- Unix变体
- 多任务
- 用户登录->调用用户选择的shell
- Shell执行fork（）系统调用以创建进程
  - 执行exec（）将程序加载到进程中
  - Shell等待进程终止或继续使用用户命令
- 进程退出时带有：
  - 代码=0 - 无错误
  - 代码>0 - 错误代码



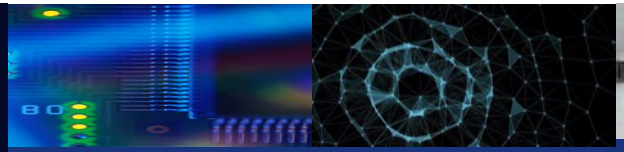


系统调用的类型

- 文件管理
  - 创建文件，删除文件
  - 打开、关闭文件
  - 读、写、重新定位
  - 获取并设置文件属性
- 设备管理
  - 请求设备，释放设备
  - 读、写、重新定位
  - 获取设备属性，设置设备属性
  - 逻辑连接或分离设备

系统调用的类型

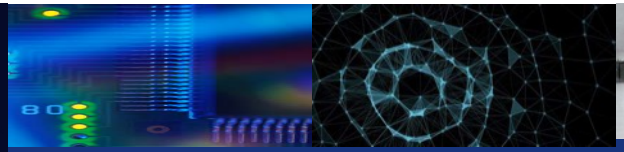
- 信息维护
  - 获取时间或日期，设置时间或日期
  - 获取系统数据，设置系统数据
  - 获取和设置进程、文件或设备属性
- 通讯
  - 创建、删除通信连接
  - 发送、接收消息（如果消息将模型传递给主机名或进程名）
    - 从客户端到服务器
  - 共享内存模型创建并访问内存区域
  - 传输状态信息
  - 连接和分离远程设备



系统调用的类型

Communications

- Message-passing model
  - Host name, IP, process name
  - `Get_hostid(), get_processid(), open_connection(), close_connection(), accept_connection(), read_message(), write_message()`
  - Useful for exchanging smaller amounts of data
- Shared-memory model
  - Remove the normal restriction of preventing one process from accessing another process's memory
  - Create and gain access to shared mem. region
    - `shared_memory_create(), shared_memory_attach()`
  - Threads: memory is shared by default
  - Efficient and convenient, having protection and synchronization issues




系统调用的类型

Protection

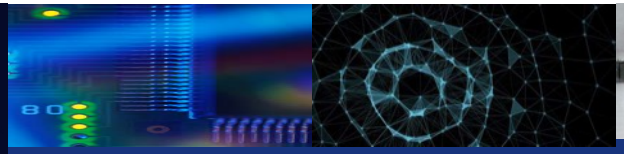
- Control access to resources
- All computer systems must be concerned
- Permission setting
  - `get_permission(), set_permission()`
- Allow/deny access to certain resources
  - `allow_user(), deny_user()`

Windows和Unix系统调用示例

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

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

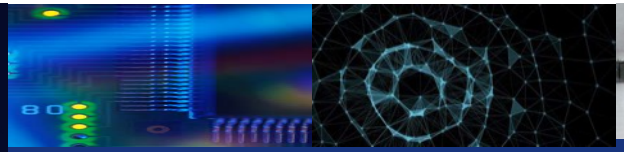
	Windows	Unix
Process control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File management	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device management	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`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



如何添加新的系统调用

- 下载Linux内核(https://www.kernel.org/)；
- 定位到syscall_32.tbl或者syscall_64.tbl（/arch/x86/entry/syscalls)；
- 添加新的系统调用号以及系统调用名称；
- 在syscall.h(/include/linux)中声明系统调用程序原型；
- 在sys.c（/kernel/sys.c）中添加相应的服务；
- 安装相关依赖并编译内核；
- 启动新内核；
- 测试新的系统调用；

Demo



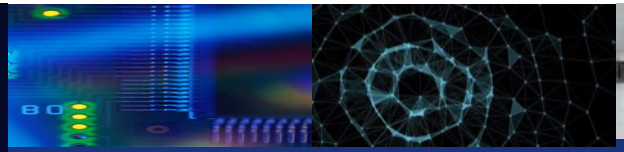
系统服务

- 系统程序为程序开发和执行提供了方便的环境。它们可分为：
  - 文件操作
  - 状态信息有时存储在文件中
  - 编程语言支持
  - 程序加载和执行
  - 通讯
  - 后台服务
  - 应用程序
- 大多数用户对操作系统的理解是由系统程序定义的，而不是实际的系统调用

系统服务

- 为程序开发和执行提供方便的环境
  - 其中一些只是系统调用的用户界面；其他的则要复杂得多
- 文件管理-创建、删除、复制、重命名、打印、转储、列出和一般操作文件和目录
- 状态信息
  - 有些人向系统询问信息——日期、时间、可用内存量、磁盘空间、用户数量
  - 其他的则提供详细的性能、日志记录和调试信息
  - 通常，这些程序格式化输出并将其打印到终端或其他输出设备
  - 有些系统实现了一个注册表，用于存储和检索配置信息

/proc 下的系统信息；




系统服务

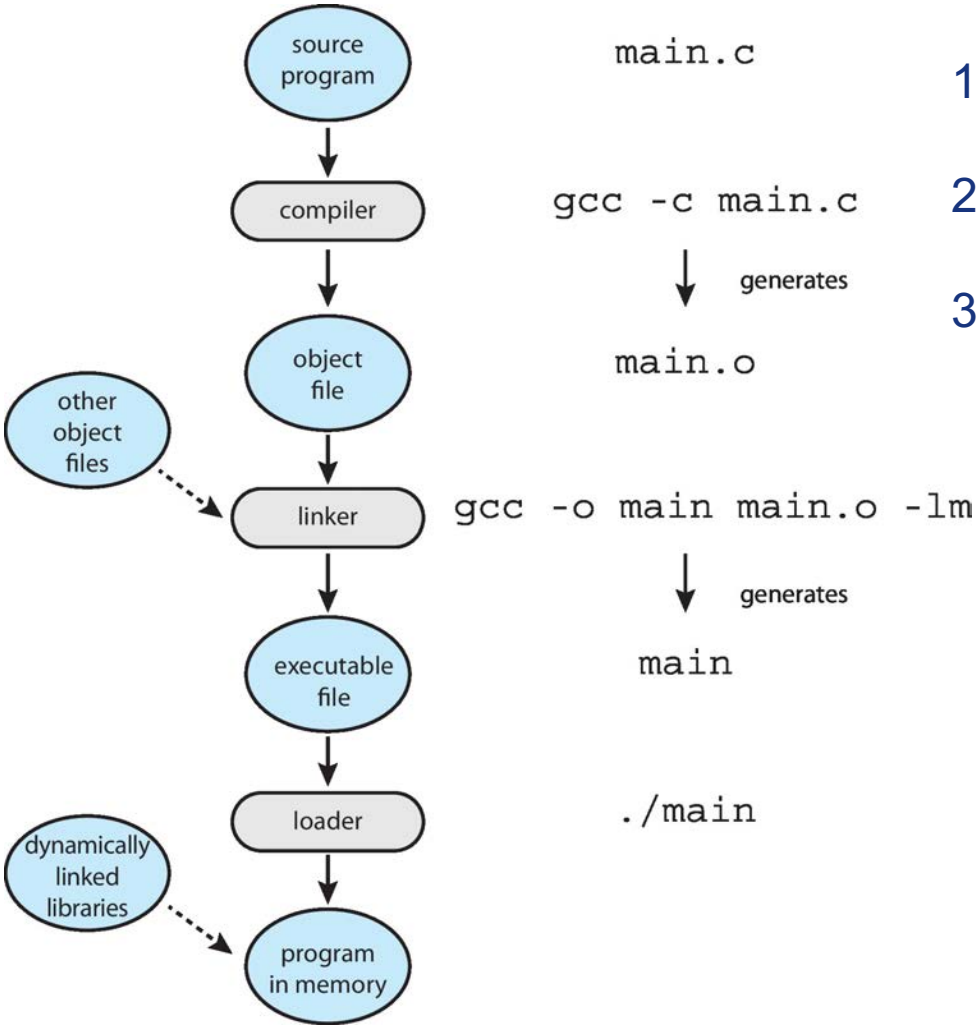
- 后台服务
  - 启动时启动
    - 一些用于系统启动，然后终止
    - 有些是从系统启动到关机
  - 提供磁盘检查、进程调度、错误记录、打印等功能
  - 在用户上下文而不是内核上下文中运行
  - 称为服务、子系统、守护进程

- 应用程序
  - 不属于系统
  - 由用户运行
  - 通常不被认为是操作系统的一部分
  - 通过命令行启动，鼠标单击；

程序编译和运行

- 源代码编译成目标文件，设计为加载到任何物理内存位置－可重定位目标文件
- 链接器将这些文件组合成一个二进制可执行文件
  - 也带来了图书馆
- 程序作为二进制可执行文件驻留在辅助存储器上
- 必须由要执行的加载程序带入内存
  - 重新定位将最终地址分配给程序部件，并调整程序中的代码和数据以匹配这些地址
- 现代通用系统不会将库链接到可执行文件中
  - 而是根据需要加载动态链接库（在Windows中，DLL），由使用同一库的同一版本的所有用户共享（加载一次）
- 对象，可执行文件具有标准格式，所以操作系统知道如何加载和启动它们

链接器和加载程序的角色



1. gcc -E helloworld.c -o helloworld.i;
2. gcc -S helloworld.i –o helloworld.o
3. gcc –o hellworld helloworld.o

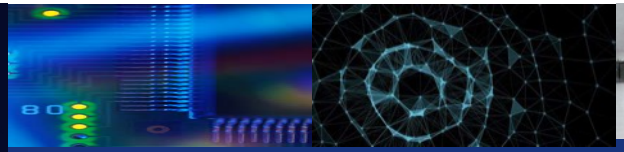
Demo

应用程序基于特定的操作系统

- 在一个系统上编译的应用程序通常不能在其他操作系统上执行
- 每个操作系统都提供自己独特的系统调用
  - 自己的文件格式等。
- 应用程序可以在多操作系统运行
  - 用解释语言编写，如Python、Ruby和解释器，可在多个操作系统上使用
  - 用语言编写的应用程序，包括包含运行应用程序的VM（如Java）
  - 使用标准语言（如C），在每个操作系统上分别编译以在每个操作系统上运行
- 应用程序二进制接口（ABI）是API的体系结构等价物，定义了二进制代码的不同组件如何在给定的体系结构、CPU等上为给定的操作系统进行接口。

操作系统设计与实现

- 操作系统的设计和实现没有完成的方案，但一些方法已被证明是成功的；
- 不同操作系统的内部结构可能差异很大
- 通过定义目标和规范开始设计
- 受硬件选择、系统类型的影响（批处理、分时、单用户、多用户、分布式、实时或通用）
- 用户目标和系统目标
  - 用户目标 - 操作系统应易于使用、易学、可靠、安全和快速
  - 系统目标——操作系统应该易于设计、实现和维护，并且灵活、可靠、正确和高效
- 指定和设计操作系统是软件工程中极具创造性的任务



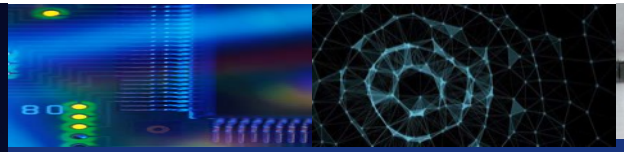
机制和策略

- 策略：需要做什么？
  - 示例：每100秒中断一次
- 机制：如何做某事？
  - 示例：计时器
- 重要原则：策略与机制分开

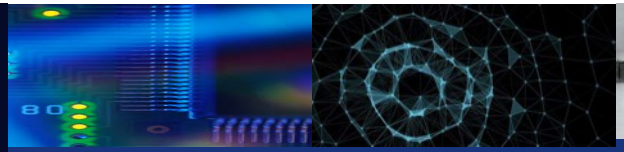
- 策略与机制的分离是一个非常重要的原则，如果策略决定以后要改变，它允许最大限度的灵活性。
  - 示例：将100更改为200

实现

- 千差万别
  - 汇编语言中的早期操作系统
  - 然后是系统编程语言，如Algol、PL/1
  - 现在C，C++
- 实际上通常是多种语言的混合
  - 装配中的最低级别
  - 主体在C中
  - C、C++、脚本语言的系统程序，如Perl、Python、shell脚本
- 更高级的语言更容易移植到其他硬件
  - 但是慢一点
- 仿真可以允许操作系统在非本机硬件上运行



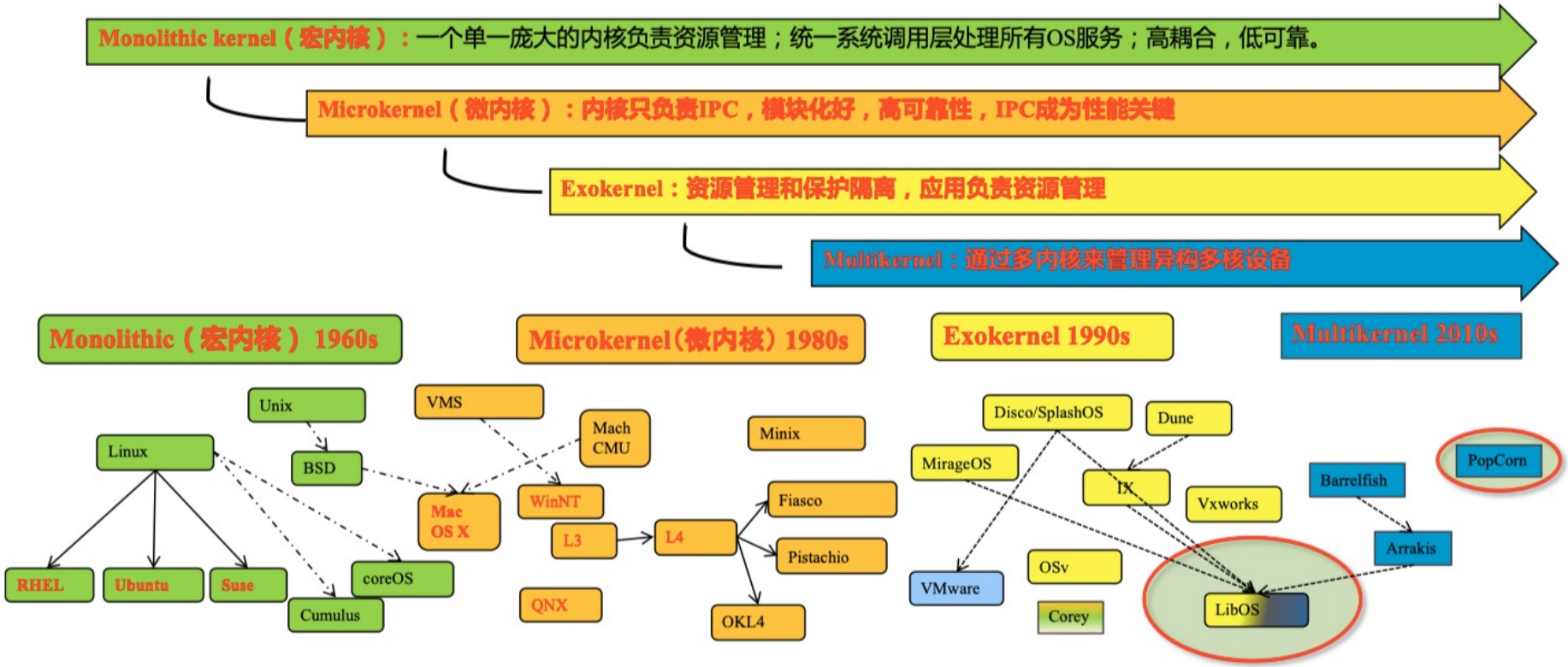
4. 操作系统的结构



操作系统的结构

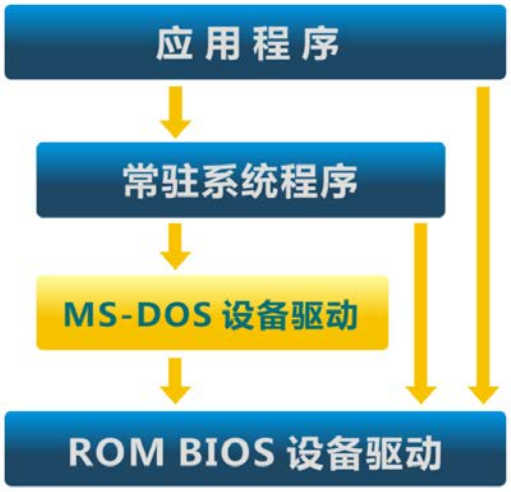
- 通用操作系统是一个非常大的程序
- 构造一个的各种方法
  - 结构简单 - MS-DOS
  - 更复杂 - UNIX
  - 分层-抽象
  - 微核-Mach

操作系统结构演化


Monolithic kernel（宏内核）：一个单一庞大的内核负责资源管理；统一系统调用层处理所有OS服务；高耦合，低可靠。
Microkernel（微内核）：内核只负责IPC，模块化好，高可靠性，IPC成为性能关键
Exokernel：资源管理和保护隔离，应用负责资源管理
Multikernel：通过多内核来管理异构多核设备
Monolithic（宏内核）1960s
Microkernel(微内核）1980s
Exokernel 1990s
Multikernel 2010s
Unix
VMS
Linux
BSD
Mach CMU
Minix
Mac OS X
WinNT
L3
L4
Fiasco
Pistachio
QNX
OKL4
RHEL
Ubuntu
Suse
coreOS
Cumulus
Disco/SplashOS
Dune
MirageOS
IX
Vxworks
Barrelfish
PopCorn
Arrakis
VMware
OSv
Corey
LibOS

操作系统结构

简单结构

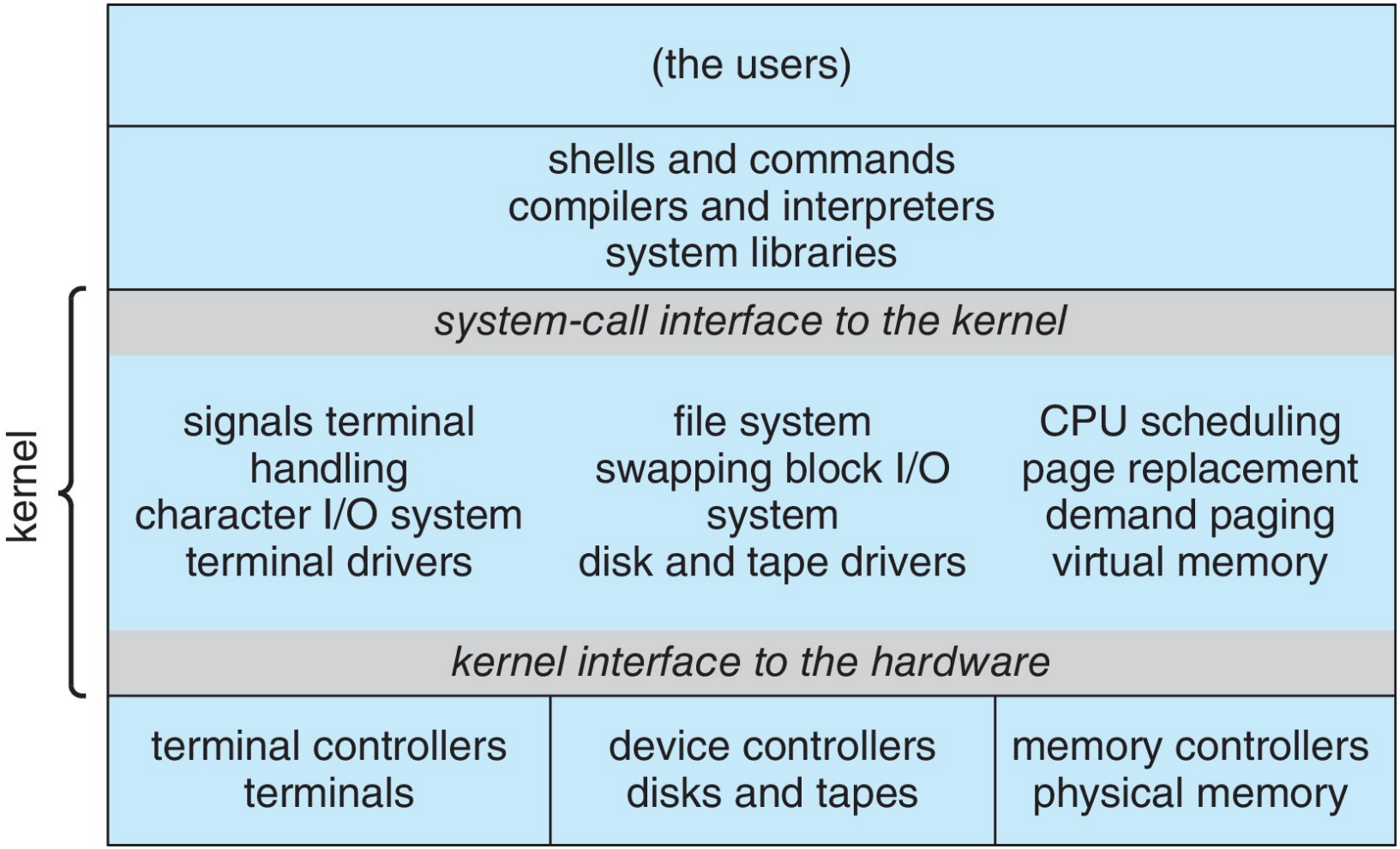


- MS-DOS：在最小的空间，设计用于提供大部分功能 (1981–1994)
  - 没有拆分为模块
  - 主要用汇编编写
  - 没有安全保护

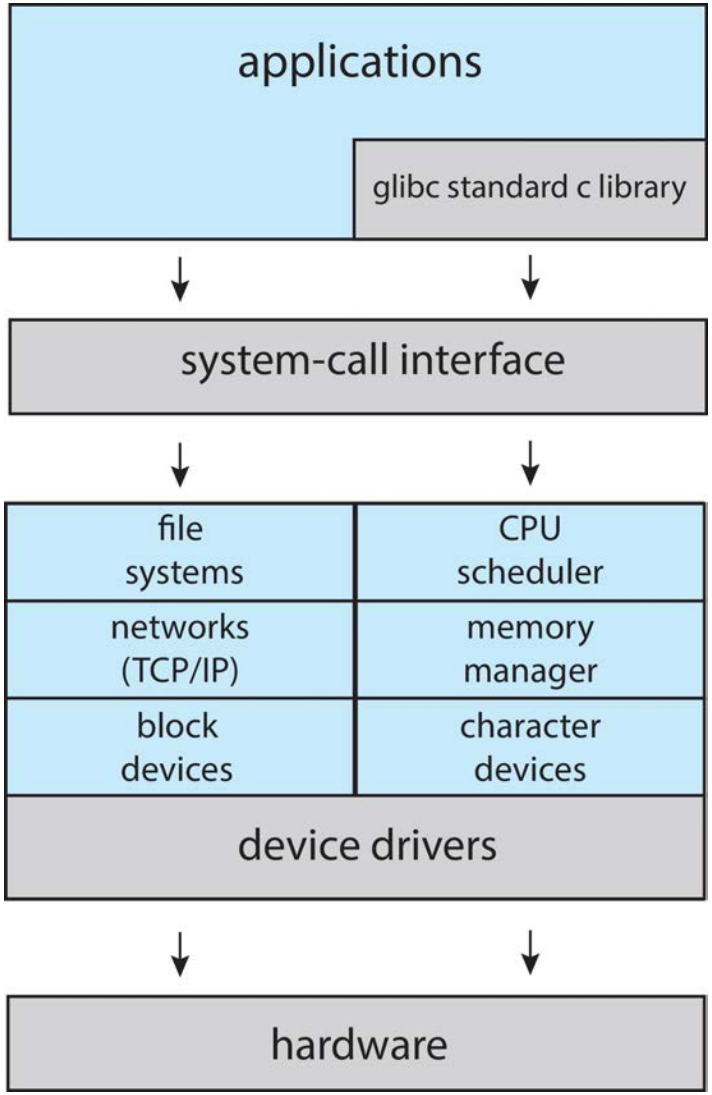
简单结构

- UNIX－受硬件功能的限制，最初的UNIX操作系统结构有限。
- UNIX操作系统由两个可分离的部分组成
  - 系统程序
  - 内核
    - 包括系统调用接口下方和物理硬件上方的所有内容
    - 提供文件系统、CPU调度、内存管理等操作系统功能；一个级别的大量函数；

简单结构（宏内核）

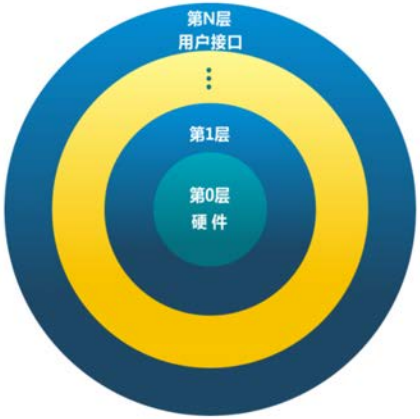

(the users)
shells and commands
compilers and interpreters
system libraries
system-call interface to the kernel
signals terminal handling
character I/O system
terminal drivers
file system
swapping block I/O system
disk and tape drivers
CPU scheduling
page replacement
demand paging
virtual memory
kernel interface to the hardware
terminal controllers
terminals
device controllers
disks and tapes
memory controllers
physical memory
kernel

简单结构（宏内核）


applications
glibc standard c library
system-call interface
file systems
CPU scheduler
networks (TCP/IP)
memory manager
block devices
character devices
device drivers
hardware

操作系统结构

单体分层结构（宏内核）

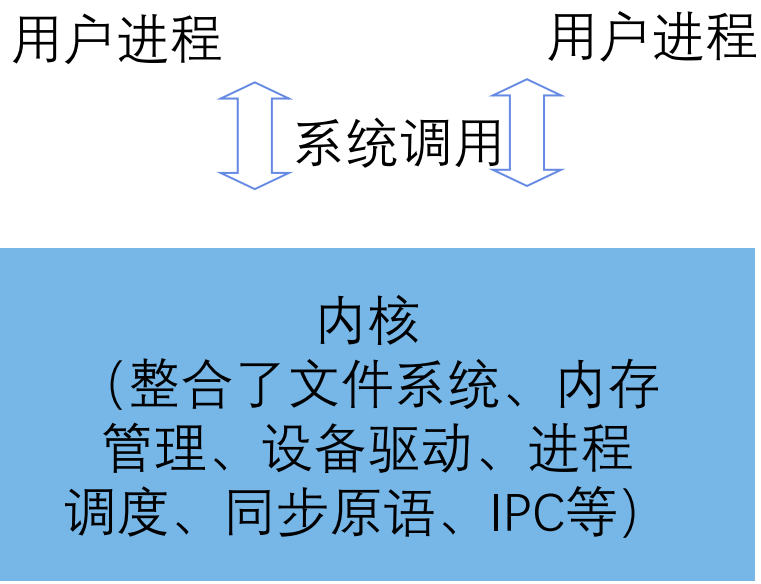


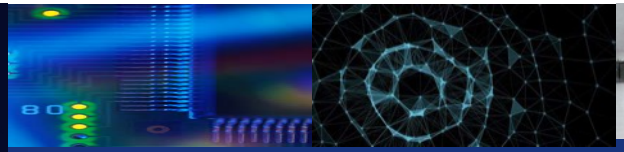
- 将单体操作系统 (Monolithic OS) 划分为多层 (levels)
  - 每层建立在低层之上
  - 最底层 (layer 0), 是硬件驱动
  - 最高层 (layer N) 是用户界面
- 每一层仅使用更低一层的功能和服务。

分层方法的倒退

简单结构（宏内核）

- **整个系统分为内核与应用两层**
  - 内核：运行在特权级，集中控制所有计算资源
  - 应用：运行在非特权级，受内核管理，使用内核服务





简单结构（宏内核）的缺点

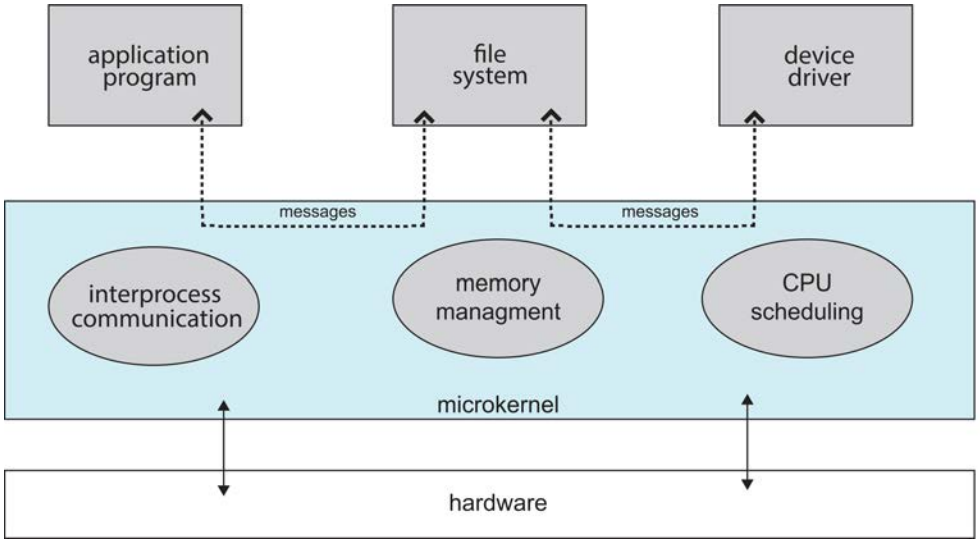
- **宏内核的结构性缺陷**

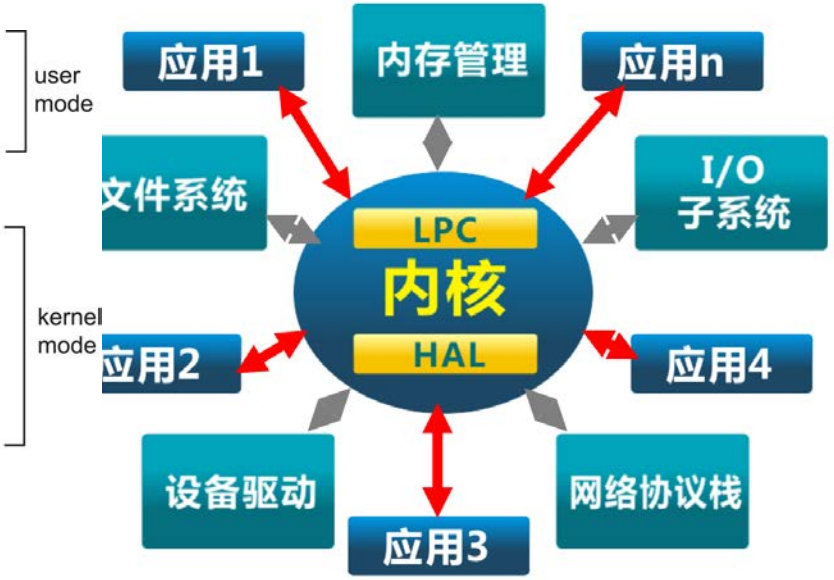
  安全性与可靠性问题：模块之间没有很强的隔离机制
  实时性支持：系统太复杂导致无法做最坏情况时延分析
  系统过于庞大而阻碍了创新：Linux代码行数已经过2千万

  - **向上向下的扩展**
    - 很难去剪裁/扩展一个宏内核系统支持从KB级别到TB级别的场景
  - **硬件异构性**
    - 很难长期支持一些定制化的方式去解决一些特定问题
  - **功能安全**
    - 一个广泛共识：Linux无法通过汽车安全完整性认证（ASIL-D）
  - **信息安全**
    - 单点错误会导致整个系统出错，而现在有数百个安全问题（CVE）
  - **确定性时延**
    - Linux花费10+年合并实时补丁，目前依然不确定是否能支持确定性时延

操作系统结构

微内核结构



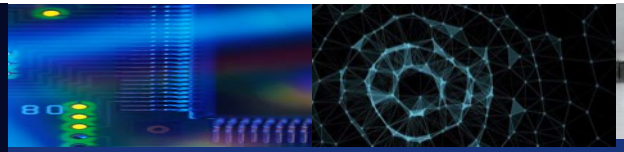


- 尽可能把内核功能移到用户空间
- 用户模块间的通信使用消息传递
- 好处：灵活/安全…
- 缺点：性能

微内核

Rick Rashid

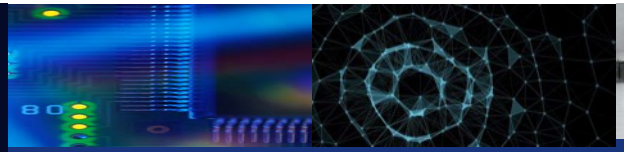
- 1985年，Mach 发布
  - 由CMU开发，Rick Rashid领导
  - 对操作系统发展产生了重大影响
- 1986年，Mach 2.5 （性能比UNIX差25%）
  - 包含大量BSD的代码，如1:1的task与process映射，导致内核比UNIX更大
  - 取得了商业成功，用于NeXT，最终被苹果收购
- 1990年，Mach 3.0（性能比UNIX差67%）
  - 规避法律风险，去掉了BSD的代码，重写了IPC以提高性能
  - 提出"continuation"，为用户态应用提供了更多控制
    - 允许应用自己在切换的时候保存/恢复上下文，进一步减小microkernel




微内核

Mach提供的功能

- **任务和线程管理**
  - 任务，是资源分配的基本单位；线程，是执行的基本单位
  - 对应用提供调度接口，应用程序可实现其自定义的调度策略
- **进程间通信（IPC）**：通过端口（port）进行通信
- **内存对象管理**：虚拟内存
- **系统调用重定向**：允许用户态处理系统调用
  - 支持对系统调用的功能扩展，例如，二进制翻译、跟踪、调试等



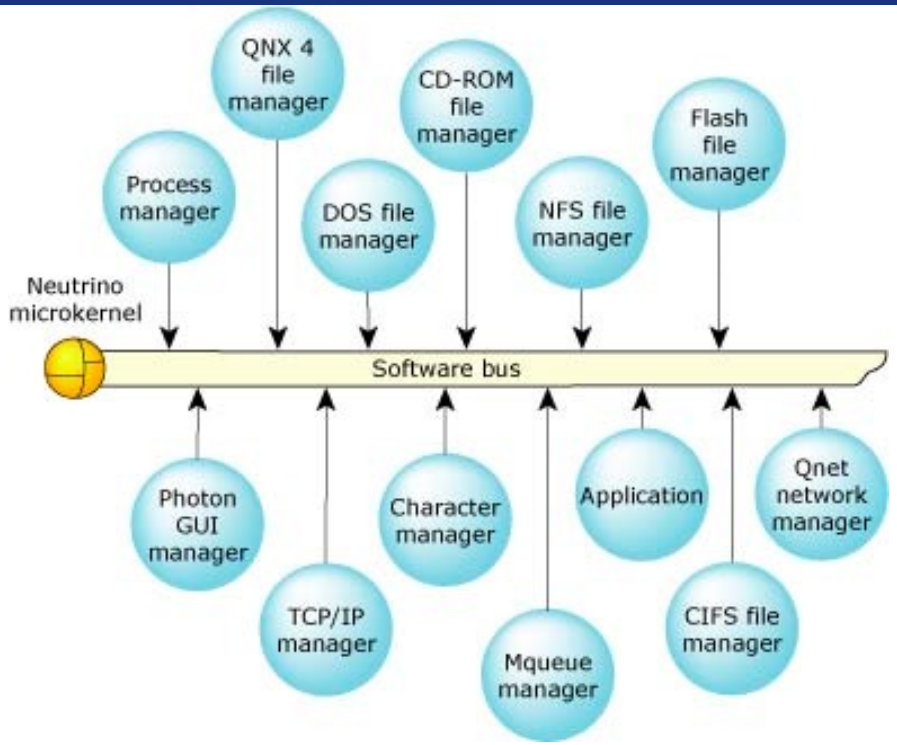
微内核

Mach提供的功能

- Mach允许用户态代码实现Paging
  - 应用可自己管理自己的虚拟内存
- 重定向功能（Redirection）
  - 允许发生中断/异常时，直接执行用户的二进制
  - 这种连接不需要对内核做修改

微内核

QNX Neutrino



- QNX：Quick UNIX
  - 使用Neutrino微内核
  - 1980年发布
  - 2004年被Harman国际收购
  - 2010被黑莓收购

- **满足实时要求**
  - 广泛用于交通、能源、医疗、航天航空领域，如波音

微内核

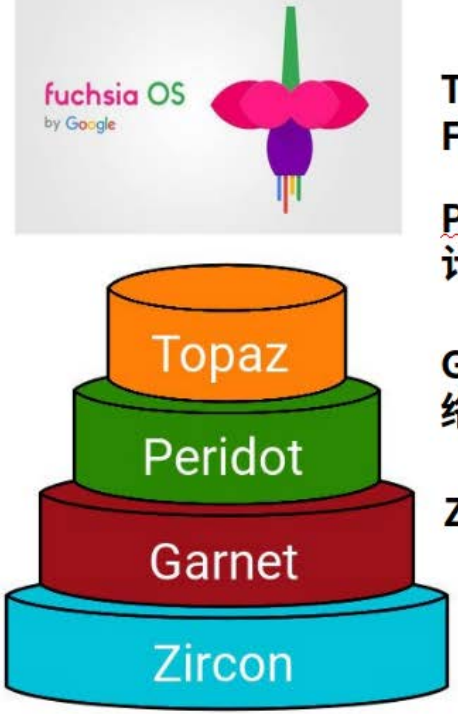
MINIX

- 教学用的微内核
  - 阿姆斯特丹自由大学，Andrew Tanenbaum教授
- 被用于Intel的ME模块
  - 也许是世界上用的最多的操作系统...

Andrew Tanenbaum

https://www.zdnet.com/article/minix-intels-hidden-in-chip-operating-system/

微内核



Topaz： Runtime/ 前端框架和系统 UI/ 系统程序，提供 Flutter 支持，及其应用程序

Peridot：系统框架和相关，处理 Fuchsia 的模块化应用程序设计，跨设备保存信息

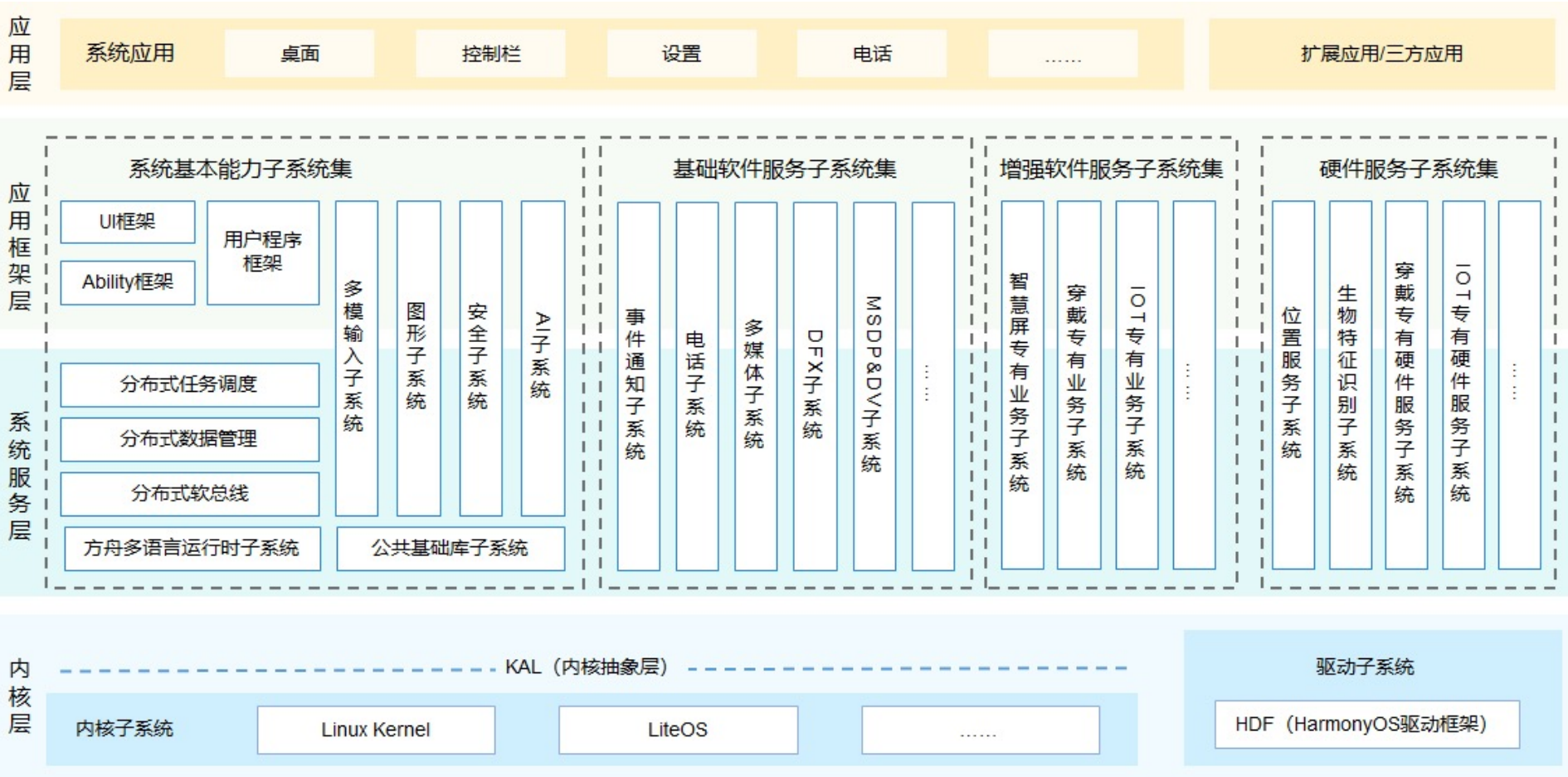
Garnet：平台相关基础库和服务，包括硬件的驱动程序（网络，图形等）和软件安装

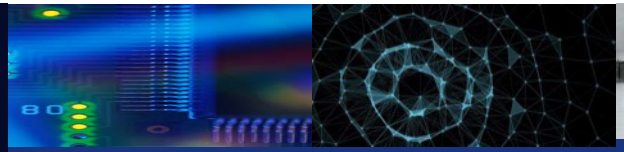
Zircon 内核：处理硬件访问和软件之间的通信

图: Fuchsia 操作系统–AIoT 操作系统的代表之一

微内核

HarmonyOS整体遵从分层设计，从下向上依次为：内核层、系统服务层、框架层和应用层。系统功能按照“系统 > 子系统 > 功能/模块”逐级展开，在多设备部署场景下，支持根据实际需求裁剪某些非必要的子系统或功能/模块。HarmonyOS技术架构如图所示。





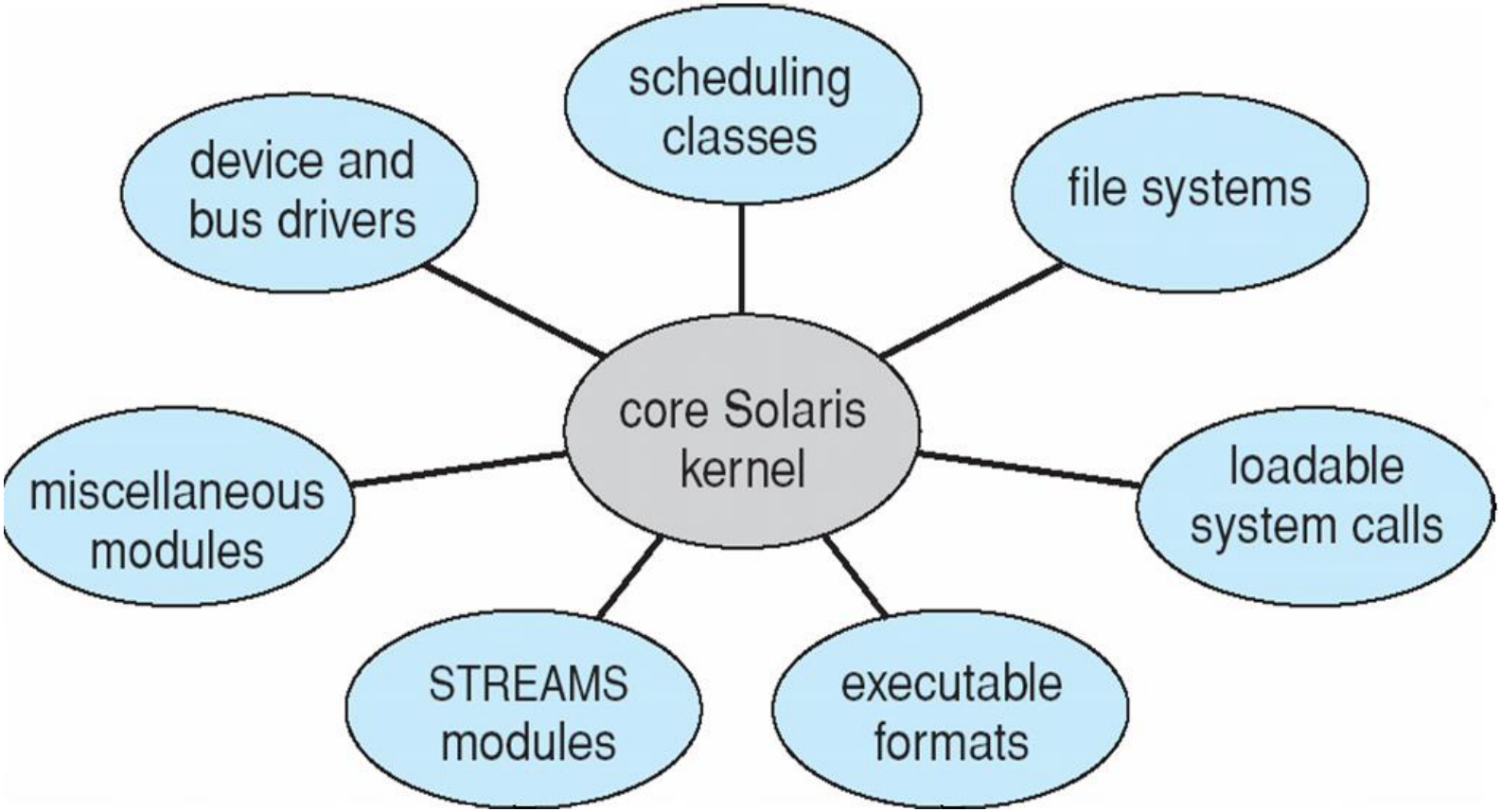
模块化

- 许多现代操作系统实现可加载内核模块（LKM）
  - 使用面向对象的方法
  - 每个核心组件都是独立的
  - 每个人都通过已知的接口与其他人交谈
  - 每个都可以根据需要在内核中加载
- 总体而言，与分层类似，但更灵活
  - Linux、Solaris等。

Demo

Solaris模块

Solaris一个核心内核有7种类型的可加载内核模块：

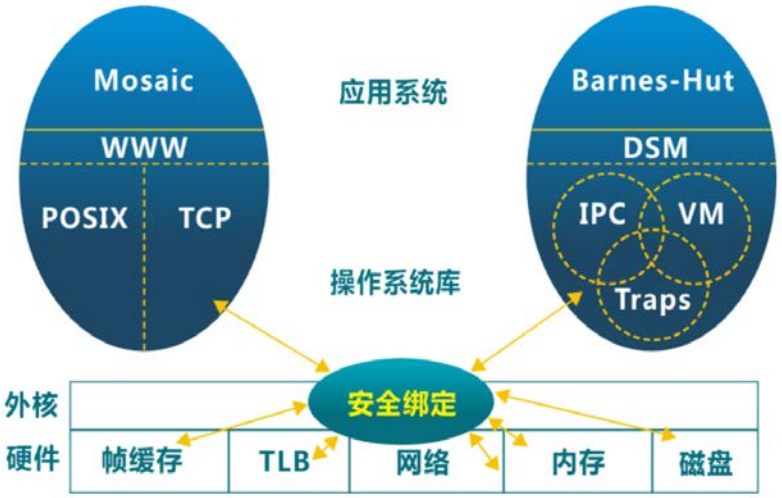


Linux 模块增加

```c
#define LINUX

#include <linux/module.h>
#include <linux/kernel.h>
#include "mp1_given.h"

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Group_ID");
MODULE_DESCRIPTION("CS-423 MP1");

#define DEBUG 1

// mp1_init - Called when module is loaded
int __init mp1_init(void)
{
   printk(KERN_ALERT "Hello, World\n");
   return 0;
}

// mp1_exit - Called when module is unloaded
void __exit mp1_exit(void)
{
   printk(KERN_ALERT "Goodbye, World\n");
}

// Register init and exit funtions
module_init(mp1_init);
module_exit(mp1_exit);
```

操作系统结构

外核结构

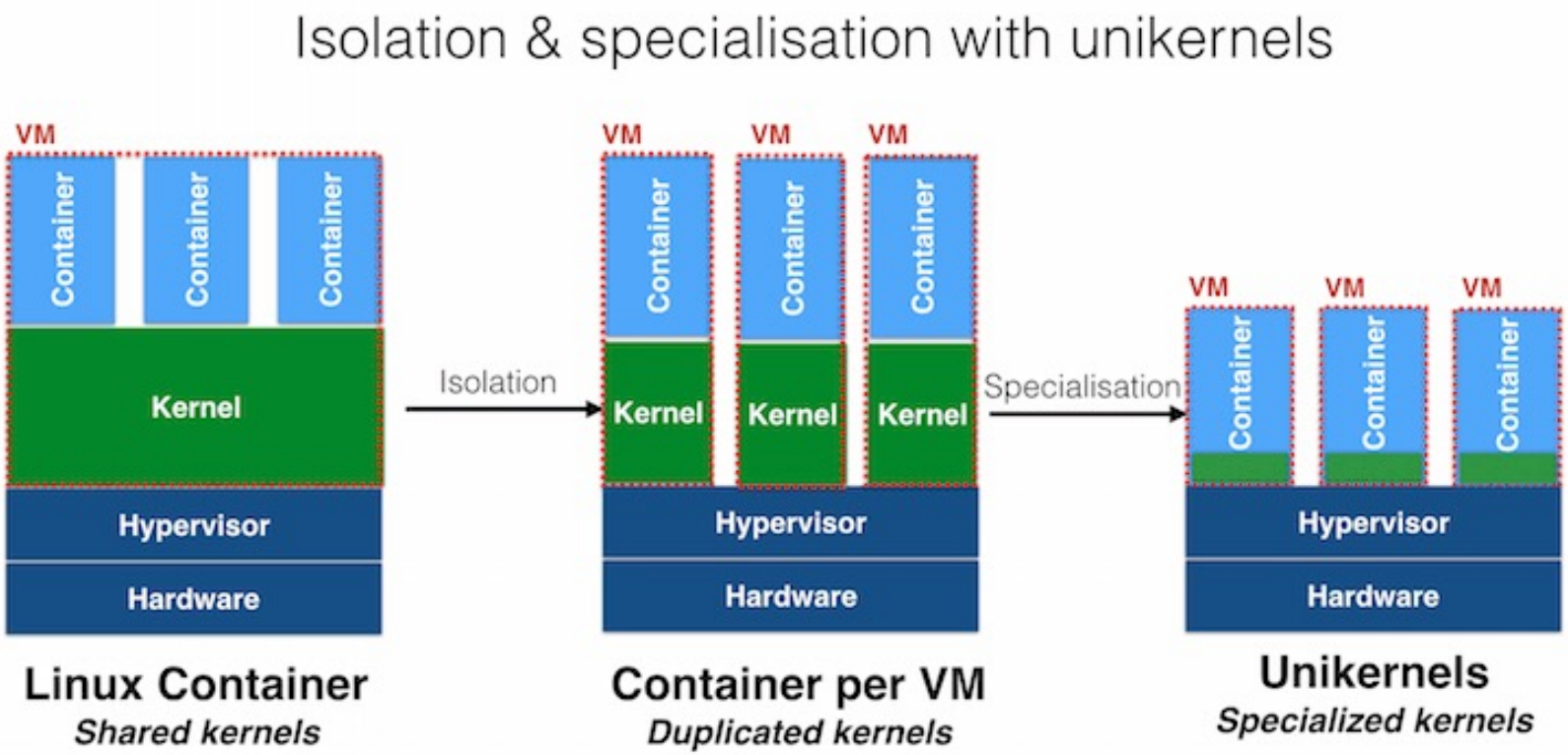


- 让内核分配机器的物理资源给多个应用程序，并让每个程序决定如何处理这些资源
- 程序能链接到操作系统库 (libOS) 实现了操作系统抽象
- 保护与控制分离

Unikernel!!

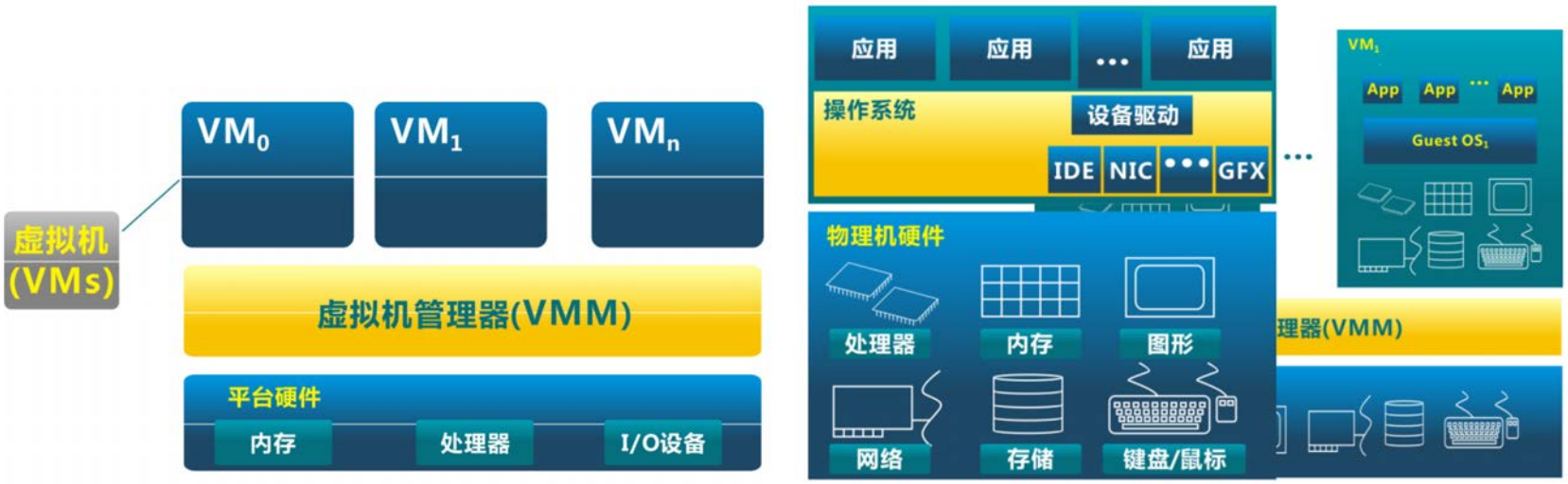
操作系统结构

Unikernel



操作系统结构

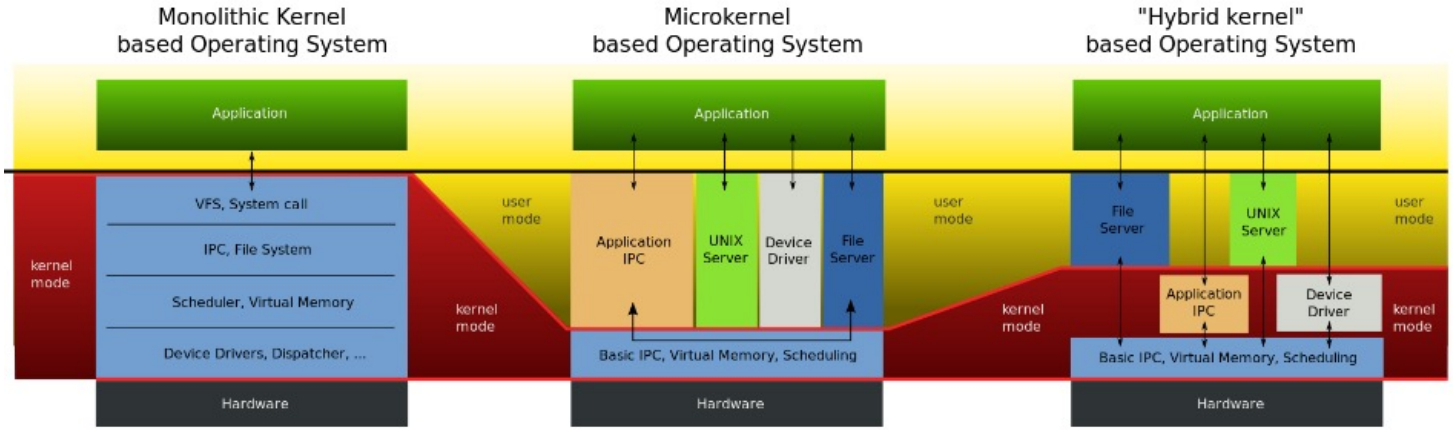
VMM（虚拟机监控器）



- 虚拟机管理器将单独的机器接口转换成很多的虚拟机，每个虚拟机都是一个原始计算机系统的有效副本，并能完成所有的处理器指令。

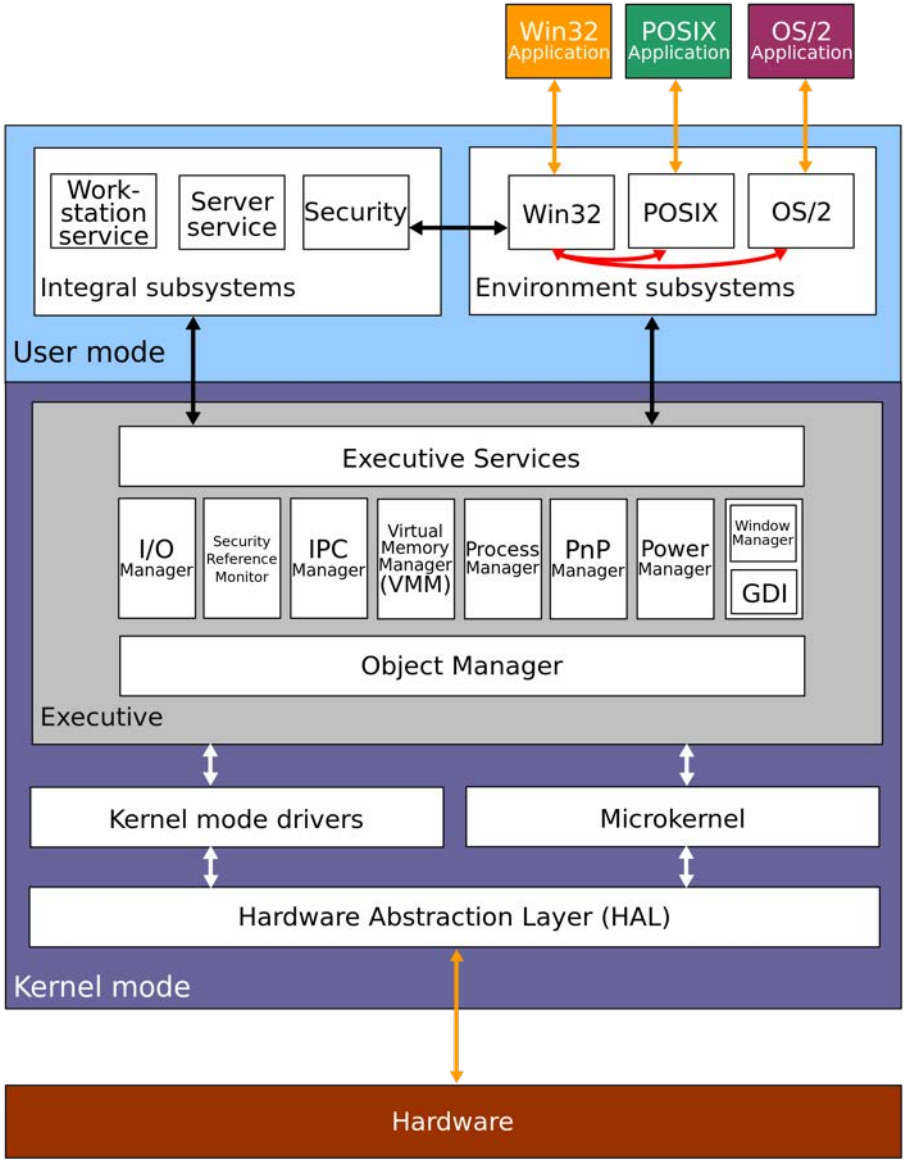
混合系统

- 宏内核与微内核的结合
  - 将需要性能的模块重新放回内核态
  - 例：macOS / iOS = Mach微内核 + BSD 4.3 + 系统框架
  - 例：Windows NT = 微内核 + 内核态的系统服务 + 系统框架



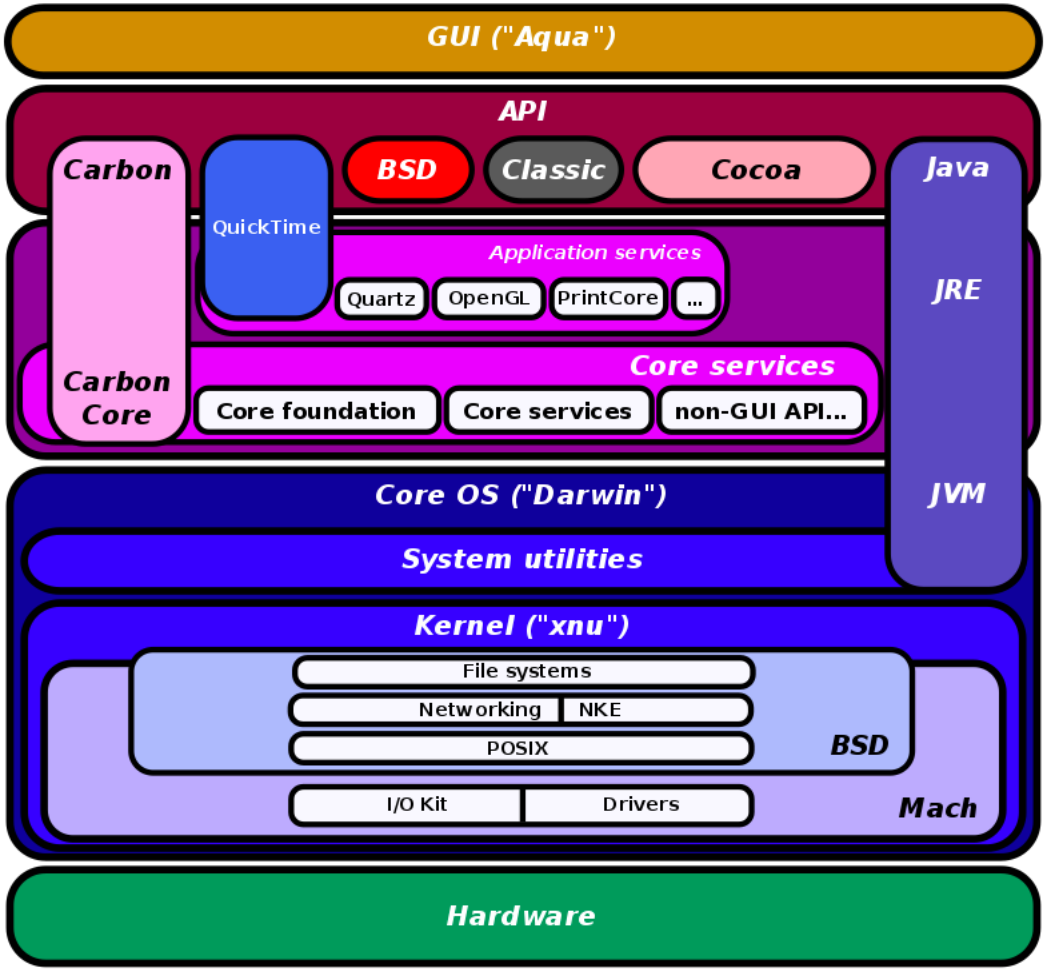
Wiondows NT

- Integral子系统（用户态）
  - 负责处理I/O、对象管理、安全、进程等
- 环境子系统（用户态）
  - POSIX
- Executive（内核态）
  - 为用户态子系统提供服务
- Microkernel
  - 提供进程间同步等功能

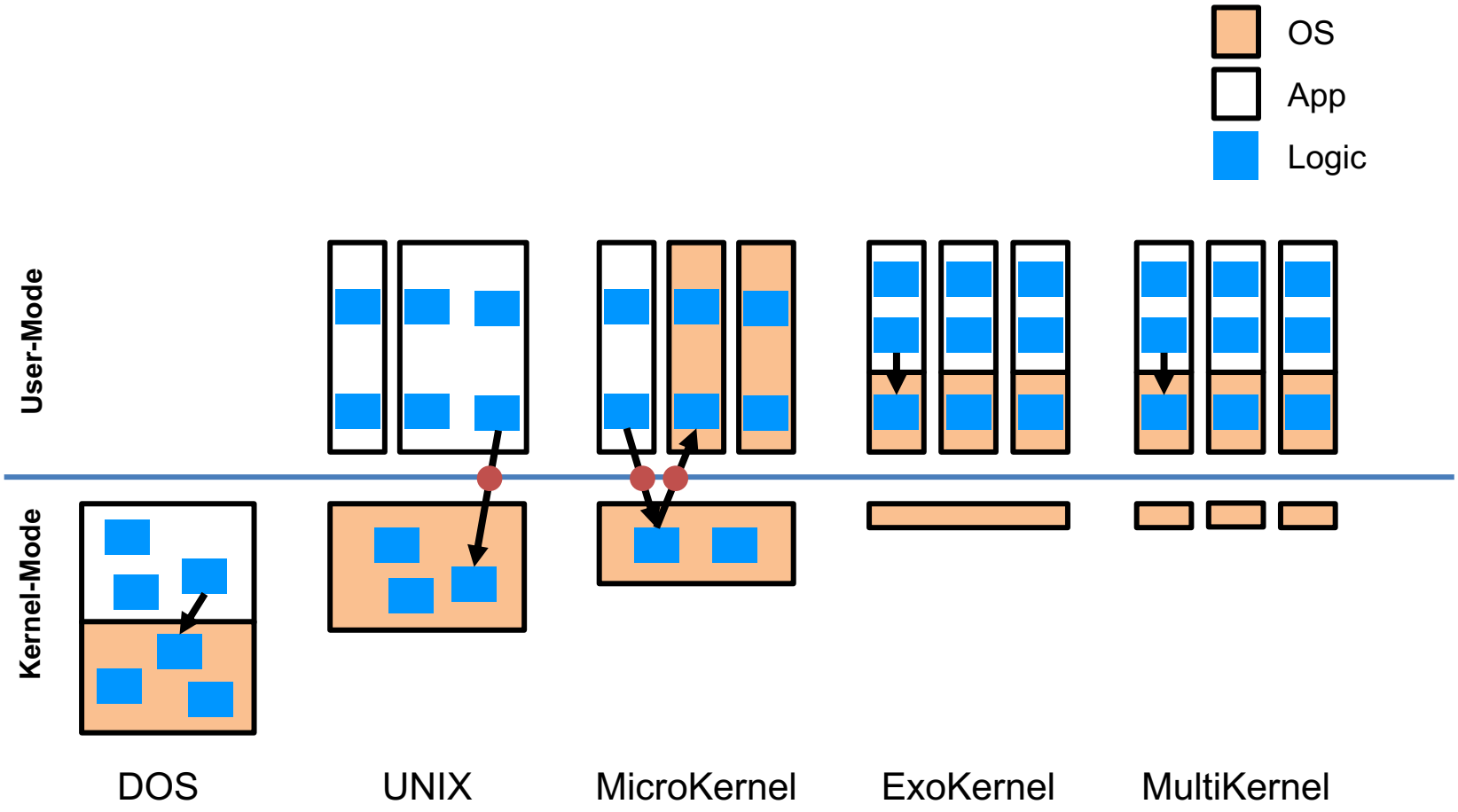


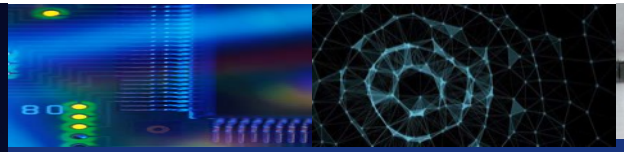
Mac OS

- XNU内核
  - 基于Mach-2.5打造
  - BSD代码提供文件系统、网络、POSIX接口等
- macOS与iOS



不同操作系统架构的对比





5. 操作系统实例

Windows对象

- Windows利用了大量的面向对象的概念；
- 关键的面向对象的概念包括：

封装 → 对象类和实例 → 继承 → 多态

Windows对象

Asynchronous Procedure Call	Used to break into the execution of a specified thread and to cause a procedure to be called in a specified processor mode.
Deferred Procedure Call	Used to postpone interrupt processing to avoid delaying hardware interrupts. Also used to implement timers and inter-processor communication
Interrupt	Used to connect an interrupt source to an interrupt service routine by means of an entry in an Interrupt Dispatch Table (IDT). Each processor has an IDT that is used to dispatch interrupts that occur on that processor.
Process	Represents the virtual address space and control information necessary for the execution of a set of thread objects. A process contains a pointer to an address map, a list of ready threads containing thread objects, a list of threads belonging to the process, the total accumulated time for all threads executing within the process, and a base priority.
Thread	Represents thread objects, including scheduling priority and quantum, and which processors the thread may run on.
Profile	Used to measure the distribution of run time within a block of code. Both user and system code can be profiled.

Table 2.5 Windows Kernel Control Objects

Windows家族

微软从DEC聘请 Dave Cutler 做Windows NT主要设计师

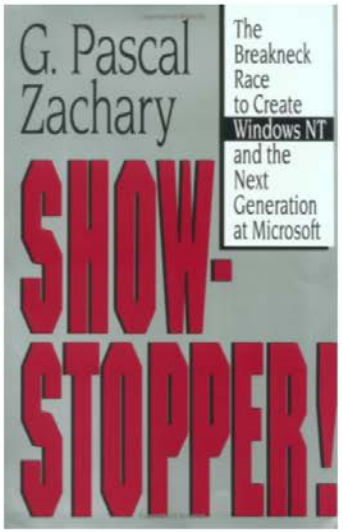
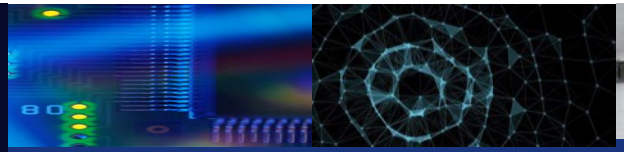


图: Windows 家族



传统的Unix

- **Were developed at Bell Labs and became operational on a PDP-7 in 1970**
- **Incorporated many ideas from Multics**
- **PDP-11was a milestone because it first showed that UNIX would be an OS for all computers**
- **Next milestone was rewriting UNIX in the programming language C**
  - demonstrated the advantages of using a high-level language for system code
- Was described in a technical journal for the first time in 1974
- First widely available version outside Bell Labs was Version 6 in 1976
- Version 7, released in 1978 is the ancestor of most modern UNIX systems
- Most important of the non-AT&T systems was UNIX BSD (Berkeley Software Distribution)

传统的Unix

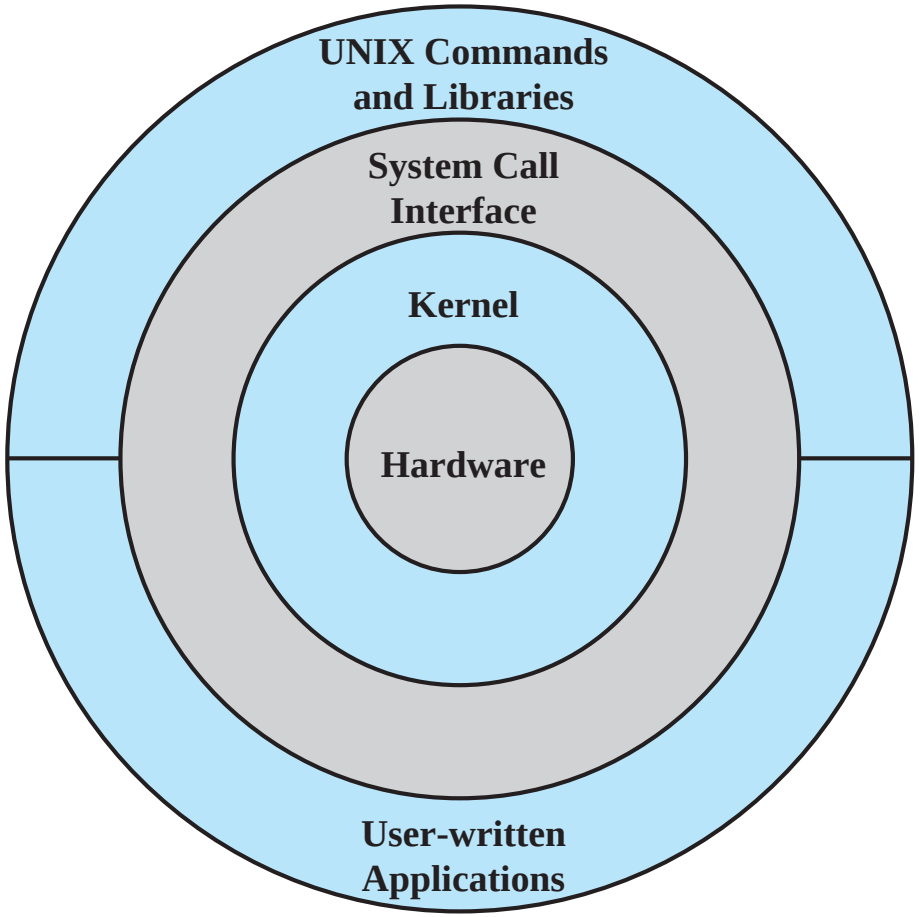


Figure 2.15 General UNIX Architecture

传统的Unix内核

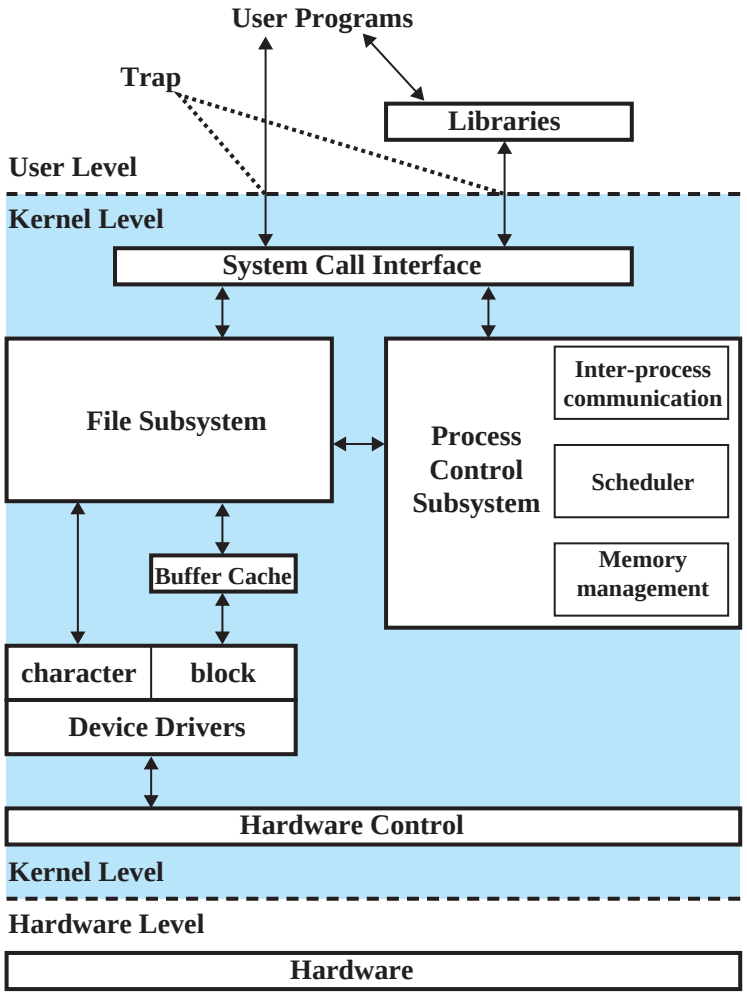


Figure 2.16 Traditional UNIX Kernel

现代Unix内核

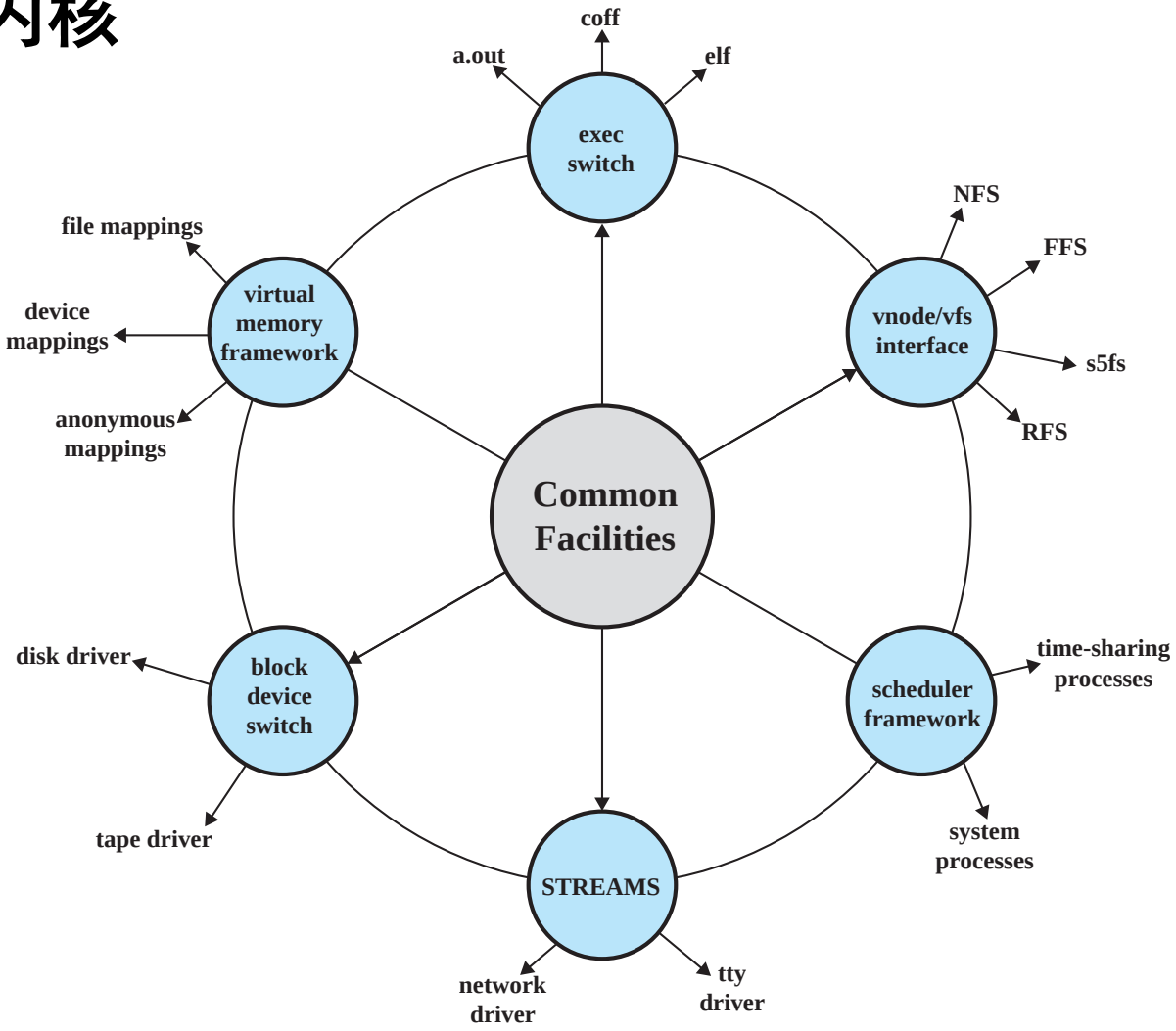


Figure 2.17 Modern UNIX Kernel [VAHA96]

Unix家族

肯·汤普森（Ken Thompson）和丹尼斯·里奇（Dennis Ritchie）

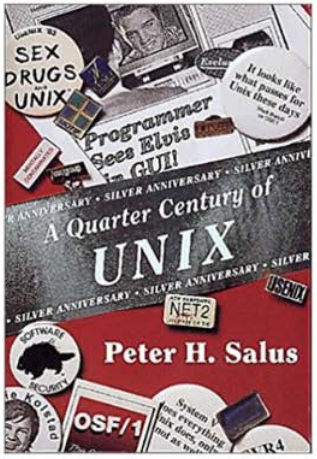


图: UNIX 家族

Unix历史

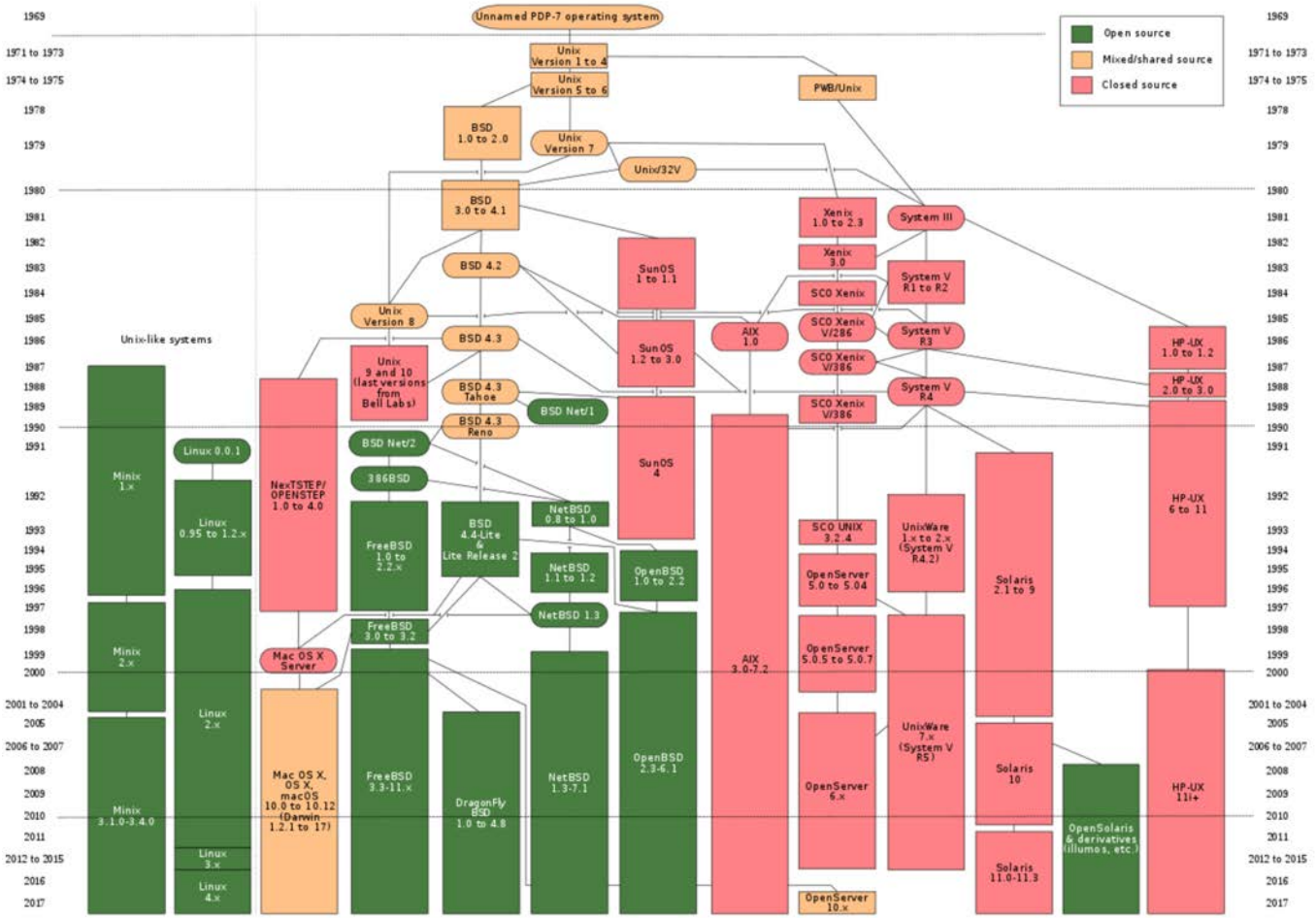
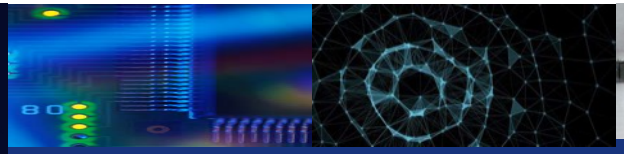


图: UNIX 历史

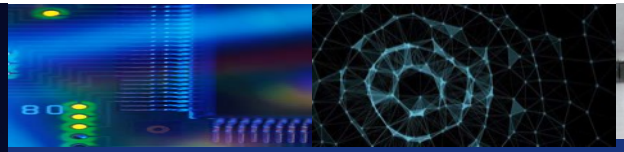


System V Release 4 (SVR4)

- **Developed jointly by AT&T and Sun Microsystems**
- **Combines features from SVR3, 4.3BSD, Microsoft Xenix System V, and SunOS**
- **New features in the release include:**
  - real-time processing support
  - process scheduling classes
  - dynamically allocated data structures
  - virtual memory management
  - virtual file system
  - preemptive kernel

BSD

- **Berkeley Software Distribution**
- **4.xBSD is widely used in academic installations and has served as the basis of a number of commercial UNIX products**
- **4.4BSD was the final version of BSD to be released by Berkeley**
  - major upgrade to 4.3BSD
  - includes
    - a new virtual memory system
    - changes in the kernel structure
    - several other feature enhancements
- **FreeBSD**
  - one of the most widely used and best documented versions
  - popular for Internet-based servers and firewalls
  - used in a number of embedded systems
  - Mac OS X is based on FreeBSD 5.0 and the Mach 3.0 microkernel



SUN Solaris 10

- **Sun's SVR4-based UNIX release**
- **Provides all of the features of SVR4 plus a number of more advanced features such as:**
  - a fully preemptable, multithreaded kernel
  - full support for SMP
  - an object-oriented interface to file systems
- **Most widely used and most successful commercial UNIX implementation**

MacOS家族

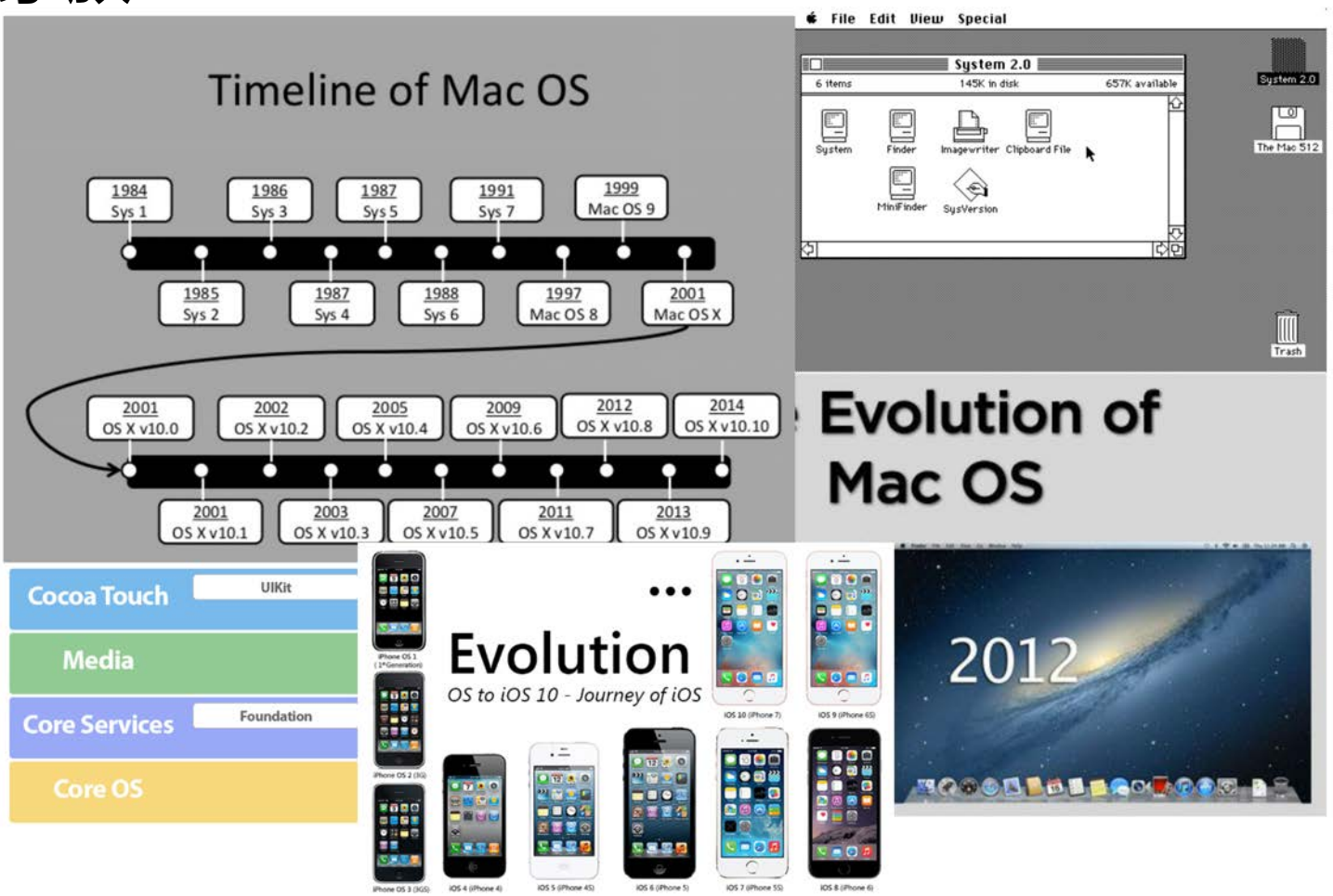
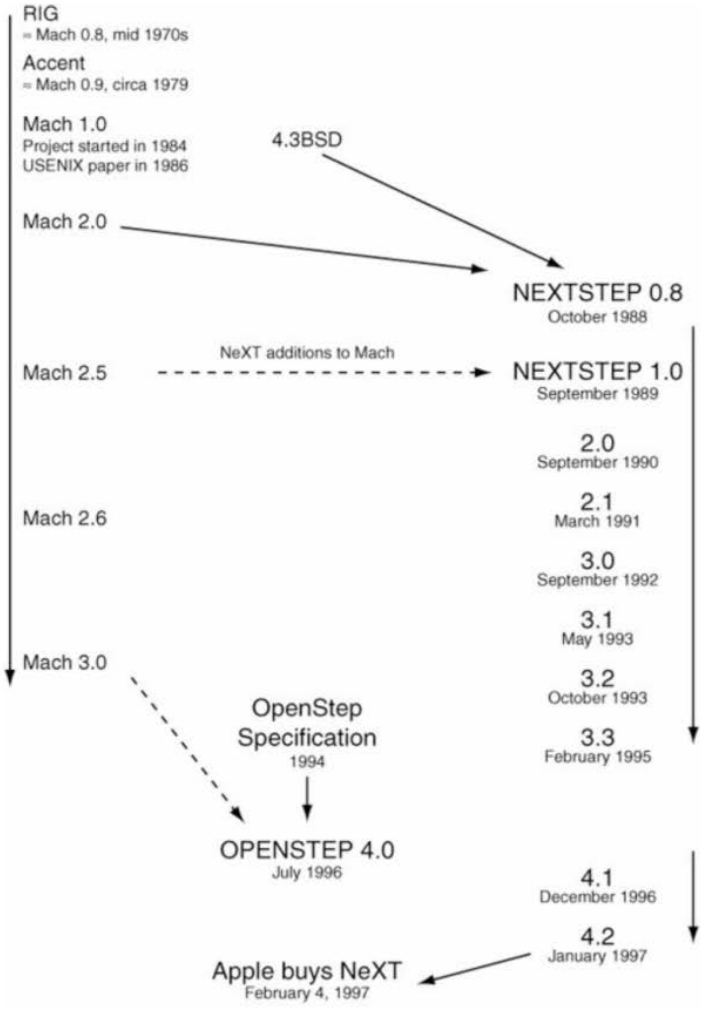
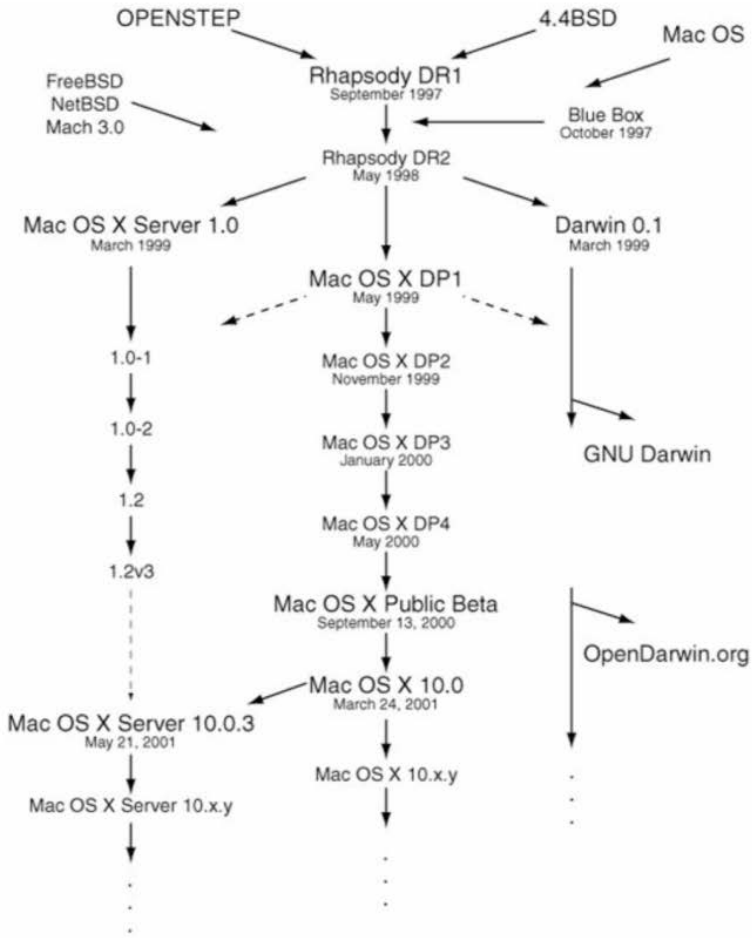
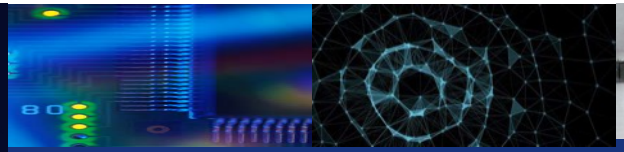


图: MacOS 家族

MacOS历史


RIG
= Mach 0.8, mid 1970s
Accent
= Mach 0.9, circa 1979
Mach 1.0
Project started in 1984
USENIX paper in 1986
4.3BSD
Mach 2.0
NEXTSTEP 0.8
October 1988
NeXT additions to Mach
Mach 2.5
NEXTSTEP 1.0
September 1989
2.0
September 1990
2.1
March 1991
Mach 2.6
3.0
September 1992
3.1
May 1993
Mach 3.0
3.2
October 1993
OpenStep
Specification
1994
3.3
February 1995
OPENSTEP 4.0
July 1996
4.1
December 1996
4.2
January 1997
Apple buys NeXT
February 4, 1997


OPENSTEP
4.4BSD
Mac OS
Rhapsody DR1
September 1997
FreeBSD
NetBSD
Mach 3.0
Blue Box
October 1997
Rhapsody DR2
May 1998
Mac OS X Server 1.0
March 1999
Darwin 0.1
March 1999
Mac OS X DP1
May 1999
1.0-1
Mac OS X DP2
November 1999
1.0-2
Mac OS X DP3
January 2000
GNU Darwin
1.2
Mac OS X DP4
May 2000
1.2v3
Mac OS X Public Beta
September 13, 2000
OpenDarwin.org
Mac OS X 10.0
March 24, 2001
Mac OS X Server 10.0.3
May 21, 2001
Mac OS X 10.x.y
Mac OS X Server 10.x.y



LINUX Overview

- Started out as a UNIX variant for the IBM PC
- Linus Torvalds, a Finnish student of computer science, wrote the initial version
- Linux was first posted on the Internet in 1991
- Today it is a full-featured UNIX system that runs on several platforms
- Is free and the source code is available
- Key to success has been the availability of free software packages
- Highly modular and easily configured

Android操作系统

- Android操作系统是为触屏移动设备设计的基于Linux的操作系统；
- 也是最流行的操作系统；
- 由Android公司开发，随后于2015年被google收购；
- 第一个商业版本Android 1.0在2008年发布；
- 最新版本是android 11
- 2007年开放手机联盟成立（OHA），OHA的宗旨是为手机指定公开标准，android发布由OHA负责；
- Android的开源属性是其成功的关键因素；

Android操作系统历史变迁

名称	版本名	API等级
Android 11	11.0	30
Android 10 [3]	10.0	29
Android Pie	9.0	28
Android Oreo	8.0-8.1	26-27
Android Nougat	7.0-7.1.2	24-25
Android Marshmallow	6.0-6.0.1	23
Android Lollipop	5.0-5.1.1	21-22
Android KitKat	4.4-4.4.4	19-20
Android Jelly Bean	4.1-4.3	16-18
Android Ice Cream Sandwich	4.0.1-4.0.4	14-15
Android Honeycomb	3.0-3.2	11-13
Android Gingerbread	2.3-2.3.7	9-10
Android Froyo	2.2	8
Android Eclair	2.0-2.1	5-7
Android Donut	1.6	4
Android Cupcake	1.5	3
	1.1	2

Android操作系统

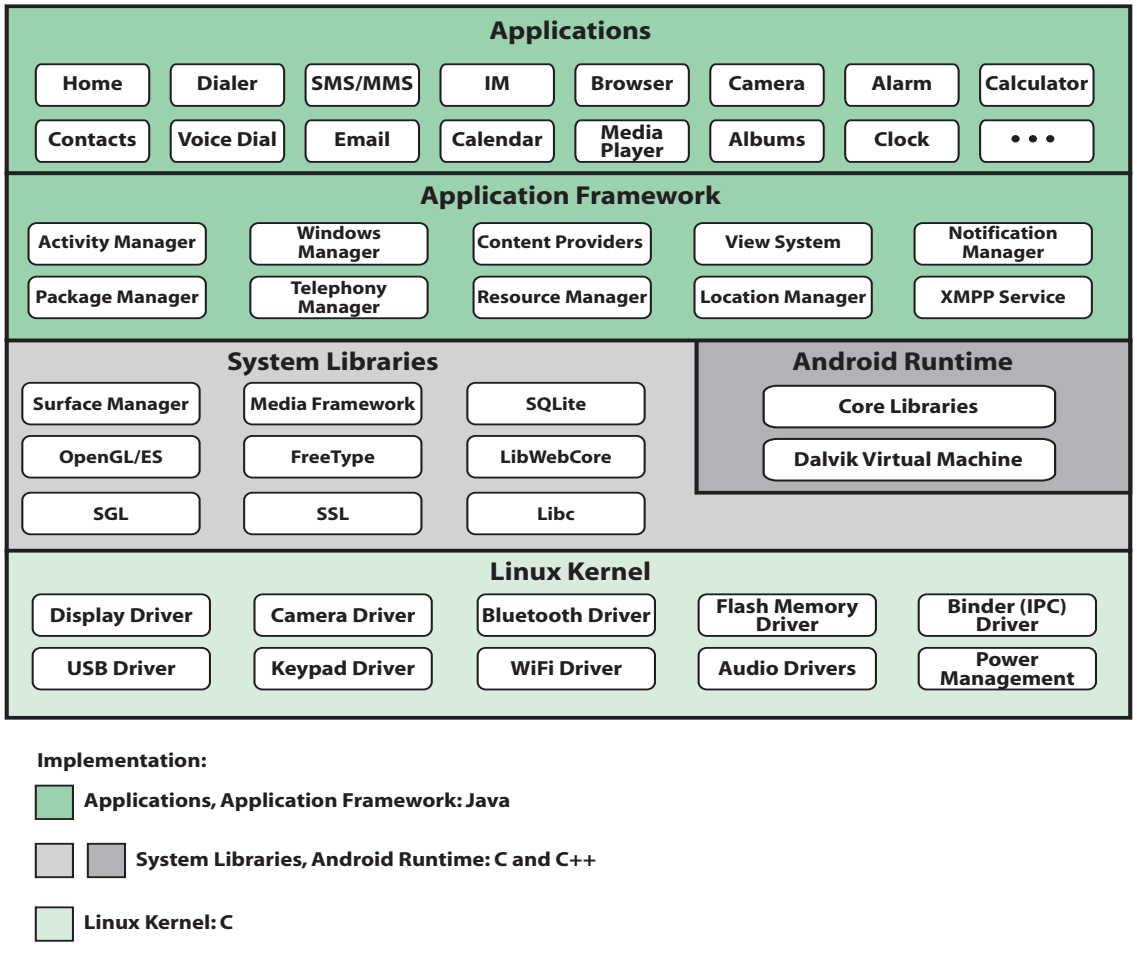


Figure 2.20 Android Software Architecture

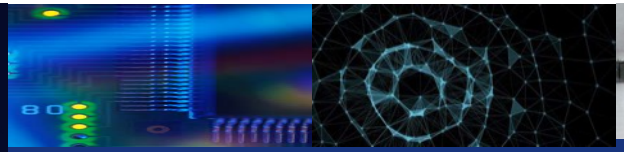
Android应用框架

- 应用框架层提供高级架构模块，为程序员开发程序提供标准化的访问接口
  - 旨在简化组件的复用
  - 关键组件如下：

Activity Manager	Window Manager	Package Manager	Telephony Manager
Manages lifecycle of applications	Java abstraction of the underlying Surface Manager	Installs and removes applications	Allows interaction with phone, SMS, and MMS services
Responsible for starting, stopping, and resuming the various applications	Allows applications to declare their client area and use features like the status bar		

Android应用框架

- Key components: (cont.)
  - Content Providers
    - these functions encapsulate application data that need to be shared between applications such as contacts
  - Resource Manager
    - manages application resources, such as localized strings and bitmaps
  - View System
    - provides the user interface (UI) primitives as well as UI Events
  - Location Manager
    - allows developers to tap into location-based services, whether by GPS, cell tower IDs, or local Wi-Fi databases
  - Notification Manager
    - manages events, such as arriving messages and appointments
  - XMPP
    - provides standardized messaging functions between applications

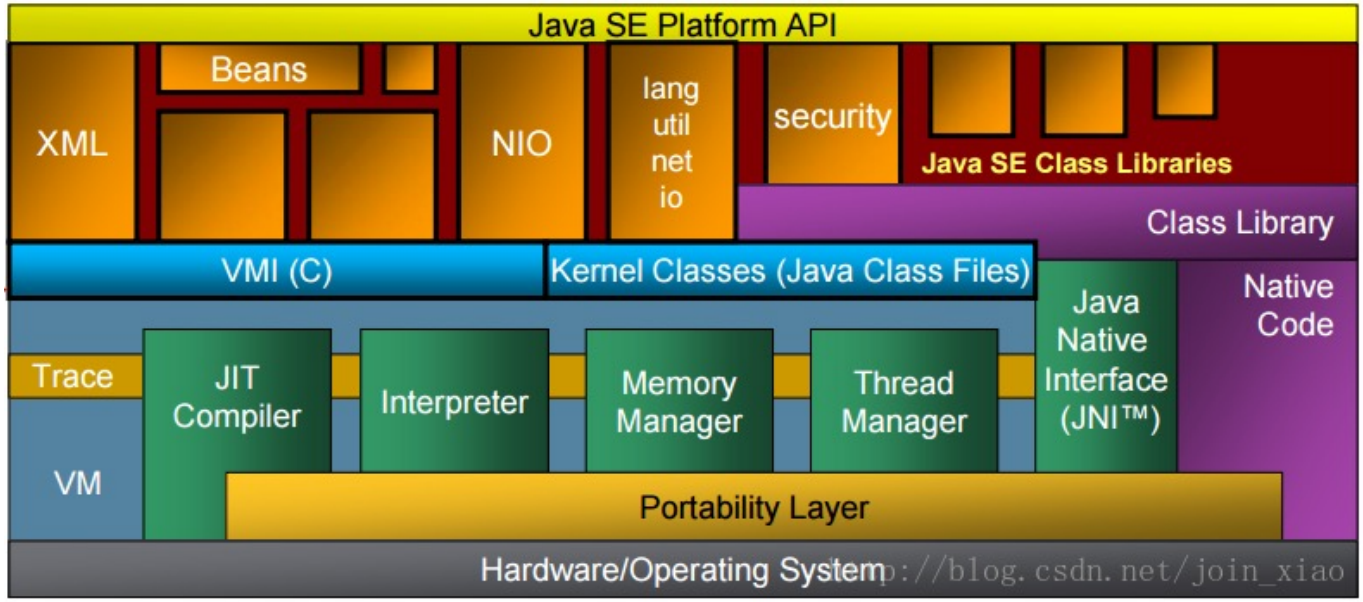


Android 系统库

- 系统组件是由C/C++编写的实用系统函数集，可以被android系统的各个组件使用。
- 应用可通过Java接口来对其进行调用，同时这些功能向开发者开放；
- 一些重要的系统库如下：
  - 界面管理器；
  - OpenGL
  - 媒体框架
  - SQL 数据库
  - Bro浏览器引擎
  - 放生Libc

Android 运行时

- Every Android application runs in its own process with its own instance of the Dalvik virtual machine (DVM);
- DVM executes files in the Dalvik Executable (.dex) format;
- Component includes a set of core libraries that provides most of the functionality available in the core libraries of the Java programming language;
- To execute an operation the DVM calls on the corresponding C/C++ library using the Java Native Interface (JNI);



Android 运行时

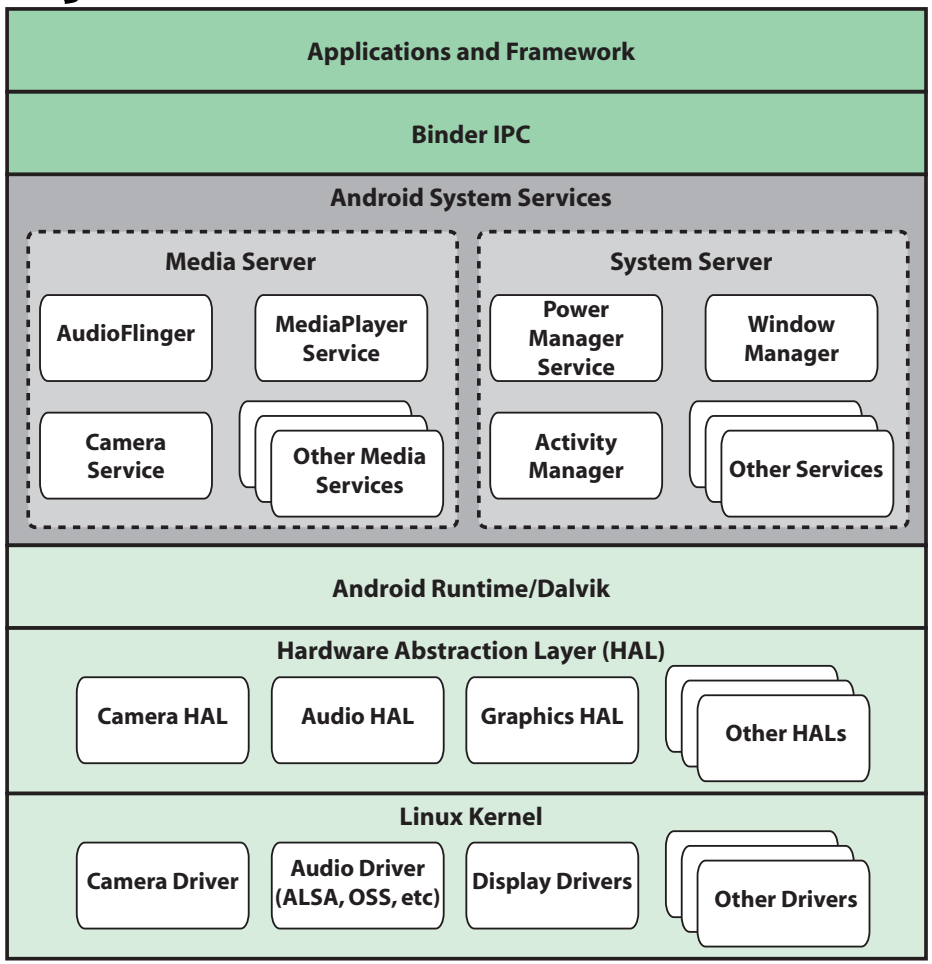
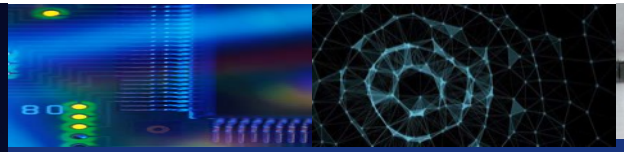


Figure 2.21 Android System Architecture

Android 活动（activity）

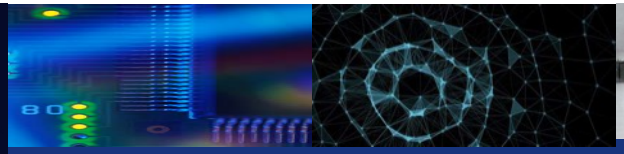
- An activity is a single visual user interface component, including things such as menu selections, icons, and checkboxes
- Every screen in an application is an extension of the Activity class
- Use Views to form graphical user interfaces that display information and respond to user actions



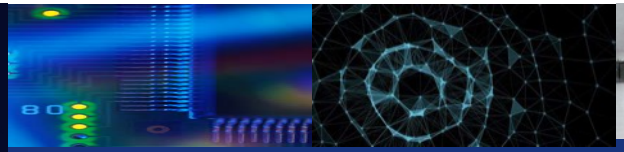
电源管理

- **Alarms**
- **Implemented in the Linux kernel and is visible to the app developer through the AlarmManager in the RunTime core libraries**
- **Is implemented in the kernel so that an alarm can trigger even if the system is in sleep mode**
  - this allows the system to go into sleep mode, saving power, even though there is a process that requires a wake up

- **Wakelocks**
- **Prevents an Android system from entering into sleep mode**
- **These locks are requested through the API whenever an application requires one of the managed peripherals to remain powered on**
- **An application can hold one of the following wakelocks:**
  - Full_Wake_Lock
  - Partial_Wake_Lock
  - Screen_Dim_Wake_Lock
  - Screen_Bright_Wake_Lock



6. 操作系统启动、调试



构建和引导操作系统

- 操作系统通常设计为在一类具有各种外围设备的系统上运行
- 通常，购买的计算机上已经安装了操作系统
  - 但是可以构建和安装其他一些操作系统
  - 如果从头开始生成操作系统
    - 编写操作系统源代码
    - 为将在其上运行的系统配置操作系统
    - 编译操作系统
    - 安装操作系统
    - 启动计算机及其新操作系统

构建和引导Linux

- 下载Linux源代码(http://www.kernel.org)
- 通过“makemenuconfig”配置内核
- 使用“make”编译内核
  - 生成vmlinuz，内核映像
  - 通过“生成模块”编译内核模块
  - 通过“makemodules\u Install”将内核模块安装到vmlinuz中
  - 通过“makeinstall”在系统上安装新内核

系统启动

- 在系统上初始化电源时，执行从固定内存位置开始
- 操作系统必须对硬件可用，以便硬件可以启动它
  - 一小段代码——存储在ROM或EEPROM中的引导加载程序、BIOS定位内核，将其加载到内存中，然后启动它
  - 有时是两步过程，其中引导块在固定位置由ROM代码加载，从磁盘加载引导加载程序
  - 现代系统用统一可扩展固件接口（UEFI）取代BIOS
- 通用引导加载程序GRUB允许从多个磁盘、版本和内核选项中选择内核
- 内核加载，然后系统运行
- 引导加载程序通常允许各种引导状态，例如单用户模式

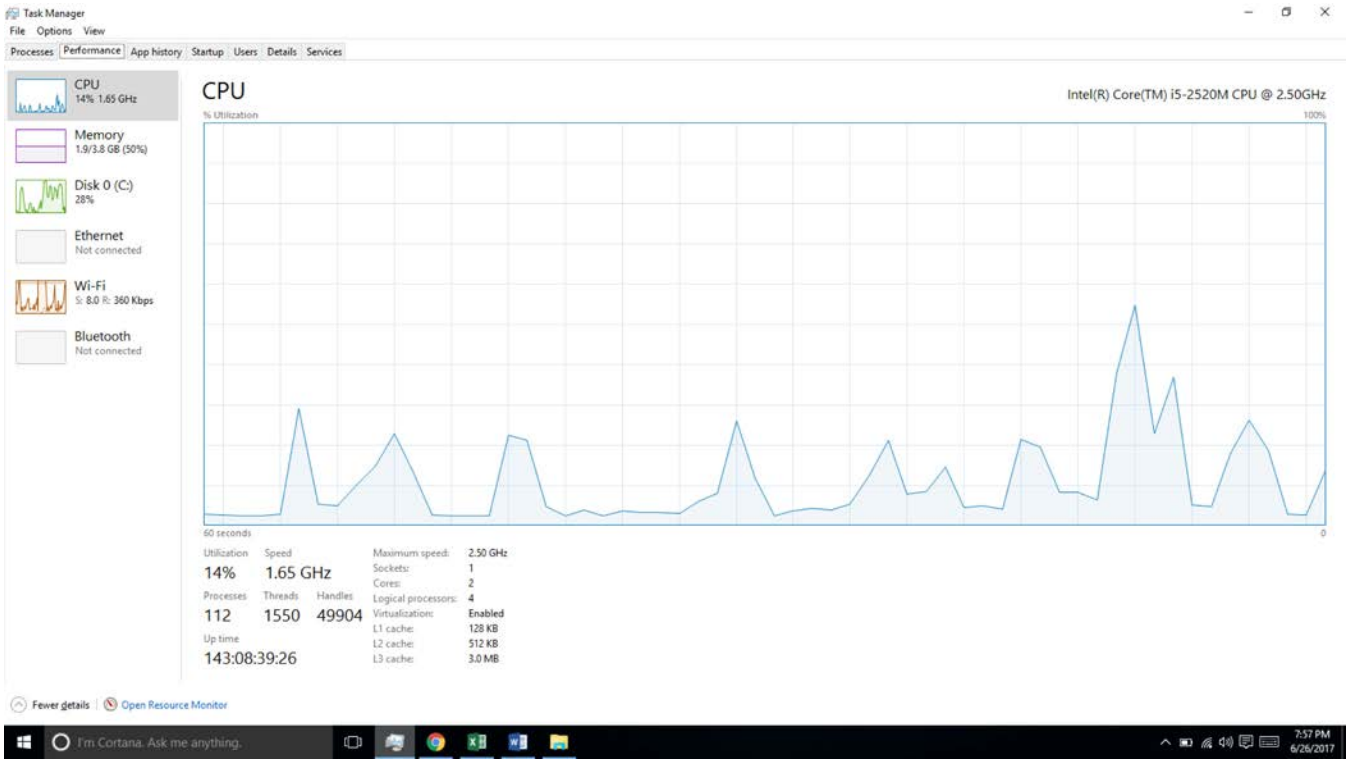
操作系统调试

- 调试是查找和修复错误或bug
- 还有性能调整
- 操作系统生成包含错误信息的日志文件
- 应用程序故障会生成核心转储文件，捕获进程内存
- 操作系统故障可能会生成包含内核内存的崩溃转储文件
- 除了崩溃之外，性能调优还可以优化系统性能
  - 有时使用活动的跟踪列表，记录以供分析
  - 评测是对指令指针进行周期性采样，以查找统计趋势

Kernighan定律：“调试的难度是一开始编写代码的两倍。因此，如果你尽可能巧妙地编写代码，根据定义，你就没有足够的智慧来调试它。”

性能调整

- 通过消除瓶颈提高性能
- 操作系统必须提供计算和显示系统行为度量的方法
- 例如，“top”程序或Windows任务管理器



Trace

- 收集特定事件的数据，例如系统调用中涉及的步骤
- 工具包括
  - strace－跟踪进程调用的系统调用
  - gdb－源代码级调试器
  - perf－Linux性能工具的集合
  - tcpdump－收集网络数据包
  - eBPF—内核追踪
  - Dtrace—Sun OS追踪；

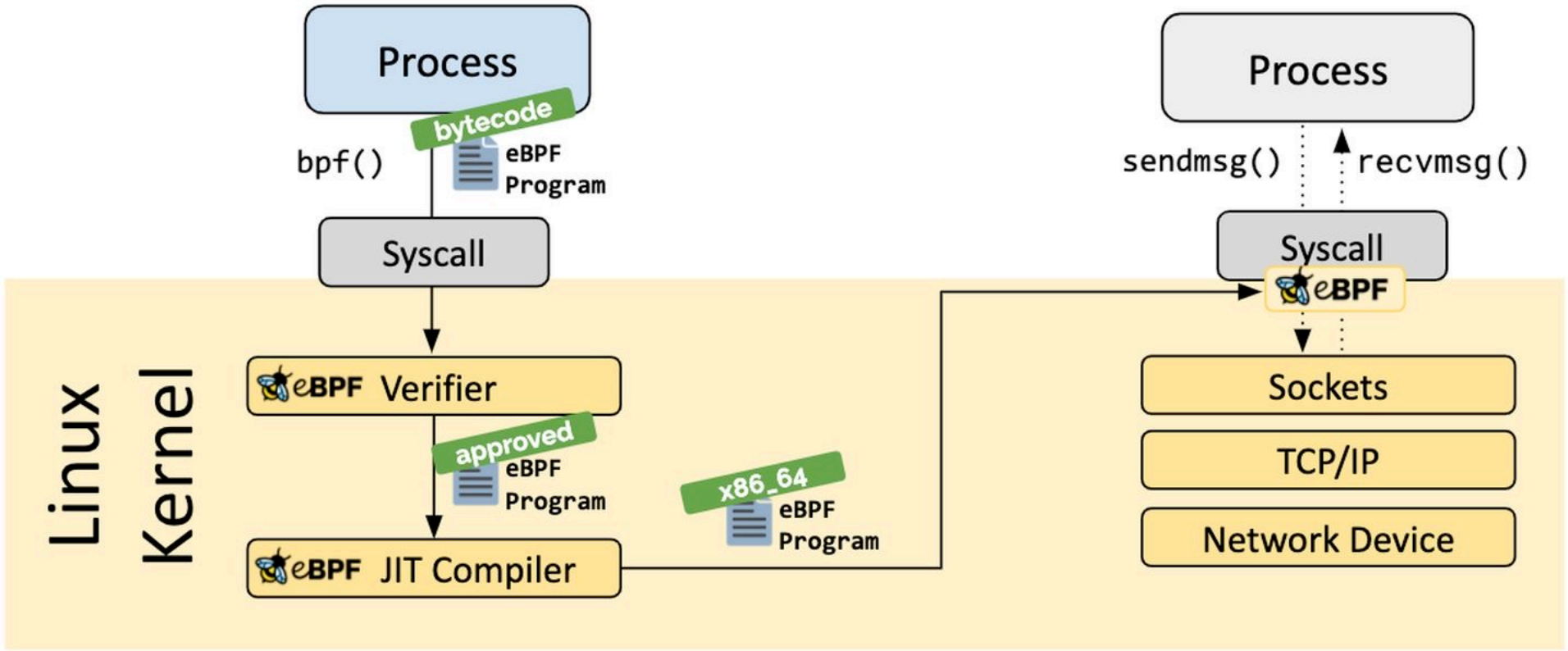
Trace

- 调试用户级和内核代码之间的交互几乎是不可能的，因为没有一个工具集可以理解用户级和内核代码之间的交互，也没有一个工具可以理解它们的行为
- BCC（BPF编译器集合）是一个为Linux提供跟踪功能的丰富工具包
  - 另请参见原始DTrace
- 例如，disksnoop。py跟踪磁盘I/O活动

TIME(s)	T	BYTES	LAT(ms)
1946.29186700	R	8	0.27
1946.33965000	R	8	0.26
1948.34585000	W	8192	0.96
1950.43251000	R	4096	0.56
1951.74121000	R	4096	0.35

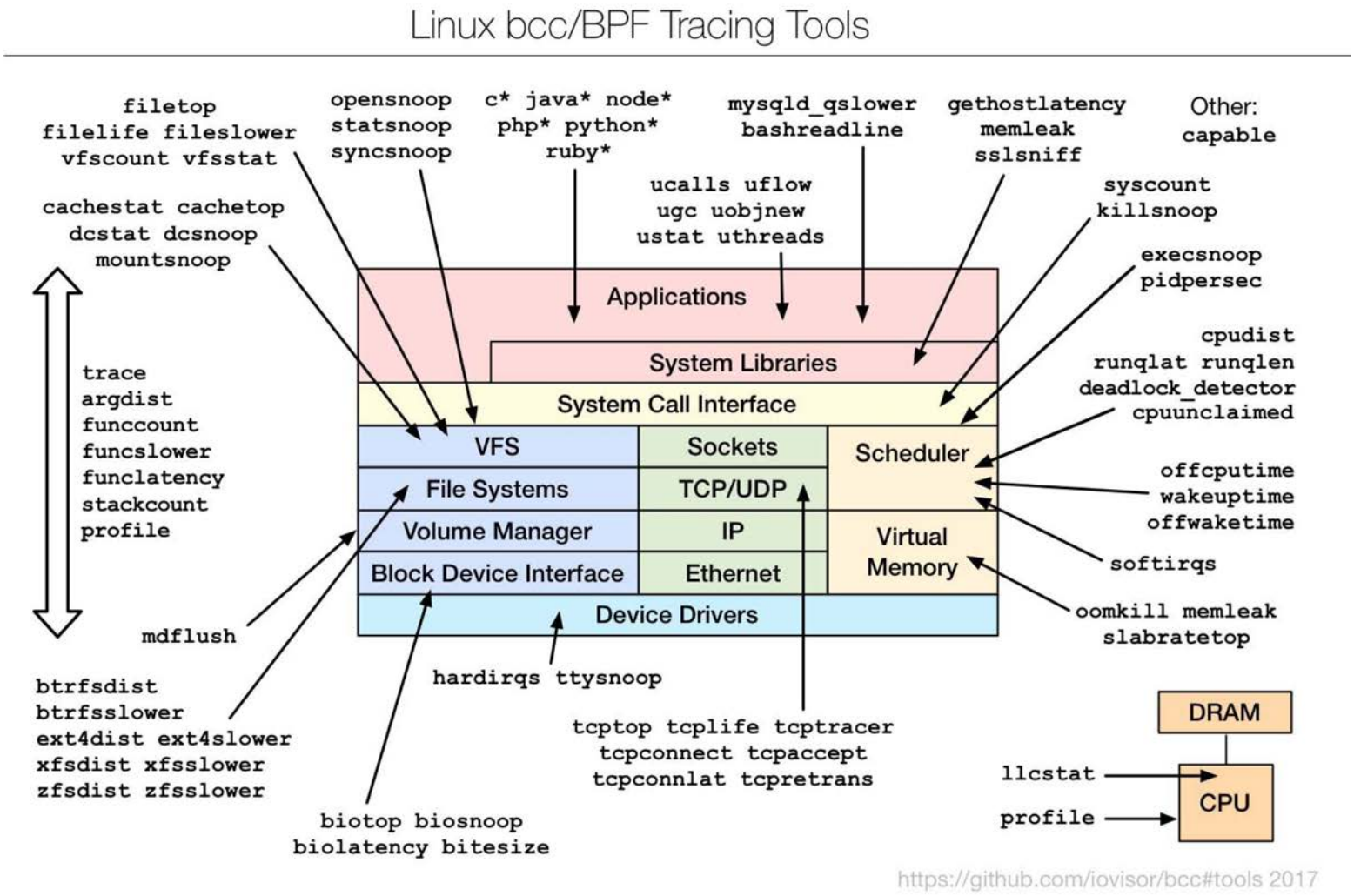
- 许多其他工具（下一张幻灯片）

eBPF


Process
bytecode
bpf()
eBPF
Program
Syscall
eBPF Verifier
approved
eBPF
Program
x86_64
eBPF
Program
eBPF JIT Compiler
Linux
Kernel
Process
sendmsg()
recvmsg()
Syscall
eBPF
Sockets
TCP/IP
Network Device

Demo

eBPF


Linux bcc/BPF Tracing Tools
filetop
filelife fileslower
vfscount vfsstat
opensnoop
statsnoop
syncsnoop
c* java* node*
php* python*
ruby*
mysqld_qslower
bashreadline
gethostlatency
memleak
sslsniff
Other:
capable
cachestat cachetop
dcstat dcsnoop
mountsnoop
ucalls uflow
ugc uobjnew
ustat uthreads
syscount
killsnoop
execsnoop
pidpersec
Applications
System Libraries
System Call Interface
VFS
Sockets
Scheduler
File Systems
TCP/UDP
Volume Manager
IP
Virtual Memory
Block Device Interface
Ethernet
Device Drivers
trace
argdist
funccount
funcslower
funclatency
stackcount
profile
cpudist
runqlat runqlen
deadlock_detector
cpuunclaimed
offcputime
wakeuptime
offwaketime
softirqs
oomkill memleak
slabratetop
mdflush
btrfsdist
btrfsslower
ext4dist ext4slower
xfsdist xfsslower
zfsdist zfsslower
hardirqs ttysnoop
tcptop tcplife tcptracer
tcpconnect tcpaccept
tcpconnlat tcpretrans
biotop biosnoop
biolatency bitesize
DRAM
llcstat
CPU
profile
https://github.com/iovisor/bcc#tools 2017

谢谢