IPADS
INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

从应用视角看操作系统抽象

上海交通大学

https://www.sjtu.edu.cn

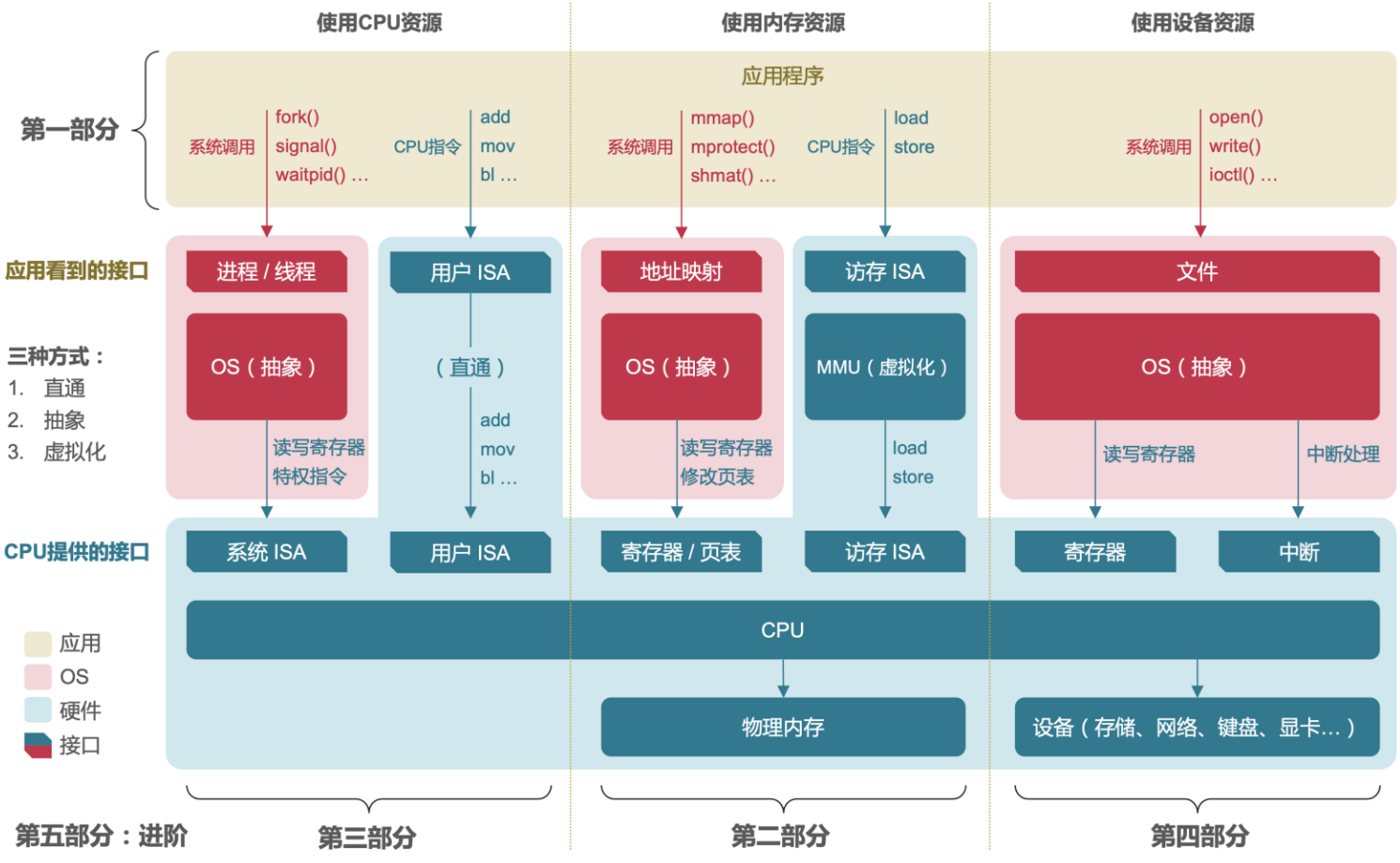
版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源
  - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：https://creativecommons.org/licenses/by/4.0/legalcode

回顾

- **特权级EL0、EL1**
- **特权级切换**
  - 同步异常与异步异常（中断）
  - 异常处理
    - 异常处理函数表
    - 系统寄存器：vbar_el1、esr_el1、elr_el1
    - 栈切换：sp_el1和sp_el0
  - 切换过程：硬件（CPU）和软件（操作系统）的分工
- **系统调用（system call）**
  - 一种特殊的同步异常
  - svc + eret

回顾


使用CPU资源
使用内存资源
使用设备资源
第一部分
应用程序
系统调用
fork()
signal()
waitpid() …
CPU指令
add
mov
bl …
系统调用
mmap()
mprotect()
shmat() …
CPU指令
load
store
系统调用
open()
write()
ioctl() …
应用看到的接口
进程 / 线程
用户 ISA
地址映射
访存 ISA
文件
三种方式：
1. 直通
2. 抽象
3. 虚拟化
OS（抽象）
（直通）
OS（抽象）
MMU（虚拟化）
OS（抽象）
读写寄存器
特权指令
add
mov
bl …
读写寄存器
修改页表
load
store
读写寄存器
中断处理
CPU提供的接口
系统 ISA
用户 ISA
寄存器 / 页表
访存 ISA
寄存器
中断
CPU
应用
OS
硬件
接口
物理内存
设备（存储、网络、键盘、显卡…）
第五部分：进阶
第三部分
第二部分
第四部分

进程

分时复用有限的CPU资源

硬件概览：

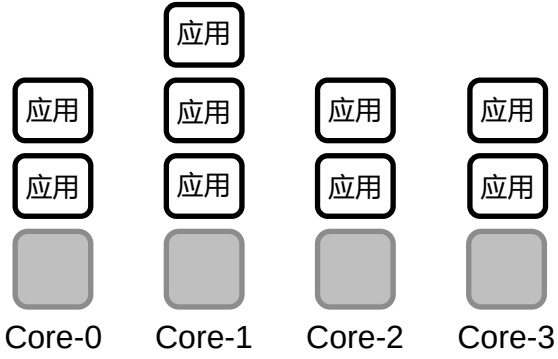
型号名称：	MacBook Air
型号标识符：	Mac14,2
芯片：	Apple M2
核总数：	8（4性能和4能效）
内存：	24 GB
系统固件版本：	7459.141.1
操作系统加载程序版本：	7459.141.1
序列号（系统）：	Q21545QKVG
硬件UUID：	A50258A0-30AA-5D36-ABA9-C8200CFF60DF
预置UDID：	00008112-001E09E83C21401E
激活锁状态：	已启用

- CPU核心数量少于应用程序数量，如何运行？
- 单个CPU核心如何运行多个应用程序？

- **类比运动场地**（分时复用）
  - 气膜的羽毛球场地数量有限（CPU核心数量）
  - 全校的师生都想去打羽毛球（需要运行的应用程序很多）
  - 由气膜管理系统安排师生在不同时间段使用场地：大家都能打上球

分时复用有限的CPU资源

- **分时复用CPU**
  - 让多个应用程序**轮流**使用处理器核心
  - 何时切换：**操作系统**决定
    - 运行时间片（例如100ms）
  - 高频切换：看起来是多个应用“同时”执行



进程：操作系统对于应用程序的表示

- **通常一个应用程序对应一个进程**
  - 在shell中输入可执行文件的名称
    - shell创建新进程，可执行文件在新进程中执行
  - 在图形界面双击应用图标
- **多进程程序：应用程序亦可自行创建新进程**
  - 创建新进程，再在新进程中运行其他应用程序或与自己一样的程序

进程在操作系统中的实现

- **操作系统提供进程的抽象用于管理应用程序**
  - 进程标识号（Process ID， PID）
  - 运行状态
    - **处理器上下文（CPU Context）**
  - 地址空间
  - 打开的文件

程序员视角看进程

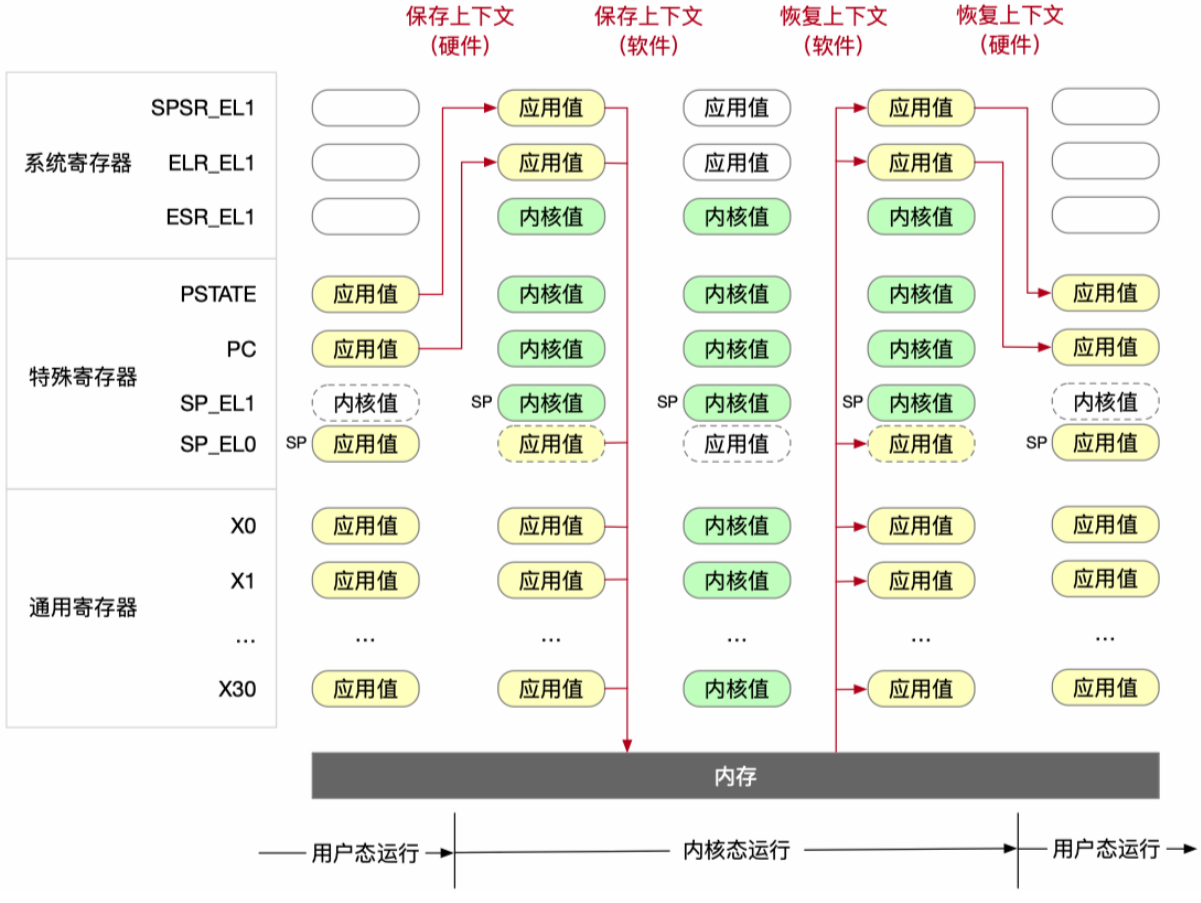
- **进程抽象为应用程序提供了“独占CPU”的假象**
  - 程序开发不用考虑如何与其他程序共享CPU
  - 简化编程
- **进程相关的系统调用**
  - 创建进程
  - 让进程执行指定的程序
  - 退出进程
  - 进程间通信

进程切换

处理器上下文(CPU Context)

- **操作系统为每个进程维护处理器上下文**
  - 包含恢复进程执行所需要的状态
  - 思考：进程A执行到任意一条指令（EI0），切换到进程B执行，一段时间后，再切回到进程A执行
    - 为完成此过程，有哪些状态需要保存?
  - 具体包括：
    - **PC寄存器值，栈寄存器值，通用寄存器值，状态寄存器值**

回顾：用户态/内核态切换时的处理器状态变化

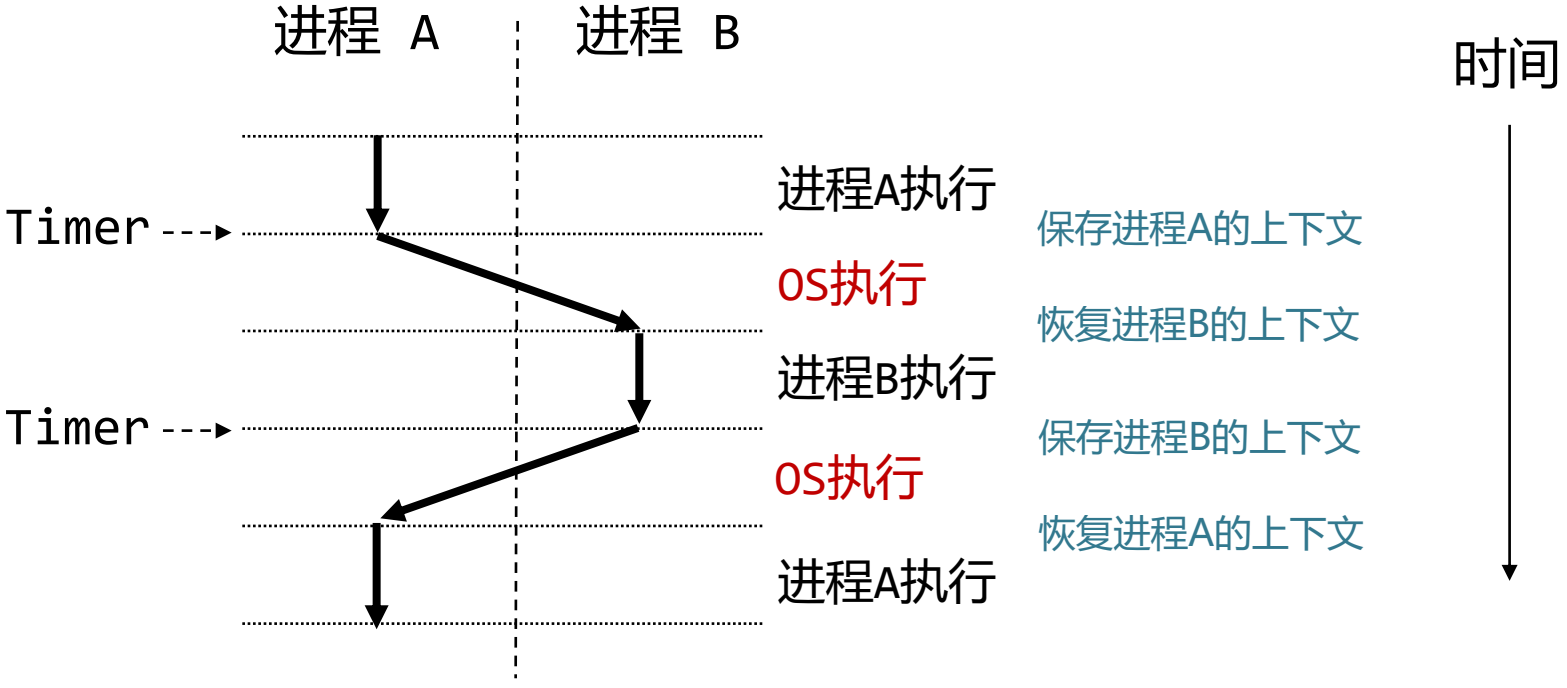


进程切换的时机

- **异常导致的上下文切换**
  - Timer中断（如基于时间片的多任务调度）

- **用户执行系统调用并进入内核**
  - 如：read/sleep等会导致进程阻塞的系统调用
  - 即使系统调用不阻塞执行，内核也可以决定执行上下文切换，而不是将控制权返回给调用进程

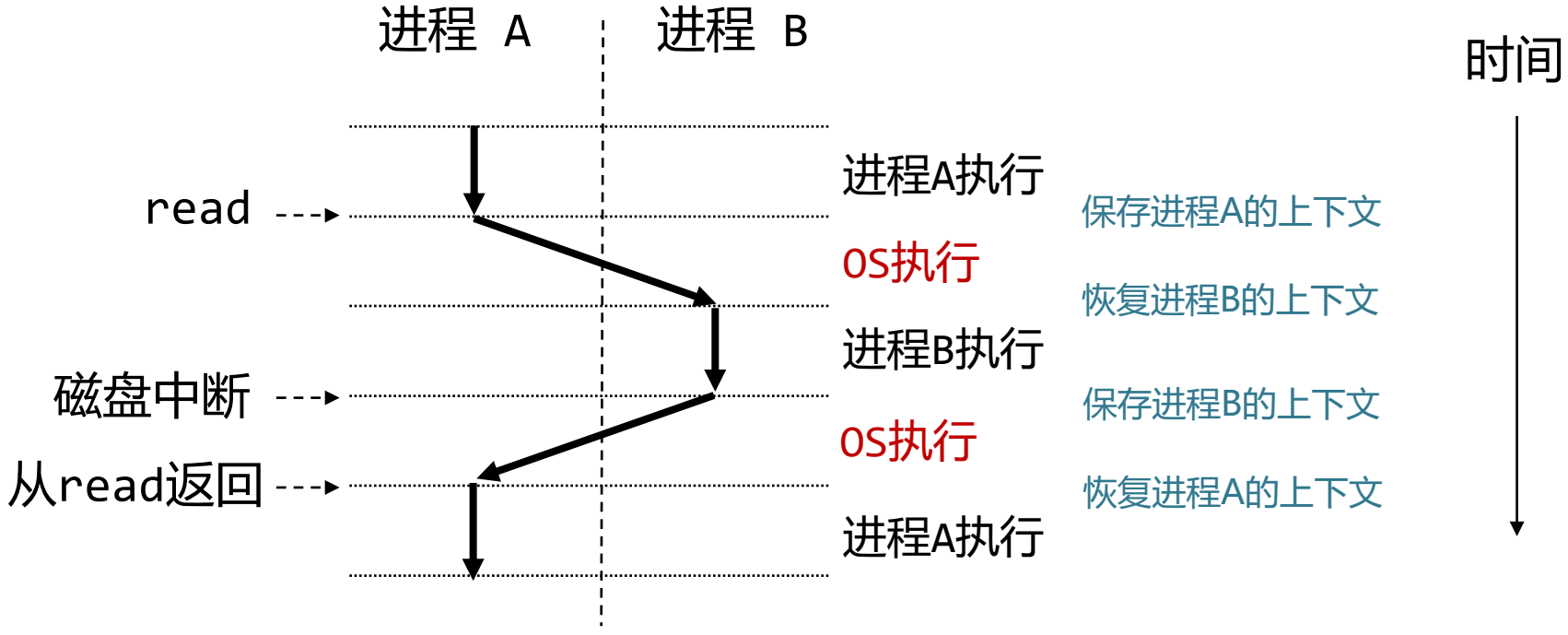
进程切换示例一



Q：Timer中断可能在进程运行过程的任意时刻抵达，应用进程都能切换吗？

Q：如果（异步）异常发生在OS运行过程中/保存上下文的过程中，会怎么样？

进程切换示例二


进程 A
进程 B
时间
read
磁盘中断
从read返回
进程A执行
OS执行
进程B执行
OS执行
进程A执行
保存进程A的上下文
恢复进程B的上下文
保存进程B的上下文
恢复进程A的上下文

常见的进程相关的接口

获取进程 ID

- **进程 ID**
  - 每个进程都有唯一的正数PID
- **getpid()**
  - 返回调用进程的PID
- **getppid()**
  - 返回调用进程父进程的PID
  - 父进程：创建该进程的进程

获取进程 ID

```
#include <unistd.h>
#include <sys/types.h>
pid_t getpid(void);
pid_t getppid(void);
```

返回值：调用进程或父进程的PID

- **getpid和getppid函数返回pid_t类型的int值**
- **pid_t**
  - Linux系统中，在types.h文件中定义为int

Exit 函数

```
#include <stdlib.h>
void exit(int status);
```

这个函数没有返回值

- exit函数终止进程并带上一个status状态

Fork 函数

- 父进程调用fork函数创建新的子进程

```
#include <unistd.h>
#include <sys/types.h>
pid_t fork(void);

        返回值：子进程为 0，父进程为子进程 PID，出错为 -1
```

Fork 函数

- **调用一次**
  - 在父进程中
- **返回两次**
  - 在父进程中，返回子进程的PID
  - 在子进程中，返回0
- **返回值提供了唯一明确地区分父进程和子进程执行的方法**

Fork 函数

```
1.  #include "csapp.h"
2.  int main()
3.  {
4.      pid_t pid;
5.      int x = 1;
6.      pid = fork();
7.      if (pid == 0) { /* child */
8.              printf("child : x=%d\n", ++x);
9.              exit(0);
10.     }
11.     /* parent */
12.     printf("parent: x=%d\n", --x);
13.     exit(0);
14. }
```

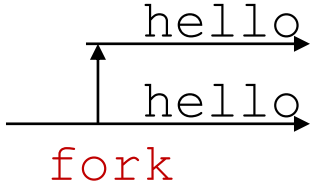
小思考：打印几行输出?

```
1. #include "csapp.h"
2. int main()
3. {
4.     fork();
5.     printf("hello!\n");
6.     exit(0);
7. }
```

(a) 调用一次 fork

小思考：打印几行输出?

```
1. #include "csapp.h"
2. int main()
3. {
4.     fork();
5.     printf("hello!\n");
6.     exit(0);
7. }
```



(a) 调用一次 fork

(b) 打印两行输出

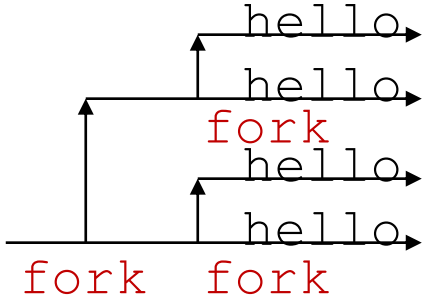
小思考：打印几行输出?

```
1. #include "csapp.h"
2. int main()
3. {
4.     fork();
5.     fork();
6.     printf("hello!\n");
7.     exit(0);
8. }
```

(a) 调用二次 fork

小思考：打印几行输出?

```
1. #include "csapp.h"
2. int main()
3. {
4.     fork();
5.     fork();
6.     printf("hello!\n");
7.     exit(0);
8. }
```



(a) 调用二次 fork

(b) 打印四行输出

Execve函数

```
#include <unistd.h>
int execve(const char *filename,
           const char *argv[], const char *envp[]);
```

返回值：成功(不返回），失败(-1)

- **加载和运行**
  - filename:可执行文件名;argv:参数列表，envp:环境变量列表
- **execve 只调用一次，且永远不会返回**
  - 仅仅在运行报错的时候，返回调用程序
  - 例：找不到filename标识的文件

Linux下的僵尸进程

- **进程终止后，内核不会立刻销毁该进程**
  - 不再运行，但仍然占用内存资源
- **进程以终止态存在，等待父进程回收**
- **当父进程回收终止的子进程**
  - 内核把子进程的exit状态传递给父进程
  - 内核移除子进程，此时子进程才被真正回收
- **终止状态下还未被回收的进程就是僵尸进程**

```
1198908 pts/32   00:00:00 sample
1198909 pts/32   00:00:00 sample <defunct>
```

Linux下的僵尸进程

- **如果父进程**
  - 在自己终止前没有回收僵尸子进程
  - 内核会安排init进程回收这些子进程
- **init进程**
  - PID为1
  - 在系统初始化时由内核创建

waitpid函数

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

返回值：成功返回子进程 PID，出错返回 -1

waitpid函数

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- pid>0：等待集合中只有pid子进程
- pid=-1：等待集合包括所有子进程
- 如果没有子进程：返回-1，errno = ECHILD
- 如果等待被中断：返回-1，errno = EINTR

waitpid函数

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- **options=0**
  - 挂起调用进程，等待集合中任意子进程终止
  - 如果等待集合中有子进程在函数调用前已经终止，立刻返回
  - 返回值是导致函数返回的终止子进程pid
  - 该终止子进程被内核回收

waitpid函数

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- **带回被回收子线程的exit状态**
  - status指针不为NULL
  - status包含导致子进程进入终止状态的信息
  - wait.h文件包含了若干宏定义，用于解释status

waitpid函数

```
1 #include "csapp.h"
2 #define N 2
3
4 int main()
5 {
6   int status, i;
7   pid_t pid;
8
9   /* Parent creates N children */
10  for (i = 0; i < N; i++)
11    if ((pid = fork()) == 0) /* child */
12      exit(100+i);
13
```

waitpid函数

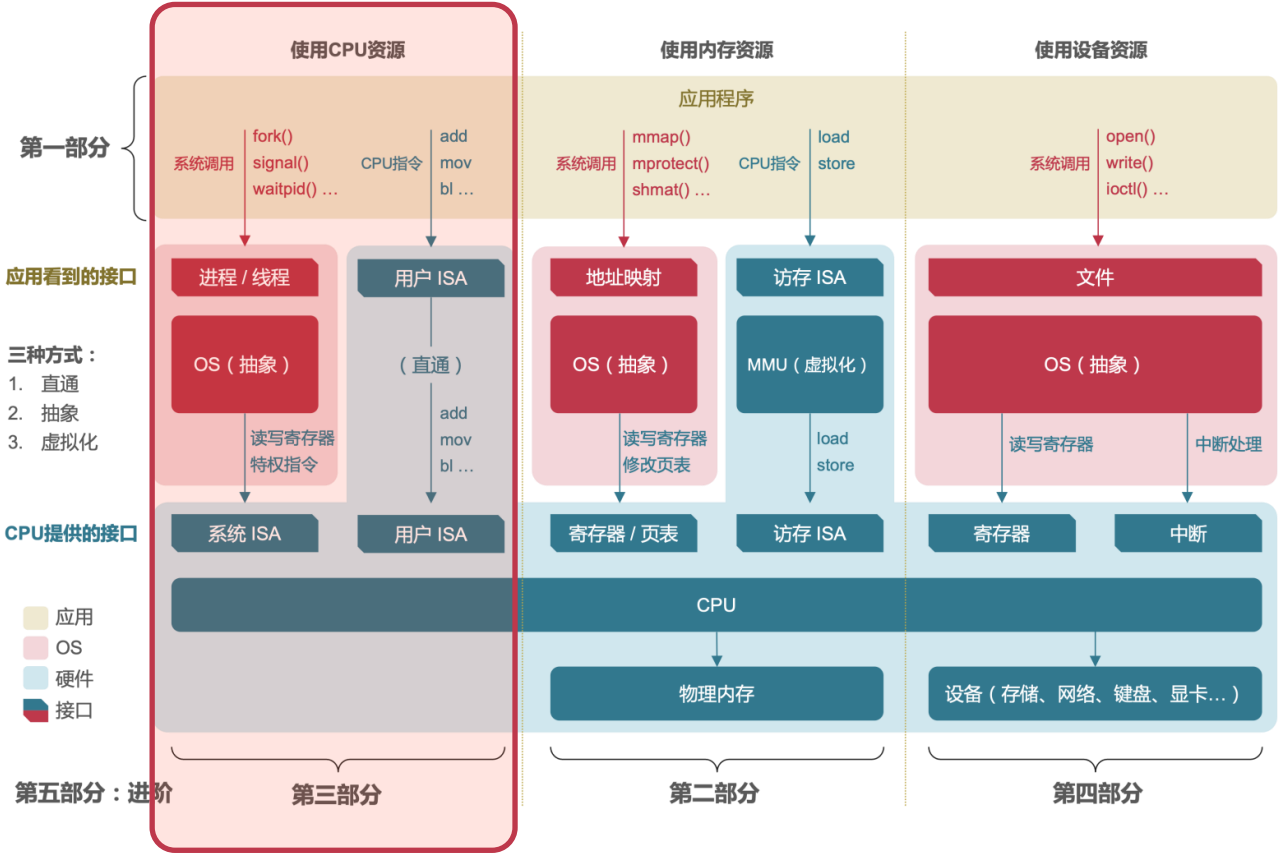
```
14   /* Parent reaps N chds. in no particular order */
15   while ((pid = waitpid(-1, &status, 0)) > 0) {
16     if (WIFEXITED(status))
17       printf("child %d terminated normally with exit
18               status=%d\n", pid, WEXITSTATUS(status));
19     else
20       printf("child %d terminated abnormally\n", pid);
21   }
22   /* The only normal term. is if there no more chds. */
23   if (errno != ECHILD)
24     unix_error("waitpid error");
25   exit(0);
26 }
```

思考：输出结果是什么?

```
unix>./waitpid1
child 22966 terminated normally with exit status=100
child 22967 terminated normally with exit status=101

unix>./waitpid1
child 22967 terminated normally with exit status=101
child 22966 terminated normally with exit status=100
```

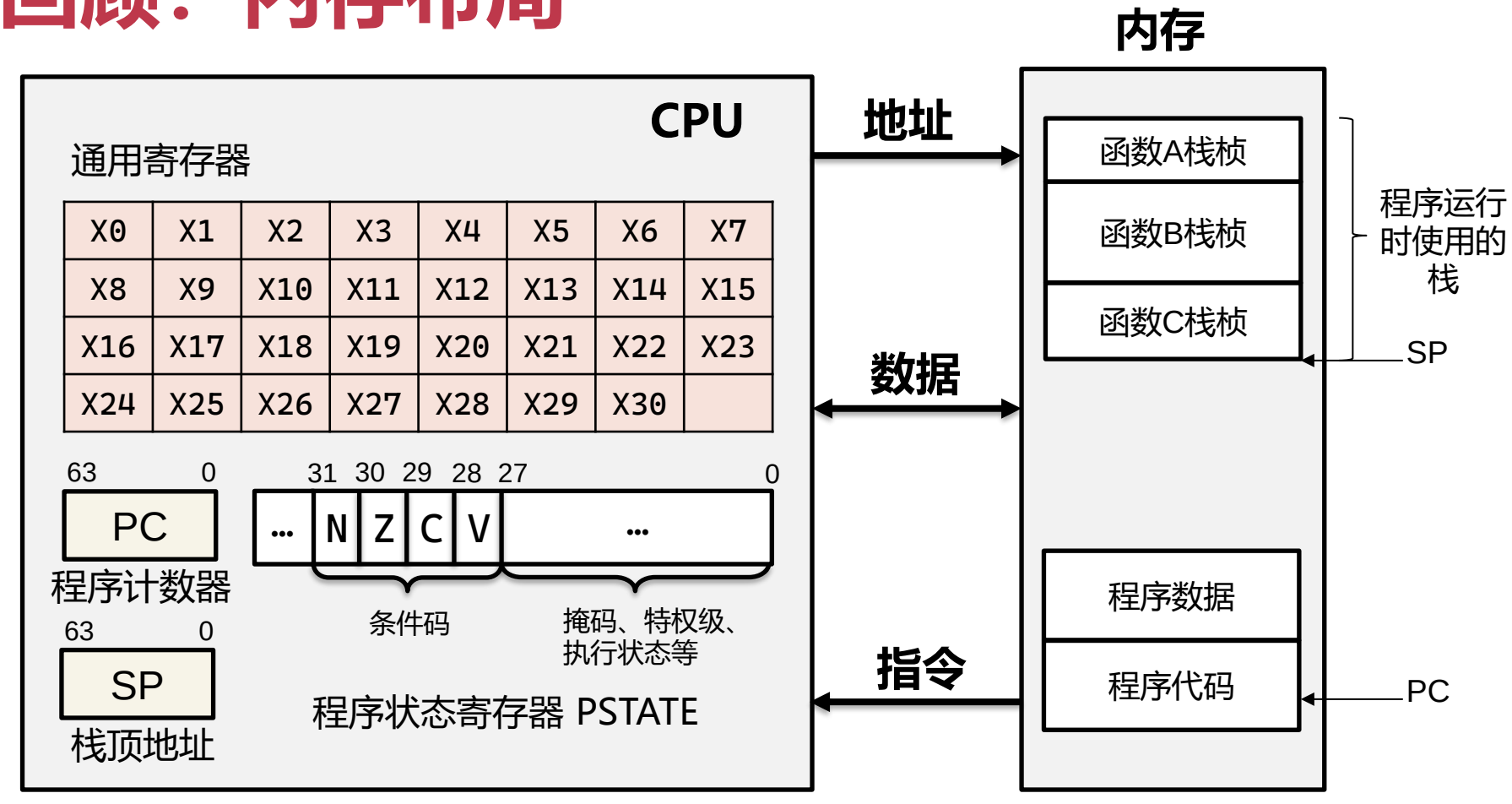
子进程回收顺序不确定，都有可能

应用程序使用CPU资源（直通+抽象）

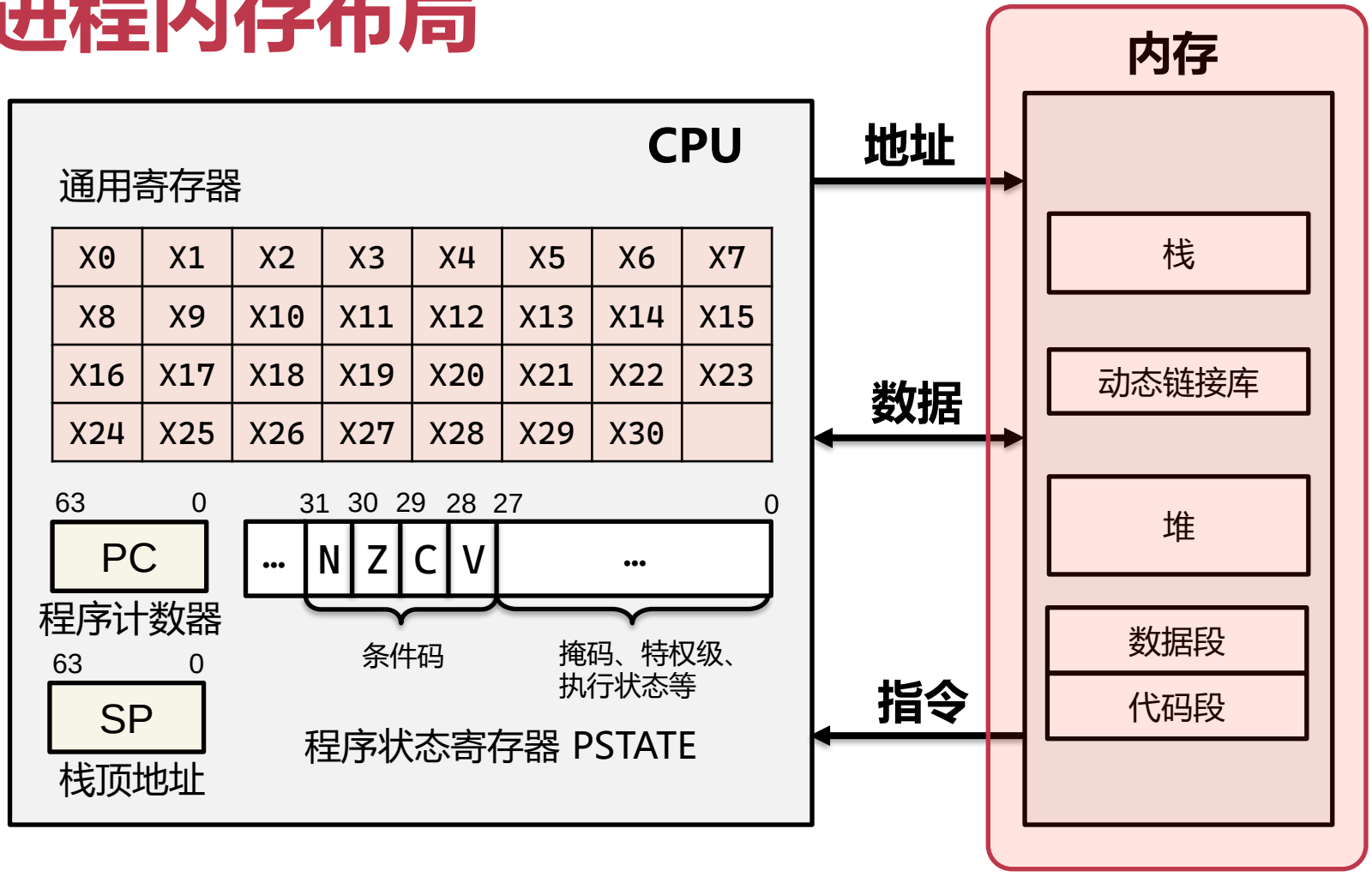


内存

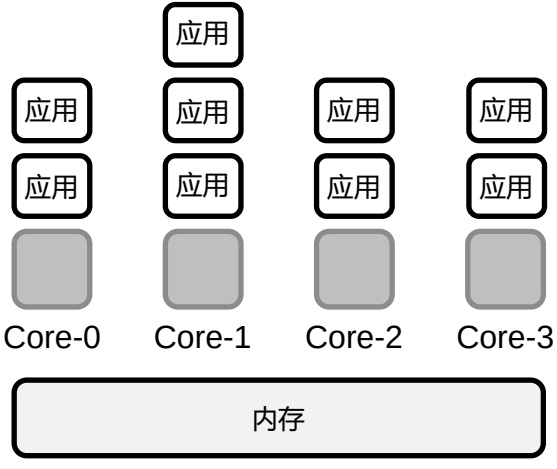
回顾：内存布局


内存
CPU
通用寄存器
X0 X1 X2 X3 X4 X5 X6 X7
X8 X9 X10 X11 X12 X13 X14 X15
X16 X17 X18 X19 X20 X21 X22 X23
X24 X25 X26 X27 X28 X29 X30
63 0
PC
程序计数器
63 0
SP
栈顶地址
31 30 29 28 27 0
… N Z C V …
条件码
掩码、特权级、
执行状态等
程序状态寄存器 PSTATE
地址
数据
指令
函数A栈桢
函数B栈桢
函数C栈桢
程序运行时使用的栈
SP
程序数据
程序代码
PC

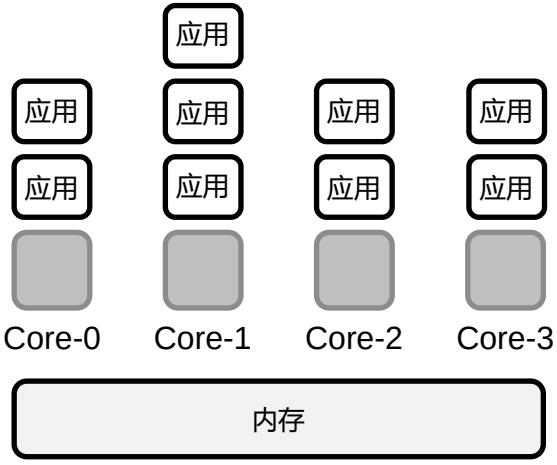
进程内存布局

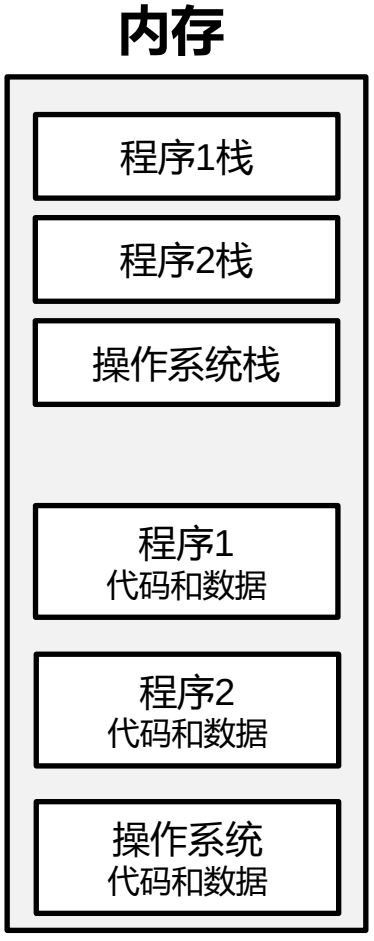

CPU
通用寄存器
X0 X1 X2 X3 X4 X5 X6 X7
X8 X9 X10 X11 X12 X13 X14 X15
X16 X17 X18 X19 X20 X21 X22 X23
X24 X25 X26 X27 X28 X29 X30
63 0
PC
程序计数器
63 0
SP
栈顶地址
31 30 29 28 27 0
… N Z C V …
条件码
掩码、特权级、
执行状态等
程序状态寄存器 PSTATE
地址
数据
指令
内存
栈
动态链接库
堆
数据段
代码段

思考：多个进程如何使用内存资源



多个进程如何使用内存资源





思路1：协商划分，类比校园土地划分

- 交大5000亩土地（8G内存）
- 若干学院（若干应用进程）
- 根据学院规模占用校园土地（根据应用规模各占一部分内存）

协商划分方案的问题

硬件概览：

型号名称：	MacBook Air
型号标识符：	Mac14,2
芯片：	Apple M2
核总数：	8（4性能和4能效）
内存：	24 GB
系统固件版本：	7459.141.1
操作系统加载程序版本：	7459.141.1
序列号（系统）：	Q21545QKVG
硬件UUID：	A50258A0-30AA-5D36-ABA9-C8200CFF60DF
预置UDID：	00008112-001E09E83C21401E
激活锁状态：	已启用

思考：存在问题：

1. 内存大小无法满足所有应用进程的内存需求总量
2. 不同应用进程之间缺乏隔离（安全性低、容错性差）
3. 程序编写编译的难度

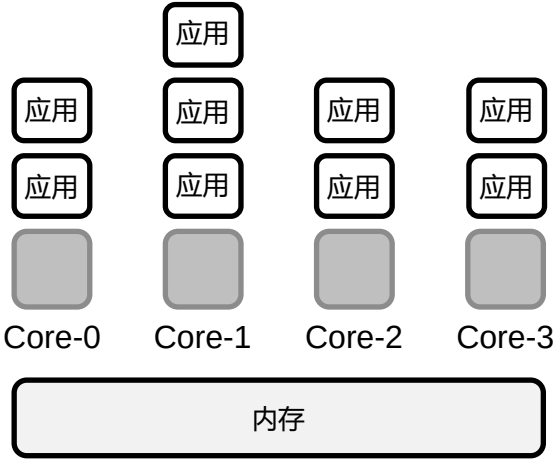
思考：程序中的地址是哪里来的?

```c
// main.c中的C代码
void multstore(long, long, long*);

int main() {
   long d;
   multstore(2, 3, &d);
   printf( “2 * 3 --> %d\n”, d);
   return 0;
}
```

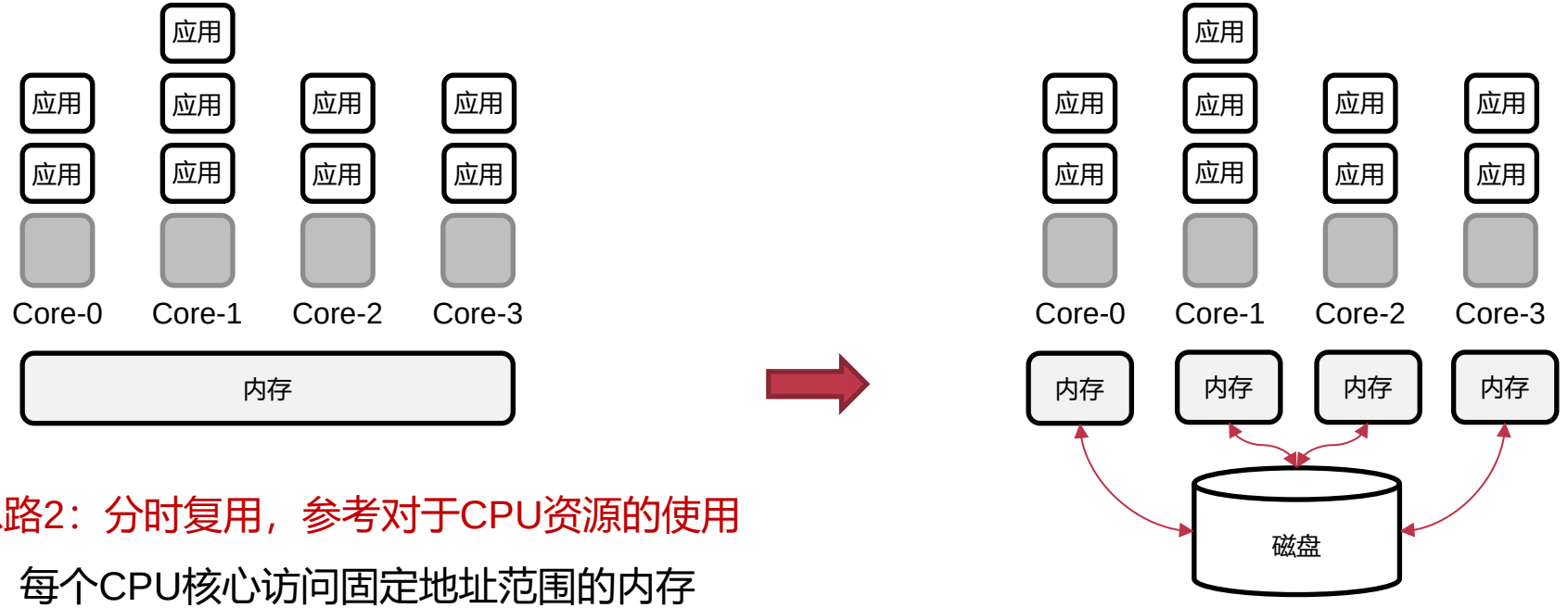
```
000000000040046c <multstore>:
 40046c:  a9be7bfd   stp  x29, x30, [sp, #-32]!
 400470:  910003fd   mov  x29, sp
 400474:  f9000bf3   str  x19, [sp, #16]
 400478:  aa0203f3   mov  x19, x2
 40047c:  94000012   bl   7ec <mult2>
 400480:  f9000260   str  x0, [x19]
 400484:  f9400bf3   ldr  x19, [sp, #16]
 400488:  a8c27bfd   ldp  x29, x30, [sp], #32
 40048c:  d65f03c0   ret
```

这些地址，在C代码中并没有，是从哪里来的？ **编译器生成**

再次思考：多个进程如何使用内存资源



再次思考：多个进程如何使用内存资源



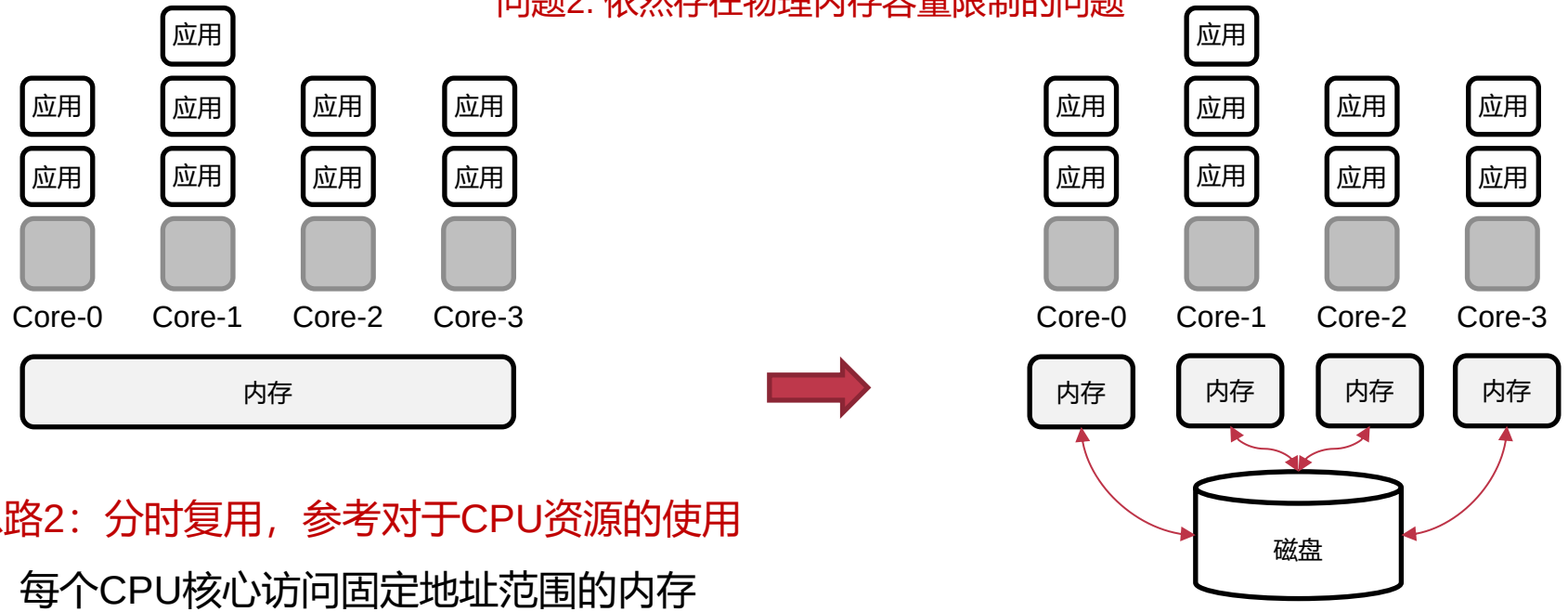
思路2：分时复用，参考对于CPU资源的使用

- 每个CPU核心访问固定地址范围的内存
- 任意时刻，一个CPU核心仅有一个应用进程执行，它能够使用该CPU核心对应的全部内存
- 进程切换时，将全部内存保存到磁盘上，并且从磁盘上恢复下一个执行进程的内存

分时复用方案的问题

问题1. 磁盘写入和读取会造成严重的性能损失

问题2. 依然存在物理内存容量限制的问题



思路2：分时复用，参考对于CPU资源的使用

- 每个CPU核心访问固定地址范围的内存
- 任意时刻，一个CPU核心仅有一个应用进程执行，它能够使用该CPU核心对应的全部内存
- 进程切换时，将全部内存保存到磁盘上，并且从磁盘上恢复下一个执行进程的内存

对内存资源使用的目标

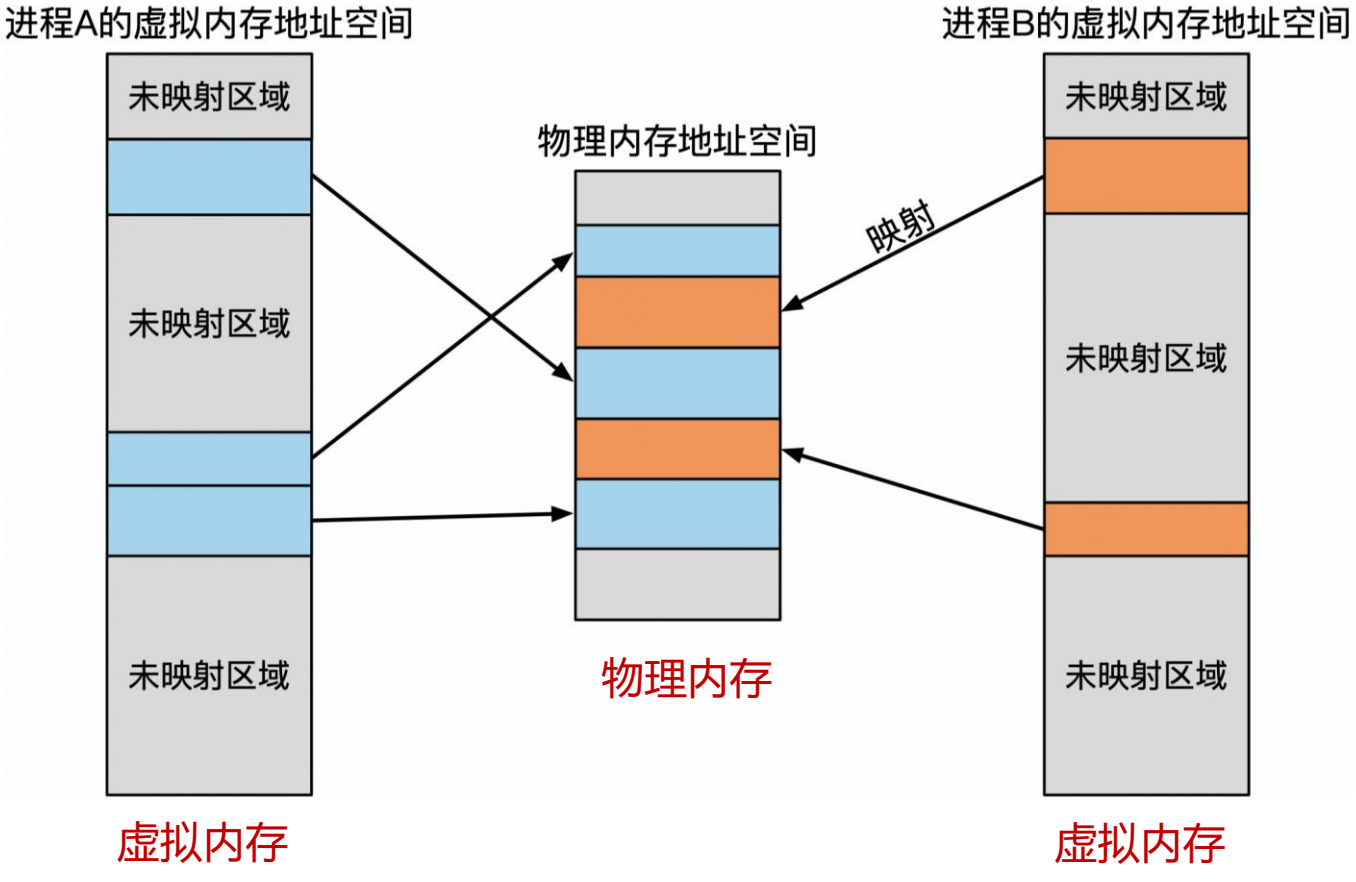
- **不同进程的内存地址空间具有独立性（彼此隔离）**
  - 进程不能直接使用物理地址访问内存（缺乏管控）
- **不同进程的内存地址空间具有统一性（易于开发）**
  - 每个进程看见的地址空间都是连续的、统一的
- **不同进程能够使用的内存总量可以超过物理内存总量**
  - 同时要保证性能开销较小

虚拟内存

- **虚拟地址空间**
  - 应用进程使用虚拟地址访问内存
  - 所有应用进程的虚拟地址空间都是统一的（方便开发）

- **地址翻译**
  - CPU按照OS配置的规则把虚拟地址翻译成物理地址
  - 翻译对于应用进程时不可见的（无需关心）

虚拟内存与物理内存


进程A的虚拟内存地址空间
未映射区域
未映射区域
未映射区域
虚拟内存
物理内存地址空间
物理内存
映射
进程B的虚拟内存地址空间
未映射区域
未映射区域
未映射区域
虚拟内存

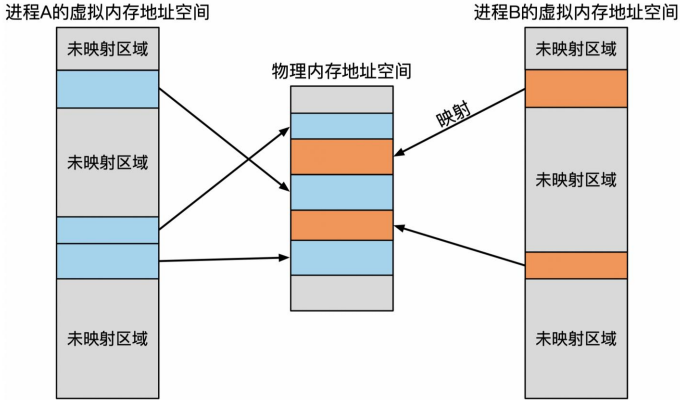
虚拟内存的优势：独立而统一的地址空间

```
tmac@intel12-pc:~/ieee-ai-os/abstractions$ cat vm1.c
#include <stdio.h>

int val1 = 1;

int main()
{
        printf("val1 memory addr: %p, memory value: %d\n", &val1, val1);
        while(1) {}
        return 0;
}
tmac@intel12-pc:~/ieee-ai-
#include <stdio.h>

int val2 = 2;

int main()
{
        printf("val2 memory addr: %p, memory value: %d\n", &val2, val2);
        while(1) {}
        return 0;
}
```

```
tmac@intel12-pc:~/ieee-ai-os/abstractions$ ./vm1 & ./vm2
[4] 1199812
val2 memory addr: 0x4c50f0, memory value: 2
val1 memory addr: 0x4c50f0, memory value: 1
```

虚拟内存的优势：突破物理内存容量限制

- **提高内存资源利用率**
  - 操作系统按需
    把进程虚拟地址映射到物理地址



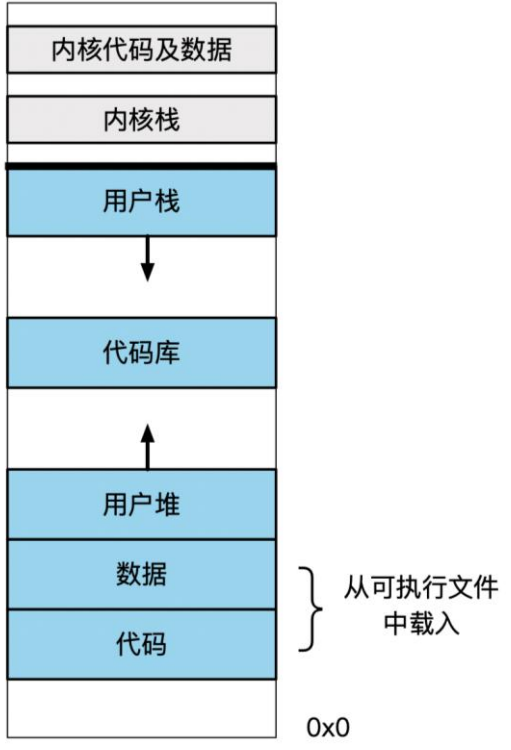
- **突破物理内存的容量限制**
  - 操作系统可以将部分虚拟内存区域
    数据暂存到磁盘上

Robert_Crovella Moderator 18 年 5 月

A pascal or volta GPU running in linux OS can have its memory “oversubscribed”. In that case, the GPU runtime will swap pages of memory as needed between host and device. In order to take advantage of this, the memory must be allocated with a managed allocator, such as cudaMallocManaged

[url]Programming Guide :: CUDA Toolkit Documentation 46
[url]Programming Guide :: CUDA Toolkit Documentation 7

Linux中进程的虚拟内存布局



cat /proc/$PID/maps

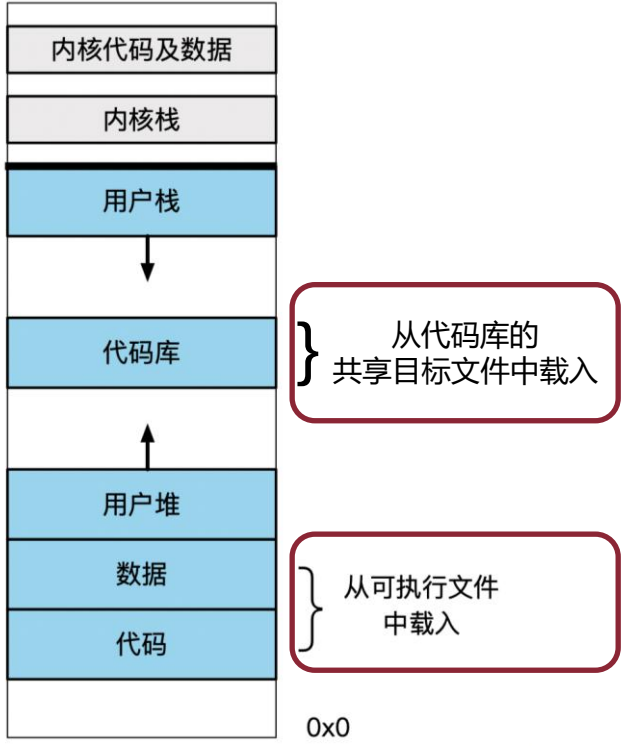
```
ffffc2122000-ffffc2143000   [stack]
ffffae6a8000-ffffae6ab000   [/lib/aarch64-linux-gnu/ld-2.24.so]
ffffae6a7000-ffffae6a8000   [vdso]
ffffae6a6000-ffffae6a7000   [vvar]
ffffae699000-ffffae69b000   (anonymous)
ffffae67d000-ffffae699000   [/lib/aarch64-linux-gnu/ld-2.24.so]
ffffae679000-ffffae67d000   (anonymous)
ffffae533000-ffffae679000   [/lib/aarch64-linux-gnu/libc-2.24.so]
aaaadb9ec000-aaaadba0d000   [heap]
aaaab2607000-aaaab2609000   [/home/xiaoming/hello] (data)
aaaab25f7000-aaaab25f8000   [/home/xiaoming/hello] (code)
```

应用程序存储格式：目标文件

- **ELF：目标文件的标准二进制格式**
  - 可执行目标文件
  - 可重定位目标文件（.o）
  - 共享目标文件（.so）
    - 支持动态链接：加载时或运行时

ELF格式：可执行目标文件

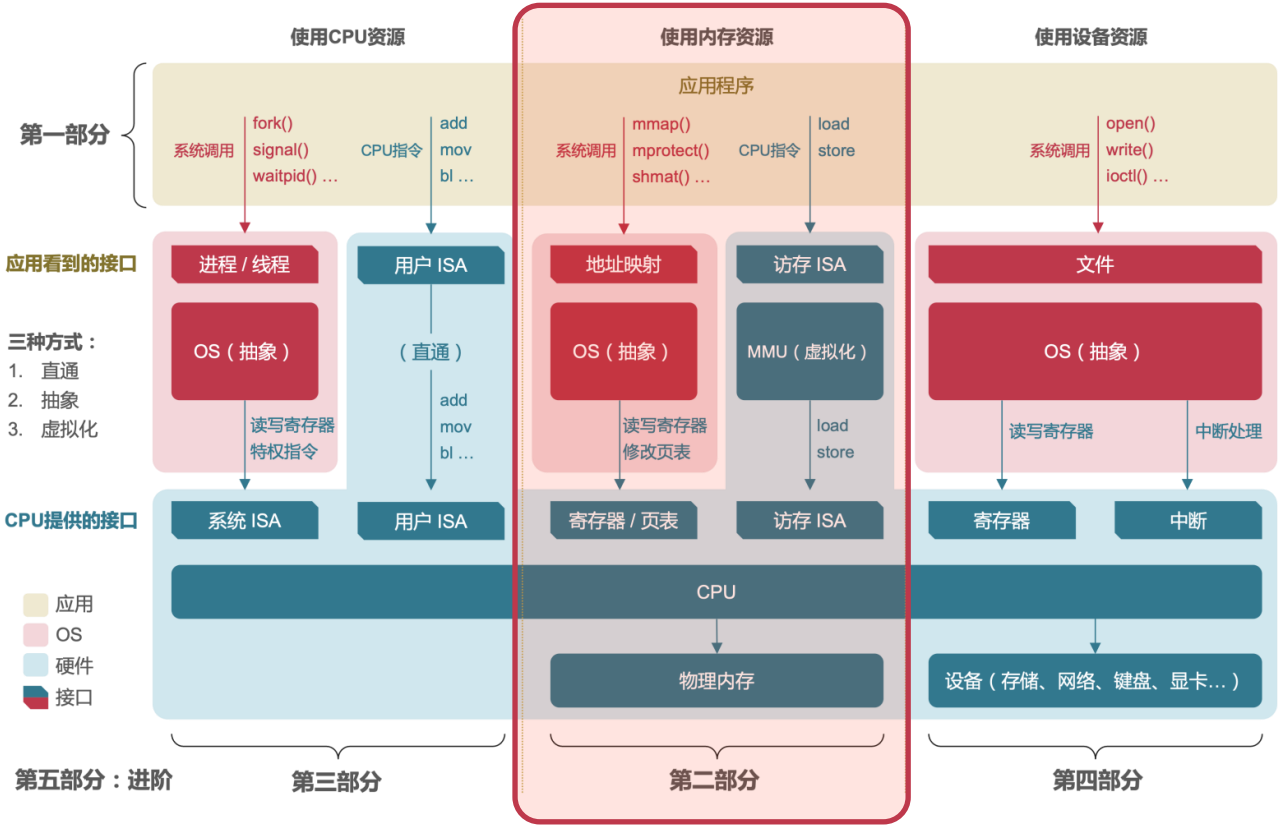
回顾：常见的进程虚拟内存布局



cat /proc/$PID/maps

```
ffffc2122000-ffffc2143000   [stack]
ffffae6a8000-ffffae6ab000   [/lib/aarch64-linux-gnu/ld-2.24.so]
ffffae6a7000-ffffae6a8000   [vdso]
ffffae6a6000-ffffae6a7000   [vvar]
ffffae699000-ffffae69b000   (anonymous)
ffffae67d000-ffffae699000   [/lib/aarch64-linux-gnu/ld-2.24.so]
ffffae679000-ffffae67d000   (anonymous)
ffffae533000-ffffae679000   [/lib/aarch64-linux-gnu/libc-2.24.so]
aaaadb9ec000-aaaadba0d000   [heap]
aaaab2607000-aaaab2609000   [/home/xiaoming/hello] (data)
aaaab25f7000-aaaab25f8000   [/home/xiaoming/hello] (code)
```

进程使用内存资源（虚拟化+抽象）



文件

Unix 文件

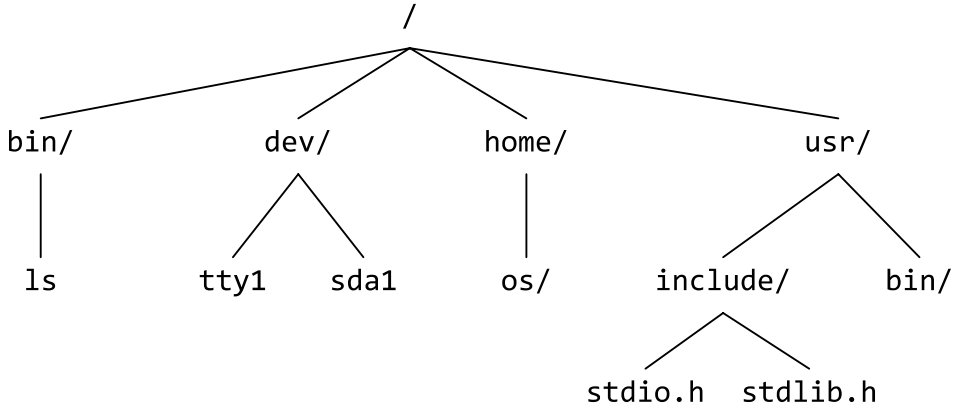
- **Unix文件是一串字节序列**
  - $B_0, B_1, \ldots, B_k, \ldots, B_m$
- **所有IO设备都被抽象成文件**
  - 如,网络设备、硬盘、终端等
  - Unix提供一个基于文件的底层应用接口，即 **Unix IO**
- **所有输入/输出都是通过读/写文件完成**
  - 让所有输入输出都有统一的表现方式

文件类型1

- **普通文件(regular file)：包含任意数据**
  - 从应用程序的角度来看有两种
    - 文本文件
    - 二进制文件
  - 从内核的角度来看，没有区别

文件类型2

- **目录(directory)也是一个文件**
  - 由一组“文件名到文件的映射”构成
- **Linux内核使用层次化目录来组织所有文件**
  - /：代表根目录



文件类型3

- **套接字(Socket)也是文件，常用于跨网络交互**
- **其他文件类型包括：**
  - 命名管道(named pipes)
  - 符号连接(symbolic links)
  - 字符/块设备(character/block devices)
  - ...

打开/关闭文件

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

int open(char *filename, int flags, mode_t mode);
```

返回值：成功(新文件标识符)，出错(-1)

```
#include <unistd.h>
int close(int fd) ;
```

返回值：成功(0);失败(-1)

文件标识符(file descriptor, fd)

读写文件

```
#include <unistd.h>

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

返回值：成功(读到字节数)，遇到EOF(0)，失败(-1)

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

返回值：成功(写入字节数)，失败(-1)

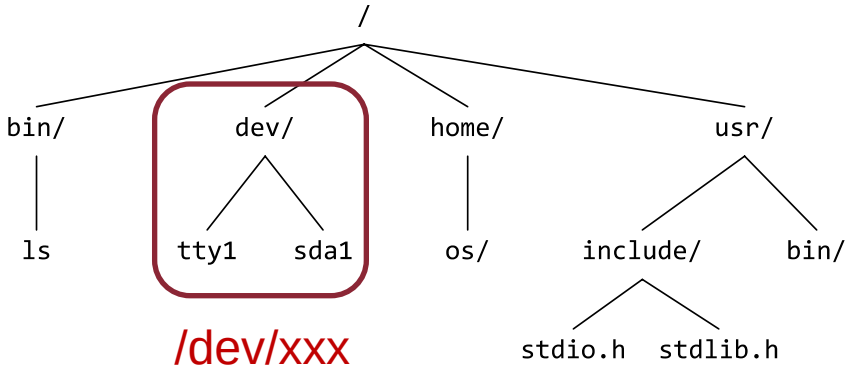
文件标识符(file descriptor, fd)

读写文件示例

```c
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    // 打开文件 hello.txt。O_RDONLY 表示以只读方式打开
    int fd = open("hello.txt", O_RDONLY);

    char result[14];
    // 从 fd 对应的文件中读取 13 个字节
    read(fd, result, 13);
    // 向屏幕输出读取到的字符串
    write(1, result, 13);
    close(fd);
}
```

文件：对所有设备的抽象

- **存储设备**
  - File
- **网络设备**
  - Socket
- **其他设备**
  - 同样是fd



操作设备的示例

```c
// file: hello-tty.c
#include <unistd.h>
#include <sys/fcntl.h>
int main()
{
  // 打开另一个终端的 TTY 设备文件，O_WRONLY 表示允许写入
  int fd = open("/dev/pts/18", O_WRONLY);
  // 向该文件进行输出
  write(fd, "Hello World!\n", 13);
  // 关闭文件
  close(fd);
}
```

Terminal 1

```
[user@osbook ~] $ tty
/dev/pts/18
[user@osbook ~] $ Hello World!
```

Terminal 2

```
[user@osbook ~] $ vim hello-tty.c
[user@osbook ~] $ gcc hello-tty.c -o hello-tty
[user@osbook ~] $ ./hello-tty
```

总结：OS对CPU、内存、设备的三大抽象