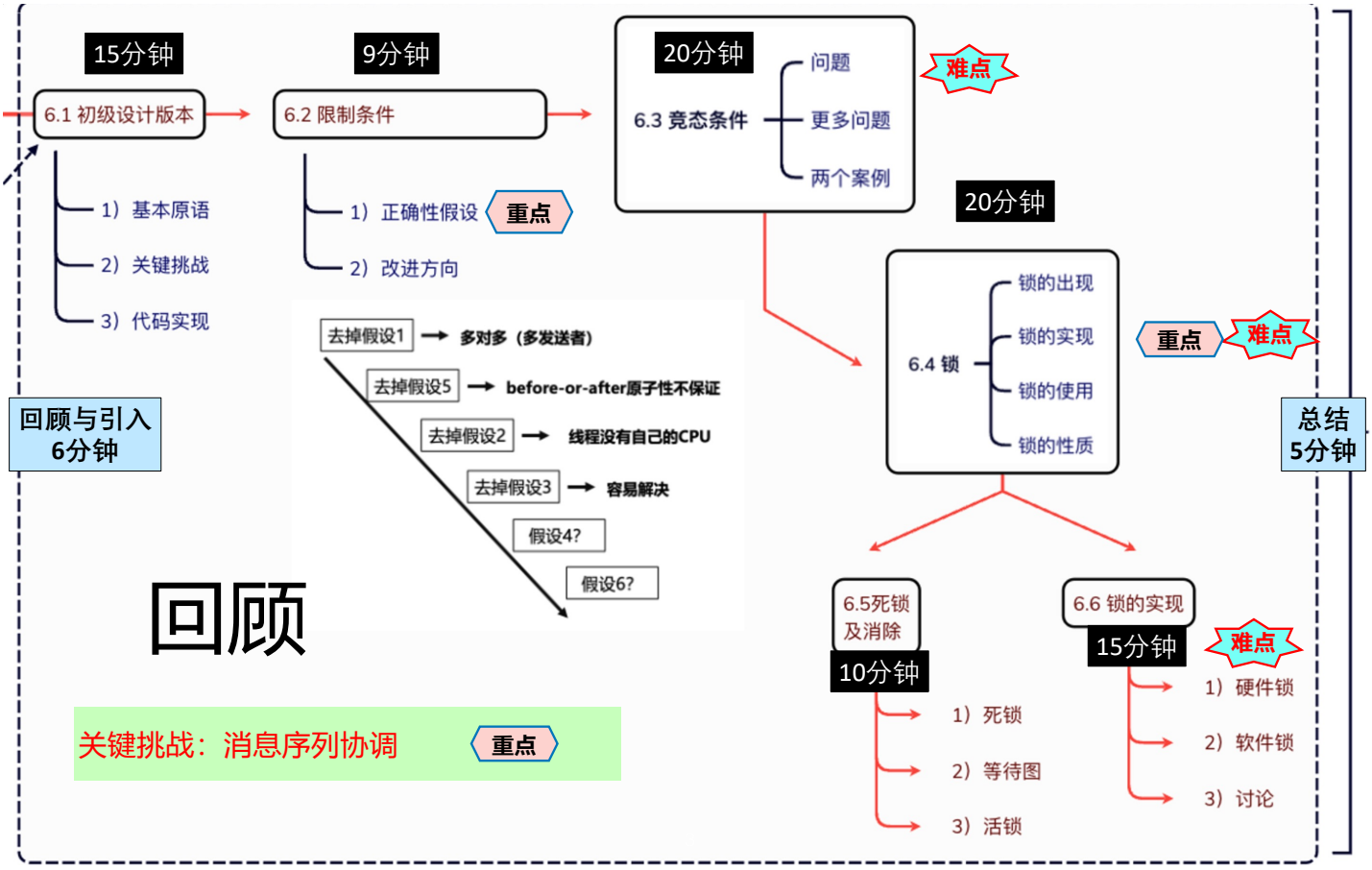
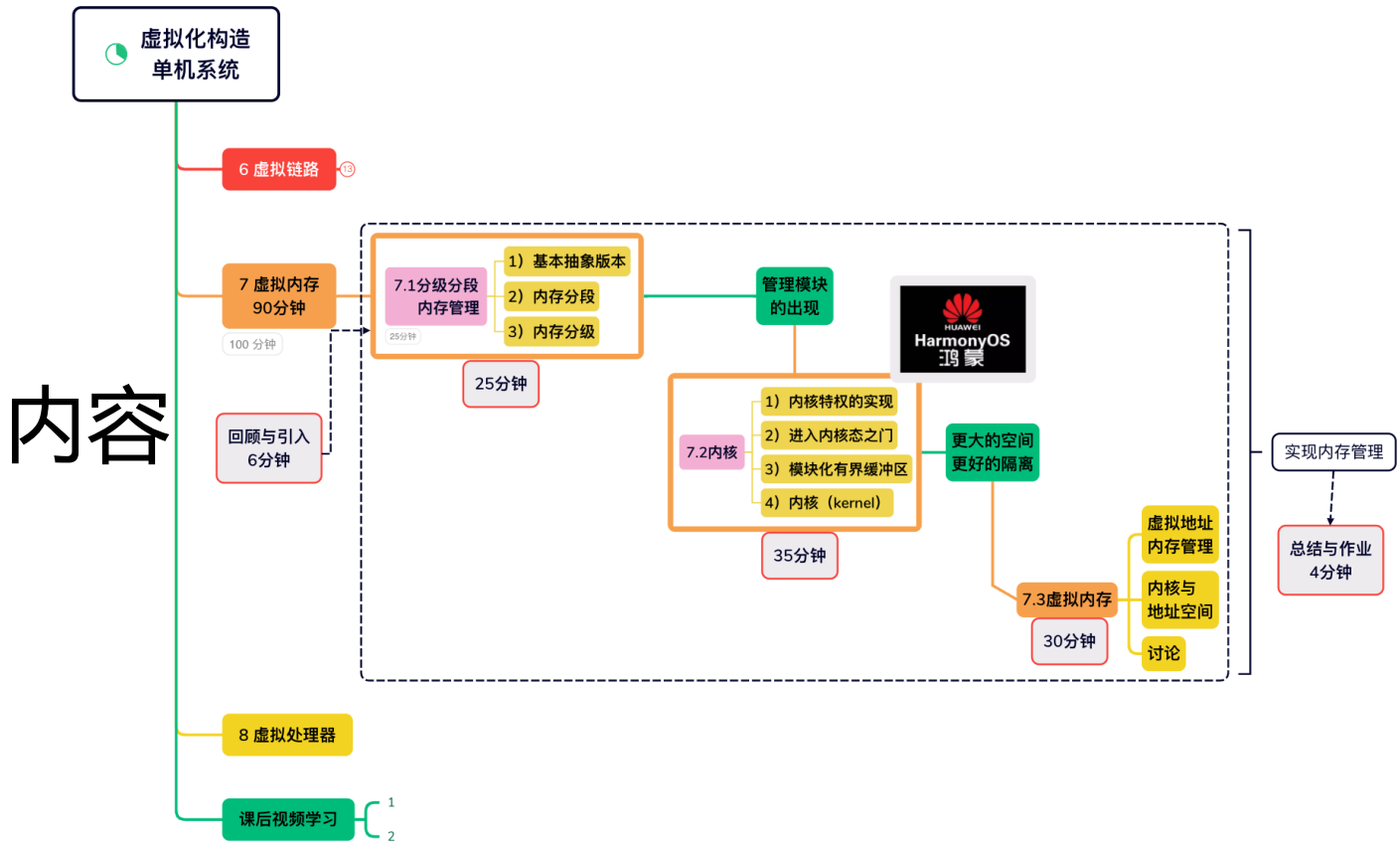
7. 内存模块化与虚拟内存

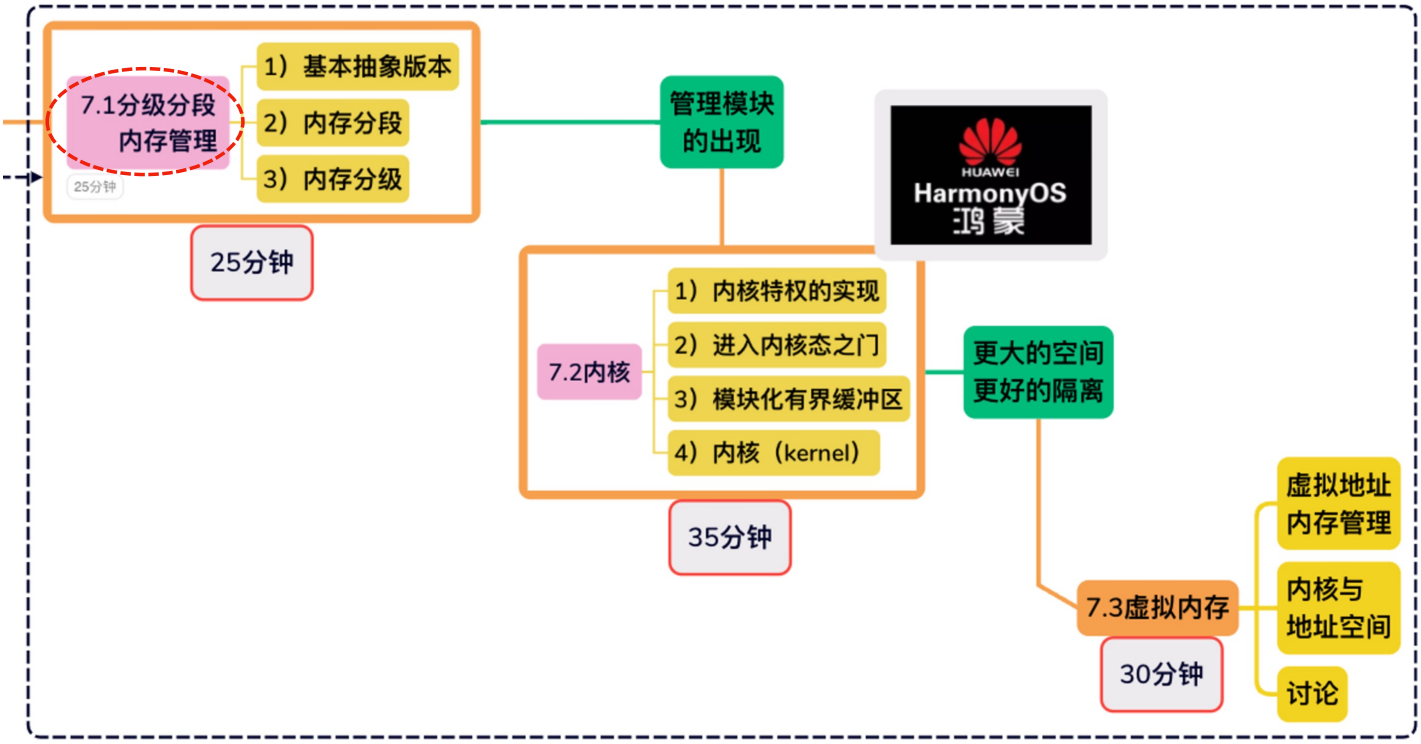
本章主要参考文献

- Andre Bensoussan, Charles T. Clingen, and Robert C. Daley. The Multics virtual memory: Concepts and design. Communications of the ACM 15, 5 (May 1972), 308–318.
- Liedtke, J. On Microkernel Construction. In Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP'15) , 237-250， 1995.
- Kolanski, R. A Logic for Virtual Memory. Electronic Notes in Theoretical Computer Science, 217, 61–77， 2008.


回顾与引入
6分钟
15分钟
6.1 初级设计版本
1）基本原语
2）关键挑战
3）代码实现
9分钟
6.2 限制条件
1）正确性假设
重点
2）改进方向
20分钟
6.3 竞态条件
问题
更多问题
两个案例
难点
去掉假设1
多对多（多发送者）
去掉假设5
before-or-after原子性不保证
去掉假设2
线程没有自己的CPU
去掉假设3
容易解决
假设4?
假设6?
回顾
关键挑战：消息序列协调
重点
20分钟
6.4 锁
锁的出现
锁的实现
锁的使用
锁的性质
重点
难点
6.5死锁及消除
10分钟
1）死锁
2）等待图
3）活锁
6.6 锁的实现
15分钟
难点
1）硬件锁
2）软件锁
3）讨论
总结
5分钟


内容
虚拟化构造
单机系统
6 虚拟链路
7 虚拟内存
90分钟
100 分钟
回顾与引入
6分钟
7.1分级分段
内存管理
25分钟
1）基本抽象版本
2）内存分段
3）内存分级
25分钟
管理模块
的出现
HarmonyOS
鸿蒙
7.2内核
1）内核特权的实现
2）进入内核态之门
3）模块化有界缓冲区
4）内核（kernel）
35分钟
更大的空间
更好的隔离
7.3虚拟内存
30分钟
虚拟地址
内存管理
内核与
地址空间
讨论
实现内存管理
总结与作业
4分钟
8 虚拟处理器
课后视频学习

7.1 分级分段内存管理


7.1分级分段
内存管理
25分钟
1）基本抽象版本
2）内存分段
3）内存分级
25分钟
管理模块
的出现
HUAWEI
HarmonyOS
鸿蒙
7.2内核
1）内核特权的实现
2）进入内核态之门
3）模块化有界缓冲区
4）内核（kernel）
35分钟
更大的空间
更好的隔离
7.3虚拟内存
30分钟
虚拟地址
内存管理
内核与
地址空间
讨论

7.1 分级分段内存管理

7

1）基本抽象版本

- 到现在为止，内存已经抽象到什么层面了？
  - （读、写）连续等长的单元**序列**
- 是否适用于多线程并发？
  - 可行性：以单元为单位划分
  - 潜在问题：
    1. **效率怎么样？**
    2. **不同线程的内存能否互相破坏？**
    3. **谁来监督或防止破坏？**
- 能否改进？

内存模块化

- 内存模块化
  - 划分为连续地址的段
  - 1个线程分配1段，互不重叠
  - 问题：
    - 如何防止线程访问其他线程内存？
      - 自律？

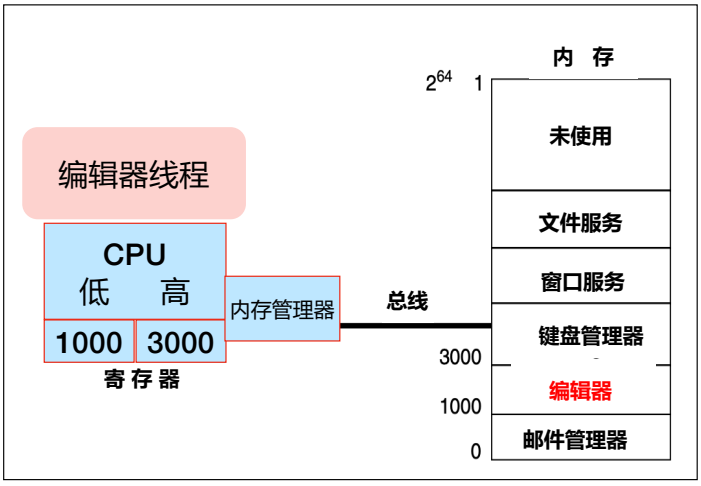
        ✗
      - 强制管理！

        ✓

模块化+抽象 才是我们的重要设计方法！

2）内存分段

- 如何强制实施？
  - 使用软件间接层，截获和判断？
    - 效率 ✗
  - 是否可以合并入原有的设计？
    1. CPU载入当前线程的分段
    2. CPU执行当前线程的指令
    3. CPU访问地址前进行判断
  - 如何实现？
    1. CPU增加分段寄存器
    2. CPU增加内存管理器
    3. CPU增加寻址前的判断



2）内存分段

- 目前已用以下硬件支持了内存分段
  - 内存分段寄存器
    - 记录分段的起始地址
  - 内存分段管理器
    - CPU访存时进行判断，越界抛出异常memory reference exception
- 问题

1. 为了共享和灵活性等需要，线程能否有多个段?

2. 能否为不同段设置不同的权限?

分段的扩展设计——更多的段

1. 更多的段

- 硬件扩展
  - 多个寄存器
    - 例，4个：data - 数据，stack - 栈，text - 代码，extra - 共享数据
  - GDT间接层
- 软件扩展
  - 分配内存：base_address ← allocate_domain (size)
  - 绑定线程：map_domain (base_address)
  - 维护内存空闲状态
    - 空闲内存表，已分配段的表（起始+大小： domain_table ）

分段的扩展设计——权限 

2. 设置内存段的权限

- 给段寄存器加权限位
  - 保护代码段不被修改 → No W = RX
  - 保护数据段不被执行 → No X = RW
  - 保护常量数据段不被改写 → No XW = R
  - 保护栈段不可执行（这种NX需求在未来出现）
- 收益：
  - 权限使程序更安全
  - 权限使共享更灵活
    - 例如：共享给别的线程仅R权限，防止写冲突

权限的实现

- 实现
  - 内存段绑定线程时，将权限存于CPU寄存器中
    - map_domain (base_address, permission)
  - 访问时，内存管理器对 <行为 , 权限> 进行合法性判定
    - 行为对应的指令
      - 读 = LOAD
      - 写 = STORE
      - 执行 = 指令地址装入PC（跳转、返回等）
- 拓展小问题：**CPU在任何时候都会检查权限吗？**（16章内容）

哪些权限组合被实际使用?

	无	W	X	WX
R	R 常量	RW 变量	RX 代码	RWX 特殊用途

讨论

内存管理器：硬件还是软件?

- 效率：硬件实现
  - 内存管理器与CPU集成在一起
- 灵活：硬件+软件实现
  - 软件部分可以拓展更多功能

内存管理的拓展应用

回顾： “第2章 I/O内存映射”

至此，设备权限也可以被内存管理器控制。

- 谁来管理、怎么管理呢?

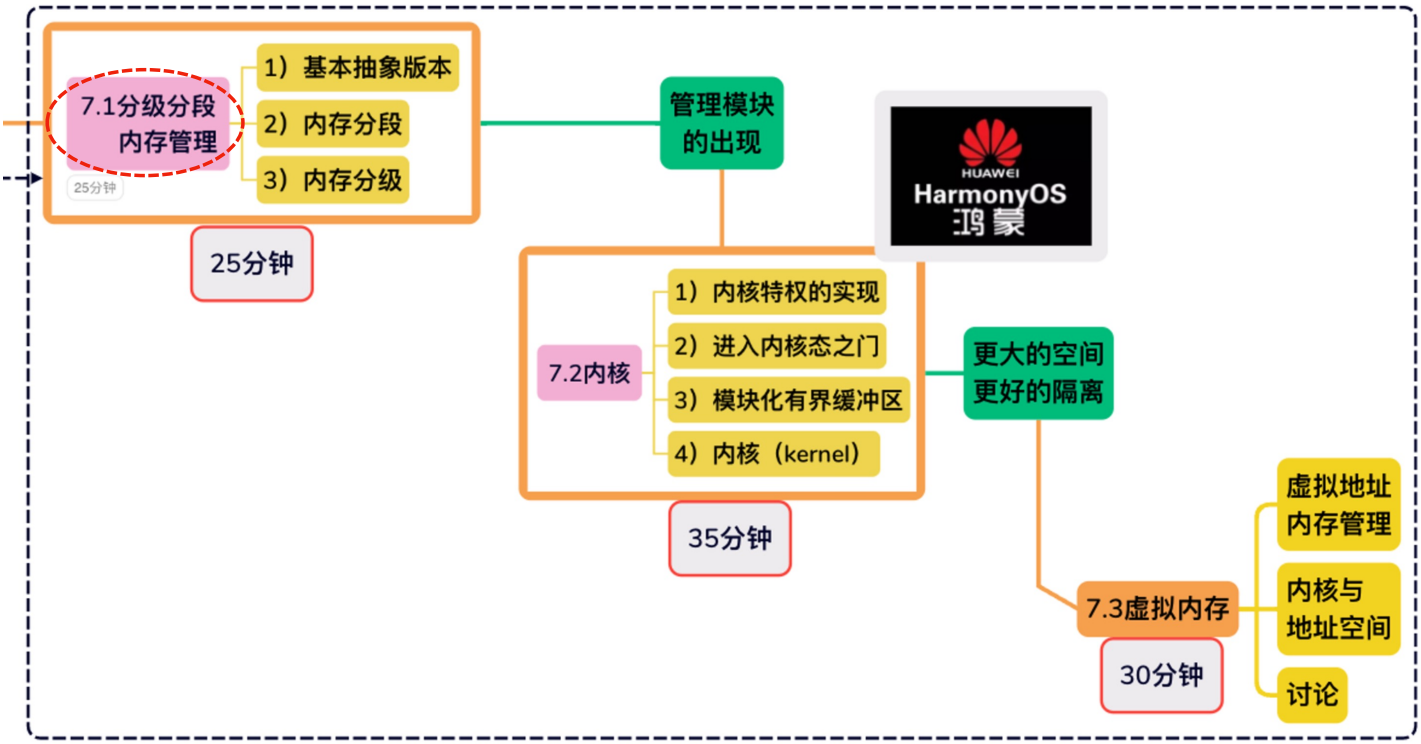
3）内存分级

- 问题：线程无权修改自己（以及他人）的内存段寄存器
  - 谁可以?
  - 权限分级：管理程序 > 应用程序(不能访问)
  - 实现

未来是否会有管理程序的管理程序？

    - 在CPU中表示当前进程的状态 → 状态位
      - 为CPU的两个状态起名：OS - 内核态，应用程序 - 用户态

- 新问题：内核态进程的内存需要额外保护
  - 内存分级：内核态内存的段寄存器有所标记，段寄存器增加1位→ kernel-only
    - 只有内核态可以访问内核空间


7.1分级分段
内存管理
1）基本抽象版本
2）内存分段
3）内存分级
25分钟
25分钟
管理模块
的出现
HUAWEI
HarmonyOS
鸿蒙
7.2内核
1）内核特权的实现
2）进入内核态之门
3）模块化有界缓冲区
4）内核（kernel）
35分钟
更大的空间
更好的隔离
7.3虚拟内存
30分钟
虚拟地址
内存管理
内核与
地址空间
讨论

7.2 内核（kernel）

管理模块的出现

问题

- 线程是否有权修改内存段寄存器?
  - 不可以!
- 谁可以修改内存段寄存器?
  - 专门的管理模块（内核，kernel）

管理模块的设计思路

- 将CPU的权限（称为特权级）分2级，赋予管理模块**高级权限**:
  - 高级：能访问内存段寄存器
  - 低级：只能访问其他寄存器

1）内核特权实现

保护段寄存器

- CPU增加1位专门的状态位寄存器，表示当前的**特权级**
  - 状态位设为**高级**时：称CPU处于**内核态**，可以修改内存段寄存器
  - 状态位设为**低级**时：称CPU处于**用户态**，访问寄存器会触发**异常**
    - illegal instruction exception

内核的程序空间也需要保护

保护内核

- CPU**段寄存器**增加KERNEL-ONLY位，含义为该段是否**仅允许内核访问**
- 内核程序的段寄存器设为 KERNEL-ONLY
  - CPU处于内核态时可访问，否则触发**异常**
    - permission error exception

特权级切换

内核态的初始化

- CPU加电后处于内核态

内核态 → 用户态

- 内核启动后，将自己的内存段设为KERNEL-ONLY
- 内核加载应用程序进入用户态

用户态 → 内核态

- 应用程序通过系统调用陷入内核，进入内核态

2）进入内核态之门

问题：

– **线程进行内存管理时，需调用内核态线程功能。如何进入内核态?**

设计：

- 设计目标：进入内核态，同时说明进入原因（管理内存的请求）
- 候选方案：PC寄存器指向内核态代码
  - 难以实现，且不安全
- 采用方案：“门”（gate）方案

① 门——进入

方案

1. 用户程序
   - 将gate值放入某寄存器
   - 执行管理程序调入指令（SVC，supervisor call instruction）
2. CPU
   - 设置状态为内核态
   - 设置PC为预定义的 gate管理程序的入口地址
3. gate管理程序
   - 读gate值并查表，跳转到对应处理代码

arm: svc
x86: sysenter
syscall
linux+x86:
int 80
windows+x86:
sysenter

原子性保证

进入内核态应为before-or-after原子操作，不能被中断

为什么?

问题:

– 如何确保原子性?

简单方案

- SVC时禁止所有中断

② 门——返回

方案

- CPU设置为用户态
- 从内存（栈）中取出PC值放入PC寄存器

无需考虑befor-or-after原子性

- 即使被中断，也不会发生越权操作问题

门——用于中断和异常

中断和异常都可以用门来解决

- 都涉及 用户态 - 内核态 切换问题

课后习题

3）模块化有界缓冲区

回顾：有界共享缓冲区可用于通信

- 问题：
  - **内存已分段分级，但有界共享缓冲区可以任意写，允许错误传播**

- 方案：模块化隔离
  - 缓冲区置于内核空间（KERNEL-ONLY）
  - SEND和RECEIVE纳入系统调用（supervisor call）
  - 使用方式：陷入内核态，执行缓冲区数据复制

3）模块化有界缓冲区

- 方案开销
  - 性能损失，多次拷贝

- 方案优化
  - 合并调用
  - 小消息寄存器传递
  - 数据结构优化
  - 等等……（扩展阅读中的研究文献）

4）内核（kernel）

关于操作系统内核的常用知识

- 内核（kernel） = the kernel program
  - 运行于内核态的模块集合称为内核程序，简称内核
  - 当我们在讨论操作系统的设计原理时，我们在讨论内核
  - 当我们在讨论操作系统的使用功能时，我们在讨论外壳（shell）
    - 或shell之下的工具、命令，或shell之上的图形界面，或C语言库
- 内核的加载与运行
- 内核的重要作用：**可信中介——**唯一可执行特权指令的程序

两种内核架构

- 宏内核
  - 设计时将设备管理、文件管理等功能均纳入内核
- 微内核
  - 最小抽象集合
    - 尽量少的功能封装在内核中，尽量多的模块暴露在内核外
    - 内核外的模块互相通信用IPC，与内核通信用SVC
  - IPC是最大的开销
  - 简单是最大的收益
  - （文献Hansen）

微内核？宏内核！

- 如何进行选择？
  1. 可靠性与容错
     - 隔离关键组件为容错带来什么收益？**
  2. 调试与实现
     - 哪个调试效率效率高？**
     - 对关键系统组件进行隔离，意义有多大？
     - 服务之间通信多，哪个实现更简单？
  3. 性能
     - IPC对性能影响有多大？
  4. 兼容
     - 如何考虑历史传承问题？

商业操作系统几乎不使用纯微内核
UNIX、GNU/Linux
Windows、MacOS
Android

历史的财富与历史的包袱……
这是否是国产操作系统的机遇？

Harmony OS（华为鸿蒙）

操作系统，是计算机系统的灵魂，涉及国家安全，但在全球信息技术领域，

问题，也可 国产操作系统可以借此

基于微内核 系统不同，具备真正的

这是我国计算机系统迈向自主可控的重要一步。

缺芯少魂，一个都不能少！

工业界为什么未采用微内核?

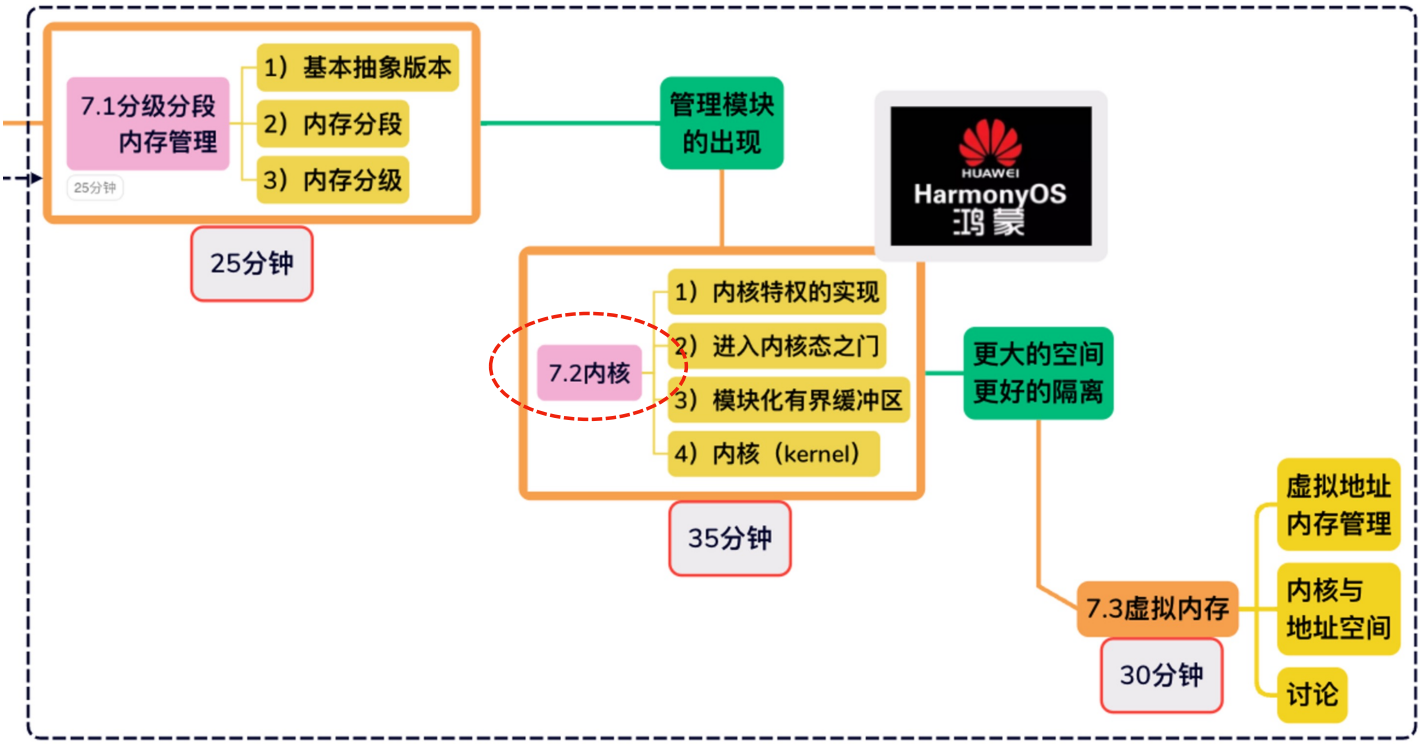
- 技术决策的思考过程
  1. 代价是确定的：改变的代价，未知的风险
  2. 收益是不确定的：尚无足够实证
  - 机会成本也是成本：决策的机会成本 = 舍弃的最大价值

- **另外的知识：
  - 微内核相比宏内核更适合于形式化安全证明
  - 如：seL4（文献Klein, G.）

The Tanenbaum-Torvalds Debate

- LINUX is obsolete?
  - Andrew Tanenbaum vs. Linus Benedict Torvalds
  - 1992
  - https://www.oreilly.com/openbook/opensources/book/appa.html

- 有兴趣的同学课后阅读，这是计算机系统的知名公案


7.1分级分段
内存管理
1）基本抽象版本
2）内存分段
3）内存分级
25分钟
25分钟
管理模块
的出现
HUAWEI
HarmonyOS
鸿蒙
7.2内核
1）内核特权的实现
2）进入内核态之门
3）模块化有界缓冲区
4）内核（kernel）
35分钟
更大的空间
更好的隔离
7.3虚拟内存
30分钟
虚拟地址
内存管理
内核与
地址空间
讨论

7.3 虚拟内存

还未解决的问题

- 问题
  1. **线程的进入、退出，使系统内存逐步变得碎片化！**
  2. **当线程增加时，每个线程的分到的内存变得很小！**
  - **如何解决这两个问题？**
- 间接层解决一切系统设计问题！
  - 用虚拟地址取代物理地址
    - 虚拟的设计目标：**连续、完备**的内存地址空间
    1. **连续**：内存地址经过了转换，解决问题1
    2. **完备**：空间大小经过了转换，解决问题2

虚拟内存（virtual memory）的出现

"虚拟内存管理器" 取代 "内存分段管理器"，增加2项特性：

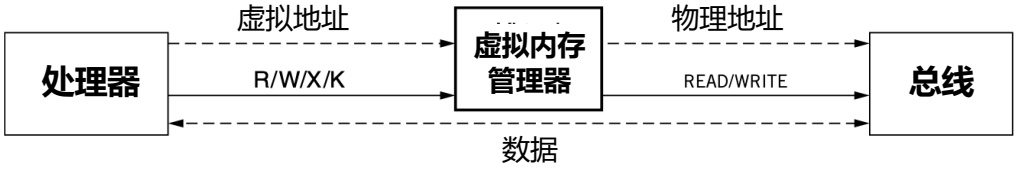
1. 虚拟地址 （virtual addresses）
   - 收益：存在间接转换，管理器可整理内存段

2. 虚拟地址空间 （virtual address spaces）
   - 收益：给每个程序全部的内存地址空间

1）虚拟地址

线程使用虚拟地址，访问时通过虚拟内存管理器转换

- 命名式间接：增加一层命名，名称解析即间接层
- （回顾第3章“命名”）



收益 （☆课下思考）：复用、增大、透明?

开销

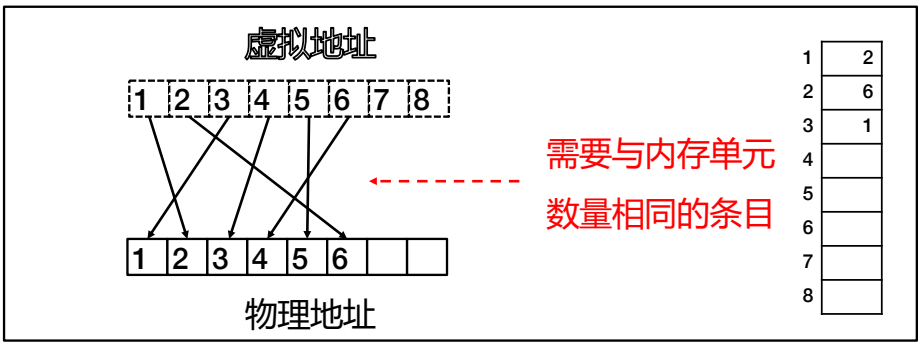
1.复杂性

2.性能

权衡：地址翻译粒度

方案1：以内存单元为单位进行地址翻译

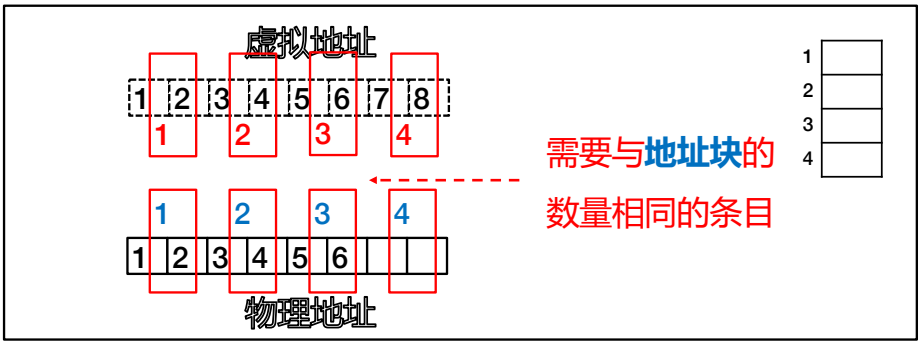
- 映射条目与内存单元的数量相同
- 映射表大小为内存的若干倍——翻译表太大！ ✗



权衡：地址翻译粒度

方案2：以较大的连续地址块为单位进行地址翻译

- 将物理内存组织成连续的**块**，虚拟内存组成成连续的**页。**将块与页进行对应，块和页内的顺序在映射时不变
- 映射条目与块或页的数量相同——可以接受！ ✓

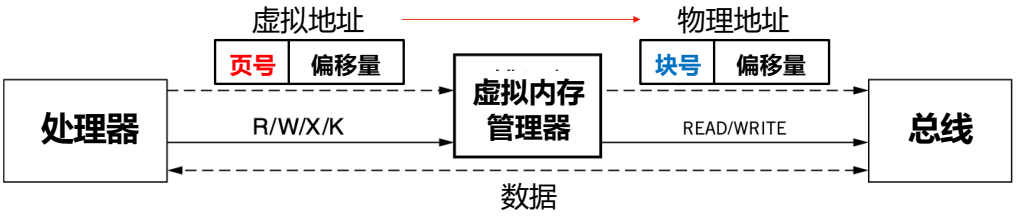


2）虚拟地址映射

- 映射方式

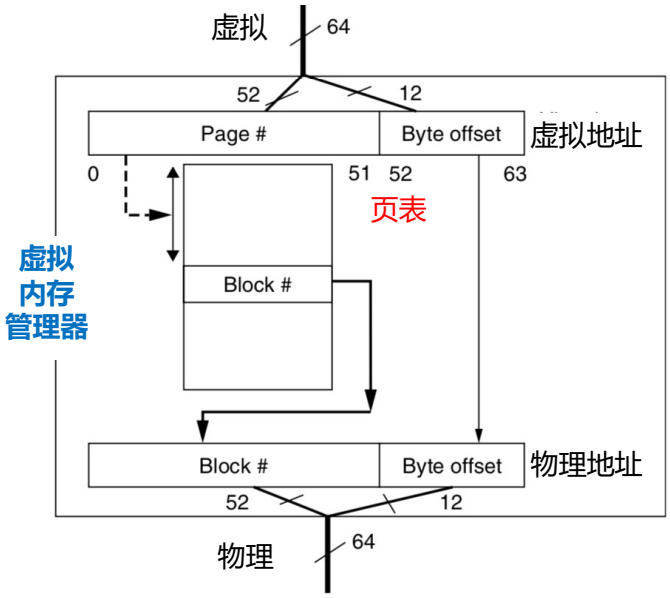
  - 虚拟内存页 页号 **映射为** 物理内存块 块号

  - 虚拟地址 = 页号 + **偏移** **映射为** 物理内存 = 块号 + **偏移**



虚拟内存管理器

- 以64位地址、4k页为例
  - 虚拟内存地址
    1. 前52位为页号
    2. 通过页表转换为块号
    3. 后12位为页内偏移，不变
    4. 块号+偏移=物理内存地址

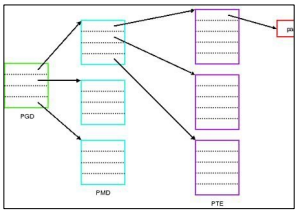


页表的实现

- 线性页表
  - 随机读写快
  - 大小不现实

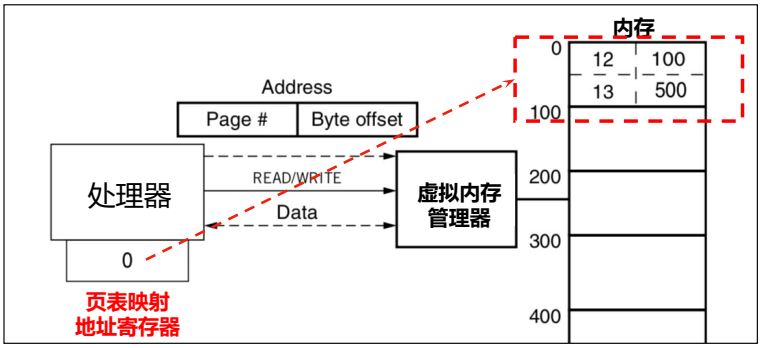
虚拟地址	物理地址
1	3456
2	999
3	18
4	无

- 多级页表（实际使用）
  - 两级结构
  - 反向结构



页表的实现

- 页表存在哪里?
  - 指定的内存中
    - 寄存器“**page-map address register（PMAR）**”指向该内存
    - 内存权限：kernel mode 可写
- 性能
  - 虚拟内存访问
    - 读×2
    - 查表×1



3）虚拟地址空间

到目前为止，**地址空间是全局的、共用的**

问题2：

– **当线程增加时，每个线程的分到的内存变得很小！**

地址不够用，能否虚拟化创造 "空间"?

方案：

- 借助间接层，虚拟出完备的空间（virtual address space）
- 从此“地址空间”成为**可数**名词

① 虚拟地址空间管理

- 管理操作

1. 创建1个虚拟空间

```
id ← create_address_space ()
```

2. 申请一些物理内存，映射到一部分虚拟地址

```
block ← allocate_block ()
map (id, block, page_number, permission)
```

3. 动态增加(申请+映射) & 动态删除(解除映射+释放)

```
unmap (id, page_number)
free_block (block)
```

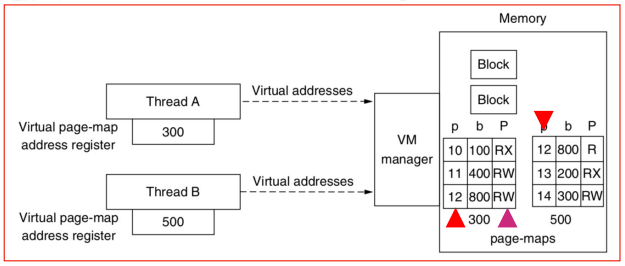
4. 删除这个虚拟空间

```
delete_address_space (id)
```

① 虚拟地址空间管理

用页映射来管理物理空间

- 访问规则：
  - 页表中存在的可访问
  - 页表中不存在的不可访问



- 设计机遇：1个物理内存可以映射在多个页表中——共享

问题：段表+段寄存器是否还有必要存在?

观察：页表中可以放置权限：R W X

结论：**可以去掉段寄存器和段表**

课后思考

思考：虚拟内存的翻译

1. **思考：过程是在什么时机由谁调用的?**
   - 虚拟内存翻译应处于位置（ ① ），用（ ② ）件实现
2. **思考：如果找不到这个条目怎么办（图中没有考虑）？**
   - 虚拟内存还需要（ ③ ）机制支持。

```
1  procedure TRANSLATE (integer virtual, perm_needed) returns physical_address
2      page ← virtual[0:41]                          // Extract page number
3      offset ← virtual[42:63]                       // Extract offset
4      page_table ← PMAR                             // Use the current page table
5      perm_page ← page_table[page].permissions      // Lookup permissions for page
6      if PERMITTED (perm_needed, perm_page) then
7          block ← page_table[page].address          // Index into page map
8          physical ← block + offset                 // Concatenate block and offset
9          return physical                           // Return physical address
10     else return error
```

线程+虚拟空间=process（进程）？

一般用其表达操作系统运行程序时进行资源分配的基本单位

具体内涵因上下文而异，例如UNIX：

- 早期版本：一个线程+私有空间
- 后期版本：一组线程+私有空间
- 后期允许共享：一组线程+私有空间+共享空间

本课程使用**"线程"、"虚拟地址空间"、及二者组合**，无歧义

② 内核与地址空间

权衡：内核放在哪里？

1. 把内核映射到每个应用程序空间
   - 收益：
     1. 内核-应用切换，无需修改页映射地址寄存器
     2. 内核可方便访问用户空间
   - 缺点：减少了用户空间大小
2. 给内核与应用无关的地址空间
   - 实现：
     - 修改SVC指令，切换页映射地址寄存器
     - 所有应用的页表都放入内核，内核通过修改映射来访问用户空间
   - 缺点：无法直接寻址应用空间

实际是怎么做的呢？
为什么？（查一查）

③ 讨论（线上学习）

虚拟内存空间让不同上下文的空间都是完备的，有多大意义?

- 有作用，但不是必须的
  - 为什么？因为PIP（位置无关程序）技术的存在
  - PIP提供了更多的设计机会（什么？）
- 会带来共享的不方便和不灵活
  - 不方便：无法设立别名
  - 不灵活：粒度必须大于块的大小
  - 学者提出解决方法，但业界很少解决
    - （课后拓展阅读文献）

**PIP：Position-Independent program

位置无关程序

- 可加载到任意地址空间
- 编译过程只使用相对地址，不使用绝对地址
- 已成为安全通用技术，支持ASLR

（线上学习）

③ 讨论（线上学习）

硬件与软件如何分工?

- 按前述讨论，虚拟内存由软件+硬件实现。
  - 如何分工？权衡：性能 – 灵活性
  - 查表必须有硬件实现!
    - 为什么?

③ 讨论（线上学习）

如何改进页表访问的性能?

- 改进1：高速缓存（cache）——TLB快表
  - 依据：局部性原理（locality of reference）
  - 开销：增加复杂度，页表切换要附带cache切换或失效
- 改进2：提高cache检索速度
  - 相联存储器（associatIve momory）代替索引存储器

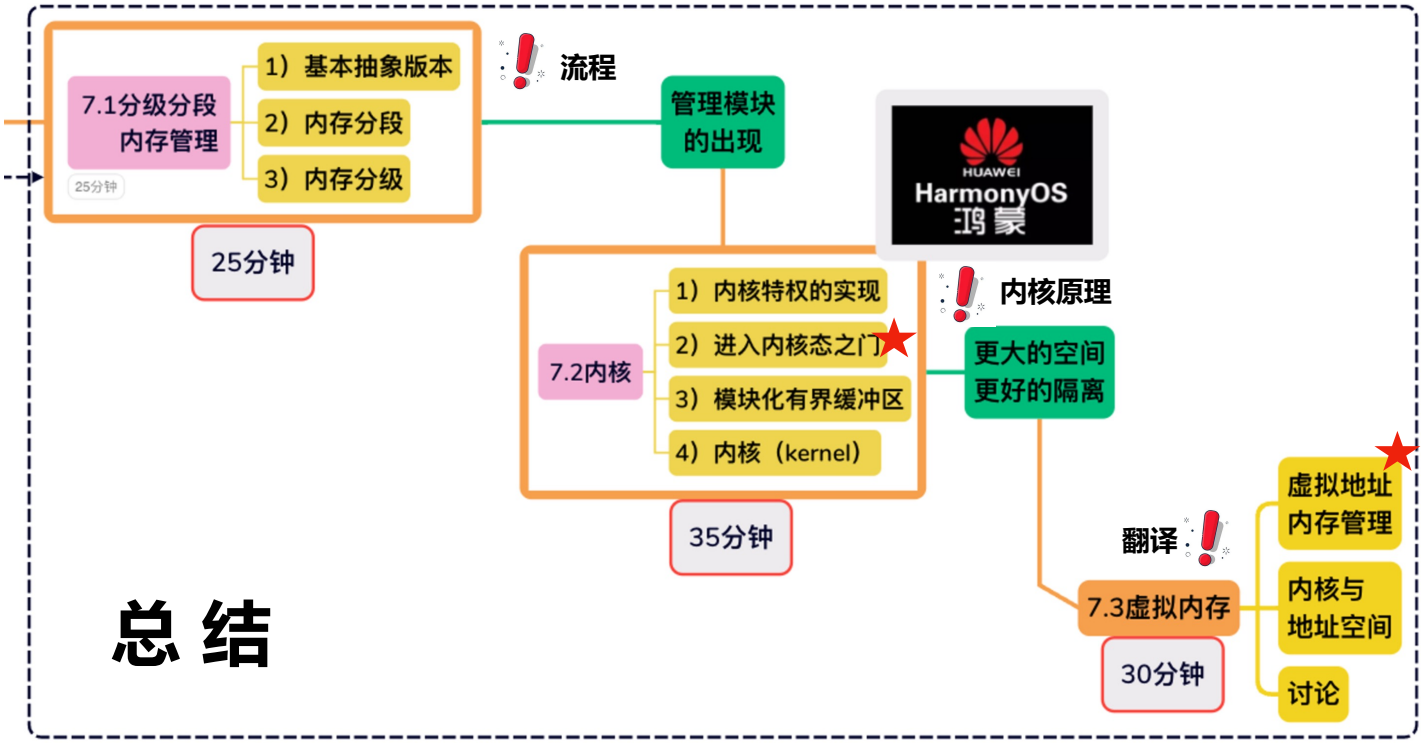
③ 讨论（线上学习）

页表格式由谁规定?

- 硬件规定
  - 例：前述MMU中 12+52格式（页映射实现）
- 软件规定（多数RISC）
  - 只有TLB是硬件实现，TLB不命中时，软件将页装入TLB
  - 软件内存管理器的高灵活性，页表可根据页多少使用不同数据结构
    - 页很少→链表/树；页太多→反向存

页表由谁维护?

- 软件维护，收益：
  - 可防止物理内存碎片
  - 可扩展物理内存大小



重要术语中英文对照

特权级：privilege level

内核态：kernel mode

用户态：user mode

内核：kernel

页表：page table

虚拟地址：virtual address

快表：TLB

内存管理单元：MMU

访问局部性原理：locality of reference

习题

小明为一台计算机开发了一个操作系统。该操作系统拥有一个内核，提供虚拟地址空间、线程和向控制台输出的功能。

每个应用程序都有自己的用户级地址空间，并使用一个线程。内核程序在内核地址空间运行，但没有自己的线程。

该计算机有一个处理器、一个内存、一个定时器芯片、一个控制台设备和连接这些设备的总线。

习题

处理器有一个用户模式位，并且设计为多个寄存器集，这意味着程序计数器（PC）、堆栈指针（SP）和页映射地址寄存器（PMAR）有两种设置。一种用于用户空间（用户模式位设置为ON）：upc、usp 和 upmar。另一种用于内核空间（用户模式位设置为OFF）：kpc、ksp 和 kpmar。只有内核模式下的程序才被允许存储到 upmar、kpc、ksp 和 kpmar 中——在用户模式下存储值是非法指令。

当发生以下三种事件之一时，处理器从用户模式切换到内核模式：应用程序发出非法指令、应用程序发出超级调用指令（使用 svc 指令）或处理器在用户模式下接收到中断。

习题

处理器通过关闭用户模式位来从用户模式切换到内核模式。当这种情况发生时，处理器继续运行，但使用当前 kpc、ksp 和 kpmar 的值。用户程序计数器、堆栈指针和页映射地址值分别保留在 upc、usp 和 upmar 中。

要从内核空间返回到用户空间，内核程序执行 RTI 指令，该指令将用户模式位设置为ON，处理器此时使用 upc、usp 和 upmar，kpc、ksp 和 kpmar 值保持不变，直到下一次 svc。

除了这些寄存器外，处理器还有四个通用寄存器：ur0、url、kr0 和 kr1。ur0 和 ur1 对在用户模式下可被程序访问和使用。kr0 和 kr1 对在内核模式下可被程序访问和使用。

习题

- 小明运行了两个用户应用程序。每个应用程序执行以下程序集：

```
integer t initially l          // initial value for shared variable t
procedure MAIN ()
    do forever
        t ← t + t
            print(t)
            YIELD()
procedure YIELD
    svc 0
```

在输出终端上打印 t 的值。输出控制台是一个仅输出的设备，不产生中断。

- 内核在每个程序自己的用户级地址空间中运行每个程序。每个用户地址空间有一个线程（有自己的栈），由内核管理：

```
integer currentthread          // index for the current user thread
structure thread [2]        // Storage place for thread state when not running
    integer sp                    // user stack pointer
    integer pc                  // user program counter
    integer pmar                  // user page-map address register
    integer r0                    // user register 0
    integer r1                    // user register 1
procedure doyield ()
    thread [currentthread].sp ← usp              // save registers
    thread [currentthread].pc ← upc
    thread [currentthread].pmar ← upmar
    thread [currentthread].r0 ← ur0
    thread [currentthread].r1 ← ur1
    currentthread ← (currentthread + 1) modulo 2     // select new thread
    usp ← thread [currentthread].sp                  // restore registers
    upc ← thread [currentthread].pc
    upmar ← thread [currentthread].pmar
    ur0 ← thread [currentthread].r0
    ur1 ← thread [currentthread].r1
```

- 为了简单起见，这个非抢占式线程管理器专门为本的内核上运行的两个用户线程量身定制。系统首先执行内核程序。以下是它的代码：

```
procedure kernel ()
    create _thread (main)            // Set up two threads
    create _thread (main)
    usp ← thread [1].sp              // initialize user registers for thread 1
    upc ← thread [1].pc
    upmar ← thread [1].pmar
    ur0 ← thread [1].r0
    ur1 ← thread [1].r1
    do forever
        RIT                          // Run a user thread until it issues an SVC
        n ← ???                      // 见问题1
        if n = 0 then doyield()
```

习题

由于内核通过RTI指令将控制权交给用户，当用户执行SVC指令时，处理器会在RTI指令之后的那条指令处继续在内核中执行。

小明的操作系统为每个用户程序设置了三个页映射，每个用户程序一个，内核程序一个。小明仔细设置了页映射，使得这三个地址空间不共享任何物理内存。

问题1 描述supervisor如何获取 n 的值，该值是调用程序所发出 svc 的标识符。

问题2 如何可以切换当前地址空间?

A. 通过内核写 kpmar 寄存器。

B. 通过内核写 upmar 寄存器。

C. 通过处理器改变用户模式位。

D. 通过应用程序写 kpmar 或 upmar 寄存器。

E. 通过 doyield 保存和恢复 upmar。

本章主要参考文献

- Andre Bensoussan, Charles T. Clingen, and Robert C. Daley. The Multics virtual memory: Concepts and design. Communications of the ACM 15, 5 (May 1972), 308–318.
- Liedtke, J. On Microkernel Construction. In Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP'15) , 237-250， 1995.
- Kolanski, R. A Logic for Virtual Memory. Electronic Notes in Theoretical Computer Science, 217, 61–77， 2008.