OS Note

AFool

2019

目录

第一章　操作系统概述

1.1　什么是操作系统

1.1.1　操作系统的定义

1. 操作系统是一个控制程序

   - 一个系统软件
   - 控制程序执行过程，防止错误和计算机的不当使用
   - 执行用户程序，给用户程序提供各种服务
   - 方便用户使用计算机系统

2. 操作系统是一个资源管理器

   - 应用程序与硬件之间的中间层
   - 管理各种计算机软硬件资源
   - 提供访问计算机软硬件资源的高效手段
   - 解决资源访问冲突，确保资源公平使用

1.1.2　操作系统软件的组成

1. Shell：命令行接口

2. GUI：图形用户接口

   WIMP：视窗 Window、图标 Icon、选单 Menu、指标 Pointer

3. Kernel：操作系统的内部

   执行各种资源管理等功能

1.1.3 操作系统内核特征

- 并发

  计算机系统中同时存在多个运行的程序，需要 OS 管理和调度；

- 共享

  “同时”访问

  互斥共享

- 虚拟

  利用多道程序设计技术，让每个用户都觉得有一个计算机专门为 TA 服务

- 异步

  程序的执行不是一贯到底，而是走走停停，向前推进的速度不可预知

  只要运行环境相同，OS 需要保证程序运行的结果也要相同

1.2 为什么学习操作系统

操作系统：计算机科学研究的基石之一。
会议：

- ACM 操作系统原理研讨会 SOSP
- USENIX 操作系统设计和实现研讨会 OSD

1.3 操作系统实例

UNIX BSD
Linux
Windows

1.4 操作系统的演变

主要功能：硬件抽象和协调管理
原则：设计随着各种相关技术的改变而做出一定的改变

- 单用户系统

- 批处理系统
- 多道程序系统
- 分时系统
- 个人计算机
- 分布式计算

1.4.1 单用户系统 (45-55)

操作系统 = 装载器 + 通用子程序库

问题：昂贵组件的低利用率

$$\text{利用率} = \frac{\text{执行时间}}{\text{执行时间} + \text{读卡时间}}$$

1.4.2 批处理系统 (55-65)

顺序执行与批处理

1.4.3 多道程序 (65-80)

保持多个工作在内存中并且在各工作间复用 CPU

1.4.4 分时系统 (70-)

定时中断用于工作对 CPU 的复用

1.4.5 个人计算机

- 单用户
- 利用率已不再是关注点
- 重点是用户界面和多媒体
- 很老的服务和功能不存在

从开始的：OS 为一个简单的服务提供者，到现在：支持协调和沟通的多应用系统。

1.4.6 分布式计算

- 网络支持成为一个重要的功能
- 通常支持分布式服务：跨多系统的数据共享和协调
- 可能使用多个处理器：松、紧耦合系统
- 高可用性与可靠性的要求

演变过程：
主机型计算 (Mainframe computing) → 个人机计算 (Personal computing)→ 网络计算 (Internet computing)→ 普适计算 (Pervasive computing)，移动计算，云计算，大数据处理

1.5 操作系统的结构

1.5.1 简单结构

MS-DOS：在最小的空间，设计用于提供大部分功能

- 没有拆分为模块
- 虽然 MS-DOS 在接口和功能水平没有很好地分离，主要使用汇编编写

1.5.2 分层结构

- 将 OS 分为多层
  - 每层建立在底层之上
  - 最底层 layer 0 是硬件
  - 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

主要贡献：可移植性
UNIX 在 1972 设计，高级系统编程语言 C 语言，创建了可移植操作系统的概念。

1.5.3 微内核

分层带来效率低的问题，因此提出**微内核** (Microkernel)：尽可能多的功能放到用户态，内核只保留进程间通讯和底层硬件支持

- 尽可能把内核功能移到用户空间
- 用户模块之间的通信使用消息传递
- 优点：灵活，安全
- 缺点：性能

1.5.4 外核结构

在内核中方更少的东西，起到资源的隔离，资源管理交给应用态的代码去完成。

- 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

1.5.5 VMM 虚拟机管理器

VMM 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类指令。

操作系统和 VMM 打交道，VMM 实现硬件交互，VMM 资源的隔离，操作系统负责资源的管理。

1.6 问题

1. 程序正在试图读取某个磁盘的第 100 个逻辑块，使用操作系统提供的（A ）接口

   A 系统调用

   B 图形用户

   C 原语

   D 键盘命令

   操作系统作为用户和计算机硬件系统之间的接口，用户可以通过 3 种方式使用计算机，命令方式、系统调用方式、图形方式。系统调用按照功能分为进程管理、文件操作、设备管理等，本题描述的是文件操作系统调用相关的执行。

2. 单处理器系统中，可并行执行或工作的对象是（D ）

1) 进程与进程
2) 处理器与设备
3) 处理器与通道
4) 设备与设备

A 1 2 3

B 1 2 4

C 1 3 4

D 2 3 4

并行指同一时刻同时发生，同一时刻单个处理器只能运行一个进程。

3. 下列选项中，操作系统提供给应用程序的接口是（A ）

A 系统调用

B 中断

C 库函数

D 原语

4. 下列选项中，在用户态执行的是（A ）

A 命令解释程序

B 缺页处理程序

C 进程调度程序

D 时钟中断处理程序

后 3 个选项都属于内核的功能，在内核态。命令解释程序则属于应用程序。

5. 计算机开机后，操作系统最终被加载到（D）

A BIOS

B ROM

C EPROM

D RAM

6. 操作系统属于 B

A 硬件

B 系统软件

C 通用库

D 应用软件

操作系统是管理计算机硬件与软件资源的计算机程序，例如 Windows，Linux，Android，iOS 等。应用软件一般是基于操作系统提供的接口，为针对使用者的某种应用目的所撰写的软件，例如 Office Word，浏览器，手机游戏等。而通用库，一般是指为了便于程序开发，对常用的程序功能封装后被调用的程序。
以 ucore OS 为例，它通过 I/O 子系统和各种驱动程序直接控制时钟，串口，显示器等计算机硬件外设，并通过系统调用接口给在其上运行的应用软件提供服务，并通过进程管理子系统、CPU 调度器、内存管理子系统、文件子系统、I/O 子系统来管理应用软件的运行和实现具体的服务。

7. 以下哪个不能用于描述操作系统 D

A 使计算机方便使用

B 可以管理计算机硬件

C 可以控制应用软件的执行

D 负责生成应用软件

解释：操作系统负责管理计算机的硬件资源，使得用户不需要关心硬件的工作过程，极大地方便了计算机的使用。我们日常使用计算机，往往已经在使用了特定的操作系统，例如 Windows，而在操作系统上，会同时运行多个应用软件，例如浏览器，音乐播放器等，为了让一个或者多个软件能够正常使用有限的硬件资源，操作系统需要管理应用程序的执行过程。一般来说，像浏览器，音乐播放器，和其他应用软件，都是由特定的个人和团队开发的，操作系统不负责生成应用软件。以 ucore OS 开发为例，有了 ucore OS，应用软件访问硬件跟简单了，有统一的文件访问方式来访问磁盘或各种外设。ucore OS 可以通过 I/O 操作控制硬件和应用软件的运行等。但编写软件是程序员的工作，把基于 C 语言或汇编语言的程序转换并生成执行代码是编译器（如 gcc,gas）、连接器 (如 link) 的工作。操作系统可加载运行应用软件的执行代码。

8. 以下不属于操作系统的功能是 C

A 进程调度

B 内存管理

C 视频编辑

D 设备驱动

视频编辑是一个特定的功能，不是系统范围内的共性需求，具体完成这个功能的是视频编辑应用软件。当然，视频编辑应用软件在涉及文件访问时，是需要操作系统中的文件子系统支持；在涉及视频显示方面，需要操作系统的显卡/GPU 等设备驱动支持。

9. 操作系统中的多道程序设计方式用于提高 B

A 稳定性

B 效率

C 兼容性

D 可靠性

解释：是在计算机内存中同时存放几道相互独立的程序，使它们在管理程序（早期的操作系统）控制之下，相互穿插的运行。两个或两个以上程序在计算机系统中同处于开始到结束之间的状态。这样可以使得几道独立的程序可以并发地共同使用各项硬件资源，提高了资源的利用率。以 ucore OS 为例，在 lab5 中支持了用户进程，从而可以在内存中存放多个程序，并以进程的方式被操作系统管理和调度。

10. 下面对于分时操作系统的说法，正确的是（C）

A 应用程序执行的先后顺序是完全随机的

B 应用程序按照启动的时间依次执行

C 应用程序可以交替执行

D 应用程序等待的时间越长，下一次调度被选中的概率一定越大

解释：选择 3 更合适。分时操作系统把多个程序放到内存中，将处理机（CPU）时间按一定的时间间隔（简称时间片）分配给程序运行，这样 CPU 就可以轮流地切换给各终端用户的交互式程序使用。由于时间片很短，远小于用户的交互响应延迟，用户感觉上好像独占了这个计算机系统。应用程序执行的先后顺序主要是由操作系统的调度算法和应用程序本身的行为特征来确定的。调度算法需要考虑系统的效率、公平性等因素。对于 1,2 而言，从系统的效率上看不会带来好处；对于 4 而言，可以照顾到公平性，但“一定”的表述太强了，比如如果调度算法是简单的时间片轮转算法（在后续章节“处理器调度”），则 4 的要求就不会满足了，且更实际的调度算法其实还需考虑等待的事件等诸多因素。以 ucore OS 为例，在 lab6 中支持实现不同的调度算法。对于分时操作系统而言，体现其特征的一个关键点就是要实现时间片轮转调度算法或多级反馈队列调度算法（在后续章节“处理器调度”）。在 ucore OS 中，可以比较方便地实现这两种调度算法。

11. Unix 操作系统属于 (A)

A 分时操作系统

B 批处理操作系统

C 实时操作系统

D 分布式操作系统

解释：选择 1 更合适。Unix 操作系统支持交互式应用程序，属于分时操作系统。比早期的批处理操作系统要强大。且它更多地面向桌面和服务器领域，并没有很强的实时调度和实时处理功能，所以一边不划归为实时系统。它虽然有网络支持（如 TCP/IP），但实际上它管理的主要还是单个计算机系统让的硬件和应用软件。以 ucore OS 为例，它模仿的是 Unix 操作系统，实现了对应的分时调度算法（时间片轮转、多级反馈队列），所以也算是分时系统。如果 ucore 实现了实时进程管理、实时调度算法，并支持在内核中的抢占（preempt in kernel），则可以说它也是一个实时系统了。

12. 批处理的主要缺点是 (B)

A 效率低

B 失去了交互性

C 失去了并行性

D 以上都不是

解释：批处理操作系统没有考虑人机交互所需要的分时功能，所以开发人员或操作人员无法及时与计算机进行交互。以 ucore OS 为例，如果它实现的调度算法是先来先服务调度算法（在后续章节“处理器调度”，相对其他调度算法，具体实现更简单），那它就是一种批处理操作系统了，没有很好的人机交互能力。

13. 关于操作系统，说法正确的是（ABC）

A 操作系统属于软件

B 操作系统负责资源管理

C 操作系统使计算机的使用更加方便

D 操作系统必须要有用户程序才能正常启动

操作系统是一种软件，特定指是系统软件，其更功能是管理计算机资源，让用户和应用程序更方便高效地使用计算机。以 ucore OS 为例，其实没有用户程序，操作系统也可以正常运行。所以选项 4 是不对的。

14. 设备管理的功能包括（ACD）

A 设备的分配和回收

B 进程调度

C 虚拟设备的实现

D 外围设备启动

进程调度是属于操作系统的进程管理和处理器调度子系统要完成的工作，与设备管理没有直接关系以 ucore OS 为例（lab5 以后的实验），与进程调度相关的实现位于 kern/process 和 kern/schedule 目录下；与设备管理相关的实现主要位于 kern/driver 目录下

15. 多道批处理系统主要考虑的是 (CD)

A 交互性

B 及时性

C 系统效率

D 吞吐量

解释：交互性和及时性是分时系统的主要特征。多道批处理系统主要考虑的是系统效率和系统的吞吐量。以 ucore OS 为例（lab6 实验）, 这主要看你如何设计调度策略了，所以如果实现 FCFS(先来想服务) 调度算法，这可以更好地为多道批处理系统服务；如果实现时间片轮转（time-slice round robin）调度算法，则可以有比较好的交互性；如果采用多级反馈队列调度算法，则可以兼顾上述 4 个选项，但交互性用户程序获得 CPU 的优先级更高。

第二章　启动、中断、异常和系统调用

2.1　BIOS

内存分为 2 个部分：RAM 和 ROM。

CPU 加电，初始化寄存器完成后，到内存中的 BIOS 启动固件中读取：

CS:IP=0xf000:fff0，CS 代码段寄存器，IP 指令指针寄存器，系统处于实模式。

PC=16*CS+IP，20 位地址空间，1MB

BIOS 启动固件的功能：

- 基本输入输出的程序
- 系统设置信息
- 开机后自检程序
- 系统自启动程序

BIOS 工作后：

- 将加载程序从磁盘的引导扇区 (512 字节) 加载到 0x7c00
- 跳转到 CS:IP=0000:7c00(加载程序)

加载程序：

- 将操作系统的代码和数据从硬盘加载到内存中
- 跳转到操作系统的起始地址

2.1.1　BIOS 系统调用

- BIOS 以中断调用的方式提供了基本的 I/O 功能

  INT 10h：字符显示

  INT 13h：磁盘扇区读写

INT 15h：检测内存大小

INT 16h：键盘键入

- 只能在 x86 的实模式下访问

2.2 系统启动流程

1. BIOS：系统加电，BIOS 初始化硬件；
2. 主引导记录：BIOS 读取主引导扇区代码；
3. 活动分区：主引导扇区代码读取活动分区的引导扇区代码；
4. 加载程序：引导扇区代码读取文件系统的加载程序。

2.2.1 CPU 初始化

1. CPU 初始化

   CPU 加电稳定后从 0XFFFF0 读第一条指令

   - CS:IP=0xf000:fff0
   - 第一条指令是跳转指令

   CPU 初始化状态为 16 位实模式

   - CS:IP 是 16 位寄存器
   - 指令指针 PC=16*CS+IP
   - 最大地址空间是 1MB（只能是最下面的 1MB）

2. 硬件自检 POST

   - 检测系统中内存和显卡等关键部件的存在和工作状态
   - 查找并执行显卡等接口卡 BIOS，进行设备初始化

3. 执行系统的 BIOS，进行系统检测：检测和配置系统中安装的即插即用设备
4. 更新 CMOS 中的扩展系统配置数据 ESCD
5. 按指定启动顺序从软盘、硬盘或光驱启动

2.2.2 主引导记录 MBR 格式

启动代码：446 字节（总共 512 字节）

- 检查分区表正确性
- 加载并跳转到磁盘上的引导程序

硬盘分区表：64 字节

- 描述分区状态和位置
- 每个分区描述信息占据 16 字节

结束标志字：2 字节（55AA）

- 主引导记录的有效标志

2.2.3 分区引导扇区格式

跳转指令：跳转到启动代码：与平台相关代码
文件卷头：文件系统描述信息
启动代码：跳转到加载程序
结束标志：55AA

2.2.4 加载程序 bootloader

1. 加载程序：从文件系统中读取启动配置信息
2. 启动菜单：可选的操作系统内核列表和加载参数
3. 操作系统内核：依据配置加载指定内核并跳转到内核执行

2.2.5 系统启动规范

1. BIOS
    - 固化到计算机主板上的程序
    - 包括系统设置、自检程序和系统自启动程序
    - BIOS-MBR(最多 4 个分区)、BIOS-GPT(多分区)、PXE(网络启动)
2. UEFI
    - 接口标准
    - 在所有平台上一致的操作系统启动服务

2.3 中断、异常和系统调用比较

2.3.1 背景

为什么需要中断、异常和系统调用

- 在计算机运行中，内核是被信任的第三方
- 只有内核可以执行特权指令
- 方便应用程序

为什么需要中断、异常和系统调用

- 当外设连接计算机时，会出现什么现象？
- 当应用程序处理意想不到的行为时，会出现什么现象？

系统调用希望解决的问题

- 用户应用程序是如何得到系统服务？
- 系统调用和功能调用的不同之处是什么？

系统调用 (system call)：应用程序主动向操作系统发出的服务请求；
异常 (exception)：非法指令或者其它原因导致当前**指令执行失败**后的处理强求。
中断 (hardware interrupt)：来自硬件设备的处理请求。

2.3.2 中断、异常和系统调用比较

1. 源头：
    - 中断：外设
    - 异常：应用程序意想不到的行为
    - 系统调用：应用程序请求操作提供服务
2. 响应方式
    - 中断：异步
    - 异常：同步
    - 系统调用：异步或同步
3. 处理机制

- 中断：持续，对用户应用程序是透明的
- 异常：杀死或者更细重新执行意想不到的应用程序指令
- 系统调用：等待和持续

2.3.3 中断处理机制

1. 硬件处理：在 CPU 初始化时设置中断使能标志，

   依据内部或外部事件设置中断标志

   依据中断向量调用相应中断服务例程

2. 软件

   - 现场保存 (编译器)
   - 中断服务处理 (服务例程)
   - 清除中断标记 (服务例程)
   - 现场恢复 (编译器)

2.3.4 中断嵌套

1. 硬件中断服务例程可被打断

   (a) 不同硬件中断源可能硬件中断处理时出现

   (b) 硬件中断服务例程中需要临时禁止中断请求

   (c) 中断请求会保持 CPU 做出响应

2. 异常服务例程可被打断

   - 异常服务例程执行时可能出现硬件中断

3. 异常服务例程可嵌套

   - 异常服务例程可能出现缺页

2.3.5 问题

1. 下列选项中，不可能在用户态发生的是（C ）

   A 系统调用

   B 外部中断

C 进程切换

D 缺页

系统调用是提供给应用程序使用的，由用户态发出，进入内核态执行。外部中断随时可能发生；应用程序执行时可能发生缺页；进程切换完全由内核来控制。

2. 中断处理和子程序调用都需要压栈以保护现场。中断处理一定会保存而子程序调用不需要保存其内容的是（B ）

A 程序计数器

B 程序状态字寄存器

C 通用数据寄存器

D 通用地址寄存器

程序状态字（PSW）寄存器用于记录当前处理器的状态和控制指令的执行顺序，并且保留与运行程序相关的各种信息，主要作用是实现程序状态的保护和恢复。所以中断处理程序要将 PSW 保存，子程序调用在进程内部执行，不会更改 PSW。

3. 中断向量地址是（B ）

A 子程序入口地址

B 中断服务例程入口地址

C 中断服务例程入口地址的地址

D 例行程序入口地址

4. 下列选项中，() 可以执行特权指令？

A 中断处理例程

B 普通用户的程序

C 通用库函数

D 管理员用户的程序

中断处理例程（也可称为中断处理程序）需要执行打开中断，关闭中断等特权指令，而这些指令只能在内核态下才能正确执行，所以中断处理例程位于操作系统内核中。而 1,3,4 都属于用户程序和用于用户程序的程序库。以 ucore OS 为例，在 lab1 中就涉及了中断处理例程，可查看 intr_enable，sti，trap 等函数完成了啥事情? 被谁调用了?

5. 一般来讲，中断来源于（A）

A 外部设备

B 应用程序主动行为

C 操作系统主动行为

D 软件故障

中断来源与外部设备，外部设备通过中断来通知 CPU 与外设相关的各种事件。第 2 选项如表示是应用程序向操作系统发出的主动行为，应该算是系统调用请求。第 4 选项说的软件故障也可称为软件异常，比如除零錯等。以 ucore OS 为例，外设产生的中断典型的是时钟中断、键盘中断、串口中断。在 lab1 中，具体的中断处理例程在 trap.c 文件中的 trap_dispatch 函数中有对应的实现。对软件故障/异常的处理也在 trap_dispatch 函数中的相关 case default 的具体实现中完成。在 lab1 的 challenge 练习中和 lab5 中，有具体的系统调用的设计与实现。

6. 用户程序通过 _____ 向操作系统提出访问外部设备的请求（B）

A I/O 指令

B 系统调用

C 中断

D 创建新的进程

具体内容可参见 10. 的回答。以 ucore OS 为例，在 lab5 中有详细的 syscall 机制的设计实现。比如用户执行显示输出一个字符的操作，由于涉及向屏幕和串口等外设输出字符，需要向操作系统发出请求，具体过程是应用程序运行在用户态，通过用户程序库函数 cputch，会调用 sys_putc 函数，并进一步调用 syscall 函数（在 usr/libs/syscall.c 文件中），而这个函数会执行“int 0x80”来发出系统调用请求。在 ucore OS 内核中，会接收到这个系统调用号（0x80）的中断（参见 kernel/trap/trap.c 中的 trap_dispatch 函数有关“case T_SYSCALL:”的实现）, 并进一步调用内核 syscall 函数（参见 kernel/syscall/syscall.c 中的实现）来完成用户的请求。内核在内核态（也称特权态）完成后，通过执行“iret”指令（kernel/trap/trapentry.S 中的“__trapret:”下面的指令），返回到用户态应用程序发出系统调用的下一条指令继续执行应用程序。

7. 应用程序引发异常的时候，操作系统可能的反应是（C）

A 删除磁盘上的应用程序

B 重启应用程序

C 杀死应用程序

D 修复应用程序中的错误

更合适的答案是 3。因为应用程序发生异常说明应用程序有错误或 bug，如果应用程序无法应对这样的错误，这时再进一步执行应用程序意义不大。如果应用程序可以应对这样的错误（比如基于当前 c++ 或 java 的提供的异常处理机制，或者基于操作系统的信号（signal）机制（后续章节“进程间通信”会涉及）），则操作系统会让应用程序转到应用程序的对应处理函数来完成后续的修补工作。以 ucore OS 为例，目前的 ucore 实现在应对应用程序异常时做的更加剧烈一些。在 lab5 中有有对用户态应用程序访问内存产生错误异常的处理（参见 kernel/trap/trap.c 中的 trap_dispatch 函数有关“case T_PGFLT: ”的实现），即 ucore 判断用户态程序在运行过程中发生了内存访问错误异常，这是 ucore 认为重点是查找错误，所以会调用 panic 函数，进入 kernel 的监控器子系统，便于开发者查找和发现问题。这样 ucore 也就不再做正常工作了。当然，我们可以简单修改 ucore 当前的实现，不进入内核监控器，而是直接杀死进程即可。你能完成这个修改吗？

8. 操作系统与用户的接口包括 _____（A）

   A 系统调用

   B 进程调度

   C 中断处理

   D 程序编译

   更合适的答案是 1。根据对当前操作系统设计与实现的理解，系统调用是应用程序向操作系统发出服务请求并获得操作系统服务的唯一通道和结果。

9. 操作系统处理中断的流程包括 _____（ABCD）

   A 保护当前正在运行程序的现场

   B 分析是何种中断，以便转去执行相应的中断处理程序

   C 执行相应的中断处理程序

   D 恢复被中断程序的现场

   中断是异步产生的，会随时打断应用程序的执行，且在操作系统的管理之下，应用程序感知不到中断的产生。所以操作系统需要保存被打断的应用程序的执行现场，处理具体的中断，然后恢复被打断的应用程序的执行现场，使得应用程序可以继续执行。以 ucore OS 为例（lab5 实验），产生一个中断 XX 后，操作系统的执行过程如下：vectorXX(vectors.S)–> ___alltraps(trapentry.S)–>trap(trap.c)–>trap_dispatch(trap.c)–>–>……具体的中断处理–>___trapret(trapentry.S) 通过查看上述函数的源码，可以对应到答案 1-4。另外，需要注意，在 ucore 中，应用程序的执行现场其实保存在 trapframe 数据结构中。

10. 下列程序工作在内核态的有 _____(ABCD)

A 系统调用的处理程序

B 中断处理程序

C 进程调度

D 内存管理

这里说的“程序”是一种指称，其实就是一些功能的代码实现。而 1-4 都是操作系统的主要功能，需要执行相关的特权指令，所以工作在内核态。以 ucore OS 为例（lab5 实验），系统调用的处理程序在 kern/syscall 目录下，中断处理程序在 kern/trap 目录下，进程调度在 kern/schedule 目录下，内存管理在 kern/mm 目录下

2.4 系统调用

2.4.1 系统调用

- 操作系统服务的编程接口
- 通常由高级语言编写
- 程序访问通常是通过高层次的 API 接口而不是直接进行系统调用
- 三种最常用的应用程序编程接口 API
  1. Win32 API
  2. POSIX API
  3. Java API

2.4.2 系统调用的实现

1. 每个系统调用对应一个系统调用号
   - 系统调用接口根据系统调用号来维护表的索引
2. 系统调用接口调用内核态中的系统调用功能实现，并返回系统调用的状态和结果
3. 用户不需要知道系统调用的实现
   - 需要设置调用参数和返回结果
   - 操作系统接口的细节大部分都隐藏在应用编程接口后
   - 通过运行程序支持的库来管理

2.4.3 系统调用和函数调用的不同处

1. 系统调用

   - INT 和 IRET 指令用于系统调用

     系统调用时，堆栈切换和特权级的转换

2. 函数调用

   - CALL 和 RET 用于常规调用

     常规调用时没有堆栈切换

2.4.4 中断、异常和系统调用的开销

1. 超过函数调用

2. 中断、异常和系统调用

   引导机制

   建立内核堆栈

   验证参数

   内核态映射到用户态的地址空间：更新页面映射权限

   内核态独立地址空间：TLB

2.4.5 问题

1. CPU 执行操作系统代码的时候称为处理机处于（C）

   A 自由态

   B 目态

   C 管态

   D 就绪态

   内核态又称为管态

2. 下列选项中，会导致用户进程从用户态切换到内核态的操作是（B ）1）整数除以 0 2）sin() 函数调用 3）read 系统调用

   A 1、2

   B 1、3

C 2、3

D 1、2、3

函数调用并不会切换到内核态，而除零操作引发中断，中断和系统调用都会切换到内核态进行相应处理。

3. 系统调用的主要作用是（C）

A 处理硬件问题

B 应对软件异常

C 给应用程序提供服务接口

D 管理应用程序

应用程序一般无法直接访问硬件，也无法执行特权指令。所以，需要通过操作系统来间接完成相关的工作。而基于安全和可靠性的需求，应用程序运行在用户态，操作系统内核运行在内核态，导致应用程序无法通过函数调用来访问操作系统提供的各种服务，于是通过系统调用的方式就成了应用程序向 OS 发出请求并获得服务反馈的唯一通道和接口。以 ucore OS 为例，在 lab1 的 challenge 练习中和 lab5 中，系统调用机制的初始化也是通过建立中断向量表来完成的（可查看 lab1 的 challenge 的答案中在 trap.c 中 idt_init 函数的实现），中断向量表描述了但应用程序产生一个用于系统调用的中断号时，对应的中断服务例程的具体虚拟地址在哪里，即建立了系统调用的中断号和中断服务例程的对应关系。这样当应用程序发出类似“int 0x80”这样的指令时（可查看 lab1 的 challenge 的答案中在 init.c 中 lab1_switch_to_kernel 函数的实现），操作系统的中断服务例程会被调用，并完成相应的服务（可查看 lab1 的 challenge 的答案中在 trap.c 中 trap_dispatch 函数有关“case T_SWITCH_TOK:”的实现）。

4. 下列关于系统调用的说法错误的是（B）

A 系统调用一般有对应的库函数

B 应用程序可以不通过系统调用来直接获得操作系统的服务

C 应用程序一般使用更高层的库函数而不是直接使用系统调用

D 系统调用可能执行失败

更合适的答案是 2。根据对当前操作系统设计与实现的理解，系统调用是应用程序向操作系统发出服务请求并获得操作系统服务的唯一通道和结果。如果操作系统在执行系统调用服务时，产生了错误，就会导致系统调用执行失败。以 ucore OS 为例，在用户态的应用程序（lab5,6,7,8 中的应用程序）都是通过系统调用来获得操作系统的服务的。为了简化应用程序发出系统调用请求，ucore OS 提供了用户态的更高层次的库函数

（user/libs/ulib.[ch] 和 syscall.[ch]），简化了应用程序的编写。如果操作系统在执行系统调用服务时，产生了错误，就会导致系统调用执行失败。

5. 以下关于系统调用和常规调用的说法中，错误的是（D）

A 系统调用一般比常规函数调用的执行开销大

B 系统调用需要切换堆栈

C 系统调用可以引起特权级的变化

D 常规函数调用和系统调用都在内核态执行

系统调用相对常规函数调用执行开销要大，因为这会涉及到用户态栈和内核态栈的切换开销，特权级变化带来的开销，以及操作系统对用户态程序传来的参数安全性检查等开销。如果发出请求的请求方和应答请求的应答方都在内核态执行，则不用考虑安全问题了，效率还是需要的，直接用常规函数调用就够了。以 ucore OS 为例，我们可以看到系统调用的开销在执行“int 0x80”和“iret”带来的用户态栈和内核态栈的切换开销，两种特权级切换带来的执行状态（关注 kern/trap/trap.h 中的 trapframe 数据结构）的保存与恢复等（可参看 kern/trap/trapentry.S 的 ___alltraps 和 ___trapret 的实现）。而函数调用使用的是”call” 和“ret”指令，只有一个栈，不涉及特权级转变带来的各种开销。如要了解 call, ret, int 和 iret 指令的具体功能和实现，可查看“英特尔 64 和 iA-32 架构软件开发人员手册卷 2a’s, 指令集参考（A-M）”和“英特尔 64 和 iA-32 架构软件开发人员手册卷 2B’ s, 指令集参考（N-Z）”一书中对这些指令的叙述。

2.5 系统调用示例

2.5.1 例子-文件复制

文件复制过程中的系统调用序列

```
1 获取输入文件名
2     显示提示
3     等待键盘输入
4 获取输出文件名
5     显示提示
6     等待键盘输入
7 打开输入文件
8     文件不存在，退出
9 创建输出文件
```

```
10      如果文件存在，退出
11  循环
12      读取输入文件
13      写入输出文件
14  直到读取结束
15  关闭输出文件
16  屏幕显示提示完成
17  正常退出
```

涉及的系统调用

```
1  #define SYS_write 5
2  #define SYS_read 6
3  #define SYS_close 7
4  #define SYS_open 10
```

在 ucore 中库函数 read() 的功能是读文件

usr/libs/file.h:int read(int fd, void * buf, int length)

库函数 read() 的参数和返回值

- int fd 文件句柄
- void * buf 数据缓冲区指针
- int length 数据缓冲区长度
- int return_value 返回读出数据长度

库函数 read() 使用示例

int sfs_filetest1.c: ret = read(fd, data, len)

2.5.2 系统调用 read 的实现

1. kern/trap/trapentry.S: alltraps()
2. kern/trap/trap.c: trap()

   tf->trapno == T_SYSCALL
3. kern/syscall/syscall.c: syscall()

   tf->tf_regs.reg_eax == SYS_read

4. kern/syscall/syscall.c: sys_read()

   从 tf->sp 获取 fd, buf, length

5. kern/fs/sysfile.c: sysfile_read()

   读取文件件

6. kern/trap/trapentry.S: trapret()

第三章　物理内存管理：连续内存分配

3.1　计算机体系结构和内存层次

内存管理：

1. 抽象

   逻辑地址空间

2. 保护

   独立地址空间

3. 共享

   访问相同的内存

4. 虚拟化

   更大的地址空间

操作系统的内存管理方式

1. 操作系统采用的内存管理方式

   重定位 (relocation)

   分段 (segmentation)

   分页 (paging)

   虚拟存储 (virtual memory)

   目前多数系统采用按需页式虚拟存储

2. 实现高度依赖硬件

   与计算机存储架构紧耦合

   MMU(内存管理单元)：处理 CPU 存储访问请求的硬件

3.2 地址空间和地址生成

3.2.1 地址空间的定义

1. 物理地址空间——硬件支持的地址空间

   起始地址 0，直到 MAX_{sys}

2. 逻辑地址空间——在 CPU 运行的进程看到的地址

   起始地址 0，直到 MAX_{prog}

3.2.2 逻辑地址的生成

高级语言 → 汇编语言 → 进一步汇编 → 链接，加上函数库 → 重定位，程序加载

3.2.3 地址生成时机和限制

1. 编译时

   (a) 假设起始地址已知

   (b) 如果起始地址改变，必须重新编译

2. 加载时

   (a) 如编译时起始位置未知，编译器需生成可重定位的代码 (relocatable code)

   (b) 加载时，生成绝对地址

3. 执行时

   (a) 执行时代码可移动

   (b) 需地址转换 (映射) 硬件支持

3.2.4 地址生成过程

1. CPU

   (a) ALU：需要逻辑地址的内存内容

   (b) MMU：进行逻辑地址和物理地址的转换

   (c) CPU 控制逻辑：给总线发送物理地址请求

2. 内存

(a) 发送物理地址的内容给 CPU

(b) 或接受 CPU 数据到物理地址

3. 操作系统

建立逻辑地址 LA 和物理地址 PA 的映射

3.2.5 地址检查

进程 P 逻辑地址空间 →CPU→ 逻辑地址，端长度寄存器，检查内存是否异常（检查偏移量）→ 如果正常，交给段基址寄存器，生成物理地址 → 到内存中找出数据

操作系统影响：段长度寄存器、段基址寄存器，设置起始（base）和最大（limit）逻辑地址空间

3.3 连续内存分配

3.3.1 连续内存分配和内存碎片

- 计算机体系结构/内存层次

  给进程分配一块不小于指定大小的连续的物理内存区域

- 内存碎片

  空闲内存不能被使用

- 外部碎片

  分配单元之间的未被利用内存

- 内部碎片

  分配单元内部的未被使用内存

  取决于分配单元大小是否要取整

3.3.2 连续内存分配：动态分区分配

- 动态分区分配

  1. 当程序被加载执行时，分配一个进程指定大小可变的分区 (块、内存块)
  2. 分区的地址是连续的

- 操作系统需要维护的数据结构

1. 所有进程的已分配分区
2. 空闲分区 (Empty-blocks)

- **动态分区分配策略**

1. 最先匹配 (First-fit)
2. 最佳匹配 (Best-fit)
3. 最差匹配 (Worst-fit)

3.3.3 最先匹配 (First Fit Allocation) 策略

思路：分配 n 个字节，使用第一个可用的空间比 n 大的空闲块

原理 & 实现

1. 空闲分区列表按地址顺序排序
2. 分配过程时，搜索一个合适的分区
3. 释放分区时，检查是否可与临近的空闲分区合并

优点

1. 简单
2. 在高地址空间有大块的空闲分区

缺点

1. 外部碎片
2. 分配大块时较慢

3.3.4 最佳匹配 (Best Fit Allocation) 策略

思路：分配 n 字节分区时，查找并使用不小于 n 的最小空闲分区

原理 & 实现

1. 空闲分区列表按照大小排序
2. 分配时，差找一个合适的分区
3. 释放时，查找并合并临近的空闲分区 (如果找到)

优点

1. 大部分分配的尺寸较小时，效果很好

   可避免大的空闲分区被拆分

   可减少外部碎片的大小

   相对简单

缺点

1. 外部碎片
2. 释放分区较慢
3. 容易产生很多无用的小碎片

3.3.5 最差匹配 (Worst Fit Allocation) 策略

思路：分配 n 字节，使用尺寸不小于 n 的最大空闲分区

原理 & 实现

1. 空闲分区列表按照大小排序
2. 分配时，选最大的分区
3. 释放时，查找并合并临近的空闲分区，进行可能的合并，并调整空闲分区列表顺序

优点

1. 中等大小的分配较多时，效果最好
2. 避免出现太多的小碎片

缺点

1. 外部碎片
2. 释放分区较慢
3. 容易破坏大的空闲分区，因此后续难以分配大的分区

3.4 碎片整理

3.4.1 碎片整理：紧凑 (compaction)

碎片整理：通过调整进程占用的分区位置来减少或避免分区碎片

碎片紧凑：通过分配给进程的内存分区，以合并外部碎片。

碎片紧凑的条件：所有的应用程序可动态重定位

注：什么时候移动，开销

3.4.2 碎片整理：分区对换 (Swapping in/out)

分区对换：通过抢占并回收等待状态进程的分区，以增大可用内存空间。
在早期内存紧张时，可实现多进程交替。

3.5 伙伴系统

3.5.1 伙伴系统 (Buddy System)

- 约定整个可分配的分区的大小为 2^U
- 需要的分区大小为 $2^{U-1} < s \leq 2^U$ 时，把整个块分配给该进程

  如 $s \leq 2^{i-1}$，将大小为 2^i 的空闲分区划分为两个大小为 $2^{i-1} - 1$ 的空闲分区

  重复切块过程，直到 $2^{i-1} < s \leq 2^i$，并把一个空闲分区分配给该进程

3.5.2 伙伴系统的实现

数据结构

- 空闲块按大小和起始地址组织成二维数组
- 初始状态：只有一个大小为 2^U 的空闲块

分配过程

- 由小到大在空闲块数组中找最小的可用空闲块
- 如果空闲块过大，对可用空闲块进行二等分，直到得到合适的可用空闲块

释放过程

- 把释放的块放入空闲块数组
- 合并满足合并条件的空闲块

合并条件

- 大小相同 2^i
- 地址相邻
- 起始地址较小的块的起始地址必须是 2^{i+1} 的倍数

3.5.3 ucore 中的物理内存管理

```
1 struct pmm_manager {
2     const char *name;
3     void (*init)(void);
4     void (*init_memmap)(struct Page *base, size_t n);
5     struct Page *(*alloc_pages)(size_t order);
6     void (*free_pages)(struct Page *base, size_t n);
7     size_t (*nr_free_pages)(void);
8     void (*check)(void);
9 }
```

3.6 问题

1. 在启动页机制的情况下，在 CPU 运行的用户进程访问的地址空间是 (B)

   A 物理地址空间

   B 逻辑地址空间

   C 外设地址空间

   D 都不是

   用户进程访问的内存地址是虚拟地址

2. 在使能分页机制的情况下，更合适的外碎片整理方法是 (C)

   A 紧凑 (compaction)

   B 分区对换 (Swapping in/out)

   C 都不是

   分页方式不会有外碎片

3. 操作系统中可采用的内存管理方式包括 (ABCD)

   A 重定位 (relocation)

   B 分段 (segmentation

   C 分页 (paging)

D 段页式（segmentation+paging）

4. 连续内存分配的算法中，会产生外碎片的是 (ABC)

A 最先匹配算法

B 最差匹配算法

C 最佳匹配算法

D 都不会

5. 描述伙伴系统 (Buddy System) 特征正确的是 (ABC)

A 多个小空闲空间可合并为大的空闲空间

B 会产生外碎片

C 会产生内碎片

D 都不对

第四章　物理内存管理：非连续内存分配

4.1　非连续内存分配的需求背景

4.1.1　非连续分配的设计目标

连续分配的特点：

1. 分配给程序的物理内存必须连续；
2. 存在外碎片和内碎片；
3. 内存分配的动态修改困难；
4. 内存利用率较低。

非连续分配的设计目标：提高内存利用效率和管理灵活性。

1. 允许一个程序的使用非连续的物理地址空间；
2. 允许共享代码与数据；
3. 支持动态加载和动态链接。

非连续分配需要解决的问题：

1. 如何实现虚拟地址和物理地址的转换；

   软件实现（灵活，开销大）

   硬件实现（够用，开销小）

非连续分配的硬件辅助机制

1. 如何选择非连续分配中的内存分块大小

   段式存储管理 (segmentation)

   页式存储管理 (paging)

4.2 段式存储管理

4.2.1 段地址空间

进程的短地址空间由多个端组成

- 主代码段
- 子模块代码段
- 公用库代码段
- 堆栈段 (stack)
- 堆数据 (heap)
- 初始化代码段
- 符号表等

段式存储管理的目的：更细粒度和灵活的分离和共享。

4.2.2 段访问机制

段的概念

- 段：表示访问方式和存储数据等属性相同的一段地址空间。
- 对应一个连续的内存“块”
- 若干个段组成进程逻辑地址空间

段访问：逻辑地址由二元组（s, addr）表示

- s ——段号
- addr ——段内偏移

段访问的硬件实现：

CPU 加载程序后，根据逻辑地址的段号，查找段表（OS 设置）的段描述符，找出基址和长度。然后硬件的 MMU 根据长度和偏移地址判断是否越界（内存异常），然后 MMU 使用段基址 + 偏移量即可。

4.3 页式存储管理

4.3.1 页式存储管理的概念

页帧（物理页面，Frame，Page Frame）

- 把物理地址空间划分为大小相同的基本分配单位
- 2 的 n 次方，如 512,4096,8192

页面（页，逻辑页面，Page）

- 把逻辑地址空间划分为大小相同的基本分配单位
- 帧和页的大小必须是相同的

页面到页帧

- 逻辑地址到物理地址的转换
- 页表
- MMU/TLB

4.3.2 帧 (Frame)

帧：物理内存被划分为大小相等的帧。
内存物理地址的表示：二元组 (f, o)

- f——帧号（F 位，共有 2^F 个帧）
- o——帧内偏移（S 位，每帧有 2^S 字节）

$$\text{物理地址} = f \times 2^S + o$$

基于页帧的物理地址计算实例：
16-bit 的地址空间
9-bit(512 byte) 大小的页帧
物理地址 =(3, 6)
F=7, S=9, f=3, o=6
实际地址：$2^9 \times 3 + 6 = 1536 + 6 = 1542$

4.3.3 页 (Page)

进程逻辑地址空间被划分为大小相等的页

- 页内偏移 = 帧内偏移
- 通常：页号大小 ≠ 帧号大小

进程逻辑地址的表示：二元组 (p, o)

- p——页号 (p 位，2^P 个页)
- o——页内偏移 (S 位，每页有 2^S 字节)

虚拟地址 $= p \times 2^S + o$

4.3.4 页式存储的地址映射——页表

- 页到帧的映射
- 逻辑地址中的页号是连续的
- 物理地址中的帧号是不连续的
- **不是所有的页**都有对应的帧

页表保存了逻辑地址——物理地址之间的映射关系
CPU 加载程序获取逻辑地址（p, o），根据逻辑地址的页号，在页表中查找相应的帧号，然后根据帧号和帧内偏移相加得到实际地址。

4.4 页表概述

4.4.1 页表结构

- 每个进程都有一个页表

  每个页面对应一个页表项

  随进程运行状态而动态变化

  页表基址寄存器 (PTBR：Page Table Base Register)

页表项的组成：

- 帧号：f

- 页表项标志

  存在位 (resident bit)

  修改位 (dirty bit)

  引用位 (clock/reference bit)

页表地址转换实例：
假定：具有 16 位地址的计算机系统，物理内存大小为：32KB，每页大小：1024 字节

4.4.2 页式存储管理机制的性能问题

- 内存访问性能问题

  1. 访问一个内存单元需要 2 次内存访问
  2. 第一次访问：获取页表项
  3. 第二次访问：访问数据

- 页表大小问题

  1. 页表可能非常大
  2. 64 位机器如果每页 1024 字节，那么一个页表的大小会使多少？

- 如何处理？

  1. 缓存 Caching
  2. 间接 Indirection 访问

4.5 快表和多级页表

4.5.1 快表 Translation Look-aside Buffer,TLB

缓存近期访问的页表项

- TLB 使用关联存储 (associative memory) 实现，具备快速访问性能
- 如果 TLB 命中，物理页号可以很快被获取
- 如果 TLB 未命中，对应的表项被更新到 TLB 中

4.5.2 多级页表

通过间接引用将页号分成 k 级

- 建立页表树
- 减少每级页表的长度

4.6 反置页表

4.6.1 大地址空间问题

对于大地址空间 (64-bits) 系统，多级页表变得繁琐

- 比如：5 级页表
- 逻辑（虚拟）地址空间增长速度快于物理地址空间

页寄存器和反置页表的思路

- 不让页表与逻辑地址空间的大小相对应
- 让页表与物理地址空间的大小相对应

4.6.2 页寄存器 Page Registers

每个帧与一个页寄存器关联，寄存器内容包括：

- 使用位 (Residence bit)：此帧是否被进程占用
- 占用页号 (Occupier)：对应的页号 p
- 保护位 (Protection bits)

页寄存器示例：

- 物理内存大小：$4096\times = 4K \times 4KB = 16MB$
- 页面大小 $4096bytes = 4KB$
- 页帧数 $4096 = 4K$
- 页寄存器使用的空间（假设每个页寄存器占 8 字节）

  8×4096=32 Kbytes

- 页寄存器带来的额外开销

  32K/16M=0.2%

- 虚拟内存的大小：任意

页寄存器优点：

- 页表大小相对于物理内存而言很小
- 页表大小与逻辑地址空间大小无关

页寄存器缺点：

- 页表信息对调后，需要依据帧号可找页号
- 在页寄存器中搜索逻辑地址中的页号

4.6.3 页寄存器的地址转换

1. CPU 生成的逻辑地址如何找对应的物理地址？
   - 对逻辑地址进行 Hash 映射，以减少搜索范围
   - 需要解决可能的冲突
2. 用快表缓存页表项后的页寄存器搜索步骤：
   - 对逻辑地址进行 Hash 变换
   - 在快表中查找对应页表项
   - 有冲突时遍历冲突项链表
   - 查找失败时，产生异常
3. 快表的限制
   - 快表的容量限制
   - 快表的功耗限制 (Strong ARM 上快表功耗占 27%)

4.6.4 反置页表

基于 Hash 映射值查找对应页表项中的帧号。

- 进程标识与页号的 Hash 值可能冲突
- 页表项中保护位、修改位、访问位和存在位等标识

4.6.5 反置页表的 Hash 冲突

匹配下一项

4.7 段页式存储管理

4.7.1 段页式存储管理的需求

- 段式存储在内存保护方面有优势，页式存储在内存利用和优化转移到后备存储方面有优势

4.7.2 段页式存储管理实现

在段式存储管理基础上，给每个段加一级页表。
段页式存储管理中的内存共享

- 通过指向**相同的页表基址**，实现进程间的段共享

4.8 问题

1. 描述段管理机制正确的是 (ABCD)

   A 段的大小可以不一致

   B 段可以有重叠

   C 段可以有特权级

   D 段与段之间是可以不连续的

2. 描述页管理机制正确的是 (ABC)

   A 页表在内存中

   B 页可以是只读的

   C 页可以有特权级

   D 都不对

3. 页表项标志位包括 (ABCD)

   A 存在位 (resident bit)

   B 修改位 (dirty bit)

   C 引用位 (clock/reference bit)

D 只读位 (read only OR read/write bit)

4. 可有效应对大地址空间可采用的页表手段是 (AB)

A 多级页表

B 反置页表

C 页寄存器方案

D 单级页表

页寄存器和反置页表很像，但它们的一个区别是进程 ID 在地址转换中的使用。没有进程 ID（也就是说页寄存器方案）时，页表占用的空间仍然是与进程数相关的（也就是每个进程对应一组页寄存器?)。反置页表的大小只与物理内存大小，与并发进程数无关。采用页寄存器的硬件开销会很大。所以现在的通用 CPU（包括 64 位的 CPU）没有采用这种方式，大部分还是多级页表。由于有 TLB 作为缓存，效率还不错。

第五章　虚拟存储概念

5.1　虚拟存储的需求背景

1. 增长迅速的存储需求：程序规模的增长速度远远大于存储器容量的增长

存储层次结构

- 理想中的存储器

  容量更大、速度更快、价格更便宜的非易失性存储器

- 实际中的存储器：使用存储结构解决

5.1.1　操作系统的存储抽象

操作系统对存储的抽象：地址空间

5.1.2　虚拟存储需求

- 计算机系统时常出现内存空间不够用

  1. 覆盖 overlay

     应用程序**手动**把需要的指令和数据保存在内存中

  2. 交换 swapping

     操作系统**自动**把暂时不能执行的程序保存到外存中

  3. 虚拟存储

     在有限容量的内存中，以页为单位**自动装入更多更大**的程序

5.2　覆盖和交换

5.2.1　覆盖技术

目标：在较小的可用内存中运行较大的程序

方法：依据程序逻辑结构，将程序划分为若干**功能相对独立**的模块，将不会同时执行的模块**共享同一块内存区域**

- 必要部分（常用功能）的代码和数据常驻内存
- 可选部分（不常用部分）放在其他程序模块中，只在需要用到时装入内存
- 不存在调用关系的模块可相互覆盖，共用同一块内存区域

覆盖技术的不足：

1. 增加编程困难
    - 需要程序员划分功能模块，并确定模块间的覆盖关系
    - 增加了编程的复杂度
2. 增加执行时间
    - 从外存装入覆盖模块
    - 时间换空间

5.2.2 交换技术

目标：增加正在运行或需要运行的程序的内存。
实现方法：

- 可将暂时不能运行的程序放到外存
- 换入换出的基本单位

  整个进程的地址空间
- 换出 (swap out)

  把一个进程的整个地址空间保存到外存
- 换入 (swap in)

  将外存中某进程的地址空间读入到内存

交换技术面临的问题

- 交换时机：何时需要发生交换？

  只当内存空间不够时或有不够的可能时换出

- 交换区大小

  存放所有用户进程的所有内存映像的拷贝

- 程序换入时的重定位：换出后再换入时要放回原处吗？

  采用动态地址映射的方法

5.2.3 覆盖和交换的比较

1. 覆盖

   - 只能在没有调用关系的模块间
   - 程序员须给出模块间的逻辑覆盖结构
   - 发生在运行程序的内部模块间

2. 交换

   - 以进程为单位
   - 不需要模块间的逻辑覆盖结构
   - 发生在内存进程间

5.3 局部性原理

虚拟存储技术的目标

1. 只把部分程序放到内存中，从而运行比物理内存大的程序。

   由操作系统自动完成，无需程序员的干涉；

2. 实现进程在内存与外存之间的交换，从而获得更多的空闲内存空间

   在内存与外存之间只交换进程的部分内容

5.3.1 局部性原理 (principle of locality)

局部性原理：程序在执行过程中的一个较短时期，所执行的指令地址和指令的操作数地址，分别局限于一定区域。

1. 时间局部性

   一条指令的一次执行和下次执行，一个数据的一次访问和下次访问都集中在一个较短的时期内

2. 空间局部性

当前指令和邻近的几条指令，当前访问的数据和邻近的几个数据都集中在一个较小的区域内

3. 分支局部性

一条跳转指令的两次执行，很可能跳到相同的内存位置

4. 局部性原理的意义

从理论上来说，虚拟存储技术是能够实现的，而且可取得满意的结果

程序编写：二维数组的遍历顺序。

5.4 虚拟存储概念

虚拟存储思路：

- 将不常用的部分内存块暂存到外存中

虚拟存储原理：

1. 装载程序时

只将当前指令执行需要的部分页面或段装入内存

2. 指令执行中需要的指令或数据不在内存（称为缺页或缺段）时

处理器通知操作系统将相应的页面或段调入内存

3. 操作系统将内存中暂时不用的页面或段保存到外存中

实现方式

- 虚拟页式存储
- 虚拟段式存储

虚拟存储的基本特征：

1. 不连续性

物理内存分配非连续

虚拟地址空间使用非连续

2. 大用户空间

提供给用户的虚拟内存可大于实际的物理内存

3. 部分交换

虚拟存储只对部分虚拟地址空间进行调入和调出

5.4.1 虚拟存储的支持技术

1. 硬件

   页式或短时存储中的地址转换机制

2. 操作系统

   管理内存和外存间页面或段的换入和换出

5.5 虚拟页式存储

5.5.1 虚拟页式存储管理

- 在页式存储管理的基础上，增加请求调页和页面置换
- 思路

  1. 当用户程序要装载到内存运行时，只装入部分页面，就启动程序执行
  2. 进程在运行中发现有需要的代码或数据不在内存时，则向系统发出缺页异常请求
  3. 操作系统在处理缺页异常时，将外存中相应的页面调入内存，使得进程能继续运行

虚拟页式存储在页表中加一个无效项，当查询无效时，交给操作系统处理，然后由操作系统进行查找缺页。

5.5.2 虚拟页式存储中的页表项结构

页表结构：
逻辑页号 + 访问位 + 修改位 + 保护位 + 驻留位 + 物理页帧号

1. 驻留位：表示该页是否在内存

   1 表示该页位于内存中，该页表项是有效的，可以使用

   0 表示该页当前在外存中，访问该页表项将导致缺页异常

2. 修改位：表示在内存中的该页是否被修改

   回收该物理页面时，据此判断是否要把它的内容写回外存

3. 访问位：表示该页面是否被访问过（读或写）

   用于页面置换算法

4. 保护位：表示该页的允许访问方式

   只读、可读写、可执行

5.6 缺页异常

5.6.1 缺页异常的处理流程

A load M，查找页表

B 页表驻留位为 0，缺页异常

C 操作系统缺页异常服务处理例程查找在外存中的页面

D 查找到后，页面换入

E 页表项修改

F 重新执行导致异常的指令

A. 在内存中有空闲物理页，分配一物理页帧，转至 E 步
B. 依据页面置换算法选择将被替换的物理页帧 f，对应逻辑页 q
C. 如果 q 被修改过，则把它写回外存
D. 修改 q 的页表项中驻留位置为 0
E. 将需要访问的页 p 装入到物理页面 f
F. 修改 p 的页表项驻留位为 1，物理页帧号为 f。
G. 重新执行产生缺页的指令。

5.6.2 虚拟页式存储中的外存管理

在何处保存未被映射的页？

- 应能方便地找到在外存中的页面内容
- 交换空间（磁盘或者文件）

  采用特殊格式存储未被映射的页面

虚拟页式存储的外存选择

- 代码段：可执行二进制文件（不放交换空间）
- 动态加载的共享库程序段：动态调用的库文件（不放交换空间）
- 其他段：交换空间

5.6.3 虚拟页式存储管理的性能

1. 有效存储访问时间 (effective memory access time EAT)
   $EAT =$ 访存时间 $\times (1-p) +$ 缺页异常处理时间 $\times$ 缺页率p

例子：访存时间 10ns，磁盘访问时间 5ms，缺页率 p，页修改率 q

$$EAT = 10 \times (1-p) + 5,000,000 \times p \times (1+q)$$

第六章　页面置换算法

6.1　页面置换算法的概念

6.1.1　置换算法的功能和目标

功能：

- 当出现缺页异常，需调入新页面而内存已满时，置换算法**选择被置换的物理页面**

设计目标：

- 尽可能**减少页面的调入调出次数**
- 把未来不再访问或短期内不访问的页面调入

页面锁定 (frame locking)

- 描述必须常驻内存的逻辑页面
- 操作系统的关键部分
- 要求响应速度的代码和数据
- 页表中的锁定标志位 (lock bit)

6.1.2　置换算法的评价方法

1. 记录进程访问内存的页面轨迹；
   虚拟地址访问用 (页号,位移) 表示:(3,0),(1,9),(4,1),(2,1),(5,3), (2,0),(1,9),(2,4),(3,1),(4,8)
   页面轨迹：3,1,4,2,5,2,1,2,3,4→c,a,d,b,e,b,a,b,c,d

2. 评价方法

   模拟页面置换行为，记录产生缺页的次数

   更少的缺页，更好的性能

6.1.3 页面置换算法分类

1. 局部页面置换算法

   置换页面的选择范围仅限于当前进程占用的物理页面内

   最优算法、先进先出算法、最近最久未使用算法

   时钟算法、最不常用算法

2. 全局置换算法

   置换页面的选择范围是所有可换出的物理页面

   工作计算法、缺页率算法

6.2 最优算法、先进先出算法和最近最久未使用算法

6.2.1 最优页面置换算法 (OPT,optimal)

基本思路：置换在未来最长时间不访问的页面
算法实现：

- 缺页时，计算内存中每个逻辑页面的下一次访问时间
- 选择**未来最长时间不访问的页面**

算法特征：

- 缺页最少，是理想情况
- 实际系统中**无法实现**
- 无法预知每个页面在下次访问前的等待时间
- 作为置换算法的性能评价依据

  在模拟器上运行某个程序，并记录每一次的页面访问情况

  第二遍运行时使用最优算法

最优页面置换算法示例

6.2.2 先进先出算法 (First-In First-Out,FIFO)

思路：选择**在内存驻留时间最长**的页面进行置换。

实现

- 维护一个记录所有位于内存中的逻辑页面链表
- 链表元素**按驻留内存的时间排序**，链首最长，链尾最短
- 出现缺页时，选择链首页面进行置换，新页面加到链尾

特征

- 实现简单
- 性能较差，调出的页面可能是经常访问的
- 进程分配物理页面数增加时，缺页并不一定减少 (Belady 现象)
- 很少单独使用

6.2.3 最近最久未使用算法 (Least Recently Used,LRU)

思路：

- 选择**最长时间没有被引用的**页面进行置换
- 如某些页面长时间未被访问，则它们在将来还可能会长时间不会访问

实现：

- 缺页时，计算内存中每个逻辑页面的**上一次**访问时间；
- 选择**上一次使用到当前时间最长的页面**

特征

- 最优置换算法的一种近似

6.2.4 LRU 算法的可能实现方法

1. 页面链表

   系统维护一个按最近一次访问时间排序的页面链表

   链表首节点是最近刚刚使用过的页面

   链表尾结点是最久未使用的页面

   访问内存时，找到相应页面，并把它移到链表之首

   缺页时，置换链表尾结点的页面

2. 活动页面栈

访问页面时，将此页号压入栈顶，并栈内相同的页号抽出

缺页时，置换栈底的页面

特征：实现开销大。

6.2.5 问题

1. 物理页帧数量为 3，虚拟页访问序列为 0,1,2,0,1,3,0,3,1,0,3，请问采用最优置换算法的缺页次数为 ()

A 1

B 2

C 3

D 4

2. 物理页帧数量为 3，虚拟页访问序列为 0,1,2,0,1,3,0,3,1,0,3，请问采用 LRU 置换算法的缺页次数为（D）

A 1

B 2

C 3

D 4

3. 物理页帧数量为 3，虚拟页访问序列为 0,1,2,0,1,3,0,3,1,0,3，请问采用 FIFO 置换算法的缺页次数为（D）

A 1

B 2

C 3

D 6

6.3 时钟置换算法和最不常用算法

6.3.1 时钟置换算法 (Clock)

思路：仅对页面的访问情况进行大致统计，又称最近未用 (Not Recently Used, NRU) 算法。

数据结构：

1. 在页表项中增加**访问位**，描述页面在过去一段时间的内访问情况
2. 各页面组织成**环形链表**
3. **指针**指向最先调入的页面

算法

1. 访问页面时，在页表项记录页面访问情况
2. 缺页时，从指针处开始顺序查找未被访问的页面进行置换

特征：时钟算法是 LRU 和 FIFo 的折中。
时钟置换算法的实现：

- 页面装入内存时，访问位初始化为 0；
- 访问页面（读/写）时，访问位置 1
- 缺页时，从指针当前位置顺序检查环形链表

  访问位为 0，则置换该页

  访问位为 1，则访问位置 0，并指针移动到下一个页面，直到找到可置换的页面

6.3.2 改进的 Clock 算法

思路：减少修改页的缺页处理开销。
算法：

1. 在页面中增加修改位，并在访问时进行相应修改
2. 缺页时，修改页面标志位，以跳过有修改的页面

指针扫过前		指针扫过后	
使用位	修改位	使用位	修改位
0	0	置换	置换
0	1	0	0
1	0	0	0
1	1	0	1

6.3.3 最不常用算法 (Least Frequently Used,LFU)

思路：缺页时，置换访问次数最少的页面

实现

1. 每个页面设置为一个访问计数
2. 访问页面时，访问计数加 1
3. 缺页时，置换计数最小的页面

特征

1. 算法开销大
2. 开始时频繁使用，但以后不使用的页面很难置换

   解决方法：计数定期右移

LRU 和 LFU 的区别：

- LRU 关注**多久未访问**，时间越短越好
- LFU 关注**访问次数**，次数越多越好

6.3.4 问题

1. 物理页帧数量为 4，虚拟页访问序列为 0,3,2,0,1,3,4,3,1,0,3,2,1,3,4 ，请问采用 CLOCK 置换算法（用 1 个 bit 表示存在时间）的缺页次数为 ()

   (a) 8

   (b) 9

   (c) 10

   (d) 11

2. 物理页帧数量为 4，虚拟页访问序列为 0,3,2,0,1,3,4,3,1,0,3,2,1,3,4 ，请问采用 CLOCK 置换算法（用 2 个 bit 表示存在时间）的缺页次数为 ()

   (a) 8

   (b) 9

   (c) 10

   (d) 11

6.4 BELADY 现象和局部置换算法比较

6.4.1 Belady 现象

现象：采用 FIFO 等算法时，可能出现分配的物理页面增加，缺页反而升高的异常现象。
原因：

- FIFO 算法的置换特征与进程访问内存的动态特征矛盾
- 被它置换出去的页面并不一定是进程近期不会访问的

思考：哪些置换算法没有 Belady 现象？

- FIFO 算法有 Belady 现象
- LRU 算法没有 Belady 现象

6.4.2 LRU、FIFO 和 Clock 的比较

1. LRU 算法和 FIFO 本质上都是先进先出的思路

   LRU 依据页面的最近访问时间排序

   LRU 需要动态调整顺序

   FIFO 依据页面进入内存的事件排序

   FIFO 的页面进入时间使固定不变的

2. LRU 可退化成 FIFO

   如页面进入内存后没有被访问，最近访问时间与内存的时间相同

3. LRU 算法性能较好，但系统开销较大

4. FIFO 算法系统开销较小，会发生 Belady 现象

5. Clock 算法是它们的折中

   页面访问时，不动态调整页面在链表中的顺序，仅做标记

   缺页时，再把它移动到链表末尾

6. 对于未访问的页面，Clock 和 LRU 算法的表现一样好

7. 对于被访问过的页面，Clock 算法不能记录准确访问顺序，而 LRU 算法可以

6.4.3 问题

1. 虚拟页访问序列为 1,2,3,4,1,2,5,1,2,3,4,5，物理页帧数量为 3 和 4，采用 FIFO 置换算法，请问是否会出现 bealdy 现象 (A)

   A 会

   B 不会

2. 下面哪些页面淘汰算法会产生 Belady 异常现象 (AB)

   A 先进先出页面置换算法（FIFO)

   B 时钟页面置换算法（CLOCK)

   C 最佳页面置换算法（OPT）

   D 最近最少使用页面置换算法（LRU）

6.5 工作集置换算法

局部置换算法没有考虑进程访存差异：

- FIFO 页面置换算法：假设初始顺序 a→b→c

6.5.1 全局置换算法

思路：全局置换算法为进程分配**可变数目**的物理页面
全局置换算法要解决的问题：

1. 进程在不同阶段的内存需求是变化的

2. 分配给进程的内存也需要在不同阶段有所变化

3. 全局置换算法需要确定分配给进程的物理页面数

CPU 利用率和并发进程数的关系：

- CPU 利用率与并发进程数存在相互促进和制约的关系

  进程数少时，提高并发进程数，可提高 CPU 利用率

  并发进程导致内存访问增加

  并发进程的内存访问会降低访存的局部性特征

6.5.2 工作集

工作集：一个进程当前正在使用的逻辑页面集合，可表示为二元函数 $W(t, \Delta)$

- t 是当前的执行时刻
- Δ 称为工作集窗口 (working-set window)，即一个定长的的页面访问时间窗口
- $W(t, \Delta)$ 是指在当前时刻 t 前的 Δ 时间窗口中的所有访问页面所组成的集合
- $|W(t, \Delta)|$ 指工作集的大小，即页面数目

工作集的变化

- 进程开始执行后，随着访问新页面逐步建立稳定的工作集
- 当内存访问的局部性区域的位置**大致稳定**时，工作集大小也大致稳定
- 局部性区域的位置改变时，工作集**快速扩张和收缩**过渡到下一个稳定值

6.5.3 常驻集

常驻集：在当前时刻，进程实际驻留在内存中的页面集合。
工作集与常驻集的关系：

- 工作集是进程在运行过程中固有的性质
- 常驻集取决于系统分配给进程的物理页面数目和页面置换算法

缺页率与常驻集的关系

- 常驻集 $\supseteq$ 工作集，缺页减少
- 工作集发生剧烈变动（过渡）时，缺页较少
- 进程常驻集大小达到一定数目后，缺页率也不会明显下降

6.5.4 工作集置换算法

工作集置换算法思路：换出不在工作集中的页面
窗口大小 τ

- 当前时刻前 τ 个内存访问的页引用是工作集，τ 被称为窗口大小

实现方法

- 访存链表：维护窗口内的访存页面链表
- 访存时，换出不在工作集的页面；更新链表
- 缺页时，换入页面；更新访存链接

6.6 缺页率置换算法

6.6.1 缺页率 (page fault rate)

缺页率：缺页平均时间间隔的倒数 $= \dfrac{\text{缺页次数}}{\text{内存访问次数}}$

影响缺页率的因素

- 页面置换算法
- 分配给进程的物理页面数目
- 页面大小
- 程序的编写方法

6.6.2 缺页率置换算法 (PFF,Page-Fault-Frequency)

通过调节常驻集大小，使每个进程的缺页率保持在一个合理的范围内。

- 若进程缺页率过高，则增加常驻集以分配更多的物理页面
- 若进程缺页率过低，则减少常驻集以减少它的物理页面

缺页率置换算法的实现

- 访存时，设置引用位标志
- 缺页时，计算从上次缺页时间 t_{last} 到现在 $t_{current}$ 的时间间隔
  - 如果 $t_{current} - t_{last} > T$，则置换所有在 $[t_{last}, t_{current}]$ 时间内没有被引用的页
  - $t_{current} - t_{last} \leq T$，则增加缺失页到常驻集中

6.6.3 问题

1. 物理页帧数量为 5，虚拟页访问序列为 4,3,0,2,2,3,1,2,4,2,2,4,0,3，请问采用缺页率置换算法（窗口 T=2）的缺页次数为 ()

   A 2

   B 3

   C 4

   D 5

6.7 抖动和负载控制

6.7.1 抖动问题 thrashing

抖动

- 进程物理页面太少，不能包含工作集
- 造成大量缺页，频繁置换
- 进程运行速度变慢

产生抖动的原因：随着驻留内存的进程数目增加，分配给每个进程的物理页面数不断减小，缺页率不断上升
操作系统需在并发水平和缺页率之间达到一个平衡。

- 选择一个适当的进程数目和进程需要的物理页面数

6.7.2 负载控制

- 通过调节并发进程数（MPL）来进行系统负载控制

  $\sum WSi =$ 内存大小