Homework 3

Operating System Principles and Implementation

PB22051080 王珏

2024 年 6 月 9 日

目录

1 术语解释 **1**

1.1 Segmentation fault：段错误 1

1.2 TLB (Translation Lookaside Buffer)：翻译后备缓冲区 1

1.3 Page fault：页错误 1

1.4 Demand paging：请求调页 1

2 Thrashing（抖动) **2**

2.1 抖动的概念 2

2.2 解释在什么情况下会发生抖动 2

3 分页系统 **3**

4 请求分页内存 **3**

5 页面替换算法 **5**

6 Belady 现象以及栈算法 **8**

7 磁盘调度算法 **10**

8 RAID 相关 **10**

9 打开文件表（open-file table） **12**

10 文件权限 **13**

11 文件系统布局的关键设计 **15**

12 带有索引分配的文件系统 **16**

13 硬链接和符号链接 **17**

14 日志记录方法 18

1 术语解释

Explain the following terms:

1.1 Segmentation fault：段错误

（暂未找到课本）是一种程序错误，通常发生在程序试图访问未被许可的内存区域时。内存是分段管理的，段错误即意味着程序访问了不属于它的内存段。段错误通常会导致程序崩溃或异常终止。

1.2 TLB (Translation Lookaside Buffer)：翻译后备缓冲区

P248 TLB 概念整理

TLB 是一种专用的、小的、查找快速的高速硬件缓冲，用于解决因访问一个字节需要两次内存访问导致访问速度减半的问题。

它的条目由标签和值两部分组成，当关联内存根据给定值查找时，它就会同时与所有的标签进行比较，如果找到条目，就得到相应值的字段。

1.3 Page fault：页错误

当进程执行和访问那些内存驻留的页面时，执行会正常进行。

但是，如果进程试图访问那些尚未调入内存中的页面时，对标记为无效的页面访问会产生缺页错误（Page fault）。分页硬件在通过页表转换地址时会注意到无效位被设置，从而陷入操作系统。这种陷阱是由于操作系统未能将所需页面掉入内存引起的。

1.4 Demand paging：请求调页

请求调页是一种实现从磁盘加载可执行程序到内存的策略。它不同于“加载整个程序导致所有选项的执行代码都加载到内存中，而不管这些选项最终是否被

使用”，它仅在需要时才加载页面，常常用于虚拟内存系统。

对于请求调页的虚拟内存，页面只有在程序执行期间被请求时才被加载。因此，从未访问的那些页从不加载到物理内存中。

2 Thrashing（抖动）

2.1 抖动的概念

课本 P283

如果进程没有需要支持活动使用页面的帧数，那么它会很快产生缺页错误。此时，必须置换某个页面。然而，由于它的所有页面都在使用中，所以必须立即置换需要再次使用的页面。因此，它会再次快速产生缺页错误，再一次置换必须立即返回的页面，如此快速进行。

这种高度的页面调度活动称为抖动 (thrashing)。如果一个进程的调页时间多于它的执行时间，那么这个进程就在抖动。

2.2 解释在什么情况下会发生抖动

P283 最下-284 上部

操作系统监视 CPU 利用率。如果 CPU 利用率太低，那么通过向系统引入新的进程来增加多道程度。采用全局置换算法会置换任何页面，而不管这些页面属于哪个进程。现在假设进程在执行中进入一个新阶段，并且需要更多的帧。它开始出现缺页错误，并从其他进程那里获取帧。然而，这些进程也需要这些页面，因此它们也会出现缺页错误，并且从其他进程中获取帧。这些缺页错误进程必须使用调页设备以将页面换进和换出。当它们为调页设备排队时，就绪队列清空。随着进程等待调页设备，CPU 利用率会降低。

CPU 调度程序看到 CPU 利用率的降低，进而会增加多道程度。新进程试图从其他运行进程中获取帧来启动，从而导致更多的缺页错误和更长的调页设备

队列。因此，CPU 利用率进一步下降，并且 CPU 调度程序试图再次增加多道程度。这样就出现了抖动，系统吞吐量陡降。缺页错误率显著增加。结果，有效内存访问时间增加。没有工作可以完成，因为进程总在忙于调页。

3 分页系统

考虑一个页表存储在内存中的分页系统。

a. 如果一次内存引用需要 50 纳秒，那么一次分页内存引用需要多长时间？

在未添加 TLB 的情况下，访问一个字节需要两次内存访问，因此一次分页内存引用需要 $50ns \times 2 = 100ns$

b. 如果我们添加 TLB，并且 75% 的页表引用都能在 TLB 中找到，那么有效的内存引用时间是多少？（假设在 TLB 中找到一个页表项需要 2 纳秒，如果该条目存在。）

$Time = 0.75 \times (50 + 2) + 0.25 \times (100 + 2) = 64.5ns$

4 请求分页内存

假设我们有一个请求分页内存。页表保存在寄存器中。如果有空页或被替换的页没有被修改，那么处理一次页错误需要 8 毫秒；如果被替换的页被修改，则需要 20 毫秒。内存访问时间是 100 纳秒。假设被替换的页有 70% 的时间是被修改的。

为了使有效访问时间不超过 200 纳秒，最大可接受的页错误率是多少？

- **给定条件**：
  - 内存访问时间：100 纳秒
  - 处理一次页错误的时间（页未修改）：8 毫秒 = 8,000,000 纳秒
  - 处理一次页错误的时间（页已修改）：20 毫秒 = 20,000,000 纳秒

- 被替换的页被修改的概率：70
- 被替换的页未被修改的概率：30
- 目标有效访问时间（EAT）：200 纳秒

- **计算平均页错误处理时间**：
  - 平均页错误处理时间 $= 0.3 \times 8,000,000 + 0.7 \times 20,000,000$
    $= 2,400,000 + 14,000,000 = 16,400,000$纳秒

- **使用 EAT 公式求解页错误率**：
  -
    $$\text{EAT} = (1-p) \times \text{内存访问时间} + p \times \text{平均页错误处理时间}$$
  - 代入已知条件：
  -
    $$200 \geq (1-p) \times 100 + p \times 16,400,000$$
  - 求解：
  -
    $$200 \geq 100 - 100p + 16,400,000p$$
  -
    $$200 \geq 100 + 16,399,900p$$
  -
    $$100 \geq 16,399,900p$$
  -
    $$p \leq \frac{100}{16,399,900}$$
  -
    $$p \leq 6.1 \times 10^{-6}$$

- **结论**：为了使有效访问时间不超过 200 纳秒，最大可接受的页错误率约为 6.1×10^{-6} 或者 0.00061

5 页面替换算法

考虑以下页面引用字符串：7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0, 1。**假设使用请求分页并且有三个页面框架，对于以下页面替换算法，会发生多少次页面错误？**

- **LRU 替换**（最近最少使用）
  - 引用 7，页面错误，内存框架状态：[7]
  - 引用 2，页面错误，内存框架状态：[7, 2]
  - 引用 3，页面错误，内存框架状态：[7, 2, 3]
  - 引用 1，页面错误，替换最近最少使用的 7，内存框架状态：[1, 2, 3]
  - 引用 2，不发生页面错误，内存框架状态：[1, 2, 3]
  - 引用 5，页面错误，替换最近最少使用的 3，内存框架状态：[1, 2, 5]
  - 引用 3，页面错误，替换最近最少使用的 1，内存框架状态：[3, 2, 5]
  - 引用 4，页面错误，替换最近最少使用的 2，内存框架状态：[3, 4, 5]
  - 引用 6，页面错误，替换最近最少使用的 5，内存框架状态：[3, 4, 6]
  - 引用 7，页面错误，替换最近最少使用的 3，内存框架状态：[7, 4, 6]
  - 引用 7，不发生页面错误，内存框架状态：[7, 4, 6]
  - 引用 1，页面错误，替换最近最少使用的 4，内存框架状态：[7, 1, 6]
  - 引用 0，页面错误，替换最近最少使用的 6，内存框架状态：[7, 1, 0]
  - 引用 5，页面错误，替换最近最少使用的 7，内存框架状态：[5, 1, 0]

- 引用 4，页面错误，替换最近最少使用的 1，内存框架状态：[5, 4, 0]
- 引用 6，页面错误，替换最近最少使用的 0，内存框架状态：[5, 4, 6]
- 引用 2，页面错误，替换最近最少使用的 5，内存框架状态：[2, 4, 6]
- 引用 3，页面错误，替换最近最少使用的 4，内存框架状态：[2, 3, 6]
- 引用 0，页面错误，替换最近最少使用的 6，内存框架状态：[2, 3, 0]
- 引用 1，页面错误，替换最近最少使用的 2，内存框架状态：[1, 3, 0]
- **总页面错误次数**：18

- **FIFO 替换**（先进先出）
  - 引用 7，页面错误，内存框架状态：[7]
  - 引用 2，页面错误，内存框架状态：[7, 2]
  - 引用 3，页面错误，内存框架状态：[7, 2, 3]
  - 引用 1，页面错误，替换 7，内存框架状态：[1, 2, 3]
  - 引用 2，不发生页面错误，内存框架状态：[1, 2, 3]
  - 引用 5，页面错误，替换 2，内存框架状态：[1, 5, 3]
  - 引用 3，不发生页面错误，内存框架状态：[1, 5, 3]
  - 引用 4，页面错误，替换 3，内存框架状态：[1, 5, 4]
  - 引用 6，页面错误，替换 1，内存框架状态：[6, 5, 4]
  - 引用 7，页面错误，替换 5，内存框架状态：[6, 7, 4]
  - 引用 7，不发生页面错误，内存框架状态：[6, 7, 4]
  - 引用 1，页面错误，替换 4，内存框架状态：[6, 7, 1]
  - 引用 0，页面错误，替换 6，内存框架状态：[0, 7, 1]
  - 引用 5，页面错误，替换 7，内存框架状态：[0, 5, 1]

- 引用 4，页面错误，替换 1，内存框架状态：[0, 5, 4]
- 引用 6，页面错误，替换 0，内存框架状态：[6, 5, 4]
- 引用 2，页面错误，替换 5，内存框架状态：[6, 2, 4]
- 引用 3，页面错误，替换 4，内存框架状态：[6, 2, 3]
- 引用 0，页面错误，替换 6，内存框架状态：[0, 2, 3]
- 引用 1，页面错误，替换 2，内存框架状态：[0, 1, 3]
- **总页面错误次数**：17

- **最佳替换**
  - 引用 7，页面错误，内存框架状态：[7]
  - 引用 2，页面错误，内存框架状态：[7, 2]
  - 引用 3，页面错误，内存框架状态：[7, 2, 3]
  - 引用 1，页面错误，替换 7，内存框架状态：[1, 2, 3]
  - 引用 2，不发生页面错误，内存框架状态：[1, 2, 3]
  - 引用 5，页面错误，替换 1，内存框架状态：[1, 5, 3]
  - 引用 3，不发生页面错误，内存框架状态：[1, 5, 3]
  - 引用 4，页面错误，替换 3，内存框架状态：[1, 5, 4]
  - 引用 6，页面错误，替换 4，内存框架状态：[1, 5, 6]
  - 引用 7，页面错误，替换 6，内存框架状态：[1, 5, 7]
  - 引用 7，不发生页面错误，内存框架状态：[1, 5, 7]
  - 引用 1，不发生页面错误，内存框架状态：[1, 5, 7]
  - 引用 0，页面错误，替换 7，内存框架状态：[1, 5, 0]
  - 引用 5，不发生页面错误，内存框架状态：[1, 5, 0]

  - 引用 4，页面错误，替换 5，内存框架状态：[1, 4, 0]
  - 引用 6，页面错误，替换 4，内存框架状态：[1, 6, 0]
  - 引用 2，页面错误，替换 6，内存框架状态：[1, 2, 0]
  - 引用 3，页面错误，替换 2，内存框架状态：[1, 3, 0]
  - 引用 0，不发生页面错误，内存框架状态：[1, 3, 0]
  - 引用 1，不发生页面错误，内存框架状态：[1, 3, 0]
  - **总页面错误次数**：13

6 Belady 现象以及栈算法

解释什么是 **Belady 现象**，以及**堆栈算法的特征**是什么，使其从不表现出 Belady 现象。

- **定义**：Belady 现象指的是在某些情况下，增加页面置换算法的可用物理内存页数（分配帧数量的增加），反而会导致页面错误率增加。（我们原本的期望，为一个进程提供更多的内存可以改善其性能）。
- **发生条件**：
  - 使用先进先出（FIFO）页面置换算法。
  - 物理内存页数从较少增加到较多。
  - 特定顺序的页面访问序列。
- **示例（课本 P275 例子）**：
  - 页面访问序列：$1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$
  - 使用 3 个物理页框：
    * 初始状态：空

    * 访问 1：页面错误，装载页面 1。
    * 访问 2：页面错误，装载页面 2。
    * 访问 3：页面错误，装载页面 3。
    * 访问 4：页面错误，替换页面 1（因为是 FIFO）。
    * 依次类推...
    * 最终，页面错误数为 9 次。
  - 使用 4 个物理页框：
    * 初始状态：空
    * 访问 1：页面错误，装载页面 1。
    * 访问 2：页面错误，装载页面 2。
    * 访问 3：页面错误，装载页面 3。
    * 访问 4：页面错误，装载页面 4。
    * 依次类推...
    * 最终，页面错误数为 10 次。

- **原因**：Belady 现象的出现是由于 FIFO 算法的特性决定的。当页框增加时，FIFO 算法并不会考虑页面访问的实际频率和时间，而是简单地按照先入先出的顺序替换页面，导致可能还需要的页面被提前替换出去，从而引发更多的页面错误。

- **解决方法**：某些页面置换算法如 LRU（Least Recently Used，最近最少使用）和 OPT（Optimal，最优页面置换）等不会出现 Belady 现象。这些算法会考虑页面的实际使用情况，从而在增加物理页框时，更好地利用内存资源，减少页面错误。

7 磁盘调度算法

假设一个磁盘驱动器有 6000 个磁柱，编号从 0 到 5999。驱动器当前正在服务于位于 2150 号磁柱的请求，上一个请求位于 1805 号磁柱。待处理请求的队列按 FIFO 顺序为：2069, 1212, 2296, 2800, 544, 1618, 356, 1523, 4965, 3681。从当前磁头位置开始，计算磁盘臂移动以满足所有待处理请求的总距离（以磁柱为单位），对于以下每种磁盘调度算法：

- a. FCFS（先来先服务）13011
- b. SSTF（最短寻道时间优先）7586
- c. SCAN 8492
- d. LOOK 9137
- e. C-SCAN 11466
- f. C-LOOK 7424

8 RAID 相关

说明 RAID 4 和 RAID 5 的关键区别，并分析它们的优缺点。

- **RAID 4**
  - **关键区别**：
    - 使用独立的奇偶校验盘：RAID 4 将（其他 N 个磁盘的块的）奇偶校验数据存储在一个单独的磁盘上。
    - 数据块级条带化：数据按块级别条带化分布在各数据盘上。
  - **优点**：

    - 读取性能好：由于数据分布在多个磁盘上，可以并行读取，读取性能高。
    - 高效的奇偶校验：奇偶校验集中在一个磁盘上，计算简单。
  - **缺点**：
    - 写性能瓶颈：每次写操作都涉及奇偶校验盘，可能导致奇偶校验盘成为瓶颈。
    - 奇偶校验盘单点故障：奇偶校验盘的故障会导致整个 RAID 组的数据保护失效。

- **RAID 5**
  - **关键区别**：
    - 分布式奇偶校验：RAID 5 将奇偶校验数据分布存储在所有磁盘上，而不是专用奇偶校验盘。
    - 数据块级条带化：数据按块级别条带化分布在各数据盘上。
  - **优点**：
    - 读取性能好：与 RAID 4 类似，RAID 5 的数据读取也可以并行进行，读取性能高。
    - 提高写性能：由于奇偶校验分布在所有磁盘上，避免了单个奇偶校验盘成为瓶颈。
    - 容错能力：任何一个磁盘故障都不会导致数据丢失，能够重建数据。
  - **缺点**：
    - 写入性能较慢：写操作需要读写多个磁盘以更新奇偶校验数据，增加了写操作的复杂性。

    * 重建时间长：磁盘故障后重建数据需要从剩余磁盘计算奇偶校验，重建时间较长。（对于大磁盘集的 RAID 5 重建，可能需要几个小时——重建性能低）

9 打开文件表（open-file table）

解释什么是打开文件表以及为什么我们需要它。

课本 P311-313 文件操作

- **打开文件表的定义**：
  - 操作系统的六个基本操作：创建文件，写文件，读文件，重新定位文件，删除文件，截断文件。大多数文件操作涉及搜索目录，以得到命名文件的相关条目。
  - 为避免这种不断的搜索，设计打开文件表。打开文件表是操作系统维护的一个数据结构，用于跟踪和维护当前被打开的文件的信息。
  - 每当一个文件被打开，操作系统会在打开文件表中创建一个条目，记录该文件的相关信息。
- **打开文件表的内容**：
  - 文件描述符：标识文件的唯一整数值。
  - 文件指针：指向文件当前读写的位置。
  - 文件打开计数：跟踪打开和关闭的次数。
  - 文件状态：文件的访问模式（读、写或读写）。
  - 文件相关的控制信息：包括文件的权限、锁定状态等。
  - 指向文件 inode 的指针：用于访问文件的元数据。
- **为什么需要打开文件表**：

- **高效管理文件访问**：
  - 打开文件表使得操作系统能够高效地管理和跟踪多个进程对文件的访问。
  - 提供文件描述符，使得进程能够方便地引用和操作打开的文件。
- **维持文件状态的一致性**：
  - 打开文件表保存了文件的访问模式和文件指针位置，确保文件读写操作的正确性。
  - 避免了多个进程并发访问文件时可能引发的不一致问题。
- **简化资源管理**：
  - 通过打开文件表，操作系统可以在文件关闭时释放相关资源，如文件描述符和内存。
  - 有助于实现文件的引用计数，确保文件在所有进程关闭之前不会被删除或修改。
- **支持文件共享和安全性**：
  - 打开文件表允许多个进程共享文件，通过文件锁定机制确保数据一致性和安全性。
  - 提供文件权限信息，确保文件访问符合安全策略。

10 文件权限

文件权限为“755”具体代表什么？

文件权限“755”可以解析为：

- 第一位“7”：文件所有者的权限（rwx）
  - 读（r）= 4

  - 写（w）= 2
  - 执行（x）= 1
  - 总和 = 7（rwx）

- 第二位“5”：文件所有者所在组的用户的权限（r-x）
  - 读（r）= 4
  - 无写（-）= 0
  - 执行（x）= 1
  - 总和 = 5（r-x）

- 第三位“5”：其他用户的权限（r-x）
  - 读（r）= 4
  - 无写（-）= 0
  - 执行（x）= 1
  - 总和 = 5（r-x）

文件权限“755”表示：

- 文件所有者拥有读、写和执行权限（rwx）。
- 文件所有者所在组的用户拥有读和执行权限（r-x）。
- 其他用户拥有读和执行权限（r-x）。

这种设置常用于可执行文件，所有者可以读、写、执行该文件，而其他用户只能读和执行。

11 文件系统布局的关键设计

课本 P344 11.4 分配方法

说明文件系统布局中连续分配、链表分配和索引节点分配的关键设计。

- **连续分配（Contiguous Allocation）**
  - **关键设计特点**：
    - 文件块连续存放。
  - **优点**：
    - 读取速度快，磁头移动次数少。
    - 顺序访问效率高。
  - **缺点**：
    - 空间利用率低，容易产生外部碎片。
    - 文件增长受限，可能需要重新分配连续空间。
- **链表分配（Linked list Allocation）**
  - **关键设计特点**：
    - 链表结构，每个数据块包含指向下一个数据块的指针。
  - **优点**：
    - 解决了连续分配的碎片问题，空间利用率高，可以利用零散的空闲块。
    - 易于文件扩展，不需要重新分配。
  - **缺点**：
    - 随机访问效率低，必须顺序遍历链表。
    - 每个数据块需要存储额外的指针，增加了空间开销。

- **索引节点分配（Index node Allocation）**
  - **关键设计特点**：
    - 索引节点（inode）存储文件的元数据和指向数据块的指针（将所有指针放在一起）。
    - 每个文件都有自己的索引块，这是一个磁盘块地址的数组。索引块的第 i 个条目指向文件的第 i 个块。目录包含索引块的地址。
  - **优点**：
    - 支持大文件，扩展性好。
    - 随机访问效率高，能够快速定位数据块。
  - **缺点**：
    - 实现复杂度高，需要维护索引节点结构。
    - 索引节点本身占用额外空间。

12 带有索引分配的文件系统

考虑一个类似于 UNIX 使用的带有索引分配的文件系统，并假设每个文件仅使用一个块。在以下两种情况下，读取位于 /a/b/c 的小型本地文件的内容可能需要多少次磁盘 I/O 操作？请提供详细的工作流程。

- **a. 没有任何磁盘块和索引节点当前被缓存**
  - 读取根目录的索引节点（1 次 I/O）
  - 读取根目录的内容，找到 /a 的索引节点（1 次 I/O）
  - 读取 /a 的索引节点（1 次 I/O）
  - 读取 /a 目录的内容，找到 /b 的索引节点（1 次 I/O）
  - 读取 /b 的索引节点（1 次 I/O）

- 读取 /b 目录的内容，找到 /c 的索引节点（1 次 I/O）
- 读取 /c 的索引节点（1 次 I/O）
- 读取 /c 文件的数据块（1 次 I/O）

总共需要 8 次磁盘 I/O 操作。

- **b. 没有任何磁盘块当前被缓存，但所有索引节点都在内存中**
  - 读取根目录的内容，找到 /a 的索引节点（1 次 I/O）
  - 读取 /a 目录的内容，找到 /b 的索引节点（1 次 I/O）
  - 读取 /b 目录的内容，找到 /c 的索引节点（1 次 I/O）
  - 读取 /c 文件的数据块（1 次 I/O）

总共需要 4 次磁盘 I/O 操作。

13 硬链接和符号链接

硬链接和符号链接有什么区别？

- **引用方式**：
  - **硬链接**：直接引用文件数据块，多个硬链接共享相同的 inode。
  - **符号链接**：引用目标文件或目录的路径，符号链接有独立的 inode。
- **链接类型**：
  - **硬链接**：只能指向文件，不能指向目录。
  - **符号链接**：可以指向文件或目录。
- **跨文件系统**：

  - **硬链接**：不能跨文件系统，只能在同一文件系统内创建。
  - **符号链接**：可以跨文件系统创建。

- **文件删除**：
  - **硬链接**：删除一个硬链接不会影响其他硬链接，只有当所有硬链接被删除后，文件数据才会被释放。
  - **符号链接**：删除符号链接不影响目标文件，但如果目标文件被删除，符号链接会变成无效链接（悬空链接）。

- **存储方式**：
  - **硬链接**：共享相同的物理数据块，数据在多个硬链接中一致。
  - **符号链接**：存储路径信息，指向目标文件或目录。

14 日志记录方法

数据日志记录和元数据日志记录有什么区别？解释每种日志记录方法的操作顺序。

- **数据日志记录（Data Journaling）**：
  - **定义**：数据日志记录是一种日志记录方法，其中数据和元数据在写入到其最终位置之前，先写入日志。
  - **操作顺序**：
    1. 将数据写入日志。
    2. 将元数据写入日志。
    3. 将数据从日志写入最终位置。
    4. 将元数据从日志写入最终位置。

5. 更新日志状态以表示操作完成。

- **元数据日志记录（Metadata Journaling）**：
  - **定义**：元数据日志记录是一种日志记录方法，其中只有元数据在写入到其最终位置之前，先写入日志，而数据直接写入其最终位置。
  - **操作顺序**：
    1. 将元数据写入日志。
    2. 将数据直接写入最终位置。
    3. 将元数据从日志写入最终位置。
    4. 更新日志状态以表示操作完成。
- **区别**：
  - **数据日志记录**：记录数据和元数据的所有变化，提供更高的完整性和一致性，但会有较高的写入开销。
  - **元数据日志记录**：只记录元数据的变化，写入开销较低，但在系统崩溃时，数据可能处于不一致状态。