作业1-Cache工作原理与应用

计科210X 甘晴void （学号：202108010XXX）

目录

一、**Cache**工作原理

1.Cache简介

一般而言，高速缓存 (cache, 读作 "cash") 是一个小而快速的存储设备，它作为存储在更大、也更慢的设备中的数据对象的缓冲区域。使用高速缓存的过程称为缓存 (caching, 读作"cashing")

存储器层次结构的中心思想是，对于每个 k, 位于k层的更快更小的存储设备作为位于 k+l层的更大更慢的存储设备的缓存。换句话说，层次结构中的每一层都缓存来自较低一层的数据对象。例如，本地磁盘作为通过网络从远程磁盘取出的文件（例如 Web 页面）的缓存，主存作为本地磁盘上数据的缓存，依此类推，直到最小的缓存——CPU 寄存器集合。

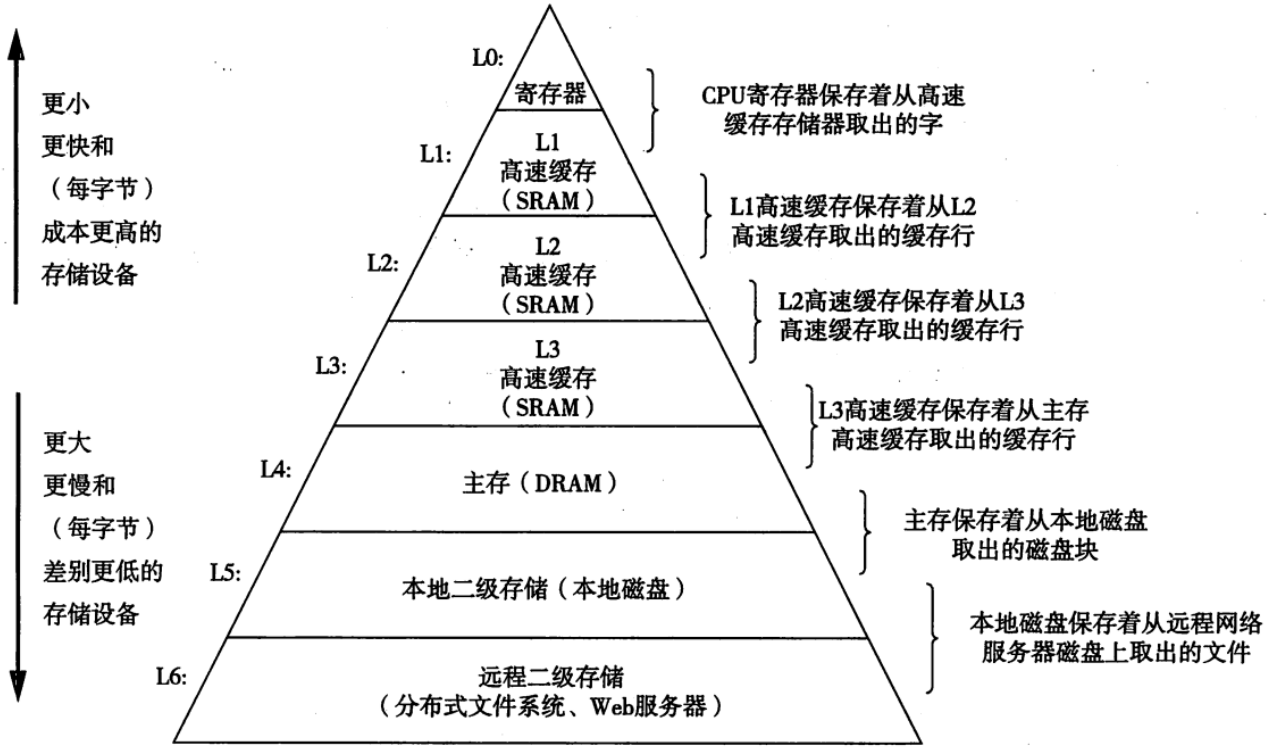

图 6-23　存储器层次结构

下面以某型号的计算机结构为例，较为一般地说明数据在Cache上的传输。

数据总是以块大小为传送单元 (transfer unit) 在第k层和第 k+l 层之间来回拷贝的。虽然在层次结构中任何一对相邻的层次之间块大小是固定的，但是其他的层次对之间可以有不同的块大小。例如，在上图中， L1和L0 之间的传送通常使用的是1个字的块。 L2和L1 之间（以及L3和L2 之间、 L4和L3 之间）的传送通常使用的是16 个字的块。而 L5和L4之间的传送用的是大小为几百或几千字节的块。

总结来说，一般层次结构中较低层（离 CPU 较远）的设备的访问时间较长，因此为了补偿这些较长的访问时间，倾向于使用较大的块。

2.Cache何时被使用

数据请求检查：在每次数据请求到达时，缓存首先检查请求的数据是否已经存在于缓存中。如果数据存在，并且未过期或无效，缓存将直接返回数据，而无需访问原始数据源。

3.Cache命中与未命中

如果请求的数据存在于缓存中，则称为“缓存命中”；如果数据不存在于缓存中，则称为“缓存未命中”。对于未命中的情况，缓存将请求发送到原始数据源，并在获取到数据后将其存储在缓存中以供后续使用。

如果Cache不命中，则需要把该数据取出来并覆盖现有数据，覆盖一个现存的块的过程称为替换 (replacing) 或驱逐 (evicting) 这个块。被驱逐的这个块有时也称为牺牲块 (victim block) 。决定该替换哪个块是由缓存的替换策略 (replacement policy) 来控制的。

4.Cache不命中种类

区分不同种类的缓存不命中有时有重要意义。

①冷不命中

如果第k层的缓存是空的，那么对任何数据对象的访问都会不命中。一个空的缓存有时称为冷缓存 (cold cache), 此类不命中称为强制性不命中 (compulsory miss) 或冷不命中 (cold miss) 。冷不命中通常是短暂的事件，不会在反复访问存储器使得缓存暖身 (warmed up) 之后的稳定状态中出现。

只要发生了不命中，第k层的缓存就必须执行某个放置策略 (placement policy), 确定把它从第 k:+-1 层中取出的块放在哪里。最灵活的替换策略是允许来自第炉 层的任何块放在第 层的任何块中。对于存储器层次结构中高层的缓存（靠近 CPU), 它们是用硬件来实现的，而且速度是最优的，这个策略实现起来通常很昂贵，因为随机地放置块，定位起来代价很高。

因此，硬件缓存通常使用的是更严格的放置策略，这个策略将第 k:+-1 层的某个块限制放置在层块的一个小的子集中（有时只是一个块）。例如，在图 6-24 中，我们可以确定第炉 层的必须放置在第 层的块 (i mod 4) 中。例如，第 k:+-1 层的块 12 会映射到第的块 0; l 、 13 会映射到块 ; 依此类推。注意，图 6-24 中的示例缓存使用的就是这个策略。

②冲突不命中

这种限制性的放置策略会引起一种不命中，称为冲突不命中 (conflict miss) , 在这种情况下，缓存足够大，能够保存被引用的数据对象，但是因为这些对象会映射到同一个缓存块，缓存会一直不命中。例如，在图 6-24 中，如果程序请求块 0, 然后块 8, 然后块 O, 然后块 8, 依此类推，在第 层的缓存中，对这两个块的每次引用都会不命中，即使是这个缓存总共可以容纳个块。

③容量不命中

程序通常是按照一系列阶段（如循环）来运行的，每个阶段访问缓存块的某个相对稳定不变的集合。例如，一个嵌套的循环可能会反复地访问同一个数组的元素。这个块的集合称为这个阶段的工作集 (working set) 。当工作集的大小超过缓存的大小时，缓存会经历容量不命中(capacity miss) 。换句话说，缓存就是太小了，不能处理这个工作集。

5.Cache替换策略

当缓存空间不足时，需要选择一些数据来替换以腾出空间。常见的替换策略包括最近最少使用（LRU）、最少使用（LFU）和随机替换等。

①**LRU**

- 访问记录维护：LRU策略需要维护一个访问记录，以跟踪每个数据项最近一次被访问的时间。这通常可以通过数据结构（例如链表、队列或哈希表）来实现。其中，链表或双向链表常用于记录数据项的访问顺序。
- 数据访问时的更新：每当某个数据项被访问时，LRU策略会更新该数据项在访问记录中的位置，通常是将其移动到记录中的最前端或最末端，表示该数据项是最近被使用的。
- 替换最久未被访问的数据：当缓存空间不足时，LRU策略将替换访问记录中最久未被访问的数据项，即位于访问记录的末尾的数据项。
- 实现优化：为了提高LRU策略的效率，可以使用一些数据结构优化，例如哈希表结合双向链表。哈希表用于快速查找数据项的位置，而双向链表用于记录数据项的访问顺序，并且在更新和替换时提供高效的操作。

②**LFU**

- 频率计数：LFU算法维护了一个频率计数器（frequency counter），用于记录每个页面被访问的频率。每次页面被访问，对应页面的计数器就会增加。这样就能够知道哪些页面是最近最少被访问的。
- 页面替换：当需要替换页面时，LFU算法选择访问频率最低的页面进行替换。即选择那些频率计数器值最小的页面作为替换对象。
- 处理频率相同的页面：当多个页面的访问频率相同时，LFU算法会根据额外的规则来选择替换哪个页面，比如选择最早插入的页面或者根据其他因素进行选择。
- 更新频率计数器：在页面被访问时，需要更新对应页面的频率计数器值。这可以通过增加页面的访问频率来实现。
- 实现细节：LFU算法的实现可能会涉及到数据结构的选择，比如使用哈希表（Hash Table）或者堆（Heap）来维护页面的访问频率计数器。

6.Cache失效与更新

缓存中的数据可能会因为过期、失效或被更新而需要重新获取。为了保持数据的实时性和准确性，缓存通常会实现一些失效机制，例如基于时间的失效（TTL）或者是在数据被更新时主动使缓存失效。

- 过期时间（**TTL**）： 一种常见的失效机制是给缓存中的每个数据项设置一个过期时间，也称为生存时间（Time To Live，TTL）。当某个数据项被存入缓存时，会同时记录下该数据项的过期时间。当读取数据时，系统会首先检查数据项是否已经过期，如果过期则认为数据失效，需要从源数据源重新获取最新的数据。这个过期时间可以由系统管理员或者开发人员根据业务需求设定。
- 基于访问模式的失效策略： 有些缓存系统采用基于访问模式的失效策略。这种策略基于数据的访问频率，对于长时间未被访问的数据项，系统会认为其可能已经过时，从而使得该数据项失效。这种策略可以根据实际情况来调整，比如可以设置一个阈值，当某个数据项的访问次数低于阈值时，就将其标记为过期数据。
- 缓存验证： 在一些情况下，缓存系统并不主动检查数据是否过期，而是等到客户端请求数据时再去验证数据是否有效。这种方式下，当客户端请求某个数据时，缓存系统会将数据返回给客户端，同时发送一个验证请求到数据源，验证数据是否仍然有效。如果数据有效，则继续使用缓存中的数据；如果数据失效，则从源数据源获取最新数据，并更新缓存。
- 主动失效策略： 有时候，缓存系统会定期地或者在特定条件下主动清理失效数据。这可能是基于一些策略，比如定时清理、内存压力情况等。定时清理可以周期性地扫描缓存中的数据，清理已经过期的数据项；而内存压力情况下，当缓存系统的内存使用达到一定阈值时，系统可以根据一定的策略来清理部分数据，以腾出内存空间。

7.Cache适用性

缓存并不适用于所有类型的数据。通常情况下，缓存更适合于对数据的读取操作频繁而写入操作相对较少的场景，因为缓存的更新与维护可能会增加写入操作的复杂度和延迟。

在以下的情况下缓存不适用

- 数据更新频繁：如果数据的更新频率非常高，而且缓存无法及时同步这些更新，就会导致缓存中的数据与源数据不一致，从而引发数据的不一致性问题。
- 数据量过大：当数据量过大时，缓存的内存容量可能无法满足存储所有数据的需求，这样就会导致缓存命中率下降，失去了使用缓存的意义。
- 数据访问模式不确定：如果数据的访问模式不确定，即数据的访问模式随时可能发生变化，那么很难通过缓存来提高性能，因为缓存需要根据数据的访问模式来进行优化。
- 对实时性要求高：有些数据需要实时更新，不能容忍任何延迟，这种情况下使用缓存可能会导致数据的过期或延迟，从而影响业务的正常运行。

- 安全性要求高：一些敏感数据或者安全性要求较高的数据，不适合缓存在客户端或者中间节点，因为缓存可能会面临数据泄露或被篡改的风险。

二、应用**Cache**原理做程序优化

1.总述

根据之前列出的Cache工作原理，可以从一下角度进行思考，使用Cache原理做程序优化。

- 缓存常用数据: 将程序中频繁使用的数据存储在缓存中，以减少对数据库或者其他外部资源的访问。这可以通过使用内存缓存、分布式缓存或者本地缓存来实现。
- 缓存计算结果: 如果某些计算结果是固定的或者是昂贵的，可以将这些计算结果缓存起来，避免重复计算。这种方式适用于一些计算密集型的任务，如数学计算、数据处理等。
- 缓存页面或资源: 对于 web 应用程序，可以缓存页面、静态资源（如 CSS、JavaScript、图片等）或者 API 响应结果，以减少网络延迟和服务器负载。这可以通过浏览器缓存、CDN（内容分发网络）或者服务端缓存来实现。
- 缓存预处理数据: 如果程序需要处理大量数据，可以预先计算并缓存部分数据，以减少运行时的计算量。这种方式可以在程序启动时或者后台任务中完成。
- 缓存连接和资源: 对于一些需要频繁创建和销毁的资源，如数据库连接、文件句柄等，可以使用连接池或者资源池进行缓存，以减少资源的创建和释放开销。
- 缓存热点数据: 分析程序运行时的数据访问模式，将热点数据缓存起来，以提高命中率和访问速度。这可以通过缓存淘汰策略和优化算法来实现。
- 缓存读写优化: 对于频繁读写的操作，可以采用批量读写、异步读写或者延迟加载等技术来减少对缓存的频繁访问，提高效率。
- 缓存失效策略: 设计合适的缓存失效策略，避免缓存数据过期或者占用过多内存。常见的失效策略包括基于时间的失效、基于访问频率的失效以及基于事件触发的失效等。

2.程序演示：处理矩阵时使用局部性原理提高Cache命中率

在学习《计算机系统》时有一个很著名的例子，我今天再接过来用一用。就是处理矩阵乘法的时候，对于循环变量层次的先后做出改变，会影响Cache命中率，因此用合适的方式可以提高Cache命中率，这是空间局部性的典型运用。以下是c++代码。

```
#include <iostream>
#include <vector>
#include <chrono>

```

```cpp
using namespace std;
using namespace std::chrono;

const int matrix_size = 1000;

// 初始化矩阵
vector<vector<double>> initializeMatrix() {
    vector<vector<double>> matrix(matrix_size, vector<double>(matrix_size));
    for (int i = 0; i < matrix_size; ++i) {
        for (int j = 0; j < matrix_size; ++j) {
            matrix[i][j] = static_cast<double>(rand()) / RAND_MAX;
        }
    }
    return matrix;
}

// 矩阵乘法函数，未优化版本
void naiveMatrixMultiply(const vector<vector<double>>& a, const vector<vector<double>>& b, vector<vector<double>>& result) {
    auto start_time = high_resolution_clock::now();
    for (int i = 0; i < matrix_size; ++i) {
        for (int j = 0; j < matrix_size; ++j) {
            for (int k = 0; k < matrix_size; ++k) {
                result[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    auto end_time = high_resolution_clock::now();
    auto duration = duration_cast<seconds>(end_time - start_time);
    cout << "Naive matrix multiplication took " << duration.count() << " seconds" << endl;
}

// 优化后的矩阵乘法函数，利用局部性原理和缓存
void optimizedMatrixMultiply(const vector<vector<double>>& a, const vector<vector<double>>& b, vector<vector<double>>& result) {
    auto start_time = high_resolution_clock::now();
    for (int i = 0; i < matrix_size; ++i) {
```

```
        for (int k = 0; k < matrix_size; ++k) {
            for (int j = 0; j < matrix_size; ++j) {
                result[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    auto end_time = high_resolution_clock::now();
    auto duration = duration_cast<seconds>(end_time - start_time);
    cout << "Optimized matrix multiplication took " << duration.count() << " seconds" << endl;
}

int main() {
    // 初始化矩阵
    auto matrix_a = initializeMatrix();
    auto matrix_b = initializeMatrix();
    vector<vector<double>> result(matrix_size, vector<double>(matrix_size));

    // 测试
    naiveMatrixMultiply(matrix_a, matrix_b, result);
    optimizedMatrixMultiply(matrix_a, matrix_b, result);

    return 0;
}

```

使用Python代码也可以实现示例。

```
import numpy as np
import time

# 使用 numpy 生成一个随机矩阵
matrix_size = 1000
matrix_a = np.random.rand(matrix_size, matrix_size)
matrix_b = np.random.rand(matrix_size, matrix_size)
result = np.zeros((matrix_size, matrix_size))

# 矩阵乘法函数，未优化版本
def naive_matrix_multiply(a, b):

```

```
    start_time = time.time()
    for i in range(matrix_size):
        for j in range(matrix_size):
            for k in range(matrix_size):
                result[i][j] += a[i][k] * b[k][j]
    end_time = time.time()
    print(f"Naive matrix multiplication took {end_time - start_time} seconds")

# 优化后的矩阵乘法函数，利用局部性原理和缓存
def optimized_matrix_multiply(a, b):
    start_time = time.time()
    for i in range(matrix_size):
        for k in range(matrix_size):
            for j in range(matrix_size):
                result[i][j] += a[i][k] * b[k][j]
    end_time = time.time()
    print(f"Optimized matrix multiplication took {end_time - start_time} seconds")

# 测试
naive_matrix_multiply(matrix_a, matrix_b)
optimized_matrix_multiply(matrix_a, matrix_b)

```

3.程序演示：函数缓存

在python中可以使用内置的 `functools.lru_cache` 装饰器来实现函数级别的结果缓存，代码如下。

```
import time
from functools import lru_cache

# 模拟一个耗时的计算函数
@lru_cache(maxsize=128)  # 设置最大缓存大小为128，可以根据实际情况调整
def expensive_calculation(n):
    print(f"Calculating result for {n}...")
    time.sleep(1)  # 模拟耗时操作
    return n * n
```

```
10
11  # 测试
12  start_time = time.time()
13
14  print(expensive_calculation(5))  # 第一次调用，会进行计算
15  print(expensive_calculation(5))  # 第二次调用，直接从缓存中获取结果，不
    会进行计算
16
17  print(expensive_calculation(10))  # 不在缓存中，需要进行计算
18  print(expensive_calculation(10))  # 从缓存中获取结果
19
20  print(expensive_calculation(5))  # 之前已经计算过，直接从缓存中获取结果
21
22  end_time = time.time()
23  print(f"Total time taken: {end_time - start_time} seconds")
24
```

在上例中，`expensive_calculation` 函数模拟了一个耗时的计算任务，通过 `@lru_cache` 装饰器，将函数的结果缓存起来。在测试中，我们可以看到第一次调用会进行计算，然后结果被缓存，后续相同的输入参数再次调用时，直接从缓存中获取结果，不再进行计算。

这是使用函数模拟了对函数结果的缓存，是Cache的一个可能的应用。

4.程序演示：Web缓存

当涉及到 Web 应用程序优化时，缓存也可以用于存储页面内容或响应结果，以减少对数据库或其他外部资源的频繁访问。以下是一个使用 Flask 框架的简单示例，演示如何使用 Flask-Caching 扩展来实现页面级别的缓存：

```
1   from flask import Flask, render_template_string
2   from flask_caching import Cache
3
4   app = Flask(__name__)
5   app.config['CACHE_TYPE'] = 'simple'  # 使用简单的内存缓存，实际生产环
    境可使用其他缓存方式，如Redis
6   cache = Cache(app)
7
8   # 模拟一个耗时的页面渲染函数
9   @cache.cached(timeout=60)  # 设置页面缓存时间为60秒，可以根据实际情况调
    整
10  @app.route('/')
```

```
def index():
    print("Rendering index page...")
    # 此处可以是复杂的页面渲染逻辑，这里简单起见只返回一个字符串
    return render_template_string('<h1>Hello, World!</h1>')

if __name__ == '__main__':
    app.run(debug=True)

```

在该示例中，`index` 路由函数模拟了一个耗时的页面渲染任务，并且使用了 `@cache.cached` 装饰器来缓存页面内容。当第一次访问页面时，会进行页面渲染，并将结果缓存起来。后续相同的请求会直接从缓存中获取响应结果，而不再执行页面渲染过程。

这种方式可以有效减少页面渲染的次数，提高网站的性能和响应速度。同时，Flask-Caching 还支持更多高级的缓存功能，如缓存特定请求参数、基于 URL 的缓存失效、自定义缓存键等，可以根据实际需求进行配置和调整。