2022 年 CCF 大学生计算机系统与程序设计竞赛

CCF CCSP 2022

时间：2022 年 12 月 7 日 09:00 ∼ 15:00

题目名称	最少充电次数	高性能 RDF 图查询系统	虚拟内存管理
题目类型	传统型	传统型	传统型
输入	标准输入	标准输入	标准输入
输出	标准输出	标准输出	标准输出
每个测试点时限	60.0 秒	5.0 秒	1.0 秒
内存限制	1024 MiB	4096 MiB	512 MiB
子任务数目	0	0	0
测试点是否等分	是	是	是

最少充电次数（charging-times）

【题目背景】

小 A 本周末受邀参加一个 CCF 举办的 CNCC，正巧小 A 最近响应节能减排号召购入了一辆电动汽车，因此计划自驾前往参会。由于天气渐冷，外加路途较远，电动汽车的续航无法直达会场，需要在沿途进行充电。小 A 希望你能帮他制订充电计划，使其能够在限定充电时间内用最少的充电次数到达会场。

【题目描述】

电动汽车从起点出发驶向会场，距离目的地`total_distance`公里，沿途有`nr_srv_area`个充电服务区，通过查阅地图可以得知第`i`个充电服务区距离起点`distance[i]`公里，其中`0 <= i < N`。由于每个充电服务区的客流量都比较大，每个人的充电时间有限，无法保证小 A 能够将爱车完全充满电，预计在第`i`个充电服务区最多能够充电`time_limit[i]`小时，每小时能够充入`charge_speed[i]`度电，其中`0 <= i < N`。另外，小 A 希望充电总时间不要超过`max_charge_time`小时以节约自己的时间。为了简化题目，假设小 A 的汽车电池容量没有上限，而从起点出发时汽车电量为`init_energy`度电，每行驶 1 公里就会消耗 1 度电，每当小 A 到达一个充电服务区都可能停下来充电。

请你帮小 A 计算，电动汽车能否顺利到达会场？如果可以，最少需要充电多少次？

注意：如果汽车到达会场时剩余电量为 0，仍然认为小 A 已经到达目的地。

【输入格式】

从标准输入读入数据。

输入的第一行包含用空格分隔的四个自然数`total_distance`（起点与目的地间总距离）、`nr_srv_area`（充电服务区总数），`max_charge_time`（最多充电总时间限制），`init_energy`（初始电量）。

输入的第二行包含`nr_srv_area`个用空格分隔的自然数`distance[0], distance[1], ..., distance[nr_srv_area - 1]`，代表每个充电服务区距离起点多少公里。

输入的第三行包含`nr_srv_area`个用空格分隔的自然数`time_limit[0], time_limit[1], ..., time_limit[nr_srv_area - 1]`，代表每个充电服务区最多能够充电多少小时。

输入的第四行包含`nr_srv_area`个用空格分隔的自然数`charge_speed[0], charge_speed[1], ..., charge_speed[nr_srv_area - 1]`，代表每个充电服务区每小时能够充入多少度电。

输入数据规模如下：

```
1 <= total_distance <= (1<<30)
1 <= nr_srv_area <= 512
1 <= max_charge_time <= 32768
```

```
1 <= init_energy <= (1<<30)
1 <= distance[0] < distance[1] < ... < distance[nr_srv_area - 1] <
    total_distance
1 <= time_limit[i] <=  100, 0 <= i <= nr_srv_area - 1
1 <= charge_speed[i] <= (1<<30), 0 <= i <= nr_srv_area - 1
```

【输出格式】

输出到标准输出。

输出一行，包含一个整数N。若N == -1则表示小 A 无法有足够的电量到达会场，否则N表示小 A 能够到达会场的最少充电次数。

【样例 1 输入】

```
1 0 0 1
```

【样例 1 输出】

```
0
```

【样例 1 解释】

【样例 2】

见题目目录下的 *2.in* 与 *2.ans*。

【提示】

解题思路与优化方向：

```
// F(x, y, z)表示前x站充y次电花费z小时，可运行的最大公里数

F(x, y, z) = max{ F(x - 1, y, z),
    F(x - 1, y - 1, z - k) + k * charge_speed[x] |
        0 <= k <= min(time_limit[x], z) &&
            F(x - 1, y - 1, z - k) >= distance[x] }

F(x, 0, 0) = init_energy

```

```
1 <= x <= nr_srv_area
1 <= y <= x
y <= z <= min(max_charge_time, sum(time_limit[1]...time_limit[x]))
```

根据状态转移方程，可以估计时间复杂度是 x,y,z,k 四个循环长度的积，空间复杂度是 x,y,z 最大值的积。

时间复杂度：`O(nr_srv_area^2 * max_charge_time^2)`

- 在 r752 上测试，样例 10 的运行时间是 60s

空间复杂度：`O(nr_srv_area^2 * max_charge_time)`

- 如果 y 相对于 x 是外层循环，则计算与新的 y 相关的 F，只需要 y-1 和当前的 y，之前的 y 都不需要存储。所以 y 那一列只需要存两行，空间复杂度优化为 `O(nr_srv_area * max_charge_time)`。
- 在 r752 上测试，样例 10 占用 1033MB 内存。
- 可以进一步将数据类型从 uint64_t 改为 uint32_t 后减少到 522MB。

高性能 RDF 图查询系统（rdf）

【题目背景】

大数据时代，信息是高度非结构化和相关联的，比如，每个学生会上不同课程，不同课程有不同导师，不同导师又隶属不同学院；如何查询相应的信息是一个重大的挑战，例如：该如何高效查找给张三上课的所有导师及他们所在的学院？John Smith 听说 RDF(Resource Description Framework) 是一种描述非结构化的数据模型，可以高效支持非结构化数据的复杂查找。John 已经将学校数据按照 RDF 进行存储，希望你帮他实现一个高性能的查询系统。

【题目描述】

1. 什么是 RDF？

RDF 是一种用图来描述数据之间关联的数据模型，简单来说，RDF 将数据表示成一张图，如图 1 所示：“张三选修操作系统”可以表示为两个顶点（张三和操作系统），他们由一条边（选修）进行关联。这样，所有数据可以按“三元组”的方式进行高效存储，即数据表示为 <subject, predicate, object> 的集合（即主语、谓语、宾语），其中 subject 和 object 即为图的顶点，而 predicate 则为边。图 2 展示了图 1 数据的 RDF 表示方法。需要注意的是，一个顶点（如张三）可能会有多条边，如他可以同时选修多门课程。

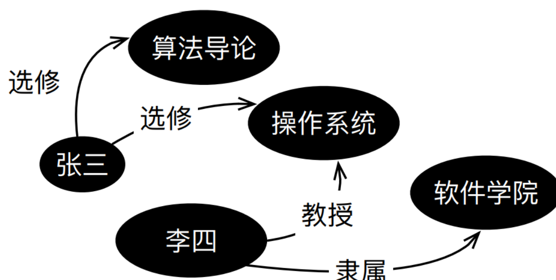


图1: 按图的方式呈现RDF数据

Subject	Predicate	Object	
张三	选修	操作系统	Triple
张三	选修	算法导论	
李四	教授	操作系统	
李四	隶属	软件学院	
…	…	…	

图2: RDF数据（三元组集合）

图 1: RDF 样例

2. RDF 数据存储

为了避免数据冗余，比如将「操作系统」这一字符串在两个顶点中存储两遍，我们需要将 triple 里的真实数据（RDF data）进行编码，即转化为整数（ID-format），随后将其进行存储。图 3 将本题的示例进行编码，将操作系统编码为了 0，张三编为 1，依次类推。显然，编码可以极大减少内存存储的数据。为了能从 ID 找回数据本身，系统也需要维护一个 String-ID mapping，存储 ID 到 String 的映射。**在本题中，John 已经对数据完成了编码和映射**

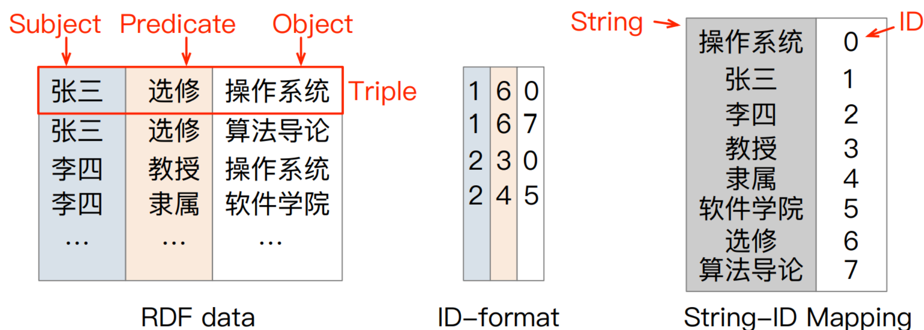


图 2: 数据编码

有了编码后，John 能很方便的对数据**存储在内存中**。由于 RDF 图基本是**稀疏**的，John 目前采用邻接表的方式存储 RDF 数据，如下所示：

```
struct Edge {
    u64 predicate_id;
    u64 subject_id;
}

struct Vertex {
    std::vector<Edge> edges;
}

typedef RDFStore std::unordered_map<u64, Vertex>;
```

3. RDF 查询语言

当讲 RDF 数据进行编码并存储后，我们可以对其进行查询以回答“给张三上课的所有导师及他们所在的学院”这类复杂查询。W3C 提出了一种查询语言 SPARQL，其核心为 Q := select RD where {GP}。其中 RD 为需要查询的顶点，而 GP 是顶点需要满足的条件。例如，“给张三上课的所有导师及他们所在的学院”可以使用如下查询语言进行描述：其中，我们首先找到张三所选修的课（X），随后根据 X 找到教授 X 的老师（Y），最后通过隶属这条边找到老师所在的学院（Z）。在我们的例子中，最后得出的结果是：李四（Y）软件学院（Z）。

和 RDF 数据存储一样，我们也可以将查询语言 ID 化，上图右边展示了查询语言的 ID-format 化。为了区分变量（X，Y）和常量（张三），我们采用正数来 encode 常量，采用负数来 encode 顶点变量。例如，第一个变量 X 的 ID 为-1，Y 为-2 等等。这样，我们可以有效将一个 Q := select RD where {GP} 的语句转化为系统能更好识别的形式。**John 已经实现了查询语言的 ID-format 化**。

4. RDF 查询语言的执行

在存储了 RDF 数据和将查询语言 ID-format 化后，系统可以很方便的进行查询。

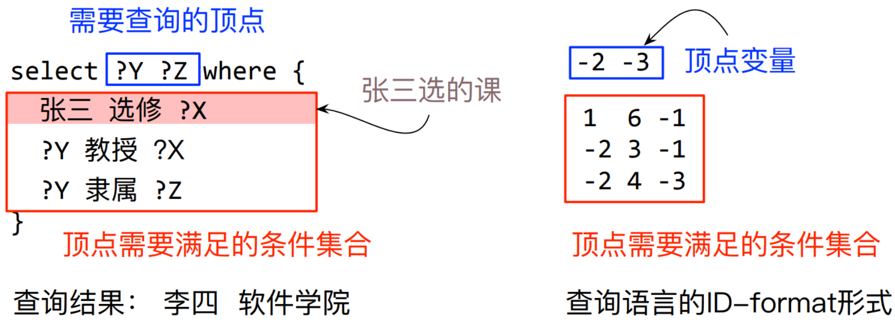


图 3: SPARQL 示例

例如，在获取了“给张三上课的所有导师及他们所在的学院”的 ID-format 形式 [[1, 6, -1], [-1,3,-2], [-2,4,-3]] 后，John 开始逐行查询所需要的变量。首先，条件一：

```
1 6 -1  // 张三 选修 ?X
```

可以找到：

```
操作系统
算法导论
```

条件二：

```
-2 3 -1  // ?Y 教授 ?X
```

可以找到：

```
操作系统 李四
```

条件三：

```
-2 4 -3  // ?Y 隶属 ?Z
```

可以找到：

```
操作系统 李四 软件学院
```

最后，剔除掉不需要的变量（X），可以得到最终结果（在本题的实现中，我们默认需要所有的变量，因此不需要剔除变量）：

```
李四 软件学院
```

John 目前采用了一种顺序的方式执行所有的查询。你的任务是：加速 John 给出的 RDF 语句查询的效率。

你可以从这几个方面优化 RDF 查询的性能：

1. 图存储在内存中的处理
2. 多线程/进程并行处理一个查询请求（比如对数据进行分块处理）

3. 优化查询执行的算法
4. 其他合理的优化方法

【代码框架】

我们提供了一个代码框架（使用 C++ 作为编程语言），帮助你完成了一些繁琐的操作，包括：

1. 读取并解析 RDF 三元组文件，将三元组排序、去重后，存到一个内存中的数组中。
2. 读取 SPARQL 查询文件，并存储在内存中的一个数据结构中。

你需要实现如下两个功能：

1. 设计并实现你的图存储数据结构（只需要存储在内存中）。你会通过上述经过处理的 RDF 三元组的数组来初始化你的图存储数据结构。
2. 实现查询算法。

图存储数据结构对应的 C++ 代码如下：

```
class MyGraphStore : public GraphStore {
    public:
        void insert_from_pos(const std::vector<Triple> &pos) override {}
        void insert_from_pso(const std::vector<Triple> &pso) override {}
        Option<std::unique_ptr<GraphStoreIterator<u32>>> get_iterator_by_subject_predicate(u32 sid, u32 pid) override {}
        Option<std::unique_ptr<GraphStoreIterator<u32>>> get_iterator_by_object_predicate(u32 oid, u32 pid) override {}
};
```

`MyGraphStore`是你的图存储数据结构，你可以在这个类中自定义字段来实现想要的图存储的功能。

`insert_from_pos`和`insert_from_pso`分别传入了按照 **predicate-object-subject** 和 **predicate-subject-object** 排序的三元组。这两个函数中，你**必须**实现**至少一个**来初始化你的图存储数据结构。

`get_iterator_by_subject_predicate`和`get_iterator_by_object_predicate`代表了两个有代表性的查询方式，从 subject/object 和 predicate 的组合，查询符合条件的 object/subject，返回一个`GraphStoreIterator`数据结构。这是一个类似于迭代器的数据结构，遍历该迭代器即可查询到所有的符合条件的 object/subject 的值。这两个函数只是用作示例，你可以不实现这两个函数。

你也可以为你的图存储数据结构添加自己的方法。

查询算法对应的 C++ 代码如下：

```
class MyQueryExecutor {
    public:
        static QueryResult* execute_query(MyGraphStore& g,
   SPARQLQuery& query) {}
};
```

你需要实现这里的execute_query函数，该函数接收你的图存储数据结构和一个 SPARQL 查询作为函数参数，返回一个指向查询结果的QueryResult指针。

SPARQLQuery和QueryResult的数据结构的代码如下（你无需实现这部分代码，你也不能修改这部分的代码）：

```
class SPARQLQuery {
   private:
       SPARQLQuery() : step(0) {}
   public:
       std::vector<std::string> pattern_strings;
       std::vector<Pattern> patterns;
       usize step;
       QueryResult result;
};

class QueryResult {
   public:
       std::vector<u32> result_table;
       std::unordered_map<u32, usize> variable_to_column;
       usize num_vars;
       usize num_rows;
};
```

你可以读取并使用SPARQLQuery中的所有的字段，其中pattern_strings字段是查询条件的 ID-format 化字符串的数组。每个字符串是由空白字符（空格或制表符）分割的三个数字；patterns字段是查询条件经过解析之后的数组（可以参考代码框架中Pattern这个数据结构的实现）；剩下的step和result字段你可以选择性的使用。例如，你可以把查询结果存在result字段中，然后返回指向该字段的指针。代码框架中确保SPARQLQuery对象的生命周期足够长。

QueryResult是存储查询结果的数据结构。num_vars代表结果中变量的个数;num_rows代表结果的行数。result_table是按行存储结果的数组。例如，假设结果有 3 个变量，行数为 2，结果为 [[x1,y1,z1],[x2,y2,z2]]. 那么result_table数组里面的值就可以是

[x1,y1,z1,x2,y2,z2]. `variable_to_column`记录了变量编号到按行存储的每个元素的偏移的映射。例如假设 x 变量 ID-format 化后的编号是-1, y 变量 ID-format 化后的编号是-2, z 变量 ID-format 化后的编号是-3. `variable_to_column`中存储的映射为 `1->2, 2->1, 3->0`, 那么最后`result_table`数据里面的值就为 [z1,y1,x1,z2,y2,x2] 或 [z2,y2,x2,z1,y1,x1]. `variable_to_column`要和`result_table`里面变量的顺序相对应，否则结果会被判为错误。注意`variable_to_column`键值对中的键为未知变量 ID-format 化后的编号的相反数。

另外，在代码中我们也提供了两个钩子函数，分别在查询执行前和程序退出前执行，你可以自由实现这两个函数。

```
class Hook {
    public:
        static void before_query_execution() {}

        static void before_exit() {}
};
```

【RDF 查询示例】

RDF 示例图如下：

```
15 1 13
12 1 13
14 1 10
15 2 12
15 2 14
```

查询 1 如下：

```
-1 1 -2
-3 1 -2
-1 2 -3
```

查询 1 的结果如下：

```
15 13 12
```

查询 2 如下：

```
-1 2 -2
```

查询 2 的结果如下：

```
15 12
15 14
```

注意在QueryResult中,每个元素在按行存储中的偏移要和variable_to_column对应，行与行之间的顺序可以是随机的。

【子任务】

题目总分为 100 分，分为两个部分。第一部分测试用例为简单的 SPARQL 查询测试，第二部分测试用例为复杂的 SPARQL 查询测试。两部分的测试用例的分数占比和其他详细信息见下表。

测试点	分值	单个查询中的条件数目	单个查询中的未知变量的数目	最长执行时间
$1 \sim 1000$	20	≤ 2	≤ 2	$5s$
$1001 \sim 1005$	80	≥ 2	≥ 2	$2s$

【评分方式】

第一部分的测试我们会连续执行所有的测试用例，然后统计测试用例执行的总时间。你所用的总时间不能超过规定的最长执行时间。如果你有任何一个查询得到了错误结果，那么第一部分的分数将会被判为 0 分。如果你成功得到了所有查询的正确结果，且执行时间没有超时，你就会得到第一部分的满分 20 分。

第二部分的测试我们会单独执行每一个查询，然后统计各个查询单独执行的时间。每个查询占 16 分，共 80 分。对于单个查询，你所用的时间不能超过规定的最长执行时间。如果你有某个查询得到了错误结果，那么该查询的分数将会被判为 0 分。如果你成功得到了某一个查询的正确结果，且执行时间没有超时，那么该查询的分数将会根据你的查询的执行时间给分。具体的公式如下，设每个查询的满分是 $s = 80$, 我们给出的最长执行时间为 t_{max}, 选手中最快的执行时间为 t_{min}, 你的查询时间为 t, 那么你该查询的分数为

$$s \times (1 - \frac{t - t_{min}}{t_{max} - t_{min}})$$

即执行时间最快的选手会获得该查询的满分，剩余选手的分数将会根据选手具体的执行时间在整个执行时间区间里面的位置给分。

【提示】

你可以参考题目描述中的几个优化点来进行优化。另外，评测平台提供了 4 个 CPU 核心，请不要创建过多的进程或者线程以免带来额外的开销。

虚拟内存管理（virtual-memory）

【题目背景】

虚拟内存（Virtual Memory）技术是现代计算机系统结构中不可分割的一部分，它使得应用程序认为它拥有连续可用的内存，而物理内存通常被分隔成多个内存碎片，还有部分暂时存储在外部存储器上，在需要时进行数据交换。虚拟内存技术使得大型程序的编写变得更容易，无需关心底层物理内存布局（可移植性），对真正的物理内存的使用也更有效率（高利用率），还可以隔离不同进程所能访问的物理内存区域（安全性）。

现代主流 CPU 架构提供了硬件内存管理单元（Memory Mangement Unit，MMU）来辅助支持虚拟内存技术。如图 1 所示，硬件 MMU 按页粒度将程序尝试访问的虚拟地址（Virtual Address，VA）作为输入，根据操作系统配置的多级页表（Page Table）结构查找对应的页表项（Page Table Entry，PTE），根据页表项内容中的物理地址（Physical Address，PA）、权限和状态信息，决定本次程序访存的合法性（例如，是否存在对应物理内存，是否符合读/写/可执行权限等）。

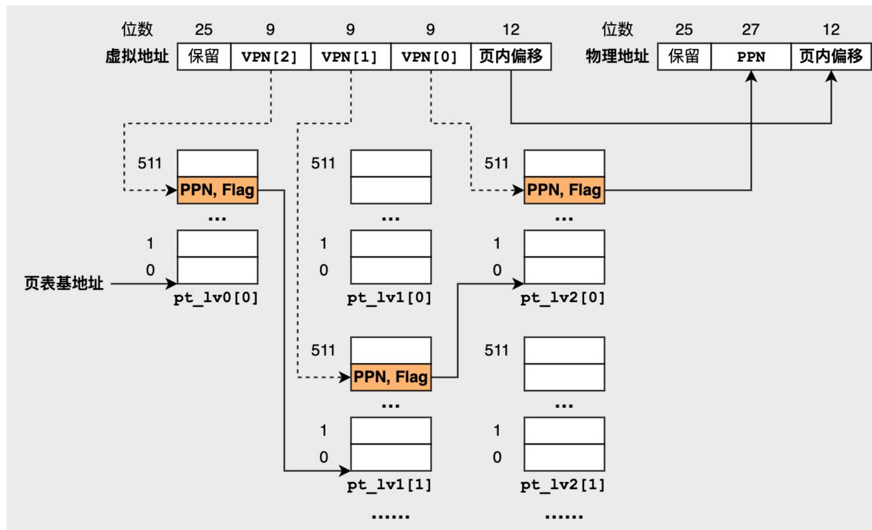
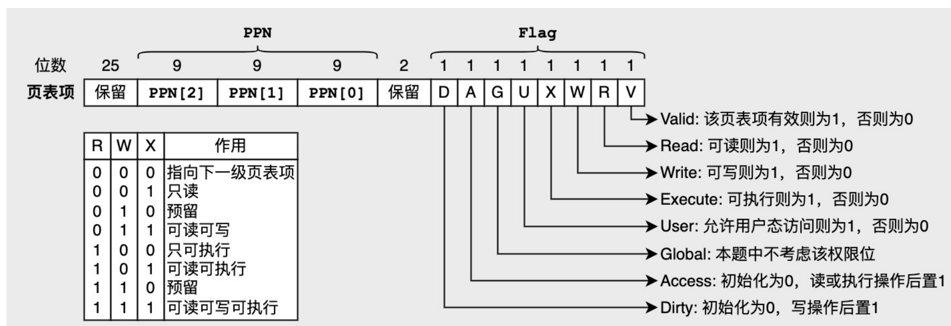


图 4: 39 位虚拟地址空间下三级页表地址翻译示意图



R	W	X	作用
0	0	0	指向下一级页表项
0	0	1	只读
0	1	0	预留
0	1	1	可读可写
1	0	0	只可执行
1	0	1	可读可执行
1	1	0	预留
1	1	1	可读可写可执行

图 5: 页表项权限与状态信息示意图

如果每次地址翻译流程都查询多级页表，则程序的每 1 次访存都会因为 3 次页表访存被放大到 4 次，这无疑大幅降低了系统性能。因此现代 CPU 通常内置了翻译旁路缓存（Translation Lookaside Buffer，TLB）来基于时间和空间局部性记录近期已完成的虚拟页到物理页的映射，然而 TLB 大小是有限的，无法覆盖所有的页表映射。采用大页映射是常见的一种优化方法，可以增加 TLB 覆盖率、减少页表级数以提升系统性能，但是同时可能造成内存碎片导致利用率低下甚至内存不足等问题。现代操作系统通常会根据当前运行状态来动态地调整页表结构与 TLB 内容，例如合并小页或拆分大页，以平衡系统的性能与可用性。

除此之外，现代操作系统通常还会利用无效的页表项存储内存交换（Swap）的有关信息。例如，当应用程序对于内存大小的需求超过物理内存实际容量时，操作系统需要将部分物理内存中的数据换出（Swap Out）到外部存储介质中，这样就可以空出这部分物理内存满足新的内存分配请求。为了保证正确性，在换出操作之前需要先将对应物理内存涉及的页表映射删除。而如果后续应用程序再次访问到被换出的数据则会因为页表映射缺失而触发缺页错误，这时操作系统需要进行换入（Swap In）操作，即将数据从外部存储介质中加载回物理内存并恢复原有页表映射。恢复页表映射需要知道原有映射对应的物理地址，为了便于实现，现代操作系统通常在换出操作删除原有映射时，依旧将物理地址保留在无效的页表项中，则在恢复映射时只需将重新填写页表项中的权限和状态信息。

【题目描述】

在本题中，你需要实现一套虚拟内存管理系统，支持以下功能：

1. 根据操作系统的请求，为 N 个进程建立与维护多级页表，并在每次页表维护操作后输出当前的页表结构。
    - 新增映射
        1. 基本操作：在指定进程的页表中建立从指定 VA 到指定 PA 的映射，配置正确的权限和状态信息。
        2. 附加操作：内存换入，需要根据页表项中保留的 PA 信息恢复原有映射。
    - 修改映射
        1. 基本操作：为指定进程修改指定虚拟地址区间的权限和状态信息。
    - 移除映射
        1. 基本操作：在指定进程的页表中移除指定 VA 的映射。
        2. 附加操作：内存换出，需要在移除指定进程中指定 VA 映射的同时在无效页表项中保留 PA 信息。
        3. 附加操作：换出系统中所有未被访问过的物理内存。
    - 拆分 2MB 大页映射

        1. 基本操作：将指定进程中的某个 VA 所属的 2MB 大页映射拆分为 512 个 4KB 小页映射，每个 4KB 小页映射的权限和状态信息与拆分前的 2MB 大页映射保持一致。
        2. 附加操作：拆分指定进程或系统中全部进程的所有 2MB 大页映射。
    - 合并 4KB 小页映射
        1. 基本操作：合并指定进程中的指定 VA 范围内的 4KB 小页映射，需要注意合并需要满足以下条件：
            - 指定 VA 范围内有足够多的、连续的 4KB 小页。
            - 4KB 小页的起止范围需要 2MB 对齐。
            - 合并的所有 4KB 小页的权限和状态信息，要么完全一致，要么能够 Copy-on-Write（即只读与可读写权限可以合并为只读）。
        2. 附加操作：合并指定进程或系统中全部进程的所有可能的 4KB 小页映射。
2. 接收不同进程的访存请求，根据对应的页表进行地址翻译与权限检查，输出翻译检查结果，需注意跨页访存。
    - 合法性报错：不存在物理地址映射
    - 权限检查报错：权限不匹配
        1. 用户态/内核态
        2. 可读
        3. 可写
        4. 可执行
    - 检查通过：无上述所有报错
        1. 维护 Access 状态
        2. 维护 Dirty 状态
        3. 访问对应物理地址

为了简化题目，假设访存报错后系统不会阻塞或崩溃，可以继续处理后续页表维护与进程访存请求。

为了简化全局页表维护操作（如换入换出、大小页转换），假设系统中进程最多不超过 3 个。

为了简化输入输出，我们要求每个三级页表结构以三个 64 位无符号整型数组的形式呈现，即使用一个 64 位无符号整数模拟一个页表项，一个 4KB 页包含 512 个页表项。实际的操作系统中页表项指向的下一级页表其物理地址并不固定，但出于实现简单考虑，我们要求你使用提前分配好的静态数组来进行模拟。这也就是说每个页表项所指向的下一级页表是已经确定好的，例如图中`pt_lv0[0]`指向`pt_lv1[0...511]`而`pt_lv0[1]`指向`pt_lv1[512...1023]`。为了保证页表维护的正确性，在评测时除了输出 va 映射到的 pa 之外，我们还要求你输出每一级页表中的有效内容。

- 如下图所示，上一级页表节点指向下一级页表节点时，使用数组索引替换物理页地址。

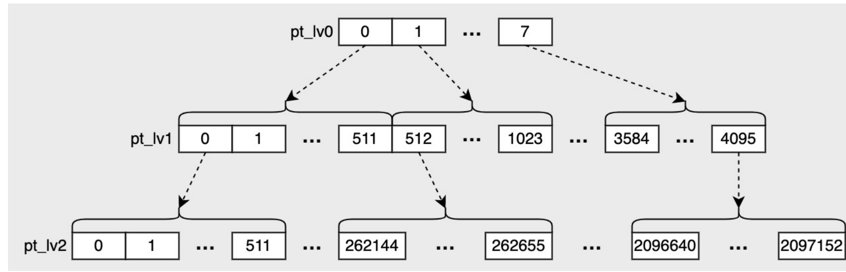


图 6: 使用数组索引替换物理页地址示意图

- 如下图所示，一个简单的三级页表可以用 3 个 pt_lv0, pt_lv1, pt_lv2 数组表示。对于 0x401000 这个虚拟地址，其三级页表的偏移量分别为 [0, 2, 1]。首先索引到 pt_lv0[0]，由于 pt_lv[0] 指向 pt_lv1[0...511]，因此偏移 2 找到 pt_lv1[2]。pt_lv1[2] 管理 pt_lv2[1024...1535]，偏移 1 后最终找到 pt_lv2[1025]。0x2af009 代表虚拟地址为 0x401000 的虚拟页映射到了物理地址为 0xabc000 的物理页，权限与状态信息为可读、不可写、可执行、不允许用户态访问。

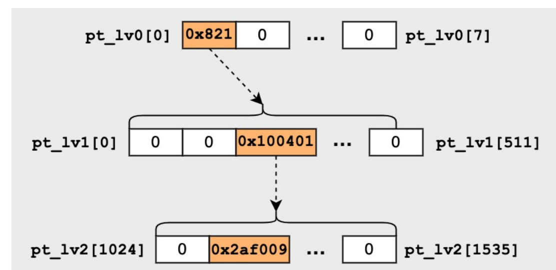


图 7: 页表项权限与状态信息示意图

为了防止虚拟内存管理系统的内存占用超出评测限制，程序输入保证每个进程的虚拟地址在 0-8MB 范围内。

【输入格式】

从标准输入读入数据。

1. 页表维护请求的输入格式，每行代表一次页表维护请求，请求的所有参数之间以空格分隔
    1. 新增映射
        - 基本操作（操作类型编号为`0`）
            1. 共 9 个参数，从左到右依次为：进程 ID ，操作类型编号，映射区域字节数，起始 VA，起始 PA，是否可读，是否可写，是否可执行，是否属于用户态

2. 例：请求对0号进程维护页表，操作类型为新增映射，映射区域大小为0x1000字节（即 4KB），起始 VA 为0x401000，起始 PA 为0xabc000，不可读，不可写，可执行，不允许用户态访问

```
# pid 0, map (op-type 0), 4KB, VA:0x401000 to
  PA:0xabc000, X only, kernel
0 0 0x1000 0x401000 0xabc000 0 0 1 0
```

- 内存换入（操作类型编号为1）
  1. 共个 4 参数，从左到右依次为：进程 ID ，操作类型编号，换入区域字节数，起始 VA
  2. 例：请求对0号进程维护页表，操作类型为内存换入，换入区域大小为0x1000字节（即 4KB），起始 VA 为0x401000

```
# pid 0, swap-in (op-type 1), 4KB, VA:0x401000
0 1 0x1000 0x401000
```

2. 修改映射
   - 基本操作（操作类型编号为2）
     1. 共个 8 参数，从左到右依次为：进程 ID ，操作类型编号，换入区域字节数，起始 VA，是否可读，是否可写，是否可执行，是否属于用户态
     2. 例：请求对 0 号进程维护页表，操作类型为修改映射，映射区域大小为0x1000字节（即 4KB），起始 VA 为0x401000，可读，可写，不可执行，不允许用户态访问（注意：修改映射操作会同时将 Access 和 Dirty 位置零）

```
# pid 0, update_attr (op-type 2), 4KB, VA:0x401000,
  RW, kernel
0 2 0x1000 0x401000 1 1 0 0
```

3. 移除映射
   - 基本操作（操作类型编号为3）
     1. 共个 4 参数，从左到右依次为：进程 ID ，操作类型编号，映射区域字节数，起始 VA
     2. 例：请求对0号进程维护页表，操作类型为移除映射，映射区域大小为0x1000字节（即 4KB），起始 VA 为0x401000

```
# pid 0, unmap (op-type 3), 4KB, VA:0x401000
0 3 0x1000 0x401000
```

   - 指定内存换出（操作类型编号为4）

   1. 共个 4 参数，从左到右依次为：进程 ID ，操作类型编号，映射区域字节数，起始 VA
   2. 例：请求对`0`号进程维护页表，操作类型为内存换出，映射区域大小为`0x1000`字节（即 4KB），起始 VA 为`0x401000`

```
# pid 0, swap-out (op-type 4), 4KB, VA:0x401000
0 4 0x1000 0x401000
```

- 未访问内存批量换出（操作类型编号为`5`）
   1. 共个 2 参数，从左到右依次为：进程 ID 或系统所有进程，操作类型编号
   2. 例：请求对系统中所有进程（用`-1`表示）维护页表，操作类型为内存换出，换出系统中所有未被访问的内存页

```
# global, swap-out (op-type 5)
-1 5
```

4. 拆分 2MB 大页映射
- 基本操作（操作类型编号为`6`）
   1. 共个 4 参数，从左到右依次为：进程 ID ，操作类型编号，拆分后每个小页字节数，目标 VA（即拆分该 VA 所属大页）
   2. 例：请求对`0`号进程维护页表，操作类型为拆分 2MB 大页，拆分后小页大小为`0x1000`字节（即 4KB），目标 VA 为`0x403000`

```
# pid 0, split (op-type 6), 4KB, VA:0x403000
0 6 0x1000 0x403000
```

- 批量拆分大页（操作类型编号为`7`）
   1. 共个 3 参数，从左到右依次为：进程 ID 或系统所有进程，操作类型编号，拆分后每个小页字节数
   2. 例：请求对`0`号进程维护页表，操作类型为批量拆分大页，拆分后小页大小为`0x1000`字节（即 4KB）

```
# pid 0, split (op-type 7), 4KB
0 7 0x1000
```

5. 合并 4KB 小页映射
- 基本操作（操作类型编号为`8`）
   1. 共个 5 参数，从左到右依次为：进程 ID ，操作类型编号，合并后大页字节数，目标 VA（即合并该 VA 所属大页），是否允许 CoW
   2. 例：请求对`0`号进程维护页表，操作类型为合并 4KB 小页，合并后大页大小为`0x200000`字节（即 2MB），目标 VA 为`0x400000`，允许 CoW

```
# pid 0, promote (op-type 8), 2MB, VA:0x400000, CoW:1
0 8 0x200000 0x400000 1
```

- 批量合并小页（操作类型编号为9）
  1. 共个 4 参数，从左到右依次为：进程 ID 或系统所有进程，操作类型编号，合并后大页字节数，是否允许 CoW
  2. 例：请求对0号进程维护页表，操作类型为合并 4KB 小页，合并后大页大小为0x200000字节（即 2MB），允许 CoW

```
# pid 0, promote (op-type 9), 2MB, CoW:1
0 9 0x200000 1
```

2. 访存请求的输入格式，每行代表一次访存请求，请求的所有参数之间以空格分隔。
   - 访存操作（操作编号为10）
     1. 共 8 个参数，从左到右依次为：进程 ID ，操作类型编号，访存字节数，起始 VA，是否为读，是否为写，是否为执行，是否来自用户态
     2. 例：0号进程请求访存，操作类型为访存，访存大小为0x200000字节（即 2MB），起始 VA 为0x403000，不是读操作，不是写操作，是执行操作，不来自用户态

```
# pid 0, access (op-type 10), 2MB, VA:0x400000, X, kernel
0 10 0x200000 0x400000 0 0 1 0
```

【输出格式】

输出到标准输出。

1. 页表维护请求处理后的输出格式
   - 操作无法完成，输出报错
     1. 第一行包含一个元素1，表示后续 1 行输出报错
     2. 第二行仅包含一个元素-1，表示操作无法完成
     3. 例如指定拆分的大页映射不存在、指定移除的映射不存在、因小页权限不一致无法合并等
   - 操作成功完成，输出页表结构
     1. 第一行包含一个元素9，表示后续 9 行为依次输出的 3 个进程的页表结构

2. 每个进程的三级页表用 3 个数组表示，共输出 3 行，第一行表示第 0 级页表的数组内容，第二行表示第 1 级页表的数组内容，第三行表示第 2 级页表的数组内容。
   1. 每行包含用空格分隔的2个元素，第0个元素是一个自然数（十进制），表示数组中有效叶子节点（即 lv1 的大页和 lv2 的 4K 页）的个数N，第1个元素为该进程该级页表所有有效叶子节点计算出的哈希值 H（十六进制）。哈希值 H 计算方式如下：假设有效叶子节点的格式为`(index, value)`，表示数组第`index`个元素的值为`value`,有效叶子节点i按照`(value << (index % 10))`的方式计算出哈希值 h_i，通过将所有 h_i 的值做异或操作得到 H。
   2. 例：某个三级页表中（注：仅作为说明举例，页表结构不一定正确），第 0 级包含 2 个非零元素，`pt_lv0[0] = 0x2, pt_lv0[1] = 0x4`（无叶子节点），第 1 级包含 0 个非零元素，第 2 级包含 2 个非零元素，`pt_lv2[0] = 0x1, pt_lv2[1] = 0x2`，则`H=(1<<0)+(2<<1)=0x5`。

```
0 0x0
0 0x0
2 0x5
```

3. 每次均按顺序输出系统中所有 3 个进程的每一级页表结构的有效叶子结点数量及哈希值，第 1-3 行表示第一个进程，第 4-6 行表示第二个进程，第 7-9 行表示第三个进程。

2. 访存请求处理后的输出格式
   - 报错情况下，输出包含一行
     1. 合法性报错，报错偏移`0`，输出`-1`（即负`1 << 0`）
        - 映射不存在
     2. 权限检查报错，输出一个负数，对应报错偏移位置为 1
        - 可读，报错偏移 1
        - 可写，报错偏移 2
        - 可执行，报错偏移 3
        - 用户态/内核态，报错偏移 4
     3. 访存范围内的第一次报错即输出报错信息，然后解除本次访存处理
   - 检查通过情况下，输出包含一行
     1. 共 2 个元素，第一个元素`0`表示通过检查，第二个元素表示访存的首地址对应的物理地址
     2. 注意：需要更新权限状态中的 A/D 位

【样例 1 输入】

```
2 0 0x1000 0x3c4000 0xe6e000 1 0 1 1
2 10 0x368 0x3c4322 1 0 0 0
2 10 0x1000 0x3c4311 1 0 0 1
```

【样例 1 输出】

```
9
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
1 0x39b81b0
0 0xe6e322
-1
```

【样例 1 解释】

操作系统请求为2号进程新增了一个 4KB 映射，在三级页表中均新增一个页表项。0号和1号进程无映射。

2号进程进程请求访问0x3c4322开始的1个字节,访问成功,返回物理地址0xe6e322。

2号进程进程请求访问0x3c4311开始的4096个字节，访问失败，合法性报错，因为0xe6f000开始没有映射。

【样例 2】

见题目目录下的 *2.in* 与 *2.ans*。

【评分方式】

共 10 组测试，所有测试都包含访存操作，除访存操作外，前 2 组仅包含新增、修改、移除映射的基本操作，中间 4 组仅包含全部基本操作，后 4 组包含全部基本操作与附加操作。