

2024 年 CCF 大学生计算机系统与程序设计竞赛

CCF CCSP 2024

时间：2024 年 10 月 23 日 09:00 ~ 21:00

题目名称	I/O 任务调度队列	数树	贝壳统计	追踪检测	NUMA 感知的调度系统
题目类型	传统型	传统型	传统型	传统型	传统型
输入	标准输入	标准输入	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	标准输出	标准输出
每个测试点时限	1.0 秒	1.0 秒	2.0 秒	1.0 秒	1.0 秒
内存限制	512 MiB	256 MiB	256 MiB	512 MiB	512 MiB
子任务数目	10	20	25	10	10
测试点是否等分	是	是	是	是	是

I/O 任务调度队列 (ioqueue)

【题目描述】

现在，你的计算机系统里面有 1 个存储设备，该存储设备能够通过两个独立的通道对文件进行读写。通道对于应用发起的任务会采用队列的方式来处理，**先提交到队列的任务先处理**。如果一个任务被提交到对应的通道上时，前面的任务还没完成，该任务需要等待前面任务完成后才能被处理。

请你设计一个系统，对于到达的任务（包含任务到达时间，以及完成对应任务所需的时间），将其分发到两个队列。如果中间存在空白（如第一通道中间没任务提交，但是后续有任务提交上来），中间空白的时间算在时延内。即，通道的时延按照通道上最后一个完成的任务的时间计算。

注意，任务到达时间不等于提交到队列的时间。一个后到达的任务如果先提交到队列，则先进行处理。

请设计系统，使得**最终完成所有任务的时延最小**，即两个通道完成所有任务时延最大的值最小。在最大的通道时延最小的情况下，保证另一个通道的时延最小。例如，如果存在一组队列的分发，通道 1 的时延为 50 的情况下，通道 2 的时延可以为 20 或 30，则通道 2 应使用 20 的时延的任务提交方式。

【输入格式】

从标准输入读入数据。

输入的第一行包含一个正整数 n ，保证 $n \leq 10^3$ 。下面是 n 行，每行包括 1 个自然数，为任务到达的时间（到达时间范围在 `int` 类型内）。提交到存储设备队列的任务所需的时间均为 10。注意，任务的**到达时间**的顺序可能有重叠（同一时间可能到达多个任务），或者乱序（输入中的到达时间不是严格递增的）。

【输出格式】

输出到标准输出。

输出两个自然数，分别对应两个队列完成最后一个任务的时间，先输出小的时间，再输出大的时间。测试保证最大的时间不超过 10^6 。

【样例 1 输入】

```
1 3
2 1
3 2
4 4
```

【样例 1 输出】

1 12 21

【样例 1 解释】

在样例 1 中，第 1 和第 3 个任务被提交到第一个队列。任务 1 在时刻 1 到达，时刻 11 完成；任务 3 在时刻 4 到达，最终的完成时间为 21。第 2 个任务被提交到第二个队列：第 2 个任务在时刻 2 到达，时刻 12 完成。

最终的结果是 12 和 21。

【子任务】

测试用例说明：本题包含 10 个测试用例， n 不大于 10^3 ，且所有到达时间不超过 10^4 。所有数据随机生成。

数树 (tree)

【题目描述】

小 H 和小 Z 喜欢逛郊野公园，他们看到了一棵绿色的参天大树，每个树枝末端和分叉点都有一些果子。现在，他们想知道这棵树果子的分布情况，请你写一个程序帮助他们算一算。具体的问题如下：

现在有一棵初始有 n 个节点的有根树，编号为 $1 \sim n$ ，每个点有一个权重 w_i 。对这棵树做以下 6 个操作：

1. 询问以 u 为根的子树中，点权严格大于 x 的个数；
2. 把 u 的点权改为 x ；
3. 添加一个编号为当前树/森林中节点数 $+1$ 的节点，其父节点为 u ，点权为 x ；
4. 删除编号为 u 与父亲的连边，并且使得 u 变为新的根；
5. 询问以 u 为根的子树中，最小点权的最小节点编号；
6. 将包含 u 的树旋转为以 u 为根节点，保持树上所有元素的连接性不变。

本题要求在线查询修改，每一轮的每一个输入（除每一轮第一个输入的操作编号和截止第一轮输入之前的）都需要异或上一轮的输出才能获得真正的输入。

提示：最终操作完的树上结点数会超过输入的 N 。

【输入格式】

从标准输入读入数据。

第 1 行有一个整数 n ，表示树的节点个数；

接下来 $n - 1$ 行，每行两个整数 u, v ，表示树上一条 u 到 v 的边；

接下来一行有 n 个整数 w_i ，表示每个点的初始点权；

接下来一行有一个整数 m ，表示操作的个数；

接下来有若干行，每一行为一个操作，操作列表如下。

- 1 $u\ v$: 询问以 u 为根的子树中，点权严格大于 x 的个数；
- 2 $u\ x$: 把 u 的点权改为 x ；
- 3 $u\ x$: 添加点权为 x 的节点，父亲为 u ；
- 4 u : 删除编号为 u 与父亲的连边，并且使得 u 变为新的根；
- 5 u : 询问以 u 为根的子树中，最小点权的最小节点编号；
- 6 u : 将包含 u 的树旋转为 u 为根节点。

【输出格式】

输出到标准输出。

对于操作 1、操作 4、操作 5，按照如下的格式输出：

- 1 u, x : 点权严格大于 x 的个数；

- 4 u : 输出以 u 为根的最大点权节点权重。
- 5 u : 输出子树中最小点权的最小节点编号;

【样例 1 输入】

```
1 9
2 1 2
3 1 3
4 2 4
5 2 5
6 2 6
7 2 7
8 3 8
9 3 9
10 6 3 8 4 5 4 5 3 4
11 7
12 5 4
13 6 2
14 3 6 3
15 5 7
16 4 15
17 1 4 7
18 1 5 6
```

【样例 1 输出】

```
1 4
2 8
3 5
4 4
5 4
```

【样例 1 解释】

初始的树有 9 个节点，上一轮答案初始为 0，每一步的过程如下。

- 首先执行 5 4: 以 4 为根的子树只有 4 一个节点，所以编号是 4，上一轮答案更新为 4。

- 第二步由于上一轮答案不为 0，取异或得到真实输入 6 6(2 XOR 4)：将包含 6 的树旋转为 6 根节点，此时他的根节点是 1。
- 第三步由于上一轮答案不为 0，取异或得到真实输入 3 2 7：添加点权为 7 的节点，父亲为 2。无输出不更新上一轮答案。
- 第四步由于上一轮答案不为 0，取异或得到真实输入 5 3：查询结果为 8，上一轮答案更新为 8。
- 第五步由于上一轮答案不为 0，取异或得到真实输入 4 7：将 7 和父节点断开，查询得到的答案为 5，上一轮答案更新为 5。
- 第六步由于上一轮答案不为 0，取异或得到真实输入 1 1 2：查询得到的答案为 4，上一轮答案更新为 4。
- 第七步由于上一轮答案不为 0，取异或得到真实输入 1 1 2：查询得到的答案为 4，上一轮答案更新为 4。

以下为前三步的具体例子，其中节点圆圈内是节点编号，每个节点左下角的方框对应其权重：

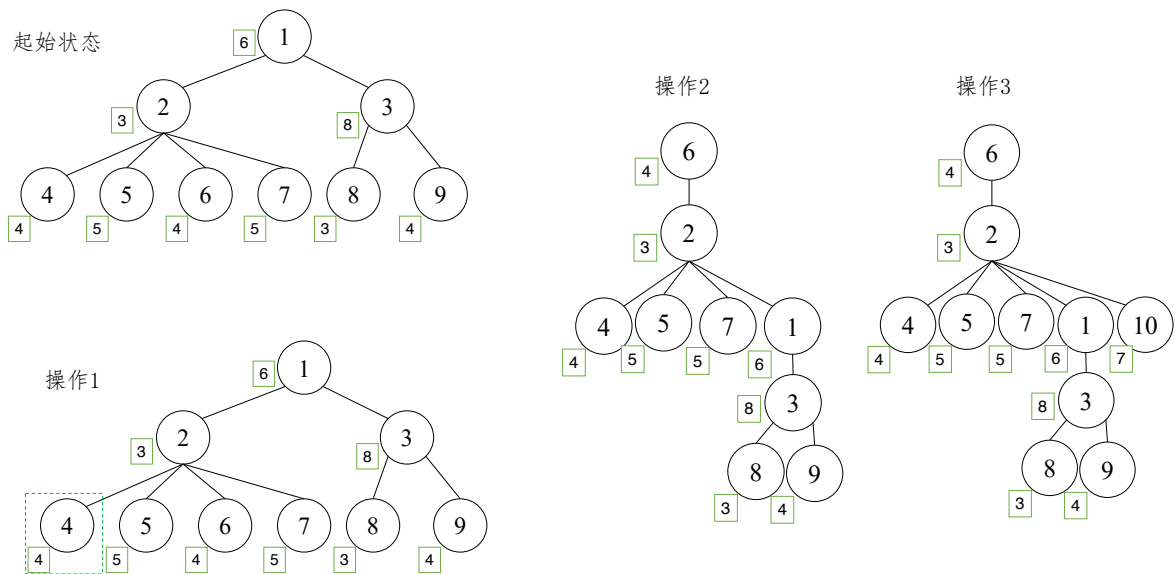


图 1: 前三步的树变换

【样例 2】

见题目目录下的 *2.in* 与 *2.ans*。

【子任务】

对所有的 w_i 满足 $w_i \leq 10^9$, $N \leq 2 \times 10^5$, $M \leq 2 \times 10^5$, 保证树是随机生成的。

测试点	操作类型
1-5	仅操作 1、操作 2
6-11	操作 1、操作 2、操作 3、操作 4
12-20	全部

贝壳统计 (shell)

【题目描述】

A 海滩上放置了一串由不同种类的贝壳组成的贝壳项链。贝壳项链按一条直线顺序放置。现在小 Z 从 I 海滩回到了 A 海滩，他很感兴趣这个海滩上的贝壳情况，希望你帮助他实现以下 3 个任务。

- 1. 统计 $[L, R]$ 区间上贝壳的种类数；
- 2. 更换某一位置贝壳的类别；
- 3. 在某一位置之后放置一个新的贝壳。

【输入格式】

从标准输入读入数据。

输入包含若干行，第一行包含两个整数 N, M ，其中 N 代表贝壳的个数， M 代表操作数。

第二行包括 N 个数代表海滩上初始放置的贝壳种类的编号序列 a_i ，数值范围为 0 至 $2N$ 的整数。

第三行至第 $M + 2$ 行代表具体操作，操作为以下三类：

- 1 $L\ R$: 查询 $[L, R]$ 区间上贝壳的种类数， L 和 R 为 1-based 下标。
- 2 $P\ V$: 更换第 P 个位置的贝壳为 V ，保证 V 为 0 至 $2N$ 的整数，下标为 1-based 下标。
- 3 $P\ V$: 在第 P 个位置后插入一个编号为 V 的贝壳，保证 V 为 0 至 $2N$ 的整数，下标为 1-based 下标。

【输出格式】

输出到标准输出。

对于每一个查询，输出查询的结果。

【样例 1 输入】

```
1 6 5
2 1 1 2 3 4 1
3 1 1 6
4 2 1 5
5 1 1 3
6 3 1 1
7 1 1 3
```

【样例 1 输出】

```
1 4
2 3
3 2
```

【样例 1 解释】

- 对于第一个操作 1 1 6，查询 [1,6] 贝壳种类数为 4。{1,2,3,4}
- 对于第二个操作 2 1 5，将第 1 个贝壳变更为编号为 5 的贝壳，此时贝壳为 (5,1,2,3,4,1)。
- 对于第三个操作 1 1 3，查询 [1,3] 贝壳种类数为 3。{5,1,2}
- 对于第四个操作 3 1 1，在第一个贝壳后插入编号为 1 的贝壳，此时贝壳为 (5,1,1,2,3,4,1)。
- 对于第五个操作 1 1 3，查询 [1,3] 贝壳种类数为 2。{5,1,1}

【样例 2】

见题目目录下的 *2.in* 与 *2.ans*。

【子任务】

本题保证所有的数据随机生成，输入数据的规模符合下表。样例 2 采用与测试点 21-25 相同的生成器生成。贝壳的种类编号 $a_i \leq 10^6$ ，每次查询 $0 \leq L \leq N$ 且 $0 \leq R \leq N$ 。

测试点	N	M	操作类型
1-2	≤ 100	≤ 200	仅操作 1
3-5	$\leq 10^4$	$\leq 10^4$	
6-7	$\leq 10^3$	$\leq 2 \times 10^3$	操作 1、操作 2
8-10	$\leq 10^4$	$\leq 10^4$	
11-15	$\leq 10^5$	$\leq 10^5$	仅操作 1
16-20			操作 1、操作 2、操作 3（操作 3<10%）
21-25			操作 1、操作 2、操作 3

追踪检测 (tracker)

【题目背景】

小 K 是一位热爱古典游戏的年轻人。这些游戏承载了他童年的美好回忆，但随着时间的推移，许多经典游戏已经在网络上难以找到。小 K 不甘心让这些珍贵的记忆消失，于是他决定采取行动。小 K 发现，通过 BT 协议可以方便地共享文件，于是他萌生了一个想法：组织一个内部的游戏分享社区，让志同道合的朋友们一起来分享和下载这些古典游戏。很快，他在网上找到了几个同样热爱古典游戏的伙伴，大家决定一起行动。然而，小 K 意识到，虽然 BT 协议非常方便，但也存在一些法律和合规问题。如果不加以控制，可能会出现版权侵权等问题。为了保证大家在使用 BT 协议时是合法合规的，他决定写一个 tracker 服务器，在实现基本的 BT 协议功能的同时，加入一些额外的检查机制，以确保共享的游戏都是合法的。

BT 下载 (BitTorrent 下载) 是一种点对点 (P2P) 文件共享协议，通过将大文件分割成小块并在多个用户之间传输，实现高效的文件分发。每个参与下载的用户不仅仅是下载者，同时也充当上传者，分享已经下载到的文件块，极大地提高了下载速度和效率。

Tracker 服务器在 BT 下载中扮演关键角色。它负责管理和协调所有参与下载的用户，记录每个用户的 IP 地址和所拥有的文件块信息。当用户请求下载文件时，Tracker 服务器会提供其他拥有该文件块的用户列表，使得下载者能够从多个源头获取数据。Tracker 服务器本身并不存储文件，只提供元数据和连接信息，确保 BT 网络的正常运行。

在本题中，你需要帮助小 K 实现一个满足要求的 BitTorrent Tracker。

【题目描述】

小 K 要实现的 Tracker 服务器从 **标准输入** 中读取请求。每个请求都会以 1 个回车为结尾 ('\n', ASCII 码为 0x0a)。对于每个请求，服务端都应当向 **标准输出** 写入响应。每个响应都需要以 1 个回车为结尾 ('\n', ASCII 码为 0x0a)。

小 K 的 Tracker 需要实现对以下请求的响应：

announce 请求

一个 announce 请求包含以下参数：

- **info_hash**，请求的资源的标识符，由 20 个可打印的 ASCII 字符组成。
- **peer_id**，20 个字符（均为可打印的 ASCII 字符）的唯一客户端标识符，每个客户端的 **peer_id** 不相同。
- **IP**，客户端的 IP 地址，均为 IPv4 地址。
- **port**，客户端正在监听的端口（十进制数字表示）。
- **uploaded**，当前已经上传的文件的字节数（十进制数字表示）。

- **downloaded**, 当前已经下载的文件的字节数 (十进制数字表示)。
- **numwant**, 可选, 希望 Tracker 返回的 peer 数目, 若不填, 默认返回 50 个 IP 和 Port。
- **event**, 可选, 该参数的值可以是 **started**, **completed**, **stopped**, **empty** 其中的一个, 该参数的值为 **empty** 与该参数不存在是等价的, 当开始下载时, 该参数的值设置为 **started**, 当下载完成时, 该参数的值设置为 **completed**, 当停止下载/上传时, 该参数的值设置为 **stopped**。

如何响应 announce 请求

当收到一个 **announce** 请求时, Tracker 需要先查询客户端所请求的 **info_hash** 是否有效, 若无效, 则报错 **invalid info_hash**。

随后根据 **event** 执行不同的操作:

- 若 **event=started**, 则将用户变为该种子的 leecher, 如果用户已经为 leecher, 则无需更改此状态。
- 若 **event=completed**, 则将该用户变为该种子的 seeder, 如果用户已经为 seeder, 则无需更改此状态。
- 若 **event=stopped**, 则将该用户从 leechers/seeder 中删除, 若用户原本就不是该种子的 seeder/leecher, 则无需操作。
- 若 **event=empty**, 则不需要改变用户的 leecher/seeder 状态, 若用户原本就不是该种子的 seeder/leecher, 则无需操作。

在上述操作完成之后, 如果用户依然存在, 则更新用户对该种子的 **uploaded**, **downloaded**。注意, 在 **announce** 中的 **uploaded** 和 **downloaded** 的具体数值并非递增, 后续 **announce** 中的 **uploaded** 和 **downloaded** 数值可能会减少, 请不要对其有单调递增的假设。

最后, 根据用户状态返回 **peers** 列表 (可能包括用户自己):

- 用户为 leecher: 返回 **numwant** 个 seeders (若 seeders 不够, 则用 leechers 补足, 如果还不够, 则返回所有 seeders 和 leechers)。
- 用户为 seeder: 返回 **numwant** 个 leechers (若 leechers 不够, 则返回所有 leechers)。
- 用户不存在或者被删除: 返回空的 **peers** 列表。

请求失败时, Tracker 返回 **ERROR** 和错误原因。请求成功时, Tracker 返回 **OK** 和以下内容:

- **complete**, 表明当前已完成整个资源下载的 peer 的数量。
- **incomplete**, 表明当前未完成整个资源下载的 peer 的数量。
- **peers**, 是一个列表, 每一个元素都包含有三个内容, 分别为:
 - **peer_id**, peer 节点的 Id。
 - **IP**, peer 节点的 IP 地址。
 - **Port**, peer 节点的端口。

为了保证结果的唯一性, 若返回的 **peers** 不为空, 则需要按照以下规则排序:

1. 若返回的 peers 全为 seeders, 则需要按照其 uploaded 降序排列, 当出现两个 seeders 的 uploaded 相等时, 按照 peer id 字典序排列, 字典序小的排在前面。
2. 若返回的 peers 全为 leechers, 则需要按照其 downloaded 降序排列, 当出现两个 leechers 的 downloaded 相等时, 按照 peer id 字典序排列, 字典序小的排在前面。
3. 若返回的 peers 同时包含 seeders 和 leechers, 需要将所有的 seeders 排在 leechers 之前, seeders 之间按照上述第一条规则排序, leechers 之间按照上述第二条规则排序。

特殊情况: 对于处于异常状态 (比如被封禁和被冻结) 的种子, 在处理 `event=started` 请求时, 直接报错 `frozen torrent`; 对于 `event` 为其他值的请求正常处理, 但是返回的 peer 列表一律为空。

scrape 请求

无参数。

如何响应 scrape 请求

返回 OK 和 Tracker 记录的所有种子的情况, 包括每个种子的 `info_hash`, 种子是否被封禁, 是否被冻结, 和该种子的做种人数 (`num_seeders`) 和正在下载的人数 (`num_leechers`)。

结果按照种子的 `info_hash` 字典序排列。

add 请求

新增一个种子。

包含一个 `info_hash` 参数, 格式与 `announce` 请求中的 `info_hash` 相同。

如何响应 add 请求

如果种子已经存在, 则无法重复添加, 返回 `ERROR`; 若添加成功, 则返回 `OK`。

del 请求

删除一个种子。

注意, 当一个种子正在被一些 peer 使用时, 无法直接删除该种子。此时, 该种子进入到冻结状态, 新加入的 peer 无法通过 `announce` 获取到该种子的信息, 等所有现存使用该种子的 peer 离线 (或者 `stopped`) 之后, 该种子被真正删除。

包含一个 `info_hash` 参数, 格式与 `announce` 请求中的 `info_hash` 相同。

如何响应 del 请求

如果种子不存在, 则无法删除, 返回 `ERROR`; 若直接成功, 则返回 `OK`; 若种子进入冻结状态 (包括该请求之前已经为冻结状态), 则返回 `FROZEN`。

run 请求

模拟时间流逝, 包含一个 **time** 参数 (十进制数字), 表示过去了多少秒。考虑到客户端可能未发出任何停止请求就直接离线, Tracker 会通过超时机制清理客户端, 即若客户端对某个种子长期不发出 **announce** 请求, Tracker 需要将其从种子的 peers (seeders 或 leechers) 列表中删除。

如何响应 run 请求 更新每个种子的 peers 列表, 超时的 peers 需要从 peers 列表中删除。Tracker 需要记录每个 peer 最近一次 **announce** 的时间, 如果当前时间与最近一次 **announce** 时间超过了 60 秒, 则需要将该 peer 删除。注意, 一个客户端对种子 X 发出的 **announce** 请求并不会刷新其对种子 Y 的 **announce** 时间。

返回 OK。

report 请求

包含一个 **info_hash** 参数, 格式与 **announce** 请求中的 **info_hash** 相同。

如何响应 report 请求

report 请求会导致以下变化:

1. 该种子处于被封禁状态
2. 所有正在上传或者下载该种子的客户端变为被封禁状态。服务器仅处理被封禁客户端的 **event=stopped** 的 **announce** 请求。一个客户端在停止了其所有被封禁的种子的上传和下载后, 其封禁状态才会被解除。
3. 被封禁的客户端不会被作为 seeder 或者 leecher 发送给其他客户端。

如果种子不存在, 则无法举报, 返回 ERROR; 若举报成功, 则返回 OK。

【输入格式】

输入由若干行组成, 每行由空格分隔为若干段, 第一段包含一个请求命令, 表示需要进行的操作, 之后的若干段表示的是操作的参数, 参数名均为大写。

- **announce** 操作: **announce INFO_HASH=0...0 PEER_ID=X...X IP=111.111.111.111 PORT=2222 UPLOADED=0 DOWNLOADED=0 NUMWANT=50 EVENT=started**
- **scrape** 操作: **scrape**
- **add** 操作: **add INFO_HASH=00000000000000000000**, 表示向 Tracker 中添加 **info_hash** 为 00000000000000000000 的种子
- **del** 操作: **del INFO_HASH=00000000000000000000**, 表示删除 Tracker 中 **info_hash** 为 00000000000000000000 的种子记录
- **run** 操作: **run 5**, 表示时间经过 5 秒
- **report** 操作: **report INFO_HASH=00000000000000000000**, 表示举报 **info_hash** 为 00000000000000000000 的种子

【输出格式】

你的输出应由若干行组成，每一行对应一个输入操作的响应。每一行由空格分隔的结果和一个可选的字符串组成，分别表示响应结果（OK/ERROR）和可能存在的返回值，具体的返回值格式如下：

- announce 操作：
 - 若 info_hash 无效，返回如下结果：ERROR: Invalid info_hash
 - 若 info_hash 被封禁，返回：ERROR: Torrent banned
 - 若 info_hash 被冻结，返回：ERROR: Torrent frozen
 - 若客户端被封禁，返回：ERROR: Client banned
 - 若 peers 为空，返回如下结果：OK: COMPLETE=2 INCOMPLETE=1 PEERS=[]
 - 若 peers 不为空，返回如下格式结果：OK: COMPLETE=2 INCOMPLETE=1 PEERS=[(X...X,111.111.111.111,2222),(Y...Y,222.222.222.222,6666)]
- scrape 操作：
 - 若 Tracker 中无任何记录的种子，返回如下结果：OK: []
 - 按照 info_hash 字典序返回 Tracker 记录的所有种子，类似如下结果：OK: [(0...0,B=0,F=0,1,1), (1...1,B=0,F=0,2,1)]，其中 B= 和 F= 分别表示封禁和冻结状态，1 表示有该状态，0 表示无该状态。
- add 操作：返回OK或者ERROR: Torrent already exists
- del 操作：返回OK或者ERROR: Invalid info_hash或者FROZEN
- run 操作：返回如下结果：OK
- report 操作：返回 OK 或者 ERROR: Invalid info_hash

【样例 1】

见题目目录下的 *1.in* 与 *1.ans*。

【样例 1 解释】

下面对应每一行的输出说明：

1. 添加成功
2. 添加成功
3. 添加成功
4. A 开始做种资源 1，没有下载者，所以 peers 为空
5. B 开始下载资源 1，peers 里面为 A 和自己
6. C 开始下载资源 3，peers 里面只有自己
7. D 开始做种资源 2，没有下载者，peers 为空
8. scrape 看到三个种子

9. 删除资源 3, 有人正在下载, 所以冻结
10. scrape 看到冻结状态
11. 时间流逝 20 秒
12. B 开始下载资源 3, 因为资源被冻结, 所以返回错误
13. C 报告其资源 1 的进度, 但是 C 从未宣称上传/下载资源 1, 因此返回基本信息, 但是 peers 返回空
14. A 报告其资源 1 的进度, A 在做种, 所以 peers 中看到下载者 B
15. 时间流逝 20 秒
16. B 报告其资源 1 的进度, B 在下载, 所以 peers 中看到 A 和自己
17. C 开始下载资源 1, 看到做种的 A 和正在下载的 B
18. D 开始做种资源 1, 看到正在下载的 B 和 C, B 下载的更多, 所以排在前面
19. E 开始下载资源 1, 看到正在做种的 D 和 A (D 上传量更多), 然后看到下载的 B (B 比 C 下载量更大)
20. B 开始下载资源 2, 看到正在上传的 D 和正在下载的自己
21. 举报资源 2, 返回成功
22. scrape 看到封禁状态
23. C 报告其资源 1 的进度, C 的状态是正在下载, peers 中看到了正在做种的 A (看不到 D 因为被封禁了)、自己、和正在下载的 E (看不到 B 因为被封禁了)
24. D 停止了资源 2 的做种
25. C 报告其资源 1 的进度, C 的状态是正在下载, peers 中看到了正在做种的 D (已经被解除封禁)、正在做种的 A, 和自己
26. scrape 查看当前状态
27. 时间流逝 50 秒
28. scrape 查看当前状态, 因为时间流逝, 下载资源 3 的 C 信息失效, C 是最后一个访问资源 3 的客户端, 因此资源 3 得以删除, 同时 A 做种资源 1 的信息也过期失效, 因而资源 1 的做种数也减少了 1

【子任务】

测试点	行数	包含的命令
1	100	announce, add, del
2	1000	
3	10000	
4	1000	announce, add, del, scrape
5	10000	
6		announce, add, del, scrape, run
7	50000	
8	100000	announce, add, del, scrape, run, report
9		
10		

NUMA 感知的调度系统 (numaScheduler)

【题目背景】

NUMA (Non-Uniform Memory Access) 是一种处理器内存架构，其设计旨在提高计算机系统的性能和可扩展性。在 NUMA 系统中，多个处理器核心和内存模块分布在不同的物理节点上，这些节点之间通常通过互连网络连接。每个处理器核心会有附近的内存块，这样可以更快地访问该内存，从而减少延迟。然而，当处理器需要访问其他节点上的内存时，就会引入额外的延迟。

NUMA 系统的主要优势在于提高性能和可扩展性。通过将处理器核心和内存分布在多个节点中，系统可以更有效地处理大规模并行计算任务。这种架构特别适合于多处理器系统，可以有效减少内存访问延迟，提高整体性能。

然而，NUMA 系统也面临一些挑战。为了充分发挥其优势，需要精心设计软件来优化内存访问模式。合理的内存分配和访问策略对于避免性能下降至关重要。

NUMA 感知的锁 (NUMA-aware Lock) 是一种处理并发编程时考虑到 NUMA 系统特性的锁机制。在 NUMA 系统中，由于不同处理器核心访问不同物理内存节点的延迟不同，传统的锁机制可能导致性能下降。为了解决这个问题，NUMA 感知的锁会考虑到处理器核心与内存之间的距离，以及内存访问的延迟差异。在实现上，这些锁机制会尝试将锁数据结构和相应的线程关联到同一个 NUMA 节点，以减少跨节点访问造成的性能损失。

【题目描述】

虽然现在 NUMA 感知的锁在锁内部的结构和设计上，已经充分考虑了 NUMA 架构的特性，但是如果存在较多的跨 NUMA 节点的对于锁的竞争，仍然会导致较差的性能表现。现在，你接受了一个任务，通过修改内核调度器，避免大量的同步操作发生在跨 NUMA 节点的情况，进一步优化 NUMA 感知的锁的性能表现。

具体来说，你需要实现一个模拟的内核调度器，这个调度器能够接收一系列的操作，这些操作对应着当前系统的进程/线程的创建情况，它们的负载，以及它们和锁相关的语义。在你实现的调度器中，进程可以包含一个或者多个线程，同一进程内的多线程共享页表和虚拟地址空间。

本题假设机器有 2 个 NUMA 节点。

你的内核调度器需要支持：

1. 进程和线程管理

管理进程的创建和销毁 (CREATE_Proc pid load、DESTROY_Thread tid)。创建进程时会附带对应的负载值 (load) 和对应的进程 ID (pid，测试保证pid始终为非 0 正整数)。当创建一个新的进程时，该进程只有 1 个线程，线程 ID (tid) 即为进程 ID

(**pid**)，线程负载即为**load**（负载和线程绑定）。删除时会传入对应的线程 ID (**tid**)，如果该线程对应的进程此时只有 1 个线程，则整个进程被删除；如果删除时线程所在进程包括多个线程，则只有该线程被销毁。

管理进程创建多线程的操作 (**CREATE_Thread pid tid load**): 创建**pid**进程下对应的新线程，线程 ID 为**tid**。该线程的负载为**load**。如果当前不存在进程的 ID 为**pid**，则属于异常情况，该命令无效。

注意，**pid**和**tid**共用一个 ID 空间。即，对于**CREATE_Proc**命令，如果当前存在已创建进程**pid**或已创建线程**tid**的值与该命令所指定的**pid**冲突，则该操作属于异常情况。在该调度器中，该命令将无效。类似的，**CREATE_Thread**也需要保证和现有的**pid**和**tid**不冲突。题目第 5 部分对异常情况进行了具体描述。

2. 共享内存管理

系统中用基于**共享内存**的锁来实现跨线程、跨进程的同步。因此，调度器需要管理不同的进程间的共享内存关系（同一进程下的多个线程共享全部地址空间）。

需要处理共享内存创建、删除、映射、解除映射的 4 个操作。

首先，**CREATE_Shmem shm_key size**，会创建一个共享内存 ID 为**shm_key**（整型类型），长度为**size**（需要大于 0）的共享内存区域。如果**shm_key**已经被使用，该次创建失败。对应的，**DESTROY_Shmem shm_key**删除对应的共享内存，如果**shm_key**未使用（即没有**shm_key**对应的共享内存资源），或者当前仍然有进程映射该共享内存，该操作失败，否则删除该共享内存，并且将对应的**shm_key**标记为未使用。

其次，**MAP_Shmem shm_key tid va size**，在**tid**对应的线程的进程的地址空间，将编号为**shm_key**的共享内存区域映射到虚拟地址起始地址为**va**，长度为**size**的区域。如果两个不同的进程 P1 和 P2 的地址空间都完成了相同的映射，则他们可以通过读写自己本地对应的虚拟地址区域，来访问到同一个共享内存区域。对应的，**UNMAP_Shmem tid va**，将**tid**对应的起始地址为**va**的共享内存解除映射。如果当前**va**所在的虚拟地址没有映射共享内存，或者不属于共享内存的起始地址（例如，**va**对应一个共享内存的中间地址），则该次**UNMAP_Shmem**操作失败。

注意，为了简化调度器，不对共享内存以及映射操作的大小作对齐限制（例如，不需要考虑按照 4KB 粒度来映射页面）。

3. 锁竞争管理

调度器需要管理锁的竞争情况。

需要处理建锁操作(**CREATE_Lock tid lock_addr**)，即**tid**线程，会在**lock_addr**的虚拟地址上对一把锁进行竞争。如果对于另一个线程**tid2**在**lock_addr2**上拿锁，且两锁所对应的虚拟地址映射到同一个物理内存（即这两个地址经过了共享内存的映射）。那么我们说**tid**和**tid2**在该位置对于同一把锁存在一个竞争关系。注意，一把锁只占用 1 个字节的空间和对应的地址。

对应的锁删除的操作为`DESTROY_Lock tid lock_addr`，即一个线程（对应 ID 为`tid`）会删除掉其对该`lock_addr`虚拟地址的锁的竞争。

由于该调度器仅考虑基于共享内存的锁的情况，创建锁所使用的虚拟地址（`lock_addr`）必须为一段共享内存映射的地址，否则该次锁创建操作失败。

4. 调度

假设调度器的迁移进程/线程的能力非常强，能够实现瞬间的迁移。你的调度器需要支持一个关键命令（`SCHEDULE_OPT`）。此时，根据当前系统的状态，调度器需要对所有的线程、进程在 CPU 核心间进行重新调度。注意如下两个情况：

- (a) **如果当前存在锁的竞争**：如果两个线程`tid1`和`tid2`存在对同一把锁的竞争关系，那么当`tid1`和`tid2`在同一个 NUMA 节点时，竞争开销为1，当在不同的 NUMA 节点时，竞争开销为10。 `tid1`和`tid2`之间如果存在多把锁的竞争关系，则分别计算（例如，在同一个 NUMA 节点内部，且有3个锁的竞争，则开销为3）。如果同一把锁存在大于2个的竞争者，例如存在`tid1`、`tid2`、`tid3`竞争同一把锁，那么两两线程分别计算。例如`tid1`和`tid2`在 NUMA-1 节点上，`tid3`在 NUMA-2 节点上，则全局的锁竞争开销为：`tid1-tid2`竞争开销（1）+ `tid1-tid3`竞争开销（10）+ `tid2-tid3`竞争开销（10）=21。 `SCHEDULE_OPT`需要满足 2 个 NUMA 节点上都至少有 1 个线程在运行（全局至少会有 2 个线程），请设计该函数，使得全局锁竞争开销最小。
- (b) **如果当前不存在锁的竞争**：请输出 0。注意，如果当前创建了一个锁，但是该锁只有 1 个线程在使用，没有其他线程和其竞争，同样不算锁竞争的情况。

5. 异常处理

作为一个系统开发者，你需要对调度器的各种可能的异常情况进行考虑和处理。除了上述的描述中的相关内容，系统的整体的异常处理包括：

- 对于系统中创建资源的命令（例如`CREATE_Proc`、`CREATE_Thread`等），如果该操作和此前的操作所创建的状态冲突，那么该命令无效（跳过该命令，不在标准输出中输出错误信息）。例如，如果 1 个`CREATE_Proc`命令中的`pid`已经存在（和现有的`pid`和`tid`冲突），则该次命令直接跳过。类似的，如果`CREATE_Thread`中`tid`出现冲突，同样跳过命令。如果 1 个`MAP_Shmem`命令中的`va`和`size`对应的区间，在该进程地址空间中已经被映射或者部分映射（存在交替区域），该次命令跳过。
- 对于系统中删除资源的命令（例如`DESTROY_Thread tid`），如果命令对应的资源不存在（即`tid`线程不存在），该命令跳过。
- 对于线程销毁命令（`DESTROY_Thread tid`），在删除该线程时，需要对应地删除该线程所创建的所有的锁。如果删除该线程后，线程对应的进程还有其他线程，则该线程所执行的`MAP_Shmem`的区域仍然保留。如果删除该线程后，进程内没有其他线程，则还需要对进程内所有的共享内存区域进行解映射（`UNMAP_Shmem`）。

- 为了避免映射出错,系统内部需要维护共享内存的引用计算。当处理`DESTROY_Shm shm_key`时,如果该区域仍然被映射在某个进程地址空间中(对应的区域没有执行`UNMAP_Shm`),那么该删除操作失败(直接跳过)。这里,如果不支持这样的依赖维护的处理的话,很有可能程序会错误地删除一个共享内存,然后在之后对其使用,触发类似 `use-after-free` 的安全问题。
- 类似地,对于`UNMAP_Shm`命令,如果当前进程中的任一线程仍然有锁是基于该命令的地址区域,则该次`UNMAP_Shm`操作失败(直接跳过)。

【输入格式】

从标准输入读入数据。

输入的第一行包含 1 个正整数 n , 保证 $n \leq 5,000$ 。

随后包括 n 行, 每一行为一个具体的命令(包括 1 个字符串的命令名和 0 个或若干个参数), 如下(具体的功能在上面题干中已有描述):

- `CREATE_Proc pid load`
- `CREATE_Thread pid tid load`
- `DESTROY_Thread tid`
- `CREATE_Shm shm_key size`
- `DESTROY_Shm shm_key`
- `MAP_Shm shm_key tid va size`
- `UNMAP_Shm tid va`
- `CREATE_Lock tid lock_addr`
- `DESTROY_Lock tid lock_addr`
- `SCHEDULE_OPT`

上述参数中,除了地址(即, `va`、`lock_addr`)是 64 位非负整型,其余均为`int`类型。

【输出格式】

输出到标准输出。

针对所有的`SCHEDULE_OPT`命令,打印 2 部分内容。首先,分别打印出当前所有的线程信息,每行一个线程,顺序以 PID 从低到高,在同一进程内部,按照 TID 从低到高输出。

每行的内容为: `Process (PID): pid Thread(Tid): tid Load: load`

其中, `pid`、`tid`和`load`需要替换为线程所属进程的 PID,线程的 TID, 以及其负载值。

其次,需要打印出当前调度策略下的全局锁竞争开销,例如:

`Contention Cost: 0`

注意，输出要以 1 个回车为结尾 ('\n')。
具体的输出格式，请参考样例输出。

【样例 1 输入】

```
1 14
2 CREATE_Proc 1 20
3 CREATE_Proc 2 30
4 CREATE_Thread 1 3 20
5 SCHEDULE_OPT
6 CREATE_Shmem 100 4096
7 MAP_Shmem 100 1 4096 4096
8 MAP_Shmem 100 2 8192 4096
9 CREATE_Lock 1 4096
10 CREATE_Lock 2 8200
11 CREATE_Lock 2 8192
12 CREATE_Lock 3 4100
13 CREATE_Lock 2 8196
14 CREATE_Lock 3 4104
15 SCHEDULE_OPT
```

【样例 1 输出】

```
1 Process (PID): 1 Thread(Tid): 1 Load: 20
2 Process (PID): 1 Thread(Tid): 3 Load: 20
3 Process (PID): 2 Thread(Tid): 2 Load: 30
4 Contention Cost: 0
5 Process (PID): 1 Thread(Tid): 1 Load: 20
6 Process (PID): 1 Thread(Tid): 3 Load: 20
7 Process (PID): 2 Thread(Tid): 2 Load: 30
8 Contention Cost: 12
```

【样例 1 解释】

在样例 1 中，对于第一个 SCHEDULE_OPT 命令，此时没有锁的创建操作，因此考虑 (b) 的情况，直接输出当前的线程信息，并且锁竞争开销为 0。

对于第二个 SCHEDULE_OPT 命令，此时线程 1 和线程 2 之间有 1 个锁的竞争关系，线程 2 和线程 3 之间有 2 个锁的竞争关系。此时，将线程 1 放在 NUMA 节点 1 的核

心上、线程 2、3 放在 NUMA 节点 2 的核心上，此时锁的竞争情况为 (10+1+1)，为最优情况。此时的输出为12。

【样例 2】

见题目目录下的 2.in 与 2.ans。

【样例 3】

见题目目录下的 3.in 与 3.ans。

【子任务】

所有测试用例中，命令数 (n) 不超过 2000。当存在锁竞争且要计算锁竞争开销时，参与竞争的线程总数不超过 30。

测试用例具体说明如下：

测试点	命令数	竞争线程数	测试描述
1	≤ 2000	0	单线程进程管理和调度操作
2			多线程进程管理和调度操作
3-10	≤ 500	≤ 30	全部操作包括异常处理