Software Engineering and Applications 软件工程与应用, 2025, 14(6), 1291-1300
Published Online December 2025 in Hans. https://www.hanspub.org/journal/sea
https://doi.org/10.12677/sea.2025.146114

Hans汉斯

连续簇内存文件系统的设计与实现

徐明毅

武汉大学水利水电学院，湖北 武汉

收稿日期：2025年10月17日；录用日期：2025年12月15日；发布日期：2025年12月23日

摘 要

在Windows环境下采用C++语言设计并实现了连续簇内存文件系统CCFS，采用链表结构管理连续簇，能够灵活适应内存空间高度可调的各类设备。针对单次写入数据量小导致降速的情况，设计了输入缓冲区，使得写入吞吐率提高30%以上，这对于进行高频少量数据读写的应用程序来说，加速效果十分明显。该文件系统在完整的存储空间内实现了数据存储和文件管理的所有功能，不仅可用于常规内存，还适用于非易失性内存，考虑磨损均衡后，还可用于写入次数有限的固态硬盘，具有广泛的适用性。

关键词

内存文件系统，C++，缓冲区，数据存储

Design and Implementation of the Continuous Cluster Memory File System

Mingyi Xu

School of Water Resources and Hydropower Engineering, Wuhan University, Wuhan Hubei

Received: October 17, 2025; accepted: December 15, 2025; published: December 23, 2025

Abstract

The continuous cluster memory file system (CCFS) was designed and implemented using the C++ language in the Windows environment. It employs a linked list structure to manage continuous clusters, flexibly adapting to various devices with highly adjustable memory spaces. In response to the issue of reduced performance due to small data volume in a single write, an input buffer was also designed, resulting in a write throughput improvement of over 30%. The acceleration effect is very obvious for applications involving high-frequency and small-scale data read/write operations. This

文章引用: 徐明毅. 连续簇内存文件系统的设计与实现[J]. 软件工程与应用, 2025, 14(6): 1291-1300.
DOI: 10.12677/sea.2025.146114

file system fulfills all data storage and file management functions within the full storage space, making it suitable not only for conventional memory but also for non-volatile memory. With wear leveling considerations, it can be applied to solid-state drives with limited write cycles, demonstrating broad applicability.

Keywords

Memory File System, C++, Buffer, Data Storage

1. 引言

2025 年被称为 AI 智能体元年，即智能大模型不再局限于回答问题，而是能自动调用不同的应用程序以完成工作任务，如总结搜索结果、整理调研报告、修改程序代码等。因此对应用程序的操作将不仅限于人类，也包括 AI 智能体，应用程序成为 AI 智能体在数字世界的操作工具，一部分程序正加以改造以适应这一变化。现今人们主要的关注点还在于 AI 本身的运行速度，还未明显体会到智能体调用工具软件的快慢影响到智能体推理的整体效率。随着 AI 应用的普及，应用程序的启动速度和运行时数据交换效率将成为影响用户体验的关键因素，是亟待加强的短板。在计算机系统中，文件系统是数据存储的基础设施，对上层应用提供存储和访问数据的基本服务。如果能够提供文件系统的纯内存处理方案，就能加速所有在其上运行的、需要处理数据的应用程序，从而提升整个系统的性能及用户体验。

在 Linux 系统中，RAMFS 是一种高效但简单的内存文件系统，它是利用现有缓存机制实现的轻量级解决方案，但会消耗固定内存，对于需要更可控内存使用的场景，TMPFS 通常是更好的选择，它增加了将数据写入虚拟交换空间的能力，并已发展出功能更加全面的内存文件系统[1]-[3]。在 Windows 系统中，通常使用 RAMDISK 等工具将内存空间虚拟为磁盘，可以加快文件的读写，但由于所用文件系统并非专门为内存介质设计，不可避免存在一些非必要的中间转换过程，效率上仍逊于专门的内存文件系统，虽可在第三方工具支持下引入 RAMFS，但功能较不完善。

为解决在 AI 智能体操纵下的应用程序的慢速数据交换问题，使用高速内存文件系统将是十分有利的。本文在 Windows 环境下用 C++语言设计并实现了简单高效的连续簇内存文件系统(Continuous Cluster Memory File System, CCFS)，为临时文件读取、高速数据交换等场景创造了便利的条件。该文件系统在完整的空间内实现了文件管理的各项功能，因此不仅适用于常规内存，也适用于非易失性内存[4] [5]，如果考虑磨损均衡，还可用于写入次数受限的固态硬盘[6] [7]，具有广阔的应用前景。

2. 空间管理

2.1. 连续簇管理

内存文件系统是指直接在内存中建立文件系统[8]，由于内存可随机访问，内存文件系统的文件组织方式不必采用固定数据块大小的组织方式，可以采用更加紧凑、更加高效的数据组织结构；与关系数据库和键值(KEY-VALUE)数据库相比，内存文件系统的存储空间具有文件语义，传统应用程序可以使用文件访问接口管理数据，这更容易满足普遍需求。

为简化内存空间管理，可先分配一大段连续内存，然后将之分为相同大小的簇，簇的尺寸可选用常

用的 512 B 或 4 KB 等。总的簇数用 4 字节整数来统计，因此管理的内存空间在簇大小为 512 B 时可达 2 T，在簇大小为 4 KB 时可达 16 T，基本能够满足个人机和中小型服务器的内存空间管理需求。

连续的簇可合并为块，这样可以避免对每个簇都进行标记，节省存储空间。用链表结构来管理块，使当前块可连接到下一块。在块的首簇头部设置链表，需要包含两个必要数据：当前块的簇总数和下一块的首簇编号，可定义如下的结构体：

```
struct headLump {
    uint next;      // 下一块的初始簇编号
    uint used;      // 当前块使用的簇总数
};
```

其中 uint 为 unsigned int 的别名，即无符号 4 字节整数。当下一块的初始簇编号为 0 时，表示链表结束。这样，连续簇文件系统不用如 FAT 文件系统一样设置专门的文件分配表，不需占用固定的初始空间，管理手段更为紧凑，但缺点是空间占用情况不能一目了然，需要通过链表整理出来，效率较低，因此在使用时要尽量避免空间的碎片化，否则链表过长会增加统计时间。

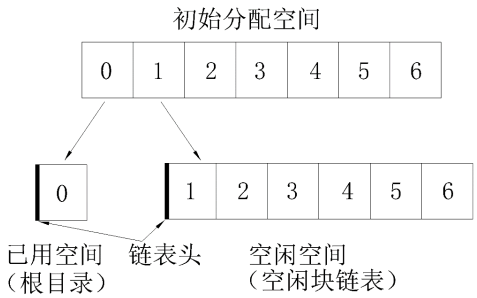

Figure 1. Initialization of the file system
图 1. 文件系统空间的初始化

连续簇文件系统的初始化过程不需要操作分配表，因此非常简单和快速。如图 1 所示，只需要将第一个簇(编号为 0 的簇)作为根目录的保存空间，链表头设置为[0,1]；将编号为 1 的簇作为空闲空间的起始位置，链表头为[0,总簇数−1]，空闲空间的初始簇编号为 1，后续所需的目录空间和文件数据空间都可从该空闲块中分配。

2.2. 空间分配

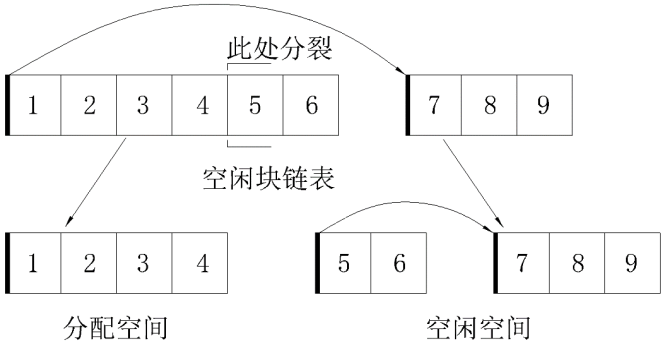

Figure 2. Allocating space from the idle block linked list
图 2. 从空闲块链表分配空间

当连续簇文件系统其他模块需要一段空间时，从空闲块链表开始，逐个统计链表上每个空闲块的空间大小，直到取出满足给定大小的空间为止。若发现取下某个空闲块后，得到的总大小超过要求的大小，则将该空闲块进行分割，并将分割剩下的部分重新挂回到当前空闲块，取得的满足要求的空闲块链表则标记为待用块链表。分配空间的具体过程如图 2 所示，若需分配 n 个簇，首先查找空闲块链表中是否有足够的簇，如果没有，说明没有可用空闲空间了，返回 0 值。否则检查刚好能满足要求的链表中的某个空闲块。如果该空闲块刚好分配完，则将后续块释放回空闲空间，而前面部分则标记为待用块，返回初始簇的编号。若没有分配完，则从中分裂出需要使用的簇，将后续的簇打包为空闲块返回空闲空间。

2.3. 空间释放

分配的空间如果不再使用，需要释放变为空闲空间。有两种方式将释放空间合并到空闲块链表，即放在头部或者尾部。对于不考虑磨损的内存介质来说，放在头部通常更快，因为下次查询的路径可能较短；但对于考虑磨损的介质来说，放在尾部则更好，因为下次可以优先使用较新的部分，避免刚使用的空间再次磨损[6] [7]。

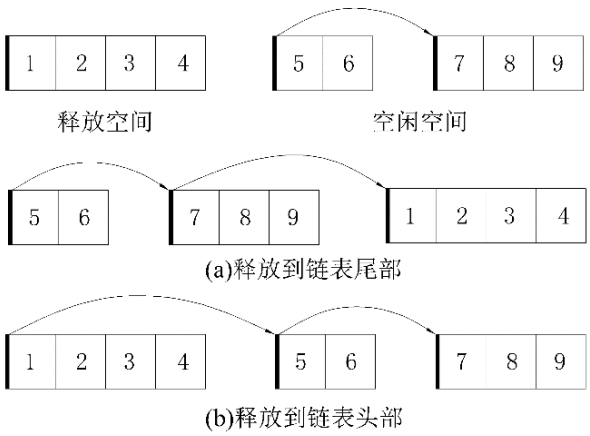

Figure 3. Two methods to release space
图 3. 空间释放的两种方式

在释放一个块链表时，首先查询该链表包含的总簇数，便于登记空闲空间的增加值。然后获取空闲空间的链表头，根据返回到空闲空间的头部或尾部方式进行操作。默认返回到链表尾，先查询出空闲块链表的尾，将下一块的编号指向释放块链表的头即可，如图 3(a)所示；若是返回到空闲块的头部，则是查询待释放块链表，将尾部的下一块的编号指向空闲块链表的头，将空闲块链表重新设置为从释放块链表开始，如图 3(b)所示。另外，在空间的释放过程中，如果释放块能够与其前面或者后面的块进行合并，则尽量将其合并，这样既能充分使用空间，又能加快后续的分配操作，可以减少空间碎片。

3. 目录和文件结构

3.1. 目录结构

目录的管理是一个结点表数组(可称为 NODE 表或 UNIT 表)，数组的每一项为 8 字节大小，其中 4 字节为结点属性(表示类别为目录或文件等)，当前只使用最低的一个字节，其它字节可用于权限等用途；另一个 4 字节则指向文件数据或下级目录所在的首簇编号，若表示目录时，该编号为 0 只能指向根目录，其它子目录则不为 0，若为文件时，该编号为 0 则表示该位置没有使用，新增的文件项可重用该位置。结

点的数据结构如下：

```
struct UNIT {
    uint startfat;     //文件或目录的首簇编号
    uint property;   //结点属性
};
```

数组每一项在结点表中的索引，就是该文件或子目录对应的索引号。结点表的总大小决定了目录能够创建的文件个数，若空间大小为 8 MB，该目录便能创建 100 万个文件。结点表的大小并不固定，可在初始化时进行指定，或者在结点表增加时，自动分配新空间，使该表的大小不断增长。在默认情况下，初始只需分配一簇或两簇的空间，在扣除链表头和目录管理信息后，所剩的空间就是初始能容纳的结点数，按簇尺寸为 512 B 考虑，也能容纳几十项，满足通常需求。

目录除了结点表外，还有其它的管理信息，其中一些数据是方便目录的查找和增删，并不是必须的，但添加后能提高目录的操作速度。目录的结构体 dirTable 如下所示：

```
struct dirTable {
    uint  maxSize;   //能够容纳的最大目录项总数
    uint  maxHead;  //首块能容纳的最大目录项总数
    uint  currNum;   //当前目录项索引位置
    uint  dirSize;     //使用的目录项总数
    char dirName[MAX_NAME/2];       //目录名称
};
```

其中 MAX_NAME 为文件名的最大字符数，此处定义为 256，目录名称一般更短，其最大字符数设定为文件名的一半，即 128。在分配目录管理的存储空间后，首簇的链表头后面就是目录头，然后紧接着才是结点表。目录信息管理可能只有一个簇，也可能是包含多个块的链表。目录中的结点数可任意调节，但上限为 4G 个，即不超过总簇数。为建立父目录和子目录之间的联系，在子目录的结点表的第一项(索引为 0)记录父目录的位置，而根目录的父目录就是它自身。

3.2. 文件结构

普通文件用于存放用户写入的数据，主要分为两部分：文件信息区和数据区，在分配空间后，链表头之后紧接文件信息，然后才是文件数据。文件信息有文件名、文件尺寸等，如结构体 slideFCB 所示：

```
struct slideFCB {
    uint  usefat;          //当前块的首簇编号
    size_t    offset;       //从簇首开始的偏移地址
    uint  endfat;          //终到块的首簇编号
    uint  remain;          //终到块的剩余空间大小
    size_t    flsize;       //文件尺寸
    char flname[MAX_NAME];   //文件名称
};
```

目录和文件的这种管理方式可类比于列车的组合和分解，即对于大小相等、连续编号的空车厢(连续排列的簇空间)，取出某个车厢将之改造为列车头(用于管理文件或目录)，然后再挂接数量不等的车厢。列车最短时可能只有一个车头，长时则根据需要可不断挂接。如果车厢的编号是连续的，挂接点只需要一个，如果车厢编号不连续，则需要多个挂接点，极端情况下，每个车厢都需要挂接点，这时的有效利

用空间最少。在列车不再使用时可解列，释放的车厢又挂接到空车厢编组上，留待下次使用。这样的空间管理方法既有规整性(为簇大小的整数倍)，又能灵活调整长度，适应目录项目和文件尺寸的变化。

4. 文件主要操作

4.1. 数据写入

对文件的操作已实现打开(OpenFile)、关闭(CloseFile)、写入(Write)、读出(Slide)、克隆(Clone)、从外部磁盘载入(LoadFile)，保存到外部磁盘(Store)等，这些操作不仅满足了在内存中对文件快速读写的要求，还具备了持久化到外部存储空间的功能。

传统文件系统在写入新数据时，一般先将数据缓存到内存中，在某个时候才将数据刷出到外部存储介质中，而内存文件系统可绕开缓存，直接将新数据写入到内存中，减少缓存转储的多余步骤。由于数据块的大小是任意的，在大块数据写入时，不使用缓存是明智的，而对于小块数据的高频写入，由于写入过程的复杂判断，因而可以先将小块数据累积在缓存中，再一次性写入，这样反而可能节省时间。

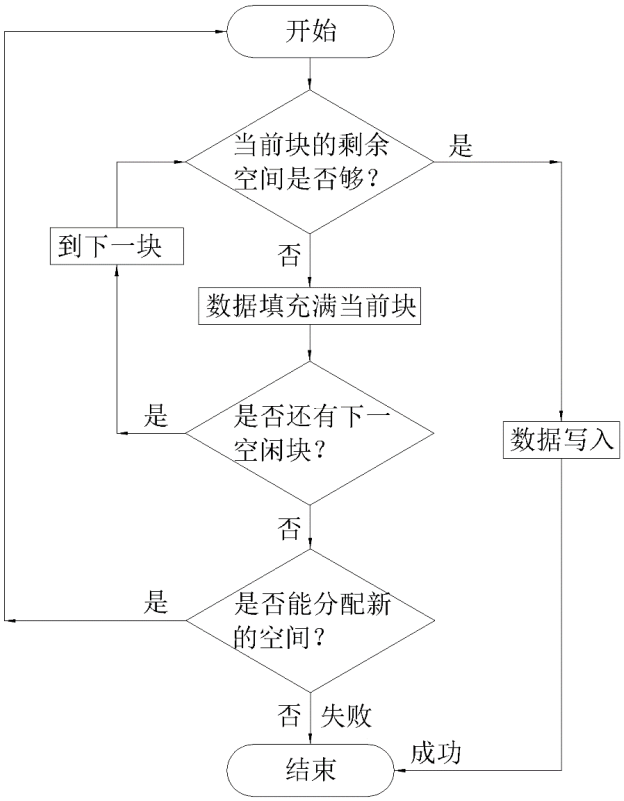

Figure 4. Process of writing file data
图 4. 文件数据写入流程

写入文件数据的基本流程如图 4 所示，首先判断当前块的剩余空间是否足够容纳新的数据，如果足够，则只需要将数据写入到连续的簇中即可。如果空间不足，则需要先填满当前块的剩余空间，然后为未写入的数据分配空间后再继续写入。为减少分配空间的操作次数，当前采用的策略是：当需要的空间小于 8 MB 时，增长空间的大小为已有文件大小，即空间加倍增长，否则每次增长 8 MB，但如果剩余待写入数据大于 8 MB，则按实际大小增加。在分配较大空间时，可能会得到离散的若干块，比如需要分配

8 MB 块时，可能会得到一个连续的 6 MB 块和另一个连续的 2 MB 块。不论何种情况，分配新空间后，就将多个新块链接到文件的数据空间，然后重新从文件的末尾开始将剩余的数据写入。可见文件数据的写入可能是分批次的，写入函数可以递归调用。

4.2. 数据读取

在读取数据时，需要给定读取缓冲区的位置及最大长度，如果能读取到最大长度，说明文件可能还有数据，如果读取长度小于最大长度，则表示文件数据已读完。

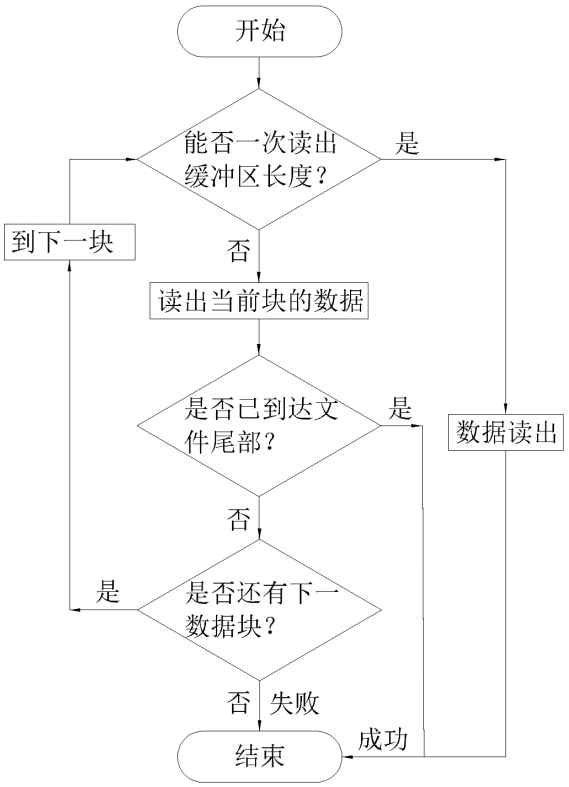

Figure 5. Process of reading file data
图 5. 文件数据读取流程

数据读取的基本流程如图 5 所示，当读取缓冲区较小时，在文件的当前块可一次填满缓冲区后成功退出；否则缓冲区不能一次填满，则先将当前块内的所有数据读出，如果已到达文件尾，读取长度小于缓冲区长度，成功退出，否则改变待读取的缓冲区长度，文件转移到下一数据块继续读取。如果既没有到达文件尾又没有下一数据块，则表明文件记录出现混乱，出错退出。

4.3. 性能测试

在初步实现以上的连续簇内存文件系统后，在 Windows 11 24H2 家庭中文版环境下进行了性能测试，其中处理器为 Intel i7 10700 F，内存为 DDR4 2400 MT/s，程序编译为 32 位模式。测试只针对简单的情况，即将一定长度的整数数组内容写入文件，如以下代码：

```
for (int i = 0; i<icycle; i++)
    pFMan->Write(pFile, (char*)intarray, length);
```

Table 1. Test of data write speed
表 1. 数据写入速度测试

数据量(Byte)	重复次数	耗用时间(s)	写入速度(GB/s)
40	10^7	0.166	2.41
400	10^6	0.127	3.15
4K	10^5	0.066	6.06
40K	10^4	0.068	5.88
400K	10^3	0.058	6.90

设定写入数据量相同，但单次的写入量不同，如单次写入 400 B 时，使之重复 100 万次，总的写入量为 400 MB，根据耗时可测试出平均的写入速度。改变单次写入的数据量，可以看出写入吞吐率的变化。

写入性能测试结果如表 1 所示，可以看到，在一次写入数据量为 4 KB 时，速度达到 6.06 GB/s，而在单次写入 40 B 数据时，速度只有 2.41 GB/s，差距较大。这说明在小数据高频写入时，由于文件写入操作较复杂以及函数调用开销，致使写入吞吐率下降。在单次写入量超过 4 KB 后，写入操作的吞吐率增长缓慢，在测试中甚至还略有下降。初步分析，一个可能的原因是内存调度影响。数据量较小时，待写入的数据基本都在高速缓存中，调取速度最快，而在数据量较大时，有一部分会保存到低速缓存中，缓存交换可能导致速度有所下降。另外也可能与 CPU 调度有关，在现有多核心系统中，每个核心运行的频率会动态变化，超线程技术让某个核心可同时运行两个任务，这使得程序每次运行的环境并不完全相同，测试数据也会相应发生波动。

总的来看，该内存文件系统的写入速度还较为满意，比普通磁盘的写入速度提高至少一个数量级，但对比当前主流的固态硬盘，则优势不太明显，还需继续优化。对于随机读写来说，内存文件系统能够利用内存的快速寻址功能，突发反应时间比固态硬盘大致快三个数量级，这对于数据读写量较小的应用程序来说提速较为明显。另外，从表中数据还可以看出，如果提供合适的输入缓冲区，高频小数据的写入速度将有可能提高。

5. 写入缓冲区设计

5.1. 缓冲区管理

由于写入文件的数据并非规整长度，而是大小不一，这些数据小到几字节，大到以 MB/GB 为单位。在单次小数据量写入时，由于写入过程的复杂判断，导致写入的效率不高，可考虑通过增加缓冲区的方式来加速。这样将复杂的文件写入操作转化为快捷的缓冲区写入操作，当数据累积到缓冲区将满时，再一次性写入文件，减少了耗时的文件写入操作。

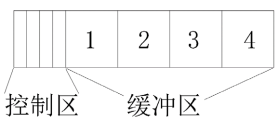

Figure 6. Layout of writing buffers
图 6. 写入缓冲区的布局

考虑用文件属性的最低字节用于输入缓冲区指示，由于值为 0 时为目录，为 1 时为无缓冲区的文件，则其它数值为文件对应的输入缓冲区编号，因此最多同时有 254 个文件有输入缓冲区。设计的缓冲区布

局如图 6 所示，分为两部分，前面部分为控制区，后面部分为等长的多个缓冲区。在控制区中记录每个缓冲区的状态，包括使用标识，当前指针位置和剩余长度，如以下结构体所示：

```
struct bufUnit {
    char* ptr;      //缓冲区当前空闲位置指针
    uint  res;      //缓冲区的剩余长度
    uint  use;      //缓冲区的使用标识
};
```

对缓冲区的操作有查询当前空闲缓冲区(AskBuffer)、缓冲区已用长度(UsedBuffer)、缓冲区剩余长度(LengBuffer)、释放缓冲区(FreeBuffer)、向缓冲区复制数据(BufferCopy)等。这些操作都很简单，不需要复杂判断，满足将少量数据快速保存到缓冲区的要求。其中，复制数据到缓冲区的步骤是，先从控制区找到缓冲区 iBuf 对应的当前位置指针，将数据复制到该位置，然后当前指针向后移动，剩余的空闲长度也相应减少，如以下代码：

```
Memcpy (bufCtrl [iBuf]. ptr, content, len); //复制数据
bufCtrl[iBuf].ptr += len;  //位置指针增加
bufCtrl[iBuf].res -= len;  //剩余长度减少
```

5.2. 有缓冲区的数据写入性能测试

定义输入缓冲区后，数据写入文件的过程如图 7 所示，首先根据文件属性判断是否有缓冲区，如果没有，则将数据直接写入文件；如果有缓冲区，再判断缓冲区剩余空间是否足够，足够则写入缓冲区；如果不够，则先将缓冲区的内容写入到文件，缓冲区恢复到初始状态，然后根据写入数据量判断，如果较多，比如大于 1/16 缓冲区长度，则不必在缓冲区中转，直接写入文件，否则暂存到缓冲区。

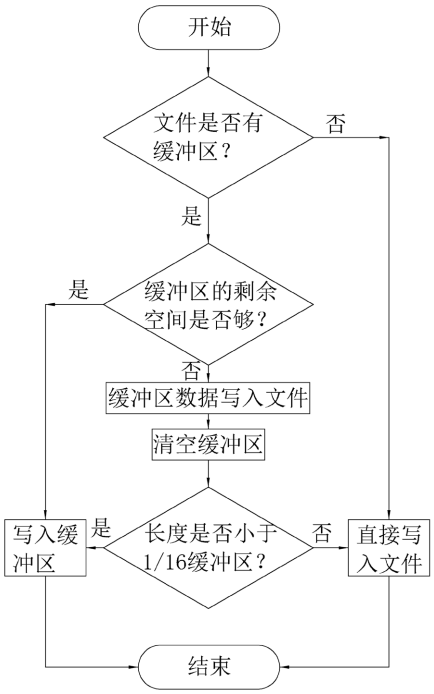

Figure 7. Data writing process with input buffer
图 7. 有输入缓冲区的数据写入流程

将输入缓冲区的大小取为 64 KB，再次测试内存文件的写入速度，结果如表 2 所示。可以看到，对于单次为 40 B 的小数据，写入平均速度从 2.41 GB/s 增长到 3.17 GB/s，速度提高 31.5%；单次为 400 B 的数据写入，速度从 3.15 GB/s 增长到 4.35 GB/s，提高 38.1%，都有明显的改善。对于 4 KB 大小的单次写入，速度基本上没有变化，而对于 40 KB、400 KB 大小的写入，速度比无缓冲区情况还可能略有降低，这是因为此时的处理判断更复杂。可以认为，在单次写入数据量与缓冲区大小差别不大时，并没有节省写入文件的操作次数，反而对数据进行了多余的转储过程，此时设置缓冲区导致降速。因此在实际使用时，缓冲区长度一般要超过单次写入量的十倍以上，否则没有明显的加速效果。

Table 2. Test of data write speed with a buffer size of 64 KB
表 2. 缓冲区为 64 KB 的数据写入速度测试

数据量(Byte)	重复次数	耗用时间(s)	写入速度(GB/s)
40	10^7	0.126	3.17
400	10^6	0.092	4.35
4 K	10^5	0.066	6.06
40 K	10^4	0.077	5.19
400 K	10^3	0.061	6.56

6. 结语

在 Windows 环境下原生设计的连续簇内存文件系统(CCFS)能够充分利用内存的高速传输带宽，具有延迟低、读写快的明显优点，能够克服常规应用程序的数据读写瓶颈，特别适合于 AI 智能体对应用程序的快速调用操作。连续簇文件系统不使用常规的文件分配表，而是采用链表形式管理连续簇，能充分利用内存空间，灵活适应空间大小高度可调的各类设备，空间的初始化操作也非常简单。针对小规模数据频繁写入导致降速的情况，通过设计输入缓冲区，可将较为耗时的文件写入操作转换为轻便的缓冲区保存操作，使得吞吐率提高至少 30%以上，这对于加速随机小数据的频繁写入十分有利。由于连续簇文件系统是在完整的存储空间内实现了数据存储和文件管理的所有功能，因此不仅适合于常规的易失性内存，也适用于非易失性内存，在考虑磨损均衡后，还可以用于写入次数有限的固态硬盘等介质。当然，在现有的初步设计中，还未能充分考虑各类存储设备的特点，普适性虽好但针对性不足，还需要结合具体设备进一步优化，在实际使用中不断提高该文件系统的适用性和健壮性。

参考文献

[1] 沙行勉, 吴挺, 诸葛晴凤, 等. 面向同驻虚拟机的高效共享内存文件系统[J]. 计算机学报, 2019, 42(4): 800-819.
[2] 周嘉铭. 基于 RDMA 和 NVM 内存文件系统一致性机制的研究与实现[D]: [硕士学位论文]. 上海: 华东师范大学, 2021.
[3] 茅志祥. 多版本内存文件系统中存储映射 I/O 机制的设计与实现[D]: [硕士学位论文]. 上海: 上海交通大学, 2018.
[4] 钟展和. 细粒度数据管理的非易失内存文件系统研究[D]: [硕士学位论文]. 武汉: 华中科技大学, 2023.
[5] 丁骆昌祺. 持久内存文件系统中崩溃一致性漏洞检测机制研究[D]: [硕士学位论文]. 武汉: 华中科技大学, 2024.
[6] 聂顺. 持久化内存文件系统磨损感知的多粒度分配机制优化研究[D]: [硕士学位论文]. 重庆: 重庆大学, 2021.
[7] 马乔. 磨损感知的持久性内存文件系统设计与实现[D]: [硕士学位论文]. 成都: 电子科技大学, 2020.
[8] 许春聪, 刘钊, 文海雄, 等. 基于内存的数据存储技术研究[J]. 科技与创新, 2018(2): 19-21.