第二十三章　实验 5: 文件系统与 Shell

23.1　简介

本实验将体现 ChCore 的微内核架构，实现一种基于**索引节点（index node，inode）**的内存文件系统：**临时文件系统（temporary file system，tmpfs）**、一个系统服务：用户态文件系统服务以及一个应用程序：**外壳（shell）**。

本实验分为两个部分：在第一部分中，实现基于 tmpfs 的用户态文件系统服务，ChCore 的文件系统服务能够提供一些文件的基本操作，如创建、删除、读写文件等。在第二部分中，实现 shell 来与用户交互，需要实现一些基本命令，以及可以以 UNIX shell 为参照拓展其他功能。

23.1.1　评分

实验 5 中，代码部分的总成绩为 80 分。可使用如下命令检查当前得分：

```
oslab$ make grade
…
Score: 80/80
```

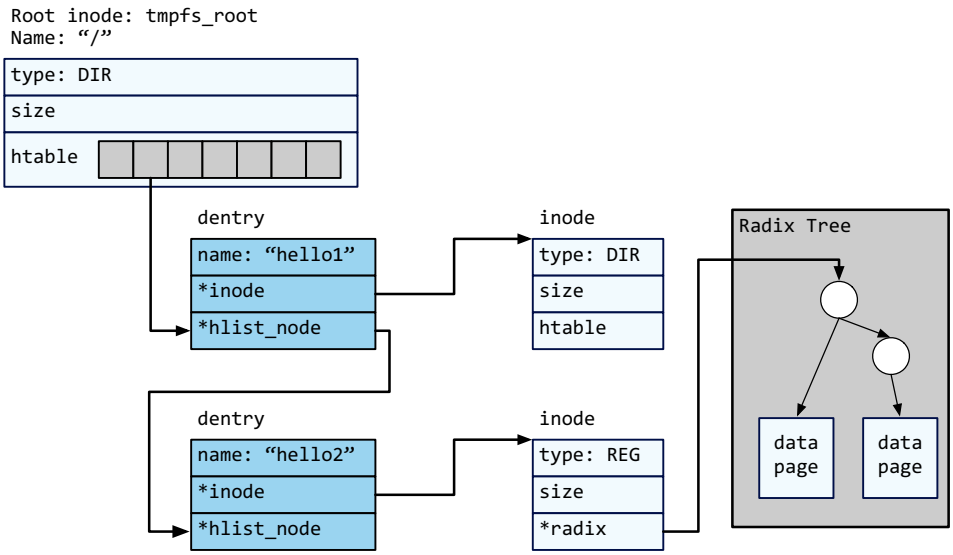

图 23.1: ChCore 中的 tmpfs 结构

23.2 第一部分：文件系统

23.2.1 tmpfs

tmpfs 是基于 inode 的内存文件系统，即使用内存空间作为文件系统的存储空间，并且将存储空间分为 inode 区域和数据区域。每个文件都有一个 inode，该 inode 保存有关该文件的一些元数据，例如文件大小，以及该文件数据块的位置。数据区域存储文件数据块，tmpfs 中的文件数据块由一系列分散的内存页组成。文件分为常规文件和目录文件：常规文件存储数据本身，目录文件存储从文件名到**目录项（dicrectory entry，dentry）**的哈希表映射。

ChCore 的 tmpfs 在`user/tmpfs/tmpfs.{h,c}`中定义，如图 23.1所示，具有以下几个重要的数据结构：

- **inode**。每个 inode 对应一个文件，记录文件类型（`type`）（常规`REG`/目录`DIR`）和文件大小（`size`）。接口`new_dir`和`new_reg`用于创建这两种类型的 inode。
- **目录**。目录 inode 存储一个指向哈希表`htable`的指针，该哈希表从文件名的哈希值映射到 dentry。哈希表中的每个哈希桶都存储 dentry 的链接表，并且通过`hlist_node`指针链接。其中，`tmpfs_root`表示根目录（`/`）的 inode。

- **常规文件**。常规文件的数据块以基数树的形式组织，树的根节点指针存在常规文件 inode 中。该树的叶节点是大小为PAGE_SIZE的数据块（即内存页），通过内存页的顺序编号搜索。

23.2.2 文件操作

tmpfs 的基本功能在user/tmpfs/tmpfs.c中定义，实现过程中请参考头文件user/tmpfs/tmpfs.h中已有的函数以及相关的代码注释。在实现基本的文件操作之前，需要先实现一些修改文件结构的辅助函数：tfs_mknode和tfs_namex。tfs_mknode函数用于创建文件时，在父目录下创建目录 inode 或常规文件 inode。tfs_namex函数用于遍历文件系统结构以查找文件。

练习 1

实现tfs_mknode和tfs_namex。

文件读写是文件系统的基本功能，tmpfs 的读写操作是指内存中的数据读入或写到内存缓冲区。tfs_file_read和tfs_file_write两个函数分别用于以一定偏移量读取和写入一段长度的数据，并且返回实际的读写字节长度（读取不能超过文件大小）。

练习 2

实现tfs_file_read和tfs_file_write。**提示：**由于数据块的大小为PAGE_SIZE，因此读写可能会牵涉到多个页面。读取不能超过文件大小，而写入可能会增加文件大小（也可能需要创建新的数据块）。

在实验 3 和实验 4 中，make user命令将用户态程序编译为 ELF 文件（ramdisk/*.bin或ramdisk/*.srv），并使用 cpio 将这些文件连接为一个后缀名为 cpio 的文件。cpio 的每个成员文件都包含一个头（在user/lib/cpio.h中定义），后面是文件内容。cpio 的末尾是一个名为TRAILER !!的空文件表示终止。[1] 在之前的实验中，我们将ramdisk.cpio的内容直接复制到没有文件结构的连续内存中。现在，tmpfs 可以通过tfs_load_image加载ramdisk.cpio中的内容。

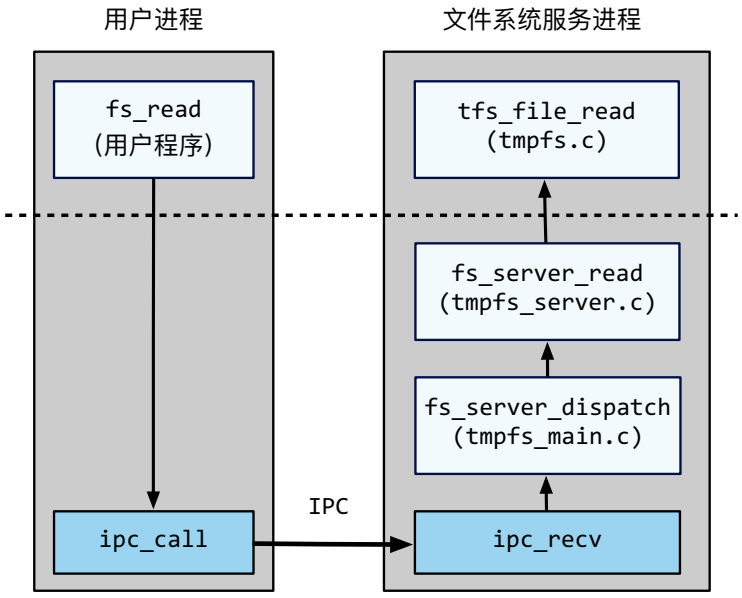

图 23.2: ChCore 中文件读取流程

练习 3

实现`tfs_load_image`函数。需要通过之前实现的函数进行目录和文件的创建，以及数据的读写。

23.2.3 文件系统服务

如图 23.2所示，为了保证进程隔离，用户无法在文件系统服务中直接调用函数，因此用户程序通过 IPC 给文件系统服务发送请求的方式来使用文件系统。以文件读取为例，虚线上方是用户程序和文件系统的主要处理逻辑，虚线下方是文件系统服务的 IPC 处理机制。用户程序通过调用由用户编写`fs_read`，使用`ipc_call`将消息发送到文件系统服务进程。

文件系统服务主函数在`user/tmpfs/tmpfs_main.c`中定义。其主要逻辑是轮询 IPC 请求，通过`fs_dispatch`将请求分派到适当的处理函数。在处理函数解析、处理后，通过 IPC 将结果发送回去。处理函数在`user/tmpfs/tmpfs_server.{h,c}`中定义。在示例中，`fs_dispatch`将请求分派到`fs_server_read`，在解析请求后，调用`tfs_file_read`来实际执行读取的文件。

练习 4

实现在user/tmpfs/tmpfs_main.c中的fs_dispatch和在user/tmpfs/tmpfs_server.c中的所有fs_server_xxx处理函数，附表给出了fs_dispatch的分发规则。

请求 ID	函数	功能
FS_REQ_SCAN	fs_server_scan	列出文件或目录
FS_REQ_MKDIR	fs_server_mkdir	创建目录
FS_REQ_CREAT	fs_server_creat	创建文件
FS_REQ_RMDIR	fs_server_rmdir	移除目录
FS_REQ_UNLINK	fs_server_unlink	移除文件
FS_REQ_WRITE	fs_server_write	写文件
FS_REQ_READ	fs_server_read	读文件
FS_REQ_GET_SIZE	fs_server_get_size	获取文件大小

正确完成练习 4 后，输入make grade可以获得 30 分。

```
LAB5: test >>>>>>>>>>>>>>>>>
./scripts/grade-lab5
runing fs_test: (9.4s)
  fs server init: OK
  fs scan: OK
  fs write read: OK
  fs deepfile: OK
  ……
Score: 30/80
```

23.3 第二部分：Shell

本实验的第二部分是实现用户态 shell，该 shell在user/apps/init_main.c中定义。它具有两个主要组件：输入解析器和 tmpfs 的客户端。在这一部分中，我们仅在user/apps/init_main.c和user/apps/init.c中提供了代码框架。

输入以下命令来编译并运行 shell：

```
chcore$ make user
…… 
copy user/*.bin to ramdisk.
copy user/*.srv to ramdisk.
add fs_test files
1944 blocks
succeed in building ramdisk.
chcore$ make build bin=init
…… 
before ninja
[63/63] Linking C executable kernel.img
after ninja
chcore$ make qemu
…… 
init loads cpio image.
Booting fs...
info_page_addr: 0x200000
[tmpfs] register server value = 0
fs is UP.

$
```

由于尚未实现在user/apps/init.c中定义的readline()，目前看不到提示符$后输入的内容。

练习 5

实现在user/apps/init_main.c中定义的getchar()，以每次从标准输入中获取字符，以及在user/apps/init.c中定义的readline。该函数将按下回车键之前的输入内容存入内存缓冲区并将其标准输出。可以使用在user/lib/syscall.{h,c}中的 I/O 函数。

练习 6

实现在user/apps/init.c中定义的bultin_cmd()以支持 shell 中的内置命令，例如ls，ls [dir]，cd [dir]，echo [string]，cat [filename]。

练习 7

实现在user/apps/init.c中定义的run_cmd，以通过输入文件名来运行可执行文件，并且支持按 tab 自动补全。可以使用在user/lib/目录下proc.h中定义的launch_process_with_pmos_caps和在liblauncher.c中定义的readelf_from_fs。

练习 8

实现top以显示线程信息。top的格式如下：（还可以在每行的末尾添加其他信息）：

```
$ top
Current CPU 0
===== CPU 0 =====
Thread ffffff0020600228 Type: ROOT      State: TS_RUNNING
  CPU 0   AFF 0   Budget 2
Thread ffffff00206007a8 Type: USER      State: TS_READY
  CPU 0   AFF 0   Budget 2
===== CPU 1 =====
Thread ffffff00001ecfa8 Type: IDLE      State: TS_RUNNING
  CPU 1   AFF 1   Budget 2
===== CPU 2 =====
Thread ffffff00001ed000 Type: IDLE      State: TS_RUNNING
  CPU 2   AFF 2   Budget 2
===== CPU 3 =====
Thread ffffff00001ed058 Type: IDLE      State: TS_RUNNING
  CPU 3   AFF 3   Budget 2
```

完成练习 8 后，将获得全部 80 分。

```
LAB5: test >>>>>>>>>>>>>>>>>
./scripts/grade-lab5
runing fs_test: (9.1s)
  fs server init: OK
  fs scan: OK
  fs write read: OK
  fs deepfile: OK
runing init_test: (9.3s)
  readline: OK
  echo: OK
  ls: OK
  ls dicrectory: OK
```

```
cat: OK
auto complement: OK
run executable: OK
top: OK
```

参考文献

[1] PTC MKS Toolkit. cpio: format of cpio archives. https://www.mkssoftware.com/docs/man4/cpio.4.asp.

实验 *5*：扫码反馈