第七章. 文件系统（lab5）(v0.1)

7.1. 实验目标

内核是操作系统最基本的部分。它是为众多应用程序提供对计算机硬件的安全访问的一部分软件，这种访问是有限的，并且内核决定一个程序在什么时候对某部分硬件操作多长时间。内核分类为微内核和单内核模式。

微内核（Microkernel kernel）：在微内核中，大部分内核都作为单独的进程在特权状态下运行，他们通过消息传递进行通讯。在典型情况下，每个概念模块都有一个进程。因此，假如在设计中有一个系统调用模块，那么就必然有一个相应的进程来接收系统调用，并和能够执行系统调用的其他进程（或模块）通讯以完成所需任务。在这些设计中，微内核部分经常只是个消息转发站：当系统调用模块要给文档系统模块发送消息时，消息直接通过内核转发。这种方式有助于实现模块间的隔离。（某些时候，模块也能够直接给其他模块传递消息。）最根本的思想还是要保持微内核尽量小，这样只需要把微内核本身进行移植就能够完成将整个内核移植到新的平台上。其他模块都只依赖于微内核或其他模块，并不直接直接依赖硬件。

微内核设计的一个长处是在不影响系统其他部分的情况下，用更高效的实现代替现有文档系统模块的工作将会更加容易。我们甚至能够在系统运行时将研发出的新系统模块或需要替换现有模块的模块直接而且迅速的加入系统。另外一个长处是无需的模块将不会被加载到内存中，因此微内核就能够更有效的利用内存。

单内核（Monolithic kernel）：单内核是个很大的进程。他的内部又能够被分为若干模块（或是层次或其他）。但是在运行的时候，他是个单独的二进制大映象。其模块间的通讯是通过直接调用其他模块中的函数实现的，而不是消息传递。

MIT 这次实验是在前面实验的基础上，实验一个简单的基于磁盘的微内核方式的文件系统，文件系统本身作为一个用户进程运行，可以支持对层次目录结构中的文件进行 create,read,write 和 delete 操作；并在此基础上实现类似 Unix 的 exec 功能的 spawn 功能，从磁盘文件系统装入并运行一个可执行文件。其他进程通过 IPC 请求来访问文件系统服务。该实验分为 2 部分：文件系统服务器、客户端和 Spawn 函数。前者为其他进程提供访问文件系统的服务，客户端进程通过发送 IPC 请求完成文件的操作；后者从文件系统加载一个进程运行。

本实验中的函数主要集中在 fs 和 lib 目录下的文件中：

fs/fs.c

fs/serv.c

lib/file.c

lib/spawn.c

在实验过程中，为了保证大家自己写的程序段的正确性，JOS 系统安排了一些检查函数，在开启文件系统的时候进行检查，与实验 2 中检查函数类似，如果这些检查函数发现大家写的程序不符合实验原来的设想的话（主要是一堆的 assert），就会提前 panic 掉。写完文件系

统相关函数可以让利用检查程序进行检查，以确保走的是正确的路。

主要的检查函数有：

check_write_block(void);该函数通过先打乱 superblock 然后重新读回，检查函数 write_block()是否能正常工作。

fs_test(); 该函数用于测试我们写的操作文件的函数是否正确，包括 file_open();file_get_block();file_flush();file_set_size();file_truncate();file_rewrite();等。

7.2. 背景知识

到目前为止，我们还只是单用户的操作系统，因此文件系统目前不支持权限和属主等属性，目前也不支持硬链接，符号链接，时间戳和设备文件。对于磁盘文件系统的结构我们简单介绍一下。

许多类 UNIX 操作系统将磁盘分成两个域：inode 区域和 data 区域。inode 用来保存文件的状态属性，以及所指向数据块的指针。data 区域中包含了 data 块，这里存放文件的内容和目录的元信息（包含的文件名以及指向文件 i 节点的指针）。如果文件系统中的多个目录都指向文件的 inode 节点，则称此文件为硬件链接的。在 JOS 系统中，由于不支持硬连接，用不到 inode，只要把文件的元数据存放在所属的目录里就可以了。

基于磁盘的文件系统结构主要由 4 部分组成：

1. 扇区（Sector）和块（Block）
2. 超级块（Super block）
3. 块位图（Block bitmap）
4. 文件元数据（file meta-data）

扇区（Sector）：

Sector 是磁盘执行读写操作的单位，一般是 512 字节，扇区大小是一个磁盘硬件的属性。

块（Block）：

Block 是文件系统分配和使用磁盘空间的单位，是 Sector 的整数倍（本实验里是 4096 个字节，与页的大小相等）。块是一个操作系统使用磁盘的属性。

超级块（Super block）：

一个特定的物理位置（一般是磁盘的第一块或者最后一块）。包含描述文件系统的元数据：block 的大小、磁盘大小、根目录位置、文件系统挂载的时间、上次进行磁盘检查的时间等等。大多数真正的文件系统维护多个超级块，通过复制分散到不同的磁盘分区上，用来防止超级块损坏带来的问题。

在本实验中，inc/fs.h 中的 **Super 结构**定义了磁盘布局。JOS 系统中只有一个超级块为 block1，block0 存放 boot sector 和磁盘分区表。

块位图（Block bitmap）：

文件按系统必须管理磁盘上的存储块以保证给定的磁盘块在一个时刻仅用于一种目的。使用块位图来**管理空闲磁盘块**，容易存储，并可以节省磁盘空间（空间换时间，带来的在内存中扫描位示图表的时间代价与之后进行的磁盘 I/O 相比是微不足道的）。为了建立一个空闲块位图，在磁盘上保留足够大的连续空间，为每个磁盘块设置一个位。

在 JOS 中，**block2 开始为块位图**，涵盖范围包括所有的磁盘块，也包括：block0，block1 和位图块本身。

文件元数据（file meta-data）：

在 JOS 系统中，文件的元数据由 **inc/fs.h** 中的结构 **File** 来描述：包括：文件名、大小、

类型和指向文件所包含磁盘块的指针等；有些域仅仅在内存中才有意义，所以每次从磁盘读 file 结构到内存去时，都要把这些域给清空（在本实验中，仅有 f_ref）。

File 结构可以表示文件或者目录，两者的区别在于 type 域（后面详细介绍 File 结构）。文件系统不会解析代表文件的 file 结构的数据块的内容；但会解析代表目录的 file 结构数据块内容来获得其所包含的文件和子目录的信息。

File 结构还保存了**文件头** 10（NDIRECT）个块（**直接磁盘块**）的全部内容，大于 10 个块的文件要间接寻址（**间接磁盘块**）。

直接磁盘块：

在 File 中的块数组存储了一个文件的前 10（NDIRECT）个块的块号，称为直接磁盘块。不超过 10*4096=40KB 的小文件，其文件的所有块的块号都可以直接放在 File 结构中。

间接磁盘块：

对于大于 40KB 的文件，需要另外分配一个磁盘块，叫做间接磁盘块，可以保存 4096/4=1024 个磁盘块号。在 JOS 系统中，为了使登记目录简单化，不使用间接磁盘块中的前 10 个块号，所以 JOS 文件系统支持最大 1024 个块也就是 4M 大小的文件。在实际的文件系统中，为了支持更大的文件，通常都会使用两个或三个间接磁盘块。

通过以上讨论我们知道了文件系统的结构，在 JOS 中给出的文件系统的磁盘块图形表示如下所示：

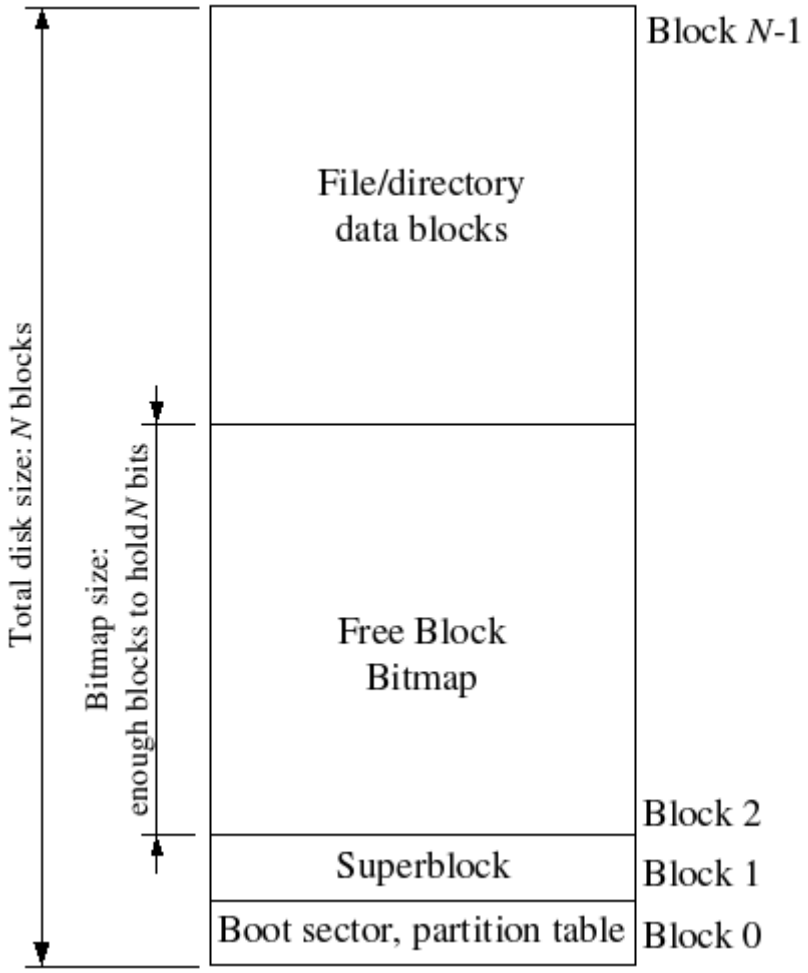

图 7-1. 磁盘块

由于本次实验用到了磁盘文件，我们需要在 bochs 配置文件.bochsrc 中增加一个磁盘镜

像，其格式为：diskd: file="./obj/fs/fs.img", cyl=128, heads=8, spt=8。

7.3. 文件系统

在这个实验中，文件系统实际上包括服务器和用户进程的访问。对于前一个部分，主要讨论文件系统提供的磁盘访问操作、磁盘块缓存、块位图操作、文件操作等，而后一部分主要是讨论其他进程如何利用 IPC 访问文件系统。

7.3.1 文件系统的实现

本次实验新增了一个 fs 目录，添加了许多文件系统相关方面的文件，我们需要浏览这些文件对 JOS 系统提供的关于文件系统的数据结构以及操作有个整体的认识，方便以后实现。

首先，我们来看一下构成文件的元数据的结构，其在 **inc/fs.h** 中定义的：

```
struct File {
    char f_name[MAXNAMELEN];   // filename
    off_t f_size;              // file size in bytes
    uint32_t f_type;           // file type

    // Block pointers.
    // A block is allocated iff its value is != 0.
    uint32_t f_direct[NDIRECT];// direct blocks
    uint32_t f_indirect;       // indirect block

    // Points to the directory in which this file lives.
    // Meaningful only in memory; the value on disk can be garbage.
    // dir_lookup() sets the value when required.
    struct File *f_dir;

    // Pad out to 256 bytes; must do arithmetic in case we're compiling
    // fsformat on a 64-bit machine.
    uint8_t f_pad[256 - MAXNAMELEN - 8 - 4*NDIRECT - 4 - sizeof(struct File*)];
} __attribute__((packed));     // required only on some 64-bit machines
```

其文件结构可以用图 7-2 表示：

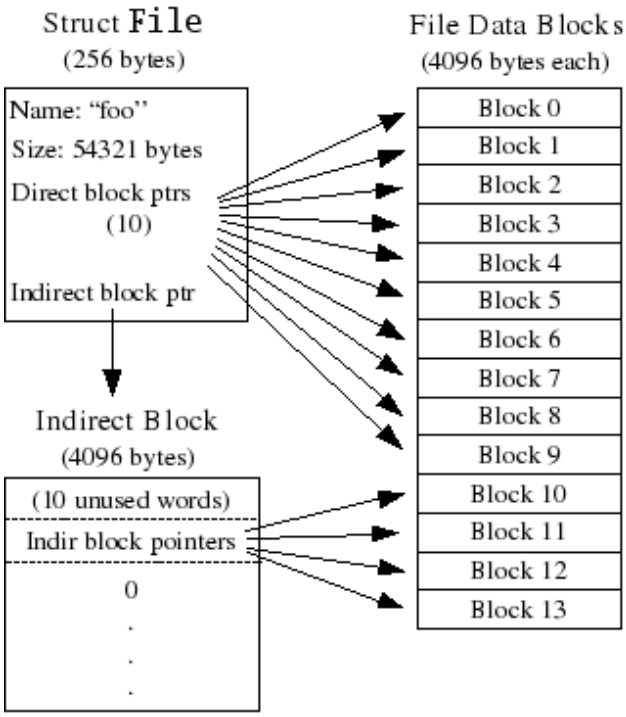

图 7-2. File 结构表示

文件对象表示进程已打开的文件。该结构存在 7 个成员，包括：文件名、大小、类型和指向文件所包含磁盘块的指针等等。

接下来，系统定义了超级块，其定义也是在 inc/fs.c 中：

```
struct Super {
    uint32_t s_magic;        // Magic number: FS_MAGIC
    uint32_t s_nblocks;      // Total number of blocks on disk
    struct File s_root;      // Root directory node
};
```

它包含了 3 个成员：一个文件系统的魔数，一个包含磁盘中 block 的总个数以及目录登录点。

同时，系统还定义了文件描述符句柄，用来支持我们用户进程使用库函数访问文件系统，主要有一下几个数据结构：

```
struct Dev {
    int dev_id;
    char *dev_name;
    ssize_t (*dev_read)(struct Fd *fd, void *buf, size_t len, off_t offset);
    ssize_t (*dev_write)(struct Fd *fd, const void *buf, size_t len, off_t offset);
    int (*dev_close)(struct Fd *fd);
    int (*dev_stat)(struct Fd *fd, struct Stat *stat);
    int (*dev_seek)(struct Fd *fd, off_t pos);
    int (*dev_trunc)(struct Fd *fd, off_t length);
};
```

对应一个块设备，并为此设备提供常用的读、写、查看状态信息等操作，其指针函数对

应 lib/file.c 文件中的函数，我们将在后面详细介绍。

```
struct FdFile {
    int id;
    struct File file;
};
```

此结构是将一个文件句柄对应一个文件对象。

```
struct Fd {
    int fd_dev_id;
    off_t fd_offset;
    int fd_omode;
    union {
        // File server files
        struct FdFile fd_file;
    };
};
```

文件句柄，其对应一个文件，保存文件的操作模式，比如只读、可写等。

```
struct Stat {
    char st_name[MAXNAMELEN];
    off_t st_size;
    int st_isdir;
    struct Dev *st_dev;
};
```

保存文件的状态信息，文件名、文件大小、文件类型以及文件所属设备。

另外，针对这些数据结构，系统还定义了一些宏以及函数来对这些结构进行操作：

#define **INDEX2FD(i)** ((struct Fd*) (FDTABLE + (i)*PGSIZE))

返回索引节点 i 所对应的文件句柄

#define **INDEX2DATA(i)** ((char*) (FILEBASE + (i)*PTSIZE))

返回文件描述符索引节点 i 所对应的数据

```
// Bottom of file data area
#define FILEBASE    0xD0000000
// Bottom of file descriptor area
#define FDTABLE     (FILEBASE - PTSIZE)
```

char* **fd2data**(struct Fd *fd);

求出文件描述符句柄 fd 所指向的数据，此函数使用宏 **INDEX2DATA(i)**来实现。

int **fd2num**(struct Fd *fd);

由函数名可知，此函数将文件描述符句柄 fd 转换为文件描述符索引节点，功能与

INDEX2FD(i)刚好相反，根据其距 **FDTABLE** 的偏移来计算。

int **fd_close**(struct Fd *fd, bool must_exist);

释放一个文件描述符句柄，此函数通过关闭 fd 指向的相关文件并取消 fd 所映射的页来完成。

int **dev_lookup**(int devid, struct Dev **dev_store);

在已有设备信息中查找与 devid 匹配的设备。

系统中还有很多已经写好的关于文件系统的操作函数，在以后的实验中会慢慢提到，这里就不一一详述了。

7.3.1.1 磁盘的访问

文件系统服务器需要能够访问磁盘，传统的微内核模式的操作系统往往在系统调用中增加一个 IDE 磁盘驱动器来允许文件系统去访问磁盘，而 JOS 系统中，我们将 IDE 磁盘驱动(**fs/ide.c 中有磁盘驱动**)作为文件系统进程用户的一部分。为了简化，系统采用轮询方式，而不是磁盘中断：查询 EFLAGS 寄存器中的 IOPL 位，此处可从代码 **fs/serv.c** 中了解。

由于我们需要访问的所有的 IDE 磁盘寄存器信息都是放在 x86 处理器的 I/O 空间，而不是映射到内存空间，如果要访问磁盘，我们需要将 I/O 特权级赋给文件系统进程。在实验 1 的报告 2.2.3 节中，我们知道，x86 处理器使用标志寄存器 **EFLAGS** 中的 **IOPL** 位（I/O 特权级位）来决定保护模式中的代码是否拥有访问 I/O 的权限。要使文件系统进程可以访问磁盘，只需要将其标志寄存器 EFLAGS 中的 IOPL 置位即可。

我们的任务就是要修改内核进程初始化函数：env.c 中的函数 **env_alloc()**，使其仅给进程 1（文件系统进程）I/O 权限。

7.3.1.2 块缓存

在我们的文件系统中，我们借助虚拟内存来实现磁盘块缓存。由于一个进程可以拥有 4G 的虚拟空间，我们文件系统进程可以处理的磁盘大小最多为 3G。在 JOS 系统中，我们将这 3G 空间固定在文件系统进程的地址空间，从 0x1000 0000(**DISKMAP**)到 0xD000 0000(**DISKMAP+DISKMAX**)作为缓存区，当磁盘块读入内存时，用来映射相关的页。比如，磁盘块 0 无论何时都映射到地址 0x1000 0000 处，磁盘块 1 映射到地址 0x1000 1000 处。其结构如下图所示：

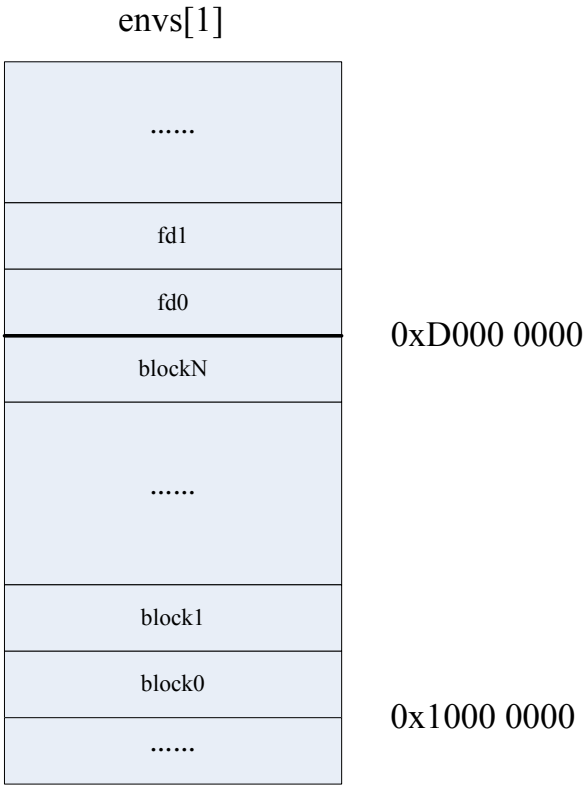

图 7-3. 块缓存

由于我们的文件系统进程的虚拟地址空间独立于系统中其他进程的虚拟地址空间，而其惟一要做的事情就是提供文件的访问，因此以这种方法保留文件系统大部分的地址空间是很有效的。

在实现块缓存之前，我们需要了解一下系统给出的一些函数：

char* **diskaddr**(uint32_t blockno)

//返回磁盘块所对应的虚拟地址

bool **block_is_mapped**(uint32_t blockno)

//检查磁盘块 blockno 是否已经映射到内存

int **map_block**(uint32_t blockno)

//检查磁盘块是否已经映射，否则分配一页保存磁盘块数据

int **ide_read**(uint32_t secno, void *dst, size_t nsecs)

//读取磁盘块 secno 中 nsecs 个扇区数据到地址 dst 处

int **ide_write**(uint32_t secno, const void *src, size_t nsecs)

//将 src 地址处的 nsecs 个扇区的数据写到磁盘块 secno 中

接下来我们就需要实现文件文件系统中读写磁盘块的函数了，**fs/fs.c** 中的 read_block 和 write_block 函数：

static int **read_block**(uint32_t blockno, char **blk)

将磁盘块 blockno 数据读入到内存中。首先要检查磁盘块 blockno 是否已经在内存，如

果没有，则需要分配一页并使用 **ide_read** 将其读入到 blockno 所对应的虚拟地址处。

void **write_block**(uint32_t blockno)

将磁盘块中的当前内容写到磁盘中。判断磁盘块中的数据是否需要写，我们只需要参看其页表中的 PTE_D 位即可，当块中的数据改变时，此位由处理器直接设置。

注意：在写回磁盘后，需要将 PTE_D 位清 0。

7.3.1.2 块位图

在 JOS 系统中，使用块位图来**管理空闲磁盘块。**在 JOS 文件系统进程的初始化中，**fs_init** 首先是调用 **read_super** 来读和检查文件系统的超级块。其次 **fs_init** 调用 **read_bitmap** 来读和检查磁盘的位示图。考虑到系统的速度和简单，我们总是将整个块位图都读取到内存中。在本实验中，需要实现 **read_bitmap** 来检查磁盘中的位示图是否正确。

uint32_t ***bitmap**; // bitmap blocks mapped in memory

//系统中用一个全局变量指向映射到内存中的位示图块，方便以后查找

bool **block_is_free**(uint32_t blockno)

//检查磁盘块 blockno 的位图是否空闲

void **read_bitmap**(void)

该函数检查文件系统中保留的所有的磁盘块的块位图，将其读到内存，标记为可用，并使全局变量 bitmap 指向块位图的首地址。

注意：位示图在磁盘中是第二个磁盘块，其中用 **block_is_free** 检验所有的位示图是否可用。

int **alloc_block_num**(void)

以 **block_is_free** 来查找一个空闲的位示图，分配之后执行 **write_block**，将位示图块的状态进行刷新，以保持文件系统的一致性。

7.3.1.3 文件操作

在操作系统中，用户接触到的一般都是文件对象，其表示进程已打开的文件，进程直接处理的都是文件，而不是超级快、索引节点。因此文件操作在文件对象中是非常重要的。在 JOS 系统中，提供一些最基本的文件操作，包括：解析和管理 File 结构，分配或寻找指定文件的指定块，扫描并管理目录文件的表项，从根目录遍历文件系统，得到文件的绝对路径等。首先我们要阅读 fs/fs.c 中所有的代码，系统提供了大部分的代码，我们只对本实验中需要用到的做简单说明：

static int **walk_path**(const char *path, struct File **pdir, struct File **pf, char *lastelem)

//从根目录开始查找，解析路径 path，保存其路径指向的文件

int **file_map_block**(struct File *f, uint32_t filebno, uint32_t *diskbno, bool alloc)

//将文件中的第 filebno 磁盘块映射到文件系统中的 diskbno 块

通过系统中提供的相关函数，我们实现文件操作就很简单了，重点是要理解系统给出的那些函数。接下来我们需要实现的就是高层的文件操作，其函数说明如下：

int **file_open**(const char *path, struct File **pf)

利用 **walk_path** 打开路径 path 中的文件，获得其文件句柄。

int **file_get_block**(struct File *f, uint32_t filebno, char **blk)

利用 **file_map_block** 和 **read_block** 从文件 f 中获得 filebno 磁盘块数据，使 blk 指向其首址。

static void **file_truncate_blocks**(struct File *f, off_t newsize)

将文件大小截为新的大小 newsize，并释放掉文件中没有使用的磁盘块，包括间接磁盘块。

void **file_flush**(struct File *f)

将文件对象 f 中的内容刷新到磁盘上，只对文件中标记为 dirty 的数据操作。

7.3.1.4 服务器与用户进程通信

在完成了上面的实验后，我们注意到，JOS 系统并没有提供文件系统中基本的操作：read 和 write。这是因为 JOS 中的文件系统是通过内核中的 IPC 将文件系统的页映射到客户进程，然后客户进程直接读写。因此我们就要使文件系统进程（服务器）允许其他进程（用户）使用文件系统。

JOS 中，我们采用 C/S 结构的文件系统访问，使得文件系统可以被其他进程访问。每一个客户进程都使用 **ipc_send** 发送消息到服务器，即提出访问文件系统的请求，然后使用 **ipc_recv** 等待响应，完成文件操作。

首先，我们来看一下客户发给文件系统请求的一些数据结构，其结构是在 inc/fs.h 中定义的：

```
struct Fsreq_open {
    char req_path[MAXPATHLEN];
    int req_omode;
};
//客户进程向文件系统服务器发送打开文件请求，以 req_omode 模式打开文件
struct Fsreq_map {
    int req_fileid;
    off_t req_offset;
};
//客户进程向文件系统服务器发送映射文件请求，大小为 req_offset
struct Fsreq_set_size {
    int req_fileid;
    off_t req_size;
};
//客户进程向文件系统服务器发送设置文件大小请求，大小设置为 req_size
struct Fsreq_close {
    int req_fileid;
};
//客户进程向文件系统服务器发送关闭文件请求
struct Fsreq_dirty {
    int req_fileid;
```

```
    off_t req_offset;
};
//客户进程向文件系统服务器发送数据修改请求，偏移量为 req_offset
struct Fsreq_remove {
    char req_path[MAXPATHLEN];
};
//客户进程向文件系统服务器发送删除文件请求，文件路径为 req_path
```

下面，我们分别对 C/S 结构文件系统的客户端和服务器端进行讨论。

对于 *client stubs*，系统已经在 lib/fsipc.c 中实现了，使用了系统中 IPC 的协议，当客户进程需要访问文件系统时，使用 *client stubs* 执行相应的请求。我们需要了解 *client stubs* 请求文件系统的服务。

static int **fsipc**(unsigned type, void *fsreq, void *dstva, int *perm)

所有的 *client stubs* 都是通过此函数向服务器发送一个请求，并等待应答。此函数使用 **ipc_send** 向服务器发送请求，**ipc_recv** 接收服务器的消息，其中 type 为请求类型，如 FSREQ_OPEN 、FSREQ_CLOSE 等，fsreq 为包含请求数据的页，如数据结构 **Fsreq_open**、**Fsreq_close** 等，dstva 为接收文件系统数据的虚拟地址，perm 为接收页表的页属性。

注意：在所有的请求都发送给进程 1，因为我们的文件系统是作为用户进程 1 来运行的，这样通过简单的 IPC 实现了其他进程访问文件系统的服务。

int **fsipc_open**(const char *path, int omode, struct Fd *fd)

向文件服务器发送打开文件请求，包括文件的路径，打开方式，返回文件句柄 fd。

int **fsipc_map**(int fileid, off_t offset, void *dstva)

向文件服务器发送块映射请求，将文件 fileid 中要求的块映射到文件 dstva 地址处。

int **fsipc_set_size**(int fileid, off_t size)

向文件服务器发送设置文件大小请求

int **fsipc_close**(int fileid)

向文件服务器发送关闭文件请求

int **fsipc_dirty**(int fileid, off_t offset)

向文件服务器发送文件中的块修改请求

int **fsipc_remove**(const char *path)

向文件服务器发送删除文件请求

int **fsipc_sync**(void)

向文件服务器发送打开文件同步请求，将修改的文件写到磁盘上。

对于 *server stubs*，需要我们在 fs/serv.c 中实现，这些 stubs 从客户进程接收 IPC 请求，解析这些命令，然后使用 fs/fs.c 中的文件访问功能完成这些服务。系统中定义了一个用于打开文件的数据结构：

```
struct OpenFile {
    uint32_t o_fileid;   // file id
    struct File *o_file; // mapped descriptor for open file
    int o_mode;          // open mode
    struct Fd *o_fd;     // Fd page
};
```

//用来保存打开文件的一些信息，如文件描述符，文件打开模式等。

server stubs 具体函数说明如下：

void **serve_open**(envid_t envid, struct Fsreq_open *rq)

此函数打开客户进程要求路径下的文件，并发送消息给客户进程，将文件页表映射到客户进程中。

注意：在文件系统服务器中为打开的文件保存了 3 个数据结构，一个是 struct File，其对应磁盘数据映射到内存中的文件，这部分内存对文件服务器来说是私有的，见图 3。第二个是 struct Fd，每一个打开的文件都有一个 Fd，相当于一个文件描述符，它用来与其他进程共享打开的文件，因此，服务器需要同步文件系统进程和客户进程中的 Fd。第三个是 struct OpenFile，其主要作用是链接上面两个结构，文件系统服务器使用一个 OpenFile 链表来记录所有文件系统中打开的文件。

void **serve_set_size**(envid_t envid, struct Fsreq_set_size *rq)

对于 envid 进程，首先查找打开的文件，然后利用 **file_set_size** 来设置文件新的大小，并给客户进程一个反馈。

void **serve_map**(envid_t envid, struct Fsreq_map *rq)

将客户进程要求的文件系统中磁盘块映射到客户进程地址空间中。

void **serve_close**(envid_t envid, struct Fsreq_close *rq)

关闭客户进程要求的文件。

void **serve_remove**(envid_t envid, struct Fsreq_remove *rq)

删除客户进程要求的文件。

void **serve_dirty**(envid_t envid, struct Fsreq_dirty *rq)

将客户进程要求的文件设置为已修改，以备后面写入磁盘。

所有的文件系统服务都是通过函数 **serve** 来分发的，首先其接收来自客户进程的请求，根据不同请求为客户进程提供不同的服务，然后再给客户进程反馈。我们以客户进程打开一个文件为例，如下图所示：

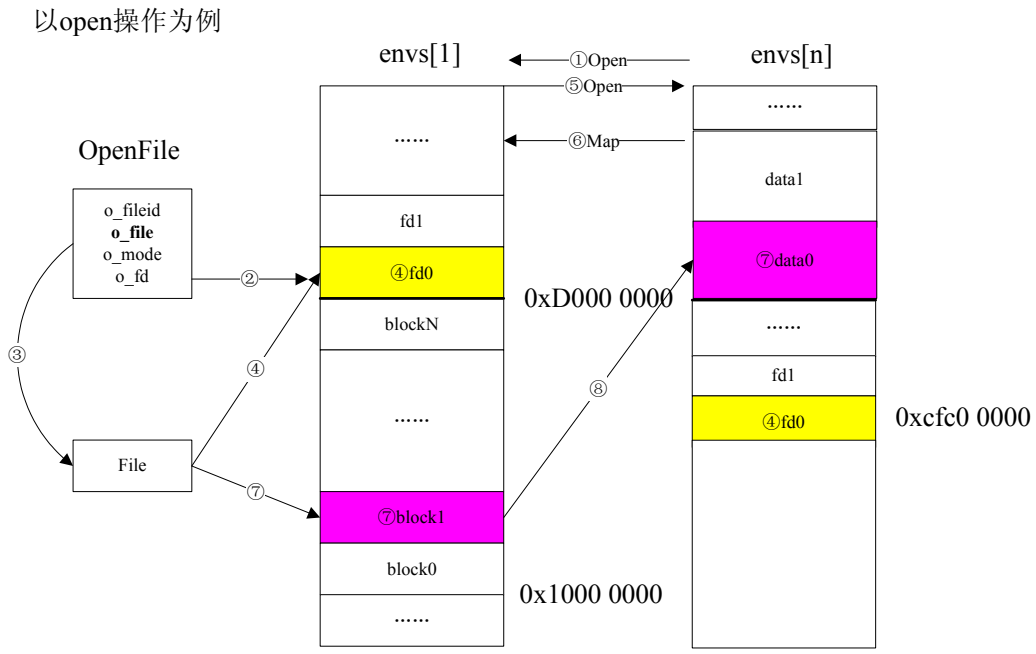

图 7-4. 客户端打开文件

客户进程通过 FSIPC 发送打开文件请求，文件系统服务器寻找一个可用的文件句柄，并将文件映射到文件句柄，然后将句柄映射到客户进程；客户进程将数据 data0 映射到文件系统的磁盘块，发送 map 命令，文件系统中将文件句柄对应的磁盘块映射到客户进程中，完成文件打开操作。

7.3.2 客户进程访问文件系统

通过以上的实验，我们可以使用 client-side stubs 直接访问文件系统服务器，进行文件操作，其基于文件系统的 IPC 接口，从以上的 open 操作我们可以看出，为了读写一个文件，进程需要先将自己的地址空间中文件的块映射到文件服务器进程的地址空间，然后读写文件，最后再向文件服务器发送关闭文件请求将文件写到磁盘上。这样实现起来比较繁琐。

接下来，在 JOS 系统中，我们通过 Client 端的文件描述符实现类 Unix 的文件操作，使客户进程可以直接读写文件。在 Client 端的地址空间中定义了两个虚拟地址，如下所示：

// Bottom of file data area

#define **FILEBASE** 0xD0000000

//进程文件数据的开始，如上图中 envs[n]中的 data0，由于在文件系统服务器中磁盘文件描述符限制大小最大为 4M，因此客户端中每个文件的映射区域大小限制在 4M。

// Bottom of file descriptor area

#define **FDTABLE** (FILEBASE - PTSIZE)

//进程文件描述符的开始，如上图中 envs[n]中的 fd0，其用一页来保存进程中使用的文件描述符（当前最多为 MAXFD32 个）。

JOS 系统已经提供了大部分的代码，我们只需要分析这些代码，然后完成以下几个函数：

int **fd_alloc**(struct Fd **fd_store);

在进程的文件描述符表中，从 0 到 MAXFD-1（**MAXFD** 为 32，即一个进程可以打开的最大文件个数）寻找一个可用的文件描述符。

int **fd_lookup**(int fdnum, struct Fd **fd_store);

查找文件描述符索引节点是否已经有相关页映射。

int **open**(const char *path, int mode)

首先用 **fd_alloc**()找到一个未用的文件描述符，然后向服务器发送一个 IPC 请求打开文件，并将文件系统中文件页表映射到客户端的地址空间中。

static int **file_close**(struct Fd *fd)

将文件服务器中的文件描述符清空，取消文件中映射的数据，然后关闭文件。

7.4. Spawn 函数

至此，我们建立了一个简单的文件系统，接下来我们就需要直接从文件系统加载镜像生成新的进程。JOS 系统中提供了 Spwan 函数，用来创建一个新的进程，从文件系统加载一个可执行文件到该进程中，启动这个进程来运行该可执行文件。

其函数说明如下：

int **spawn**(const char *prog, const char **argv)

Spawn 函数与 Unix 的 exec 功能类似，从磁盘文件系统装入并运行一个可执行文件，其中 prog 为文件系统中可执行文件的路径，argv 为可执行文件的参数。该函数首先打开文件并读取 ELF 文件头到内存，创建一个新的进程并使用函数 **init_stack** 在地址 USTACKTOP - PGSIZE 为其分配一个堆栈，将读取的 ELF 文件中程序的代码段、数据段和 bss 段加载到进程的地址空间，初始化子进程的寄存器状态，然后由系统进行调度。

注意：在加载 ELF 文件头到进程地址空间是一个难点，其分为两部分：当 ELF 文件中只包含代码段和只读的数据时，只需要将段内容读取映射到子进程的地址空间；对于可读写的段，仅仅将第一个 p_filesz 大小的段加载到进程空间，其他的段我们需要分配页来映射。

在 JOS 后面的实验中，扩展了 Spawn 函数，使其可以传递参数到新的进程。我们需要做两方面工作：一个是父进程，一个是子进程。

在父进程方面：spawn 在生成子进程时，需要设置子进程的堆栈，使其能够将参数传到子进程，父进程就需要调整内存的格式，如下图所示：

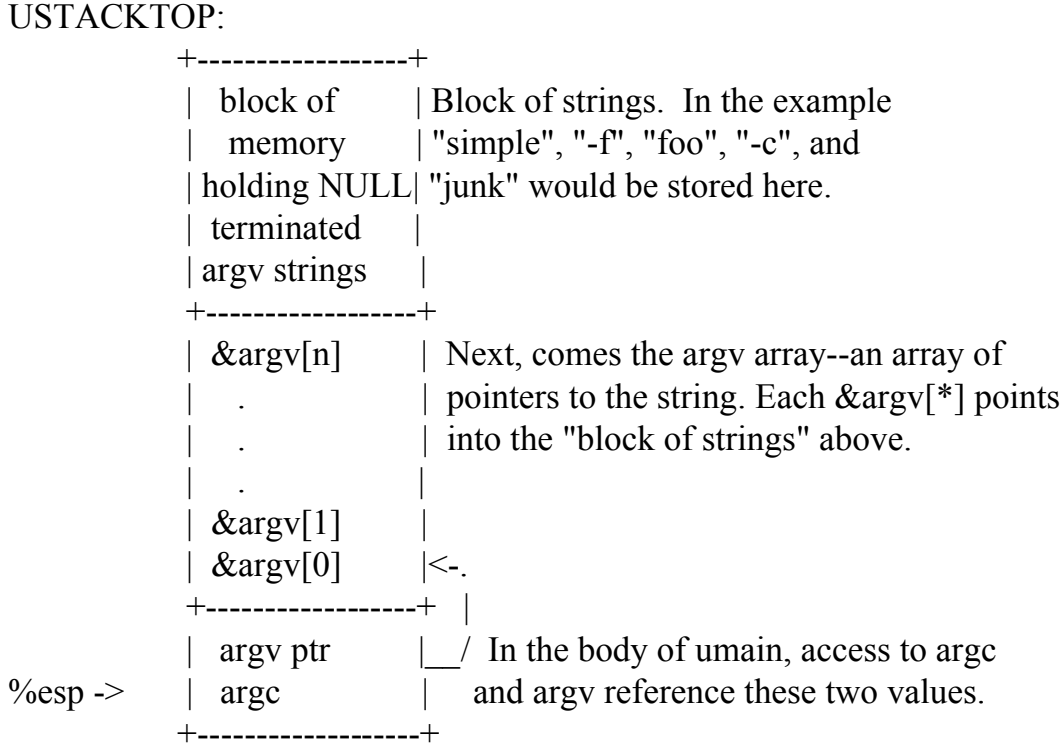

图 7-5. 父进程内存格式

在父进程使用 spawn 创建子进程时，子进程是共享父进程的所有寄存器信息的。同时将 USTACKTOP 处的内存全部映射到了子进程相应的内存处。在子进程方面：我们看到，父进程 Libmain(int argc, char **argv)在调用用户进程 umain(argc, argv);时，将其参数全部传给了用户进程，而由上图我们知道，父进程将内存信息重新映射到了子进程相应的内存处，因此，实现了父进程传递参数到子进程。

下面我们对这个函数进程说明：

#define **UTEMP2USTACK**(addr)((void*) (addr) + (USTACKTOP - PGSIZE) - UTEMP)

//该宏将地址 addr 由地址 UTEMP 转换成相应子进程的堆栈

static int **init_stack**(envid_t child, const char **argv, uintptr_t *init_esp)

设置子进程的堆栈，分配一临时页，并将其映射到父进程地址空间中固定地址处，然后将此页重新映射到子进程地址空间中 USTACKTOP 的末尾，设置新进程的堆栈指针指向设置好的堆栈。

注意：该函数中需要将父进程中传递的参数放置到一个合适的位置，由上图知 strings 在堆栈的最顶端，因此是放在 UTEMP + PGSIZE - string_size 之处，UTEMP 是临时页首地址。而且最后设置子进程时，需要将 umain(argc, argv);中的参数放置到子进程的堆栈首，可以由上图可知。

在完成了 Spawn 函数后，我们知道 JOS 在前面也实现了一个 LoadIcode 函数，其两者的区别如下：

1. Spawn 是在用户态下执行，行为类似 fork；loadicode 在内核态执行，是内核自己创建进程。
2. Spawn 从磁盘读程序；loadicode 读取内核内存映像中的程序。

3. Spawn 需要在初始化栈时处理程序运行参数的问题。

7.5.调试经验

Lab5 主要是完成微内核模式的文件系统，本实验中 JOS 给出了大量的代码，我们首先需要浏览系统中给出的代码，对给出的数据结构有清晰的认识，对系统中给出的注释要好好的理解，大部分内容系统都给出了解决方案，我们只需要利用系统中的函数完成工作。