哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验报告

开课学期：2020 年秋季

课程名称：操作系统

实验名称：简单文件系统的设计与实现

实验性质：设计型

实验时间：8 地点：T2210

学生班级：18 级 5 班

学生学号：180110518

学生姓名：胡智胜

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2020 年 12 月

1. 实验目的

本次实验要求我们自主设计并实现一个简单的仿 EXT2 文件系统，该文件系统的基础功能包含①创建文件或文件夹，②读取文件夹内容，③复制文件，④关闭系统，⑤系统再次启动时应能恢复为上次文件系统的状态；同时还需设计实现一个简单的 shell 方便控制该 EXT2 文件系统。在本次实验中，实现的文件系统有以下指令可供使用（具体请阅读 5.用户手册）：

1) mkdir 指令可以在指定路径上创建文件夹；
2) touch 指令可以在指定路径上创建文件；
3) ls 指令可以显示指定路径的文件夹中包含的所有文件；
4) cp 指令可以复制一个文件到另一个路径上；
5) cd 指令切换当前路径；
6) tee 指令可以编辑（非目录）文件内容；
7) cat 指令可以显示（非目录）文件内容；
8) shutdown 指令可以关闭并退出文件系统；

2. 实验环境

编程语言：C
开发环境：vscode
项目运行环境：ubuntu18.04
使用到的工具：CMake

3. 文件系统介绍

3.1 文件系统基本布局

在本实验中，我们设计的文件系统管理的磁盘大小为 4MB，文件系统的每个块“block”大小为 1KB，而虚拟磁盘接口所用的磁盘块“dblock[1]”大小是 512B，即每个系统块 block 其实由两个磁盘块 dblock 组成，这就使得在本文件系统中，第 block_no 号的系统块号其实是由第 2*block_no 号磁盘块和第 2*block_no+1 号磁盘块组成。

EXT2 文件系统的基本架构由超级块 super_block，inode 组和 data 组构成，其中 super_block 里包含了 inode 数组和 block 数组的位图 bitmap，这些 bitmap 记录了对应下标的 inode 或者 block 是否被占用，在实现的文件系统中关于 block 的位图记录的是整个 4MB 大小磁盘空间的系统块的位图（不是磁盘块 dblock），因此在 block 的位图中，首先被置为 1 就是超级块，inode 区所在的 block(因为这些 block 已被占用)。

在实现的代码中，super block 的大小是 656B，因此需要占用一个系统块，而在 super block 里的 inode 位图总共有 1024 位，而每个 inode 结构体的大小为 32B，则 inode 数组的大小为 32*1024

[1] 在本报告中，虚拟磁盘的磁盘块(大小为 512B)称为磁盘块，dblock；而简单 EXT2 文件系统的磁盘块（大小为 1024B）称为系统块，block。

=32KB，需要占用 32 个系统块，因此在最初的文件系统初始化时，我们会将文件系统的 block 位图的前 33 个块都标记为已被占用。同时也在 super block 中记录了 first_inode_block=1 和 first_data_block=33，来分别标记出第一个 inode 所在的块号和第一个 data block 的块号。因此，本文件系统的基本布局如下图：

3.2 super block

super block 的结构体如下：

```
struct super_block {    // 其中的所有的 block 都指的是 1024 大小的
    int32_t magic_num;                  // 幻数
    int32_t free_block_count;           // 空闲数据块数
    uint32_t first_inode_block;         // 第一个 inode 所在块的块号
    uint32_t first_data_block;          // 第一个 data block 块的块号
    uint32_t block_map[128];            // 数据块占用位图
    uint32_t inode_map[32];             // inode 占用位图
};
```

在 super block 中除了刚才已经介绍了的 block 位图(block_map)、inode 位图(inode_map)、inode 区域起始块号 first_inode_block、data 区域起始块号 first_data_block 之外，还有幻数 magic_num，空闲数据块数 free_block_count。其中幻数的作用就是用于判断文件系统是否已经存在或者文件系统是否完整，例如在启动文件系统时，会首先从虚拟磁盘中读取第 0 块（即 super block 所在块），如果 super block 中的幻数等于设定的幻数(0x112233)，就说明磁盘组的文件系统已存在，即不需要进行文件系统初始化。

3.3 文件的 inode 结构

inode 的结构体如下（其中 INODE_ADDR_LENGTH=6）：

```
struct inode {
    uint32_t size;                  // 文件大小
    uint8_t file_type;              // 文件类型（文件/文件夹）
    uint8_t link;                   // 连接数
    uint16_t inode_id;                  // inode 号
    uint32_t block_point[INODE_ADDR_LENGH];     // 数据块指针
};
```

在实现的文件系统中，对于每个文件采用的磁盘块分配策略是固定分配，因此每次新建一个文件便直接分配 6 个对应的 data block，所支持的最大文件大小为 6*系统块大小=6KB。在 inode 的 size 字段记录了每个文件的已写入数据的字节大小，file_type 有两类 T_DIR 代表是文件夹和 T_FILE 代表是单个文件，link 则是文件的连接数（在本简单文件系统中，每当一个子文件夹创

建时，其中都会包含“.”与“..”，其中“.”代表的就是当前文件夹，“..”则代表的是上一层文件夹，因此子文件夹中就会有一个指向外层文件夹的链接，因此外层文件夹的 link 数就该加 1），inode_id 也就是该 inode 在文件系统的 inode 数组中的下标，block_point 数组中装载了 6 个指向该文件数据所在数据块的块号。（注意，在本实验的示例代码中，inode 结构体并不含有 inode_id 字段，为了方便地在将 inode 从内存写回磁盘时能够快速找到 inode 的位置，特别地在原来 inode 地结构体中腾出了 16 字节的空间用来存储 inode_id 同时保持 inode 结构体的大小不变。因为在本实验中文件的类型就两种 T_DIR 和 T_FILE，因此存储文件类型完全不需要用 16 字节，8 字节就足够，同时在本简单文件系统中，文件的 link 数也不会超过 2^8 因此也可以只用 8 字节存储 link 数，而之所以用 16 字节的空间存放 inode_id 是因为本文件系统中最大 inode 数量为 1024，因此只用 8 位存储是不够的，而 16 位则足够了。）

3.4 文件夹的 inode 结构

文件夹的 inode 结构和文件的完全相同，只不过在文件夹的数据块中存储的数据都是一条一条的目录项 dir_item 组成，dir_item 结构体如下：

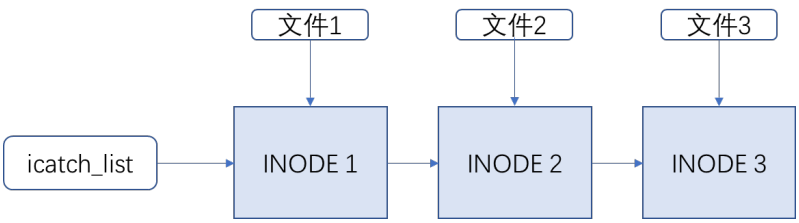
```
struct dir_item {
    uint32_t inode_id;          // 当前目录项表示的文件/目录的对应 inode
    uint16_t valid;             // 当前目录项是否有效
    uint8_t type;               // 当前目录项类型（文件/目录）
    char name[121];             // 目录项表示的文件/目录的文件名/目录名
};
```

在 dir_item 中，inode_id 代表 inode 在 inode 数组中的下标，valid 代表该条目录项的内容是否有效，type 代表该文件的类型是 T_DIR 还是 T_FILE，name 数组存放着该文件的文件名。每个 dir_item 结构体的大小为 128B，因此一个文件夹最多存放的目录项个数为 6KB/128B=48 条，而除去“.”和“..”，可供分配的目录项有 46 条。

3.5 inode 的内存管理

1. icatch_list

在文件系统的运行和使用中，我们需要在内存中分配 inode 的空间并从磁盘中读取 inode 或者在内存中进行 inode 的操作，这些 inode 的内存的管理在本实验的文件系统中由 icatch 负责。每当在文件系统中需要在内存中暂存一个文件的 inode 时，直接调用 malloc 分配 inode 大小的空间即可，但是为了及时地将这些通过 malloc 分配的内存空间回收，我们将这些空间存放在一个链表 icatch_list 上，



每次执行完一次一次用户操作，系统便会“刷新”一次 icatch，具体操作就是遍历一遍 icatch_list，对于每个内存中不是代表当前路径 inode(在下面会介绍当前路径 cwd)的 inode，将其

通过 iupdate 函数写回到磁盘上，接着将该 inode 内存空间释放掉，并将该 inode 从 icatch_list 中删除即可。

2. iupdate

iupdate 函数用于将内存中的 inode 写回到磁盘，每个 inode 中都存放了对应文件的 inode_id，因此可以计算出其所在的虚拟磁盘 dblock 的块号为：

dblock_no=first_inode_block*2+(inode_id / INODE_PER_DBLOCK)

上面的公式首先通过 first_inode_block*2 计算出 inode 区域起始的 dblock 的块号，再加上(inode_id / INODE_PER_DBLOCK)即可得到 inode 所在 dblock 的块号。

得到 inode 所在 dblock 的块号后便可通过虚拟磁盘接口，将该 dblock 数据读取到 512B 的缓冲区 buf 中，接着将 inode 中的数据写入到 buf 的对应的位置中，再把 buf 写回虚拟磁盘中即可。

3.6 文件的创建以及数据块的创建

1. inode 的创建

文件的创建主要就是创建对应文件类型的 inode，在 fs.c 中的 create_inode 函数。

首先在 super block 的 inode 位图中找到第一个 0 所对应的下标，该下标便是新创建的文件的 inode_id，接着通过 inode_id 计算出其 inode 结构体所在虚拟磁盘块的块号 dblock_no。之后调用 malloc 函数申请新的 inode 的内存，将该内存加入到 icatch 所管理的链表中，接着将文件类型 file_type、inode_id 等已知信息填入该 inode 内存中之后，再调用 6 次 block_alloc 函数（该函数下面会介绍）申请空闲的 block 作为该 inode 的数据块，将 6 个新申请 block 的块号 block_no 填入到 inode 的 block_point 数组中，最后再将该 inode 写回到 dblock_no 号的磁盘块中（inode 的内存并不会释放掉，而是加入到 icatch 中）。create_inode 函数最后会返回一个该 inode 所在内存空间的指针。

2. data block 的创建

数据块的创建就是在上述文件被创建时进行的，本简易版 EXT2 为每个文件分配固定数量的数据块，在 fs.c 中的 block_alloc 函数。

首先在 super block 的 block 位图中找到第一个 0 所对应的下标，该下标便代表着该数据块的块号 block_no，接着调用 block_init 函数将此系统块所对应的两个磁盘块 dblock 进行初始化，具体就是往第 2*block_no、2*block_no+1 号的 dblock 中写入全 0（实际上不需要这么做，因为虚拟磁盘所创建的磁盘空间已经是全 0 了，但是为了避免意外，本实验实现的 EXT2 系统还是进行对分配的磁盘块进行写 0 初始化的操作）。在 block_alloc 函数的最后将刚分配的块号 block_no 作为返回值。

3.7 按路径查找文件

按照给定的路径在文件系统查找某个文件，其实就是按路径查找并返回该文件的 inode。在 fs.c 的 namei 函数中实现对给定路径查找并返回对应文件 inode 的功能。

1. 根目录和当前目录

在本实验实现的文件系统中，设置了两个全局变量 inode root_inode 和 inode* cwd_inode，一

个是结构体一个是结构体指针。其中 root_inode 就是存放的根目录文件的 inode 结构体，而 cwd_inode 指针指向的就是当前所在路径的 inode 结构体，在文件系统初始化时，会将第 0 号 inode（即根目录的 inode）从磁盘中读取出来并存放到 root_inode 中，接着再让 cwd_inode 指针指向 root_inode 的地址，这样在 cd 切换路径时只需切换 cwd_inode 指针所指向的 inode 即可完成当前路径的切换。

2. dirlookup 函数

dirlookup 函数，接收的参数有代表文件夹的 inode 以及代表所要查找的文件的文件名的字符串 name，通过读取 inode 的数据（inode 数据的读取和写入在后面会介绍），每次读取一个 dir_item 目录项结构体的数据，接着判断该目录项是否有效且目录项中记录的文件名和传入的参数 name 是否相同，如果名称相同就代表找到了对应的文件，而目录项中又记录有该文件的 inode_id，通过 inode_id 便可从磁盘组读取 inode 到内存中，最后 dirlookup 函数返回一个指向找到的文件 inode 所在内存空间的指针。当然如果在指定文件夹中没有找到相同名称的文件，则返回空指针。

3. namex 函数

namex 函数其实是 xv6 系统中用于查找指定路径上的文件 inode 的函数，在本实验的文件系统中借鉴了 xv6 系统中的该函数。首先 namex 会被两个函数调用，代表两种情况，namei 代表直接查找路径 path 对应的文件 inode，而 nameiparent 则是查找路径 path 所对应的文件所在的文件夹的 inode。在 namex 函数中，会循环处理传入的文件路径 path，每次调用 skipelem 函数获取到路径 path 中的下一个文件名 name，并使 path 指针前移，接着会在当前文件夹（inode 指针为 ip）中通过调用上述 dirlookup 函数查找文件名为 name 的文件，如果找到则将 dirlookup 返回的 inode 指针赋值给 next 指针，之后再将当前文件夹 inode 指针 ip 指向 next，并继续在路径上搜索下一个文件，如此循环下去。当然在第一次调用 dirlookup 函数时，所用的 ip 指针是当前工作路径 cwd_inode，代表从当前路径开始查找文件，之后 ip 再替换为下一层文件夹的 inode 指针。

4. dirlink 函数

dirlink 函数负责将给定的文件 inode_id，文件名 name，文件类型 file_type，写入到给定目录 inode 的数据块中。

首先 dirlink 函数将调用 dirlookup 函数在给定的目录中查找 name，如果已经有文件名和 name 相同的目录项了，则代表文件夹已经存在同名文件，因此 dirlink 将会报错并退出函数。如果不存在同名文件，则会在目录的数据块中找到第一个空闲(dir_item 的 valid=0)的目录项位置，将传入的文件信息写入到该目录项中，并将该目录项的有效位 valid=1 即可。

3.8 文件写入和读取

1. writei 函数

int writei(struct inode* ip, char* src, int offset, int len)

writei 函数参数格式如上，代表着从内存中源地址 src，向文件 ip 的第 offset 个字节开始写入 len 个字节数据。

对于数据写入，我们用变量 tot 和 m 来完成数据写入的过程。首先 tot 代表了当前已经写了多少字节数据，m 代表对于文件当前的磁盘块 dblock 我们应该写入的数据量。

```
for(tot=0; tot<len; tot+=m, offset+=m, src+=m){
  int dblock_no = ip->block_point[offset/BLOCK_SIZE]*2 + (offset%BLOCK_SIZE)/DBLOCK_SIZE;
```

```
  disk_read_block(dblock_no,buf);
  m = min(len - tot, DBLOCK_SIZE - offset%DBLOCK_SIZE);
  memmove(buf+(offset%DBLOCK_SIZE), src, m);
  disk_write_block(dblock_no,buf);
}
```

writei 中关键代码如上，从 tot=0 开始，每次计算当前数据写入的起始位置 offset 所在的虚拟磁盘块 dblock 的块号 dblock_no，因为每个系统块 block 对应两个磁盘块 dblock，因此需要计算(offset%BLOCK_SIZE)/DBLOCK_SIZE 来判断当前 offset 是在 block 对应的哪一个 dblock。

计算出应写入的磁盘块的块号 dblock_no 后，便可从该磁盘块中读取数据到缓冲区 buf 中。接着计算我们此次写入的数据长度 m，m 的作用是在数据写入第一个 dblock（或最后一个 dbloc 时需要从 dblock 的某个中间位置开始写入（或者到某个中间位置结束）。在写入完成后再调用虚拟磁盘的接口函数将数据写回虚拟磁盘块。完成一次写入后，便将 tot、offset、src 都增加 m，代表前面 m 字节的数据已经写入完毕，需要移动数据写入指针到下一位置。

2. readi 函数

readi 函数的和 writei 函数的参数、结构相似，只不过一个是从文件的第 offset 字节开始读取 len 字节数据到指定内存地址 src，而另一个是从内存地址 src 开始向文件的第 offset 字节写入 len 字节数据。其大概操作流程也和 writei 相似，即计算数据读取的起始位置 offset 所在的磁盘块的块号 dblock_no，接着调用虚拟磁盘接口函数读取磁盘块中的数据到 src 中，每读取一部分数据便将相应代表读取位置的参数更新到下一位置。

3.9 系统指令函数

在介绍完以上文件系统基础功能函数后，我们便可利用这些功能函数完成多种用户指令函数来供用户使用，在本实验实现的文件系统中，提供的指令有 mkdir、touch、ls、cp、cd、tee、cat、shutdown，接下来将介绍以上几种指令的实现思路。

1. mkdir

mkdir 指令是在指定路径 path 上完成文件名为 name 的文件夹创建，这部分功能通过调用 create_inode_in_path 函数完成。首先调用 namiparent 函数获取到 path 所指向的文件的上一级文件夹的 inode，接着调用 dirlookup 函数在该文件夹中查找是否已经有名称为 name 的文件（如果有则报错并退出），接着调用 create_inode 函数创建类型为 T_DIR 的 inode，之后再调用 dirlink 函数在新创建的目录文件的数据块中添加“.”和“..”两个目录项，最后调用 iupdate 函数将上一级文件夹的 inode 更新到磁盘中去即可。

2. touch

touch 指令和 mkdir 相似，也是调用 create_inode_in_path 函数，只不过创建的文件类型是 T_FILE，而且不需要往文件的数据块中写入“.”和“..”两个目录项。

3. ls

ls 指令就是展示指定路径 path 下的所有文件信息，首先调用 namei 函数获取到该目录文件 inode，接着调用 readi 函数从该目录文件的数据中读取目录项信息，对于每个 valid=1 的有效目录项，打印出其文件名和类型名即可。

4. cp

cp 指令将指定位置上的一个文件复制到另一个位置上，在本实验的文件系统中的 cp 指令，对于文件夹复制，只是简单地复制文件夹中的目录项数据，而并没有对文件夹中的文件再进行复制（只是浅复制）。

cp 指令首先调用 namei 获取到要复制的文件的 inode，再调用 create_inode_in_path 函数在指定路径上创建与之前文件同类型的新文件。接着，调用 readi 文件读取要复制的文件的所有数据，并调用 writei 将这些数据写入到新文件中即可。

5. cd

cd 指令实现切换当前工作路径 cwd 的功能，主要通过 namei 拿到 path 对应的目录文件，之后再将 cwd_node 切换为该目录文件即可。

6. tee

tee 指令实现对普通文件内容进行写入字符串数据的功能，主要通过 namei 拿到要编辑的文件的 inode，接着从标准输入中读取字符串数据，每次读取到一行字符串，就调用 writei 写入到文件末尾中（并不覆盖之前内容）。当然每次读取到一行字符串都会与“quit\n”进行比较，如果输入了 quit 则退出编辑文件的界面。

7. cat

cat 指令实现显示普通文件内容的功能，主要就是展示之前通过 tee 指令写入到文件中的字符串数据。首先通过 namei 拿到要读取文件的 inode，接着从该 inode 中调用 readi 函数读取数据到缓冲区 buf 中，再 printf 打印出来即可。

8. shutdown

shutdown 指令实现文件系统的安全关闭并退出的功能。主要是调用 icatch_free 函数将 icatch_list 中管理的所有内存中的 inode，通过调用 iupdate 函数将这些 inode 写回到磁盘中，最后还需要将根目录文件的 inode 写回到磁盘，以及将 super block 从内存中写回到磁盘中去。最后退出整个文件系统程序。

4. 总结及实验课程感想

这次的文件系统实验使我对 EXT2 文件系统的结构以及相应文件系统的功能实现有了更深的理解，也让我对操作系统的知识掌握得更加牢固。希望在之后的学习中能够进一步学习和实践有关计算机系统的知识。

5. 系统运行指南

进入到 MyEXT2 文件夹中，输入如下指令即可运行本文件系统：

```
mkdir build
cd build
cmake ..
make
```

```
./main
```

6. 用户手册

本实验实现的文件系统支持①创建文件或文件夹，②读取文件夹内容，③复制文件，④关闭系统，⑤系统再次启动时应能恢复为上次文件系统的状态，⑥向文件写入字符串数据，⑦从文件中读取字符串数据并显示出来。

1. mkdir <path>

功能：在指定路径上创建文件夹；
说明：mkdir 使用的 path 不会识别绝对路径寻址，所有使用的 path 都被认为是相对路径（即使前面加斜杠/）
示例图如下：

```
huhu@huhu-virtual-machine:~/HITSZ2020_OS/MyEXT2/build$ ./main
info: 系统初始化成功
@ mkdir a
@ mkdir b
@ ls
a       DIR
b       DIR
@ cd a
@ ls
.       DIR
..      DIR
@ cd ..
@ ls
a       DIR
b       DIR
@
```

2. touch <path>

功能：在指定路径上创建文件；
说明：touch 使用的 path 不会识别绝对路径寻址，所有使用的 path 都被认为是相对路径（即使前面加斜杠/）
示例图如下：

```
huhu@huhu-virtual-machine:~/HITSZ2020_OS/MyEXT2/build$ ./main
info: 系统初始化成功
@ touch a
@ ls
a       FILE
@
```

3. ls <path>

功能：展示指定文件夹中所有的文件；
说明：ls 使用的 path 不会识别绝对路径寻址，所有使用的 path 都被认为是相对路径（即使前面加斜杠/）
示例图如下：

```
huhu@huhu-virtual-machine:~/HITSZ2020_OS/MyEXT2/build$ ./main
info: 系统初始化成功
@ mkdir a
@ touch a/1
@ touch a/2
@ touch a/3
@ ls
a       DIR
@ ls /a
.       DIR
..      DIR
1       FILE
2       FILE
3       FILE
@
```

4. cp <target_path> <dest_path>

功能：将 target_path 指向的文件复制到 dest_path 上去；
说明：cp 使用的 path 不会识别绝对路径寻址，所有使用的 path 都被认为是相对路径（即使前面加斜杠/）
示例图如下：

```
huhu@huhu-virtual-machine:~/HITSZ2020_OS/MyEXT2/build$ ./main
info: 系统初始化成功
@ mkdir dir1
@ mkdir dir2
@ touch dir1/file1
@ ls
dir1    DIR
dir2    DIR
@ ls dir1
.       DIR
..      DIR
file1   FILE
@ cp dir1/file1 dir2/new_file1
@ ls dir2
.       DIR
..      DIR
new_file1       FILE
@
```

5. cd <path>

功能：切换当前工作路径为 path；
说明：**cd 使用的 path 会进行识别是绝对路径还是相对路径；**
示例图如下：

```
huhu@huhu-virtual-machine:~/HITSZ2020_OS/MyEXT2/build$ ./main
info: 系统初始化成功
@ mkdir dir1
@ mkdir dir1/dir1_1
@ ls
dir1    DIR
@ cd dir1
@ ls
.       DIR
..      DIR
dir1_1  DIR
@ cd dir1_1
@ ls
.       DIR
..      DIR
@ cd /
@ ls
dir1    DIR
@
```

6. tee <path>

功能：编辑指定路径上的文件；
说明：tee 使用的 path 不会识别绝对路径寻址，所有使用的 path 都被认为是相对路径（即使前面加斜杠/）；tee 无法编辑文件夹类型的文件；
示例图如下：

```
huhu@huhu-virtual-machine:~/HITSZ2020_OS/MyEXT2/build$ ./main
info: 系统初始化成功
@ mkdir dir1
@ touch file1
@ tee dir1
error: 文件夹不存在或路径名代表的不是文件
@ tee file1
info: 输入quit即可退出编辑
-> Hello World!
-> quit
@ cat file1
Hello World!
@
```

7. cat <path>

功能：展示指定路径上的文件；
说明：cat 使用的 path 不会识别绝对路径寻址，所有使用的 path 都被认为是相对路径（即使前面加斜杠/）；cat 无法展示文件夹类型的文件；
示例图如下：

```
huhu@huhu-virtual-machine:~/HITSZ2020_OS/MyEXT2/build$ ./main
info: 系统初始化成功
@ mkdir dir1
@ touch file1
@ tee dir1
error: 文件夹不存在或路径名代表的不是文件
@ tee file1
info: 输入quit即可退出编辑
-> Hello World!
-> quit
@ cat file1
Hello World!
@
```

8. shutdown

功能：安全退出文件系统；
示例图如下：

```
huhu@huhu-virtual-machine:~/HITSZ2020_OS/MyEXT2/build$ ./main
@ ls
dir1    DIR
file1   FILE
@ shutdown
huhu@huhu-virtual-machine:~/HITSZ2020_OS/MyEXT2/build$
```