Filesystems HW6

Features and requirements

Features and requirements

There is only one directory — the root directory — all files reside there.

Features and requirements

There is only one directory — the root directory — all files reside there.

You do not need to implement caching or buffering on either the reading or writing side of the filesystem.

Features and requirements

There is only one directory — the root directory — all files reside there.

You do not need to implement caching or buffering on either the reading or writing side of the filesystem.

All calls can operate synchronously with the disk. That is, you can block the calling thread until the disk has actually performed all of the requested operations.

Features and requirements

There is only one directory — the root directory — all files reside there.

You do not need to implement caching or buffering on either the reading or writing side of the filesystem.

All calls can operate synchronously with the disk. That is, you can block the calling thread until the disk has actually performed all of the requested operations.

file block indexing is direct — you need not implement indirect blocks, double-indirect blocks, etc.

Features and requirements

There is only one directory — the root directory — all files reside there.

You do not need to implement caching or buffering on either the reading or writing side of the filesystem.

All calls can operate synchronously with the disk. That is, you can block the calling thread until the disk has actually performed all of the requested operations.

file block indexing is direct — you need not implement indirect blocks, double-indirect blocks, etc.

Each file can be referenced by at most one file descriptor at a time.

Features and requirements

There is only one directory — the root directory — all files reside there.

You do not need to implement caching or buffering on either the reading or writing side of the filesystem.

All calls can operate synchronously with the disk. That is, you can block the calling thread until the disk has actually performed all of the requested operations.

file block indexing is direct — you need not implement indirect blocks, double-indirect blocks, etc.

Each file can be referenced by at most one file descriptor at a time.

A single file descriptor,however, can be shared among multiple threads.

Features and requirements

There is only one directory — the root directory — all files reside there.

You do not need to implement caching or buffering on either the reading or writing side of the filesystem.

All calls can operate synchronously with the disk. That is, you can block the calling thread until the disk has actually performed all of the requested operations.

file block indexing is direct — you need not implement indirect blocks, double-indirect blocks, etc.

Each file can be referenced by at most one file descriptor at a time.

A single file descriptor,however, can be shared among multiple threads.

There are no hard links or symbolic links.

Features and requirements

There is only one directory — the root directory — all files reside there.

You do not need to implement caching or buffering on either the reading or writing side of the filesystem.

All calls can operate synchronously with the disk. That is, you can block the calling thread until the disk has actually performed all of the requested operations.

file block indexing is direct — you need not implement indirect blocks, double-indirect blocks, etc.

Each file can be referenced by at most one file descriptor at a time.

A single file descriptor,however, can be shared among multiple threads.

There are no hard links or symbolic links.

Some filesystem operations are defined to only operate on files that are not open.

Features and requirements

There is only one directory — the root directory — all files reside there.

You do not need to implement caching or buffering on either the reading or writing side of the filesystem.

All calls can operate synchronously with the disk. That is, you can block the calling thread until the disk has actually performed all of the requested operations.

file block indexing is direct — you need not implement indirect blocks, double-indirect blocks, etc.

Each file can be referenced by at most one file descriptor at a time.

A single file descriptor,however, can be shared among multiple threads.

There are no hard links or symbolic links.

Some filesystem operations are defined to only operate on files that are not open.

Disk hardware is defined to be free of failures.

```
int MT_fs_open (const char *name,
  int create);
```

File names are limited to 63 characters and are null-terminated like other C strings.

You must support up to 100 open files at a time

it is an error to call MT_fs_open when there are already 100 open files.

```
int MT_fs_close (int fd);
```

Closes the file associated with the file descriptor.

Return 0 on success or -1 if fd is out of rangeor is not open.

```
int MT_fs_unlink (const char *name);
```

Delete the closed file with filename name.

Return 0 on success or -1 if the filename is invalid, the file does not exist, or the file is currently open.

On success, all disk blocks associated with the file must be freed.

```
int MT_fs_read (int fd,
  void *buf, int count);
```

Read count bytes from the file referred to by fd into the memory buffer buf. Quoting from the Linux man pages:

Linux Man Pages

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (because we were close to end-of-file). On error, -1 is returned. In this case it is left unspecified whether the file position changes.

```
int MT_fs_write (int fd,
  void *buf, int count);
```

Write count bytes to the file referred to by fd from the memory buffer buf.

Linux Man Pages

On success, the number of bytes written are returned (zero indicates nothing was written). On error, -1 is returned. If count is zero, 0 will be returned without causing any other effect.

```
int MT_fs_getpos (int fd);
```

Returns the current file position on success or -1 on error. This function will only fail if fd is not a valid file descriptor.

```
int MT_fs_truncate (const char *name,
  int length);
```

Linux Man Pages

On success, the closed file with filename name has a length of length bytes. If the file was previously longer, it is shortened. If it was previously shorter, it is extended and zeros are written to the bytes between the previous end-of-file and the new end-of-file. Return 0 on success or -1 if the filename is invalid, the file does not exist, the file is currently open, or an allocation error occurs (filesystem runs out of free blocks or free index slots) while the truncate is in progress.

`int MT_fs_lseek (int fd, int pos);`

On success, the current position of the open file associated with the descriptor is set to pos. If pos is -1 then the position is set to the end-of-file. If pos is larger than the current end-of-file then the file is extended, with zeros being written between the previous end-of-file and the new end-of-file.

Returns the new file position on success or -1 on error (for example, if fd is invalid or an allocation error occurs while extending the file).

Additional requirements

As in UNIX, each valid file descriptor should refer to an in-kernel (the MT kernel, not the OS kernel) data structure that represents the state associated with the open file, such as the current file position.

All MT filesystem functions are required to operate atomically with respect to the associated file. This means that if, for example, two threads take turns writing to a file (sharing the same file descriptor) then there is no interleaving between chunks of data written by individual write commands issued by the threads.

The process containing MT threads should be able to terminate at an arbitrary time — as long as no threads are currently executing an MT filesystem function — without leaving the filesystem in an inconsistent state.

The filesystem must be left in a consistent state after “normal” allocation errors occur. For example, consider a situation where the filesystem runs out of blocks while extending a file to accommodate an lseek beyond the previous end of the file. In this case the lseek fails, but the file is left in a usable state: the end-of-file and current file position should be set to the last byte of the last block that was successfully allocated.

On-Disk Format

int zero_block[128]:

```
zero_block[0]   int magic1 == 0xabbaabba;
zero_block[1]   int magic2 == 0xf00bf00b;
zero_block[2]   int block size == 512;
zero_block[3]   int block cnt;
zero_block[4]   int free list head;
zero_block[5]   int file0;
zero_block[6]   int file1;
zero_block[7]   int file2;
...
zero_block[127] int file122;
```

int inode[128]:

```
inode[0]     int magic == 0xdeadbeef;
inode[1–16]  char filename[64];
inode[17]    int size;
inode[18]    int block0;
inode[19]    int block1;
inode[20]    int block2;
...
inode[127]   int block110;
```

The block device interface

The block device interface

The block device interface is asynchronous.

The block device interface

The block device interface is asynchronous.

asking it to do something is a non-blocking operation.

The block device interface

The block device interface is asynchronous.

asking it to do something is a non-blocking operation.

disk is defined to be busy between when a read or write is initiated and when the associated interrupt arrives, and idle at all other times

block device API

```
int MT_init_block_dev (const char *fn,
  disk_int_t int_routine,
  int *block_cnt,
  int *block_size);

int MT_initiate_block_get (int block_num, void *buf);

int MT_initiate_block_put (int block_num, void *buf);

void __MT_raw_disk_handler__ (void);
```

Optional - synchronous disk interface

inconvenient (to say the least)

implement a synchronous disk interface on top of the asynchronous interface

implement your filesystem in terms of these synchronous operations.

Optional - synchronous disk interface

synchronous operations should block the calling thread until the associated operation has completed.

will need to implement a disk request scheduler as part of this layer

first-come first-serve is sufficient

synchronous free_block() and find_free_block()

Optional - cached copy of the zero block

can increase performance

always write out the zero block every time the cached copy is changed

you never know when the user program is going to exit

filesystem should always be in a consistent state between operations.

Testing

Invalid and out-of-range file descriptor arguments

File names that are too long

Running out of inode slots in the zero block

Running out of free disk blocks

Running out of data block slots in an inode

Writes past the end of a file —these are legal but require data blocks to be allocated

Reads and writes that span many disk blocks—these are legal as well, but the operation

must implemented in terms of smaller operations that work on blocks or parts of blocks