Subdirectories

TABLE OF CONTENTS

The current Pintos file system supports directories, but user programs have no way of using them (i.e. files can only be placed in the root directory right now). You must add the following system calls to allow user programs to manipulate directories: `chdir`, `mkdir`, `readdir`, `isdir`. You must also update the following system calls so that they work with directories: `open`, `close`, `exec`, `remove`, `inumber`. You must also add support for **relative paths** for any syscall with a file path argument. For example, if a process calls `chdir("my_files/")` and then `open("notes.txt")`, you should search for `notes.txt` relative to the current directory and open the file `my_files/notes.txt`. You also need to support absolute paths like `open("/my_files/notes.txt")`. You need to support the special `"."` and `".."` names, when they appear in file path arguments, such as `open("../logs/foo.txt")`. Child processes should inherit the parent's current working directory. The first user process should have the root directory as its current working directory.

Implementation details

Implement support for hierarchical directory trees. In the basic file system, all files live in a single directory. Modify this to allow directory entries to point to files or to other directories. Make sure that directories can expand beyond their original size just as any other file can.

The basic file system has a 14-character limit on file names. You may retain this limit for individual file name components, or may extend it. **You must allow full path names to be much longer than**

14 characters.

Maintain a separate current directory for each process. At startup, set the file system root as the initial process's current directory. When one process starts another with the `exec` system call, the child process inherits its parent's current directory. After that, the two processes' current directories are independent, so that either changing its own current directory has no effect on the other. (This is why, under Unix, the `cd` command is a shell built-in, not an external program.)

Update the existing system calls so that anywhere a file name is provided by the caller, an absolute or relative path name may be used. The directory separator character is forward slash (`/`). You must also support special file names `.` and `..`, which have the same meanings as they do in Unix.

Update the `open` system call so that it can also open directories. You **should not** support `read` or `write` on a file descriptor that corresponds to a directory. You will implement the `readdir` and `mkdir` syscalls for directories instead. You **should** support `close` on a directory, which just closes the directory.

Update the `remove` system call so that it can delete empty directories (other than the root) in addition to regular files. Directories may only be deleted if they do not contain any files or subdirectories (other than `.` and `..`). You may decide whether to allow deletion of a directory that is open by a process or in use as a process's current working directory. If it is allowed, then attempts to open files (including `.` and `..`) or create new files in a deleted directory must be disallowed.

Here is some code that will help you split a file system path into its components. It supports all of the features that are required by the tests. It is up to you to decide if and where and how to use it.

```
/* Extracts a file name part from *SRCP into PART, and updates *SRCP so that the
   next call will return the next file name part. Returns 1 if successful, 0 at
   end of string, -1 for a too-long file name part. */
static int get_next_part(char part[NAME_MAX + 1], const char** srcp) {
  const char* src = *srcp;
  char* dst = part;

  /* Skip leading slashes.  If it's all slashes, we're done. */
  while (*src == '/')
    src++;
  if (*src == '\0')
    return 0;
```

```
  /* Copy up to NAME_MAX character from SRC to DST.  Add null terminator. */
  while (*src != '/' && *src != '\0') {
    if (dst < part + NAME_MAX)
      *dst++ = *src;
    else
      return -1;
    src++;
  }
  *dst = '\0';

  /* Advance source pointer. */
  *srcp = src;
  return 1;
}
```

Syscall signatures

Implement the following new system calls:

chdir

```
bool chdir(const char* dir)
```

Changes the current working directory of the process to `dir`, which may be relative or absolute. Returns true if successful, false on failure.

mkdir

```
bool mkdir(const char* dir)
```

Creates the directory named `dir`, which may be relative or absolute. Returns true if successful, false on failure. Fails if `dir` already exists or if any directory name in `dir`, besides the last, does not already exist. That is, `mkdir("/a/b/c")` succeeds only if `/a/b` already exists and `/a/b/c` does not.

readdir

```
bool readdir(int fd, char* name)
```

Reads a directory entry from file descriptor `fd`, which must represent a directory. If successful, stores the null-terminated file name in `name`, which must have room for `READDIR_MAX_LEN + 1` bytes, and returns true. If no entries are left in the directory, returns false.

`.` and `..` should not be returned by `readdir`

If the directory changes while it is open, then it is acceptable for some entries not to be read at all or to be read multiple times. Otherwise, each directory entry should be read once, in any order.

`READDIR_MAX_LEN` is defined in `lib/user/syscall.h`. If your file system supports longer file names than the basic file system, you should increase this value from the default of 14.

isdir

```
bool isdir(int fd)
```

Returns `true` if `fd` represents a directory, `false` if it represents an ordinary file.

inumber

```
int inumber(int fd)
```

Returns the inode number of the inode associated with `fd`, which may represent an ordinary file or a directory.

An inode number persistently identifies a file or directory. It is unique during the file's existence. In Pintos, the sector number of the inode is suitable for use as an inode number.

We have provided the `ls` and `mkdir` user programs, which are straightforward once the above syscalls are implemented. We have also provided `pwd`, which is not so straightforward. The `shell` program implements `cd` internally.

The `pintos extract` and `pintos append` commands should now accept full path names, assuming that the directories used in the paths have already been created. This should not require any significant extra effort on your part.
