ARIZONA STATE UNIVERSITY

CSE 430 — **Operating Systems** — Fall 2012

SLN 71250

Instructor: Dr. Violet R. Syrotiuk

Project #2

Handed out Tuesday 10/23/2012; due Tuesday 11/20/2012

In shared memory multicore architectures, threads can be used to implement parallelism. A standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard. Implementations that adhere to this standard are referred to as *POSIX threads*, or *pthreads*. The purpose of this project is to gain some experience using *pthreads* to create and synchronize threads using semaphores while implementing a file system accessed by multiple users, simulated by threads.

1 A File System

Design a program called **u6fs** that allows a UNIX user access to the file system of a foreign operating system, UNIX Version 6. The **u6fs** program must accept a two arguments:

u6fs disk k

where `disk` is the name of the file that physically represents the disk storing the file system and `k` is the number of users accessing the file system.

The basic operation of **u6fs** is to first have the main thread initialize the file system. The file system can be initialized in two ways:

1. If the file `disk` does not exist then file system is initialized from scratch. This involves setting all of the data blocks to free, and setting all i-nodes as unallocated. Allocate 40 blocks to i-nodes, i.e., the i-list has length 40. Since i-nodes are 32 bytes long, 16 of them fit in a block. Therefore, a maximum of 40×16 files in total may exist on the `disk`.

2. If the file `disk` exists then the file system is initialized from the file contents; it is assumed that the file `disk` adheres to the format of the version 6 file system.

In both cases, you are to map the file system onto a block of main memory and work on it in memory. The file system is written to the file `disk` by the main thread, before it terminates.

Once the file system is initialized, the main thread then spawns `k` threads each simulating a user. Each user i, $1 \leq i \leq \mathtt{k}$, executes a sequence of commands found in a file `useri.txt`. Valid commands include:

1. **cpin** `yourUnixfile v6file`

   Create a new file called `v6file` in the current directory of the file system and copy the contents of `yourUnixfile` to the newly created file. No wildcards are permitted in specifying the filenames. The owner of the file is user i and the mode should be read, write, execute by owner.

2. **cd** `v6directory`

   Change the current working directory *on the v6 disk* to the specified directory.

3. **cpout** `v6file yourUnixfile`

   If the specified `v6file` exists, create the specified `yourUnixfile` and copy the `v6file` into the `yourUnixfile`. If `yourUnixfile` is `stdout` then simply copy the contents of the `v6file` to standard out.

4. **mkdir** `directory`

   Create the named `directory` with mode read, write, execute by owner, group, others.

5. **chmod** `mode file`

   Changes or assigns the mode of a file. The mode of a file specifies its permissions and other attributes. The `mode` is zero followed by 4 octal digits and is used to set the `flags` field in the i-node.

6. **exit**

   This is the last command of the sequence and it causes thread i to terminate.

A trace of each user thread's execution should be made to the file `useri-trace.txt` for $1 \leq i \leq \texttt{k}$.

When a file or directory name on the v6 disk is required, a convention similar to that of any UNIX system should be allowed. The name "/" refers to the top-level directory on the v6 disk (**not** the root of your UNIX filesystem). A name that begins with "/" denotes a path relative to that top-level directory. A name that does not begin with "/" is a path relative to your current v6 working directory (see the **cd** command below).

Because multiple threads may access the superblock and i-list concurrently, care is required to maintain the integrity of these data structures. Use semaphores to ensure mutual exclusive access to these data structures while at the same time maximizing concurrency.

The main thread waits for each user thread to terminate, and then writes the entire file system from memory to `disk`.

2 Format of the Version 6 File System

The detailed structure of a disk pack structured for the UNIX v6 operating system is as follows:

The disk pack is divided into 1000 blocks of 512 characters each. These blocks are addressed as $0, 1, 2, \ldots, 999$.

The zero-th block is unused by UNIX, but typically contains the boot loader if the disk is intended to be used as the "root device". You should ignore it since it contains no useful information.

The first block is called the "super-block" of a UNIX file system device. The super-block defines the layout of the rest of the file system volume. Section 2.1 gives the precise format of the super-block as it exists on any disk formatted for use as a v6 file system volume. In addition, it gives the detailed format of **i-nodes** on the file system disk. i-nodes are "file descriptors". There is one active i-node per file on the file system disk. Directories, as well as regular data files, have associated i-nodes.

Directories are in a format as described in Section 2.2. Directory entries associate symbolic names of files with an index into the i-node area of the disk (the i-list). A most important fact is that the i-node for the root directory of the hierarchy on the disk is the first i-node in the area set aside as the i-list. Given that initial hook into the directory structure, you should be able to traverse the directory tree of the file system.

2.1 Format of File System Volume

Every file system storage volume has a common format for certain vital information. Every such volume is divided into a certain number of 256 word (512 byte) blocks. (Here, 1 word is 2 bytes long.) Block 0 is unused and is available to contain a bootstrap program, pack label, or other information.

Block 1 is the "super block." The format of a super-block is:

```
struct superblock {
   unsigned short isize;
   unsigned short fsize;
   unsigned short nfree;
   unsigned short free[100];
   unsigned short ninode;
   unsigned short inode[100];
   char flock;
   char ilock;
   char fmod;
```

```
    unsigned short time[2];
};
```

isize is the number of blocks devoted to the i-list, which starts just after the super-block, in block 2. In this implementation isize is 40.

The free list for each volume is maintained as follows. The free array contains, in

$$\mathtt{free}[1], \ldots, \mathtt{free}[\mathtt{nfree} - 1]$$

up to 99 numbers of free blocks. free[0] is the block number of the head of a chain of blocks constituting the free list. The first word in each free-chain block is the number (up to 100) of free-block numbers listed in the next 100 words of this chain member. The first of these 100 blocks is the link to the next member of the chain. To allocate a block: decrement nfree, and the new block is free[nfree]. If the new block number is 0, there are no blocks left, so give an error. If nfree became 0, read in the block named by the new block number, replace nfree by its first word, and copy the block numbers in the next 100 words into the free array. To free a block, check if nfree is 100; if so, copy nfree and the free array into it, write it out, and set nfree to 0. In any event set free[nfree] to the freed block's number and increment nfree.

For this project, the fields fsize, ninode, the array inode, flock, ilock, fmod, and time in the i-node will not be used and can be ignored.

i-numbers begin at 1, and the storage for i-nodes begins in block 2.

Also, i-nodes are 32 bytes long, so 16 of them fit into a block. i-node 1 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each i-node represents one file. The format of an i-node is as follows:

```
struct inode {
    unsigned short flags;  /* +0: see below */
    char nlinks; /* +2: number of links to file */
    char uid;    /* +3: user ID of owner */
    char gid;    /* +4: group ID of owner */
    char size0;  /* +5: high byte of 24-bit size */
    unsigned short size1;  /* +6: low word of 24-bit size */
    unsigned short addr[8];        /* +8: block numbers or device number */
    unsigned short actime[2];      /* +24: time of last access */
    unsigned short modtime[2];     /* +28: time of last modification */
};
```

The flags are as follows:

```
    100000   i-node is allocated
    060000   2-bit file type
    000000   plain file
    040000   directory
    010000   large file
    004000   set user-ID on execution
    002000   set group-ID on execution
    000400   read (owner)
    000200   write (owner)
    000100   execute (owner)
    000070   read, write, execute (group)
    000007   read, write, execute (others)
```

The address words of ordinary files and directories contain the numbers of the blocks in the file (if it is small) or the numbers of indirect blocks (if the file is large). Byte number n of a file is accessed as follows. n is divided by 512 to find its logical block number (say `b`) in the file. If the file is small (the large file flag 010000 is 0), then `b` must be less than 8, and the physical block number is `addr[b]`.

In this project, we only implement small files using direct blocks, i.e., each file is at most 8 blocks in size (or a maximum of $4K$).

2.2 Format of Directories

A directory behaves exactly like an ordinary file, except that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry. Directory entries are 16 bytes long. The first word is the i-number of the file represented by the entry, if non-zero; if zero, the entry is empty.

Bytes 2–15 represent the (14-character) file name, null padded on the right. These bytes are not cleared for empty slots.

By convention, the first two entries in each directory are for "**.**" and "**..**". The first is an entry for the directory itself. The second is for the parent directory. The meaning of "**..**" is modified for the root directory of the master file system and for the root directories of removable file systems.

3 Bonus

For a bonus, implement the following additional commands:

1. **ls**

   List the files in the current working v6 directory.

2. **mv** `v6file1 v6file2`

   Rename the v6 file named by `v6file1` to `v6file2`.

3. **pwd**

   Writes the absolute pathname of the current working directory.

4. **find** `v6file`

   Recursively descends the directory tree and reports the absolute path of each occurrence of `v6file` found. (This is a simplified version of find; the actual find command can find amazing things!)

4 Hand-in instructions

Submit electronically, before 9:00am on 11/20/2012 using the submission link for Project #2, a zip file named `FirstName-LastName.zip` containing the following items:

Design and Analysis (30%): Type up a spell-checked description of the methodology you followed to produce a correct *pthreads* threaded program synchronized using semaphores to implement the file system. Describe how you managed concurrency.

Implementation (50%): Your documented C/C++ source code of your *pthreads* solution to the file system problem. Also include a `README` file that *explains in detail* how to compile and run your code, including assumptions about your runtime environment. You may write your code only in C or C++.

You must not alter the requirements of this program in any way.

Correctness (20%) You will sign up to demo your program to our TA.

Bonus (10%) A bonus is available for those who implement the additional commands in Section 3.