Lab # 3: The Yalnix File System

Project Due: 11:59 PM, Friday, April 24, 2015

1. Project Overview

Your task is to implement a simple file system for the Yalnix operating system, called YFS. Your file system will support multiple users, with a tree-structured directory, and will be similar in functionality and disk layout to the "classical" Unix file system. Unlike the Unix file system implementation, though, your file system will be implemented as a *server* process running on top of the Yalnix kernel. Client processes using the file system send individual file system requests as interprocess communication (IPC) messages to the file server process. The server handles each request, sends a reply message back to the client process at completion, and then waits for another request to arrive at the server. Interfacing to the disk hardware is handled by the Yalnix kernel, which provides kernel calls for use by the file server for reading and writing sectors on the disk. The file system server definition in this project is simplified in order to make the project more manageable. For example, the file system server is not multi-threaded; it handles only a single request at a time.

Section 2 of this document defines the disk representation of YFS file system and the characteristics of the (simulated) disk on which you are to implement YFS. Section 3 defines the new kernel calls that have been added to the Yalnix kernel for reading and writing the disk. Section 4 defines the file system requests that your file system server must be able to handle, and sketches the data structures you should maintain in the memory of your file system server process. Finally, the specific requirements for this project are described in Section 5.

2. File System Representation

The C include file `filesystem.h` defines important data structures and constants that you will need in your file system implementation. This header file is located in

```
/clear/courses/comp421/pub/include/comp421/filesystem.h
```

and can be include in your source file using

```
#include <comp421/filesystem.h>
```

The format of your file system on disk *must* conform to the format defined in the `filesystem.h` include file. The contents of the disk remain on disk between runs of Yalnix; by using the same data structure format on disk for your file system, you can try your file system server process on various disk images, including disks built by other people. This section describes the required format of the data structures on the disk for the Yalnix file system.

2.1. Disk Hardware Characteristics

The Yalnix hardware simulation has been enhanced with the addition of a hard disk drive, on which you will implement your file system. The important physical characteristics of the disk drive are defined by the following constants:

- `SECTORSIZE`: The size (in bytes) of each sector on the disk.
- `NUMSECTORS`: The total number of sectors on the disk.

2.2. File System Disk Layout

The Yalnix file system layout on disk is designed to make efficient use of space on small disk drives like the one provided by the hardware simulation.

The file system uses a block size of one sector, and thus each file system block occupies exactly one disk sector. The constant `BLOCKSIZE` in `filesystem.h` is thus defined equal to `SECTORSIZE`.

2.2.1. The Boot Block

Block 0 is the *boot block*. In a real system, this first block of the disk contains executable code used in booting the operating system. At boot time, the hardware reads this boot code into memory and executes it; this code then reads the operating system kernel into memory and finally begins execution of the kernel. The boot block is not used by the file system, but must be reserved on the disk for booting the operating system. Since the boot block is never used by the file system, the block number value 0 can be used by the file system implementation as a special value, such as to indicate that no block number has been assigned yet.

2.2.2. The File System Header and Inodes

Following the boot block is a number of sequential disk blocks containing file system control information, beginning with block 1. At the beginning of block 1 is a small data structure that is used to define the format of the file system on that disk drive. This data structure, called in YFS the *file system header*, is similar to the Unix *super block*, although in Unix, it is large enough to require a whole block to itself.

The file system header defines the information necessary to determine the size and location of the remaining two areas of the file system disk layout. The format of the file system header is defined in the include file `filesystem.h` by the following C structure declaration:

```
struct fs_header {
    int num_blocks;     /* total blocks in file system */
    int num_inodes;     /* number of inodes in file system */
    int padding[14];    /* make fs_header and inode same size */
};
```

The number of blocks given by `num_blocks` specifies the total size of the file system in blocks. This size *includes* the sizes of the boot block, the inodes, and all file system data blocks. The `num_inodes` field specifies the total number of *inodes* (described below) in the file system.

Each file in the file system is defined by exactly one inode, specifying the size of the file (in bytes), the number of directory entries that point to this inode (names under which this file is known), and the pointers to the data blocks of this file. After the file system header, the remainder of block 1 is filled with inodes, and all blocks following block 1 contain more inodes, for a total of `num_inodes` inodes. The file system header is exactly the same size as an inode and is given by the constant `INODESIZE` defined in `filesystem.h`. This size is an integral fraction of the file system block size (`BLOCKSIZE`), and thus no inode will be split across a block boundary.

Since the file system header takes the place of what would be inode number 0 (the file system header is the same size as an inode), the first valid inode number is inode number 1, and inode number 0 is *not* a valid inode number within the file system. The value `num_inodes` is a count of only inodes and does not include the file system header. Also, since the inodes are laid out consecutively on the disk following the file system header, *it is easy to determine the location of any inode given its inode number*. For example, inode number 1 immediately follows the file system header, inode number 2 follows inode number 1, and so on. There are exactly `BLOCKSIZE/INODESIZE` inodes in each block worth of inodes, except in the

first of these blocks, which has one less inode in it due to the space taken up by the file system header in that block, and possibly also in the last of these blocks, which might have not all of its space full of inodes (the end of this block could be wasted space).

The format of each inode is defined in the include file `filesystem.h` by the following C structure declaration:

```
struct inode {
    short type;         /* file type (e.g., directory or regular) */
    short nlink;        /* number of links to inode */
    int   reuse;        /* inode reuse count */
    int   size;         /* file size in bytes */
    int   direct[NUM_DIRECT]; /* block numbers for 1st 13 blocks */
    int   indirect;     /* block number of indirect block */
};
```

The field `type` specifies the type for this file in the file system. The following three type values are defined in `filesystem.h`:

- `INODE_FREE`: This inode is not in use for any file.
- `INODE_DIRECTORY`: This inode describes a directory.
- `INODE_REGULAR`: This inode describes a regular data file.
- `INODE_SYMLINK`: This inode describes a symbolic link.

The field `nlink` specifies the number of (hard, not symbolic) links that point to this inode; as such, this count is also equal to the total number of directory entries across the file system that have this inode number in their `inum` field (see Section 2.3). Hard links are created with the `Link` file system operation (see Section 4.2).

The field `reuse` indicates the number of times that this particular inode has been "reused" since the file system was originally formatted. When the file system is formatted, the `reuse` field in each inode is initialized to 0. Each time an inode is allocated (from being free), the `reuse` count in the inode must be incremented. The reuse count in each inode allows your file server process to ensure, for example after doing an `Open` on some file, that on later attempts to `Read` or `Write` that open file, the *same* file still exists and that the inode has not instead been reused since that `Open` for some different file.

The field `size` specifies the total size (in bytes) of the data contained in the file. Although space allocation is done by whole blocks, some space at the end of the last block allocated to a file may be unused.

The data blocks allocated to a file are specified by the `direct` and `indirect` fields in the file's inode. These fields contain file system block numbers, *not* memory pointers. The block number of each of the first `NUM_DIRECT` blocks of the file are directly specified by the entries in the `direct` array. For files smaller than `NUM_DIRECT` blocks, the `indirect` field and the later entries in `direct` are unused but are still present in the inode. For files larger than `NUM_DIRECT` blocks, the |indirect| field gives the block number of the file's *indirect block*, containing the block numbers of the remaining blocks of the file. The indirect block is formatted as an array of block numbers allocated to the file, indexed by the block number within the file minus `NUM_DIRECT`. Each block number in the indirect block is represented as an "`int`" (occupying 4 bytes). Only one indirect block may be used for each file, defining the maximum file size supported by the file system by the number of block numbers (4 bytes each) that will fit in a single file system block (of size `BLOCKSIZE`) and the number of block numbers contained in the inode (`NUM_DIRECT`). Any attempt to extend a file to be larger than can be mapped with a single indirect block should be considered as an error by your file system.

There is no list of free inodes recorded on the disk. Instead, the file system builds its own free list in memory at startup, by scanning the inodes looking for ones of type `INODE_FREE`.

2.2.3. Data Blocks

Beginning with the first whole block after the inodes, all remaining blocks in the YFS file system are data blocks to be used by files. These blocks may either be allocated for use to store file data or directory data, or for use as indirect blocks to allow extending a file beyond the `NUM_DIRECT` blocks that can be directly indicated by the inode. The *total* number of blocks in the file system is given by the `num_blocks` field of the file system header; the number of data blocks present must be determined by subtracting from this the number of blocks required to hold the boot block, the file system header, and `num_inodes` inodes.

There is no list of free blocks recorded on the disk. Instead, the file system builds its own free list in memory at startup, by scanning the inodes and indirect blocks to determine what blocks are already allocated.

2.3. Directory Format

The YFS file system supports a tree-structured directory, similar to Unix and Linux. The directory is used to translate a file name given by a user of the file system, into the inode number for that file. Some files in the file system are regular data files and some are directories (indicated by the `type` field in each file's inode). The tree-structured directory is formed by allowing entries in one directory to give the inode number of another directory file. Although this structure could be used to represent an arbitrary graph of directories pointing at other directories, the Yalnix file system restricts the structure to a tree. This is achieved by preventing users from creating links (using the `Link` file system operation) to directories, thus guaranteeing that the directory structure contains no loops.

As in Unix, one specific directory in the file system is known as the *root* directory. In Yalnix, the root directory is always represented by inode number `ROOTINODE`, which is defined in `filesystem.h` to be 1. (in Unix, it is inode number 2, for historical reasons). Thus, your file system can always find the root inode, and can thus always access the root directory. All other directories are found by following entries in one directory, through the inode number contained in that directory entry, to another directory. The constant name `ROOTINODE` should be used to refer to the inode number of the root directory.

The space allocated to a directory is represented in exactly the same way as the space allocated to a regular file, but the data cannot be explicitly written by user file system requests. It is only accessible implicitly through file system requests that take file names as arguments, such as `Create`, `Open`, `Link`, and `Unlink`. Although user programs cannot directly write to a directory, the data in a directory (the directory entries) *can* be read by user programs by a normal `Read` file system request.

The directory data is divided into a sequence of *directory entries*, all of the same size. Each directory entry maps a single file name to the corresponding inode number, according to the following C structure definition from `filesystem.h`:

```
struct dir_entry {
    short inum;               /* inode number */
    char  name[DIRNAMELEN];   /* file name (not null-terminated) */
};
```

The size of the directory entry structure is an integral fraction of the file system block size (`BLOCKSIZE`), and thus no directory entry will be split across a block boundary.

In each directory entry, the `inum` field gives the inode number of the file mapped by this entry. If this field is 0 (an invalid inode number), this directory entry is currently unused. As file names are removed

from the directory (on an `Unlink` or a `RmDir`), the corresponding directory entry is modified so that its `inum` field is 0. The directory entry is then said to be *free*.

The `name` field of a directory entry gives the name of the file mapped by this entry. This field has a constant length of `DIRNAMELEN` characters (defined in `filesystem.h`). The file name in a directory entry may contain *any* characters other than a slash (`'/'`) or the null character (`'\0'`). For example, even "unusual" characters such as a space, a backspace, or even a newline character are all legal in a file name; each of these characters should be treated exactly the same as any other character in a file name.

If the actual file name is shorter than `DIRNAMELEN` characters, the remainder of the characters in this field should be filled with the null character (`'\0'`). If the file name is exactly `DIRNAMELEN` characters long, there will be *no* null character at the end of the file name in the directory entry (it would waste a lot of disk space to leave room store the null character for every directory entry, and it is not needed to understand the disk format). You thus cannot use the functions `strlen` or `strcpy` to access the file name, since the null terminating character may not be present. When looking at the file name in a directory entry, the name ends at the first null character *or* after `DIRNAMELEN` characters, whichever occurs first. Any attempt to create a file name longer than `DIRNAMELEN` characters should be considered as an error by your file system.

The `name` field of a free directory entry (with `inum` set to 0) need have no special value and should be ignored by your file system.

2.4. Pathnames

As in Unix, each file is identified by a *pathname*, such as `/dirname/subdirname/file.` The slash character (`'/'`) at the beginning of the pathname indicates that this is an *absolute* pathname, meaning that processing of that pathname begins at the inode of the *root directory*. A pathname such as `otherdir/otherfile` (or just `myfile.c`), which does *not* begin with a slash character is a *relative* pathname; processing of a relative pathname begins at the inode of the *current directory* of the requesting user process, set by the process calling the `ChDir` operation (the initial current directory of each process when the process begins execution should be "`/`").

The slash character is also used to separate individual directory names along the path to the file, and to separate the last directory name from the name of the particular file in that directory. More than one slash character in a row within a pathname should be processed the same as if a single slash character had occurred there (except that those additional slash characters still do count in the total length of the pathname string). For example, the pathname such as `//////dirname//subdirname///file` refers to exactly the same file as does the pathname `/dirname/subdirname/file.` A pathname with trailing slash characters at the *end* of a pathname (with no further characters after the last slash character) is equivalent to as if the pathname ended in the final component "`.`" (described below) following the last slash character. An *empty* pathname (a null string) should result in an error being returned for any Yalnix file system operation attempting to use such an empty pathname.

When processing a pathname, e.g., as part of an `Open` operation, you should process the pathname one *component* at a time. A *component* of a pathname is the name that occurs between two slash characters (or before the first slash character, or after the last slash character). As noted above, processing begins either with the root inode (for an absolute pathname) or at the inode for the calling process's current directory (for a relative pathname). This starting inode defines the initial "*current lookup inode*" for this pathname. For each next component of the pathname, beginning with the leftmost component and working component by component across the pathname, processing of that component is performed relative to the then-current lookup inode in the processing of that pathname. If that component is a directory name, for example, the current lookup inode becomes the inode for that directory (if that directory itself was found), and processing

of the following component of the pathname proceeds with the inode for that directory as the current lookup inode.

A pathname, when presented as an argument to a Yalnix file system call is represented as a normal C-style string, terminated by a null character. The maximum length of the entire pathname, *including* the null character, is limited to `MAXPATHNAMELEN` characters. This limit of `MAXPATHNAMELEN` characters applies *only* to the length of the pathname string when presented as an argument to a Yalnix file system call. The fact of whether this pathname is an absolute pathname or a relative pathname, or the possible presence of symbolic links encountered while processing this pathname, do *not* count against that limit of `MAXPATHNAMELEN` characters. The limit of `MAXPATHNAMELEN` characters literally applies *only* to the argument of the call itself.

Within each directory, two special directory entries must exist:

- "`.`" (dot) : This directory entry has the name "`.`" and the inode number of the directory within which it is contained.
- "`..`" (dot dot) : This directory entry has the name "`..`" and the inode number of the parent directory of the directory within which it is contained. In the root directory, the "`..`" entry instead has the same inode number as "`.`" (the inode number of the root directory, which is defined as `ROOTINODE` in `filesystem.h`).

The "`.`" and "`..`" entries are created in a directory when it is created (by the `MkDir` request) as the *first two entries* in the new directory. These two directory entries subsequently cannot be explicitly deleted, but are automatically deleted along with the rest of the directory on a successful `RmDir` request.

2.5. Symbolic Links

The Yalnix file system supports symbolic links, as in the Unix file system. A symbolic link to some other file is represented in the Yalnix file system by an inode of type `INODE_SYMLINK`; the format of this file is otherwise the same as an `INODE_REGULAR` file. However, the *contents* of this file (the data stored in the data blocks hanging off of this inode) is interpreted by the file system as the *name* of the file to which this symbolic link is linked.

The file name to which a symbolic link points may be either an *absolute* pathname or a *relative* pathname. If a relative pathname, it is interpreted *relative to the directory in which this symbolic link file itself occurs*; that is, the processing of the symbolic link target begins with the *current lookup inode* (see Section 2.4) being the inode of the directory in which the symbolic link itself was found. For example, consider the pathname "`/a/b/c`", where "`b`" within this pathname is a symbolic link to the relative pathname "`x/y`". Since the target of the symbolic link "`b`" in this example is a *relative* pathname, "`x/y`", the search for "`x`" when, for example, attempting to Open the name "`/a/b/c`", begins in the *same* directory in which the name "`b`" itself was found. Thus, attempting to Open the file "`/a/b/c`" is ultimately "equivalent" to attempting to Open the name "`/a/x/y/c`". If, instead, "`b`" within the pathname "`/a/b/c`" is a symbolic link to the *absolute* pathname "`/p/q`", then attempting to open the file "`/a/b/c`" is then "equivalent" to attempting to open the name "`/p/q/c`".

As another example, suppose that:

- You are attempting to Open the pathname "`/a/b/c`".
- In doing this Open, you find that the name "`b`" within this pathname is a symbolic link to "`d/e`".
- You further find that the name "`e`" is a symbolic link to "`/f/g/h`".
- And you finally find that the name "`c`" is a symbolic link to "`j`".

The combined effect of attempting to Open this original pathname and encountering these symbolic links during processing that Open attempt is "equivalent" to attempting to Open the pathname "`/f/g/h/j`".

However, to process a pathname such as in these examples, in which you might encounter one or more symbolic links during processing that pathname, you should *not* attempt to build up the complete pathname that the original name is "equivalent" to. Rather, as with *any* pathname, you should process each component of the pathname one at a time. If you encounter a symbolic link, you should *recurse*, attempting to first process the target of the symbolic link, before resuming processing what remains of the name you were processing when you encountered *that* symbolic link. That is why I wrote "equivalent" in quotes above, when saying that attempting to Open one pathname is "equivalent" to attempting to Open some other pathname. Attempting to Open the first pathname is not *literally* the same as attempting to Open the second, but the final effect in terms of which file ends up being Opened is the same. Processing a symbolic link as part of processing a single pathname in this way is referred to as a *symbolic link traversal*.

When creating a symbolic link, it is *not* an error if the target of the new symbolic link does not exist. Indeed, when creating a symbolic link, the target is simply recorded as-is, without any processing, except that it is an error to attempt to create a symbolic link to an empty pathname (a null string). Furthermore, if the target of an existing symbolic link is later deleted, this is *not* an error. In both cases, such a "dangling" symbolic link is not a problem; however, if such a dangling symbolic link is encountered while processing some other pathname (such as during an `Open` operation), an error would be returned upon encountering the dangling symbolic link.

Also, note that, whereas it is an error to attempt to create a hard (regular) link to to a directory, creating a *symbolic* link to a directory *is* allowed; this is *not* an error.

Within the complete processing of a *single* pathname *passed as an argument to any Yalnix file system operation*, the maximum number of symbolic link traversals that may be performed is limited to `MAXSYMLINKS` symbolic link traversals (defined in `filesystem.h`). If, in processing any single pathname that was an argument to any Yalnix file system operation, you would need to traverse more than `MAXSYMLINKS` symbolic links, you should terminate processing of that pathname and instead return an error as the result of that Yalnix file system operation.

Lastly, *note the following special exception to handling a symbolic link when looking up a pathname*, with respect to the *last* component of that pathname: If the *last* component of the pathname is the name of a symbolic link, then that symbolic link *must not* be traversed *unless* the pathname is being looked up for an `Open`, `Create`, or `ChDir` file system operation (see the definition of the file system operations in Section 4.2). For example, if the last component of the pathname passed to an `Unlink` operation is the name of a symbolic link, then the symbolic link itself should be removed, but if the last component of the pathname passed to an `Open` operation is the name of a symbolic link, then the file to which the symbolic link refers should be opened, *not* the symbolic link itself.

2.6. Formatting a File System on the Disk

The hardware disk is initially empty, just as it would arrive if purchased new from the manufacturer. Before Yalnix can access a file system on this disk, the file system must first be *formatted* on the disk. Formatting a file system means to write the necessary disk data structures onto the disk to initially create a valid file system layout on the disk.

We provide a Unix (rather than a Yalnix) program to create an empty, validly-formatted Yalnix file system on the disk. To use this program, execute the Unix program

```
/clear/courses/comp421/pub/bin/mkyfs
```

(Run this program just as shown above from a Unix shell; do not run it under Yalnix.) This will create a YFS file system with 47 inodes (6 blocks worth of inodes). If you want a different number of inodes, put

the number of inodes as the command line argument. The file system will contain only a root directory with "`.`" and "`..`" in it. Run `mkyfs` as a *Unix command*, not under Yalnix, from the same directory where you will run Yalnix.

If you want to, you can modify this `mkyfs.c` program to set up test cases for yourself; the source code to this program is in the file `/clear/courses/comp421/pub/samples-lab3/mkyfs.c`. For example, before you get the `MkDir` file system request working correctly in your server, you can modify `mkyfs.c` to make test directories for yourself. Run that version of `mkyfs` before you then run your YFS server under Yalnix, and you could then test looking up files in directories and subdirectories, even before you get `MkDir` working in your server. This is just one example of what you could do to test some things in your server before other things in your server are working correctly yet. Since `mkyfs.c` runs as a Unix program, you can create anything in the Yalnix disk that you want to, and then run your server and see what your server can do with the contents that it finds on the disk.

3. New Yalnix Kernel Calls

For this project, a Yalnix kernel executable will be provided for your use. This Yalnix kernel has been enhanced with the addition of support for reading and writing individual sectors of the disk. The disk is accessed by identifying the sector number of the sector to be read or written, and supplying the address of a sector-sized buffer. The following two kernel calls are available for accessing the disk:

- `int ReadSector(int sectornum, void *buf)`

  Initiate a read of disk sector number `sectornum` into the buffer at address `buf`. The buffer `buff` must be of size `SECTORSIZE` bytes. The calling process is blocked until the disk read completes. `ReadSector` returns the value 0 if the operation was completed successfully; however, if the indicated sector number is invalid or the indicated buffer in memory cannot be written, `ReadSector` returns `ERROR`.

- `int WriteSector(int sectornum, void *buf)`

  Initiate a write of disk sector number `sectornum` from the buffer at address `buf`. The buffer `buff` must be of length `SECTORSIZE` bytes. The calling process is blocked until the disk write completes. `WriteSector` returns the value 0 if the operation was completed successfully; however, if the indicated sector number is invalid or the indicated buffer in memory cannot be read, `WriteSector` returns `ERROR`.

These kernel calls wait for the read or write to be completed before returning. For this project, you may assume that the disk hardware is perfectly reliable, and that all read and write operations with valid arguments will eventually complete and return, unblocking the calling process; you may assume that no disk hardware errors are possible during the read or write.

The provided Yalnix kernel executable has also been enhanced with additional Yalnix kernel calls for inter-process communication. You may find some or all of these new IPC calls useful in the project:

- `int Register(unsigned int service_id)`

  Registers the calling process as providing the service `service_id`. The Yalnix kernel does not enforce that the process actually provides the service and does not know to what specific service a particular `service_id` value refers. The `service_id` values are abstract and are chosen by convention. After a process is registered with `Register`, it is referred to as a "server" for this service. Other processes can then use the `Send` kernel call to send messages to it by specifying this `service_id` value instead of the process ID as the destination of the message sent. Returns 0 on

success; returns `ERROR` on any error, including if another current process is already registered for that `service_id`. (As defined below in Section 4.3, your YFS server should register itself for the `FILE_SERVER` service id.)

- `int Send(void *msg, int pid)`

  The argument `msg` points to a fixed-sized 32-byte buffer holding a message to be sent. The `Send` kernel call sends this message to the process indicated by the argument `pid`. The calling process is blocked until a reply message sent with `Reply` is received. *Note that the* `Send` *kernel call always sends exactly 32 bytes, beginning at address* `msg` *as the message. Likewise, the reply message sent with* `Reply` *always overwrites exactly 32 bytes, beginning at address* `msg`. If the value `pid` is positive, it is interpreted as the process ID of the process to which to send the message. If the value `pid` is negative, it is interpreted instead as an indication of the service to which to send the message, and the message is sent to the process currently registered with `Register` as providing service `-pid`. On success, the call returns 0; on any error, the value ERROR is returned.

- `int Receive(void *msg)`

  The argument `msg` points to a fixed-sized 32-byte buffer in the calling process's address space. The `Receive` kernel call receives the next message sent to this process with `Send`, copying that message into the buffer at `msg`. If no unreceived such message has been sent yet, the calling process is blocked until a suitable message is sent. *Note that the* `Receive` *kernel call always receives (and thus overwrites) exactly 32 bytes, beginning at address* `msg`. On success, the call returns the process ID of the sending process; on any error, the value ERROR is returned.

- `int Reply(void *msg, int pid)`

  The argument `msg` points to a fixed-sized 32-byte buffer holding a reply message to be sent by the calling process. The `Reply` kernel call sends this reply message to the process with process ID `pid`, which must be currently blocked awaiting a reply from an earlier `Send` to this process. The reply message from the calling process overwrites the original message sent in the address space of the process with process ID `pid` (indicated by the `msg` pointer passed by that process on its earlier call to `Send` that is currently blocked awaiting this reply). *Note that the* `Reply` *kernel call always replies with exactly 32 bytes, beginning at address* `msg`. On success, the call returns 0; on any error, the value ERROR is returned.

- `int CopyFrom(int srcpid, void *dest, void *src, int len)`

  This call copies `len` bytes beginning at address `src` in the address space of process `srcpid`, to the calling process's address space beginning at address `dest`. The process `srcpid` must be currently blocked awaiting a reply from an earlier `Send` to the calling process. On success, the call returns 0; on any error, the value `ERROR` is returned.

- `int CopyTo(int destpid, void *dest, void *src, int len)`

  This call copies `len` bytes beginning at address `src` in the address space of the calling process, to the address `dest` in the address space of process `destpid`. The process `destpid` must be currently blocked awaiting a reply from an earlier `Send` to the calling process. On success, the call returns 0; on any error, the value `ERROR` is returned.

As noted above in the description of the `Send`, `Receive`, and `Reply` kernel calls, *all messages are always exactly 32 bytes in length*. Thus, for example, the following code is *incorrect*:

```
int msg = 12345;
Send((void *)&msg, pid);
```

This *incorrect* code sends 32 bytes, but only 4 bytes (the size of an integer) are supplied. The remaining $(32 - 4) = 28$ bytes that are sent are whatever unknown data is in memory immediately following the `msg` integer variable. In addition, when a `Reply` is sent, these unknown 28 bytes will be overwritten, since the reply message is also always 32 bytes in length.

Similarly, the following code is *incorrect*:

```
Send((void *)"hello world", pid);
```

This *incorrect* code sends 32 bytes, but only 12 bytes (the length of the string "`hello world`", including the null character at the end of the string) are supplied. The remaining $(32 - 12) = 20$ bytes that are sent are whatever unknown data is in memory immediately following this character string. In addition, when a `Reply` is sent, these unknown 20 bytes will be overwritten, since the reply message is also always 32 bytes in length. Furthermore, the 12 bytes of the character string may be overwritten, since the `Reply` overwrites the entire original 32 bytes: depending on the compiler and options used in compiling the program, the character string "`hello world`" may be in read-only memory, in which case the `Reply` will fail; in other cases, in which the compiler places this string in writable memory, the "`hello world`" character string (which is supposed to be a *constant*) will actually get *overwritten* by the reply message.

Instead, the best (*correct*) way to use the `Send`, `Receive`, and `Reply` kernel calls is to define a `struct` of length 32 bytes as your message. For example, defining the following `struct` for a message would be a correct message:

```
struct my_msg {
    int data1;
    int data2;
    char data3[16];
    void *ptr;
};
```

The size of each `int` is 4 bytes (32 bits), and the size of the `char` array here is 16 bytes. The size of the `ptr` pointer is 8 bytes (64 bits). The total size of this `struct` is 32 bytes. You could, for example, define a single generic `struct` such as this to suite your needs for *all* messages, or you could define a different `struct` for each type of message you need to send. It is recommended that you put a "type" value (such as an `int` or `short` or `char` value) as the first thing in every message, so that the receiver can look at this value to determine the format of the rest of the message.

Also, be aware that the C compiler will insert "padding" into your structure if needed in order to keep each member of the structure aligned on a natural boundary for its size. For example, the compiler will ensure that a `short` is aligned on a multiple of 2 boundary from the beginning of the structure, an `int` is aligned on a multiple of 4 boundary from the beginning of the structure, and any pointer is aligned on a multiple of 8 boundary from the beginning of the structure. Also, the compiler will insert padding at the end of the structure to ensure that the total size of the structure is a multiple of 8 in size. Thus, if the members of the structure above were simply reordered as follows

```
struct my_msg {
    int data1;
    void *ptr;
    int data2;
    char data3[16];
};
```

then the total size of this structure will no longer be 32 bytes. Instead, the compiler will insert 4 bytes of padding before the pointer `ptr` member, and will insert another 4 bytes of padding at the end of the structure; thus, in this case, the total size of the message would be 40 bytes. This message definition *will not* work as you might intend with Yalnix message passing operations, since only the first 32 bytes of it will end up actually being sent.

You are strongly advised to confirm the actual size of any of your message structure definitions by using the C language's `sizeof` operator to determine the size that the compiler actually thinks your structure is.

4. The Yalnix File System Server

4.1. Overview

The Yalnix File System operates as a server executing as a regular user process outside the Yalnix kernel. Other processes using the file system send *requests* to the server and receive *replies* from the server, using the message-passing interprocess communication calls provided by the Yalnix kernel.

When making a file system request such as `Open` or `Read`, a user process calls a library procedure known as a *stub* function, which in general packages the appropriate arguments into a request message and sends this message to the file server server. Sending this message (with the Yalnix `Send` kernel call) also blocks the requesting process until a reply (from `Reply`) from the server is received. The server executes the request and sends a reply message back to the requesting process when the request has completed. There is a *separate* copy of the library linked into each Yalnix user program that uses the file server.

The YFS server process retains *no state* on behalf of individual client processes or open files. Thus, the file server process has no knowledge of specific file descriptor numbers that represent specific open files in specific processes. Instead, all state about a particular open file is retained within the copy of the file system library that is linked in to the client process itself; any necessary state is passed in the message to the server on each file system request. This allows the server to operate without worrying whether the client process that opened any individual file still exists, greatly simplifying the design of the file server.

The file system server process maintains a cache in memory of recently accessed inodes and recently accessed data blocks. These caches are each of a *constant* size, defined by the following two constants from `filesystem.h`:

- `BLOCK_CACHESIZE`: The cache of recently accessed disk blocks in the file server must be of *constant* size `BLOCK_CACHESIZE` blocks. Initially, all blocks in the cache are unused, but as new blocks are accessed, your server must manage the constant number of blocks in the cache using a *write-back* LRU policy.

- `INODE_CACHESIZE`: The cache of recently accessed inodes in the file server must be of *constant* size `INODE_CACHESIZE` inodes. Initially, all inodes in the cache are unused, but as new inodes are accessed, your server must manage the constant number of inodes in the cache in a *write-back* LRU policy.

Again, the block cache and the inode cache *must* each be a *constant* size, as described above. If a new item must be brought into the cache, you must decide which existing entry in the cache to replace to make room for it in the cache.

While executing a file system request from some client process, the file system server will generally need to read or write a number of disk blocks or inodes. You will need to maintain a hash table to allow a block in the cache (if present) to be found quickly given the block's disk block number. Similarly, you will need to maintain a (separate) hash table to allow an inode in the cache (if present) to be found quickly given the inode's inode number. The cache management policy is *write-back* in that dirty blocks (or inodes) are

left in the cache (marked "dirty") until that cache block (or cache inode) is needed for holding a different disk block (or inode). At that time, the cached value is written back to disk and the new value is read into that space in the cache. When writing a cached disk block back to disk, you will need to use the `WriteSector` kernel call, but when writing an inode back to disk, you should simply write it back to the disk block cache for the disk block in which that inode lives; you leave this disk block in the cache marked "dirty," and actually write it out to the disk later, when you need that space in the block cache for a different disk block.

4.2. User Process File System Operations

A user process using the YFS file system makes file system requests by calling one of the procedures defined within your file system library. The library remembers the current directory of the process and maintains information about each file open by the process. The file system library interacts with the file server process through the Yalnix IPC facilities. The interface to the file system library is defined in the C include file `iolib.h`. This header file is located in

```
/clear/courses/comp421/pub/include/comp421/iolib.h
```

and can be include in your source file using

```
#include <comp421/iolib.h>
```

When opening a file with either the `Open` or `Create` request, your *library* must allocate a data structure to remember information about the open file. Specifically, in your library, you will need to remember

- the file's inode number, and
- the current position within the file.

The library must support up to a maximum of `MAX_OPEN_FILES` open files at a time. For an `Open` or `Create` request, your file system library learns the file's inode number from the file server process (the current position within the file is initialized to 0) and stores this within the data structure in the library representing that open file. This information is then available for later accesses to this open file.

You should maintain an *open file table* within the *library* in the user process, giving a pointer to the data structure representing each open file. Since you need to support only `MAX_OPEN_FILES` open files at a time, this table can efficiently be implemented as a static array, dimensioned at `MAX_OPEN_FILES`, with each array entry containing a pointer to the corresponding data structure representing that open file. The index in this array then becomes that open file's *file descriptor number* and is returned as the result of the (successful) `Open` or `Create` request. The user process then uses this file descriptor number on future calls (such as `Read` and `Write`) to refer to this open file.

The file descriptor numbers used by a process are assigned by and managed by the copy of your library linked into the user program running in that process. The file descriptor number is part of the procedure call interface between a user program and the copy of your library linked into that program; the file descriptor number is not part of the message interface between your library and your file server process.

Your library, together with your file server process, must support the following procedure call requests, with the indicated procedure names, arguments, and return values:

- `int Open(char *pathname)`

  This request opens the file named by `pathname`. If the file exists, this request returns a file descriptor number that can be used for future requests on this open file. If the file does not exist, or if any other error occurs, this request returns `ERROR`. It is *not* an error to `Open()` a directory; the contents of the open directory can the read using `Read()`, although it *is* an error to then attempt to `Write()` to the open directory. If the `Open` is successful, the current position for `Read` or `Write` operations on this file descriptor begins at position 0.

- `int Close(int fd)`

  This request closes the open file specified by the file descriptor number `fd`. If `fd` is not the descriptor number of a file currently open in this process, this request returns `ERROR`; otherwise, it returns 0.

- `int Create(char *pathname)`

  This request creates and opens the new file named `pathname`. All directories in the named `pathname` must already exist; the `Create` request creates only the last file name component in the indicated `pathname`. If the named file already exists, this request instead truncates the size of the existing file to 0 and opens the now empty file. On success, this request returns a file descriptor number that can be used for future requests on this open file. Otherwise, this request returns `ERROR`. If the `Create` is successful, the current position for `Read` or `Write` operations on this file descriptor begins at position 0.

- `int Read(int fd, void *buf, int size)`

  This request reads data from an open file. The argument `fd` specifies the file descriptor number of the file to be read, `buf` specifies the address of the buffer in the requesting process into which to perform the read, and `size` is the number of bytes to be read from the file. This request returns the number of bytes read, which will be 0 if reading at the end-of-file; the number of bytes read will be the minimum of the number of bytes requested and the number of bytes remaining in the file before the end-of-file. Any other error should return `ERROR`. It is *not* an error to attempt to `Read()` from a file descriptor that is open on a directory. Unless this `Read` operation returns `ERROR`, the current position for subsequent `Read` or `Write` operations on this file descriptor advances by the number of bytes read (the value returned by the `Read` request).

- `int Write(int fd, void *buf, int size)`

  This request writes data to an open file. The argument `fd` specifies the file descriptor number of the file to be written, `buf` specifies the address of the buffer in the requesting process from which to perform the write, and `size` is the number of bytes to be written to the file. This request returns the number of bytes written. Any error should return `ERROR`. It *is* an error to attempt to `Write()` to a file descriptor that is open on a directory. Unless this `Write` operation returns `ERROR`, the current position for subsequent `Read` or `Write` operations on this file descriptor advances by the number of bytes written (the value returned by the `Write` request).

- `int Seek(int fd, int offset, int whence)`

  This request changes the current file position of the open file specified by file descriptor number `fd`. The argument `offset` specifies a byte offset in the file relative to the position indicated by `whence`. The value of `offset` may be positive, negative, or zero. The value of whence must be one of the following three codes defined in `iolib.h`:

  - `SEEK_SET`: Set the current position of the file to be `offset` bytes from the beginning of the file. The `offset` must be greater than or equal to zero in this case.
  - `SEEK_CUR`: Set the current position of the file to be `offset` bytes after the current position in the open file.
  - `SEEK_END`: Set the current position of the file to be `offset` bytes after the end of the file. The `offset` must be less than or equal to zero in this case.

It is an error if the seek attempts to go before the beginning of the file or past the end of the file. This request returns as its result the new position in the open file, unless an error is encountered. As an example, `Seek(fd, 0, SEEK_END)` seeks to the end of the file and returns the file size.

- `int Link(char *oldname, char *newname)`

  This request creates a (hard) link from the new file name `newname` to the existing file `oldname`. The files `oldname` and `newname` need not be in the same directory. The file `oldname` must not be a directory. It is an error if the file `newname` already exists. On success, this request returns 0; on any error, the value `ERROR` is returned.

- `int Unlink(char *pathname)`

  This request removes the directory entry for `pathname`, and if this is the last link to a file, the file itself should be deleted by freeing its inode. The file `pathname` must not be a directory. On success, this request returns 0; on any error, the value `ERROR` is returned.

- `int SymLink(char *oldname, char *newname)`

  This request creates a symbolic link from the new file name `newname` to the file name `oldname`. The files `oldname` and `newname` need not be in the same directory. It is an error if the file `newname` already exists. The file `oldname` need not currently exist in order to create a symbolic link to this name. Indeed, when creating a symbolic link, the target is simply recorded *as-is*, without any processing, except that it is an error to attempt to create a symbolic link to an empty pathname (a null string). On success, this request returns 0; on any error, the value `ERROR` is returned.

- `int ReadLink(char *pathname, char *buf, int len)`

  This request reads the name of the file that the symbolic link `pathname` is linked to. On success, this request returns the length (number of characters) of the name that the symbolic link `pathname` points to (or the value `len`, whichever is smaller), and places in the buffer beginning with address `buf` the name that the symbolic link points to, up to a maximum number of characters of `len` characters; if the name that the symbolic link points to is longer than `len` bytes, the name is truncated as returned in the buffer `buf` (this is not an error). The characters placed into `buf` are *not* terminated by a `'\0'` character. On any error, the value `ERROR` is returned.

- `int MkDir(char *pathname)`

  This request creates a new directory named `pathname`. The "`.`" and "`..`" entries in the directory are automatically created. It is an error if the file `pathname` exists. On success, this request returns 0; on any error, the value `ERROR` is returned.

- `int RmDir(char *pathname)`

  This request deletes the existing directory named `pathname`. The directory must contain no directory entries other than the "`.`" and "`..`" entries and possibly some free entries. On success, this request returns 0; on any error, the value `ERROR` is returned.

- `int ChDir(char *pathname)`

  This request changes the current directory within the requesting process to be the directory indicated by `pathname`. The current directory of a process is remembered within the file system library in that process, and is passed to the file server on each request that takes any file name arguments. The

file `pathname` must be a directory. On success, this request returns 0; on any error, the value `ERROR` is returned.

- `int Stat(char *pathname, struct Stat *statbuf)`

  This request returns information about the file indicated by `pathname` in the information structure at address `statbuf`. The information structure is defined within `iolib.h` as follows:

```
struct Stat {
    int inum;           /* inode number of file */
    int type;           /* type of file (e.g., INODE_REGULAR) */
    int size;           /* size of file in bytes */
    int nlink;          /* link count of file's inode */
};
```

  The fields in the information structure are copied from the information in the file's inode. On success, this request returns 0; on any error, the value `ERROR` is returned.

- `int Sync(void)`

  This request writes all dirty cached inodes back to their corresponding disk blocks (in the cache) and then writes all dirty cached disk blocks to the disk. The request does not complete until all dirty inodes and disk blocks have been written to the disk; this request always then returns the value 0.

- `int Shutdown(void)`

  This request performs an orderly shutdown of the file server process. All dirty cached inodes and disk blocks should be written back to the disk (as in a `Sync` request), and the file server should then call `Exit` to complete its shutdown. As part of a `Shutdown` request, the server should print an informative message indicating that it is shutting down. This request always returns the value 0.

4.3. Running Yalnix, the Server, and User Processes

For your use in this project, we have provided an enhanced Yalnix kernel executable that supports access to the disk and IPC calls, as described in Section 3. You should implement your server process to run on top of this Yalnix kernel. The provided Yalnix kernel is located on CLEAR at

```
/clear/courses/comp421/pub/bin/yalnix
```

Do not copy this program to your own directory. To use this Yalnix kernel, you may either run it using the full pathname above, or you may put the `/clear/courses/comp421/pub/bin` directory on your shell's executable search path and can then run this kernel as just `yalnix`.

As in Lab 2, the Yalnix kernel automatically starts only one process (other than the idle process) at boot time. This process, commonly known as the "init" process, must then use `Fork` and `Exec` to create any other processes or run any other programs, as needed.

For this project, you should execute your YFS server process as the Yalnix "init" process. As part of its initialization, the YFS server should use the `Register` kernel call to register itself as the `FILE_SERVER` server. It should then `Fork` and `Exec` a first client user process. In particular, like any C `main()` program, your file server process is passed an `argc` and `argv`. If this `argc` is greater than 1 (the first argument, `argv[0]`, is the name of your file server program itself), then your server should `Fork` and the child process should then `Exec` the first client user program, as:

```
Exec(argv[1], argv + 1);
```

This uses the second argument as the name of the process to execute, and passes this and the remaining arguments to that process as its command line arguments.

For example, to execute the Yalnix kernel, with your file server as the kernel "init" process, and a program `testprog` as the first client user process, you should use the shell command:

```
yalnix yfs testprog testarg1 testarg2 testarg3 ...
```

where *testarg1*, *testarg2*, *testarg3*, etc., are command line arguments to give in `argv` to the the program `testprog`. The program `testprog` can then create any additional processes itself as needed using `Fork` and `Exec`. In essence, this `testprog` program serves the role played by the init process in Lab 2.

Within your file system library, a client process can send a message to your file server process using the `Send` kernel call by sending a message to the `FILE_SERVER` service id:

```
Send(msg, -FILE_SERVER);
```

Since the file server process registers itself as the `FILE_SERVER` service id, this message will be sent to your file server process without client process needing to know the real process id of the file server.

As in Lab 2, you can use `TracePrintf` to help with debugging your YFS server process. The `TracePrintf` call is used the same as it was in Lab 2:

```
TracePrintf(int level, char *fmt, args...)
```

where `level` is an integer tracing level, `fmt` is a format specification in the style of `printf`, and `args...` are the arguments needed by `fmt`. The tracing level for the YFS server process is independent of the tracing level for the hardware, kernel, or regular user processes. To set the tracing level for the YFS server process, add "`-ly` *level*" to the `yalnix` command line. For example, using

```
yalnix -ly 5 yfs testprog testarg1 testarg2 testarg3 ...
```

sets the file server `TracePrintf` level to 5. You may also include any of "`-lh` *level*", "`-lk` *level*", and/or "`-lu` *level*" on the `yalnix` command line to set, respectively, the hardware, kernel, and/or user `TracePrintf` tracing level, as well. The `-ly` level setting affects `TracePrintf` calls made by your Yalnix server process.

5. Your Assignment

5.1. General Specification

You are to implement the Yalnix file system, consisting of *both* of the two parts described below:

- A *YFS file server process*. You must implement a Yalnix user program named `yfs`. This process receives messages from client processes, executes the requested file system operations, and returns reply messages to the requesting processes.
- A *YFS file system library*. You must implement a Unix library archive file named `iolib.a`. This library defines a procedure for each of the file system requests defined in Section 4.2. Each of these procedures communicates with the server process using the Yalnix IPC kernel calls in order to request the server process to perform the necessary file system operation, and finally returns a status return value as the result of the library call.

In other words, your *file server process* implements most of the functionality of the Yalnix file system, managing a cache of inodes and disk blocks and performing all input and output with the disk drive. Your *file system library* consists of a collection of procedures called by Yalnix user programs using the file server; the file system library maintains information on the current directory of the process and on each file open by that process, and uses the Yalnix IPC facility to communicate with the server. Mostly, each file system library procedure is just a stub procedure that formats and sends a request message to the Yalnix file system server process, which then completes that request and returns a reply message. Your project must execute on CLEAR using the provided enhanced Yalnix kernel.

The interface to your file server *library* consists of the procedures defined in Section 4.2. The format of the data that you must store on disk for the file system is defined in Section 2. You must design your own messages for communication between your file system library and file system server.

A template `Makefile` for this project is available as `Makefile.lab3.template` in the directory `/clear/courses/comp421/pub/templates`. You should copy this file to your project directory and edit it as described in the comments in the file. *In particular, this template* `Makefile` *contains special rules for compiling and linking programs, including your Yalnix file server* `yfs`*, which must be used for the project to work.*

The project should be done in groups of two students. This project requires a reasonable amount of work, so you will need to divide the load between the two group members. If you *really* want to, you may work on this project by yourself, without a partner, but I *strongly* recommend *against* this. All projects will be graded in the same way, regardless of whether you worked with a partner or not, so you are only making more work for yourself.

As with Labs 1 and 2, we will be using Piazza for class discussion (please use the "`lab3`" folder). *Please check Piazza regularly; you will likely find it very helpful in the project.* As before, when posting questions or answers about the project on Piazza, please be careful about what you post, to avoid inadvertently violating the COMP 421 course Honor Code policy described in the course syllabus and below in Section 5.3. *Specifically, please do not post details about your own project solution, such as portions of your source code or details of how your code works, in public questions or answers on Piazza.* If you need to ask a question that includes such details, please make your question private on Piazza by selecting "Instructor(s)" (rather than "Entire Class") for "Post to" at the top, so that only the course instructor and TAs can see your posting.

5.2. Submitting your Project for Grading

To submit your Lab 3 for grading when you have completed it, please run the program

```
/clear/courses/comp421/pub/bin/lab3submit
```

This is basically the same as how you submitted Lab 2. Before running this program, please make sure that your current directory is set to the directory where your files for the project for grading are located. This should include all files for both partners in a project group. When you run the submission program, it will submit everything in (and below) your current directory, including all files and all subdirectories (and everything in them, too).

Please also make sure that you have a file named "`README`" in this top-level directory, in which you include a statement of who your partner is on this project. Also in this file, please describe anything else you think it would be helpful for the TAs to know in grading your project.

Running the `lab3submit` program will check with you that you are in the correct directory that you want to submit for grading, and finally, will normally just print "`SUCCESS`" when your submission is complete. If you get any error messages in running lab2submit, please let me know.

5.3. Honor Code Policy

All assignments in this course are conducted under the Rice Honor Code. For programming assignments such as this one, students are encouraged to talk to each other, the TAs, the instructor, or anyone else about the assignment. This assistance, though, is limited to discussion of the problem; *each project group must produce their own solution*. Consulting another student's or group's solution (even from a previous COMP 421 class) is prohibited, and submitted solutions may not be copied from any source. For more information on the Rice Honor System, visit `http://honor.rice.edu/`.