A File System for the J-Machine [1]

Yair Zadik and Stephen Taylor
Scalable Concurrent Programming Laboratory
California Institute of Technology

July 7, 1993

1 Overview

This document describes the file system to be implemented for the MIT J-machine. When completed, the J-machine file system will provide all the functionality of a standard file system, including NFS support. Additionally, the file system and potential the language tools will incorporate specific optimizations for concurrent computing.

The J-machine hardware can be partitioned into three different environments: I/O nodes, compute nodes, and host. All three contain a Message Driven Processor and some memory. The I/O nodes also contain a SCSI disk interface and attached disks. The host is actually a SPARC workstation with a special interface card containing an MDP. The SPARC host will be connected to a LAN and provides the only access to the MDP mesh (including the J-machine disks) from the outside world.

The disk system is layered as follows:

1. Low level block read/write (I/O nodes)
2. Error correcting (RAID) block read/write (I/O nodes)
3. NFS-like and concurrent file system calls (all nodes)
4. Unix file system compatibility library (compute nodes) and network NFS support (SPARC host)

The low level and RAID block I/O are supplied by MIT and are fully described in the MIT documentation. The other layers are described in this document.

The NFS-like file system calls will be available at all nodes and provides a uniform method for accessing the file system for both the compute nodes and host nodes. This

[1]The research described in this report is sponsored primarily by the Advanced Research Projects Agency, ARPA Order number 8176, and monitored by the Office of Navel Research under contract number N00014-91-J-1986.

layer implements the file structure described in Appendix A and the file and directory manipulation calls which allow application programs, Unix compatibility library, and network NFS support to function. The concurrent file system calls will also be part of this layer, but are not as yet defined as are the subject of this research.

The highest layer of the system actually consists of two independent modules: a Unix compatibility library which implements Unix-like calls in terms of the NFS-like calls, and a host side NFS interface. The later will be a daemon which will replace the standard program "nfsd" on the SPARC host and implement the NFS RPC protocol by communicating through the MDP mesh to the I/O nodes. Any workstation on the network (including the SPARC host itself) will thus mount the J-machine disks as if they were on just another disk server. Standard commands can then be used to manipulate files and directories on the J-machine disks.

The Unix compatibility library allows current Unix programs (including the standard C library) to port easily to the J-machine files system. This library is implemented on top of the NFS-like file systems calls. NFS like calls were provided as primitives instead of using calls because NFS calls are stateless and are thus more appropriate for parallel use, and NFS calls must be implemented anyhow for access across a LAN.

The full functionality of the file system will not be available with the initial system. The intention is to provide all the functionality of a standard file system as soon as possible (reserving optimization and experimentation with concurrency for later) so that the disks system may be used as soon as possible by the applications writers. To facilitate this, the file system will be brought up in stages:

1. A flat file system with fixed limits on number of files and no support for manipulation of the file system from the host. This limited file system must support at least two files per node: one for input data and another for checkpoint output.

2. Host file system manipulation. This will require the S-Bus card to function properly, since the transfer of data from host to J-machine and back will function differently once the S-Bus card is available.

3. Network file system manipulation. This requires writing an NFS server that will use the previous stage.

4. Hierarchical directory structure. This stage will also define a flexible directory file format and remove the limitation on number of files per directory.

5. Experimentation with the layout of files on the disk, file system calls, and integration with the compilation to improve efficiency.

Note that not all these stages are interdependent so that the order of these stages may change depending on the availability of the S-Bus card and the usability of the system without it.

2 File System Calls

System calls will be implemented in stages:

1. basic NFS-like directory manipulation: create, remove, readdir
2. basic NFS-like file manipulation: lookup, read, write, setattr (to set size)
3. basic Unix-like calls: open, read, write, lseek, tell, close, creat
4. implement protection mechanisms and related NFS-like (get/set file attributes) and Unix-like calls (chmod, fstat)
5. more complicated directory manipulation: rename, link
6. hierarchical directory manipulation: mkdir, rmdir
7. optionally symbolic links: symlink, readlink

A description of these routines follows. The NFS-like routines depart from NFS primarily by using multiple arguments instead of one structure argument and by providing a pointer to a space for the returned value when it is more than a status.

2.1 Types

- fs_fhandle - object of size FS_FHSIZE used by the operating system to store any necessary information on location and state of a file. This will be defined as "char [FS_FHSIZE]" outside of file system code and as a structure within the file system code.
- fs_cookie - object of size FS_COOKIESIZE which is used to pass information between subsequent calls to readdir
- filename - null terminate string not to exceed FS_MAXNAMLEN in size
- fs_time - date time information structure. Contains unsigned integers seconds and useconds which are the number of seconds and microseconds since midnight January 1, 1970 GMT.
- fs_sattr - settable file attributes structure. Contains unsigned integers mode, uid, gid, size and "fs_time" atime (last access time), mtime (last modification time). A value of MAX_UINT in any of the fields means ignore it.
- fs_fattr - file attributes structure. Contains type (non-file, regular, directory, block-special (unused), character special (unused), or symbolic link) and the integers mode, nlink (number of links), uid, gid, size, blocksize (of block in file), rdev (unused), blocks (used by file), fsid (file system identifier, unique per partition), fileid (unique for file), atime (access time), mtime (modification time), and ctime (creation time).

- fs_status - enumerated type which may be OK or some error.

3 File System Calls (NFS-like)

```
fs_status fs_create(fs_fhandle dir, char *filename,
            fs_sattr new_attributes)
```

Create a new file in directory pointed to by "dir" named "name" with file attributes "new_attributes". The value of "dir" should be all zero (for the logical root of the file system) or a value returned from lookup on a directory.

```
fs_status fs_remove(fs_fhandle dir, char *filename)
```

Remove the file in directory "dir" named "name".

```
fs_status fs_readdir(fs_fhandle dir, fs_cookie next, unsigned maxsize,
                  entry *entries, boolean *eof)

struct fs_entry {
   unsigned fileid;
   char *name;
   fs_cookie cookie;
   fs_entry *next_entry;
}
```

Read entries of up to "maxsize" bytes from the directory "dir" starting at "next". The "next" may be all zero bits (indicating start at the beginning) or a value retrieved from the "cookie" field of an entry within the directory. Entries are returned in the buffer pointed to by entries which must be at least maxsize bytes long. The filename within each entry will also be within this buffer. The "next_entry" field will be zero at the end of the list. "eof" will contain true if the end of the directory was encountered while filling "entries".

```
fs_status fs_lookup(fs_fhandle dir, char *filename, fs_fhandle *file,
                 fs_fattr *attributes)
```

Lookup a file "name" in a directory "dir" and return a file handle to it and its attributes. The file handle may be used in readdir (if its a directory) or in read or write (if its a file).

```
fs_status fs_lookup_path(fs_fhandle cwd, pathname path,
                      fs_fhandle *file, fs_fattr *attributes)
```

This is the same as lookup except that the file is specified using a handle to the current working directory and a Unix pathname (i.e. "." is current directory, ".." is previous directory, "/" seperates file names unless it is the first character in the path, in which case it is the logical root of the file system.) NFS does not define anything like this because the definition of "path" varies widely from system to system (compare VMS, MSDOS, and Unix for example). However, this is a more convient form for most uses (including implementing open) since it hides the path parsing steps.

```
fs_status fs_read(fs_fhandle file, unsigned offset,
               unsigned count, unsigned *ret_count,
               char *data, fs_fattr *attributes)
```

Read "count" bytes beginning at "offset" bytes from the start of "file" and return them in data. File attributes after the read are returned in "attributes". The actual number of of bytes read is returned in ret_count, which is less than count if an error occurs or the end of file is reached.

```
fs_status fs_write(fs_fhandle file, unsigned offset,
                char *data, unsigned count,
                unsigned *ret_count,
                fs_fattr *attributes)
```

Write "count" bytes beginning at "offset" bytes from the start of the "file." Actual number of bytes written is via ret_count and file attributes after the write are returned in attributes.

```
fs_status fs_getattr(fs_fhandle file,
          fs_fattr *attributes)
```

Get the file attributes for "file" and return them in attributes.

```
fs_status fs_setattr(fs_fhandle file, fs_sattr change, fs_fattr *new)
```

Set the attributes of file "file" according to "change" and return the attributes of the file after the set in "new".

```
fs_status fs_mkdir(fs_fhandle dir, char *filename,
                   fs_sattr new_attributes)
```

Create a new directory "name" with file attributes "new_attributes" in the directory "dir". The value of "dir" should be all zero (for the logical root of the file system) or a value returned from fs_lookup or fs_lookup_path on a directory.

```
fs_status fs_rmdir(fs_fhandle dir, char *filename)
```

Remove the directory "name" in directory "dir". The value of "dir" should be all zero (for the logical root of the file system) or a value returned from lookup on a directory.

4 Unix compatibility library

`int open(char *pathname, int flags, int mode)`

Unix-like open. Returns a file descriptor to that may be used with other Unix-like calls to read and write files. flags is any of O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, or O_CREAT bitwise OR'd together. mode is the permissions for the file (rwx for group, user, other as in Unix chmod) and is only used during file creation. If an error occurs, -1 is returned and the global variable errno is set appropriatly.

`int read(int fd, char *buf, int nbyte)`

Read "nbyte" bytes from file with file descriptor "fd" into buffer "buf" which must be at least nbyte in size. The return values is the number of bytes actually read (if no error) or -1 (if an error occurs). If an error occurs, the global variable "errno" is set.

`int write(int fd, char *buf, int nbyte)`

Write "nbyte" bytes from buf to file with descriptor fd. Same return as read.

`long lseek(int fd, int offset, int whence)`

Position file pointer for the file with descriptor "fd" at "offset" bytes from "whence". "whence" may be SEEK_SET (beginning of file), SEEK_CUR (current position), or SEEK_END (end of file). The value returned is the position in the file after seek.

`long tell(int fd)}`

Returns current position in file.

`int close(fd)`

Close the file associate with file descriptor fd. This descriptor is no longer valid for use with read, write, or seek. Returns 0 on succes and -1 on error (with errno set.)

`int creat(char *path, int mode)`

Create the file path with permissions mode and return a write-only file descriptor to it. errno is set and a -1 is returned if an error occurs.

```
int stat(char *path, struct stat buf)
int lstat(char *path, struct stat buf)
int fstat(int fd, struct stat buf)
```

Return the file attributes in buf for the file with pathname path or file descriptor fd. "lstat" and "stat" differ in that the first will return information on the symbolic link instead of the file it points to if the path is the path of a symbolic link, so these two routines are identical until symbolic links are implemented . The "stat" structure is defined as in Unix. stat, lstat, and fstat return 0 on success and -1 on failure, with an error in errno.

```
int chmod(char *path, int mode)
int chown(char *path, int owner, int group)
int fchmod(int fd, int mode)
int fchown(int fd, int owner, int group)
```

Sets the permissions or owner/group of path or fd according to mode. Returns 0 on success, -1 on failure (and sets errno). pp

```
int mkdir(char *path, int mode)
```

Create a directory path with permissions mode. errno is set and a -1 is returned if an error occurs. A 0 is returned on success.

```
int rmdir(char *path)
```

Remove a directory path. errno is set and a -1 is returned if an error occurs. A 0 is returned on success. Only empty directories (one containing only its parent and itself) will be deleted.

A Disk Layout

This section describes the layout on disk of the initial file system. It includes provisions for a hiearchial directory structure which will not be implemented in the first file system, but will be present in subsequent versions. This section is provided for informational purposes only and the actual layout will change as needed.

The MIT provided block I/O system treats all physical disks in the system as as one logical disk with each (64K) block striped across all disks in disk module (40 disks: 32 data + 4 parity + 4 hot spares). This organization is something that should be examined after the first implementation of the files system to see if more parrallelism can be extracted from the system by altering this organization. Also, the MIT system will implement RAID as a error detection and correction system. Since this is present, it is not necessary to make redundant copies of critical information (like the free block list and the partition table).

Block 0 will be the file system information and free block list. It will start with a file system header which will provide the following information:

- A magic number and version information, including size of header

- Partition size and first block of next partition. This is to allow the system to be partitioned so that different versions of the file system may coexist to enable file system expermentation while applications work is being done.
- Physical block size
- Logical block size
- Name of the partition
- Logical block number of first directory block (root directory)
- Logical block number of first i-node block
- Logical block number of bad block list
- Sequence number of this block in free list: This is 0 for first block in free list, 1 in second block, etc.
- Reserved words for future expansion
- Checksum to ensure uncorrupted header

Following this, will be a one bit per block bitmap representing free space on the disk (i.e. bit set means disk block is free). Since the maximum capacity of the J machine is 64 disks, 8 parity, and 8 hot spares with each disk having a capacity of 422Mbytes, and the standard size of a block being 64Kbytes, this organization requires exactly one block for the free block list. However, the code should be written to calculate the number of blocks needed for the list from the information in the header. If more than one block is needed for the list (for instance, if we change the physical block size to 16kbytes or use 128 disks) subsequent free blocks duplicate the header except the sequence number is incremented. Typically, the block with the most free space will be cached so as to avoid multiple block reads for block allocation.

Following the free block list, will be a bad block list formatted identically to the free block list, except the magic number will be different.

The first data block is allocated for the root directory on the system. In the initial stages of the implementation, this will be the only directory on the system. A directory consists of a linked list of blocks, each with a directory block header specifying:

- magic number, version information, and header size
- directory name (pointer within block to string table entry)
- first logical block of directory
- first logical block of parent directory (0 if root)
- sequence number of this block

- logical block number of next block in directory (0 if last)
- logical block number of previous block in directory (0 if first)
- size of file entry within directory block
- first byte within block available for new file entry info
- last byte within block available for new file entry info
- reserved space for future additions

followed by a number of fix sized file entries, some free space, and a string table. File entries contain pointers to entries in the string table. New entries are added after the file entries, with the associated strings needed added before the string table. Strings are null termiated. To avoid frequent string moving, strings are zero padded to the next multiple of STRING_GRANULARITY (32 seems right, we'll see). When entries are removed, the associated strings are clear to zero. Subsequent changes to entries may reuse this space. When an entry changes so that it can no longer fit in a block, it will be removed and placed in the next directory block with sufficient free space (possibly creating a new directory block).

Each file entry will contain:

- a file name (as a pointer to an entry in the string table)
- block and entry number of i-node for file
- block and file entry number of next link to this file (0 if first)
- block and file entry number of previous link to this file (0 if last)
- reserved space for future expansion (maybe extended permissions info)

Each i-node block will contain a header specifying:

- magic number, version information, and header size
- logical block number for next i-node block (0 if last)
- logical block number for previous i-node block (0 if last)
- length of i-node block header
- length of i-node entry

followed by an array of i-node entries each specifying:

- type and permision information

- length of file
- number of blocks used
- creation, modification, and access times
- depth of index block tree
- block number of file index block or only block in file
- reserved space for future expansion

Each file consists of a BLOCK_SIZE/4-way tree of index blocks, with leafs nodes as data blocks. A depth of zero indicates no index blocks: there is only one block in the file and it is a data block. A depth of one indicates that the file consists of BLOCK_SIZE/4 data blocks which are listed in one index block. A depth of two indicates that the file consists of a one index block which points to BLOCK_SIZE/4 depth one index blocks which in turn each point to BLOCK_SIZE/4 data blocks. Similarly for higher depths. An index block consists of an array of BLOCK_SIZE/4 (16k) entries per block, with each entry containing the logical block number for a data block or a lower depth index block. If an entry is zero (never a valid block number since this is where the partition information is), then no block has been allocated for that section of the file and it is read as all zeros. Thus each entry in a depth d index block contains the information to access $g = BLOCK\_SIZE * ((BLOCK\_SIZE/4)^{(}d - 1))$ bytes of data. Entry n in the root index block represents data from byte $(n-1)g$ to byte $ng - 1$ of the file.