SCIENCE AND ENGINEERING RESEARCH COUNCIL
RUTHERFORD APPLETON LABORATORY

COMPUTING DIVISION

DISTRIBUTED INTERACTIVE COMPUTING NOTE 744

PERQ UNIX IMPLEMENTATION NOTE # 41
Accent/Unix: SwitchBoard

Issued by
J.C.Malone

26 January 1983

DISTRIBUTION:
R W Witty
K Robinson
C Prosser
A S Williams
L O Ford
T Watson
E V C Fielding
J C Malone
P J Smith
A J Kinroy
J M Loveluck
C P Wadsworth
Martin Ritchie (ICL Dalkeith)
P Palmer (ICL Dalkeith)
RL Support/PERQ/Unix Implementation Notes file

Since OBOE has no idea of the type of the file it is accessing, all Open and Creat requests [5] must be sent to one process to be directed to the relevant server. The server will return the port on which it will service future requests from OBOE, still without OBOE's knowledge of the file type [2].

SwitchBoard has the task of rerouting open requests to specific servers, and also provides the services of the disk file and pipe servers [3].

The following sections describe the modules which comprise SwitchBoard.

1. SWBTYPES

This holds type definitions which are used in the calls to SwBExport routines, and constants which would be useful to those programs using SwitchBoard.

2. SWBIMPORTS

Imports modules used by all SwB modules.

3. SWBPRIVATE

SwBPrivate holds routines and data structures internal to the SwitchBoard process.

3.1 Data Structures and Manipulating Routines

3.1.1 The Port Table

All open files are connected to a port, to which file access requests are sent. In the case of special files, these are sent directly to the driver [2]. Disk file accesses and pipe reads/writes [3] are sent to SwitchBoard, which must map between the port and the corresponding file image or pipe header.

Note that file entries generated by open or creat will have a new port entry, and separate read/write pointer, but files duplicated on a fork will share entries (file descriptor dups are handled in OBOE).

```
PortEntry = record    { of all information for a port }
             Valid : boolean;
             PortNumber : Port;

             case entrytype : EntryT of
                DiskFile : (FileAccess : AccessMode;
                            ForkCount : integer; {files on this port}
                            FileI : FileIndex;{into ArrayWorkingFile}
                            RWOffset : long;{pointer to Virtual file}
                            DiskID : SegID
                            );
                ReadPipe : (RPipeI : PipeIndex); {to ArrayPipeBlock}
                WritePipe : (WPipeI : PipeIndex) {to ArrayPipeBlock}
            end;

PortTable : array [PortIndex] of PortEntry; { all port information }
```

procedure GetPortEntry
    Returns first unused entry in the port table, or ErrorFlag = ENFILE if there are none.

procedure MakePortEntry
    Set the valid flag to show that this entry is in use.

procedure KillPortEntry
    Cancel the valid flag for this port entry.

procedure SearchSegEntry
    Find first valid entry in port table for given seg id.

procedure SearchPortEntry
    Find first valid entry in port table for given port.

procedure FillFileEntry
    Used by Open and Creat to fill port table entry for a newly opened working file.

3.1.2 The Open File Table

There is only one image for every open file, which may be accessed through a number of different ports, each of which has its own associated read/write pointer.

When the usecount has been decremented to zero, the file may be written back (if modified and not unlinked), and the virtual memory holding it invalidated.

FileLength is the current size of the file in virtual memory, whereas ValidSize shows how much memory has been validated to hold it.

```
WorkingFile = record
                MountI : integer; {Index into partition table}
                FileWritten : boolean; {has a file been written to}
                usecount : integer; {number of ports accessing file}
                FileLength : long;  {length in bytes}
                ValidSize : long; {amount of validated VM in buffer}
                BaseAddress : VirtualAddress;{of file in VM}
                Delete : boolean {if still inuse, to delay delete from
                                  close }
              end;

ArrayWorkingFile : array [FileIndex] of WorkingFile;


procedure GetFileEntry

procedure KillFileEntry
    An unused entry has a zero usecount.
```

3.1.3 The Pipe Header and Queue

The port table entries for the read and write ends of a pipe [3] both point to the same pipe block entry. Read and Write counts are incremented on forks.

All pipe write requests are linked together using pipebuffer headers. Here the ReadP indicates the beginning of the data to be read, and Size shows how much more is valid. A read request may take part or all of one buffer, or may require a number of buffers to be concatenated.

If a read request has been held up, waiting for data, then it is necessary to know the port of the write end of the pipe, which will free the read. This is kept in WPort.

```
PipeBuffer = record { Pipe writes are kept in original buffers
                      and linked }
              Start : VirtualAddress;
              Size : long;          {number of bytes left to be read}
              ReadP : long;         {may have read part of this buffer
                                     also, write may have included a
                                     byte offset, so whole of buffer
                                     was not valid }
              Next : PPipeBuffer
             end;


PipeBlock = record              {of information for one pipe}
             ReadCnt,           {readers on pipe}
             WriteCnt,          {writers on pipe}
             WPort    : Port;   {port on which waiting read is to
                                 be queued}
             First,
             Last     : PPipeBuffer
            end;

ArrayPipeBlock : array [PipeIndex] of PipeBlock; { pipes in VM }


procedure KillPBufHdr
     Removes first entry in pipe buffer queue and releases space.

procedure GetPipeEntry

procedure KillPipeEntry
     An unused entry has zero reader and writer counts.
```

3.1.4 The Mount Table

When a block special file [2] is mounted an entry is generated in the mount table giving the special and file names. In this implementation, special files are restricted to partition names.

The UseCount indicates the number of open files, or working directories on a partition.

```
MountEntry = record
              Valid : boolean;
              UseCount : integer;
              Special : PathName; {Full pathname of mounted partition
                                     e.g. sys:user> }
              FName : PathName;   {UNIX-type name
                                     e.g. /usr/
                                   all references to this name will
                                   be changed to refer to the
                                   special file directory}
              WritePerm : boolean {is writing allowed}
             end;

MountTable : array [1..MaxMount] of MountEntry;
```

procedure GetMountEntry

procedure KillMountEntry
    Cancel the valid flag for this mount table entry.

procedure CreateMountEntry
    Allocates and fills mount entry.

procedure SearchPartition
    Looks for valid entry for special file name.

procedure SearchFileName
    Looks for valid entry for file name.

procedure ActivePartition
    Increments use count, e.g. when file opened.

procedure ReleasePartition
    Decrements use count, e.g. when file closed.

3.1.5 The Device Table

When a device registers with SwitchBoard [2,4], it sends a port on which it expects to receive open and creat requests on the device name.

```
DevEntry = record
             MountI : integer;
             DevSegId : SegId;
             DevPort  : Port
           end;

DeviceTable : array [1..MaxDevs] of DevEntry;
```

procedure GetDevEntry

procedure KillDevEntry
    An entry is unused while the MountI value is zero.

procedure SearchDSegId
    Finds entry for given seg id.

procedure SearchDPort
    Finds entry for given port.

3.2 Routines Directly Interfaced To FileSystem

n.b. The file is always left closed after these routines.

This is to remove the problem of the Spice file system disallowing multiple file updates - which would occur, say, when a file is already open and a chmod is performed.

Also the routines may then open in the appropriate mode.

procedure FileLookUp
    If the file is not found, then add the suffix .DR and try again. This is because UNIX does not distinguish between names of files and directories, whereas POS does. FileName parameter will only be altered if the file is found with the suffix.

procedure CreateDirEntry
    Creates new file and enters in directory.

procedure TruncateFile
    Destroys file contents.

procedure ReadWholeFile

procedure WriteWholeFile

procedure DiskBufRead
    Reads File header block.

procedure DiskBufWrite
    Writes File header block.

function SegMakeDirectory
    Directly calls PFilesys routine, and converts returned file id to a seg id.

3.3 Data Conversion Routines.

function Chunked
    Memory to hold a file is validated in chunks. Given a number of bytes, this routine rounds up into a whole number of chunks.

function SizeInBlocks
    Converts bytes to file blocks.

function GetBlks
    Given the file header, checks on file type and returns:

        if normal file : Number of blocks in file

        if directory : Number of highest block used + 1

function SizeInBytes
    Given the file header, returns number of bytes in file.

function AbsolutePtr
    Given virtual address and byte offset, produces a byteptr.

function OwnerToUId
    Currently No-op, as Uids not stored in file header.

function GroupToGId
    Currently No-op, as Gids not stored in file header.

3.4 Routines Dealing with Stored File Characteristics.

The following are passed the file header information (which has been received on, e.g. a file lookup).

function IsDirectory
Checks FileType for DirFile.

function IsDeviceDriver
Checks FileType for DevFile.

function getmode
Converts FileType to Unix type and adds FileRights, which holds rwx permissions, to produce file mode.

The following are given the file segid and obtain the file header through DiskBufRead.

procedure getfilestat
Uses DataEntToStatStruct to convert Spice file header to Unix stat structure.

procedure setprotect
Extracts rwx permissions from given mode and stores in file header field FileRights.

procedure setfilemode
Stores permissions in FileRights, and also sets FileType.

3.5 General Support Routines.

Procedure DataEntToStatStruct
Converts file header information into stat_struct.

procedure getPipeStat
Fills out stat_struct information for a pipe.

function LongMin

function AccessPerm
All that is currently checked is that the given mode of access is permitted on this partition. It disallows writing on partitions mounted read-only.

function EOF1

function EmptyPipe
Returns True if the pipe data chain is empty.

function BytesInPipe
Works through linked pipe blocks, totaling block sizes.

function StdPosName
Used to standardise POS-style filenames so that they may be tested for equality. This just means stripping off trailing directory symbols, although it could be extended to convert to upper case.

function StdUnixName

Used to standardise Unix-style filenames so that they may be tested for equality. This just means stripping off trailing directory symbols, although it could be extended to convert to upper case.

procedure UToPFile

Converts the full Unix filename, as supplied by OBOE, into the filename recognised by the Spice file system.

Finds which partition the file is on, by finding the filename in the mount table which matches the longest part of the name. Then replaces this prefix by the special file name. Finally, replaces all Unix directory symbols with POS ones.

procedure AppendDr

If there is not already a .dr suffix, and it is possible to add one, then appends it. Returns a flag to say whether this has been done.

function PartNo

Given partition information, finds the index for a partition name.

function CheckPartition

Obtains partition information from Accent, then uses PartNo to see if the given filename is a partition name.

procedure UpdateDisk

Writes back all files which have been updated, but not unlinked.

procedure SendDriver

Finds the device port from the device table, sends an open request and receives the user's device port.

4. SWBEXPORT

The SwitchBoard module, in the SwBExport file, holds the system call routines which are called from OBOE via a message interface.

procedure SwBInit
Called at the beginning of SwBControl.

Performs table initialisation, allocates SwitchBoard reply port, sets time to 1st Jan 1980 and mounts the system partition.

procedure SwBSTime
Sets clock, given time in seconds.

procedure SwBTime
Returns time in seconds.

procedure SwBFTime
Fills in timeb_struct, using GetTimeStamp module and param.dfs.

procedure SwBUTime
Sets file access and write times in the file header. However, on writing back, these are updated to the current time by the Spice file system.

procedure SwBRegister
Called by device drivers on initialisation. If the device name does not exist it is made with SwBMkNod. The device name segid and driver port are entered in the device table.

procedure SwBMount
Checks that the files exist, that the special file is a partition, that the filename is a directory and that neither are already in the mount table.

procedure SwBUMount
If the special file is in the mount table and there is currently no open files, working or root directories on this partition, then it is removed from the mount table.

procedure SwBInUse
Used to tell SwitchBoard that a partition is to be used, when it would otherwise not know, e.g. when changing current directory, which is all done in OBOE.

procedure SwbRelease
To inform SwitchBoard that the partition is no longer required for the purpose for which SwBInUse was called.

procedure SwBFreeBlks
This will later be used for the DF system call.

procedure SwBOpen
Opens an existing file, without truncation. If the file is already open it shares the working file, otherwise a copy of the file is read in to virtual memory.

Also opens directories (read-only) and special files (by sending an open request to the driver, which returns the port on which the user may directly access the driver for future requests).

procedure SwBCreat

Creates new file, or truncates existing one (also truncates image in virtual memory, if already open).

Calls to create a special file cause SwitchBoard to send an open request to the device driver.

procedure SwBChMod

Used to change file rwx permissions only - calls SetProtect.

procedure SwBSync

Updates disk image of all open files - calls UpdateDisk.

procedure SwBClose

The use counts for the file or pipe are decremented and, when this was the last user, ports are deallocated.

If the port corresponds to a file then the disk will be updated, if the file is no longer in use, memory will be invalidated.

If the close is for the last reader of a pipe, then any unused pipe buffers are thrown away. If there are no more writers then the pipe header may be destroyed.

The final close of the write end of a pipe results in any readers waiting on the pipe being sent EOF.

procedure SwBDeadDevice

Called by SwBControl whenever it receives an emergency message to say that a port to which SwitchBoard had access has died. The device table is searched and dead devices removed.

procedure SwBLink

This is implemented as a rename, name1 being renamed to name2.

procedure SwBLSeek

If the port entry is for a file and the arguments produce a legal file pointer, then the RWOffset is updated.

procedure SwBRead

The read returns a byte address and the number of bytes that follow, or EOF.

File reads are taken from the working file area, beginning at the current file position.

Pipe reads are taken from the linked list of buffers. If the first buffer satisfies the read then the address is taken as the start of the valid data. Otherwise several buffers must be moved to one contiguous area, and the address of this is returned.

Used buffers are removed from the list; those which have only been partly read will have their offset and size updated.

If a read request receives the end of the buffer then the Deallocate flag is set when sending back the reply. When several buffers are combined, they are deallocated after being moved, and the whole buffer is deallocated in the reply message. When a large buffer supplies several read requests, then any pages which have already been read are invalidated, this is to keep the offset byte pointer within integer range.

procedure SwBWrite

If the port maps on to a file then the data is moved to the current file pointer, validating more memory to hold it if necessary. File pointer, FileWritten flag and size (if necessary) are updated.

If the port corresponds to the writer of a pipe and there are no readers then the signal SIGPIPE must be sent to the writer. SwitchBoard returns the error EPIPE to OBOE, which directly invokes the trap handler [6]. Otherwise the write buffer is added to the chain of buffers, noting the offset of the beginning of the valid data within the buffer and the size.

procedure SwBpipe

Sets up new port table and pipe block entries.

procedure SwBmknod

First checks that the file does not exist, then calls either SegMakeDirectory or CreateDirEntry, finally uses setfilemode.

procedure SwBstat

If file exists, returns file information in stat_struct.

procedure SwBFStat

Calls getfilestat or getpipestat, depending on type of file open on given port.

procedure SwBunlink

Deletes the file and, if it is currently open, sets file delete flag so that it will not be written back.

procedure SwBFork

Increments file usecount and forkcount, or pipe read or write count, depending on structure associated with port.

procedure SwBIOCtl

Calls on devices will go straight to the device port, so that all calls coming into SwitchBoard are illegal.

5. SIFSWB

This holds the server routine which decodes requests and calls the relevant SwBExport routine [1].

6. SWBCONTROL

The SwBControl routine is called by SwBBoot and controls the services provided by SwitchBoard. The majority of requests are received, satisfied and answered, except for pipe reads.

All pipe read request messages which cannot be immediately satisfied are queued on separate queues, according to the port of the write which will free them. When a write has been performed, the queues are checked and as many read requests reissued as can be satisfied. Similarly, if the write end of a pipe is closed, then the reads are reissued to receive EOF.

Entries must be removed from the queue when satisfied or when a CancelReq request is received.

REFERENCES

1. L.O. FORD, "Perq Unix Implementation Note # 45 - Accent/Unix: Overview of Serverloops and remote procedure call interfaces", DIC Note # 748, Rutherford Appleton Laboratory (January 83). [to be published].

2. J.C. MALONE, "Perq Unix Implementation Note # 51 - Accent/Unix: Special Files, Mountable File Systems", DIC Note # 756, Rutherford Appleton Laboratory (January 83).

3. J.C. MALONE, "Perq Unix Implementation Note # 52 - Accent/Unix: Pipes", DIC Note # 757, Rutherford Appleton Laboratory (January 83).

4. J.C. MALONE, "Perq Unix Implementation Note # 44 - Accent/Unix: TermDriver", DIC Note # 747, Rutherford Appleton Laboratory (January 83).

5. A.S. WILLIAMS, "Perq Unix Implementation Note # 31 - UNIX System Call Specification", DCS Note # 727, Rutherford Appleton Laboratory (November 1982). [In Preparation].

6. A.S. WILLIAMS, "Perq Unix Implementation Note # 50 - Accent/Unix: Signals and deadports", DIC Note # 755, Rutherford Appleton Laboratory (December 82). [to be published].