Getting Started with *Choices*

David Raila

September 15, 1994

With revisions for Version 0.5 beta by the Mungi Group at the University of NSW, Australia.

Abstract

This manual is an introduction to *Choices*. It is designed to help the beginning *Choices* user become acclimated with the the source, tools, and techniques used to build the components of *Choices*. It covers the source code, techniques for building and modifying kernels and applications, and descriptions of the different subsystems, etc. It should contain enough information for the reader to become familiar with the *Choices* distribution, basic terminology and basic design.

Draft Notification

This draft is an attempt to expand the information available to users of *Choices*. If anybody has any suggested improvements, please contact Ellard Roush. The goal is to reduce the time spent trying to figure out how to used *Choices*, especially for those of us still learning the system. I eventually will be submitting the changes to David Raila for possible inclusion in the base line. Some *Choices* shell features (remote execution, command files, and some help items) have not yet been submitted for the baseline.

Contents

1 Introduction

1.1 Syntax Conventions

- Throughout this report, user include and command names will appear in **bold face**. System output will appear in a `constant width font`. Files, as opposed to directories, will appear in standard font.

1.2 How to find help

- Look through this manual first
- Try posting to the newsgroups uiuc.cs.choices or uiuc.cs.choices.bugs.
- Send mail to choices@cs.uiuc.edu
- Bug some of the *Choices* gurus in the lab

1.3 How to report bugs

- Try posting to the newsgroup uiuc.cs.choices.bugs.
- Send mail to choices@cs.uiuc.edu

1.4 What is *Choices*?

- *Choices* is an object-oriented operating system
- Written in C++
- Machine and processor dependent portions are in assembler
- It has an object-oriented system interface
- Operating system entities are objects
  - Process, Semaphore, CPU
  - MMU, Disk, etc.

2 The *Choices* Distribution

Choices distributions are referred to as stables. Each stable contains the source code for the operating system and applications. Tools sources will be included in future stable releases.

2.1 Source Tree Description

The top level stable directory hierarchy is illustrated in Figure 1. Here is a brief description of the top level. Each item will be covered in more detail later.

- Applications - The Applications directory contains object oriented application sources for *Choices* applications. The Examples directory within it contains some basic UNIX like programs and some test programs. *FiSh* is a *Choices* shell. The others are more srg group specific and not addressed here.
- BSDNet - sources to support a TCP/IP protocol network interface.
- Common - Sources common to the kernel and applications. This directory contains mostly reference-counting code (Stars and Refs, covered later).
- Configure - Directory for building components of *Choices*. This is where libraries, kernels, and applications are built. (See Figure 3)
- Distributed - Contains separate directories for message passing and distributed name server code.
- FileSystems - Filesystem sources.
- HISTORY - File detailing features added, and changes made to *Choices* over time.
- IODevices - Machine independent device driver sources.
- Includes - Directory for ALL header (.h) files. This directory contains the kernel include files. It also has subdirectories for Common, ConfigurationFiles, MachineDependent, ProcessorDependent, Memory, etc. It mirrors the top level hierarchy and is shown in Figure 2. The ConfigurationFiles directory is machine dependent and contains files which are included in other files. These files contain things such as #define LOCK SS1Lock, which allows code that uses locks to be independent of the machine.
- Instrument - Sources for instrumentation code.
- Kernel - Kernel sources.
- Libraries - Library sources. There is a general purpose library that is portable, and system interface libraries for each machine that applications link with.
- MachineDependent - Machine specific sources, grouped by machine. Machine specific device drivers, boot codes, etc. reside here.
- Memory - Memory management subsystem sources. Also contains a directory for distributed virtual memory code.
- Networks - Networking protocol sources.

Stable

- Applications
- BSDNet
- Common
- Configure
- Distributed
- FileSystems
- HISTORY
- IODevices
- Includes
- Instrument
- Kernel
- Libraries
- MachineDependent
- Memory
- Networks
- ProcessorDependent
- Schedulers
- Tools

Figure 1: The top level of the *Choices* source tree

- ProcessorDependent - Processor specific sources. This code does things like context switching, CPU and MMU operations, etc.
- Schedulers - Source for a bunch of schedulers.
- Tools - Miscellaneous tools.

2.2 Support Software

Other tools are needed to work with *Choices*. A C++ compiler, a linker, and an assembler for your platform. The Free Software Foundation gnu tools are a good base to get started with. If using the gnu tools, you will need all of the libraries, and the patches supplied with the stable, to be installed. You may also need to experiment with the various gnu linkers in order to get a clean compile.

- Includes
  - Common
  - ConfigurationFiles
  - Distributed
  - FileSystems
  - IODevices
  - Instrument
  - Libraries
  - MachineDependent
  - Memory
  - Networks
  - ProcessorDependent
  - Schedulers
  - System

Figure 2: The Includes directory hierarchy

- Configure
  - Top level makefiles
  - Applications
  - Libraries
  - System
  - Tools

Figure 3: The Configure directory hierarchy

3 Building *Choices* Components

This section describes building the different *Choices* components from a clean stable. Certain parts are dependent on others and are presented first.

3.1 Creating a Working Environment

The way to work with a *Choices* stable is to make a shadow if it. The shadow contains real directories, but all the files within them are symbolic links back to the stable. This saves space and allows easier identification of changed files specific to your node.

3.1.1 newnode

The **newnode** command is used to create a *Choices* node. On our system, the name of the directory containing the *Choices* stables is '/Nodes/Stables'. The name of the directory containing newnode is /mnt/tapestry/Tools/Bin, which should be included in your $PATH variable. Assuming that you are in your home directory and wish to create the new *Choices* node there, you would issue the following command :

```
newnode /Nodes/Stables/latestStable Choices
```

3.1.2 localclean

After the new *Choices* node has been created, the node should be prepared for local use (in this case for Virtual Choices node) by the following commands:

```
cd Choices/Configure/System/VChoices
gnumake localclean
```

3.1.3 clobber

Normally it is recommended that the new *Choices* node be purged, so that everything will be built from source. The following commands perform the purge and then build the system from source (in this case for a Virtual Choices node):

```
cd Choices/Configure/System/VChoices
gnumake clobber
gnumake
```

3.1.4 breaklink

The **breaklink** command is used when you wish to modify source files. Given a list of files, it replaces the links with modifiable files. A link to the original file is created with '.O' added to the original file name. The command will report an error and do nothing if a file already exists with the '.O' name.

```
breaklink filename filename ...
```

Making a copy of the originals is recommended.

3.2 *Choices* makefiles

Choices uses a hierarchy of Makefiles and depends on features of gnumake. In the top level Configure directory you see files such as:

- Makefile - The top level makefile. It knows about all the components and how to tell each component to make itself. If you type **gnumake** here, this makefile will build everything for every platform. It takes a long time to do so.
- ChoicesCommon.mk - Defines some targets common to all parts of *Choices* such as **localclean** (**gnumake localclean** removes files built from other files), **clobber** (**gnumake clobber** adds removal of dependencies to **localclean**). This file is included within other makefiles.
- <machine>/Common.mk - Defines tools, compilers, compiler flags common to all *Choices* components, and basic rules for compiling files. This file is included by other makefiles.

The makefiles in Configure/System define targets and rules for compiling the *Choices* kernel. They contain things that are common to every system. A **gnumake** from this level creates every kernel for every platform.

Configure/System/<machine> has two files for building the kernel for a particular platform. Makefile is the Kernel makefile for that platform. It defines all the machine and processor specific targets for that platform and any rules specific to the kernel for that platform, such as a link specification. The configuration file Config.mk in this directory is used to turn on and off different subsystems and options in the kernel (for more details see the section on configuration files).

If we type **gnumake** from Configure/System/SS1 the makefile includes SS1Common.mk from the top level and SystemCommon.mk from the Configure/System level. In turn SystemCommon.mk includes the top level ChoicesCommon.mk, CommonSources.mk, FilesystemCommon.mk, etc.

To add a file to a single kernel, a simple addition of that file to the targets in Configure/System/<machine>/Makefi is all that is necessary. To add a file to all kernels an addition is made to SystemCommon.mk which is automatically included in all systems.

The application makefiles have a very similar structure. One difference is that applications must be loaded to specifications which come from Configure/Libraries/SystemInterface/LoaderOptions.mk.

3.3 *Choices* configuration files

3.3.1 Config.mk

There is a configuration file Config.mk for each platform. The file is located in Choices/Configure/System/<machine> This configuration turns on or off subsystems for a *Choices* platform. Entire programs are either used or not. This configuration file further controls which parts of programs are compiled into the system.

The options are selected by defining the corresponding term. Entering a "Yes" after the equal sign defines the option, while an empty line does not define the option.

BSDfile – causes the incorporation of the BSD file system. Any normal configuration will include this option.

RECORDFile – causes record oriented I/O to be supported. Also needed for any normal configuration.

Networking – causes the network interface to be supported.

Distributed – causes message passing to be supported along with features that depend on message passing. DistributedNameServer is among those features added by this switch. This switch requires that "Networking" also be turned on.

BSDNet – causes message passing to be supported using the Unix BSD model that involves sockets. This option is incompatible with the "Distributed" option. This switch requires that "Networking" also be turned on. The BSDNet software has not yet been ported to run under Virtual Choices.

Zoot – Zweig Object Oriented TCP subsystem is not operational.

3.3.2 /Configuration/Local

There should be a /Configuration/Local file for each physical machine. The *Virtual Choices* system should have this file in its virtual file system. However, the file cannot be loaded into the virtual file system until after it has been brought up the first time. *Virtual Choices* can function without this file, though it obviously will not know some things. This configuration file is used to initialize various parameters at system startup.

The configuration file is a text file. Any line beginning with a '#' character is treated as a comment line and ignored. The other information consists of two character strings per line separated by white space. The system software performs a lookup on one character string to find the other. Some of the important data entries are:

- hostname – "name of this host"
- hosttype – "type of this host", for example SparcStation
- hostIPaddress – IP address
- gatewayIPaddress – IP address
- IPsubnetMask – 8 digit hex mask, for example FFFFFF00
- gatewayEthernetAddress – ethernet address
- otherhostname – ethernet address

3.4 Parallel Build

The make files for *Choices* support a degree of parallelism in building the system when a multi-processor is used. The gnumake -j option is used for parallelism. On a single processor system the parallel option is simply ignored. The following example shows the parallel build option for building the Virtual Choices kernel on a 3 processor machine:

```
cd /Choices/Configure/System/VChoices
gnumake -j3
```

3.5 Building the GeneralPurpose Libraries

The GeneralPurpose libraries are required by all other parts of *Choices* and so must be built first when modified, cleaned, or clobbered. The library is built from Configure/Libraries/GeneralPurpose/<machine>. Once in that directory type **gnumake**.

3.6 Building the Kernel

The SystemInterface library depends on the kernel, so the kernel must be built next if a **gnumake localclean** or a **gnumake clobber** has been done in the Configure/System/<machine> directory, or if kernel sources have been modified. A **gnumake** command in that directory builds the kernel and also libSystem.a, the kernel part of the SystemInterface library.

3.7 Building the SystemInterface library

As already mentioned, the SystemInterface library depends on the kernel, so the kernel must be built first. Once the kernel is up to date, a **gnumake** in Configure/Libraries/SystemInterface/<machine> builds the SystemInterface library for that machine.

3.8 Building Applications

Whenever changes occur in the GeneralPurpose library, kernel, or SystemInterface library, it is necessary to build the applications. This does not occur automatically.

To build a set of applications for a platform, execute the following commands:

```
cd  Configure/Applications/<application>/<machine>; gnumake
```

To build *FiSh* (the shell) for Virtual Choices execute the commands:

```
cd  Configure/Applications/FiSh/VChoices; gnumake
```

3.9 Creating Multiple Virtual Choices

Multiple Virtual Choices are needed when Virtual Choices is used in a distributed environment. Each active Virtual Choices system requires its own file system, which are simulated by the files vdisk0 and swap0. Multiple Virtual Choices can also be used on one physical machine to simulate a network with multiple computers (naturally things slow down in this case).

It is not necessary to create another *Choices* node for this situation. After the first Virtual Choices system has been created in the normal way, the following commands show how to create another Virtual Choices system with its own file system:

```
cd  Choices/Configure/System/VChoices
rm swap1 vdisk1
gnumake swap1 vdisk1
```

4 *Choices* Startup and Halt

4.1 General Startup and Shutdown Procedures

Booting your kernel or the kernel your applications are built for is a machine specific process. In general the procedure is something like this:

1. Copy your kernel and applications to the target machine. This can be done while the machine is running UNIX with ftp, rcp, etc.
2. Obtain exclusive access to the machine. This is organization dependent and is accomplished by getting exclusive access to the console in our lab.
3. Shut down UNIX cleanly.
4. Boot your kernel. This is machine specific. Some machines can boot over a network (Sun) and some must boot from their disk.
5. Bring down *Choices*.
6. Boot UNIX, making sure that a filesystem check is run properly.
7. Release the console.

See the instructions for your machine for booting and shutdown details. There is a srg *Choices* environment manual for the srg lab and an instructional lab environment manual for the labs at the University of Illinois.

If the system encounters with the file /Configuration/dev/nit, it means that this machine is not properly configured for *Choices*.

4.2 *Choices* Kernel Commands

When *Choices* boots properly you should see some messages, a copyright notice, any debugging messages, and then a prompt:

```
Enter path of binary executable application file:
```

At this point you can type in the path to your application in the filesystem to run it. The *Choices* shell **FiSh** is usually run at this point.

4.3 *Choices* Builtin Commands

While at the enter application prompt you can execute a number of kernel-builtin commands by typing 3 digit numbers.

The following commands are available:

- 111 - Stop instrumentation gathering.
- 123 - Run context switching times benchmark (valid on multiprocessors).
- 321 - Shut down *Choices*. This command works for all types of *Choices*.
- 555 - Break to debugger.

- 666 - Print kernel heap statistics.
- 777 - A basic timing benchmark.
- 888 - Print Active Objects.
- 999 - Synchronize filesystem.

4.4 Targets

At the University of Illinois a set of machines are dedicated to running Choices, and they are called Targets.

4.4.1 Target Access

Programmers access a target machine via **telnet**. The programmer is connected to the target machine's system console. The programmer can select the machine by number or name. An example follows of connecting to a machine which already has Unix running:

```
studius% telnet targets
Trying 128.174.240.40 ...
Connected to targets.cs.uiuc.edu.
Escape character is '^]'.

Rotaries Defined:
    ss2t1                           1
    ss2t2                           2
    ss2t3                           3
    ss2t4                           4
    ss2t5                           5
    ss2t6                           6
    ss2t7                           7
    ss2t8                           8
    ss2t9                           9
    ss2t10                          10
    x10pcycle                       12
    envius                          13
    tedius                          14
    studius                         15
    gregarius                       16
    cli                             -

Enter Annex port name or number: 6
6
Attached to port 6

ss2t6 login:
```

Access conflict are resolved simply. Since each target machine 'ssXtN' has only one console, if you get the console prompt then you have the machine.

4.4.2 Unix Startup and Shutdown

A Unix system is halted by logging in with the user id 'halt'.

The system console prompt is simply the '>' character.

Unix is booted by the following command:

```
>b disk
```

4.4.3 Problem Resolution

If the machine does not respond, one can try sending a break command to it.

```
telnet> send break ss2t1
```

If the machine is really dead, the following command will power the machine off and back on.

```
powercycle ss2t1
```

4.5 *Virtual Choices*

Virtual Choices is implemented as a UNIX process using signals to simulate interrupts. The *Virtual Choices* kernel is built by typing **gnumake** while in the Configure/System/VChoices directory (refer to the section on Building Choices for more details).

4.5.1 Startup

There are two ways of bringing up Virtual Choices.

Since Virtual Choices is really an application program as far as Unix is concerned, it can be run like an application with a lot of debugging help. When Virtual Choices is started under **gdb**, the Virtual Choices file system is automatically checked for file system corruption and bad data is purged. The following commands show how to fully take advantage of the debugging possibilities when running Virtual Choices:

```
cd Choices/Configure/System/VChoices
emacs                   (emacs is a powerful editor)
M-x gdb                 (M-x is Meta-x in emacs)
Choices                 (gdb asks which file to run)
set verbose on          (ensures gdb always finds source)
go                      (command to start Choices)
```

Virtual Choices can also be run directly. The primary advantage of this approach is that the system starts a little faster, at the cost of less debug support.

```
cd Choices/Configure/System/VChoices
Choices
```

4.5.2 Multiple *Virtual Choices* Startup

More than one *Virtual Choices* systems are needed when communications are to be exchanged between systems. The first system is brought up as described in the prior section. The next system requires one difference in the system startup. The first system uses vdisk0 and swap0 by default.

An example of second system startup under debugging follows:

```
cd Choices/Configure/System/VChoices
emacs
M-x gdb
Choices
set verbose on
go1
```

An example of second system startup without debugging support follows:

```
cd Choices/Configure/System/VChoices
Choices 1
```

4.5.3 *Virtual Choices* File System

Loading Files The file system for *Virtual Choices* is implemented on a file called *vdisk0* in the same directory as the kernel. To load application programs into the file system, use the **cpon** (copy on) command at the shell or kernel prompt. The usage of this command is:

```
cpon UNIXPathname ChoicesPathname
```

For example, to load *FiSh* from its directory in the UNIX file system into the *Virtual Choices* file system enter the following cpon command. Note that the command will usually wrap to a subsequent line, but do not break up the command with a carriage return.

```
Enter path of binary executable application file: cpon
../../Applications/FiSh/VChoices/FiSh FiSh
```

Now you can run FiSh by typing:

```
Enter path of binary executable application file: FiSh
```

The corresponding command to transfer a file from the *Virtual Choices* file system to the Unix file system is 'cpoff'. However the function has not been implemented.

File Corruption In the case of file corruption, the following commands will usually correct the Virtual Choices file system:

```
cd Choices/Configure/System/VChoices
/etc/fsck -y vdisk0
/etc/fsck -y swap0
```

An alternative sequence of commands obliterates the old virtual file system and creates a new one:

```
cd Choices/Configure/System/VChoices
rm vdisk0 swap0
gnumake vdisk0 swap0
```

4.6 Sparc *Choices*

One configuration of *Choices* runs directy on the Sparc2 machines.

4.6.1 Installing Boot Image

At installation time the target machine must be running Unix. Otherwise the installation commands will run to completion without error, but have no effect. The following commands install native Choices on a Sparc machine:

```
cd Choices/Configure/System/SS1
setenv TARGET ss2t1
gnumake install
```

4.6.2 Installing Other files

At installation time the target machine must be running Unix. Any desired files, such as configuration files, command files, or application programs, are loaded by a remote copy command:

```
rcp FiSh ss2t1:/Choices/myFiSh
```

4.6.3 Sparc *Choices* Startup

Choices is booted by the following command, when at the system console prompt, which is simply the '>' character.

```
>b net
```

5 The *Choices* Shell – *FiSh*

The current *Choices* shell is a UNIX-like shell with some object-oriented features.

5.1 Help

FiSh provides a help facility. By entering either **help** or **?** at the shell prompt, the program will respond with a description of the help facility. Entering the help command followed by "?" will result in a list of the available commands. Entering the help command followed by the name of a command results in a short description of that command.

5.2 File System Commands

FiSh supports UNIX shell commands like **append, cat, cd, chgrp, chmod, chroot, cp, create, ln, ls, mkdir, mount, mv, rm, rmdir, umask, umount** for the ordinary file system operations. There are usually fewer options than in Unix.

5.2.1 Mounting and Dismounting local file systems

You can mount a MemoryObject that contains a file system by giving the name of the MemoryObject you want to mount and the path where you want it to be mounted. For example the command:

```
FiSh> mount fd0 /diskette0
```

will open the MemoryObject `fd0` which corresponds to the floppy diskette as a MemoryObjectContainer and will mount the file system under the path `/diskette0`. The file system can be unmounted by using the command `umount`. For instance, the diskette from the earlier example can be unmounted by using the command:

```
FiSh> umount /diskette0
```

5.2.2 Mounting and Dismounting remote file systems

A distributed file system has been added to *Choices*. The distributed file server is stateless. The distributed file system makes access to files location transparent, except for the mount command. In other words the same commands are used, except for the mount command. The mount command requires that an IP address preceed the Memory Object. The following provides an example:

```
FiSh> mount 123:123:123:123/fd0 /diskett0
```

5.2.3 Unix to VChoices File Copy

The shell supports the **cpon** command to copy a file from the Unix file system to the VChoices file system. Naturally this command only works on VChoices. The command was made available from the shell so that it could be used in indirect command files in VChoices. Currently there is no command to copy files in the other direction. Note that it can take a little while to copy large files. An example of the command is provided:

```
FiSh> cpon ../../Applications/FiSh/VChoices/FiSh fish
finished copying file
FiSh>
```

5.3 Program Execution

FiSh treats any command that it does not recognize as the name of an application program. The shell causes the application program to execute, and waits for the application program to terminate. There is no significant difference between starting an application from under the shell or the Kernel. If the application program does not have a read pending and the user enters a carriage return, the shell will respond with a prompt.

5.4 Raid

FiSh can be used to control RAID debugging (refer to the software development section for more details).

5.5 Object Examination

Fish provides commands that examine the objects and the class hierarchy in *Choices*.

5.5.1 The hierarchy command

This command displays the ancestor classes and descendant classes of a specified class, as in the following example:

```
FiSh> hierarchy UNIXInode
hierarchy UNIXInode
Object 12
ProxiableObject 40
MemoryRange 48
MemoryObject 96
PersistentMemoryObject 320
FileObject 340
UNIXInode 448  <<<
BSDInode 456
```

5.5.2 The ancestors command

This command displays the ancestor classes of a specified class, as in the following example:

```
FiSh> ancestors UNIXInode
ancestors UNIXInode
Object 12
ProxiableObject 40
MemoryRange 48
MemoryObject 96
PersistentMemoryObject 320
FileObject 340
UNIXInode 448
```

5.5.3 The descendants command

This command displays the descendant classes of a specified class, as in the following example:

```
FiSh> descendants UNIXInode
descendants UNIXInode
UNIXInode 448
BSDInode 456
```

5.5.4 The members command

This command displays the actual objects of a specified class, as in the following example:

```
FiSh> members UNIXInode
members UNIXInode
0 instances.
```

5.5.5 The kindred command

This command lets the user view all the objects of a particular class and of its descendants, as in the following example

```
FiSh> kindred UNIXInode
kindred UNIXInode
BSDInode[0x2a4000]{3}(ProxyMo)(:0:0:4)
BSDInode[0x2a4200]{1}(.data+.bss)(:0:0:5)
BSDInode[0x2a4600]{1}(ArgumentMO)(:0:0:6)
BSDInode[0x266e00]{1}(FiSh)(:0:0:4)
BSDInode[0x254200]{1}(/)(:0:0:2)
5 instances.
```

5.5.6 The instance command

This command is currently not operational.

```
FiSh> instance 0x266e00
instance 0x266e00
Assert failed
Assertion (*to != 0) failed
in ../../../Kernel/ObjectProxyKernel.cc, line 712
Debug Trap
```

5.6 Indirect Command Files

Fish can execute commands from a file instead of from the keyboard. Only one command is allowed per line, and the commands are the same as those entered from the command line. The Indirect Command File must have the execute privilege turned on. When *FiSh* encounters a "@" character followed by at least one space, the shell then executes commands from the file named in the next command line argument. An example follows:

```
FiSh> @ indirectcommandfile
```

5.7 Comment Lines

FiSh ignores any command line that has as the first non-blank argument a "#" or "?" character (which must be separated by at least one blank from the rest of the line). This is useful primarily for documenting indirect command files.

5.8 Remote Shell

FiSh supports execution of commands on remote systems when the appropriate networking support is available. Any command is executed on a remote system when the normal command is preceded by two arguments. The first argument is **rexec** and the second argument is the name of the remote host. These two arguments are stripped from the command line prior to execution of the command line on the remote host. An example follows:

```
FiSh> rexec system2 ls
```

The remote system must already have *Choices* operational and have the remote shell active. The name of the remote shell is *RFiSh* and it is located in the same directory as the *FiSh* shell. The remote shell is activated by running it in the same manner as any other application program.

5.9 Options

The shell **set** allows one to set the following options:

GID – group ID

HOME – current directory

HOST – host name

PS1 – prompt string

UID – user ID

The above variables can be displayed by preceding the above names with a '$' character in the **echo** command.

5.10 The pause command

The **pause** command causes shell processing to pause until the user enters a carriage return.

5.11 The mkpo command

The **mkpo** command stands for "make persistant object". It is used for some of the unusual file types to establish type. It is not used for normal files. The following example shows how it used to create a Archive file entry and then load an Archive file from Unix under VChoices (Note that the use of UNIXfish was recommended):

```
mkpo ARContainer x.ar
cpon x.ar x.ar
```

5.12 The inst command for Simple Instruments

The **inst** command is used to control the Simple Instrument package. The command requires 3 arguments in addition to the command name. The first argument is either 'm' for member or 'k' for kind. The second argument is the name of the "class". The third argument is either 'on' to turn on the instruments, or 'd' for the display function, or 'r' for the reset function, or 'off' to turn off the instruments. An example follows:

```
inst m Process on
```

6 Debugging Support for *Choices*

6.1 *Choices* Debug Statements

A streams console I/O Debugging facility is available and activated on a per file basis by defining the preprocessor symbol DEBUG and then including OutputStream.h. It is much the same as using `Console()` for output. For example:

```
#define DEBUG

...
#include "OutputStream.h"

...
Debug << "Got here: i(hex) = " << asHex(i) << ": (dec) " << i << "\n" << eor;
```

The `eor` symbol flushes the output. When DEBUG is not defined these statements compile away to nothing.

6.2 Assertions in *Choices*

Choices code can contain Assertions. When ASSERT is defined in a file before Assert.h is included, there are two statements that check assertions:

```
Assert( i == 0); // Fails if i != 0
Assert( i == 0)("i should be zero, it is %d\n", i);
```

The first style prints the kernel line and file and the condition if it fails. The second is meant to provide information to applications and prints the `printf` style message before the rest of the information. In either case the current process is looked up and if it is an application the process is killed. If the process is a system process the kernel halts. It is advisable to always run with Assertions enabled unless you are specifically looking for performance numbers.

6.3 RAID Debugging

Raid statements inside the *Choices* kernel code print information when they are enabled on a class and category basis. **Raid** defines different levels of messages and is dynamically controlled. The next sections describe how to control **Raid** statements from *FiSh*.

6.3.1 Usage

There are two commands for controlling **Raid** debugging:

1. **odb**, which turns on debugging for existing objects belonging to a given class, and
2. **cdb**, which turns on debugging for future objects belonging to a given class.

The formats of the commands are:

```
FiSh> odb {member|kind} ClassName mask [messageCount]
```

and:

Bit	Mask	Name	Type
0	1	**RaidError**	error conditions (not used much yet)
1	2	**RaidConstructor**	constructor/destructor functions
2	4	**RaidReference**	reference()/unreference()/noRemainingReferences() functions
3	8	**RaidMethod**	public member functions
4	16	**RaidProtected**	protected and private member functions
5	32	**RaidDetail**	detailed information within any type of member function

Figure 4: **Raid Debugging Masks**

```
FiSh> cdb {member|kind} ClassName mask [messageCount [objectCount]]
```

For both commands, a mask of zero will turn off debugging. The paragraph below explains the possible values of the mask. Examples of these commands now follow.

To turn on debugging for the public methods for all currently instantiated members of class *Parsley*, use the following command:

```
FiSh> odb member Parsley 8
```

To turn on all debugging for all currently instantiated members of class *Sage* and its descendants, but restricted to the first 10 messages per object, use the following command:

```
FiSh> odb kind Sage 63 10
```

To turn on constructor and reference count debugging for all future objects instantiated from class *Rosemary*, use the following command:

```
FiSh> cdb member Rosemary 6
```

To turn on all debugging for the next 3 objects instantiated from class *Thyme* and its descendants, use the following command:

```
FiSh> cdb kind Thyme 63 32767 2
```

6.3.2 Raid Masks

One can turn on six different categories of debugging statements for either existing instances or future instances of specific classes or classes and their children. These debugging categories are encoded using the six bit masks shown in Figure 4.

When entering masks for **odb** or **cdb** statements (see above), one can use the logical sum of the desired masks. For example, if one wants to see constructors, destructors, and reference counting methods, use a mask of 6 (`RaidConstructor|RaidReference`). One can also use mask names as in `"Constructor|Reference"`. These names must be enclosed in double quotation marks (") and they should not contain any blanks or tabs. The names may, however, be abbreviated to as little as a single character.

When using masks in **Raid** statements (see below), use the name of the mask. Usually a single **Raid** statement will have only one mask, while **odb** and **cdb** statements can have one or more masks.

6.4 GDB Debugging

6.4.1 General GDB Information

See the **gdb** documentation and the **gdb** quick reference card for details on available commands.

Since we are debugging a kernel there are places where breakpoints can't be set and places that can't be single stepped without crashing the kernel. The well known places on *Choices* where breakpoints should not be set include the lock code, the boot and trap code in **boot.s**, and the parts of the context switching code that manipulate register windows.

The set p dereference off command line tells **gdb** not to dereference pointers. This is very useful in operating systems since dereferencing a pointer can cause a page fault, which you may happen when you are debugging.

6.4.2 GDB and Virtual Choices

Since Virtual Choices is implemented as a Unix application program, GDB can provide debugging support in the same manner as for an ordinary application program. The section on Virtual Choices startup shows how to run Virtual Choices under GDB.

6.4.3 Remote GDB Debugging

The SPARC port of Choices supports remote kernel debugging via **gdb**, the gnu debugger. The 555 kernel command traps to **gdb**. If the kernel will not boot up to the point of having a prompt the flag _debugFromStart can be set with the following **adb** commands:

```
adb -W Choices _debugFromStart?W 1 ^D
```

Where $\hat{\text{D}}$ is control-D. When a debugger trap is taken all kernel activity stops and the the kernel polls serial port A for **gdb** I/O. Using another SPARCstation connected to the serial port and using **gdb** version 4.2 the following commands connect to the remote kernel

```
%% gdb Choices
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.2, Copyright 1991 Free Software Foundation, Inc...
(gdb) set p dereference off
(gdb) target remote /dev/ttya
SPARCCPU::debugBreakpoint ()
    at ../../../Includes/ProcessorDependent/SPARC/SPARCCPU.h:101
101             inline void debugBreakpoint(){TrapToDebugger();}
Breakpoint 1 at 0x1710c: file
../../../ProcessorDependent/SPARC/SPARCSourceOfZerosMemoryObject.cc, line 42.
Breakpoint 2 at 0x20114: file ../../../Common/ProxiableObject.cc, line 136.
Breakpoint 3 at 0x201a8: file ../../../Common/ProxiableObject.cc, line 143.
(gdb)
```

By executing the **bt** command we can see that we are in SPARCCPU::debugBreakpoint:

```
(gdb) bt
#0  SPARCCPU::debugBreakpoint ()
```

```
    at ../../../Includes/ProcessorDependent/SPARC/SPARCCPU.h:101
#1  0x2b334 in Kernel::getMemoryObjectOfInitialApplicationProcess (
    this=0x124b80) at ../../../Kernel/Kernel.cc:1587
#2  0x2a47c in Kernel::dispatchApplications (this=0x124b80)
    at ../../../Kernel/Kernel.cc:1252
#3  0x2930c in Kernel::main (this=0x124b80) at ../../../Kernel/Kernel.cc:1049
#4  0x28c5c in Kernel::setupProcessEntry (k=0x124b80)
    at ../../../Kernel/Kernel.cc:927
#5  0x13f00 in SystemProcessReturnLabel ()
(gdb)
```

6.5 CDB Debugging

The Att6386 platforms use a kernel debugger called CDB. See the SRG lab environment description for details.

7 Programming Overview of *Choices*

This section describes briefly the internals of some of the *Choices* subsystems. It should be enough to give you a general view of how *Choices* works and where in the code to look for details. The 1991 Ph.D. thesis An Object Oriented Operating System by Vincent Russo provides a more detailed general overview of the *Choices* design. While things have changed, the basic design remains.

7.1 Processes

In *Choices* a process is a a simple thread of control running in a Domain, which is a collection of memory regions. For a multi-threaded application multiple threads are created for the same Domain. An example can be found in Applications/Examples/twoproc.cc. Section 9 provides more details on processes.

Processes are managed by objects of type class ProcessContainer. Things such as Schedulers and CPUs are ProcessContainers. When a Process is put into the CPU, the CPU is made to run that Process.

7.2 Memory

Choices supports a paged virtual memory system with multiple caching strategies. Memory regions can be shared between processes. Section 10 provides an implementation overview.

7.3 File System

Choices supports multiple file system formats. The basic file system emulates the Unix BSD file system format. The file system has been enhanced to act as a distributed file system. The distributed file system is stateless and does not enforce consistency. The distributed file system supports multiple caching strategies that can be selected on a per file basis. Section 11 provides more detail on the file system.

7.4 I/O and Devices

Choices runs on a number of platforms, each with different requirements for device I/O. For example the SPARC uses DVMA (DMA with virtual address translation through the MMU), while other platforms use traditional DMA, or I/O instructions. The device details are hidden behind standard interfaces, such as read and write commands. While the lower level interface software takes advantage of the features provided by the hardware. Record I/O is supported.

7.5 Interrupts

In *Choices* an object of class Exception is created that defines a basicRaise() method. The Exception is installed by vector number into the CPU. The processor dependent code calls hardwareInterruptAssist on the CPU, which looks up the Exception and raises it. Since any Domain may be active when the interrupt occurs, the processor hardware or processor specific assist code makes sure we are running on a stack in the Kernel Domain (each ProcessorContext defines a kernel and user stack for application processes). Ideally drivers should be structured so that there is a SystemProcess that loops doing P() on a Semaphore and the basicRaise method does a V() on the Semaphore to signal that an interrupt occurs. This keeps low level driver running on a borrowed system stack to a minimum. The basicRaise can handle the interrupt directly, but if there is an error in the driver (say an Assertion failure) the current Process will be looked up and killed even though it was not really at fault. With a separate SystemProcess an unlucky application is not killed, although the Kernel may halt anyway.

7.6 Reference Counting and Garbage Collection

Choices implements reference counting on objects by using "Refs" and "Stars" which behave like smart C++ references and pointers (*'s). The Refs and Stars implement reference counting and allow things to be passed around more easily, and automatically deleted when no longer needed. The following code shows the use of a reference counted Semaphore:

```
SemaphoreStar s = new Semaphore(); // Would have been Semaphore * =...
s->P();
extern void semaphoreUser( SemaphoreStar s);
semaphoreUser(s);
s = 0; // unreference and delete if reference count == 0
// (if semaphoreUser didn't pass it on increasing reference count)
```

7.7 First Class Classes

Choices allows access to the Class hierarchy at run time much like Smalltalk. Every class derived from Object is inserted in the Class system and can lookup access to its static and run time hierarchy information. The constructors are also available for Classes enabling creation of a member of that class from the Class hierarchy. See the file Class.h for operations on Classes.

7.8 The System Interface

Choices does not have a typical system call interface, such as UNIX. Instead, the *Choices* Kernel gives Proxy objects to Applications that they can use as if they were normal C++ objects. When a method on a Proxy is accessed it causes a trap to the Kernel which validates parameters and executes the method on behalf of the Application.

7.8.1 Proxies

Choices implements Proxies by introducing an extra level of indirection. The pointer returned to the user is really a pointer to an ObjectProxy, which contains information about the real object and a pointer to the ObjectProxy C++ vtable. The vtable is used to implement virtual function calls. The vtable points to the virtual function dispatch table for the class of the object. In the ObjectProxy case the vtable points to a table of functions in the kernel that cause a trap. The trap code then handles the call and arranges for return values. If the return value is another Kernel object an ObjectProxy is created to that Object and returned instead.

In *Choices* there is an extra keyword "proxiable" added to the C++ header (interface) files. A class that is derived from the class ProxiableObject can be accessed from an Application level ObjectProxy. Each virtual method of a class derived from the ProxiableObject class can have the proxiable keyword prepended to it allowing that method to be called on the object.

7.8.2 Use of Proxies

The strength of using Proxies is that the implementation allows you to write code that is the same whether in or out of the Kernel. For example, in an Application we can write:

```
SemaphoreStar sem = new Semaphore();
sem->P();
Console() << "Got signal\n" << eor;
```

Section 8 provides information on how to use proxies and how to make kernel methods proxiable.

7.9 System Boot and Initialization Sequence

This section describes how *Choices* boots and initializes itself.

7.9.1 Machine Dependent Basic Boot Loading

Each machine has it's own way of booting a kernel. The goal is to get the Kernel loaded into low memory with VM disabled and jump to the entry point of the Kernel (usually called _start). Some machines, such as the SPARCstation, boot with VM enabled or have other things in memory such as memory mapped I/O or EEPROM monitors. With a good bootloader the Kernel will be loaded automatically and jumped to. Other machines may need an intermediate bootstrap that the bootloader loads and then the intermediate bootloader loads and runs *Choices*. In any case we assume that the bootloader and machine dependent boot code loads the Kernel into low (1-1 mappable) memory and calls Main(0,0);, followed by processorStart();

7.9.2 Kernel Initialization

Once the bootloader has loaded the *Choices* Kernel into memory and jumped to it, Main(0,0) is called. Main sets up the boot allocator, does some initialization and calls MAIN for that particular machine. The machine dependent main creates the Console and debugging will work now. Main then calls constructors for statically allocated objects, creates the Kernel object, initializes the Class objects, and calls Kernel::initialize().

Kernel::initialize creates input and output streams for the console, creates a scheduler and builds and readies the first process, the Kernel setup process. It then continues, calling basicInitialize which sets up the physical memory map and VM constants. When basicInitialize returns, the AddressTranslation::init() is called, so that memory allocated for page tables will be 1-1 mapped since the allocation comes off the bootHeap.

Now initial VM mappings are laid out in Kernel::initialize. The Kernel has three contiguous parts: the low kernel and high kernel are separated by the ReadOnly section that contains the ObjectProxy trap tables. The high part of the Kernel contains the code and data past the read-only section, plus any memory allocated by the boot allocator, which starts at the end of the kernel and allocates memory linearly. After the Kernel MemoryObjects are inserted, MemoryObjects for mapping all of memory 1-1 are created and inserted, and finally a MemoryObject for the kernel heap is created and inserted.

At this point Kernel::initialize sets the Kernel memory allocator to be the kernelHeap, which uses the heap memory object inserted above. At this point Kernel::initialize returns to Main, which returns to the entry point function which calls processorStart.

processorStart looks up the current CPU, gets the idleProcess (created in the CPU constructor), finds the Processes context, and dispatches it. The dispatch activates the Domain on the MMU, enables VM, interrupts, etc. The idleProcess loops giving the processor away. The first process it finds to give the processor to is the Kernel setup process. The setup process calls Kernel::main.

Kernel::main sets up some object names, does some Proxy initialization, creates and initializes devices and device managing processes such as disks, ethernets, etc. builds the SystemInterface, creates a cleanup process but does not ready it, and enters the loop dispatching applications. When the dispatch loop terminates the cleanup process is readied, which brings the system down.

7.9.3 Memory Layout Assumptions

Once the Kernel has successfully booted the virtual memory layout should look something like this (although it is slightly machine dependent):

```
 --------------- 0x0
| invalid (opt.)|
 --------------- _kernelStart
|   low Kernel  |
 ---------------
|readOnly Kernel|
 ---------------
|  high Kernel |
 ---------------  &end
|   Boot Heap   |
 ---------------
|  Kernel Heap  |
 ---------------
|               |
 ---------------  _kernelStart + _kernelSize
|  Application  | _applicationStart
|               |
 --------------- _applicationEnd
|  DVMA         | dvma 0xff000000
|---------------| eeprom 0xffd00000
|  EEPROM       |
|---------------| IO_SLOT_VBASE 0XFFFE0000
|  IO_SLOT      |
|---------------| IO_BUILTIN_VBASE 0XFFFF0000
|  IO_BUILTIN   |
 --------------- 0xffffffff
```

8 Choices Proxies

Choices proxies are special objects that represent other objects across protection boundaries. They are designed to be able to be used to call into the kernel, call out of the kernel to applications, and even to call into or out of intermediate protection levels. Proxies are most commonly used to call from applications into the kernel, however, and this document only deals with that configuration. In the most recent stable as of this writing, proxies are only used to call from applications into the kernel.

Proxies are transparently inserted and removed, both from the point of view of application code and from the point of view of kernel code. A UNIX tool called Proxify++ examines header files and generates enough information to enable the proxy call handler in the kernel to automatically manage the proxies.

For more details on *Choices* proxies, please see the 1992 PhD thesis Object Oriented Hierarchies Across Protection Boundaries by David Dykstra. The examples chapter may be particularly helpful.

8.1 Making a kernel class proxiable

These are the steps to make a kernel class accessible to applications through proxies:

8.1.1 Subclass of ProxiableObject

The first step is to make sure that your class is a subclass (somewhere in the class hierarchy) of ProxiableObject. The most important feature of the ProxiableObject class is reference-counting; many classes are already subclasses of ProxiableObject just to take advantage of the reference-counting, but this alone does not make the class accessible to applications.

8.1.2 Define Class of Object

Next, make sure that your class has a 'Class' object representing it. For example, if your class is named Myclass and has a parent class of Myparentclass, you would put this at the end of your ".cc" file:

```
#include "ClassInstaller.h"
DEFINECLASS("Myclass",Myclass,Myparentclass,0,0);
```

The `DEFINECLASS()` macro defines the 'Class' object for you. In addition, the macro requires that you have two things in your ".h" file or there will be a compilation error where the macro is expanded. The two things are:

1. `extern ClassStar MyclassClass;` at global scope.

2. A method declaration in your class description: `Class * classOf();`. You don't need to actually define the method, the `DEFINECLASS()` macro does that for you.

Defining a 'Class' object may also be useful for classes that are not accessible by applications, but it is required for classes that are accessible.

8.1.3 Star Files

You will probably also want to make "Star" files for your class which define reference-counting-assist pointers for your class ('Star' and 'Ref' classes). Most "Star" files are placed in the "Common" directories. For example, for `Myclass` you should make an "Includes/Common/MyclassStar.h" and "Common/MyclassStar.cc" which define both 'MyclassStar' and 'MyclassRef'. I usually just copy existing "Star" files and do global edits on them.

8.1.4 Proxiable Methods

Decide which methods applications will be allowed to call and declare them **proxiable** in the header file. The **proxiable** keyword should be the first keyword in the declaration. Proxiable methods must also be declared **virtual** because the proxy mechanism relies on the C++ virtual function mechanism. Also, if you want applications to be able to create instances of the class with **new**, mark the constructor of the class **proxiable** (it need not and indeed cannot be **virtual**).

Note that if your class is overloading a **proxiable** method defined by one of the parents of your class, that method is automatically assumed to be **proxiable** and you do not need to re-declare the method to be **proxiable**, although it doesn't hurt. This is similar to the way the **virtual** attribute is inherited. The **proxiable** attribute of constructors is not inherited, however.

You should be very careful about which methods you make **proxiable** if you care about security. Always keep in mind that the user is not to be trusted and can pass in any value for the parameters. Make sure that **proxiable** methods will not do any harm no matter what the values are passed.

Parameters that are pointers are particularly dangerous, although the system automatically handles much of the potential problems. These are the things to remember about pointer parameters:

1. For pointers to subclasses of ProxiableObject, you would normally expect the user to pass in a pointer to a proxy to another object in the kernel. The system automatically checks to make sure that the user passes in a pointer to a valid proxy that points to a kernel object of the expected type or a subtype of the expected type. The proxy is automatically stripped off so the kernel method gets a pointer that points directly to the object.

2. (NOTE: This item only applies to outcalls, and outcalls are not supported in all stables. They were supported in Stable.9.23.1991 but support was removed in Stable.1.20.1992. Perhaps they will be supported again in later stables.)

   Applications are allowed to make their own subclasses of proxiable classes. Sometimes it may be desirable to allow them to pass in instances of their own subclasses rather than instances of kernel classes, and then later allow the kernel to call out to those objects with outcalls. Normally the system does not allow applications to pass in their own subclasses, but it does allow it if the class definition of the expected class is marked with the **proxiable** keyword. For example, if a proxiable method expects a pointer of type '`Myclass *`' and the definition of Myclass begins like:

   ```
   proxiable class Myclass : public Myparentclass
   ```

   then the application is allowed to pass in an instance of its own subclass of Myclass.

   If an application does this, the kernel is given a proxy to the application object with which the kernel can do an outcall.

3. If the pointer is to something that is not a subclass of ProxiableObject, either a **class**, **struct**, or some elementary data type like a '`char *`' pointer, the system does a range check on the pointer to make sure it is in the address space of the caller. This is to ensure that the called kernel method will only read or write in the application's address space and will not overwrite portions of the kernel itself or read from portions of the kernel that may contain sensitive data.

   Proxify++ does not allow any '`**`' pointers to be passed as parameters to proxiable methods because there is no way for the system to know the format of the data that is pointed to. Pointers to structures or classes that are not subclasses of ProxiableObject are allowed, but it is the responsibility of the proxiable method to make sure there are no pointers inside those structures or classes because they will not be flagged as errors by Proxify++ and the pointers will not be range-checked by the system. If pointers are contained in those structures or classes, users could assign them to addresses within the kernel and the kernel method would read or write those kernel addresses.

Return values of proxiable methods that are either '`*`' pointers to subclasses of **ProxiableObject** (for example `Myclass *`) or 'Ref' pointers to subclasses of **ProxiableObject** (for example `MyclassRef`) automatically have proxies allocated for them as they return the values to the application. The application may then use those proxies to invoke other proxiable methods on the returned objects. 'Ref' pointers are just like '`*`' pointer except that the reference count on the proxy is pre-incremented and the user is required to assign the return value to a 'Star' pointer (the user may assign a '`*`' return value to a 'Star' but is not required to). The advantage of 'Ref' over '`*`' is that if the user is going to assign both to a 'Star', the latter will cause an extra system call to increment the reference count on the proxy and the former will not because the reference count is pre-incremented.

8.1.5 Makefile Includes

Make sure that your header file gets added to the `PROXYINCLUDES` make variable in "Configure/System/SystemCommon.mk". Usually that is done through one of the `PROXY*INCLUDES` variables that are broken up per directory. For example, if Myclass was in the "Kernel" directory you should add "Myclass.h" to `PROXYKERNELINCLUDES`.

If you have made "MyclassStar" files, add them to the make variables in "Configure/System/-CommonSources.mk". Add "MyclassStar.h" to `COMMONSTARINCLUDES` and "MyclassStar.cc" to `COMMONSTARSOURCES`. This will make the "Star" files accessible to applications as well as to the kernel.

8.2 Using proxies from applications

If you have never written a *Choices* application before it's a good idea to first look at the example applications in "Applications/Examples".

These are the steps for an application to use a proxied kernel class:

8.2.1 Include Header File

The first thing to do is to "#include" the header file for the proxied class. Applications use the alternate header files that are generated by Proxify++; they are located in the Includes/System/<Machine> directory where <Machine> is the machine name, for example Att6386, Multimax, or SS1. Your makefile should already be set up to automatically search that directory for include files. Examine the alternate header file to determine the interface that is available for the class. Comments are not copied from the original header files into the the alternate header files, so you may also want to look at the original files.

8.2.2 Build Applications

For every proxied class for which your application includes a header file, version strings are checked when your application starts up. If the interface to the class changes but you don't re-make your application, the version will not match and a warning will be printed. If that happens, you should re-make your application.

8.2.3 Preallocated Proxies

By the time your application's `main()` is called, some proxies have already been allocated for you. Any time a proxy call returns a pointer to an object that is of a type of a sublass of ProxiableObject, another proxy is allocated for you. These are the names of the global variables that contain pointers to preallocated proxies:

1. `StandardSystemInterface` (type SystemInterface) - This proxy is to an instance of a class of miscellaneous functions into the system. You probably will not need to use it directly. An example of indirect use of this is the `thisProcess()` function which returns a pointer to the current Process object; that translates into a call to the `StandardSystemInterface`.

2. `StandardOutput` (type OutputStream) - This proxy points to the standard output stream for your application.

3. `StandardInput` (type InputStream) - This proxy points to the standard input stream for your application.

4. `StandardNameServer` (type NameServer) - This proxy points to the standard name server for your application. The primary purpose of the name server is to obtain other proxies; objects that the system wants to make available to all applications are given a name in the name server. A popular example is the 'SystemTimer'. The `StandardOutput` and `StandardInput` variables were initialized for you (by the application start up code) by looking up their names in the name server.

   There is one name server per application. If a name is not found in your application's name server, the name is looked up in the system's name server. In this way individual applications can be given private copies of some objects such as 'StandardOutput'.

   The `NameServer::lookup()` method which you use to do the lookup returns a 'Ref' pointer so you should assign the result to a 'Star' variable.

5. `*Class variables` (type Class) - A proxy for the 'Class' variable for each proxiable system class is also made available. Some examples are `ObjectClass`, `ProxiableObjectClass`, and `SemaphoreClass`[1]. The primary reason that these 'Class' variables are available is to allow you to make 'Class' objects for subclasses of these classes in your own application (more about application subclasses below).

[1] Stable.3.19.92 removed all but `ObjectClass` and `ClassClass`, but the rest will probably be restored in a future stable.

8.2.4 Creating New Proxied Objects

You can create instances of kernel proxiable classes by using **new**. You should first check the alternate header file for the proxiable class to see if the constructor is declared there; it will be there only if the constructor was marked **proxiable** in the original header file. If the original constructor was not marked **proxiable**, the alternate class will not have any constructor. If you try to do a **new** on the class it will just call the constructor of the parent class, for example the constructor of ProxiableObject. This will not cause a compilation error but the created object will not be useful. For example, if the constructor that was called was that of ProxiableObject, the created object will be a local instance of that class. If the parent class was another proxiable class which did have a **proxiable** constructor, the created object will be a proxy to an instance of the parent class.

8.2.5 Star Variables

It is generally a good idea to assign proxies to 'Star' variables, although there are situations where you can get away without doing so. To create a 'Star' variable you need to include the appropriate "Star" header file; for example, to create a new Semaphore you might use code like this:

```
#include "Semaphore"
#include "SemaphoreStar"
...
  SemaphoreStar sem = new Semaphore( 0 );
```

Each proxy that you get holds one reference to the kernel object. Proxies returned from a **new** begin with a reference count of zero, and assigning the result to a 'Star' increments the reference count to one. Reassigning the 'Star' variable to some other value will first decrement the reference count on the proxy which will delete the proxy if your application had not incremented the reference count on the proxy from somewhere else in the meantime. The deletion of the proxy will then decrement the reference count on the kernel object, deleting that if nothing else had referenced it.

If you wish you may be able to leave the reference count on a proxy at zero and then use **delete** to delete the proxy. However, if something else in your application had incremented the reference count on the proxy in the meantime, an error will be printed and your application process will be killed. These are the ways that the reference count on a proxy can be incremented:

1. By your application assigning the proxy to a 'Star' variable.
2. By your application explicitly calling the `reference()` method on the proxy (you probably should not be doing this).
3. By the proxy being returned from a method that returns a 'Ref'. When that happens, the reference count on the proxy is pre-incremented by the system, and you should definitely assign the return value to a 'Star' so that when the 'Star' variable gets reset it will unreference the proxy.

Also note that if the same kernel object is returned by more than one proxy call, the same proxy will be given to your application in both places. If that possibility exists, you should use 'Stars' for that proxy because if you used **delete** from one place in your application the proxy will get deleted, and the other place in your application will not be able to use it anymore; if you instead used 'Stars' the proxy will not be deleted until both places are finished with it.

When your application exits, all if its proxies are deleted no matter what the reference counts on them are, so you don't need to worry about cleaning them all up before your application exits.

8.2.6 Scope of New Proxied Object

For every proxiable class that you can do a **new** on, you can also declare an instance at global scope or as a local variable. For example,

```
f()
{
  Semaphore sem(0);
  sem.V();
}
```

Note that this is less efficient than using **new** because the method calls go through local stubs which forward method calls through the proxy that was created when the constructor was called, but this is supported to preserve `C++` semantics.

8.2.7 Subclassing Proxied Object

You can also make subclasses of proxiable kernel classes. When you create an instance of your subclass, the constructor of the proxiable class will automatically get called which will create a proxy to the parent portion of your object in the kernel (assuming the parent class has a **proxiable** constructor). For every method that your subclass does not overload, a method stub will be inherited which forwards the method call through the proxy to the parent portion in the kernel.

You may want to create a "Star" file for your subclass; if you do, put the file in the same directory as your application. You may also want to use `DEFINECLASS()` to define a 'Class' object for your subclass (see the example in the "Making a kernel class proxiable" section).

9 Processes in *Choices*

9.1 Process Types

Processes are classified into System and Application processes. One type of System process is not preemptible. All Application processes are preemptible. Naturally System processes execute in the kernel, while Application processes execute in user space.

9.2 Process Domain

A *Choices* process contains a thread of control and a Process Domain. The Process Domain identifies the memory regions accessible to the process. Memory regions can be shared between processes. Thus a *Choices* process can be either a heavyweight or lightweight thread depending on the degree of memory sharing.

9.3 Process Priority

Currently any ready System Process is executed prior to all Applications Processes. There is no prioritization within the set of System or Application processes.

9.4 Input and Output Parameters

Choices follows the Unix process model in regards to process arguments and return values. The *Choices* process receives two input arguments. The first argument is an integer count of the argument text strings. The second argument is a pointer to an array of text strings which represent the parsed command line arguments. The process is expected to return an integer value that in theory should represent the execution status. Often in practice no input arguments are provided and exit value is left to a default value.

9.5 Lightweight Process Creation

An application process can be created that shares the memory regions of its parent, which in effect is a lightweight thread. The child's entry point is the name of a specified routine in the parent. The program Choices/Examples/twoproc demonstrates lightweight process creation. The following is an example of the relevant commands:

```
ProcessStar * Proc =
  new AppplicationProcess(
      routineName,              // integer function in parent
      numberOfArguments,        // integer count
      pointerToArrayOfArgs);    // parsed arguments
Proc->setName( ``routineName'' );  // give it a useful name
Proc->ready();                  // tell system to run it
```

9.6 Heavyweight Process Creation

An application process can be created that has its own memory region indepent of the parent. In effect this provides a heavyweight thread. The parent can wait for the completion of the child process. The routine 'execProgram' from the file Choices/Applications/FiSh/ShellCommon.cc provides a general purpose example.

9.7 Application Examples

The directory Choices/Applications/Examples contains a set of example application programs. These programs serve two purposes. They can be used to demonstrate whether your *Choices* node is in working order. Perhaps more importantly the examples demonstrate how system services are provided.

10 Virtual Memory Management

This section presents the implementation of the virtual memory management system. Figure 5 and Figure 6 present the framework and the class hierarchy of the virtual memory management system.

10.1 AddressTranslation

The abstract AddressTranslation[2] class defines a machine architecture independent interface to the machine architecture dependent virtual memory address translation hardware. Subclasses of AddressTranslation support different computer architectures. An instance of the AddressTranslation class, known as an address translation, interacts with the machine-dependent address translation hardware to map virtual memory addresses to physical memory addresses. It has methods such as `addMapping` to add virtual-to-physical address mappings at a given protection level; `removeMapping` to remove mappings for a virtual address range; and `changeProtection` to change protection for a virtual address range.

10.2 PhysicallyAddressableUnit

An instance of the PhysicallyAddressableUnit class, known as a physically addressable unit, represents a block of continuous physical memory. In most systems, a physically addressable unit represents a page of physical memory. The PhysicallyAddressableUnit class has methods to return the physical address of a physically addressable unit, to read and modify the attributes of a physically addressable unit. These attributes include the *referenced* and *modified* status of the unit. Address translations update these attributes in `removeMapping`. Each physically addressable unit also has a *next link* attribute which is used to link several physically addressable units into a list.

10.3 Store

Each host has an instance of the Store class, known as the store. The store manages allocation of physically addressable units on the host. It owns the collection of all unallocated physically addressable units. Its clients invoke its `alloc` method with a size parameter to obtain physically addressable units from the store. The store also remembers all clients who are holding some physically addressable units. The store's clients uses its `dealloc` method to return previously allocated physically addressable units to the store. In most cases, the store's clients are instances of the MemoryObjectCache class. The store also maintains a system daemon, known as the *pager*. The pager is awakened when there are few free physically addressable units left on the host. The pager invokes the `pageOut` method of the store's clients. This causes the clients to page out physically addressable units that contain stale data[3] and return these physically addressable units to the store.

10.4 Domain

An instance of the Domain class is a domain. A domain manages the virtual address space of a context. It maintains a collection of bindings between memory objects and virtual address regions.

[2]A word in sans serif font starting with an uppercase letter denotes a class name. A word or a string of consecutive words in sans serif font starting with a lowercase letter denotes an instance or instances of a class. A word in fixed width `typewriter` font denotes a method name. For example, AddressTranslation is a class name, address translation is an instance of the AddressTranslation class, address translations are multiple instances of the AddressTranslation class, and `addMapping` is a method.

[3]The definition of stale data depends on the client's cache replacement policy.

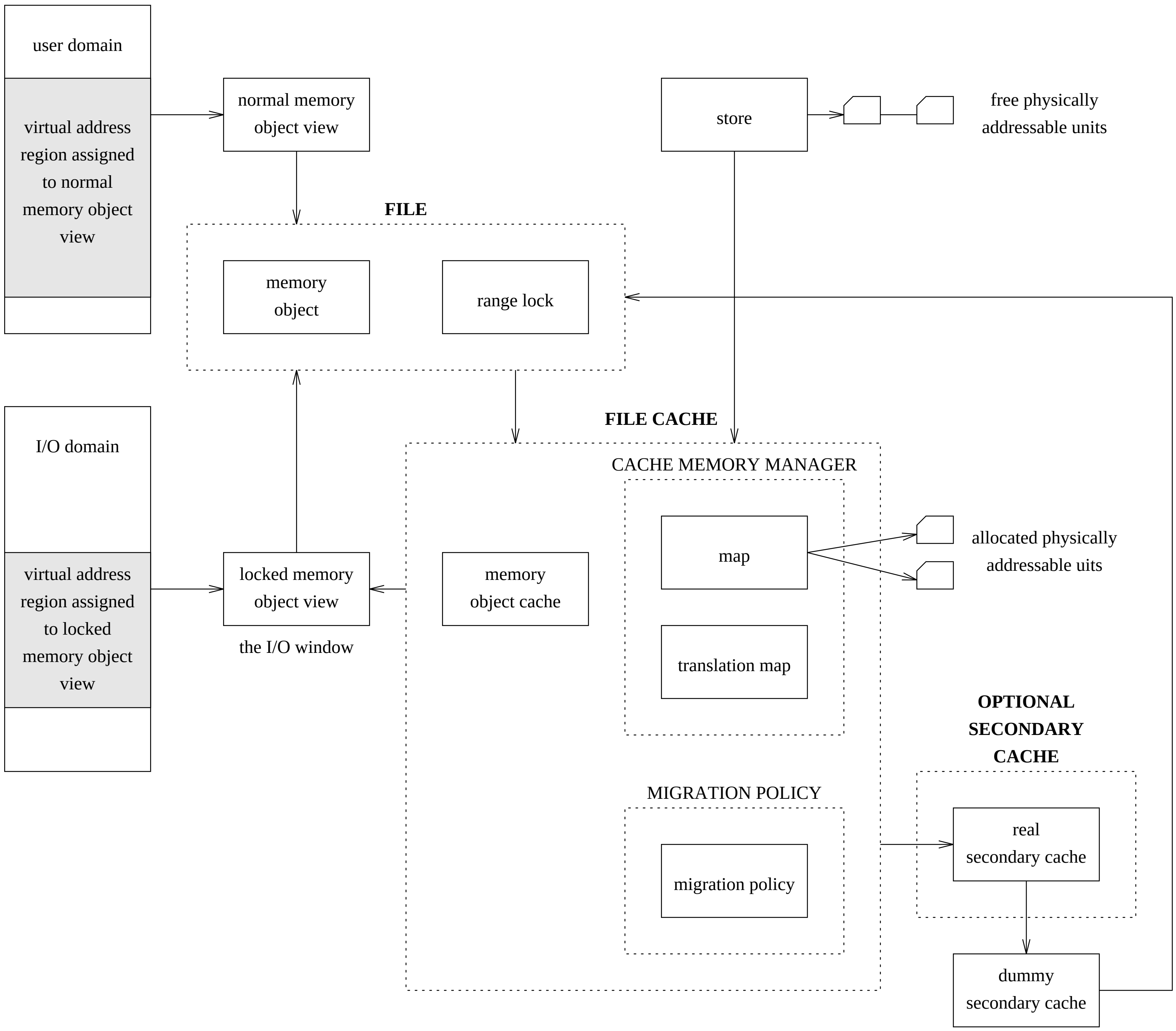

Figure 5: The framework of the revised *Choices* virtual memory management system.

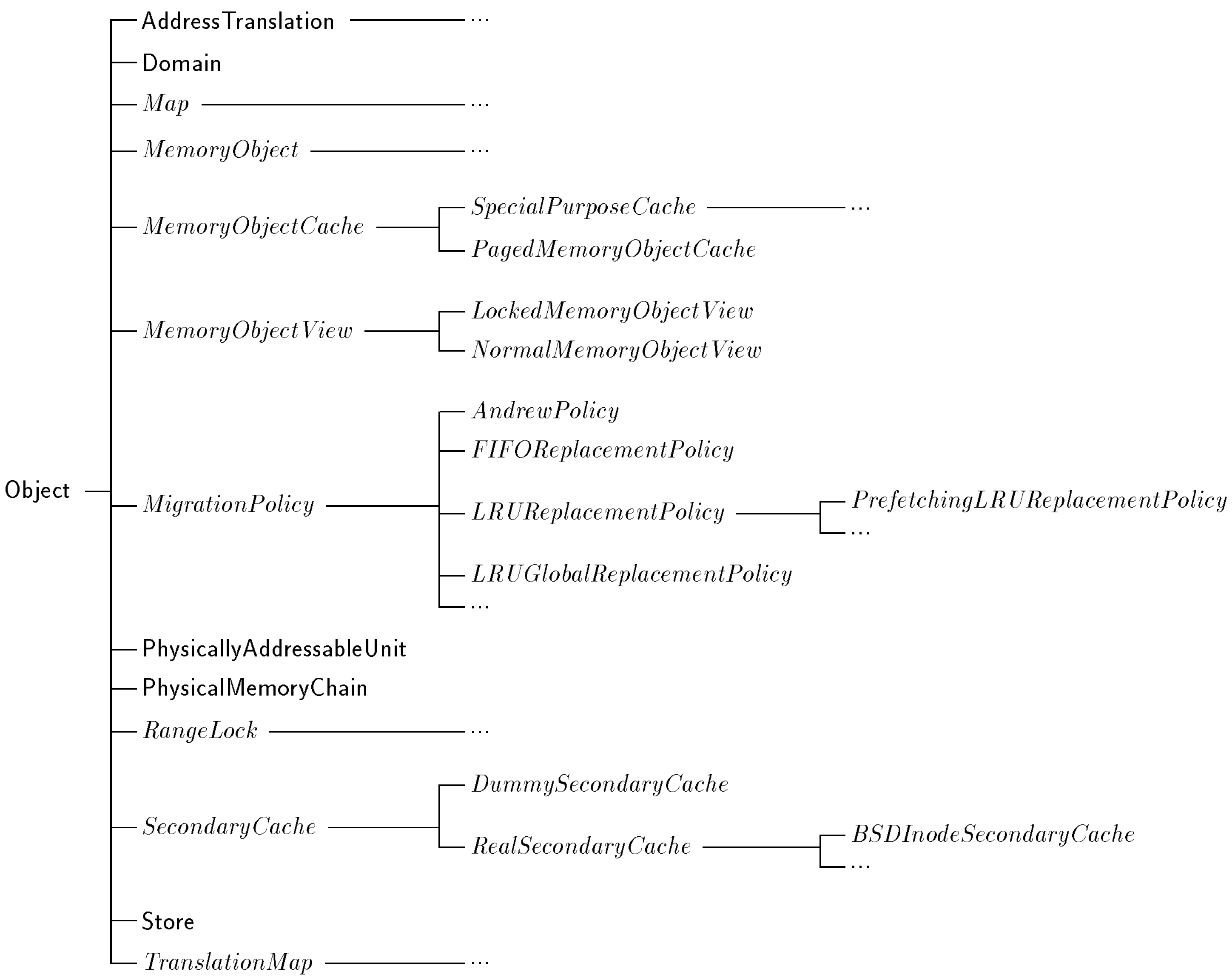

Figure 6: The class hierarchy of the revised *Choices* virtual memory management system. New or significantly revised classes are shown in *italics*.

Each domain has an associated address translation that manages the virtual-to-physical address mappings of the domain. The Domain class has a add method to map a memory object to a virtual address region, *i.e.* to add a binding to the domain. It also has a `remove` method that is used to unmap a memory object from a virtual address region, *i.e.* to remove a binding from the domain. A memory object can be mapped into more than one domain. When a page fault occurs, the exception handler calls the `repairFault` method of the faulting domain. The faulting domain is the domain whose address translation is active when the fault occurred. The `repairFault` method examines the domain's collection of bindings to determine which memory object is mapped at the faulting address. Then, it invokes the cache method of the memory object cache associated with the memory object mapped at the faulting address. As discussed earlier, this method makes the faulting address addressable. The Domain class also has a `makeResident` method and a `constructChain` method. These methods take a virtual address offset and a length in bytes as their arguments. The `makeResident` method makes sure that the virtual address region described by its arguments is resident and addressable. The `constructChain` method is mostly used by I/O device drivers. It makes sure that the virtual address region described by its arguments is resident and locked so that it cannot be selected for replacement. It returns an instance of the PhysicalMemoryChain object, known as a physical memory chain. A physical memory chain encapsulates a list of physically addressable units which have been locked. When the physical memory chain is destroyed, the acquired lock is released.

10.5 MemoryObject

Instances of the MemoryObject class are memory objects. A memory object in *Choices* is a storage entity that consists of a sequence of identically sized storage units. The attributes of a memory object include its unit size and its length in number of units. The MemoryObject class has methods to read and modify these attributes. It also has `read` and `write` methods to read and write the storage entity. These methods take as their arguments a starting unit number, number of units, and a virtual address. Subclasses of the MemoryObject class support different types of storage entities, such as disks and files.

10.6 MemoryObjectCache

An instance of the original *Choices* virtual memory system MemoryObjectCache class, known as a memory object cache, allows a memory object to be mapped into one or more continuous virtual address regions and allows portions of the memory object to be cached in physical memory. Methods of the MemoryObjectCache class operate on units which are the same size as the physically addressable unit. When a virtual address in one of these regions is accessed and the virtual address is not addressable, a page fault occurs and the cache method of a memory object cache is invoked. The cache method ensures that the unit associated with the faulting address is resident in physical memory and makes the unit addressable by the faulting process. If the unit is not resident in main memory, the cache method obtains free memory from the store and initializes the memory by invoking the `read` method of the memory object. The cache method makes the unit addressable by adding the appropriate virtual-to-physical address mappings to the faulting domain's address translation. The mcache method is similar to the cache method except it operates on more than one unit. The MemoryObjectCache class also has a `pageOut` method that is invoked by the store. This method is responsible for removing stale data from the cache and returning memory occupied by the stale data to the store. Subclasses of the MemoryObjectCache class support different caching schemes. For example, instances of the SpecialMemoryObjectCache class are used to cache kernel code and data. They never page out, *i.e.* their `pageOut` methods do nothing. Instances of the PagedMemoryObjectCache class permit page outs. Subclasses of the PagedMemoryObjectCache class implement different cache replacement policies.

The original *Choices* MemoryObjectCache class has been split up into four separate classes: the revised MemoryObjectCache class, the Map class, the TranslationMap class, and the MigrationPolicy class. In the revised virtual memory management system, each memory object cache "contains" three objects; a map, a translation map, and a migration policy. The map and translation map implements the functionality of the Cache Memory Manager. The migration policy implements the functionality of the Migration Policy module. Since a memory object cache's migration policy is defined by a separate object, it allows the memory object cache to change migration policy more easily. In the original virtual memory management system, the only way to change migration policy is to destroy the memory object cache and replace it with another memory object cache. This is inefficient because data cached by the memory object cache needs to be flushed before the memory object cache can be destroyed. The revised MemoryObjectCache class has a `setMigrationPolicy` method that is used to select and change the migration policy of a memory object cache.

10.7 SecondaryCache

A memory object cache only interacts with its top-most secondary cache. Each secondary cache interacts with its parent and its child. Its parent is the secondary cache *above* itself on the stack. Its child is the secondary cache *below* itself on the stack. The top-most secondary cache has no parent and the bottom-most secondary cache has no child. The SecondaryCache class defines the protocol used by a secondary cache to coordinate caching activities between its parent, its child, and itself.

The SecondaryCache class has two subclasses; the DummySecondaryCache class and the RealSecondaryCache class. The DummySecondaryCache class is an implementation convenience. It allows a memory object cache to do all its I/O using a secondary cache. It ensures that at least one secondary cache is attached to a memory object cache. The dummy secondary cache[4] "caches" data in the original memory object. It is always the bottom-most secondary cache. Its `read` method reads from the original memory object and its `write` and `rawWrite` methods write to the memory object. It does nothing for invalidation and synchronization requests.

Subclasses of the RealSecondaryCache class implement secondary caches that actually do cache data. For example, an instance of the BSDInodeSecondaryCache class caches data in a BSD inode. It can be used to cache data from a remote memory object on the client's local BSD file system.

10.8 MigrationPolicy

Subclasses of the MigrationPolicy class implements different cache replacement policies and different transfer policies. Transfer policies include prefetch strategies and write strategies. Each memory object cache has an associated migration policy. The MigrationPolicy class has methods which represent different events. A memory object cache invokes these methods when these events occur.

A global page replacement policy can be implemented by cooperating migration policy objects. For example, the LRUGlobalReplacementPolicy class implements the global LRU cache replacement policy. Its instances update a shared data structure.[5] When a memory object cache delegates a `pageOut` invocation to an instance of this class, the shared data structure is consulted to determine which is the least recently used unit among all the memory object caches that are using the same

[4] A dummy secondary cache is an instance of the DummySecondaryCache class.

[5] In this case, a *static* member variable in C++.

global LRU cache replacement policy. The least recently used unit paged out need not belong to the memory object cache which received the page out request.

10.9 Map

Each memory object cache has a map. It uses this map like a database to keep track of which regions of its memory object is currently cached in main memory and the physically addressable units which are used to cache these regions. Several subclasses of the Map class have been implemented. Each subclass uses a different data structure to maintain its database.

10.10 TranslationMap

Each memory object cache has a translation map. Since a *Choices* memory object can be mapped into multiple domains, virtual-to-physical address mappings for the memory object may exist in multiple address translations. The translation map interacts with these address translations. It maintains a database of which units have mappings in which address translations. It consults this database when its memory object cache directs it to unmap one or more units from all address translations. This normally occurs before the memory object cache is about to page these units out. Different subclasses of the TranslationMap class use different data structures to maintain their databases.

10.11 MemoryObjectView

A memory object view[6] represents a continuous region of a memory object. It is defined by an offset into a memory object and a length which determines the size of the region. In the revised virtual memory management system, only memory object views are added to a domain. To map an entire memory object into a domain, a memory object view representing the entire memory object is added. Memory object views are required for the following reasons:

1. *To allow large* memory objects *to be memory-mapped -*
   When a memory object is too large to be memory-mapped because of a lack of virtual address space, memory object views allow parts of the memory object to be memory-mapped.

2. *To allow efficient loading of applications -*
   A *Choices* application program is stored in a persistent memory object. There are at least two *sections* in an application program; the *text* section and the *bss* section. The text section contains read-only executable code. A memory object view allows the text section of an application program to be memory-mapped without copying. Without memory object views, the text section of an application program must be copied to a new memory object before it can be memory-mapped.

3. *To allow a* memory object *to change its size after mapping -*
   In the original virtual memory management system, the size of a memory object determines the amount of virtual address space reserved for it in a domain when it is memory-mapped. This causes problems if a memory object changes its size while it is memory-mapped. Adding a memory object view to a domain instead of a memory object separates virtual address space reservation from the size of the memory object. While the size of a memory object may change, the size of a memory object view never changes. A memory object view also allows a memory object that is expected to grow in size to reserve more virtual address space in a domain than

[6] A memory object view is an instance of the MemoryObjectView class.

the current size of the memory object. This is accomplished by adding a memory object view which is larger than the memory object to a domain.

The abstract MemoryObjectView class has two concrete subclasses: the NormalMemoryObjectView class and the LockedMemoryObjectView class. A normal memory object view[7] is used by an application to map a region of a memory object into a domain. A locked memory object view[8] is used by a memory object cache to support I/O.

10.12 MemoryObject

The MemoryObject class has been revised significantly to support file system caching. In particular, concurrency control has been improved and the I/O methods of the MemoryObject class have been extensively revised.

10.12.1 Concurrency control

In the original virtual memory management system, there are two different locks which can be acquired for each physically addressable unit. One of these locks is maintained by the physically addressable unit, *i.e.* the physically addressable unit has a *locked* attribute. The other lock is maintained by a lock manager which is a part of the original memory object cache that the physically addressable unit is allocated to. These locks introduce several concurrency control problems. They are inefficient because two locks need to be required. They are also complicated because the programmers of the virtual memory management system must be careful to avoid deadlocks. In addition, the original memory objects did not have concurrency control to ensure integrity and consistency in a multi-processing environment.

The revised virtual memory management system addresses these concurrency control problems. It requires each memory object to have a lock manager, *i.e.* an instance of the RangeLock class. A revised memory object cache no longer has its own lock manager, it uses the lock manager of its memory object. A single lock manager shared between a memory object and its memory object cache allows the two objects to cooperate better with each other to maintain mutual consistency. Good cooperation is essential since a method in one of these objects will frequently need to invoke methods in the other to accomplish its task. For example, a memory object cache servicing a page fault may need to read data from its memory object.

The original lock associated with each physically addressable unit has also been removed. The purpose of this lock was to prevent a physically addressable unit from being selected for replacement. In the revised virtual memory management system, this lock is no longer required. Once a physically addressable unit is allocated to a memory object cache, the physically addressable unit belongs to the memory object cache. Hence, the memory object cache controls whether the physically addressable unit is eligible for replacement.

Since a single lock manager is shared by each pair of memory object and memory object cache, the revised virtual memory management system significantly reduces the complexity of programming the system to eliminate deadlocks and concurrency control problems

10.12.2 I/O methods

The original MemoryObject class has two I/O methods: the `read` and `write` methods. These methods read and write to the memory object directly. With the integration of file system caching,

[7]A normal memory object view is an instance of the NormalMemoryObjectView class.
[8]A locked memory object view is an instance of the LockedMemoryObjectView class.

these methods are no longer sufficient. At least two sets of I/O methods are required. The first set of I/O methods does I/O directly from and to memory objects like the original `read` and `write` methods. The revised virtual memory management system uses these methods to read data into main memory and write data out of main memory. The second set of I/O methods supports file system I/O. These methods do I/O through a memory object cache if a memory object is cached.

In the revised virtual memory management system, the original `read` and `write` methods of the MemoryObject class has been renamed `rawRead` and `rawWrite`.[9] The `rawRead` and `rawWrite` methods do I/O directly from and to memory objects. Then, new `read` and `write` methods are defined. If a memory object is cached, these methods invoke the `cacheRead` and `cacheWrite` methods of the memory object's memory object cache. If a memory object is not cached, these methods invoke the memory object's `rawRead` and `rawWrite` methods.

10.12.3 Support for containers

A memory object container is backed by a memory object,[10] known as the *containing* memory object, and it contains memory objects,[11] known as the *contained* memory objects. A contained memory object translates an I/O request to one or more I/O request(s) to its containing memory object. For example, a BSD container is backed by a disk partition and it contains BSD inodes. A BSD inode translates a `rawWrite` invocation to one or more `write` invocation(s) on its containing disk partition. The revised virtual memory management system allows both the containing and the contained memory objects to be cached using memory object caches. This introduces a need to distinguish writes of uncommitted data from writes of committed data, much like the SecondaryCache class does. For uncommitted data, the data can be cached by the memory object cache of the containing memory object. For committed data, write-through is required. For example, committed data from a BSD inode should be written through the memory object cache of the BSD inode's disk partition.[12]

---

[9] The decision to rename the original `read` and `write` methods is based on the number of lines of source which must be changed. In this case, there are fewer references to these methods in the virtual memory management system than the file system.

[10] Actually, a memory object container is backed by a persistent memory object.

[11] Actually, a memory object container contains persistent memory objects.

[12] Assume that the disk partition is cached.

11 File System

11.1 The Original *Choices* File System

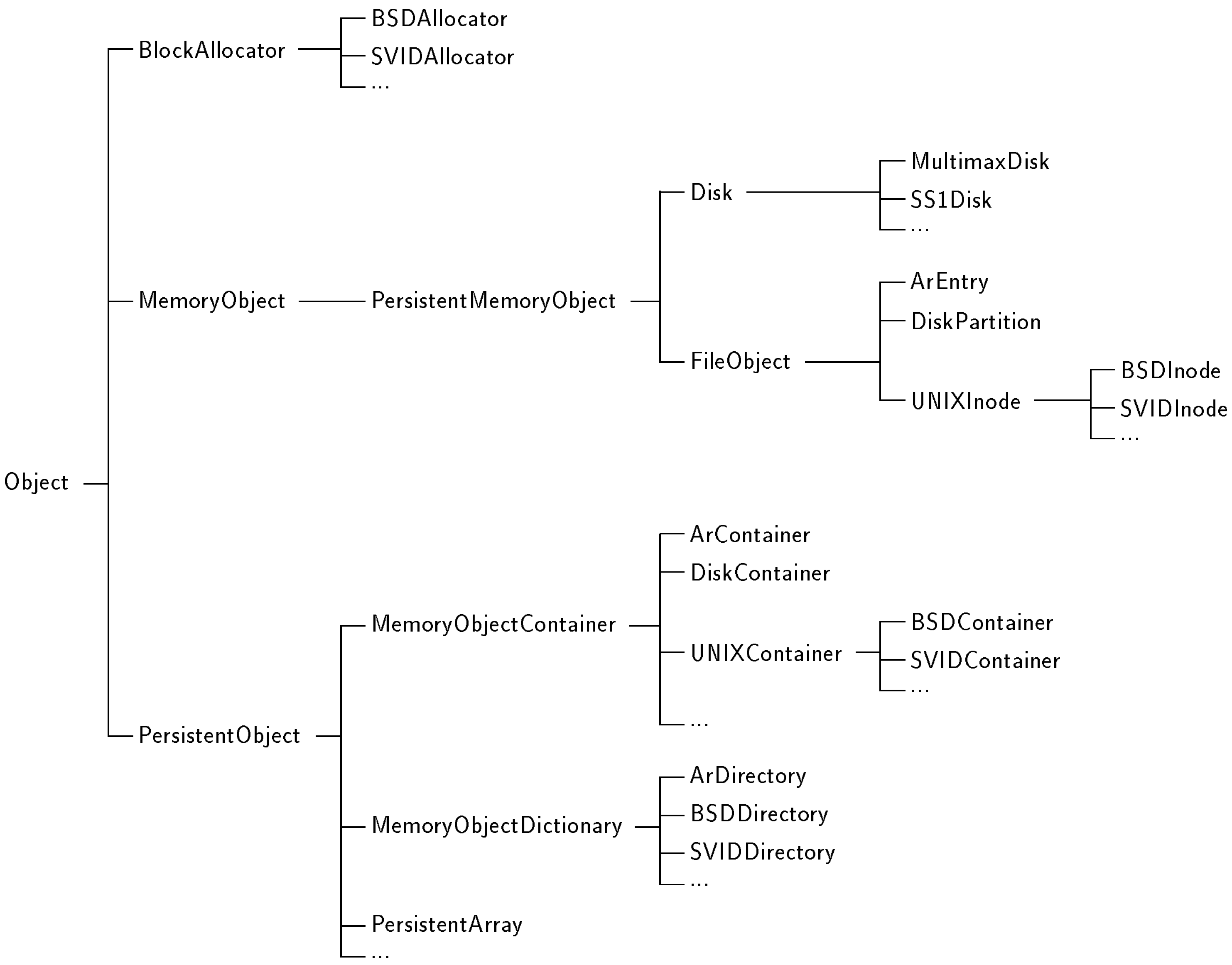

Figure 7: The class hierarchy of the original *Choices* file system.

Figure 7 presents the class hierarchy of the original *Choices* file system. Figure 8 presents an example of how objects of various classes are composed to implement a BSD file system. There are two main classes of objects in the *Choices* file system framework; the PersistentMemoryObject class and the PersistentObject class represented by rectangles and ovals, respectively.

The PersistentMemoryObject class is a subclass of the MemoryObject class. A persistent mem-

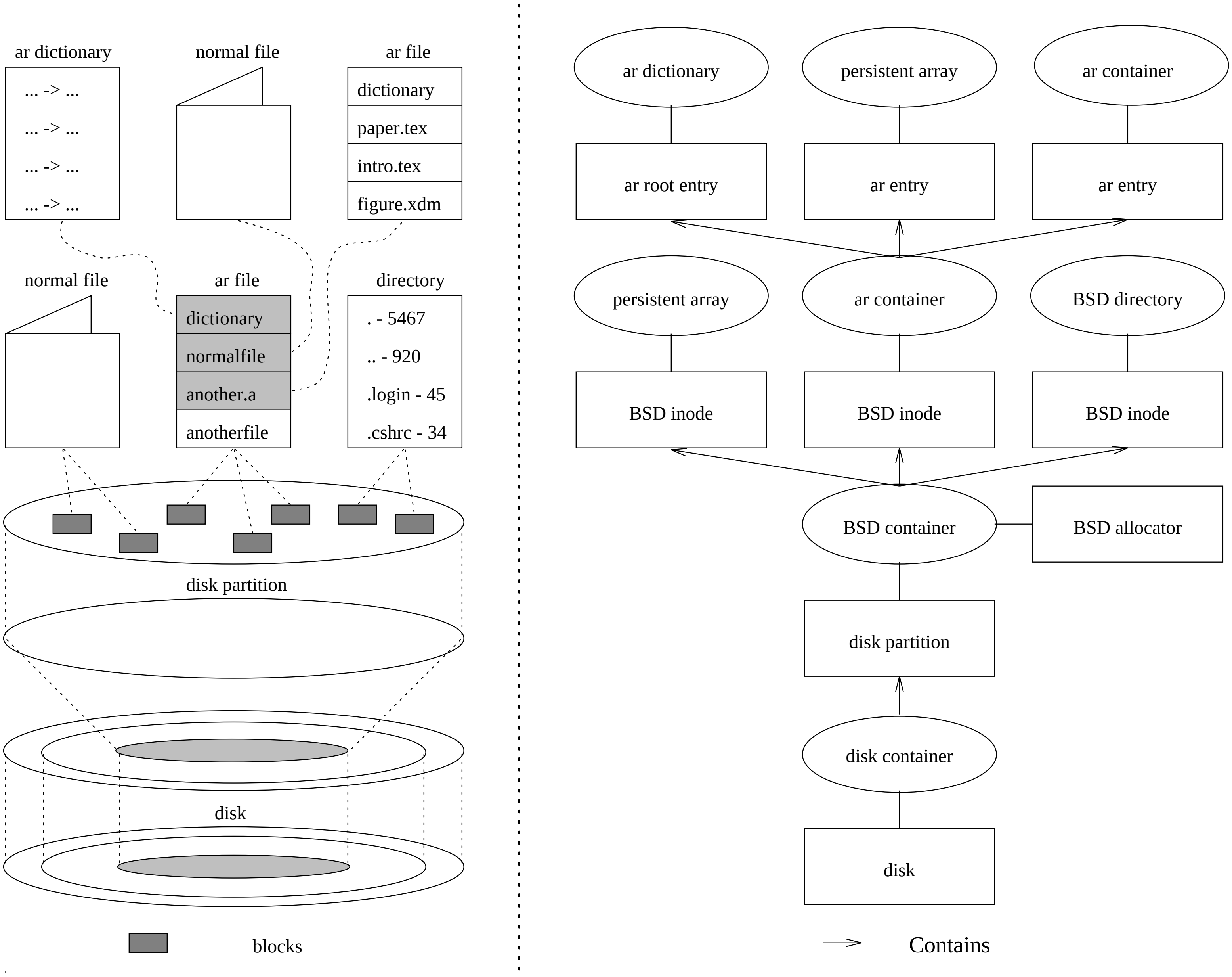

Figure 8: The framework of the original *Choices* file system.

ory object[13] has all the properties of a memory object and is persistent. Like a memory object, a persistent memory object has a unit size and a length in number of units. A persistent memory object is essentially an uninterpreted sequence of persistent unit-sized blocks, *i.e.* its `read` and `write` methods retrieve and store unit-sized blocks of persistent data. Different subclasses of the PersistentMemoryObject class support different kinds of persistent memory objects. All Persistent-MemoryObject subclasses have the same *public*[14] interface. For example, the Disk class is a subclass of the PersistentMemoryObject class. It supports physical disk devices. The FileObject class supports persistent memory objects whose persistent storage is provided by another persistent memory object. The MemoryObjectPartition class, the BSDInode class and the ArEntry class are subclasses of the FileObject class. A disk partition represents a partition on a disk drive and its storage is provided by a disk. A BSD inode represents a BSD file system inode and its storage is provided by a disk partition. An ar entry represents storage occupied by an entry in an `ar` archive and its storage is provided by a BSD inode.

An instance of the PersistentObject class is a persistent object. A persistent object is always associated with a persistent memory object. A persistent object encapsulates and interprets persistent data stored in its persistent memory object. It invokes its persistent memory object's `read` and `write` methods to access persistent data stored in its persistent memory object. There are three main types of persistent objects or subclasses of the PersistentObject class. They are the MemoryObjectContainer class, the MemoryObjectDictionary class, and the PersistentArray class.

An instance of the MemoryObjectContainer class, known as a memory object container, contains an indexed collection of persistent memory objects. It has `open`, `create`, and `close` methods to open, create, and close member persistent memory objects. For example, an instance of the DiskContainer class contains a collection of disk partitions, and an instance of the BSDContainer class contains a collection of BSD inodes.

An instance of the MemoryObjectDictionary class, known as a memory object dictionary, implements a naming service. A memory object dictionary typically contains a collection of string key to number associations. This number is an index into a memory object container's collection of persistent memory objects and it is used to open the persistent memory object associated with a particular key. Methods of the MemoryObjectDictionary class include `open` to open a persistent memory object associated with a given key; `keys` to list all the keys in a memory object dictionary; `associations` to enumerate key to index number associations; `add` to associate a persistent memory object with a new key; and `remove` to delete an association. For example, the BSDDirectory is a subclass of the MemoryObjectDictionary class. Each of its instances represent a BSD file system directory.

The PersistentArray class has methods that allow a persistent memory object to be accessed like an array. Its methods include `at` to retrieve data and `atPut` to store data.

Some memory object containers may contain a dynamic number of persistent memory objects or they may contain persistent memory objects whose size may change. The BlockAllocator class compliments these memory object containers by defining a protocol for allocating and deallocating units from the container's underlaying persistent memory object to the contained persistent memory objects. Subclasses of the BlockAllocator class compliment different subclasses of the MemoryObjectContainer class. Each subclass of the BlockAllocator class implements an allocation policy that is appropriate for its associated MemoryObjectContainer class. For example, the BSDAllocator class compliments the BSDContainer class and implements the BSD file system block allocation policy.

[13] A persistent memory object is an instance of the PersistentMemoryObject class.
[14] As in *public* methods in C++.

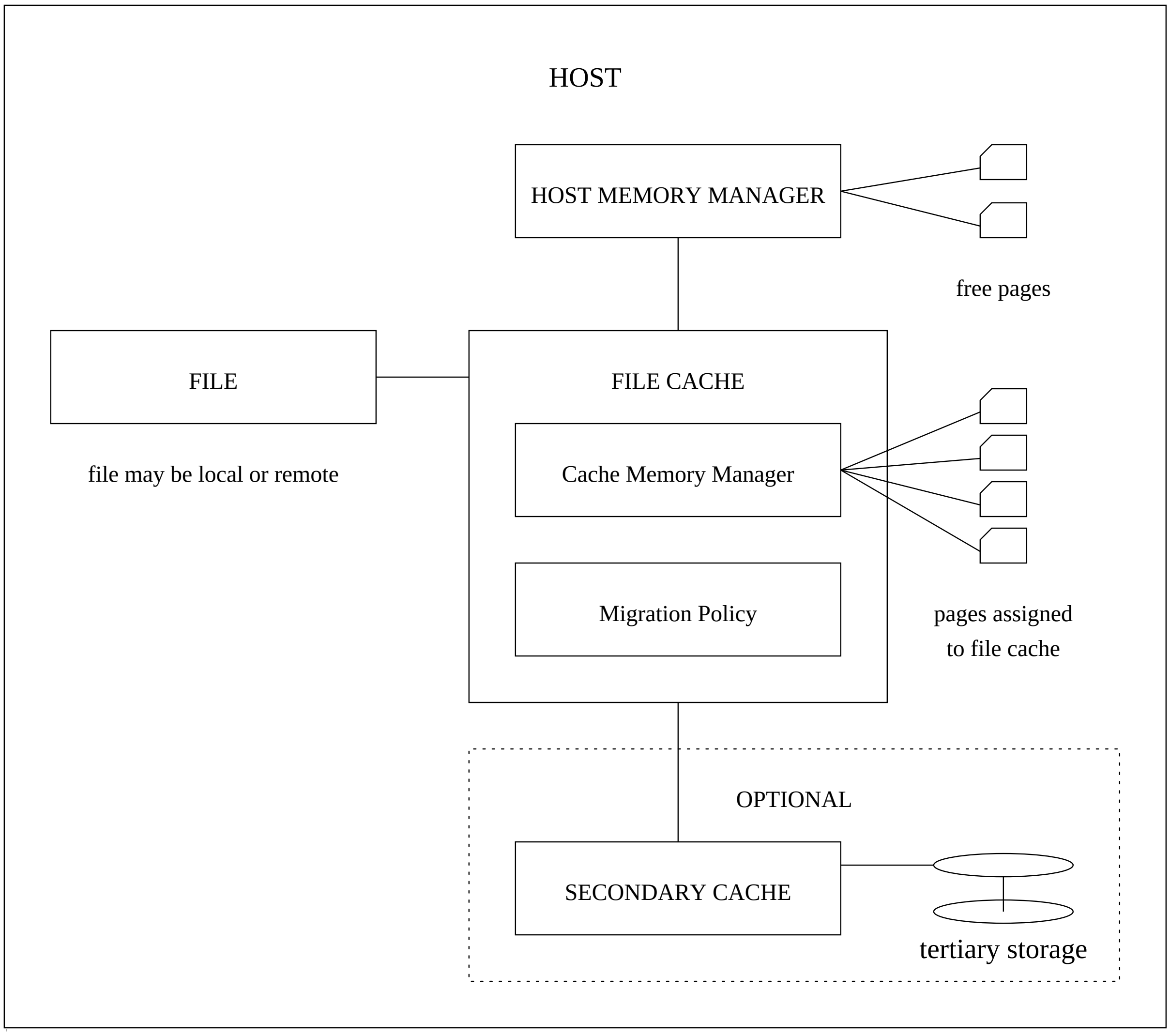

Figure 9: The architecture of a flexible file system caching service.

11.2 Distributed File System Architecture

This section presents the architecture for a distributed file system.

Figure 9 presents an overview of the architecture. *File* is the in-memory representative of a file on the host. It is similar to a *vnode* in SunOS[15] and a file's *object descriptor* in Sprite. It can represent either a local file or a remote file.

Each host has a single *Host Memory Manager*. The Host Memory Manager manages the allocation of main memory on the host. It distributes cache memory to each cached file on the host. The memory allocated to each file is managed by the file's *File Cache*. In other words, the Host Memory Manager distributes available cache memory to the File Caches. When the host is about to run out of memory, the Host Memory Manager will ask some or all of the File Caches to return some of the memory allocated to them. Different hosts can have different Host Memory Managers which implement different main memory allocation policies. These policies determine how main memory is distributed to the various File Caches on the host. One possible policy is to allocate

[15] SunOS is Sun Microsystems Inc.'s variant of UNIX.

memory to each File Cache in proportion to the frequency of new memory allocation requests from the File Cache. Another possible policy is to allocate equal amount of memory to each File Cache on the host.

Each cached file on a host has a File Cache. The File Cache has attributes and contains modules that determine how data from the file is cached in main memory, the transfer policy used to move data in and out of main memory, and the cache replacement policy. The *cache unit size* is an attribute of the File Cache. It is equivalent to the block size in other distributed file systems. Data transfers will always take place in sizes which are multiples of the cache unit size. A large cache unit size forces large transfers.

The *Cache Memory Manager* manages memory that has been allocated by the Host Memory Manager to the File Cache. It interacts with the Host Memory Manager to obtain more memory for the cache and to return allocated memory to the Host Memory Manager. It will also always allocate and free memory in multiples of the cache unit size. In addition, the Cache Memory Manager keeps track of which part of the file is cached, and where it is cached in main memory.

The *Migration Policy* module defines the cache replacement policy and the transfer policy. The transfer policy determines when and how data from the file is transferred in and out of the File Cache. Prefetch strategies are a part of the transfer policy. Possible prefetch strategies include *fetch on-demand*, *pre-fetch next unit*, and *pre-fetch all on initial access*. Write strategies, such as *write-through*, *delayed-write*, and *write-on-close*, are also a part of the transfer policy. Different Migration Policy modules implement different transfer and replacement policies. For example, a Migration Policy module optimized for large sequential file accesses may implement the MRU cache replacement policy and the *pre-fetch next unit* pre-fetch strategy. Another Migration Policy module optimized for small files may implement the LRU cache replacement policy and the *pre-fetch all on initial access* pre-fetch policy. The transfer policy may also determine if Secondary Cache(s) are attached to the File Cache.

Although each File Cache has its own Migration Policy module, *global*[16] cache replacement policies can be supported because multiple Migration Policy modules can collaborate to define a global policy. For example, a global LRU cache replacement policy can be implemented by attaching the same type of Migration Policy modules to all File Caches on the host. These modules cooperate to determine which unit is the least recently used unit in their File Caches.

The optional Secondary Cache allows data from a file to be cached in tertiary storage. It caches data in tertiary storage on behalf of the File Cache. It allows a tertiary storage system to act like a backing store for the File Cache. Different types of Secondary Cache(s) support different types of tertiary storage systems. Multiple Secondary Caches may be "stacked" to mirror the memory hierarchy of the host. The Secondary Cache allows the file system caching service to exploit the memory hierarchy of a host. A File Cache only has to attach the appropriate Secondary Cache(s) to itself to use the available tertiary storage systems on the host for caching. A good implementation of the architecture should allow Secondary Cache(s) to be attached and detached at any time. For efficiency, it should also allow the File Cache to page data directly from the file into main memory without copying the data into the Secondary Cache(s) first.

Figure 10 presents an example of how this architecture is extended to work in a distributed system. In this example, the client invokes the server to open a file. This file resides on a CD-ROM in the server. First, the server opens the file by creating an in-memory representative for the file,

[16] A *local* cache replacement policy selects unit(s) for replacement from within a single File Cache. A *global* cache replacement policy selects unit(s) for replacement from all the File Caches on single host that are using the same cache replacement policy. Unless specified otherwise, a cache replacement policy refers to a local cache replacement policy.

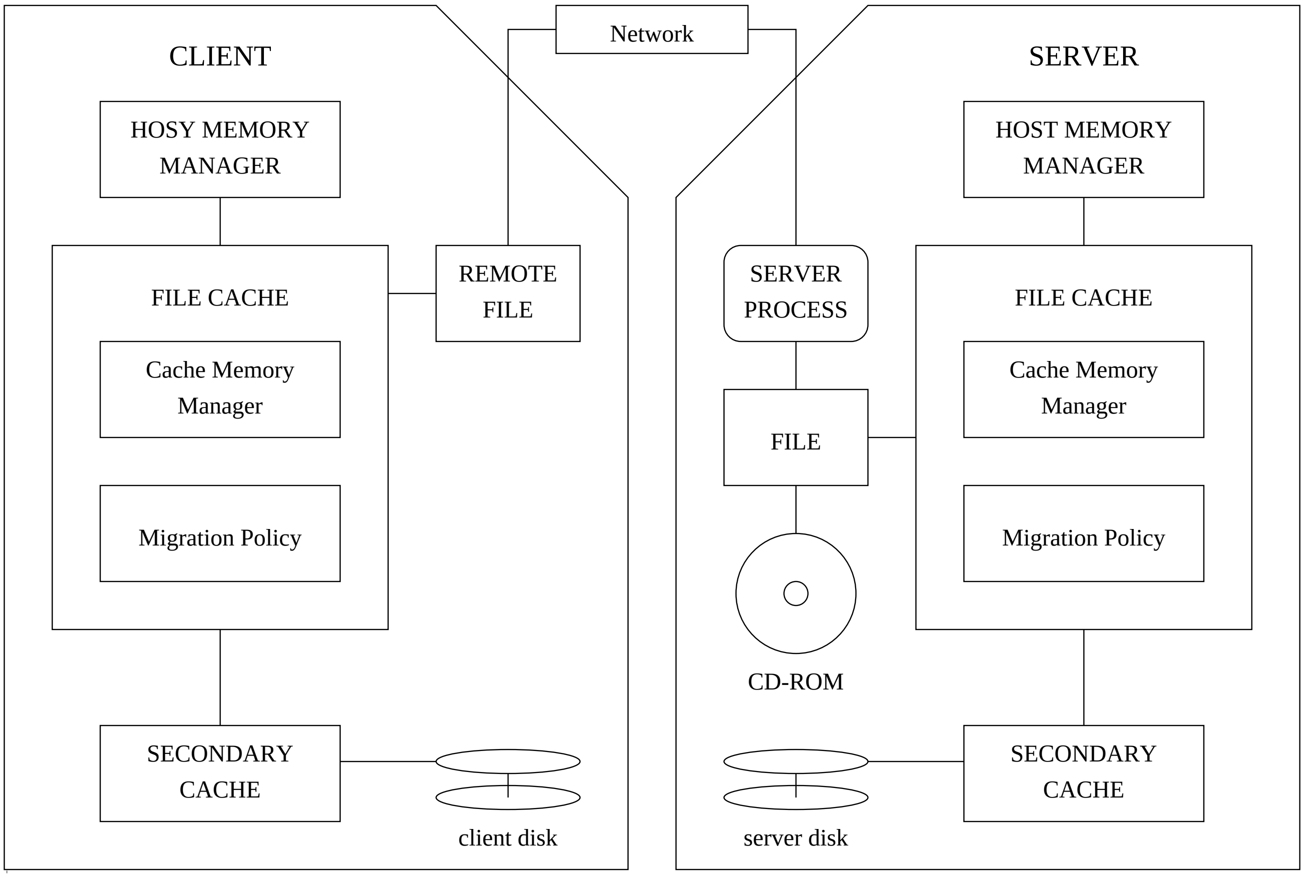

Figure 10: Distributed file system caching.

i.e. the File on the server. Then, the server attaches a File Cache to the File. The File Cache allows data from the file to be cached in main memory. In addition, the server also attaches a Secondary Cache to the File Cache. This particular Secondary Cache allows the server to cache the file's data in a disk that is faster than the CD-ROM. This capability enables faster subsequent accesses to the contents of the CD-ROM resident file by the same or different clients. The Secondary Cache is optional and is used primarily to improve performance.

The Migration Policy module of the File Cache on the server determines how the file's data is transferred from the CD-ROM into the server's main memory and into the disk cache. One such module may copy the entire file into the disk cache on initial access. A different Migration Policy module may bypass the disk cache and page data directly into the server's memory on demand. It will write to the disk cache only when it has to page data out to free memory. After the server has opened the file successfully, it sends a reply message containing a token or an identifier which represents the file to the client.

When the client receives the reply from the server, it encapsulates the returned token or identifier in the Remote File. The Remote File represents the CD-ROM resident file on the client. Then, the client attaches a File Cache to the Remote File so that it can cache remote file data in it's main memory. The client also attaches a Secondary Cache to the File Cache. This particular Secondary Cache allows the client to cache remote file data in a local disk. This capability increases the amount of memory on the client that can be used to cache data from remote files. The client File Cache's Migration Policy module determines how data from the remote file is transferred from the file server into the client's main memory and into the client's disk cache.

This architecture is flexible because each file can have a different caching strategy. In addition, the caching strategy used on the server need not be the same as the caching strategy used on the client. It is customizable because the application and the distributed operating system can "mix and match" components and modules of different types to define caching strategies that best suit the needs and requirements of the distributed computing environment. It is also extensible since new components that implement new policies can be added easily. It can exploit host, network, and application file access characteristics with customized caching strategies. For example, the architecture can address network latency problems with large cache unit sizes, client disk caching, and suitable pre-fetch strategies.

11.3 Distributed File Access in *Choices*

A *Choices* "file" is represented by a persistent memory object which encapsulates the persistent storage of the file, and a persistent object which interprets the data in the persistent storage. This section presents a brief overview of our extensions to the *Choices* file system to permit distributed access to the files in *Choices*, *i.e.* to make the *Choices* file system a distributed file system. The *Choices* distributed file system supports two distinct remote access strategies. This section also presents some of the optimizations in the distributed file system to improve performance.

11.3.1 Strategies

The *Choices* distributed file system supports two strategies.

The first strategy is to permit remote access to the methods of a persistent object. It allows a remote client to invoke remotely the methods of a persistent object using an RPC-like scheme. For example, a remote client can invoke the `add` and `remove` methods of a memory object dictionary on a server to manipulate the dictionary. This strategy has a straightforward implementation; a proxy object is used on each client to represent a persistent object.

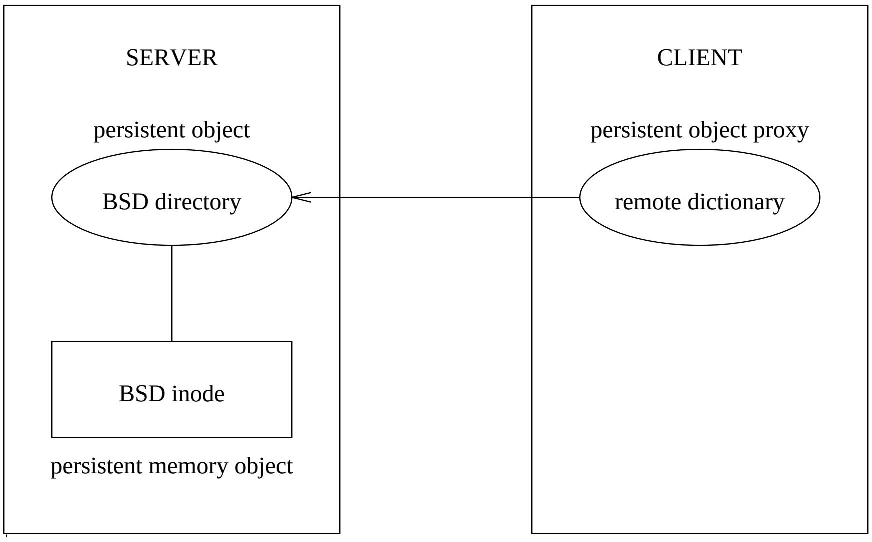

Figure 11: Remote access of persistent object

Figure 11 illustrates an example of the first strategy. In this example, the file is a BSD file system directory. Its proxy on the client is the remote dictionary.[17] Method invocations on the remote dictionary is forwarded to the BSD directory on the server. A similar strategy is used by NFS for directory operations.

Major advantages of this strategy are that it preserves the exact semantics of the *Choices* file and does not require additional concurrency control support from the distributed file system. A disadvantage is that network latency, parameter marshaling, and server contention can make it inefficient. The server on which a persistent object resides may become a bottleneck since the server is required to handle every method invocation. This strategy also does not permit client caching.

The second strategy is to permit remote access to the persistent storage of a *Choices* file. This is also accomplished through the use of proxy objects. In this case, a proxy object, known as a remote memory object, is used on each client to represent a persistent memory object. A client can access the persistent storage of a persistent memory object by invoking the I/O methods of the remote memory object. Alternatively, it can memory-map the remote memory object and access the persistent storage using memory reads and writes. A remote memory object is just like any other memory object. It can be cached by attaching a memory object cache to itself. This allows a client to cache data from a persistent memory object that resides on a remote server. In other words, client caching of a file is activated by attaching a memory object cache to a remote memory object.

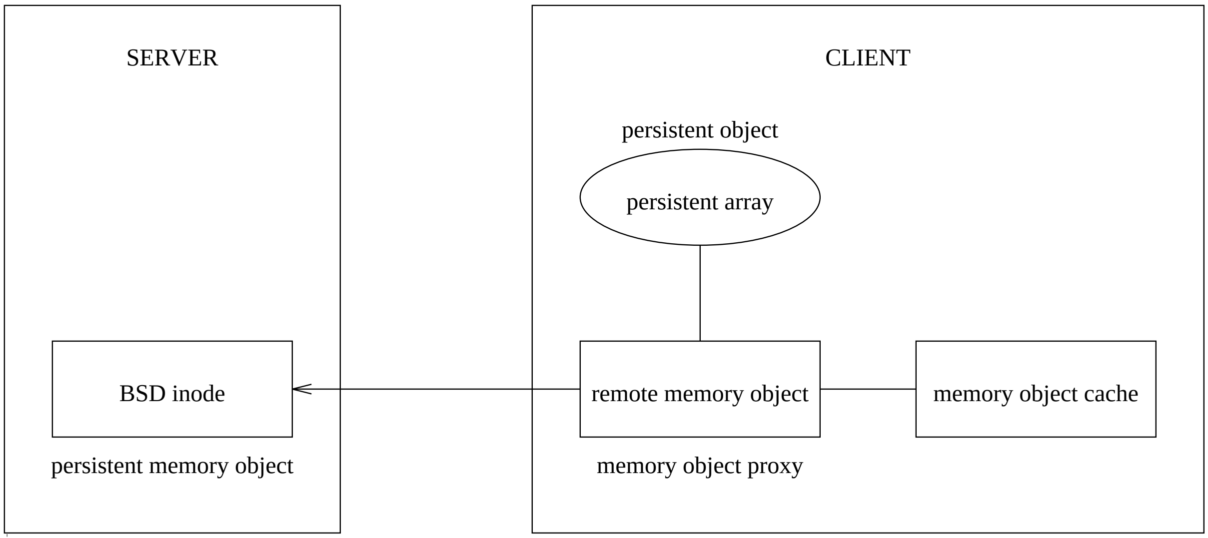

Figure 12: Remote access of persistent storage (or persistent memory object)

Figure 12 illustrates an example of the second strategy. In this example, the *Choices* file is a normal file in the BSD file system, which in *Choices* is represented by a persistent array. Method invocations on the remote memory object is forwarded to the BSD inode on the server. The remote memory object is cached in this example.[18] The client invokes the I/O methods of the remote memory object to page persistent data into and out of the client's memory object cache. A similar strategy is used by LOCUS to cache directory data.

The main advantage of the second strategy is that it permits client caching. However, the second strategy also introduces additional cache coherence and concurrency control problems when

[17]All subclasses of the MemoryObjectDictionary class have the same public interface as the MemoryObject-Dictionary class. Hence, a single kind of memory object dictionary proxy, known as the remote dictionary, is sufficient to represent all kinds of memory object dictionaries.

[18]The BSD inode can be cached as well.

a file is cached by more than one client. There are many ways to address these problems. presents how these problems are addressed in the context of *Choices*.

11.3.2 File access hints

The *Choices* distributed file system uses file access hints to help select appropriate caching strategies for a file. These hints may be supplied by the application, the persistent memory object, and/or the persistent object.

11.3.3 Zombies

Choices uses reference counting for garbage collection. When an application opens a file, a pointer object is returned to the application. In most cases, this pointer object references a persistent object. When this pointer object is destroyed, the reference count of the persistent object is decremented. When the reference count reaches zero, the persistent object is destroyed. Hence, an application closes a file by destroying the pointer object returned by the file system. When a persistent object is destroyed, its memory object cache and its persistent memory object are also destroyed. This is because the persistent object references its persistent memory object and its persistent memory object references its memory object cache. Destroying a memory object cache flushes the file's data from main memory. This is not desirable since the file may be opened again soon.

The distributed file system addresses this problem with *zombies*. Zombies are objects which are not referenced and should be garbage collected. The MemoryObjectCachingContainer class is a subclass of the MemoryObjectContainer class. A memory object caching container[19] can "cache" zombies. When a file, whose persistent memory object is contained in a memory object caching container, is closed, its persistent memory object and its persistent object become zombies (with the help of a reference counting trick). A zombie persistent memory object allows its memory object cache to stay "alive" because it still holds a reference to its memory object cache. When the file is opened again, its persistent memory object and its persistent object are referenced and they become "un-zombified". A memory object caching container eventually destroys or garbage collects its zombies. Figure 13 shows the additions and changes to file system class hierarchy.

11.3.4 Prefix caching

The distributed file system also implements prefix caching. Prefix caching reduces remote directory lookups. In *Choices*, remote memory objects and remote dictionaries on a host collaborate to maintain a prefix table. A prefix table caches associations of path names to remote memory objects.

[19] A memory object caching container is an instance of the MemoryObjectCachingContainer class.

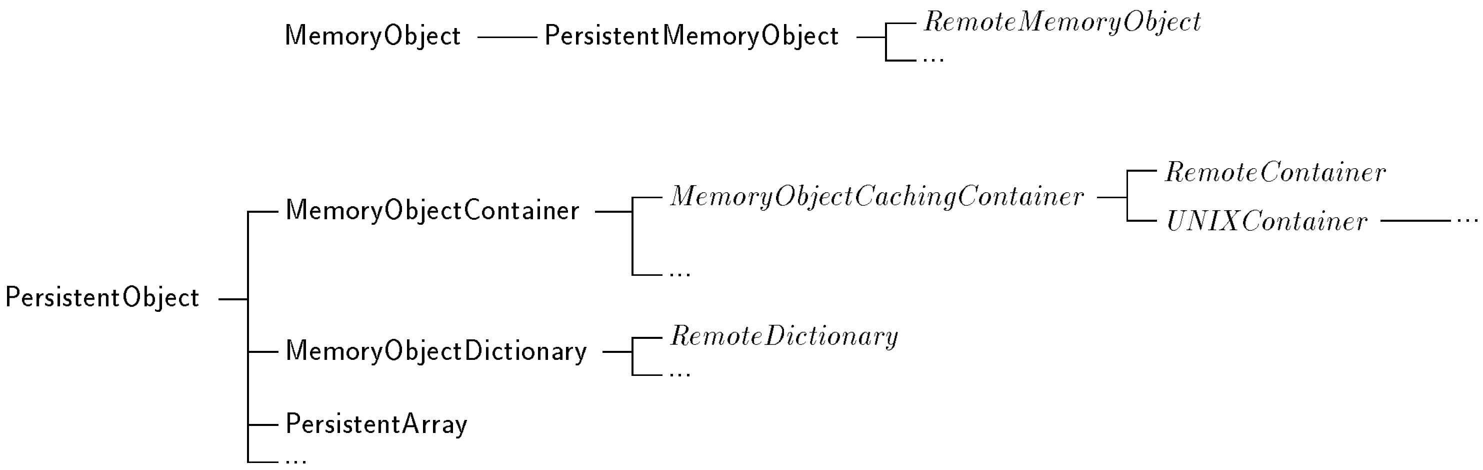

Figure 13: The additions and changes to *Choices* file system class hierarchy. Additions and changes are shown in *italics*.

12 Miscellaneous Useful Information

This section contains information useful in working with *Choices* or the facility at the University of Illinois.

– Warning –

This information will probably be moved when a better place is found. Other sites will wish to amend this section.

12.1 Searching

Choices does not have any real browsing utility. Check the *Choices* shell section for class hierarchy commands.

The following command is arcane, but essential for work on this system. The command recursively searches all files in the subdirectories for occurences of the search string. Note that a space is required after the curly braces.

```
find . -name 'filenamePattern' -exec grep searchString {} \; -print
```

12.2 *Choices* Machines

'ss2t1' through 'ss2t9' are Sparc2 machines.

'ss10t1' and 'ss10t2' are Sparc10 machines. Virtual Choices are not supported.

The following named systems are here (MPx = x multiprocessor system)

1. cassius –
2. clitus –
3. voracius –
4. envius – MP4
5. tedius – MP4
6. studius –
7. devius –
8. delirius –
9. gregarius – MP4

Currently the print function at some of the machines does not work (will enquire further on this situation).