# Guide to the *Choices* Development Environment

Choices Group

January 21, 1992

**Abstract**

This manual is an introduction to the Choices Development Environment. It is designed to help the beginning *Choices* developer become acclimated with the the source, tools, and techniques used to build *Choices* in a fast and efficient manner. It covers the *Choices* source code layout, techniques for building and modifying *Choices* kernels and applications, descriptions of the different *Choices* subsystems, and instructions on how to boot Choices withing certain laboratory environments within the Department of Computer Science at the University of Illinois. As with any manual, understanding all of the sections is not necessary.

## 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 objecs
  - Process, Semaphore, CPU
  - MMU, Disk, etc.

# 2 The Choices Distribution

*Choices* is 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 will be covered in more detail later.

- Applications - The Applications directory contains object oriented application sources for Choices applications. The Examples directory below 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.
- Common - Sources common to the kernel and applications. It 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. It is shown in figure 3
- FileSystems - Filesystem sources
- History - File detailing features and changes
- 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 included in other files containing things such as #define LOCK SS1Lock, allowing 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
- Networks - Networking protocol sources
- ProcessorDependent - Processor specific sources. These codes do things like context switching, CPU and MMU operations, etc.

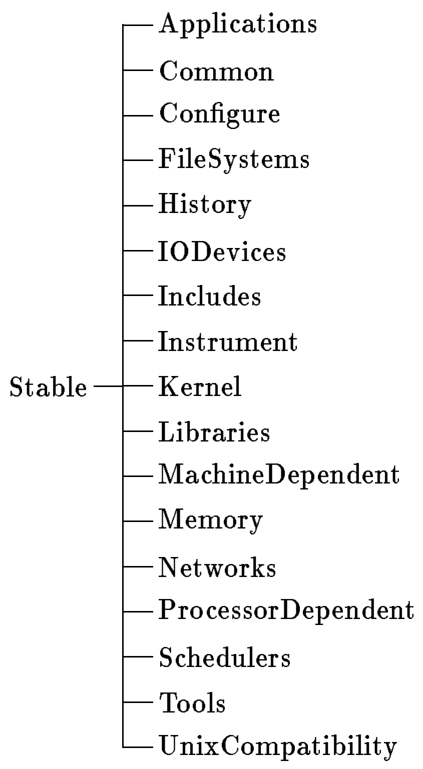

Figure 1: The top level of the *Choices* source tree

- Schedulers - Source for a bunch of schedulers
- Tools - Miscellaneous tools, should contain more
- UnixCompatibility - Source for UNIX compatibility stuff

## 2.2 Creating a Working Environment

The way to work with a Choices stable is to make a shadow if it. The shadow node 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.

The newnode command is used to create a shadow node. It takes a source node and a destination name and asks a few questions before setting up the shadow node.

The breaklink command is used when you wish to modify source files. Given a list of files, it replaces the links with files, copying the original file with a .O file extension. With the -n option the .O file is still a link back to the original instead of a copy. Breaklink -n is recommended.

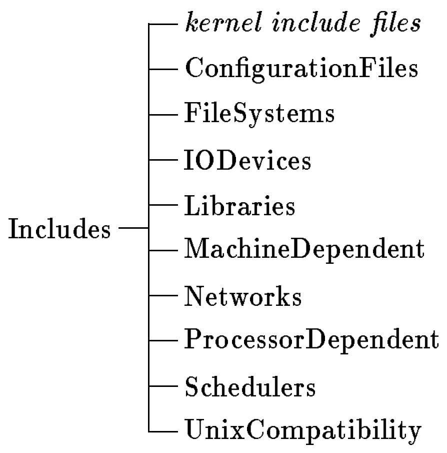

Figure 2: The Includes directory hierarchy

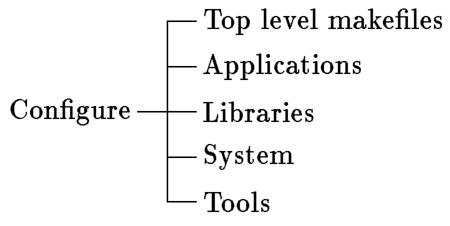

Figure 3: The Includes directory hierarchy

You will need other tools to work with Choices. A C++ compiler, a linker, and assembler for your platform. The Free Software Foundation gnu tools are a good base to get started with.

## 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 *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 make here this makefile will build everything for every platform and it takes a long time to do so.
- ChoicesCommon.mk - Defines some targets common to all parts of Choices, such as clean (make clean removes files built from other files), clobber (make clobber adds removal of dependencies to clean). 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. It contains things that are common to every system. A make 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. Config.mk in this area is used to turn on and off different subsystems and options in the kernel. Things like BSDfile for the BSD filesystem can be defined, causing the BSD filesystem code to be compiled in. Certain options in Config.mk do not make sense for certain platforms.

If we type make from Configure/System/SS1 the makefile includes SS1Common.mk from the top level, 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 in the targets in the Configure/System/¡machine¿/Makefile is all that is necessary. To add a file to all kernels and 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 specific specifications which come from Configure/Libraries/SystemInterface/LoaderOptions.mk.

### 3.2 Building the GeneralPurpose Libraries

The general purpose libraries are required by all other parts of Choices so must be built first when modified, cleaned, or clobbered. The library is built from Configure/Libraries/GeneralPurpose/¡machine¿. Once in that directory type make.

### 3.3 Building the Kernel

The SystemInterface library depndends on the kernel, so the kernel must be built next if a make clean or a make clobber has been done in the Configure/System/¡machine¿ directory has been done or if kernel sources have been modified. A make command in that directory builds the kernel and libSystem.a, the kernel part of the system interface library.

### 3.4 Building the SystemInterface library

The SystemInterface library depends on the kernel, so the kernel must be built first. Once the kernel is up to date a make in Configure/Libraries/SystemInterface/¡machine¿ builds the system-interface library for that machine.

### 3.5 Building Applications

Once you have an up to date general purpose library, kernel, systeminterface library you can build applications. A make in Configure/Applications/¡application¿/¡machine¿ will build an up to date application. To build FiSh (the shell) for a Multimax we execute the commands: cd Configure/Applications/FiSh/Multimax; gnumake

## 4 Using Choices

### 4.1 Using Your Kernel and/or Applications

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 form 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 a Instructional lab environment for these labs at the University of Illinois.

### 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.

### 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
- 555 - Break to debugger
- 666 - Print kernel heap statistics

- 777 - A basic timing benchmark
- 888 - Print Active Objects
- 999 - Synchronize filesystem

## 4.4 *Fish,* the *Choices* shell

The current *Choices* shell is a UNIX-like shell with some object-oriented features. It supports UNIX shell commands like `ls, cd, cp, mv, rm, mount, umount, chmod` for the ordinary file system operations. It also supports other commands that reveal the *Choices* object-oriented design and implementation. In this section I will give a short description of most of these "object-oriented" commands.

### 4.4.1 Mounting file systems

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

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

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

```
FiSh> umount /diskette0
```

### 4.4.2 The `kindred` command

This command lets the user view all the objects of a particular class and of its descendants. The command `members` shows only the members of a class. Two examples follow:

```
FiSh> kindred InputStream
InputDevicesStream[0x263a60]{2}
InputDevicesStream[0x2635a0]{2}(InputStreamOnKeyboard)
InputDevicesStream[0x263960]{3}
InputStream[0x1b9c64]{1}
4 instances.
FiSh> members InputStream
InputStream[0x1b9c64]{1}
1 instances.
FiSh>
```

# 5 Software Development for Choices

## 5.1 Debugging

### 5.1.1 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.

### 5.1.2 Assertions

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.

### 5.1.3 RAID Debugging

Raid statements inside the *Choices* kernel code print information when they are enabled in 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.

### 5.1.4 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:

```
        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 follow.

To turn on debugging for the public methods for all currently instantiated members of class

**Abstract**

Parsley, use the following command:

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

To turn on all debugging for all currently instantiated members of class

**Abstract**

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

**Abstract**

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
```

### 5.1.5 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.

| 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**

### 5.1.6 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 $\hat{D}$ 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)
```

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 the SPARC 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 tell 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 to be debugging.

#### 5.1.7 CDB Debugging

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

# 6 Choices Implementation Overview

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.

## 6.1 Memory

The files in the Includes/Memory and Memory directories implement the memory subsystem. The basic abstractions are:

- Domain - A Virtual memory environment that Processes can run in. The Kernel Domain is contained within all other Domain so it is always active and accessible although it is protected.
- MemoryObject - An object that can be mapped into Domain for direct access via VM addresses. Typical MemoryObjects are for text and data sections of programs, shared memory, sections of the kernel, etc. A MemoryObject can be mapped into multiple Domains multiple times, to pass arguments to a user Process for example.
- MemoryObjectCache - Implements paging of MemoryObjects. It has a policy module and communicates with the Store, which manages the physical pages, called PhysicallyAddressable-Units, in the system.
- AddressTranslation - Implements the data structures that the MMU uses. Each platform defines a subclass of this, or uses a pre-existing one such as TwoLevelPageTable. Each Domain in the system contains an AddressTranslation and manages the mappings for the translation in response to faults, etc.
- AddressTranslator - A MMU implementation. Each machine defines a subclass, SPARCMMU for example.

## 6.2 Processes

In Choices a process is a a simple thread of control running in a Domain. For a multi-threaded application multiple threads are created for the same domain. An example is in the Applications/Examples/twoproc.cc. The classes of Process within Choices that you should be concerned with are:

- SystemProcess
- ApplicationProcess

The class ProcessContainer is a base class for classes which hold Processes. Things such as Schedulers and CPUs are ProcessContainers. When a Process is put into the CPU the CPU is made to run that Process.

## 6.3 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.

### 6.3.1 Memory Locking and Physical Addressing

The class PhysicalMemoryChain is used to lock down memory during I/O operations. A PhysicalMemoryChain is created by calling Domain::constructChain with an address and length. The chain contains structures representing the physical pages it has locked down and paged in (if necessary) from the MemoryObjects in the Domain in that address range. When the block within which the chain is used exists, the destructor releases the locks on the pages. A PhysicalMemoryChain-ContiguousBlockIterator is used to iterate over contiguous ranges of physical memory in the chain for efficient I/O.

### 6.3.2 DMA/DVMA and Memory

In order to support both DMA and DVMA the Kernel initialization maps all physical memory in the machine 1-1 to the same virtual address in the Kernel Domain. This enables the address of a PhysicallyAddressableUnit (page) to be used by either type of system with the same code.

### 6.3.3 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 the 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.

### 6.3.4 A typical driver sequence

- Lock down memory by creating a chain.
- On DVMA systems mirror the VM mappings for the virtual address in the Kernel so they are always accessible if another Domain is active when an interrupt occurs.
- Set up the I/O and P() a semaphore.
- Wait for interrupts to drive the I/O. Use the Iterator to traverse the chunks of physical memory if necessary.
- When the I/O is complete
  - On DVMA systems remove the kernel mirror mappings.
  - V() the semaphore, letting the block containing the chain exit causing the chain destructor to unlock the memory.

## 6.4 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 show 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)
```

## 6.5 First Class Classes

Choices allow access to the Class hierarchy at run time much like Smalltalk allows. 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.

## 6.6 The SystemInterface

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.

### 6.6.1 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 an 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 an class derived from the ProxiableObject class can have the proxiable keyword prepended to it allowing that method to be called on the object.

### 6.6.2 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;
```

## 6.7 System Boot and Initialization Sequence

This section describes how Choices boots and initializes itself.

### 6.7.1 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();

### 6.7.2 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 the 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), find 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.

### 6.7.3 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
|               |
|               |
 --------------- 0xffffffff
```