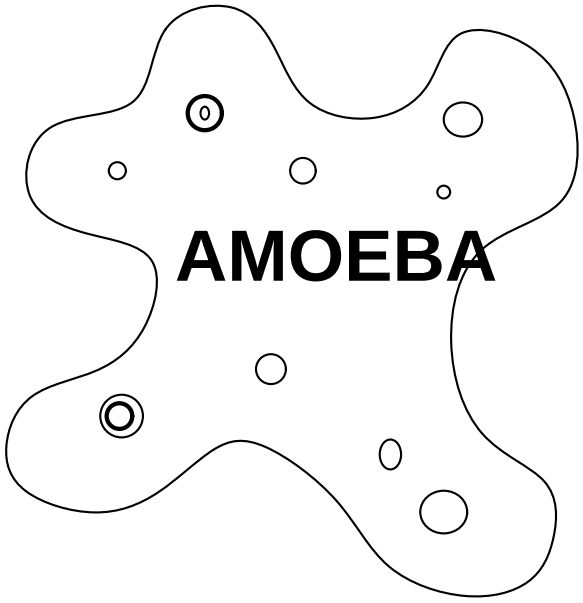
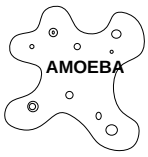
The Amoeba Reference Manual

System Administration Guide



Amoeba is a registered trademark of the Vrije Universiteit in some countries.



is a trademark of the Vrije Universiteit.

Sun-3, Sun-4, NFS, SPARCclassic, SPARCstation, MicroSPARC, SunOS and Solaris® are trademarks of Sun Microsystems, Inc.

SPARC is a registered trademark of SPARC International, Inc.

UNIX is a trademark of Unix International.

Irix and Personal Iris are trademarks of Silicon Graphics Corporation.

DEC, VAX, MicroVAX, Ultrix and DEQNA are registered trademarks of Digital Equipment Corporation.

Intel 386, Pentium, Multibus and Intel are trademarks of Intel Corporation.

Ethernet is a registered trademark of Xerox Corporation.

IBM and PC AT are registered trademarks of International Business Machines Corporation.

X Window System is a trademark of the Massachusetts Institute of Technology.

WD, EtherCard PLUS and EtherCard PLUS16 are trademarks of Standard Microsystems Corporation.

Logitech is a trademark of Logitech, Inc.

Novell is a trademark of Novell, Inc.

3Com and Etherlink II are registered trademarks of 3Com Corporation.

MS-DOS is a trademark of Microsoft Corporation.

Amoeba includes software developed by the University of California, Berkeley and its contributors.

Contents

1 Using This Manual

Intended Audience

This manual is intended for use by system administrators.

Scope

It is assumed that the reader is familiar with the introductory material in the User Guide. This manual explains how to install, configure and maintain an Amoeba system. Security, hardware, resource management and user management are all described.

Advice on Using this Manual

The manual is divided into several major sections. The first chapter describes how to install Amoeba or upgrade an existing Amoeba system to a new version. Check the release notes to see if the latter is possible. The introductory material explains basic things such as the supported hardware and how to build configurations. This material is necessary for understanding the installation instructions.

Next come the installation instructions for the various supported architectures. It is vital that you read the introductory material and the installation guide completely before beginning so that you have an overview of the entire process and do not get any surprises on the way.

The sections on adding new hosts and new users are largely self-contained but you should read the manual pages of the utilities mentioned before using them since they offer more detail about the various options.

The section on porting Amoeba is for the very brave. The target machine may be the subject of a port by somebody else so ask around before beginning. It can save a lot of effort.

There is a short chapter on *Routine Maintenance* and another on *Trouble-shooting* which identifies some common problems during installation and day to day runnning and how to solve them.

The manual pages for utility programs are before the index. They are intended as a reference work for system administrators and should be thoroughly read to gain a good overview of the available possibilities.

Throughout the manual there are references of the form *prv*(L), *ack*(U) and *bullet*(A). These refer to manual pages at the back of the various guides. The following table explains what each classification refers to and in which guide the manual page can be found.

Letter	Type	Location
(A)	Administrative Program	System Administration Guide
(H)	Include file	Programming Guide
(L)	Library routine	Programming Guide
(T)	Test Program	System Administration Guide
(U)	User Utility	User Guide

There is a full index at the end of the manual providing a permuted cross-reference to all the material. It gives references to both the manual pages and the introductory material which should direct you to enough information to clarify any difficulties in understanding.

2 Installing Amoeba

2.1. Introduction

The Amoeba software consists of four major parts:

- The Amoeba kernel.
- The servers and utility programs for native Amoeba.
- The Amoeba FLIP driver for various UNIX kernels.
- The servers and utilities that run under UNIX and use Amoeba RPC.

In addition to these there is documentation and a suite of test software. The distribution comes with binaries for various Amoeba configurations and the SunOS version of UNIX.

Amoeba runs on various hardware platforms which are described in section 2.2.

It is often possible to upgrade an existing Amoeba system to a new version. Section 2.3 describes how to do this. Check the release notes to see if it is possible to upgrade your current version to the new release. If this is not possible or if installing Amoeba for the first time then follow the instructions in section 2.4.

Section 2.4 and following describes how to install Amoeba directly from the distribution medium onto the computers that will run it. The installation of the UNIX-based software, including the Amoeba network protocol in the UNIX kernel and the UNIX-based Amoeba utilities is described in the chapter *Amoeba and UNIX*.

Installing Amoeba is not a trivial task and should be assigned to someone with some experience of system management. An understanding of how computer networks function is a great asset to understanding the process of installation. It is important to understand network structuring when configuring a system. For example, if possible, Amoeba hosts should be separated by a bridge from other systems on the same network or put on a separate network behind a gateway. This is due to the ability of Amoeba to consume the entire network bandwidth.

There are some important concepts which must be understood before beginning the installation process. The most important is that Amoeba is a *distributed* system. Furthermore, the name server and the file server are two separate servers. The domain of an Amoeba system is effectively defined by the name server for that system, since it defines the name space for Amoeba. Therefore normally only one name server should be installed. The installation process automatically installs one file server and one name server, so it should only be run once for a local area network, unless more than one name space is explicitly desired. If additional file servers are desired they should be installed later using *disklabel*(A), *mkfs*(A), *mkbootpartn*(A), etc.

In general it is a bad idea to put workstations or file servers in the processor pool. This will degrade their performance and therefore the performance of the entire system. If possible, try to ensure that there are enough hosts running Amoeba to leave workstations and file servers free to do their specialized tasks. Note that it is essential that you have some hosts in your processor pool. If sufficient processors are available it may also be interesting to have more than one processor pool, and so divide the available CPU power between various groups of users. It is possible to have processors of more than one architecture in a pool (so called *heterogeneous* pools). See *run*(A) for details on how to set this up.

The rest of this chapter is structured as follows. There is a brief description of the currently supported hardware configurations. This is followed by details of how to load a native Amoeba system directly from the tape, setting the time of day clock and the timezone, how to install the source code, including a description of the source directory structure, and finally where to install third party software, such as X windows. The chapter, *Configuring an Amoeba System*, describes how to configure the various servers to suit the available hardware and needs of the user community. It also describes how to add new hosts to the system and how to register new users.

2.2. Supported Hardware Configurations

2.2.1. General Requirements

The Amoeba distributed operating system can run on many different types and brands of computers. It is intended that it should run on a network with **at least** five computers. Although it is technically possible to run Amoeba with just a single processor, it is not a satisfactory way to use the system. It is, after all, a *distributed* operating system. To get good results from the system you should have a separate machine for the file server, one workstation per user (to run the user interface) and a group of pool processors to perform the actual work and to run various servers. In general, at least five hosts are needed for pleasurable computing. All the Amoeba hosts must be connected by a network so that they can communicate with each other, although they need not necessarily be all on the same network, as long as all the networks are connected via gateways.

Networking

Currently Amoeba only has drivers for Ethernet but other network drivers are under consideration. Amoeba sets no upper limit on the number of computers on a local network but in some cases it is limited by the network bandwidth or topology. (More commonly, it is limited by the hardware budget.) **Note well**: Amoeba does not demand exclusive use of the network. It can use the same network as computers running other operating systems. However it is in the nature of distributed systems to consume more network bandwidth than centralized systems. It is therefore a good idea to isolate the Amoeba hosts behind a bridge if the other hosts on the network are performing important functions which require good network response.

Heterogeneity

Amoeba is reasonably portable and has run on five architectures. However it is currently only available on the following architectures: the Motorola MC680x0 family, the Intel 80386 family and certain members of the SPARC family.

It should be noted that some of these architectures are big-endian and others little-endian. A mixture of different endian machines should not be a problem as standard servers and utilities deal with this. Combinations of different architectures only requires that the binaries are available for all the architectures. Processor pools may contain processors of more than one architecture. The run server (see *run*(A)) chooses the optimal processor for the architectures for which the binaries are available.

There are several configurations supported for the three architectures. This is to provide the various types of processor required: pool, workstation and specialized server.

File Servers

The file server distributed with Amoeba is called the Bullet Server. It uses a large buffer cache to provide very high performance. Thus the file server should be run on a machine with a large amount of RAM, and, of course, a disk.

For the file, server the more memory present, the better the performance. However on the Sun 3/60, Amoeba can use a maximum of 12 MB due to restrictions in the memory

management code. The following table shows for each architecture the *minimum* amount of memory needed for a file server host. It also shows the largest main memory support by the kernel. In most cases this will not be supported by the physical machine.

Architecture	Minimum Memory Requirement	Software Limit on Memory
MC680x0	12 MB	12 MB
i80x86	16 MB	64 MB
SPARC	16 MB	512 MB

Fig. 2.1. Minimum memory requirements for file server machines and maximum memory configurations supported by Amoeba.

On the various 80x86 machines it may be possible to insert up to 128 MB †, depending on the machine. In older SPARCstations it should be possible to insert up to 64 MB and in SPARCclassics up to 96 MB. See the hardware documentation for the physial limits of your hardware.

Workstations

The primary function of workstations is to run the user interface (for example the X window system) and so they do not require large amounts of memory. 8 MB is enough for color X servers and 4 MB for monochrome X servers. It is also possible to use an X terminal as a workstation. (If the X terminal does not understand the Amoeba network protocol it will be necessary to use a TCP/IP server to convert network protocols which reduces perceived performance.)

If no bitmap display is available a regular terminal can be used. It will not run X windows, of course.

Pool Processors

Nearly all the computing occurs on the pool processors. There is no virtual memory in Amoeba so it is important that pool processors have enough memory to run the processes entirely in core. Between 3 and 4 MB of memory is the minimum for normal use. If large programs such as matrix multiplication or TEX are to be run then more memory will be required. Each processor pool should have at least 4 or 5 processors to do any useful work.

---

† The current BIOS call to obtain the memory size only returns 16 bits of information – enough to represent 64 MB. The rest of the kernel can cope with much more. This limit will be fixed when the new BIOS call for larger memories becomes known to the Amoeba developers.

2.2.2. Supported Hardware

Below is a summary of the machines and related equipment for which Amoeba is currently available.

SPARC Systems

Amoeba runs on the Sun4c and the Sun4m MicroSPARC families of processors. It has been tested on the following Sun4c machines: SPARCstation 1, 1+, 2, SLC, IPC and ELC. It will probably also work on the IPX. A should work on the following MicroSPARC I & II machines: the SPARCclassic, SPARCstation LX, SPARCstation 4 and SPARCstation 5.

The audio hardware is not supported. Nor is the ‘‘le/buffer’’ Ethernet/SCSI SBus card. All the other standard devices are supported. Of the SBus graphics cards, the BW2 and CG3 are supported. The CGX will work but the graphics accelerator is not used (since there is no documentation available on how to drive it).

Intel 80386 Systems

Amoeba runs on Intel 80386, 80486 and Pentium-based machines with an ISA bus (i.e., an IBM PC AT bus). It has also successfully run on PCI-bus and VL-bus machines. However, it does not take advantage of ‘‘plug-and-play’’ facilities. The following devices are supported.

Disk controllers:

SCSI, ESDI and EIDE. If the controller contains intelligence / caching, it may be necessary to turn the intelligence off to get it to work with Amoeba. The supported SCSI cards are the Adaptec 154X cards. These include the Adaptec 1540, 1542A, 1542B 1542C and 1542CF cards. SCSI tapes are also supported.

Graphics cards:

CGA, Hercules, EGA, and VGA. (See notes on X windows below.)

Floppy Drives:

Floppy drives of 1.44 MB and 1.2 MB are supported, and the driver allows floppy disks with the following densities to be used: 360 KB, 720 KB, 1.2 MB, and 1.44 MB. 2.88 MB floppy drives may work; they are treated as 1.44 MB floppy drives.

Network cards:

Novell NE1000, NE2000, NE2100
WD 8003E, WD 8013EB, and WD 8013EP.
SMC 8013EPC.
3Com 503 Etherlink II.

Note: The Ethernet cards normally use IRQ 3 and IRQ 5 in Amoeba. If that is inconvenient then alternate IRQ levels can be used. Below is a table showing the default IRQ allocations. To use different IRQs for Ethernet it will be necessary to alter the file *etherconf.c* in the kernel templates and recompile the kernel (see the section on building kernels for more details).

interrupt vector number	bus pin	devices using interrupt
0	-	clock
1	-	keyboard
2	-	expansion PIC (see IVN 9)
3	IRQ3	serial port 2
4	IRQ4	serial port 1
5	IRQ5	not used
6	IRQ6	floppy controller
7	IRQ7	parallel port
8	IRQ8	real time clock
9	IRQ2	not used (wired to IVN 2)
10	IRQ10	not used
11	IRQ11	Adaptec SCSI controller (optional)
12	IRQ12	not used
13	IRQ13	iAPX387 math co-processor
14	IRQ14	hard disk controller
15	IRQ15	not used

The Xfree version of X windows is provided. It runs on most commercially available video cards. A VGA controller is recommended. One of the following mice is also required for X windows: Logitech Bus mouse (at IRQ level 5), Mouse System's mouse or Microsoft mouse (serial). Note that the color X server needs 8 MB to run effectively.

MC680x0 Systems

Amoeba kernels exist for the Sun 3/60, Sun 3/50 and various MC68020 and MC68030 CPU boards. In particular the Force30 board is supported.

Sun 3/50 and Sun 3/60

The following devices are supported:

- Lance-based Ethernet interface
- The monochrome bitmap display
- The uart/timer devices
- The SCSI interface to winchester disks and cartridge tape drives

Amoeba can currently only take advantage of up to 12 MB of memory in a Sun 3/60.

2.3. Upgrading an Amoeba Distribution

For those sites that already have an Amoeba system running it may be possible to upgrade to a new release without running the entire installation process. Check the release notes to see if a simple upgrade from your current version to the new version is possible. If in doubt contact your Amoeba distributor.

All that needs to be updated during most upgrades is the system binaries, sources and kernels. Occasionally it may be necessary to install a new directory server or boot server but this can usually be done without difficulty. It is vital that there is no damage to the directory */super/admin/bootinfo* under Amoeba since it contains details of how the system should be started and maintained. The same is true for the directories */super/hosts*, */super/pool* and */super/users*.

NB. When upgrading, unless otherwise specified in the release notes, perform the upgrade in the order of the following sections. Otherwise things may go wrong and it may even become necessary to install from the coldstart. Also note that if a new directory server and boot server must be installed the system should not be rebooted until after that has taken place.

2.3.1. Installing the Binaries

If you have a FLIP driver installed in a UNIX system from which you can copy files to Amoeba then the upgrade process is relatively simple. First make sure that nobody is logged in to Amoeba since the new binaries may not be compatible with the old and may therefore cause difficulties for users.

Copy the tar file for your architecture (eg., sun4/tapefile.2 or ibm_at/BinTree.am). Untar the ibm_at version on UNIX under the directory *`amdir`/BinTree.am*. Untar the Sun version on UNIX under the directory *`amdir`/BinTree.am/super*.

If X windows and/or MMDF are to be installed then untar the X and MMDF tar files in *`amdir`/BinTree.am/super/module*.

To install the system binaries and manuals the following commands under UNIX will copy them from the *BinTree.am* tree to Amoeba.

```
cd `amdir`/BinTree.am/super
W=/super
tar cf - admin/bin util unixroot/bin \
        module/x11 module/amoeba/lib | \
        ax -E _WORK=$W -C WORK=$W /bin/tar xvf -
```

MMDF is a little more difficult since it mixes site dependent files with binaries. Preserve the mmdftailor file and any local scripts and lists and then use a similar command to the one above to install the MMDF binaries.

2.3.2. Installing the Source Code

Once all the binaries have been installed the sources can be installed from UNIX to Amoeba. Note that the current source code is read-only and that various directories and files may have been deleted in the new sources. Therefore it is a good idea to delete or move the existing source tree. The old sources can be deleted using:

```
del -d /super/module/amoeba/doc
del -d /super/module/amoeba/src
del -d /super/module/amoeba/templates
```

or if they are to be saved by:

```
mv /super/module/amoeba/doc /super/module/amoeba/doc.old
mv /super/module/amoeba/src /super/module/amoeba/src.old
mv /super/module/amoeba/templates /super/module/amoeba/templates.old
```

The new source tree can then be installed using:

```
W=/super/module/amoeba
zcat SRC.tar.Z | ax -E _WORK=$W -C WORK=$W /bin/tar xvf -
```

2.3.3. Installing the Kernels

First transfer the kernels from UNIX to Amoeba:

```
cd `amdir`/BinTree.am/super/admin
W=/super/admin
tar cf - kernel | ax -E _WORK=$W -C WORK=$W /bin/tar xvf -
```

Next, login to Amoeba as someone with the ‘‘super’’ capability, such as *Daemon*. If any hosts are booted using TFTP then go to the directory */super/admin/tftpboot* and update the relevant TFTP links. This is best done with *getdelput*(U). For example:

```
cd /super/admin/tftpboot
getdelput ../kernel/sun4.workst 84563123.SUN4C
```

The next step is to update the kernel booted from disk. This should be done using *mkbootpartn*(A). Note that the Sun kernels in */super/admin/kernel* are already stripped and converted to the form in which they boot. (This was done using *installk*(A)). The i80386 kernels need to be converted to a different form before being suitable for booting via TFTP. See *tftp*(A) and *isamkimage*(A) for details.

If the name server and or boot server need to be upgraded then do **not** reboot the file server or the hosts where the boot server and soap server run. (Usually they all run on the same host.) Otherwise it should be safe to reboot the system now. If for some reason the bootstrap of the file server fails then make a tape or floppy under UNIX from which to boot. This can be used to restart the system and correct the problem(s).

2.3.4. Installing the New Soap and Boot Servers

It is not always necessary to replace the *soap*(A) server or *boot*(A) server binary. If it is not done correctly it can have potentially catastrophic effects so only do this when necessary.

When installing the new servers do not delete the old servers until the new ones are installed and running. Make a copy of the current Bootfile and save it on a UNIX hosts so that it can be restored if anything goes wrong.

In the following substitute for `<arch>` the architecture (for example, mc68000 or i80386) of

the host where the soap and boot servers will run. Also, substitute for `<host>` the host where the boot server is running.

cd /super/admin/bootinfo
cp /super/admin/bin/boot/pd.<arch> boot
cp /super/admin/bin/soap/pd.<arch> soap
c2a soap

edit the bootfile and replace the capability on the

cap SOAPBINARY = ... ;

line with the capability produced by the command `c2a soap`. It is very important not to leave out the semicolon on the end of the line. Once this has been done then the bootfile should be installed.

iboot -b boot -f bootfile /super/hosts/<host>/vdisk:02
boot_reinit /super/cap/bootsvr/<host>

should report success. If not then correct the bootfile and run *iboot* and *boot_reinit* again. Once *boot_reinit* completes successfully reboot the file server and then the installation is complete. **N.B.** Do not stop or reboot the host where the boot server or soap server is running until the *boot_reinit* has succeeded.

2.4. The Binary Distribution

Typically Amoeba is installed directly from tape or floppy onto a computer with a disk. The distribution medium for Amoeba is dependent upon the hardware on which it is to be installed. The medium may be an Exabyte tape, QIC-150 tapes, QIC-24 tapes or even floppies.

The distribution can be seen as consisting of three parts.

1. The kernels and binaries for installing directly onto disk.
2. The source code.
3. The UNIX binaries and shell scripts.

The source code installation is described in section 2.6. The installation of the sources and UNIX binaries under UNIX is described in the chapter *Amoeba and UNIX* later in this manual.

The installation of the binary distribution and source code depends to some extent on the distribution medium. The Sun distribution is on cartridge tape. The Intel 80386 distribution is on 5 floppy disks plus a tape loaded over the network from a UNIX system, or via a SCSI tape unit on the i80386 system. Separate installation instructions follow for the various machines. Choose the installation instructions that correspond to the machine type upon which you are installing Amoeba.

Before proceeding further, a brief explanation of the *date*(U) command is given since it will be necessary to set the date during the course of the installation. The following will set the default *time of day* server (the one whose capability is in */super/cap/todsvr/default*) which will be on the machine on which you are installing Amoeba. It is set using the *date*(U) command which has the form:

date `yyyymmddhhmm[.ss]`

The square brackets mean that the seconds are optional. For example, to set the time to 10 November 1992, 2:45 pm issue the command:

date 199211101445

To specify the seconds, the following would set the time to 13 January 1994, 10:45 and 32 seconds:

date 199401131045.32

2.4.1. Installing Amoeba on Intel 80386 Systems

The Amoeba version for the i80386 ISA bus systems will also run on i80486 and Pentium ISA, PCI and VL-bus systems. It uses the standard partition table format which is common among these types of machines. This allows one to have partitions containing other operating systems (such as MS-DOS). If partitions for other operating systems are desired, install these before installing Amoeba. Remember to leave an empty partition of at least 70 MB to hold Amoeba. 300 MB or more is recommended for a system with many users or if X windows is to be installed. Once this is done the following describes how to install Amoeba.

The binary distribution for Amoeba on the 386 consists of five floppy disks containing the binaries for a tiny Amoeba system, and a large tar file containing the remaining images.

The floppy disk labeled *KERNELS* contains a number of bootable kernels, among others the installation kernel. The floppy disks labeled *RAMDISK-IMAGE* contain a ramdisk image that will be used by the installation kernel. The remaining floppies, labeled *AMOEBA*, hold a *starch*(U) image containing all the system binaries to be installed.

To use these floppy disks the following hardware is required:

- An i80386 or i80486 computer with an ISA (AT) bus architecture. The computer should be equipped with at least 16 MB of memory. The more the better!
- A 1.44 MB floppy disk drive.
- A hard disk of at least 70 MB.[1]
- One (or more) of the following Ethernet boards:
  - A Novell NE2100 or compatible.[2] Note that it is not possible to put more than one NE2100 board in a machine. If a gateway is to be built, one of the following boards can be used as the second board. They can also be used as the primary board but have poorer performance than the NE2100 board.
  - A WD 8003E, WD 8013EB, WD 8013 Elite, WD 8013EP or SMC 8013EPC.[3] The SMC 8013EPC will give the best performance of the SMC (formerly Western Digital) Ethernet boards.
  - A Novell NE1000 or NE2000 or compatible.[4] The NE2000 gives much better performance than the NE1000.
  - A 3Com 503 Etherlink II or compatible.[5] The performance is poorest of the cards listed. 16-bit 3Com cards may work but have not been tested.

In the instructions below the **bold** font will be used for what you have to type at the

———————

1. Note that if using an Adaptec SCSI controller it should be configured with I/O address 0x330 and IRQ 11.
2. The Novell NE2100 should be configured with I/O address 0x300 and IRQ level 3.
3. The SMC/Western Digital Ethernet boards should be configured with I/O address 0x280, IRQ level 3, and memory base address 0xD0000. If a second Ethernet board is added it should be at I/O address 0x300, IRQ level 5 , and memory base address 0xD4000.
4. The NE1000 or NE2000 should be configured with I/O address 0x280 and IRQ level 3 or I/O address 0x300 and IRQ level 5.
5. The 3Com 503 should be configured at I/O address 0x280 and IRQ level 3. The share memory should be enabled at any free address (e.g., 0xCC000).

computer, and the `standard font` for what the computer types back at you. A ⇒ marks the beginning of each step.

Preliminaries

⇒ Before starting with the installation procedures write down the Ethernet address of your computer. This address is used later on in the installation phase. If the address cannot be obtained easily, boot a workstation kernel from the *KERNELS* floppy and write down the Ethernet address which is printed as part of the standard banner. How to bootstrap a workstation kernel is explained below.

⇒ Choose a name for the new Amoeba host. Host names should be chosen with care. The host name must be a single word containing no spaces or tabs. Ideally, it should consist only of letters and digits. Other characters may be interpreted by the shell when typing the host name.

The Installation

⇒ Power on the machine and boot from the floppy labeled *KERNELS*. The following message will appear:

```
Amoeba 5.3 Standalone Boot Program

Default kernel: coldstart (just press <ENTER> or wait ~30 seconds)

boot:
```

The *KERNELS* floppy contains a number of Amoeba kernels. To get an overview type a single question mark ('?'). This will display a list of available kernels, the values between the parenthesis denote the offset and the size (in blocks) of the kernel.

```
boot: ?
Bootable kernel(s):
    coldstart (1:499)
    smbul (500:900)
    pool (1400:700)
    workstation (2100:700)
```

To start any kernel, just type its name followed by *return* (sometimes known also as *enter*). An exception to this mechanism is the default kernel which is booted if no name is given (just a return), or when there has been no keyboard activity for approximately thirty seconds.

⇒ To setup an Amoeba system, boot the kernel named *coldstart*. This is the default kernel so simply press return. Booting this kernel causes the system to display some descriptive messages (version, creator, creation data, some device driver information).

⇒ After a few moments a request for a floppy will appear. Remove the *KERNELS* floppy, insert the *RAMDISK-IMAGE-0* floppy and press return. When it prompts for the next floppy remove the *RAMDISK-IMAGE-0* floppy, insert the *RAMDISK-IMAGE-1* floppy and press return.

Once both floppy disks have been loaded, the *coldstart* kernel will start the necessary servers

for the installation. While this is going on, various messages that look like errors might be displayed. If at this time or any other time the message:

```
WARNING: winchester multi sector I/O functionality turned off
```

is displayed, do not be alarmed. It is simply that the hard disk will perform a little more slowly because it was not able to handle the maximum performance demanded by the disk software. Please report this to your Amoeba distributor if it occurs during normal system operation. Do not worry about other messages. They are either initialization information or from servers that do not understand a system that is only partially running.

⇒ Eventually, after some minutes, a shell prompt will appear:

```
Welcome to standalone Amoeba
#
```

Remove the *RAMDISK-IMAGE-1* floppy at this point.

Verify that everything works by listing the contents of the machine directory. This will provide a list of all devices that are available during the installation phase.

```
# dir –l /super/hosts/coldstart
bootp:00                        @  1289736 KB
bootp:01                        @     2439 KB
floppy:00                       @     1440 KB
proc                            process/segment server
ps                              %--1234
random                          random number server
tod                             TOD server
tty:00                          +
vdisk:80                        @     2288 KB
vdisk:81                        @       99 KB
vdisk:82                        @       49 KB
```

The entry marked `bootp:00` is the name of the hard disk attached to your machine.

At this point the disk must be partitioned and labeled. As already mentioned, the Amoeba version for the AT/386 uses the standard partition table format which is common among these types of machines.

The partitioning procedure is divided into two steps. The first step consists of creating a partition table entry for Amoeba. The partition should be large enough to hold an Amoeba system (at least 70 MB), and should be made active. Partitions are created using *fdisk*(A). *Fdisk* is menu driven and allows you create and delete partitions, activate partition table entries and scan the disk for bad blocks.

⇒ To start *fdisk* type:

```
# fdisk /super/hosts/coldstart/bootp:00
```

See the *fdisk* manual page for details on how to partition the disk.

Note: if you have a hard disk with more than 63 sectors/track, or more than 63 heads, or more than 2048 cylinders it important to place the bootable kernel vdisk within the first 2048 cylinders of the disk. This is necessary since the BIOS can

only boot systems from the first 2048 cylinders.

Having partitioned the disk, the second step is to sub-partition the Amoeba partition. For bootstrap purposes the following four virtual disks are required. They should each consist of one sub-partition.

vdisk:01 This is used for booting from disk and should have room to contain one or more kernels. In almost all normal circumstances 2 MB is recommended. Minimum 1 MB.

vdisk:02 This is used by the *boot*(A) server, and 100 KB is sufficient.

vdisk:03 This is for the administration of the directory server. A 1 MB partition will hold over 30000 directories and should be sufficient for most purposes.

vdisk:04 This will hold the files for the first Bullet Server started. It should be at least 65 MB. If the machine has only 16 MB of main memory then make this partition at most 900 MB. Larger partitions may work but if it is much larger than this there may be insufficient core to cache the inode table when a full Bullet Server is started later. If more main memory is present then the size of the partition can be proportionally larger.

If any disk is left after that, put it in vdisk:05.

⇒ Having planned the sizes for the partitions, call the disklabel program:

disklabel coldstart

This is an interactive program and is described in *disklabel*(A).

NB. When labeling it is important to realize that the disk *bootp:01* is **not** a physical disk and should not be selected for labeling!

When creating the Amoeba sub-partitioning the ‘a’ partition corresponds to vdisk:01, ‘b’ to vdisk:02, etc.

⇒ After the disk is labeled reset the machine and boot the *coldstart* kernel from floppy again. To do this, insert the *Amoeba kernels* floppy in the floppy drive and press the keys **<CRTL> <ALT> <DEL>** simultaneously. The process of loading the ramdisk images must be repeated now.

When the machine is up and running verify that the virtual disks were indeed created, by listing the kernel directory again with *dir*(U).

⇒ Now check the time and date by typing the command:

date

If it is not correct, set the time and date in the time of day server using the command:

date –s /super/hosts/coldstart/tod `yyyymmddhhmm[.ss]`

where *yyyy* is the year, *mm* the month, etc.

⇒ Start the rest of the installation by:

. installit

This script will prompt you for the name of the machine and for its Ethernet address, e.g., **0:0:c0:0:12:34**. It is important that you choose the name for the

machine carefully. In general, Amoeba systems comprise many hosts so use common sense when selecting host names. In particular make sure that the name is one word, since it will be entered in the directory */super/hosts*.

Note that if there is more than one Ethernet interface then give the address of the connection to the Amoeba network.

⇒ After some descriptive messages of programs and services that are started by the installation script, *starch*(U) will ask you to insert the floppy disks labeled *AMOEBA*. It will be necessary to change floppies after a few minutes. They are numbered in the order in which they should be inserted, beginning with zero.

⇒ After the script finishes it will print

```
Installation complete
```

Now you can boot Amoeba from the hard disk. To do this remove any floppy in the floppy drive and press **<CRTL> <ALT> <DEL>** simultaneously. The boot program on hard disk will work exactly like the boot program on the *KERNELS* floppy. Note that the *installit* script installed one kernel. Start this kernel.

⇒ After a minute or two an Amoeba login message will appear. (Be careful, it might be lost in the verbose boot server output.) Log in as user *Daemon*. There is no password, so at the `Password:` prompt just press the return key.

⇒ The login name *Daemon* has the super capability giving access to the entire directory graph so it is important to protect this login against misuse. Once logged in, set the password for the login name *Daemon* using the command *chpw*(U).

It is still necessary to install the bulk of the utilities for Amoeba. However, before doing that, it may be necessary to install the FLIP driver in a UNIX kernel. This will be dealt with first and then the installation of the remaining utilities under Amoeba will be described.

⇒ Install the UNIX FLIP driver as described in the chapter entitled, ‘‘Amoeba and UNIX’’. When convinced that the FLIP driver works correctly, Daemon’s Amoeba capability under must be installed on UNIX so that the binaries can be downloaded to Amoeba from UNIX. The following command executed under Amoeba obtains Daemon’s capability:

```
amoeba% c2a /
<ASCII representation of capability>
```

This information should be kept secret. Do not let others see it. To install this capability under UNIX, then do the following:

```
unix% a2c ´<ASCII representation of capability>´ – > $HOME/.capability
unix% chmod 400 $HOME/.capability
```

Test this by running the *dir*(U) command under UNIX. The *dir*(U) command gives an overview of *Daemon*’s root directory.

⇒ The remaining Amoeba binaries should now be installed. These are in the tar file *BinTree.tar* that came with the distribution. If you have a SCSI-based disk system and a tape unit, the rest of the binaries can be loaded from the tape unit. Simply put the *BinTree.tar* file on a tape (if it is not already) and load it using the following command:

cd /
tar -kxf /super/hosts/*host*/**tape:00**

where *host* is the name of the host given to *installit*. There will be various error messages about its refusal to overwrite extant files. Ignore these. Once the tape is loaded proceed to the section on setting the time.

If you do not have a tape drive then the rest of the binaries can be loaded over the network from UNIX. First untar the *BinTree.tar* file under UNIX. The following commands run under UNIX will then install the binaries. Note that it may easily take up to an hour, depending on performance of the computers and the network.

```
unix% SPMASK=0xff:0x2:0x4 export SPMASK
unix% amdumptree `pwd`/BinTree.am /
```

Now go on to the section on setting the time.

2.4.2. Installing Amoeba on the SPARCstation

The distribution tape for Amoeba for the SPARCstation contains three tape files: a bootable kernel containing a ramdisk driver, the contents for the ramdisk, and a *tar*(U) image containing all the system binaries.

To use this tape the following hardware is required:

- A SPARCstation with at least 16 MB memory.
- A Sun keyboard and monitor, or a 9600 baud terminal connected to ttya.
- A QIC-150 or QIC-24 tape drive.
- A SCSI disk of at least 100 MB. 300 MB or more is recommended for a system with many users or if X windows is to be installed.

In the instructions below the **bold** font will be used for what you have to type at the computer, and the `standard font` for what the computer types back at you. A ⇒ marks the beginning of each step.

⇒ Power on the machine and write down the Ethernet address it displays in the banner. You will need this later.

⇒ If the machine is a sun4c and is already powered on type the command **reset** to the PROM's `ok` prompt. This will cause the banner to be redisplayed and put the machine into a known state. N.B. It is important to reset a sun4c with SCSI devices before booting an Amoeba kernel.

⇒ Make sure that the boot tape is read-only. Insert the distribution tape into the tape unit and boot from tape by typing

```
ok boot tape
```

Note that for this to work, the tape drive must be the default unit. If not, consult the Sun Openprom documentation on how to boot from the correct tape unit.

The tape should spin, the kernel should boot and start to initialize itself, loading the ramdisk from tape, and starting various servers. While all this is going on various messages will be displayed. They come from the boot server. Do not worry about them. After some minutes a shell prompt should appear:

```
Welcome to standalone Amoeba
#
```

You can verify that things are working by typing

```
# dir –l /super/hosts/coldstart
```

which will give a directory listing of all devices present on your machine of which one will be the disk:

```
bootp:00                              @     170498 KB
```

The '@' is the symbol for a disk and the number after it is the size of the disk in kilobytes. In this example it is a 170 MB disk.

At this point you need to partition and label your disk. The Amoeba disk driver code recognizes Sun labels and partition tables and makes it possible to use part of the disk for Amoeba and part of it for UNIX. However this is not supported for booting.

It is recommended to put a Sun label on the disk and make a single partition spanning the whole disk. We use the 'a' partition, but for compatibility with SunOS the 'c' partition might be more logical. In any case, Amoeba does not care.

Inside this one partition the Amoeba disk driver code will recognize subpartitions. These are the building blocks of the Amoeba *vdisk*'s. For bootstrap purposes four virtual disks are needed. For simplicity each one should consist of a single subpartition.

vdisk:01 This is used for booting Amoeba from disk and should have room to contain one or more kernels. In almost all normal circumstances 2 MB should be plenty. More than that is probably a waste of space. Minimum 1 MB.

vdisk:02 This is used by the *boot*(A) server, and 100 KB is sufficient.

vdisk:03 This is for the administration of the directory server. A 1 MB partition will hold over 30000 directories and should be sufficient for most purposes. It is a bad idea to make this partition much more than 1MB or the installation procedure may fail. It should be at least 500 KB.

vdisk:04 This will hold the files for the first Bullet Server started. It can take up the remainder of the disk.

⇒ Having planned the sizes for your partitions you should run the disklabel program:

```
# disklabel coldstart
```

This is an interactive program that is explained in *disklabel*(A).

NB. When labeling it is important to realize that the disk *bootp:01* is **not** a physical disk and should not be selected for labeling.

When creating the Amoeba subpartitioning the 'a' partition corresponds to vdisk:01, 'b' to vdisk:02, etc.

⇒ After the disk is labeled **reboot** the machine. To get a monitor prompt back press **L1** and **a** simultaneously. (If working with a terminal on ttya use **<CTL>^** followed by **r**. This will cause the machine to try to reboot. As soon as it begins to boot send a *break* character and the `ok` prompt should appear.

⇒ Now boot from tape again, and list the kernel directory again (with *dir*). (Do not forget the reset command if the machine is a sun4c.) This time you should see the entries for vdisk:01 up to vdisk:04.

⇒ Now set the time and date in the time of day server using the command *date*(U) command as previously described. Remember it has the form:

```
# date –s /super/hosts/coldstart/tod yyyymmddhhmm[.ss]
```

⇒ Start the rest of the installation by:

```
# . installit
```

First you will be asked if the machine type: either sun4m or sun4c. Older SPARCstations such as the SPARCstation 1, 1+, 2, SLC, ELC and IPC are of the type sun4c. Newer models such as the SPARCclassic and SPARCstation LX are of the type sun4m. Finally you will be prompted for the name of the machine and for its Ethernet address. The machine name must be a single word (no spaces or tabs) and the Ethernet address should be in the usual format (e.g., **8:0:20:0:12:34**). After that no further user input is required and the disk will be loaded

from the tape. It will print details of servers started and of each file as it is loaded so that it is clear that it is still active.

After the script finishes, which can easily take half an hour, it will print

```
Installation complete
```

⇒ You should boot Amoeba from disk. Go back to the Sun PROM monitor as described above and then type:

```
ok boot disk
```

Note that for this to work the disk must be the default disk. If not consult the Sun Openprom documentation to boot from the correct disk.

⇒ After some minutes an Amoeba login message should appear (be careful, it might be lost in the verbose boot server output), and you should log in as user *Daemon*. There is no password, so at the `Password:` prompt you should just press the return key.

⇒ The login name *Daemon* has the *super* capability giving access to the entire directory graph so it is important to protect this login against misuse. Once logged in you should set the password for the login name *Daemon* using the command *chpw*(U).

Now go on to the section on setting the time.

2.4.3. Installing Amoeba on the Sun 3/60

The distribution tape for Amoeba on the Sun 3/60 contains three tape files: a bootable kernel containing a ramdisk driver, the contents for the ramdisk, and a *tar*(U) image containing all the system binaries.

To use this tape the following hardware is required:

- A Sun 3/60 with exactly 12 MB of memory.
- A Sun keyboard and monitor, or a 9600 baud terminal on ttya.
- A QIC-24 tape drive. Note that a QIC-150 tape drive will not work with a Sun 3 boot PROM. (See the chapter *Trouble-shooting* for details.)
- A SCSI disk of at least 70 MB. 300 MB or more is recommended for a system with many users or if X windows is to be installed.

In the instructions below the **bold** font will be used for what you have to type at the computer, and the `standard font` for what the computer types back at you. A ⇒ marks the beginning of each step.

⇒ Power on the machine and write down the Ethernet address it displays in the banner. You will need this later.

⇒ Insert the distribution tape into the tape unit, first making sure that it is read-only, and boot from tape by typing

>b st()

The tape should spin and the kernel should boot. If this fails consult the chapter *Trouble-shooting*. Once the kernel boots it starts to initialize itself, loads the ramdisk from tape, and starts various servers. While all this is going on various messages will be displayed. They come from the boot server. Do not worry about them. After some minutes a shell prompt should appear:

```
Welcome to standalone Amoeba
#
```

You can verify that something works by typing

dir –l /super/hosts/coldstart

which will give a directory listing of all devices present on your machine of which one will be the disk:

```
bootp:00                              @     70498 KB
```

The '@' is the symbol for a disk and the number after it is the size of the disk in kilobytes. In this example it is a 70 MB disk.

At this point you need to partition and label your disk. The Amoeba disk driver code recognizes Sun disk labels and partition tables and makes it possible to use part of a disk for Amoeba and part of it for UNIX. However, this is not supported for booting.

It is recommended to put a SunOS label on the disk which specifies only one partition, spanning the whole disk. We use the 'a' partition, but for compatibility with SunOS the 'c' partition might be more logical. In any case, Amoeba does not care.

Inside this one partition, the Amoeba disk driver code will recognize subpartitions. These are

the building blocks of the Amoeba *vdisk*'s. For bootstrap purposes four virtual disks are needed. For simplicity each one should consist of a single subpartition.

vdisk:01 This is used for booting Amoeba from disk and should have room to contain one or more kernels. In normal circumstances 1 MB should be plenty. More than 2 MB is probably a waste of space. Minimum 750 KB.

vdisk:02 This is used by the *boot*(A) server, and 100 KB is sufficient.

vdisk:03 This is for the administration of the directory server. A 1 MB partition will hold over 30000 directories and should be sufficient for most purposes. It is a bad idea to make this partition more than 1MB or the installation procedure may fail. It should be at least 500 KB.

vdisk:04 This will hold the files for the first Bullet Server started. This can be up to about 600 Mbytes in size. Larger partitions may work but if it is larger than this then there may be insufficient core to cache the inode table when a full Bullet Server is started later. It should be at least 65 MB.

If you have any disk left after that you can put it in vdisk:05 but you might as well give it to the Bullet Server.

⇒ Having planned the sizes for your partitions you should run the disklabel program:

```
# disklabel coldstart
```

This is an interactive program that is explained in *disklabel*(A).

NB. When labeling it is important to realize that the disk *bootp:01* is **not** a physical disk and should not be selected for labeling!

When creating the Amoeba subpartitioning the 'a' partition corresponds to vdisk:01, 'b' to vdisk:02, etc.

⇒ After the disk is labeled **reboot** the machine. To get a monitor prompt back the Sun standard **L1-A** does **not** work. Instead you should press the key labeled **R11** and then **r**. When working on ttya use **<CTL>^** followed by **r**. Then boot from tape again, and list the kernel directory again (with *dir*). This time you should see the entries for vdisk:01 up to vdisk:04.

⇒ Now set the time and date in the time of day server using the *date*(U) command. It has the form:

```
# date –s /super/hosts/coldstart/tod yyyymmddhhmm[.ss]
```

⇒ Start the rest of the installation by:

```
# . installit
```

This script will prompt you for the name of the machine and its Ethernet address. The machine name must be a single word (no spaces or tabs) and the Ethernet address should be in the usual format (e.g., **8:0:20:0:12:34**). After that no further user input is required and the disk will be loaded with the binaries and data files. It will print details of each file as it is loaded so that it is clear that it is still active.

After the script finishes, which can easily take half an hour, it will print

```
Installation complete
```

⇒ You should be able to boot Amoeba from disk. To do this go back to the Sun monitor, as described above and then type:

>b sd()

⇒ After some minutes an Amoeba login message should appear (be careful, it might be lost in the verbose boot server output), and you should be able to log in as user *Daemon*. There is no password, so at the `Password:` prompt you should just press the return key.

⇒ The login name *Daemon* has the super capability giving access to the entire directory graph so it is important to protect this login against misuse. Once logged in you should set the password for the login name *Daemon* using the command *chpw*(U).

Now go on to the section on setting the time.

2.5. Setting The Time

Earlier in the installation, the time was set but the timezone information was not yet present and so the time of day clock was not totally correct. Before proceeding further it is a good idea to set the timezone and reset the time so that the timestamps on the directory entries bear some relationship to the time that the source code and new directory entries were actually installed.

The first step is to determine which timezone you are in. This is usually based on the country that you are in. Assuming that you are still logged in as *Daemon*, change directory to */super/module/time/zoneinfo* and look at the available countries/regions . To do this use the commands:

cd /super/module/time/zoneinfo
dir –l

This should print a list of available timezone files and directories.

Find the file that corresponds most closely to your timezone. If there is no file or directory with the name of the country you are in, look for a file with the name of your region. For example, if you live in Japan or China the file *asia* covers your region. If you live in the USA, Canada or Mexico then you will need the file *northamerica*.

Note that files covering regions begin with a lowercase letter and files covering actual timezones for a country begin with an uppercase letter.

You will need to generate the timezone file for your country if it is not already present. To do this use the command *zic*(A). For example, if you live in Japan type the command:

zic –d . asia

This will produce several files, one of them called *Japan*. The next step is to link this file to the file *localtime*. The way to do this is as follows:

get Japan | put localtime

If you live in a country like the USA or Canada which has many timezones for one country the process is slightly more complicated. For the USA, Canada or Mexico run the command:

zic –d . northamerica

However, instead of generating a single file for the countries USA, Canada and Mexico it generates directories called *Canada*, *Mexico* and *US*. In each directory is a set of files, one for each timezone in each country. Choose the timezone for the part of the country that you are in (say *Pacific*) and make a link from there to the file *localtime* in the *zoneinfo* directory. For example, for the USA this can be done with the command:

get US/Pacific | put localtime

After this it is necessary to set the time again using the command *date*(A). Recall that it has the form:

date –s /super/hosts/`xxx`**/tod** `yyyymmddhhmm[.ss]`

where *xxx* is the name of the host that was given to the *installit* program.

If you intend to have more than one time of day server, you should set each server separately once the other servers are installed.

2.6. Installing the Source Code

The source code is distributed in *tar*(U) format. It is distributed on either Exabyte tape or QIC-24 tape. If you FTPed the sources then you can put the tar file on the medium of your choice.

The sources do not include those for third-party software such as X windows, MMDF or TEX. Only the necessary changes for third party software to run under Amoeba are on the tape. Third party source code can be obtained from your Amoeba distributor if it is not available from any other source and you satisfy the licensing/distribution requirements for that software (if any).

The first step is to read the tape in. This requires about 25 MB disk space. In fact it is a good idea to reserve about 100 MB for the entire distribution. If X windows is to be loaded then another 200 MB should be reserved for sources and binaries. On Amoeba systems without a tape drive, the source will have to be loaded onto a UNIX host with a tape unit and installed on Amoeba over the network. See the chapter *Amoeba and UNIX* later in this manual for details.

To install the sources from Amoeba, insert the tape into the tape drive and type the command

```
# cd /super/module/amoeba
# tar xf /super/hosts/machine/tape:00
```

where *machine* is the name chosen during the binary installation for the host where you are installing the system.

It is possible that the UNIX binaries are also on the source tape. In this case it is probably a good idea to delete them from your Amoeba system if disk space is running short and they are not needed.

2.6.1. Installing the On-line Manuals

The on-line manual pages for Amoeba can be viewed using the command *aman*(U). This command can either display preformatted manual pages or, if no preformatted version is available it will try to produce the manual page from the source. Under Amoeba the latter is not possible since the text formatting programs are not available. Normally the preformatted manual pages are installed as part of the binary distribution.

Should it be necessary to remake or add to the manpages this must be done under UNIX. This is done by going to the top level of the distribution tree and then changing to the directory *lib/man*. The command *catman* can be found here and it should be run without arguments in this directory. Once it has successfully completed the manual pages should be installed under Amoeba using *amdumptree*(A). The command to do this is

```
amdumptree `amdir`/lib /super/module/amoeba
```

2.6.2. The /super Directory Graph Structure

When Amoeba is installed it creates a cyclic, directed, directory graph structure. Some parts of this graph look the same for all users and some parts are customizable per user. In addition, there are some parts of the directory graph that normal users never see. These are accessible via a special directory called */super*. This capability is normally only available to system administrators, known also as *super-users*. In the description below, users without the */super* capability will be referred to as *normal users*.

Super-users see all the same things as normal users, as described in the introduction to the User Guide. In addition they also see the special subtree called */super*. Below is a description of what the Amoeba directory graph looks like from */super*.

/super

Only trusted users should be given a copy of the capability for the directory */super*. This directory gives read/write access to the entire directory graph. All copies of the capability for this directory should always have the mode `FF:0:0` to prevent other users from gaining access to the ‘‘hidden’’ parts of the directory structure.

/super/admin

This directory is not visible to normal users. It contains data and commands for administration of the system. This directory contains the hostname information for the system, the */Environ* file template used by *newuser*(A) and several subdirectories. The important subdirectories are:

bin – the binaries for system administration.

bootinfo – this directory contains the *boot*(A) server binary, the *soap(A)* server binary and the *boot* server’s database.

kernel – the various kernels (for example, workstation, pool) and secondary bootstraps are stored here.

module – various system servers with sensitive data (for example, the run server) can store it in a subdirectory of this directory.

tftpboot – this directory contains links to entries in */super/admin/kernel*. The entries in this directory which conform to the naming requirements of the *tftp*(A) server will be used for booting hosts that boot using `TFTP`.

/super/cap

This contains subdirectories, one for each system server that needs to present a capability to users. For example, the capabilities for the various Bullet Servers are stored in */super/cap/bulletsvr*. Normal users have read-only links to most of the subdirectories of */super/cap*. They access it via the name */profile/cap*. One entry which users do not normally have a link to is */super/cap/bulletsvr*. Usually each user has a private directory */profile/cap/bulletsvr* with a copy of one of the Bullet Server capabilities registered under the name *default*. This can then be easily changed on a per user basis to help balance the disk usage and the load on the file servers.

/super/group

This directory contains only subdirectories. Each subdirectory contains the root capabilities for the members of a group. For example, the root capability for each PhD student can be found in */super/group/phd*. Furthermore, each capability in */super/group/phd* grants access to the second column (i.e., group access) of the directory. In this way all the members of a group have group access to each others' root directories.

There is one exception to this and that is the directory */super/group/users*. This contains the capability for the root directory of all users and provides access to the 3rd column.

Each user has a read-only link to */super/group* called */profile/group*.

/super/module

This directory is used by various modules in the system to store publicly available data and binaries. Normal users have a read-only link to it called */profile/module*. Examples of information stored here include the X window system, the MMDF mail system, various compilers and their accompanying libraries and include files, python and the timezone data base.

/super/pools

Each subdirectory of this directory contains a processor pool as recognized by the *run*(A) server.

/super/unixroot

This directory contains several subdirectories necessary for the UNIX emulation. This path provides write permission to the various files in the UNIX emulation. Normal users have directories such as */bin* and */etc* which are in fact read-only links to */super/unixroot/bin* and */super/unixroot/etc*, respectively.

/super/users

This directory contains the root directories of all the users. It is a link to */super/group/users*.

/super/util

This directory contains the binaries for Amoeba specific programs which are not shell scripts and do not form part of the UNIX emulation. Normal users have a read-only link to this called */profile/util*.

2.6.3. Amoeba Distribution Directory Structure

This section gives a guided tour of the distribution tree structure for Amoeba, which consists of the source code, documentation, and configuration tools. In addition to the sources, programs that run under UNIX have been distributed with the sources to facilitate installation under UNIX. There is a separate installation package which contains all the binaries which are to be installed directly under Amoeba.

The binary objects for UNIX and Amoeba are not built in the source tree since several

different configurations may be required at one site. For example, a site having more than one machine architecture which runs Amoeba, or wanting to build both the UNIX and Amoeba utilities (which share source code) at the same time. The program *amake* is used to configure the distribution, with the exception of third party software which may have its own build commands, such as the X windows system.

At the top level of the distribution tree there are five directories:

bin.scripts

bin.<mach> These directories contain the binaries of Amoeba utilities that run under UNIX. The *bin.scripts* directory contains shell scripts and shared data files. There is also a subdirectory per supported architecture/operating system version. The directory *bin.scripts* and the appropriate machine subdirectory should be put into your path before attempting to build Amoeba under UNIX.

doc The *doc* directory contains the source of the on-line documentation and the source for the printable version of the manual.

lib The *lib* directory contains support files for various programs. In particular the *cat*able versions of the manuals are stored here.

src The *src* directory contains the sources for the Amoeba kernel, administrative programs, utilities plus the Amoeba network driver for UNIX kernels and UNIX specific utilities. In the source directories there are files with the name *Amake.srclist*. These are lists of the sources which comprise a particular object (such as a utility). These are included by the *Amakefiles* in the template tree which actually build the object.

templates The *templates* directory contains the templates of the *Amakefiles* for configuration tree for Amoeba. The templates are parameterized with the path name of the source tree to use, the configuration tree where the binaries are to be made and the architecture for which the binaries are to be compiled. The kernel *Amakefiles* are also parameterized with the machine type.

The various subdirectories are now described in detail.

bin.scripts and bin.<*mach*>

These subdirectories contains the binaries and shell scripts that run under UNIX but talk with the Amoeba system. They are divided into subdirectories by machine/operating system. For example the subdirectory *bin.sun3* contains the binaries and shell scripts that will run on the Sun 3/50 and 3/60 under SunOS 4.1. The subdirectory *bin.sol* contains the binaries that run on sun4 systems running Solaris.

doc

The reference manual is written using troff and related tools. The *ms* macros are used with some slight modifications and extensions which are in the file *am_macros*.

The reference manual is in the directory *ref_manual*. The reference manual is divided into four sections, each having a separate subdirectory. Volume 1 is the users' guide, which contains introductory material about Amoeba and how to use the utilities. This is in the

subdirectory *user*. Volume 2 is the programmers' guide, which contains detailed material about library routines, programming tools and how to write programs for Amoeba. This is in the subdirectory *programming*. Volume 3 is the system administrators' guide, which contains information about how to install, boot and maintain an Amoeba system. This is in the subdirectory *sysadmin*. In addition there are release notes in the subdirectory *release*.

lib

This contains library data and the *cat*able versions of the manual pages.

lib/whereis

The database used by the program *amwhereis* is stored in this directory. The database is created the first time *amwhereis* runs.

lib/man

This contains the program *catman* for generating the data files required by the *aman* command. The data files are also stored in this directory or subdirectories hereunder. The source for the manual pages is in the *doc* subtree.

src

The source directory is the largest and most complicated of the distribution tree. It contains all the sources that are directly a part of the Amoeba system. Third party software such as X windows is provided separately from the Amoeba distribution. It is usually installed in the top-level directory *thirdp*.

Note that nearly all assembly code has to be passed through the C preprocessor since it has been written in a ''higher level'' macro language so that it can be compiled by the different assemblers.

The subdirectories of *src* are now described in alphabetical order.

src/admin

This directory contains the source code for the administrative utilities and servers. Various shell scripts are in the subdirectory *scripts*. The source for servers is kept separate from related administrative utilities. For example, the subdirectory *soap* contains the utilities and *soapsvr* contains the source code for the actual Soap Server.

src/Amakelib

This directory contains the definitions of the *amake* tools used to build the Amoeba distribution.

src/X11R6.am

Porting X to Amoeba required altering some of the source files. This directory contains the sources that needed modification. The subdirectories parallel the original X distribution tree structure. Since X is compiled using a configuration tree with symbolic links to the sources (instead of path names as with Amoeba) the relevant symbolic links in the configuration are changed to point to the modified version (see *Installing the Source Code*.)

src/h

The include files are stored in this sub-tree. At the top level are the include files used by user programs and libraries and which are not specific to the UNIX emulation or a server.

src/h/ajax

This directory holds the include files specific to the *Ajax* POSIX emulation.

src/h/byteorder

There are two include files to support architecture dependencies due to byte ordering, one for little-endian and one for big-endian architectures. Programs that need these include files do not include them from here but get them from the *machdep* subdirectory where there are links to these files.

src/h/class

This directory contains AIL class definitions for various servers and some include files. The include files are generated by AIL and probably not worth reading too closely.

src/h/machdep

This sub-tree contains the include files for the machine dependent code. It is divided into three sub-trees:

arch architecture dependent includes, including a link to the byteorder include file.

dev device dependent include files

mmu mmu dependent include files.

src/h/module

The module include files are primarily function prototypes for specific library modules such as server stubs.

src/h/posix

This sub-tree is the collection of STD C and POSIX include files.

src/h/server

The server sub-tree contains the include files specific to particular servers. For each server there is a subdirectory.

src/h/sys

This directory has the include files for the machine independent part of the kernel.

src/h/window

This directory contains include files for the original window manager for Amoeba. These are still used by the kernel for display management on the Sun3.

src/kernel

This directory contains all the source code for the Amoeba kernel. It is divided into the machine dependent code in *machdep*, the servers that may reside in a kernel in *servers* and the machine independent code in *sys*.

src/kernel/machdep

As with the include files the machine dependent code is divided into architecture dependencies, mmu dependencies and device drivers.

src/kernel/machdep/arch

The architecture dependent code is typically the exception handling and system call interface. There is a subdirectory for each supported architecture.

src/kernel/machdep/dev

A device driver is classified as *generic* if it works for all machines. There are few of these. The rest of the devices are divided up according to which machines they work on and there should be a directory here for each supported machine.

src/kernel/machdep/mmu

For each supported memory management unit there is a subdirectory containing the run-time start-up for the kernel and the routines for managing the memory mappings.

src/kernel/server

For each kernel server there is a subdirectory for that server containing the necessary source code for it. Some of these servers can run in either kernel or user space.

src/kernel/sys

This contains the machine independent part of the kernel. It includes the transaction layer, system initialization and one special server which is not optional for the kernel: the ‘sys’ server which provides the kernel directory and access to internal system tables.

src/lib

This contains the source code for the Amoeba libraries. Some of this is specific to native Amoeba but much of it is also useful for programs which run under UNIX and do Amoeba transactions. It also contains the run-time start-up for the native Amoeba executables.

src/lib/head

For each architecture there is a subdirectory containing the source code for the run-time start-up for native Amoeba of that architecture. It is typically written in assembler.

src/lib/libajax

This directory contains the sources for the part of the POSIX emulation that requires a session server to manage I/O.

src/lib/libam

This directory contains the source code for the architecture independent utility routines that are specific to Amoeba.

src/lib/libc

This directory contains the implementations of STD C and POSIX required functions for use in native Amoeba programs. It is divided into several subdirectories.

ajax	The POSIX emulation routines that are independent of the session server.
machdep	The machine dependent code, which is in many cases written in assembler for efficiency. For each architecture there is a sub-tree containing the various routines classified according to function: either string handling, memory handling, miscellaneous or system based. Where possible a portable version of the routine has been written and stored under the *generic* sub-tree.
misc	This directory contains miscellaneous routines such as malloc and getopt.
stdio	This directory contains the STD C stdio library routines.
time	This directory contains the STD C / POSIX routines for manipulating time.

src/lib/math

This contains the STD C math library routines. For each architecture there is a subdirectory plus a generic subdirectory containing a portable version of each routine.

src/lib/stubs

For each server interface there is a subdirectory containing the RPC stubs for that server. In general, stubs return the type *errstat* which is the error status of the RPC.

src/lib/sysam

This directory contains architecture specific routines for Amoeba programs and in particular the system call stubs. For each architecture there is a subdirectory and in addition a directory for generic routines.

src/mmdf.am

This directory contains those source files from the MMDF II mail system that were modified from the original sources. Note that if the MMDF II mail system is to be built then under UNIX a symbolic link or similar must be made from *src/mmdf* to the original sources. Under

Amoeba the sources should be installed under *src/mmdf*.

src/test

This sub-tree contains utilities and test programs for Amoeba. There are tests for the kernel, utilities and servers. They serve as a set of acceptance tests for the system.

src/tex.am

This directory contains those source files from the TEX text formatting system that were modified from the original sources. Note that if the TEX text-processing system is to be built then under UNIX a symbolic link or similar must be made from *src/tex* to the original sources. Under Amoeba the sources should be installed under *src/tex*.

src/unix

This sub-tree contains the Amoeba network driver that runs in the UNIX kernel, system dependent library routines (such as system call replacements) and special utilities and servers that only run under UNIX.

src/unix/admin

Administrative utilities and servers which are only capable of running under UNIX are kept here, each in its own subdirectory.

src/unix/flip-driver

This contains the source code for the FLIP driver that runs in the UNIX kernel. UNIX.

src/unix/h

The include files that are specific to Amoeba under UNIX are kept here.

src/unix/lib

Library routines specific to programs that run under UNIX such as system call replacements are kept here.

src/unix/sol_flipd

This contains the flip driver sources for the Solaris version of SunOS. It is based on streams.

src/unix/util

There are a few user utilities specific to UNIX. Many programs written for native Amoeba also run under UNIX but their source code is kept in *src/util*. The most important utility is *ainstall*(U) which installs an Amoeba executable onto a Bullet file server so that it can be started from UNIX on an Amoeba machine. There are also programs for transferring an arbitrary file from a UNIX file system onto a Bullet Server and vice versa and for manipulating the Soap directory server.

src/util

This directory contains the user utilities used by native Amoeba. Many of them also work under UNIX although a few may have #ifdefs for this purpose. Each subdirectory contains one or more programs. For example the subdirectory *soap* contains several programs for manipulating the Soap Server. The *ajax* subtree is special. It contains versions of programs typically available under UNIX. The *scripts* subdirectory contains shell scripts. These are compatible with the Bourne shell and work with the shell provided with the distribution.

templates

The templates are used to generate the configuration tree for Amoeba. They are parameterizable *Amakefile*s. The idea is that using the tools provided, a configuration tree for a particular architecture can be made by cloning the template directory structure, with special editing of the *Amakefiles* before they are installed in the configuration tree. To each *Amakefile* is added the source tree root, configuration tree root and the architecture for which the configuration is to be made. This information is used to select the correct source files and libraries. Kernel templates are handled separately from the rest of the process since the *Amakefiles* may need local adjustments or other servers and drivers from those selected by the standard *Amakefile*.

There are several subdirectories each of which is briefly described.

amoeba

This sub-tree contains the *Amakefiles* for the libraries, servers and utilities for native amoeba.

doc

This contains the template for the *Amakefile* for building the Amoeba manuals.

kernel

There is a subdirectory under *kernel* for each machine type. For each machine type there is a set on configuration files for different types of kernel. Typically there will be a workstation kernel, a pool processor kernel, a Bullet Server kernel and an IP server kernel.

toolset

For each set of compilers, loaders, etc. there is a standard set of clusters and tools for making the libraries, programs and kernels. These can be further tuned on the basis of local preference for a particular compiler or loader.

unix

This sub-tree contains the *Amakefiles* for the libraries, servers and utilities for Amoeba programs that run under UNIX.

thirdp

The third party software that is not tightly integrated with Amoeba is kept here. In particular, GNU software is kept here so that it can be distributed independently of the rest of Amoeba. The program *patch* is also kept here. It can be used for including patches into extant source trees.

2.7. Installing Third-party Software

There is a large amount of freely available or licensed software which works under Amoeba. Installation of the currently supported third-party software is described below for each package. In general, the software is freely available from the Internet with little or no restriction on its use or redistribution. Where such restrictions apply, details are provided in the installation instructions. If you do not have the necessary network connection or bandwidth to obtain the software from the network it is possible to obtain the software from your Amoeba distributor for a small cost for the distribution medium, the making of the tape and any postal charges.

Note that besides the large packages there is also a small set of programs in the directory *thirdp*. Some of these come with an *Amakefile* and others simply with a *Makefile*. In the cases where *Amakefile*s have been provided they should be installed somewhere in the *templates* directory so that versions of the thirdparty software can be compiled in the normal configuration tree.

2.7.1. Installing X Windows

The X Window System is a product that is publicly available but it is not in the public domain. It may be used for commercial purposes. The current version of X that works with Amoeba is X11 Release 6. The following instructions are valid also for when the Amoeba and X sources are installed under UNIX.

Warning: The X sources, excluding the contributed software require about 111 Mbytes disk space. The standard *contrib* sources require at another 40 Mbytes. There are man non-standard contrib sources to collect as well. If you then build and install a configuration for Amoeba it will take as much as another 100 Mbytes per supported architecture. Caveat emptor and buy large disks!

If the X windows sources are installed under Amoeba they should be put under */super/module/amoeba/src/X11R6.* Under UNIX a link should be made from *src/X11R6* to the place where the X sources are installed. (A symbolic link is probably best since many tree searching programs do not follow symbolic links.)

Building an X configuration for a particular architecture is fairly complicated. To spare people from this, a shell script has been devised called *buildX*(A). It will build the configuration tree, adjust various links to point to the modified sources for Amoeba (which are found in the directory *src/X11R6.am*) and generate and execute the *makefiles* (see *make*(U)). See the relevant manual pages for details of how to use the script. Note well: the links made under Amoeba are hard links, not symbolic links and so it is important that no *del −f* (see *del*(U)) or *std_destroy*(U) be done on files in the configuration tree.

Once the sources are installed in the right place, the instructions on how to build the X configuration can be found in the chapter *Building Amoeba Configurations*.

2.7.2. Installing MMDF II

MMDF is an electronic mail transport system plus user interface programs. MMDF is an acronym for Multi-channel Mail Delivery Facility. The mail system can send mail to other Amoeba users, and if the TCP/IP server is present, mail can be sent to and received from sites on the Internet.

Only the changes to MMDF to make it run under Amoeba are present in the Amoeba source tree. This is in the directory *src/mmdf.am*. It is necessary to obtain the source for MMDF II to build the software. There may be restrictions on the availability of the source. It can be obtained either from your Amoeba distributor or from the Internet. The source code should be loaded under the directory *src/mmdf*. A symbolic link from *src/mmdf* to the actual sources is sufficient under UNIX.

Note that compilation of MMDF under Amoeba has not yet been tested. The MMDF binaries are not normally provided on the distribution tape. See *buildmmdf*(A) and the chapter on building the system for details of how to compile and install MMDF.

2.7.3. Installing TEX

TEX is a text formatting system which is freely distributable. It can be run on Amoeba hosts with large amounts of memory. (TEX programs are very large.) If there are no hosts with at least 12 MBytes of memory there is little point in compiling and installing TEX.

Only the changes to TEX to make it run under Amoeba are present in the Amoeba source tree. This is in the directory *src/tex.am*. It is necessary to obtain the source for TEX to build the software. This can be obtained either from your Amoeba distributor or from the Internet. The source code should be loaded under the directory *src/tex*. A symbolic link from *src/tex* to the actual sources is sufficient under UNIX.

The TEX binaries are not provided on the distribution tape. Organizing the TEX Makefiles to cross-compile for different architectures is difficult. To build them for a particular architecture it is necessary to install the sources under Amoeba and then install the replacement files from *src/tex.am* over the top of the originals. It should then be possible to configure and build the TEX binaries under Amoeba . Make sure that the only processors in your processor pool are of the target architecture! Note that compilation of TEX under Amoeba has not yet been tested. It is not supported!

3 Configuring an Amoeba System

3.1. Introduction

There are many aspects of an Amoeba system that are configurable. They will be discussed in the order that they need to be confronted just after having installed a new system.

The first problem is usually to add new hosts to the system so that there is more than one Amoeba host. This is necessary since there are several servers needed to make the system function satisfactorily, and they will perform much better if they do not all run on the same host. Also note that after the initial installation of Amoeba that the file server is in the processor pool. It should be removed from the pool as soon as other processors are available to perform the work.

The next step is to configure the various servers that are nearly always in demand or that are necessary for good functioning of the system. Some servers can be replicated or can replicate their data and there is a special server which will provide lazy replication of objects plus do garbage collection of objects that are no longer in use or accessible from the directory graph.

N.B. It is very important for the performance and security of Amoeba that a *run*(A) server be started. Under no circumstance should this server be omitted.

3.2. Configuring Amoeba Hosts

3.2.1. Adding New Hosts

Under Amoeba, users are not normally aware of which processors are running their jobs or even how many processors are present. However the system does need to know what hardware is available if it is to use it. Adding a new host to an Amoeba system is a fairly straight forward operation.

Install the hardware (according to the manufacturer's instructions) and attach it to the network. Then determine its network address. (Amoeba currently supports Ethernet as the network medium and so the following description is oriented towards that.)

The hard part is to think of a name by which the host is known. Users generally will not see these names but the system administrator will need some way of addressing a particular machine to load a new kernel into it. The system also needs a name (actually a capability) for a machine so that it can start processes on it. There are many guidelines for naming machines, the most important of which is, do not name it after the manufacturer of the machine. Choose a theme, such as boat types or dance names. Since you may have dozens of pool processors it may be a good idea to number them. For example, all the mc68000 processors might get the names *pogo00* to *pogo99*. **N.B.** Host names must not contain the character '.'. Otherwise such hosts will be ignored by the *run*(A) server and the process creation mechanism.

The next step is to create a capability for the new host and install it in the directory */super/hosts*. This is done using the command *mkhost*(A). Suppose one wished to create an entry for the host *pogo00* which has Ethernet address `00:00:2c:11:ab:09`, then the command

```
mkhost /super/hosts/pogo00 00:00:2c:11:ab:09
```

would achieve this. It is important to make sure that the capability has the right mode. For normal operation the mode `FF:2:4` is recommended. This prevents unprivileged users from accessing disks and kernel memory and from destroying processes they do not own. To set the mode you should type the command

```
chm ff:2:4 /super/hosts/pogo00
```

It should now be possible to download an Amoeba kernel into the host and to examine the kernel directory (using *dir*(U)). Downloading the kernel will depend on the type of hardware. Intel 80386, 80486 and Pentium machines can be booted from floppy. Sun 3 and Sun 4 hosts should be booted using *tftp*(A). If a system is already running it can be rebooted using *reboot*(A). Note that Sun 3 machines do not accept the *reboot*(A) command.

The subsequent steps taken now depend on the function of the new host. If it is to be a pool processor then the capability for the host needs to be put into one or more *pool* directories. For example, if the new hosts is called *pogo00* and is an i80386 machine then the commands

```
get /super/hosts/pogo00 | put /super/pool/i80386/pogo00
chm ff:2:4 /super/pool/i80386/pogo00
```

will install the hosts capability in the pool. Currently the processor-pool directories are grouped by architecture and are found in */super/pool*. By entering the capability in a pool directory, all the users who have that directory under their */profile/pool* directory will be able to use that pool processor to run their jobs. Note that it is possible to have more than one pool for a particular architecture. One might contain half the mc68000 machines and another the other half. Furthermore, a pool processor may be in more than one pool.

If a host is required to respond to the Reverse ARP (RARP) and/or TFTP protocols (for example, Sun computers) then it will need to have its Ethernet address and host name entered in the file */etc/ethers*. To do this the file */super/unixroot/etc/ethers* must be edited. (This is actually the same file as */etc/ethers* but the path name */super/unixroot/etc/ethers* gives write permission to the file and the other path name does not.) Continuing the previous example, the line

```
00:00:2c:11:ab:09 pogo00
```

should be added to the file.

If the host is also required to use the TCP/IP or UDP network protocols (for example, TFTP booting or to communicate with UNIX hosts or over the wide-area network) or for any reason needs to have its name mapped to an Internet address then it will be necessary to assign an Internet address to the host. (Do **not** just make up Internet addresses. Obtain a set through the appropriate channels.) The host name and the Internet address should be entered into the file */etc/hosts*. This is done by editing */super/unixroot/etc/hosts*. In the example, if we had assigned the Internet number 192.31.237.44 to *pogo00*, then the line

```
192.31.237.44 pogo00
```

should be added to the end of the file */super/unixroot/etc/hosts*.

If the file is to be booted using *TFTP* then it will also be necessary to add an entry to the directory */super/admin/tftpboot* which is the capability for the file containing the kernel to be

booted on the host. The name of the file entered in the *tftpboot* directory is heavily dependent on the system to be booted. Sun uses the Internet address written as a hexadecimal number. (For Suns it is therefore necessary to give each machine an Internet address if it is to be booted using TFTP.) In the above example the file would be called `C01FED2C` for a Sun 3, `C01FED2C.SUN4C` for a Sun4c and `C01FED2C.SUN4M` for a Sun4m machine.

One final complication of booting via TFTP is starting a RARP and TFTP server to boot the new hosts. It is necessary to install a special kernel (on different host from the *bullet* server) which contains the TCP/IP server. This kernel can be either the *tcpip* kernel or the *smbulip* kernel (short for *small bullet with TCP/IP*). See *installk*(A) for how to install this kernel on the hard disk. Thereafter the boot server (see *boot*(A)) can be used to start the RARP and TFTP servers and boot the other hosts. See below for details of where to find the *Bootfile*. An example configuration for the RARP and TFTP servers can be found in the default *Bootfile*. If the file server machine contains insufficient memory to run all these servers then the other hosts will have to be booted from UNIX.

Other systems of booting may require yet other support but the above examples should give sufficient indication of the steps involved in adding a new host. It is important at this stage to boot the pool processor and workstation hosts.

If a new workstation is added it is important that a login process be started that allows users to log in. To do this it will be necessary to tell the boot server (see *boot*(A)) to start the program *login*(A) on that machine when it comes up. The examples in the *Bootfile* (found in the directory */super/admin/bootinfo*) provided with the system should indicate how to do this. The basic sequence of steps is:

1. Add a suitable entry to the *Bootfile*.
2. Run *iboot*(A). Note that you should never use the **–I** option unless installing a new disk. A command of the form

   ```
   iboot -f Bootfile /super/hosts/xxx/vdisk:02
   ```

   (where *xxx* is the name of the host with the boot server's disk partition) should be all that is required.
3. Run the command

   ```
   boot_reinit /super/cap/bootsvr/xxx
   ```

   This should indicate success if the *Bootfile* was valid. If not then do not reboot the system under any circumstance. Rather run

   ```
   std_status /super/cap/bootsvr/xxx
   ```

   to find out the problem and then correct the *Bootfile* and repeat the installation process.

3.2.2. Deleting a Host

Deleting a host is basically the inverse operation of the installation. Ensure that the *boot*(A) server no longer uses the host to start a process. Next, bring down the kernel on the machine. Then delete the directory entry for that host from the */super/hosts* and *pool* directories. Delete the entry for the host from the */etc/hosts* and */etc/ethers* files. Delete any entries for

the host from bootstrap directories such as */super/admin/tftpboot*.

3.2.3. Replacing a Host

If you replace a host with new hardware several things may change and need attending to. For example, if the new machine has a different Ethernet address, the entries in */super/hosts* and the *pool* directories must be recreated and the file */etc/ethers* modified to reflect the new address. If the Ethernet address remains unchanged but the architecture of the machine is different, it may be necessary to move the machine to a different *pool* directory.

3.3. Server Configuration

There are several servers which are essential to the good health and continuous operation of an Amoeba system. Some of them are built into the kernel but most of them run in user space. It is important to ensure that they are all highly available. This is done by either replicating the servers, by making sure that they are rapidly restarted in the event of failure, or both.

Let us begin with rapid recovery. There is a server, known as the *boot* server, whose function is to ensure that servers registered with it are running. It does this by regularly polling each server. The frequency with which it polls each server is set at registration time. If the server does not reply within a specified period it attempts to restart the server. For details of how to register a server with the *boot* server see *boot*(A). The system as installed from tape has a *boot* server. This is started from the kernel as the first user process and its first function is to start the directory server and the login daemon. It is, of course, possible to have two or more *boot* servers and for each *boot* server to register with another so that if one of them goes down it will be restarted.

Unfortunately it is not yet possible to ensure that a kernel is downloaded onto a machine using the *boot* server. This must still be done by hand. This problem might be addressed in a future release of Amoeba, although some aspects of it may be beyond software control, such as resetting a machine that has crashed, or detecting that there has been a hardware failure on the machine in question.

Replication of objects, and in particular servers, is a very effective way of providing high availability. Certain services are crucial to the functioning of the system. They are the *run* server, the time of day server, the random number server, the Bullet Server and the Soap Server.

One server that is essential for Amoeba's performance is the *run*(A) server. The function of this server is to select a processor on which a process is to be run. If the *run* server is not present then this can take a long time. It is also responsible for implementing the exclusive access features of the reservation server (see *reserve_svr*(A)). The *run* server should be started by the *boot* server after the directory server is running. It should preferably not run on the same host as the file server. See the manual page for the run server for details on how to put the various processor pools under its control. This is not automatic.

The time of day server and the random number server are kernel-based servers and so they are available if the kernel containing them is up. However it is perfectly possible to have these servers in more than one kernel. In fact it is strongly recommended. The server that everyone uses is known as the *default* server. In the case of the time of day server it is

registered as */profile/cap/todsvr/default*. However, if the host containing the time of day server goes down for any reason the many programs that rely on this server will fail if a new server is not quickly installed. The best way to do this is to have a second server available and let the various *boot* servers regularly poll the capability */super/cap/todsvr/default*. The first one to detect the absence of the server can immediately replace the capability there with the capability of a time of day server that is still up. An identical strategy is required for the random number server.

The login program is called *login*(A). This can also be started by the *boot* server. It is normally only used on a conventional terminal. If the workstation is running an X windows server then *xlogin* should be used. It is also a good idea to let the *boot* server ensure that an X server is running on the workstation which the *xlogin* must operate on. Note that *xlogin* need not run on the workstation upon which it operates but it is more reliable that way. Then the login program will be present whenever the workstation is up. It will be relying on the minimum amount of other software and hardware functioning correctly. In the case of an X terminal this is not possible and *xlogin* must run on some other processor.

It is possible to have more than one Bullet Server in a system. At present there is exactly one *default* Bullet Server but it is possible to register many capabilities in the directory */super/cap/bulletsvr* under the name of each server and then link one of them to the name *default*. Then if the default goes down it is possible to use the *boot* server to switch automatically to an alternate server. Alas, the files that are on the crashed server will be unavailable until it is restarted. It is also possible to let different users have a different default Bullet Server (and other default servers, for that matter) by making different versions of the */super/cap* directory and installing them in a particular user's */profile/cap* directory, since this is the path name the users use to access the information.

There are several advantages to having more than one Bullet Server. It means that a server that explicitly uses a Bullet Server for storing data can replicate it on a second server. Then if one Bullet Server goes down the server can continue functioning. Since the Soap Server allows multiple capabilities for an object to be stored in a single directory entry there is another advantage to having more than one Bullet Server. It is possible to do lazy replication of files from one Bullet Server to another so that using only one directory entry, either the original or its replica can be recovered depending on the availability of the file servers. To aid in replicating objects there is a special server called the object manager (see *om*(A)) which supports replication and garbage collection. It will be described shortly.

The Soap directory server (see *soap*(A)) is at the heart of management of objects under Amoeba. It maps an ASCII string to a capability-set for an object. The Soap Server can be configured in a large variety of ways, depending on the available hardware and the service availability requirements. Along with the Bullet Server, it is one of the most important servers on the system since it provides access to all the objects. It is dependent on a time of day server, a random server and at least one Bullet Server. Note that it does not use the default servers but its choice of servers is fixed when it is started.

It is possible to duplicate a Soap Server. If correctly installed, it is possible that two copies of *soap* share the same data base and handle requests. (Each keeps its own administration, however.) Updates are passed on to the partner via a special protocol. If either one of the servers goes down the other will continue to handle requests. When the other comes back up it will first bring its copy of the administration data up to date before proceeding.

The Soap Server can also duplicate its data base on more than one Bullet Server. This can be done regardless of whether or not the Soap Server is duplicated. There is also a special command *chbul*(A) to change either of the two Bullet Servers that *soap* is using.

Starting *soap* is quite a difficult business and is usually best accomplished by the *boot* server. (In fact, in the absence of another Soap Server, the *boot* server is the only way to start *soap*!) It is vital that when installing a new version of the Soap Server that you get it right first time if you have only one Soap Server. If *soap* is run in two-copy mode then it is possible to kill one of the servers and try to replace it with the new version. If that fails there is still another Soap Server running to allow a second chance at getting the installation correct. Note that this mechanism also allows installation of a new directory server with almost zero down-time. First replace one of the servers and bring up the new version. Then kill the other and bring up the new version. The only time the service is off the air is during the synchronization between the two servers when the second one starts and this is normally a matter of a few seconds.

The object manager (see *om*(A)) is used for two primary functions. The first is to replicate objects on multiple servers. The second is to support garbage collection. (The latter is necessary since servers have no way of knowing which of their objects are still required. If someone loses a capability for an object it could continue to consume resources forever since the server will never hear of the loss. See the chapter *Routine Maintenance* for more details.) *Om* works by regularly traversing the entire Soap directory graph and examining each entry. If it contains an object for a server that it must replicate, it replicates it. It also *touches* each object (see *std*(L)). This resets the life-time of each *touched* object to maximum. (This also happens whenever an object is accessed.) Once it has touched all the objects it then issues a command to the relevant servers to *age* all its objects by reducing their life-time by one. Any object whose life-time reaches zero is destroyed by its server.

The TCP/IP server (see *ipsvr*(A)) is used to convert between the Amoeba network protocol and TCP/IP. This is very useful for communicating with hosts on the Internet, X terminals or other operating systems on the same network. To install a TCP/IP server follow the instructions in the server manual page.

3.4. Adding New Users

Giving new users access to Amoeba is extremely simple. It has two aspects to it. The first is to create a home directory and the relevant information for a new user so that they can log in. The second relates to permitting access to Amoeba from UNIX hosts using the Amoeba network protocol. In order for users to access their accounts from UNIX they need a file in their home directory called *.capability* which contains the capability for their Amoeba root directory.

Creating an account for a new user is done with the command *newuser*(A). It creates a new directory with the name of the new account under the directory */super/users*. This directory then becomes the root directory of the new user. That is, it becomes the directory that the new user will think of as /. It then constructs a directory graph under the new account with links to *pool* directories and the *profile* directory. It will also create links to the UNIX emulation directories, such as */bin* and */etc*. All the capabilities that link to public directories have restricted rights so that the public information cannot be modified or destroyed. As an option, it is possible to give the capability for the directory */super* to a new user. However

this capability gives the user *write* and *destroy* permission for the entire directory graph so it should not be given to unreliable people or people who do not understand the nature of the directory system. (Since there can be circuits and arbitrary links in the directory graph it is easily possible to start a recursive walk through the graph which touches far more objects than expected!)

Note that the command *newuser* can be run from UNIX by someone with the super capability for the directory graph or it can be run under Amoeba by the system administrator logged in as *Daemon*. The user's new password will be requested at the end of the process. The password is stored in encrypted form in the file *Environ* in the user's new root directory. (See *chpw*(U) for more details about changing passwords.)

The second aspect of adding a new user is the possibility of accessing Amoeba from under UNIX. In this case, instead of getting the capability for the root directory by giving a password, the capability for the root directory is stored in the file *$HOME/.capability*. There is a command called *sendcap*(A) which can be run by someone with the super capability for the directory graph and this will mail the *.capability* file (uuencoded) to the user along with instructions about how to install it.

There is, of course, a chicken and egg problem here for those who installed Amoeba using the installation tape. They have to get the first *.capability* file installed under UNIX before the *sendcap* command (or any other Amoeba command under UNIX for that matter) will be able to access Amoeba. The easiest way to do this is to use the command

```
c2a /
```

to print out the capability for / under Amoeba and then create a binary file called *.capability* using the inverse program *a2c* in your home directory under UNIX with the binary contents as per the output of *c2a*. Needless to say it is not a good idea to do this while others standing around and able to make a note of the capability printed. Be sure that the *.capability* file under UNIX has the mode 400.

4 Building Amoeba Configurations

4.1. Introduction

This section describes how to build the complete Amoeba distribution from the source code. There are several ways different sections of the distribution which can be built separately depending on your requirements.

The four major sections are:

- The Amoeba kernel.
- The servers and utility programs for native Amoeba.
- The Amoeba RPC driver for the UNIX kernel.
- The servers and utilities that run under UNIX that use Amoeba RPC.

Building of the last two is described in the chapter *Amoeba and UNIX* later in this manual.

Since Amoeba can be built for many architectures and machines the configurations are not built in the source code tree. Instead, special tools (see *amoebatree*(A), *kerneltree*(A) and *doctree(A))* exist for building configuration trees which are parameterized by architecture and, in the case of kernels, by machine type. One of the requirements of Amoeba is that all machines of a particular architecture (that is, CPU type) run exactly the same binaries. This means that the user programs are made per architecture and not per machine type. For example, the user and administration programs compiled for the *mc68000* architecture run on Sun 3/60s, Motorola MVME 134 processor boards and Force 30 boards. Each board has a CPU of the same architecture but a different memory management unit or CPU version. The kernels for these different machines are naturally all different.

In addition to these requirements, different compilers have different implementations of various aspects of the C language and require different include files (for example, in relation to *setjmp* and *longjmp*). Also, the differences between STD C and old-style C can play a role. Therefore it is also necessary to specify which compiler tools are to be used to build the system.

This is all taken into account by the programs that build the system configurations. They work by copying the *templates* directory tree structure. Each leaf directory contains an *Amakefile* (see *amake*(U)) which is edited as part of the tree cloning process to contain the parameters such as architecture and machine type wherever these are relevant. On the basis of these parameters, the source code appropriate for a particular architecture or machine is selected from the source tree. Note that the sources are never copied but simply referred to by the *Amakefile*.

Toolsets

In addition, the configuration programs select a set of compiler tools known as the *toolset* and place a copy of it in the configuration directory tree. Amoeba is set up in such a way that it is possible to cross-compile for several different architectures using several different compilers. For example, you can use the Amsterdam Compiler Kit (ACK) to construct a version of Amoeba to run on the i80386 and the Sun C compiler, loader, archiver, etc. to construct a version for the Motorola 68000 architecture. To support this there are templates and parameter files for each set of compiler tools. The directory *templates/toolset* contains a directory of *amake* tools for each supported compiler set.

4.2. Building the Amoeba Utilities

This section describes the configuration and installation of the native Amoeba utilities, both user and administrative. In the examples below binaries for the *i80386* architecture will be built using the ACK compiler. Other architectures and compiler sets can be substituted. If after reading this you are uncertain about how to proceed, the script *makeconf*(A) gives an example of how to build an entire system from scratch, including the UNIX utilities.

Begin by constructing a tree in which the configuration is to be built. This is done with the command *amoebatree*(A). For example, suppose that your Amoeba distribution is installed in the directory */amoeba* and you want to build your configurations for native Amoeba in the directory */amoeba/conf/amoeba*. The command *amdir*(U) must return the string */amoeba* in this case. (Note that under Amoeba it is customary to let it point at */profile/module/amoeba*.) Make the directory */amoeba/conf* if it does not already exist and then run the command

amoebatree conf/amoeba src templates i80386 ack

This says that the configuration is to be for the *i80386* architecture and to be build using the ACK (Amsterdam Compiler Kit) compiler set. Since the path names were relative it prepends the output of *amdir* to them. Therefore this command will make the directories */amoeba/conf/amoeba* and */amoeba/conf/amoeba/i80386.ack* if they do not already exist and then proceed to build a clone of the tree */amoeba/templates/amoeba* under it. (If the configuration already exists it will go through the configuration adding any missing directories and interactively enquiring if it should update any files that are different from the template version.)

Once this process is complete it is possible to build the various libraries and utilities. Change directory to the configuration (in the above example */amoeba/conf/amoeba/i80386.ack*) and type the command

build > build.out 2>&1

This will take some time to complete. Note that the simplest way to check for errors is

chkbuild build.out

(see *chkbuild*(A)).

4.2.1. Building X Windows

If X windows is required then first build the Amoeba utilities as described above. The file *src/X11R6.am/xc/config/cf/Amoeba.cf* contains some configuration defaults which must be adjusted for the local system. The variables `AmoebaTop`, `AmoebaBin` and `AmoebaConf` need to be overridden with the correct local values. Once that is done then run the following command to build the X server and clients:

buildX /amoeba/conf/amoeba /amoeba/src i80386 ack > buildX.out 2>&1

If the contributed software should also be built then the **–c** option should be given to *buildX*. See *build*(A) for more details. This command may take as long as 8 hours to complete on slow systems. It can take as much as 2 hours on a moderately fast system. Inspect the output file to ensure that the build completed successfully. To install the binaries produced, change directory to */amoeba/conf/amoeba/i80386.ack/X11R6/xc* and type the command

make install

See the release notes to determine which version and what patch level of the X sources are required to compile X for Amoeba.

NB. It is not possible to build the X Windows system under UNIX for the Sun 3 with the *sun* toolset at present. If hardware floating point is used then it probably can be built with judicious editing of the X configuration files. It should build correctly for the Sun 3 with the *ack* toolset under both UNIX and Amoeba.

4.2.2. Building MMDF II

Before the binaries for MMDF II can be made it is necessary to make an Amoeba configuration as described above. The libraries *libajax.a* and *libamoeba.a* are used in linking the MMDF binaries. Once these libraries are present the next step is to run the command *buildmmdf*. This is a trivial shell script and it should be called as follows:

cd `amdir`/*conf*/**amoeba**
buildmmdf *conf src arch toolset* **> buildM.out 2>&1**

where *amdir*(U) provides the root of the Amoeba distribution tree, *conf* is the name of the configuration directory where the binaries should be made, *src* is the name of the source directory where the *mmdf* and *mmdf.am* directories *arch* is one of the standard architectures such as *sparc* or *i80386* and *toolset* specifies which compilers to use, for example, *sunpro* or *ack*. This will make a configuration tree in

`amdir`/conf/*amoeba*/arch.toolset/*mmdf*.

It assumes that the sources can be found in *`amdir`*/src/*mmdf* and *`amdir`*/src/*mmdf.am*. The process of making all the binaries for MMDF takes some considerable time. When the build completes inspect the file *buildM.out* to ensure that the compilation proceeded correctly. If it is desired to recompile a particular subdirectory of the configuration then go to that directory and type

gen

Typing *make* will not work except at the top levels of the configuration tree.

If the binaries have compiled satisfactorily then they can be installed by changing to the top level configuration directory and typing the command

make aminstall

This installs the system binaries in */super/module/mmdf*. The user utilities such as *msg* and *send* are installed in */super/util*.

To install a particular binary then go to the directory where it was made and type

gen aminstall

For details of how to configure MMDF see the MMDF documentation in *src/mmdf/doc*. This is a non-trivial exercise and involves correctly setting up the *mmdftailor* file.

4.3. Building Amoeba Kernels

This section describes how to configure, compile and install the Amoeba kernel for the various supported machines.

The script *kerneltree*(A) is used to build the configuration tree for Amoeba kernels.

Note well: you must make the Amoeba utilities' configuration tree and build the libraries as described in the previous section before you can build Amoeba kernels.

The *kerneltree* program has arguments similar to those of *amoebatree*(A). The extra parameter is for the specific machine type that you wish to make the kernel for. A typical call to build the kernel for the Sun4m machines is as follows:

```
kerneltree conf/amoeba src templates sparc sun4m sunpro
```

This creates the directory */amoeba/conf/amoeba/sparc.sunpro/kernel/sun4m* and under this several subdirectories. Typically they are *pool*, *bullet*, *smbul*, *tcpip* and *workstation*. These contain configurations to build kernels with appropriate drivers for pool processors, Bullet file servers, TCP/IP and workstations.

Select the type of kernel that is required, go to that directory and type *amake*. This will build the standard kernel of that type and store it in the file named *kernel*. This is the default target of *amake*. There are other useful targets that can be built. For example,

```
amake lint
```

Will run lint over the particular kernel configuration.

```
amake files
```

will create a file called *files* which contains the names of all the source files used to build the kernel, one per line.

```
amake tags
```

will create a tags file for the *vi* editor (under Amoeba it is the editor *elvis*(U)).

4.3.1. Configuring a Kernel

It is possible that the standard kernel is not suited to your particular hardware configuration. In this case it is necessary to modify the *Amakefile* to include or delete particular servers and/or drivers. For example, one of your machines has a disk but no tape unit and you need a Bullet Server kernel for it. In this case it is relatively simple to delete the tape driver and tape server from the standard bullet configuration. Edit the *Amakefile* and you will find that it begins by including various *Amake.srclist* files. These define various *amake* variables which are used to build up the list of `SOURCEFILES` which comprise the kernel. The symbolic names `K_SVR_TAPE` and `K_DEV_SCSI_TAPE` refer to the tape server and the SCSI tape device driver respectively. Deleting these two names and the file *tapeconf.c* from the source list and then running *amake* will result in a kernel without the tape server or driver. In principle configuring a kernel is that simple, although there are some exceptions. The addition or deletion of a define may be necessary for some modifications.

For some systems it might be necessary or interesting to modify the FLIP configuration parameters. This can by done by adding defines to the `FLIP_CONF` variable. It is also possible to override the number of threads and the stack size for a kernel server. For

example, for the virtual disk server you can add special defines to override the defaults in the source code. In this case would modify the line

```
$K_SVR_VDISK,
```

to add some extra defines particular to the server. It then looks like

```
$K_SVR_VDISK[flags={ '-DNR_VDISK_THREADS=5', '-DVDISK_STKSIZ=6000' }],
```

This sets the number of virtual disk server threads to five and the stack size to 6000 bytes. To discover the names of these variables it will be necessary to look in the source for the server (typically found in *src/kernel/server*) but they should all be of the form NR_name_THREADS and name_STKSZ.

At the bottom of the *Amakefile* is a variable called LDADDR. This defines the virtual address at which the text segment of the kernel begins. If the variable LD_DATA_ADDR is defined, it determines where the kernel data segment will begin. If it is not defined then the kernel data segment will begin at the default place for the architecture/compiler combination.

In addition to these basic kernel features there is a large number of defines which can modify the nature of the kernel. A description of some of the possible defines follows for the true enthusiast. This list is very difficult to keep current. Please report any omissions or incorrect information to your distributor.

The Kernel Defines

This section lists most kernel configuration parameters. Per parameter, it tells what functionality the define selects or modifies, what other defines it needs to function correctly, for which configurations it is useful, and what other features depend on it.

BS_MEM_RESERVE

This define can be set to the amount of main memory a kernel-based Bullet Server will leave free for other uses. (Normally a Bullet Server takes all the free memory it finds.) For a machine that will only run a Bullet Server it can be set very low. The default assumes some user processes and is set to 3 Mbytes.

Configuration: Any kernel with bullet server support; required.

Affects: the Bullet Server.

BYTES_REVERSED

This define tells the lance driver that the in-core structures used by the lance have their bytes reversed. It might have to be set on some big-endian machines.

Configuration: Kernels for big-endian machines with a lance network interface; required.

Affects: the lance driver.

COMM_LOCK

Enables the inclusion of a few routines to use test-and-set instructions on 68K based machines.

Configuration: Kernels for 680x0-based machines; optional.

Affects: as.S

FLIPGRP

Set this flag when making a kernel with the group communication primitives.

Affects: FLIP RPC code.

IOMAPPED

Define this if I/O to the lance device registers is through I/O space, not memory mapped. Probably only needed for 8086 type machines, do not define.

Configuration: None that I know of.

IOP

This define enables support for the IOP driver, the low-level interface to the screen, keyboard and mouse used by X windows. Currently, IOP drivers are available for a monochrome VGA/i80386/i80486 systems, for the monochrome Sun 3/60 and both color and monochrome Sun 4 systems. This define should be used in workstation kernels.

> The IOP driver for this machine should also be included. Also, it might be necessary to include the uart driver even with NOTTY defined.

Configuration: A workstation kernel for a i80x86, a Sun 3 or a Sun 4; required.

Affects: the uart driver, maybe some other things.

LADEBUG

Enables some (reasonably cheap) debugging output on lance errors, etc. This option should probably be defined, unless the volume of the debug output to the console is too high to be comfortable, on workstation kernels, for instance.

Configuration: Any machine with a lance; recommended.

Affects: lance.c

LADUMP

Enables code to dump the status of the lance to the console. It should probably be defined on most machines.

Configuration: Any machine with a lance; recommended.

Affects: lance.c

LANCE

Enables code to support the lance network interface driver in other modules. This option should be defined on all machines that use the lance network driver, machdep/dev/generic/lance.c.

Configuration: Any machine with a lance; required.

Affects: various files.

LAREBOOT

It is rumored that there are older revisions of the lance that will sometimes turn off their transmitter without giving an interrupt. This option enables code that tries to detect this situation and repair it. Do not use, unless you see the symptoms: a machine that refuses to talk to the outside world.

Configuration: Any machine with a lance; optional.

Affects: lance.c

LASTAT

Enables code to gather and dump lance statistics. It should probably be defined on most machines.

Configuration: Any machine with a lance; recommended.

Affects: lance.c

MAPCASES

Enables code in the tty driver to do upper case to lower case mapping on upper case only terminals. You probably do not want this.

Configuration: Any kernel with a tty driver; optional.

Affects: the tty driver.

NDEBUG

This define disables all general debugging code in the kernel, all assertions, the ability to print stack dumps on kernel crashes, etc. Since the debugging code toggled by this option is not expensive, it is not recommended to use this option for general use.

Configuration: All; not recommended.

Affects: Everything.

NODIRECTORY

This option disables the pseudo-directory maintained by the kernel. It is only used by bootstrap kernels. Note that kernel services are unreachable when this option is turned on, so it is rather useless to include any.

Configuration: Bootstrapping kernels; required.

Affects: systask.c

NONET

This option disables all networking. It is only used in the disk labeling kernel.

Configuration: Disklabel kernels; required.

Affects: Almost everything.

NOPROC

This option disables support for user processes. It is used in special purpose kernels like the bootstrap kernel. Depending on your configuration, it might be advisable to enable it on your file server, to disallow user processes on it. If you define this option you should also not include the sources in servers/proc.

Configuration: Kernels without support for user processes; required.

Affects: Many things.

NORANDOM

This option disables support for the random number numbers in the kernel. It should not be used.

NOTIOS

This option is for compatibility only: it instructs the tty driver to use old-style signals instead of POSIX compatible signals. Do not define.

Configuration: Any kernel with a tty driver; not recommended.

Affects: the tty driver.

NOTTY

This option disables support for the tty driver. Note that this means the high-level tty driver, it does not affect the kernel printing to the console. If this option is *not* set, the sources in server/tty should be included. Also, an uart driver for this machine should be incorporated in the kernel.

Configuration: Any kernel without tty server support; required.

Affects: the uart driver and various other things.

Dependencies: If this option is *not* set, an uart driver should be included in the kernel. Also, the options NO_INPUT_FROM_CONSOLE and NO_OUTPUT_ON_CONSOLE should not be set.

NO_INPUT_FROM_CONSOLE

This option disables handling input from the console in the uart driver. This option can be set in pool processor kernels and bootstrap kernels.

Configuration: Any kernel that does not need keyboard input; optional.

Affects: the uart driver and putchar code.

NO_IOCTL

This option takes ioctl support out of the tty driver. Do not use.

Configuration: Any kernel with tty server support; not recommended.

Affects: the tty server and the uart driver.

NO_OUTPUT_ON_CONSOLE

This option disables output on the console. Kernel output still goes to the console buffer where it can be examined using the *printbuf*(A) command. Actually, this option impliesd NO_INPUT_FROM_CONSOLE, and completely obviates the need for a uart driver. This can also be useful for preventing access to a pool processor from the uart.

Configuration: Any kernel for a machine without an uart; required.

Affects: main.c, some machine dependent files.

PROFILE

Enables support for kernel profiling. It is also necessary to include the profiling server. (See *profsvr*(A).)

Configuration: Any kernel; optional.

ROMKERNEL

This define should be set only for bootstrap kernels. It has many effects all meant to make the kernel as small as possible. Among other things it disables virtual memory support and makes the kernel use only a minimal number of network buffers.

Configuration: Bootstrap kernels; required.

Affects: Various files.

Dependencies: NOTTY, NOPROC, NORANDOM, and SMALL_KERNEL should be defined. NDEBUG should be defined as well. No drivers should be included.

`RTSCTS`

Disables support for modem control signals in the tty server and the uart driver.

Configurations: Any kernel with tty server support; optional.

Affects: The tty server and the uart driver.

`SMALL_KERNEL`

Use this to obtain a physically small kernel which still provides most of the normal Amoeba services.

`STATISTICS`

Enables code to gather some transaction layer statistics, obtainable with the kstat command. Define this for normal kernels.

Configurations: All; recommended.

Affects: the RPC layer, maybe some other things.

`VERY_SIMPLE`

This option strips most of the code out of the tty server, leaving only the bare necessities. Do not define this.

Configuration: Any kernel with tty server support; not recommended.

Affects: the tty server and the uart driver.

4.4. Building the Amoeba Documentation

The documentation, like the rest of the system is not built in the source tree. Instead it is built in the configuration tree. To build the documentation tree from the templates there is a special command called *doctree*(A). It functions in a similar way to the *amoebatree(A)* and *kerneltree*(A) programs. Unlike the source code for the utilities and the kernel, the documentation is not dependent on the architecture or a compiler. The documentation is currently written using the *troff* text formatter using the *ms* macro package. Unfortunately these tools are not available under Amoeba. Therefore it only makes sense to build the documentation under UNIX. In a subsequent release it is hoped that this problem will be rectified.

5 Amoeba and UNIX

It is possible to install the FLIP network protocol in many versions of UNIX. Although the performance of the network protocol under UNIX is often significantly poorer than with Amoeba, it makes it possible to access Amoeba systems from UNIX and vice versa. Commands can be started under Amoeba from a UNIX system. Furthermore, many commands such as *dir*(U) can be executed under either Amoeba or UNIX. Downloading of Amoeba kernels from UNIX is also possible, as is compiling Amoeba binaries under UNIX and installing them on the Amoeba file system.

It is possible to run an Amoeba shell in a UNIX window. The shell runs under Amoeba but uses the UNIX window for I/O. Amoeba can also communicate with UNIX using TCP/IP. This is convenient when it is desired to run a UNIX shell in a window on an Amoeba workstation.

Running commands under UNIX or starting them from UNIX is very useful when porting to a new machine and no other Amoeba machines are available. It is also useful when it is desired to use compilers which are available under UNIX but not available under Amoeba.

This chapter describes how to install the Amoeba sources under UNIX, how to build binary configurations for both Amoeba and UNIX, how to install the Amoeba network protocol in various versions of UNIX and how to install Amoeba binaries from UNIX onto an Amoeba system.

Important

Note that before beginning, the script *amdir*(U) in the top level directory *bin.scripts* must be modified as per the instructions for installing Amoeba. Furthermore, the directory *bin.scripts* and the appropriate machine subdirectory should be put into your PATH before attempting to build Amoeba under UNIX.

5.1. Loading the Source Tape

The Amoeba sources require approximately 25 Mbytes of disk space. This includes the binaries for a single version of UNIX. If you wish to build all the possible configurations then also allow 80 Mbytes for each architecture you wish to use. The reason for the large space for the binaries is X windows, which requires 60 Mbytes for the binaries. Note that the sources and binaries can be on separate disk partitions. The compilation process does not occur in the source tree. Therefore, to load the source tape it is only necessary to find a disk with 25 Mbytes free. When you have done that, make a directory under which the Amoeba sources will be stored and then change to that directory. Insert the source tape into the tape drive and then type the command:

tar xf /dev/*tapedev*

where *tapedev* is the name of the tape device in which the tape is inserted. On BSD-like systems this is probably **rst0** or **rst8**. If in doubt consult your local UNIX system administrator.

Once the sources are loaded it is necessary to do a few things which were (alas) not possible to do correctly on the tape. The first is to decide how to protect the configuration directories. The scripts which generate a configuration tree deduce the *umask* that they should use for creating files and directories from the script *src/admin/scripts/localumask*. If it should use the *umask* of the person creating the configuration then this script should simply be empty. If however a group or all people should have access to the configuration then set the *umask* in this file accordingly.

Many commands have been simplified by setting the root of the Amoeba distribution in the script *src/util/scripts/amdir*. This file should be edited to print the root of the Amoeba distribution under UNIX. The initial value is that for Amoeba.

The program *unixtree* is just a link to *amoebatree*(A). It is a good idea to check that the link is correctly set since the various versions of UNIX may not correctly translate the link as it is recorded on the tape.

If you wish to compile various third party software such as X windows then they should be linked into the source tree at the places described in the chapter *Installing Amoeba*. The various commands to build configurations can then be used to make the various binaries.

5.2. Building the UNIX Utilities

This section describes the configuring and installation of the UNIX utilities. The configuration tree is built using the command *unixtree* (which is a link to the command *amoebatree*(A)). For example, suppose you have installed the distribution tree under the directory */amoeba* and you wanted to build the `mc68000` version of the UNIX utilities using the SunOS C compiler and related tools. Further suppose that you wanted to build all your configurations under the directory `conf` then the following command will generate the entire configuration tree for the UNIX software that relates to Amoeba.

unixtree /amoeba/conf/unix /amoeba/src /amoeba/templates mc68000 sun

Note that this does not compile the software but merely constructs the tree with *Amakefiles* parameterized to build the mc68000 version using the Sun compiler. This tree will appear under the directory */amoeba/conf/unix/mc68000.sun*.

On *Ultrix* systems you should use the compiler set *ultrix*. It is also a good idea to keep the paths as short as possible on *Ultrix*. Certain libraries are too large for the shell to cope with the commands used to build them and so it is necessary to do the following to the configuration before running the build commands. In the directories *lib/amoeba* and *lib/kernel* in the Amoeba configuration and the directory *lib/amunix* in the UNIX configuration you should make a symbolic link called `S` to the source tree and edit the *Amakefile* so that the definition of the `SRC_ROOT` becomes `S`. This is necessary to keep the path names short since they are used in commands to generate the lint libraries and the library itself. A better alternative is to get a serious shell that can handle long commands.

The next step is to actually compile the programs in the configuration tree. The program *amake*(U) will be needed for this. The binaries for SunOS are distributed with Amoeba. For other versions of UNIX it will be necessary to first compile *amake*. This is done in the configuration tree. Go to the directory *util/amake* in the new configuration tree and follow the instructions in the `README` file.

Once a copy of *amake* has been installed the rest of the build process s largely automated. Change directory to the configuration (in the above example, */amoeba/conf/unix/mc68000.sun*) and type the command

build > build.out 2>&1

It is strongly recommended that you redirect the output since there is rather a lot of it. The command *build*(A) will traverse the configuration tree and execute any *Amakefile* that it finds, beginning first with any libraries.

The compilation time depends on the type of machine. On a SPARCstation it will take about 30 minutes, depending on the model and possibly much more on a slow machine. The *build* command lets *amake* run with the default amount of parallelism. This is currently set at four parallel jobs but if your machine does not have much memory (less than 8 megabytes) it may be advisable to edit the script `build.domake` and add the flag **`-p2`** to the line `$MAKE.`

To install the utilities in a *bin* directory use the command *amuinstall*(A). For details of this see the manual page.

5.3. Building the Amoeba Utilities under UNIX

This section describes how to build the Amoeba utilities under UNIX and then install them on an Amoeba system. The building of a configuration tree and compiling all the utilities is the same as described in the chapter *Building Amoeba Configurations* with the only addition being that some UNIX systems have problems with long command lines as observed in the previous section and the solution described there is equally applicable here. The configuration tree is built using the command *amoebatree*(A). Thereafter the various commands *build*, *buildlibs* and *buildX* (see *build*(A)) can be used to compile the various utilities, servers and kernels.

Once the build is complete there should be many utilities that should run under Amoeba. To run under Amoeba they must be installed on a Bullet Server. The binary cannot be started by loading it from a UNIX file system. It may be possible to construct a Bullet Server that runs under UNIX using a Virtual Disk Server running under UNIX but that is beyond the scope of this document.

The binaries produced by the compiler under UNIX have a UNIX *a.out* header at the front. This must be replaced with an Amoeba process descriptor and then the file installed on a Bullet Server. This is all done by a program called *ainstall*(U) or *tob* if the program was compiled using ACK. It uses the default Bullet Server unless otherwise instructed. If you wish to install an entire configuration then *aminstall*(A) can be used.

NB. Before the Amoeba binaries can be generated under UNIX it is necessary to build the UNIX utilities for Amoeba. In particular, the program *flex*(U) must be built since it cannot be distributed with the other UNIX binaries since it has various site dependencies.

5.4. The Unix Amoeba Driver

The following two sections describe how to install the Amoeba network protocol in a UNIX kernel. The version of UNIX currently supported is SunOS (version 4.0.3 and higher). The driver for Amoeba 4.0 worked in Irix 3.3 and Ultrix (version 2.2 and higher) and there is a reasonable chance that it could be made to work in those versions of UNIX. The installation procedures for the different systems follow.

Note that once the driver is installed in the UNIX kernel and the kernel is running, a capability will still be needed to access Amoeba. Therefore, once the Amoeba driver is installed return to these instructions which explain how to obtain the required capability.

Normally the capability for each user's Amoeba root directory (that is, /) is stored in the file *$HOME/.capability* under UNIX. There is a shell script *sendcap*(A) provided which runs under UNIX and which mails this file to a new user. (Note that only users with the *super* capability can run *sendcap*.) The problem is getting the first capability for the system administrator. This is done as follows. Log in to Amoeba as yourself. The following obtains the capability:

```
amoeba% c2a /
<ASCII representation of capability>
```

This information should be kept secret. Do not let others see it. To install this capability under UNIX, then do the following:

```
unix% a2c ´<ASCII representation of capability>´ – > $HOME/.capability
unix% chmod 400 $HOME/.capability
```

Test this by running the *dir*(U) command under UNIX. All the commands that access Amoeba from UNIX should now work.

5.4.1. Adding Amoeba RPC to a SunOS 5.x/Solaris 2.x kernel

If you have SunOS 5.4 (Solaris 2.4) or newer you can add Amoeba RPC as a loadable pseudo-device to the kernel while it is running. This is done with the aid of a package.

Step by step installation

The first thing to do is to create a binary file to link with the SunOS kernel. It is assumed that the configuration tree has been created as described in the previous section. The compiler to use is either the SUNWspro compiler or the GNU C compiler.

In this example we assume the SUNWspro compiler is used on a Solaris 2.4 system running on a sun4m machine. We also assume that no FLIP driver has been installed yet. If that is not the case then the old FLIP driver should be uninstalled before proceeding. See below for details of how to do this.

In the configuration directory *`amdir`/conf/unix/sparc.sunpro/flip-driver*) there is an *Amakefile* that will create the driver and related utilities. In this case it will create the files, *amoeba.sun4m.o* and *flipd*. This is done by typing the command

amake -DUNIX_VERSION=sunos5.4

Note that if you have SunOS 5.5 then use `sunos5.5` as the `UNIX_VERSION` argument. The *amake* command will also create programs called *flip_dump* and *flip_stat* which can be used to get status information from the enhanced UNIX kernel. These should be copied to the directory *`amdir`/bin.sol*.

To make the package, change directory to *PKG* and run *amake*. This will create 2 subdirectories, *VUCSflipr* and *VUCSflipu*, which comprise the package. Copy these directories to where you normally keep your packages (for example, */usr/local/packages*). This can be done using tar. Then run *pkgadd*. If `PKGDIR` is the place where your packages are normally kept then the following should achieve the desired result.

tar cf - VUCSflipr VUCSflipu | (cd $PKGDIR ; tar xf -)
pkgadd -d $PKGDIR VUCSflipu
pkgadd -d $PKGDIR VUCSflipr

If this completes successfully then run the command

/etc/init.d/flip-init start

to load and start the FLIP driver.

Uninstalling the FLIP driver

Before a new FLIP driver can be installed it is necessary to remove any active FLIP driver from the kernel. This is relatively simple. The steps are:

1. Kill all processes using the FLIP driver (e.g., *ax*(U)).
2. Kill the *flipd* process.
3. Get the module id of the currently loaded FLIP driver. It is the first number on the line produced by the command

   modinfo | grep flip

4. Run the command

   modunload *module-id*

   where *module-id* is the number found in step 3.
5. Remove the old package from the kernel.

   pkgrm VUCSflipr
   pkgrm VUCSflipu

5.4.2. Adding Amoeba RPC to a SunOS 4.1.1 kernel

Requirements

If you have SunOS release 4.1.1 or higher you can add Amoeba RPC as a loadable pseudo-device to the kernel while it is running. The kernel must have the VDDRV option installed (which is the default).

Step by step installation

The first thing to do is to create a binary file to link with the SunOS kernel. It is assumed that the configuration tree has been built as described in the previous section.

The FLIP driver for the *sun4m* will be used as the example in the following description. In the configuration directory (for the sun4m this is *`amdir`/conf/unix/sparc.sun/flip-driver*) there is an *Amakefile* that will create the driver and related utilities. In this case it will create one file, *amoeba.sun4m.o* for use on the respective kernel architectures for the sun4. This is done by typing the command

amake -DUNIX_VERSION=sunos4.1.1

Note that if you have SunOS 4.1.3 then use `sunos4.1.3` as the `UNIX_VERSION` argument. The *amake* command will also create programs called *flip_dump* and *flip_stat* which can be used to get status information from the enhanced UNIX kernel. Some other programs, for wide area communication might also appear. Ignore them. Move the file *amoeba.sun4m.o* into a subdirectory called *vldrv* in the top level of the Amoeba distribution tree. If this directory does not exist then create it. Move the programs *flip_stat* and *flip_dump* to the directory *bin.sun4* in the top level of the Amoeba distribution tree.

Now the driver can be installed. First become super-user on the machine where the driver is to be installed. Then put the top level directories *bin.scripts* and *bin.sun4* in your `PATH`. Then type the command

loadflipdriver

This will, if everything works, create in */dev* the entries used to access the network driver. Their names all begin with *flip*.

Loadflipdriver should print

```
vdload
module loaded; id = 1
```

The *module id* will be a number higher than one if other loadable device drivers have been loaded already. Something about a *kid* (kernel id) might also be printed, depending on the level of debugging in the driver. If the script prints

```
flip_stat failed
```

Then something probably went wrong with the driver or its installation.

If everything seems to have worked then the interface should be tested. Make the *.capability* file and perform the basic tests as described in the introduction.

After the driver has been tested successfully it is probably wise to put the *loadflipdriver* call

in the file */etc/rc.local* of the machine so that it is loaded every time the machine is rebooted.

The driver has been tested on machines with as many as three Ethernet interfaces. It will automatically function as a FLIP-gateway when there is more than one network-interface.

5.4.3. Adding Amoeba RPC to an Irix 3.3 Kernel

This procedure has not been tested for many years. It is included only for exemplary purposes.

Introduction

This document describes how to add an Amoeba driver to a Silicon Graphics 4D/25 (also known as a Personal Iris or PI) running Irix 3.3 (or 3.3.1). Probably the same procedure will work for any other system running Irix 3.3, but this has not been tested.

Installing the Amoeba driver under Irix is more difficult than adding it to Ultrix or SunOS, because Irix has no easy way to add new protocols to the system. For that reason, you will have to modify one binary file to arrange that incoming Amoeba packets are handed off to the driver.

Building The Amoeba Driver.

The procedure for building the UNIX configuration for Amoeba is described earlier in this chapter. Build a configuration tree for machine **mipseb** using toolset **irix**, compile that configuration and finally build the driver. The driver is built in *$CONF/mipseb.irix/amoeba-driver*. The command to use is

amake -DUNIX_VERSION=irix3.3

Modifying The Irix Sysgen Files.

Next, you have to modify some files that are used to build your Irix system. This has to be done as super-user. The first thing to do is to modify the Ethernet packet switch to hand Amoeba packets to the Amoeba driver. This is done by taking the Ethernet driver, *ether.o*, from the archive *bsd.a*, renaming it to *am_ether.o*, and using a binary editor to change the reference to *drain_input* to *am_dr_input*. Note that the two names are the same length, so there should not be too much difficulty changing this. One editor that can edit binary files is *emacs*. If you have no binary editor it is not too much work to write a C program to change the file. After changing the name, use *nm* to check that your change worked.

cd /usr/sysgen/boot
ar x bsd.a ether.o
mv ether.o am_ether.o
edit *am_ether.o* and change *drain_input* to *am_dr_input*.
nm am_ether.o

check that there is an external *am_dr_input* and no *drain_input*.

Now, copy the Amoeba driver you built in the first step to the *boot* directory:

cp ..../mipseb.irix/amoeba-driver/amoeba.o /usr/sysgen/boot/amoeba.o

Next, add two files to the *master.d* directory identifying the two new files you have just created. The file */usr/sysgen/master.d/amoeba* will look as follows:

```
*
* amoeba - amoeba protocol pseudo-device
*
*FLAG   PREFIX  SOFT    #DEV    DEPENDENCIES
cosp    am      60      -       bsd,am_ether


$$$
```

The file */usr/sysgen/master.d/am_ether* has the following contents:

```
* amoeba - amoeba protocol pseudo-device
*
*FLAG   PREFIX  SOFT    #DEV    DEPENDENCIES
x       am      60      -       bsd


$$$
```

The number *60* in both files above is the major device number. You might have to change this to another number if you have already added another third-party device to the system that uses this number. The SGI documentation will tell you how to pick another number.

The last file to edit is your system configuration file, normally called */usr/sysgen/system*. Add the following lines to the end to include the Amoeba driver:

```
* Include the next two lines to get Amoeba support:
INCLUDE: am_ether
INCLUDE: amoeba
```

Building And Installing The New Kernel

Now you can build and install your kernel, and reboot the system to try it. It is probably wise to keep your old kernel in a safe place, so you can use it if the new kernel fails to run:

make
cp /unix /unix.old
lboot -u /unix.install
sync
reboot

The next thing to do is to create the device special file for the Amoeba driver, and to try the driver out.

```
/etc/mknod /dev/amoeba c 60 0
chmod 666 /dev/amoeba
dir
```

The number *60* above is the major device number again, the same as used in the previous section.

The kernel is now ready for testing. Proceed according to the instructions in the introduction.

5.4.4. Adding Amoeba RPC to an Ultrix kernel

This procedure has not been tested for many years. It is included only for exemplary purposes.

Requirements

To add Amoeba RPC to your Ultrix kernel you need a binary license for Ultrix, at least release 2.2. It is also possible to add an Amoeba pseudo-device to older versions of Ultrix, but you will need a source license for this. This driver has not been tested with Amoeba 5.0 but the Amoeba 4.0 driver worked using these installation instructions so there is some hope that it might work.

Adding the driver has been tested on a MicroVax 2000 and a MicroVax II under Ultrix 2.2 but there is little reason to believe that it will not work with other machines or intermediate releases of Ultrix.

It is probably a good idea to get some feeling for what happens during kernel configuration, so if you have never configured a kernel before you should try a standard configuration first. See the system administrators manual for details.

Adding the driver to a binary only system

Even if you have a source license it is probably a good idea to try to follow this path first, since it will change the fewest source files. It is probably prudent to store copies of these files away in a safe place, so you will be able to revert to your originals in case you do something disastrous. The following files are going to need changes:

conf/files
vax/conf.c
data/uipc_domain_data.c
data/if_to_proto_data.c
h/socket.h

Setting things up.

We assume here that you have already loaded the Amoeba driver distribution. If not, do so now. A reasonable place to put it is probably */usr/src/sys/amunix*.

Next, copy the files named above to a safe place.

Modifying your system files.

As stated before, some of the files in your Ultrix distribution will need a few changes. We will describe them file by file here.

conf/files

This file contains the names of all sources used to build a unix kernel. Add a line

```
amunix/amoeba.c       optional amoeba Binary
```

to the end. Note that there is no such file, actually, but that does not matter since we are building from binary anyway.

vax/conf.c

This file lists all devices that can be supported by Ultrix. Add the following lines at a convenient point:

```
#ifdef AMOEBA
extern int     amioctl();
#else
#define amioctl nodev
#endif AMOEBA
```

Also, add the following lines to the bottom of the initialization of the `cdevsw` array:

```
nulldev,    nulldev,    nodev,      nodev,      /*NN*/
amioctl,    nodev,      nodev,      0,
seltrue,    nodev,      0,
```

Replace the *NN* by a number one higher than that of the previous entry. (It is 54 when the driver is added to a virgin Ultrix 2.2 distribution). If you are not running 2.2 but some other version you will have to find out what the fields in a `cdevsw` entry are, and what to put in them. *Nulldev* or *nodev* is a reasonable value for pointers to functions, 0 for most other things, probably.

data/uipc_domain_data.c

This file lists all communication domains recognized by the networking code. Add the following lines to the routine *domaininit()*, somewhere before the *for* loop (in virgin 2.2, it will be right after the code with *ADDDOMAIN(dli)* in it):

```
#ifdef AMOEBA
        ADDDOMAIN(amoeba);
#endif
```

data/if_to_proto_data.c

This file contains the initializes a table that will map ethernet protocol numbers to address families and domains. There is a comment saying that ‘‘The DLI entry should be last ...’’. The following lines should be inserted before that comment:

```
#ifdef AMOEBA
#include "../amunix/am_tp.h"
#define ETHER_AMOEBA \
        { ETHERTYPE_AMOEBA,      AF_AMOEBA,     0,              0}
#else AMOEBA
#define ETHER_AMOEBA    IFNULL
#endif AMOEBA
```

Also, the entry *ETHER_AMOEBA* should be added to the initialization of `if_family[]` array, just before *ETHER_DLI*.

h/socket.h

The last file to change is *socket.h*, to define the Amoeba address family number. The actual number used is unimportant. It is only used internally. Before the line defining *AF_MAX* add a line defining *IAF_AMOEBA*. Use the old value of *AF_MAX*, and increment *AF_MAX* by one.

Building the driver.

Building the driver is fairly simple. First, edit the *Makefile* to see whether it is setup to compile for your version of Ultrix. Drivers for various kinds of unixen are made from the same Makefile, so you might have to comment out some lines. There are two blocks where you have to customize something: the definitions at the top and the rule for compiling C sources a little below that.

Now, typing

make

should result in the file *amoeba.o*, that should be copied to */usr/src/sys/BINARY.vax/amoeba.o* , where the Ultrix *make* will find it later on.

Also, the program *flip_stat* should have been built. This program can be used to peek at some of the data structures of the Amoeba driver.

Building a kernel.

First, you have to make a configuration file. For instance, if your old configuration file is called */usr/sys/conf/HOST* you copy that file to */usr/sys/conf/AMHOST*, and edit the new configuration file to add the following two lines:

```
options          AMOEBA
pseudo-device    amoeba
```

Somewhere near the end is probably a good idea.

Next, you make the directory */usr/sys/AMHOST*. Then in the directory */usr/sys/conf* run

/etc/config AMHOST

Now change to */usr/sys/AMHOST* and run the commands

make depend
make

Hopefully, this will build a UNIX kernel with Amoeba support.

Boot the new kernel (be sure to save the old */vmunix* somewhere in case something went wrong), and make the device entry, with

/etc/mknod /dev/amoeba c *NN* **0**

replacing the *NN* by the same number you entered into *conf.c*.

Now, you are ready for testing. Proceed according to the instructions in the introduction.

6 Routine Maintenance

6.1. Backup

There are several ways to do backup under Amoeba. The program *starch*(U) is probably the best program to use since it understands the Amoeba directory system much better than programs like *tar*(U). *Starch* can dump to tape, floppy or disk, and can even send its output over a network to some other operating system (with the Amoeba network protocol built into it) so that it can be dumped to a tape or disk on the remote operating system.

6.2. Garbage Collection

In Amoeba the directory server and the file server are separate. It is therefore possible to create files without ever storing the capability in the directory server. This is also true for other types of objects (including directories). It may also be the case that a program creates a (possibly large) temporary file but then crashes or is stopped before it can register the capability in the directory server. In this case there will be a file which no one can access or destroy taking up valuable disk space. To free the resources consumed by objects which are no longer accessible a *garbage collection* mechanism is used. It works by periodically marking all objects still reachable from the directory graph and then letting the server destroy any objects which were not marked. Under Amoeba, each relevant server keeps a *time-to-live* field for each object it manages. When the object is accessed (or *touched* in Amoeba terms) the field is reset to the maximum value. The periodic marking or *touching* is then followed by a command to all relevant servers to *age* each of their objects by decrementing the *time-to-live* field by 1. If it reaches zero the object is destroyed.

The current system for performing garbage collection is implemented by *om*(A). It traverses the entire accessible directory graph and touches all objects for servers registered for garbage collection. It then sends the STD_AGE command to each registered server. All objects not touched within the aging period of the server will be destroyed. The servers currently managed this way are the Soap Server (see *soap*(A)) and the Bullet Server (see *bullet*(A)). It is very important to run *om* in server mode to prevent the disk filling up with garbage and to ensure that disused directory slots are freed. Garbage can fill up disks quite quickly. In general, Amoeba does not actually destroy objects when using commands such as *rm*(U) but merely deletes the directory entry. This is in case someone else holds the capability for the object (for example a link). It is expected that the garbage collection mechanism will deal with any objects which are no longer accessible.

7 Trouble-shooting

In general, Amoeba is fairly reliable, but occasionally things can go wrong due to a corrupted disk or a power problem. The more trivial problems such as *make sure it is plugged in* will not be discussed here. There are however possibilities for other problems to arise and these are described in the following sections. The section headings indicate the types of problems addressed.

7.1. Installation Failure

Sometimes when booting a Sun 3/60 from tape it will print the message

```
getbyte error, target never set REQ
getbyte error, phase mismatch
invalid status msg = FFFFFFFF
```

Normally it will proceed from here and continue to boot without problem. If it does not, but instead returns to the PROM monitor then the most probable cause is that the tape or the tape drive is a 150M (QIC-150) tape. The Sun 3 PROM does not understand 150M tapes. Only QIC-24 and QIC-11 tapes and tape drives can be used to install Amoeba. Once Amoeba is running 150M tapes and tape drives can be used with the Sun 3.

7.2. Kernel Crashes

From time to time an Amoeba kernel may crash. If this occurs it will probably be due to an assertion failure in the kernel. In the case of Sun and MC68000 kernels it will generally be the case that if the machine is rebooted *without power-cycling the machine*, the command *printbuf*(A) can be used to display the contents of the console just before the machine crashed. This output will include a stack trace and details of the assertion that failed, if any.

In the case of the i80386 ISA bus machines it is often the case that the console buffer information is not available after a reboot. After crashing the i80386 machines normally attempt to reboot, destroying the console record and clearing it from the console screen. Therefore at boot time it is possible to specify the *-noreboot:1* option (see *isaboot*(A)) which will prevent the kernel from rebooting automatically after a crash. This will leave any stack trace and assertion failure information on the console screen for inspection and analysis.

If a Sun 3 crashes with an NMI trap it will print something like

```
NMI trap: memory control register = d2
virtual address = f340004
context         = 3
DVMA cycle      = 0
```

This indicates that a parity error occurred at the address specified. Check that all the memory SIMMs are correctly seated and if so then replace the SIMMs one for one until the problem goes away.

If a Sun 4c panics just after it has been booted then it is probably the case that resetting the machine and rebooting should solve the problem. There may have been interrupts pending from the previously running kernel which were not cleared before rebooting. Sun 4 machines with PROM revisions higher than 2 do not have this problem since they automatically reset when a reboot is requested.

Certain SBUS cards which worked on the Sun 4c may not work with the Sun 4m. If a card does not work with Amoeba, see if SunOS/Solaris can drive it.

If an i80386 ISA bus machine has been running DOS and then Amoeba is booted without a hard reset of the machine, then the Amoeba kernel may crash when the first user process runs. Therefore, after running DOS, press the reset button or power-cycle the machine before booting Amoeba.

7.3. Amoeba — UNIX Communication

When the FLIP driver has been installed in a UNIX kernel it is sometimes the case that communication between the UNIX host and the Amoeba system does not work. The following are common reasons for difficulty:

1. No *.capability* file is present. In this case a *dir*(U) command under UNIX will complain about the absence of the file.
2. An incorrect *.capability* file was installed.
3. The UNIX host and the Amoeba host are not on the same network.
4. There is a bridge or gateway between the UNIX host and the Amoeba host that is filtering out the FLIP packets.

The last three possibilities will cause a *dir*(U) command under UNIX to report ‘‘not found’’. In this case the first step is to ensure that the *.capability* file contains the correct value. If that does not solve the problem then the best way to check where the problem lies is to physically connect the UNIX machine to the Amoeba network and see if communication can be established. If so then it is question of local networking which must be solved by the local network administrator. Of course, if a UNIX host is acting as a gateway between the Amoeba network and the UNIX network then it must also have a FLIP driver installed in it to route the FLIP packets.

8 Manual Pages

Collection of manual pages for servers and utilities under the system administrator’s control.

Name

add_route – add routes to an IP server

Synopsis

```
add_route -g gateway [-d destination [-n netmask ]] [-I IP-cap]
```

Description

Add_route can be used to manually add routes to an IP server. Other ways to add routes to the routing table of an IP server are the *irdpd*(A) daemon and the built-in routing table. Using *irdpd*(A) is probably the best way to install routing information in an IP server.

The only argument that is mandatory is the gateway. It can be a symbolic host name. It specifies an IP gateway to another network.

If a network has more than one gateway *add_route* should be called for each gateway.

Options

–I *IP-cap*
: This option specifies the IP server capability. It can either be the name of a host running an IP server or the complete name of the IP server capability. If the **–I** option is not present then the value of the environment variable `IP_SERVER` is used. If this is not present then the default IP server (as defined in *ampolicy.h*) is used.

–d *destination*
: This option specifies the destination. When omitted 0.0.0.0 is assumed. This means all hosts not on this network. The symbolic name of a host can be used.

–n *netmask*
: This option specifies the netmask. If no **–n** option is given the netmask is calculated from the destination, i.e. 0.0.0.0 for the 0.0.0.0, 255.0.0.0 for a class A network, 255.255.0.0 for a class B network and 255.255.255.0 for a class C network.

Environment Variables

`IP_SERVER` The value of the environment variable overrides the default path for the IP server. The **–I** option overrides this environment variable.

Examples

```
add_route -g 1.2.3.4 -d 5.6.7.8 -I /super/hosts/armada1E/ip/ip
```

and *add_route* will respond with:

```
adding route to 5.6.7.8 with netmask 0.0.0.0 using gateway 1.2.3.4
```

The same command could also be given as:

```
add_route -g 1.2.3.4 -d 5.6.7.8 -I armada1E
```

or

```
add_route -g 1.2.3.4 -d 5.6.7.8 -I /super/hosts/armada1E
```

See Also

ipsvr(A), irdpd(A), pr_routes(A).

(UNIX ONLY)

Name

amdumptree – dump a UNIX directory tree to Amoeba

Synopsis

```
amdumptree [-n] unixdir amoebadir
```

Description

This utility is primarily for use during the installation of Amoeba. It is used to dump a directory tree from UNIX to Amoeba. It creates copies of the files and directories from UNIX under Amoeba if they do not already exist. It replaces any existing version of a file under Amoeba with the new version from UNIX if the two files are different. If the **–n** option is specified then the existing version will be overwritten. Otherwise the old version will be renamed with *.old* added as a suffix to the file name. Existing entries under Amoeba which are not present under UNIX will not be altered. *Amdumptree* follows symbolic links, rather than reproducing the link under Amoeba.

Example

```
amdumptree `amdir`/BinTree.am/super /super
```

will dump the UNIX directory tree *`amdir`/BinTree.am/super* under the Amoeba directory */super*.

See Also

amdir(U), aminstall(A).

Name

aminstall – install Amoeba utilities from a configuration tree

Synopsis

```
aminstall [-m] [-u] amoebaroot confroot srcroot arch toolset [machtype]
```

Description

This command works under Amoeba and on UNIX systems with an Amoeba driver installed. It is used to install the Amoeba executables, consisting of shell scripts and binaries on an Amoeba file system. It expects the standard Amoeba configuration tree and source tree structure. It installs both servers and utilities from the configuration tree specified by `confroot/arch.toolset` and the scripts from `srcroot`*/util/scripts*. They are installed under `amoebaroot`*/unixroot/bin* if they are emulations of POSIX programs and otherwise under `amoebaroot`*/util*. System administration programs are stored in `amoebaroot`*/admin/bin*. If the system bootblocks and related utilities are required for a particular architecture then the optional *machtype* argument should be given.

Options

–m Per default this command assumes that the target directories where the binaries are to be installed already exist. If they do not, then without the **–m** flag they are not created. Instead a warning is printed. With this flag they are created if they do not already exist.

–u This option is only available under UNIX. Instead of installing the executables under Amoeba this option causes the *amoebaroot* option to be interpreted as a UNIX directory on the current host. This is primarily intended for creating distributions.

Example

If the system is installed under the name */amoeba* under UNIX, including the configuration in the directory *conf* then the following command will install the SPARC binaries compiled with the sunpro compiler set. It will also install the bootblock utilities for the system.

```
aminstall /super /amoeba/conf/amoeba /amoeba/src sparc sunpro sun4
```

Note that for the Sun 4m and Sun 4c machines the *machtype* argument is *sun4*.

See Also

amoebatree(A), amuinstall(A).

Name

amoebatree – build a configuration tree for Amoeba

Synopsis

```
amoebatree [-v] [-s subtree]
        conf-root src-root template-root arch toolset

unixtree [-v] [-s subtree]
        conf-root src-root template-root arch toolset
```

Description

Amoebatree and *unixtree* are used to construct the configuration trees for building the binaries of Amoeba. *Amoebatree* creates the tree for the binaries that run under Amoeba and *unixtree* creates the tree for the binaries that run under UNIX. Note that they do not perform the actual build but merely create the hierarchy with the necessary *toolset* and *Amakefiles*. The configuration system has been designed so that versions for different architectures can be easily managed and that cross-compiling is straight-forward.

The configuration is constructed by cloning a template configuration tree which contains the directory structure and necessary *toolset* and *Amakefiles*. If the configuration tree specified does not exist it will be created. If it already exists, then it will be brought up to date. Note that any parts that are no longer in the template tree will not be deleted. If any files in the configuration are different from those in the templates then the program will enquire as to whether it should overwrite the file or not. If you respond by typing *q* then no change will be made and the program will terminate. If you type *d* then it will show the differences. If you type *y* the change will be made. Any other response will be interpreted as ‘‘do not overwrite but continue’’.

The *conf-root* specifies the place where the configuration should be built. If it is not an absolute path name then the output of the *amdir*(U) command will be prepended to it. Otherwise it will be used as is.

The *src-root* specifies the source code from which the system is to be built. If it is not an absolute path name then the output of the *amdir*(U) command will be prepended to it. Otherwise it will be used as is.

The *template-root* specifies where to get the templates for the *Amakefiles*. If it is not an absolute path name then the output of the *amdir*(U) command will be prepended to it. Otherwise it will be used as is. The *Amakefiles* will be parameterized with the *conf-root*, *src-root* and the target architecture for which you are compiling.

Note that if a relative path name is used with the above three arguments, *amdir* is prepended to the path name. If the resultant path name begins with */profile* (as will typically be the case under Amoeba) and the user has the directory */super* then the path will be converted to use */super* since this has more chance of having write permission in the directory tree to be created / updated.

The *arch* argument specifies the target architecture for which you are compiling. Currently valid values are *mc68000*, *i80386*, and *sparc*. Other values will be defined as Amoeba is ported to other architectures. The user programs and libraries are architecture dependent.

Only kernels are also machine dependent.

The *toolset* argument specifies which compiler, loader, archiver, etc. are to be used to build the system. Under Amoeba this is typically *ack(U)*. Under UNIX this will be one of the compilers available with the UNIX system. All the tools will be placed in a directory in the configuration tree and included from there by the *Amakefiles*.

Options

–s subtree This option causes it to create/replace only the subtree specified. At present the *toolset* will always be done in addition to any subtree specified here. It is not possible to give multiple subtrees in one command.

–v The verbose option. This causes the program to print details of everything it is doing. This is should be avoided except for debugging purposes or with the **–s** option since it creates an enormous number of files and directories.

Example

```
amoebatree /amoeba/conf/amoeba /amoeba/src /amoeba/templates \
        i80386 ack
```

will create under the directory */amoeba/conf/amoeba/i80386.ack* a tree structure containing all the *Amakefiles* to build the libraries and binaries for Amoeba. If the command *amdir* produces */amoeba* then

```
amoebatree conf/amoeba src templates i80386 ack
```

is equivalent.

```
unixtree /amoeba/conf/unix /amoeba/src /amoeba/templates i80386 ack
```

will create under the directory */amoeba/conf/unix/i80386.ack* a tree structure containing all the *Amakefiles* to build the libraries and binaries for Amoeba programs that run under UNIX with an Amoeba-driver.

See Also

amake(U), amdir(U), build(A), doctree(A), kerneltree(A).

Name

amuinstall – install UNIX utilities for talking to Amoeba

Synopsis

```
amuinstall conf-root bin-directory
```

Description

This command is used to install the utilities which run under UNIX but which interact with Amoeba. It copies them from the subtree *conf-root/util* to the directory *bin-directory*. It expects the standard Amoeba configuration tree and source tree structure. It only installs executable files and does not install the *tools*. These should be installed separately by the system administrator since they will probably need local customization.

Example

If the system is installed under the name */amoeba* on UNIX, including building the configuration in the directory *conf* then the following command will install the SPARC binaries compiled with the sunpro compiler set for Solaris.

```
amuinstall /amoeba/conf/unix/sparc.sunpro /amoeba/bin.sol
```

See Also

aminstall(A), amoebatree(A).

Name

bdkcomp – compact a Bullet Server's virtual disk to eliminate fragmentation

Synopsis

```
bdkcomp bullet_server
```

Description

Bdkcomp compacts the virtual disk of the Bullet Server specified by the *bullet_server* argument. The primary function of this is to reduce the amount of internal resources required for managing the disk free list, which becomes quite large in the presence of fragmentation. A secondary usage is to try to maximize available disk space when a disk is almost full.

The capability specified by the argument must be the super-capability of the Bullet Server with full rights. This capability is typically found in the kernel directory of the Bullet Server host.

The effect of this command is to move all the files to the beginning of the virtual disk so that all free disk space is collected into a single contiguous chunk. Note that any holes larger than 1 Mbyte will not be compacted. If the disk is large or badly fragmented this process can take several hours if files are being continually created and deleted. For example, it is a good idea to stop the object manager (see *om(A))* so that the compaction is not continually interrupted. Normally it will take about 30 minutes to compact a moderately fragmented disk. During this time the Bullet Server will be extremely sluggish so it should only be run if absolutely necessary and at a time when the system is not heavily loaded (unless you enjoy complaints from users).

No message is returned to the caller unless an error is encountered. Possible causes of errors are:

a) The look up of the Bullet Server capability failed.

b) The Bullet Server capability is not the super capability with full rights.

c) The Bullet Server is not running.

d) The internal free lists of the Bullet Server have been corrupted.

Example

```
bdkcomp /super/hosts/lulu/bullet
```

If all is well this will be silent. It may however take a long time. You can check the degree of compaction using the *bstatus*(A) command.

See Also

bstatus(A), bullet(A), gbullet(A).

Name

bfsck – check bullet file system consistency

Synopsis

```
bfsck [bullet_server]
```

Description

Bfsck runs a file system check on the Bullet Server specified by the *bullet_server* argument. If no argument is given the default Bullet Server, as defined by DEF_BULLETSVR in *ampolicy.h*, is checked. (DEF_BULLETSVR is typically */profile/cap/bulletsvr/default*.) The capability specified by this path name must be the super-capability of the Bullet Server with full rights. This capability is typically found in the kernel directory of the Bullet Server host.

Bfsck goes through the file system and eliminates inconsistencies by deleting any files which offend against the consistency rules. Firstly, if the Bullet Server is compiled with debugging on, the cache consistency is checked. Then the inode table is traversed looking for files which a) have data blocks which lie in or before the inode table, b) have data blocks beyond the size of the disk, or c) consist of a negative number of disk blocks. Any such files are deleted. If any files are deleted then a message is printed on the ‘‘console’’ of the Bullet Server. No message is returned to the caller unless an error is encountered. Possible causes of errors are:

a) The look up of the Bullet Server capability failed.

b) The Bullet Server capability is not the super-capability with full rights.

c) The Bullet Server is not running.

Example

```
bfsck /super/hosts/bullet1/bullet
```

If all is well this will be silent. To check if there were errors in the file system run the command

```
printbuf /super/hosts/bullet1
```

See Also

bstatus(A), bullet(A), gbullet(A), printbuf(A).

Name

boot – keep system services available

Synopsis

```
boot [-dtv] [dir]
```

Description

The *boot* server polls capabilities to find out whether services are available, and starts processes to make them available if they are not. Normally, it is started by the *bootuser*(A) thread in the kernel of the main machine in an Amoeba system. When started by hand, you should pass the processor on which it is supposed to find its virtual disk in *dir*, so that *boot* sets its root directory correctly. Also, it is wise to run the *boot* server on that machine, since it is possible that the *boot* server's configuration depends on this.

The server requires a virtual disk and an operational file server, which is currently reasonable since the Bullet Server is in the kernel. The virtual disk is used to hold the configuration data and to store data that must survive server crashes. The file server holds executables. It does not depend on any directory server, since it needs to be able to start one.

The lay-out of the virtual disk is defined in *h/server/boot/bsvd.h*. To initialize the disk, use *iboot*(A).

Options

–d Set the debugging flag, which produces output for gurus.

–t This makes the *boot* server print the time as part of each messages it prints on the console.

–v Turns on the verbose flag, which makes the *boot* server report more than usual. Normally, the *boot* server is only verbose until it has scanned the services it keeps alive a few times. With this flag it will remain verbose.

Configuration format

The configuration file is a human readable file. It is currently copied to the virtual disk by *iboot*(A). Below is described the syntax, each containing a precise syntax definition, an explanation, and sometimes an example.

The keywords are: down, procsvr, machine, after, bootrate, pollrate, cap, environ, capv, argv, program and pollcap. Furthermore, C-style identifiers and strings, ar-format capabilities (see *ar*(L)) and decimal numbers are recognized. Comments start with a ‘‘#’’ symbol, and extend till the end-of-line.

```
config_file: ( assignment | service | stringenv ) *
```

The configuration file consists of a list of variable assignments, services and a string environment. The string environment is defined below, and used to set the string environment of the *boot* server. If you do not set `TZ` (see *ctime*(L)) correctly, all times reported by the *boot* server are expressed in UTC (GMT). For example,

```
environ { "TZ=:/super/module/time/zoneinfo/localtime" };
```

will set the timezone.

NB. If when the *boot* server starts there is no *soap*(A) server running, then it will not be possible to look up and access the timezone information. This will result in the times being shown in UTC no matter what value is given to the TZ variable.

The variables hold capabilities, and are used to avoid extensive repetition of capabilities and paths in the services. Variables can be assigned to once, and must be assigned to before usage.

```
assignment: 'cap' IDENT '=' reference ';'
reference:  capability [ STRING ]
capability: CAPABILITY | IDENT | '`' IDENT '`'
```

Examples of assignment:

```
cap SLASH = e8:cc:77:15:4a:2a/287(ff)/94:cd:23:6f:c0:a7;
cap HOSTS = SLASH "profile/hosts";
cap MYPOOL = SLASH "profile/pool/i80386";
cap THISMACHINE = `ROOT`;
# Where to dump core:
cap CORE_DIR = SLASH "profile/module/boot/dumps";
```

After the above definitions, you can use the identifier SLASH instead of the ASCII encoded capability (known as a capability literal) at the right of the equals sign. Starting from SLASH, the capabilities *profile/hosts* and *profile/pool/i80386* are looked up, to assign them to HOSTS and MYPOOL respectively. Any leading slashes in the string arguments are stripped before the *name_lookup* (see *name*(L)) is performed, to stop it from using the *boot* server's ROOT capability.

To generate the capability literal for a given capability, you should use *c2a*(U). For example, if you want HOME to point to your home directory, the command

```
c2a /home
```

prints it for you. The value for SLASH in the above example was obtained with

```
c2a /
```

Note that because an *ar*-format (see *ar*(L)) is just as powerful as a normal capability they should be protected. The capabilities in the bootfile may give access to the entire system so you should protect this file.

Variables are really syntactic sugar. They do not cut down the number of directory lookups performed by the server in any way.

The second last assignment in the above examples makes the *boot* server assign the *boot* server's environment capability ROOT to THISMACHINE. This is usually the capability for the machine the *boot* server runs on. This way, any capability in the *boot* server's environment can be found. The capabilities STDIN, STDOUT and STDERR can be modified at run-time by the *boot* server in response to a *boot_setio* request (see below). Of course, the change only takes effect for services that need to be rebooted.

The last assignment is a magic one. The special name `CORE_DIR` is recognized by the *boot* server as the directory to store core dumps. The core dump will have the name of the service. If a file by that name exists already, no dump will be made. Core dumps are produced for programs that get an exception or a negative stun.

```
service: IDENT '{' option* '}' ';'
option: 'down' ';'
      | 'after' serviceIDENT ';'
      | 'bootrate' NUMBER ';'
      | 'pollrate' NUMBER ';'
      | 'machine' reference ';'
      | 'procsvr' reference ';'
      | 'pollcap' reference ';'
      | 'program' command ';'
```

Each service describes a service that should be available, or a program that should be running. Note that throughout this document, the word ‘‘service’’ is used to mean either an instance of the above syntactic definition, or the concrete system service which it keeps alive.

The *down* option means that the *boot* server should not manage the service, e.g., because you are already managing it. A service with this option is not polled or restarted by the *boot* server. However, if there is already a process running for the service, the *boot* server remains the process owner, and will remember any process capability associated with the service.

The *after* option gives the name of a service that has to be running before it is useful to even look at this service. This can be used to only start most services after the directory service, or to start an X-application only after the X-server is running.

The *bootrate* specifies how many milliseconds should pass between successive attempts to boot a program. This is to prevent *boot* from starting a second copy of a program that takes a long time to initialize.

The *pollrate* specifies how often the availability of a service should be tested, in milliseconds.

The *machine* reference points to a processor directory on which the program should run. The reference is resolved just before the program is started, allowing the process server to pick a new capability each time it starts. In fact, all the paths in references are resolved when needed.

Procsvr is slightly different from *machine*. Instead of a processor directory, the capability should point directly to a process server. This can be *proc* in a processor directory, or the *run* server. Because the process server of a machine is found in *proc* of the processor directory, the option

```
procsvr MACH "proc";
```

is equivalent to

```
machine MACH;
```

The *pollcap* points to any object that understands *std_info*(L). This is useful when you cannot poll a process capability, e.g., because no process is associated with a running kernel.

Using *pollcap* when possible has other desirable properties, as discussed below. If no *pollcap* is specified, the process-capability is polled.

The *program* directive specifies how the service should be started.

```
command: '{' reference cmdopt* '}'

cmdopt : stringenv ';'
       | arguments ';'
       | capenv ';'
       | 'stack' NUMBER ';'

stringenv: 'environ' '{' STRING (',' STRING)* '}'
arguments: 'argv' '{' STRING (',' STRING)* '}'
capenv   : 'capv' '{' envcap (',' envcap)* '}'

envcap : IDENT ['=' reference]
```

Commands describe how a program must be started. The reference points to an executable file. The string environment defined with *environ*, is passed as is, so the strings had better each contain one ‘‘=’’ sign. The *argv* is the command line to use. *capv* is the capability environment. If the value of a capability is missing, the capability is inherited from the *boot* server. With *stack* you can set the stack size, in bytes. If not specified, the default for the architecture of the binary will be used.

This is an example service for the soap server:

```
Soap_0 {
    pollcap SLASH;
    machine `ROOT`;
    bootrate 40000;
    program {
         SOAPBINARY;
         stack 50000;
         argv { "soap", "0" };
         capv {
             STDOUT, STDERR,
             TOD = `ROOT` "tod",
             RANDOM = `ROOT` "random",
             BULLET = `ROOT` "bullet",
             SOAP_SUPER = `ROOT` "vdisk:05"
         };
         environ { "LOGNAME=Daemon" };
    };
};
```

Programming Interface

Besides listening to its super capability, the *boot* server listens to the owner capabilities of the processes it starts. The *boot* server supports the standard operations listed below.

A *std_info* on the super capability returns the string ‘‘bootsvr supercap 0x<rr>’’, where <rr> is the rights in two hexadecimal digits. A *std_info* on an owner capability returns a string of the form ‘‘bootsvr: <name>’’, where <name> is the identifier for the service.

The *std_restrict* request works for both the super capability and the owner capabilities, with the behavior defined in *std*(L).

Regrettably, the parser for the bootfile is wired in the server, as opposed to in *iboot*, where it belongs. Any errors detected while parsing the bootfile are reported in the *std_status* message. The rest of the message contains one line per service. The line starts with the name of the service, followed by a ‘‘flags’’ field. The flags are:

```
0   The service will not be restarted if it fails.
1   The service will be restarted if it fails.
X   The service has not yet responded to a poll.
    It may not have been polled at all.
P   The boot server knows the process' capability.
    This is used (possibly in addition to the
    pollcap) to see if the process is still alive.
```

Note that the *boot* server stores process capabilities on disk so this information should survive a restart of the *boot* server.

The flags-field is followed by a ‘‘text’’ field that can hold either an error message or a quoted *std_info* string suggesting that the service is ok. If the service is restarted by the current incarnation of the *boot* server, the time it was started is printed in the `INFO` field as well. Beware that this time is in UTC unless you have set the `TZ` environmental variable of the *boot* server.

An example of a status message is:

```
configuration ok
NAME            FLAGS   INFO
Soap_0          1P      "Run: Daemon: soap -k"    Tue Nov 19 18:49:33 1991
Login_self      1P      "Run: Daemon: login"      Wed Nov 20 07:55:15 1991
Random_cap      1       "random number server"
Tod_cap         1       "TOD server"
Smtp_svr        1P      "SMTP server"             Tue Nov 19 19:45:00 1991
Deliver_svr     1P      "Deliver Server"          Tue Nov 19 19:44:45 1991
Xsvr_chuck      1P      "X server on chuck"
Xlogin_chuck    1P      "Run: Daemon: xlogin "   Wed Nov 20 07:46:35 1991
Xsvr_ast        0X      stdinfo process error: invalid capability Wed Nov
20 07:52:17 1991
Xlogin_ast      1X      After Xsvr_ast
```

Note in the last line that the ‘‘After Xsvr_ast’’ means that ‘‘Xlogin_ast’’ is waiting for ‘‘Xsvr_ast’’ before any attempt will be made to start it. The ‘X’ flag is on because the

service has not been polled.

There is also a set of *boot* server specific requests:

boot_reinit	reinitializes the *boot* server.
boot_shutdown	gracefully terminates the server.
boot_setio	sets the servers I/O capabilities.
boot_conf	tells the server (not) to deal with a service.
boot_ctl	tells the server to be (not) verbose.

Their stubs are not in the library. They are documented in *bootutil*(A).

Pollcap Entries Are Good.

This chapter explains some the internals of the *boot* server (especially initialization), and why its title is a true statement.

On initialization, the *boot* server repeatedly executes *boot_shutdown* until it returns RPC_NOTFOUND (server not found), to make it easy to start a new, compatible version of the server without shutting down all of your system. This strategy does not prevent two *boot* servers from getting in each others' way after a network partitioning, but it is hard to see why anybody would setup a system so that this can actually occur. The *boot* server does not reply to the *boot_shutdown* request before it has saved its internal state, and all server threads have exited. At this point, during which the new incarnation of the server is finding out that it cannot reach old incarnations any more, the *boot* server is deaf to all requests.

Then it starts with an empty set of services to maintain. It executes a *boot_reinit*, which involves reading the *bootfile* on the virtual disk. The super-capability is also found this way. It is not reasonable to expect the *boot* server to do a *name_lookup* for this, because the directory server may be down. Conceptually, old services that are not in the new set of services are killed if possible. This is a no-op for the first reinit action, since the set of services is initially empty. The *boot* server kills processes by sending them a stun code of −9. It may also decide to stun a process when it reboots a service, e.g., after the *pollcap* stopped responding. Then the server reads data − owner capabilities, process capabilities etc., saved by previous incarnations − from the virtual disk. If this initialization fails, the server exits. Otherwise it starts serving.

After the first reinitialization, the *boot* server never panics.

Note that being deaf is a potential source of problems with the *boot* server. This is exactly the reason why the *boot_shutdown* request was invented. Using this, the operator can essentially *minimize* the time during which the server is deaf.

Any *ps_checkpoint* or *pro_swapproc* request made in this time interval can get lost, since its timeout is probably in the same order of magnitude as the one the *boot* server uses to find out there is no other *boot* server around. Losing the *ps_checkpoints* is not so much of a problem: the *boot* server will notice that the process has gone anyway, and start a new incarnation. (Note that the *boot* server does not consider exit statuses to be worthwhile.). At worst, you would lose a core dump.

Losing the *pro_swapproc*, however, is definitely a problem for processes polled without the *pollcap* option described above. The new incarnation of the *boot* server will assume that the last-known process capability is to be polled, instead of the one published in the missed

pro_swapproc request. The new process-capability will not be known by the server, so it will decide to restart it eventually, since the *boot* server strives for availability - not reliability.

See Also

ar(L), bootutil(A), c2a(U), iboot(A).

Name

bootuser – start first user process

Synopsis

Built into the kernel.

Description

Bootuser is an optional kernel thread that starts the first user process; typically the boot server (see *boot*(A)). It is usually installed on the file server machine so that when the system is cold-started the first user process can be started. It scans the virtual disks found in the processor directory in a random order, until it finds one with the magic string ‘‘–boot–’’ in block zero. This disk must be written using *iboot*(A), and contains the capability for the executable that is to be started.

The process is owned by *bootuser*. It responds to *std_info* requests with the string ‘‘bootuser thread’’. If a checkpoint is received, *bootuser* will print this on console, but the process is never restarted. Other requests are not supported.

Environment

The only argument for the process is hard-wired in *bootuser*. The string environment of the process is empty. The capabilities for the process’ environment are looked up in the processor directory:

ROOT	points to the processor directory
WORK	idem
STDIN	points to "tty:00"
STDOUT	idem
STDERR	idem
TTY	idem
TOD	points to "tod"
RANDOM	points to "random"

See Also

boot(A), iboot(A).

Name

bootutil – boot server control

Synopsis

```
boot_ctl scap option
boot_conf scap servicename on
boot_reinit scap
boot_setio scap tty-cap
boot_shutdown scap
```

Description

These utilities provide an interface to the boot server (see *boot*(A)). They all operate on its super-capability, and require the administrative right, which has the same value as the Bullet Server's `BS_RGT_ADMIN` right.

Boot_ctl is used to toggle several server-internal flags, such as the level of verbosity, in the same style as *kstat*(U). Use

`boot_ctl` *scap* `?`

to get a list of possible options, where *scap* is the server capability of the *boot* server. *Boot_ctl* reports an ''invalid argument'' if the option is not supported by the boot server.

Boot_conf tells the boot server (not) to bother about the service *servicename*. The argument *on* must be 0 to stop the boot server from managing the service or 1 to resume. The server reports an invalid argument if *on* has any other value. This is really a temporary way of (re)setting the ''down'' option described in *boot*(A): the down flag will be reset to its original value by the next *boot_reinit*, or when the *boot* server is started.

Boot_reinit causes the boot server listening to *scap* to reinitialize. This involves re-reading the configuration file, stunning processes associated with services that no longer exist, and undoing any effect of a *boot_conf* operation.

Boot_shutdown tells the boot server to exit. It returns `STD_NOTNOW` if the server is already in the process of shutting down. The boot server does this itself on startup, in an attempt to stop any other boot server that listens to the same port.

Boot_setio tells the boot server to use *tty-cap* for printing messages. It returns ''invalid argument'' if the capability belongs to a server that is currently unreachable, or if it does not support the *fswrite* command (see *fs*(L)).

Diagnostics

These utilities may report standard errors as described in *std*(L).

Warnings

The diagnostics and output are as horrible as one can expect from automatically generated programs.

See Also

boot(A), iboot(A).

Name

bstatus – print the state of a Bullet file server

Synopsis

```
bstatus [bullet_server]
```

Description

Bstatus prints the current usage statistics for the Bullet Server running on the machine specified by the capability *bullet_server*. If the name *bullet_server* is not an absolute path name it is looked up in the directory */profile/cap/bulletsvr*. If no name is given it looks up the default Bullet Server (that is, */profile/cap/bulletsvr/default*). The server capability must be the super-capability of the Bullet Server with the read right. This capability is looked up in the kernel directory of the specified host.

If there are no errors, *bstatus* returns a table of statistics from the Bullet Server. These statistics include the disk block, inode and cache usage. On error it prints a short message indicating the problem. Possible errors include:
a) the look up of the Bullet Server capability failed; b) the Bullet Server capability is not the super-capability with full rights; c) the Bullet Server has crashed.

Example

```
bstatus moggy
```

will print a table of the following form for the Bullet Server on the host *moggy*:

```
                  In Use      Free      Holes
Inodes             21894     97273        N/A
Disk Blocks       327126    268470        39
Cache Memory     3438592   5122048         4

Committed Files          1150976 bytes cache,   442100 bytes fragmentation
Uncommitted Files              0 bytes cache,        0 bytes fragmentation

Free Cache Slots              10
Free Hole Structs           3491

cache hits         75967  cache misses       206198  compactions         14
read misses        23587  disk reads          23589  disk writes     167626
timed out files        0  aged files              0  std_age              7
std_copy(local)        0  std_copy(remote)      101  std_destroy      10568
std_info             626  std_restrict           27  std_touch       201394
b_sync                 0  b_status                1  b_fsck               0
b_size             10845  b_read              37269  b_create          3925
b_modify           21767  b_insert                0  b_delete             0
b_disk_compact         0  moved files             0
```

The first information printed relates to resource usage. It tells how much of each resource is in use, how much is free and the number of regions of unused resource (holes), which gives

an indication of the state of fragmentation. In particular, for the two categories of files (committed and uncommitted) it tells the total number of bytes of cache memory in use and the amount of that usage which is internal fragmentation.

The ‘‘Free Hole Structs’’ tells how many unused slots there are in the free list manager for holes. The rest of the statistics tell how many times each kind of operation accepted by the Bullet Server has occurred and give an indication of the cache performance. Only a few need clarification:

Cache hits and *cache misses* refer to the number of times an inode was referenced by a command and was found or not found in the cache. Note that all inodes are all kept in core, but the cache can contain an inode without the corresponding file being read in. (For example, to destroy a file it is necessary to cache the inode temporarily but there is no need to read the file into the cache.)

Compactions refers to the number of times the Bullet Server attempted to compress the cache to eliminate fragmentation. It does this when it is either idle for a long period or if it is desperately short of cache memory.

Read misses refers to the number of times the *b_read* command was forced to read a file from disk.

Disk reads refers to the number of calls to do disk I/O that the Bullet Server has done.

Timed out files refers to the number of uncommitted files that were destroyed because they were not modified or committed within the timeout period.

Aged files refers to the number of files destroyed by the std_age command.

Moved files refers to the number of files that have been moved from one place on the disk to another by the most recent disk compaction. If the disk compaction is still running it reflects the number moved so far.

See Also

bfsck(A), printbuf(A), std_age(A) std_status(L), std_status(U).

Name

bsync – ensure that all committed Bullet files are written to disk

Synopsis

```
bsync [bullet_server]
```

Description

Bsync ensures that all the committed Bullet files on the Bullet Server specified by the *bullet_server* argument are written to disk. (This happens regularly in the *sweeper* of the Bullet Server but is also useful if the server must be shutdown.) If no argument is given, the default Bullet Server, as defined by DEF_BULLETSVR in *ampolicy.h*, is flushed. (DEF_BULLETSVR is typically */profile/cap/bulletsvr/default*.) The capability specified by this path name must be the super-capability of the Bullet Server. No rights are required to do this.

Example

```
bsync /super/cap/bulletsvr/bullet3
```

This will flush all committed files to disk that were not already there.

See Also

bstatus(A), bullet(A), gbullet(A), printbuf(A).

Name

build – build an Amoeba binary configuration

Synopsis

```
build [dirname]
buildlibs [dirname]
buildX [-c] confdir srcdir arch toolset
buildmmdf confdir srcdir arch toolset
```

Description

Creating a set of Amoeba binaries for a particular architecture is done in two steps. The first is to make a configuration tree using *amoebatree*(A) or *unixtree*. This creates a directory tree full of *Amakefiles*. The second step is to recursively descend the tree and execute *amake*(U) on each *Amakefile* to create all the binaries. That is the function of the *build* command. Since third-party software (such as X windows) needs the Amoeba libraries, if any third-party software is required it can be built after this step. Some of the third-party software is large, complex and difficult to configure so scripts are provided to build commonly used packages.

build, buildlibs

The *buildlibs* and *build* commands take an optional argument. This must be the name of a directory. If no argument is given they use the current directory. Beginning at this directory *build* makes a recursive descent of the entire directory tree. *Buildlibs* only descends the subdirectory *lib*. Each time they encounter a file called *Amakefile* they execute *amake*(U).

Note that they check to see if the starting directory looks like a configuration tree. That is, it must have a directory called *lib* in it or they will refuse to go to work. Therefore, *build* cannot be used to remake a partial tree. Its function is to bring the entire configuration up to date.

Note that *build* will not make the kernels. It does not descend into the top-level directory *kernel*. Kernel configuration trees are made with *kerneltree*(A). *Makeconf*(A) is the only program at present which automatically builds kernels.

buildX, buildmmdf

BuildX and *buildmmdf* are used to build X windows and the MMDF II electronic mail system respectively. Before they can be run it is necessary to have run *buildlibs* or *build*. Then the appropriate build command may be executed. Note that *amake* is not used to build these systems. The standard *make*(U) and *imake* configuration tools provided with these packages are used with minor changes to provide binaries suitable for Amoeba.

The *buildX* and *buildmmdf* commands take the same arguments (except that *buildX* takes an extra optional **–c** option as described below). The first argument is the place where the configuration is to be constructed. This is typically *`amdir`/conf/amoeba*. The next argument is the directory where the source directories (e.g., *mmdf* and *mmdf.am* for *buildmmdf*) can be found. This is typically *`amdir`/src*. The third argument is the

architecture for which the binaries are to be built. For example, *mc68000*, *sparc* or *i80386*. The fourth argument is the toolset to be used to compile the software. This is typically *sunpro*, *gnu-2* or *ack*.

BuildX this takes a long time to complete (2-3 hours on a SPARCstation) and uses a lot of disk space. If the *contrib*uted X software is required then *buildX* should be called with the **–c** option.

Diagnostics

The following error messages may be produced by *build* or *buildlibs*.

`: directory ‘`*dirname*`’ not found!`

This is printed if it could not change directory to the directory *dirname* specified on the command line.

```
: doesn’t look like an amoeba configuration tree
bye bye
```

will be printed if the directory that *build* is told to start in is not a proper configuration directory.

Environment Variables

The default command executed by the *build* and *buildlibs* commands is *amake*(U). This can be overridden with the MAKE environment variable.

Examples

```
amoebatree `amdir`/conf/amoeba `amdir`/src `amdir`/templates sparc sunpro
cd `amdir`/conf/amoeba
build sparc.sunpro > build.m68k.out 2>&1
```

will cause the building of the MC680x0 libraries and binaries for Amoeba.

Note that the program *chkbuild*(A) can be used to examine the output of *build*. It strips away almost everything except the error messages.

An example of *buildX* where the contrib software is not made:

```
buildX `amdir`/conf/amoeba `amdir`/src sparc sunpro > buildX.out 2>&1
```

See Also

amake(U), amdir(U), amoebatree(A), chkbuild(A), doctree(A), kerneltree(A), make(U).

Name

bullet – a high performance immutable file server

Synopsis

```
bullet [-c capability] [-i numinodes] [-m memsize]
                  [-t nthread] [vdiskname]
```

or

```
built into the kernel (no options)
```

Unix Only:

```
ubullet [-c filename] [-i numinodes] [-m memsize] [vdiskname]
```

Description

The Bullet Server implements a high speed immutable file service under Amoeba. High performance is achieved using a large buffer cache to reduce secondary storage access and by taking advantage of the fact that files are immutable to store them contiguously, both in the cache and on secondary storage. The Bullet Server uses the virtual disk server (see *vdisk*(A)) to read and write the disk.

The Bullet Server can be run either as a user mode program or in the kernel of the machine with the virtual disk to be used by the server. When run in the kernel, the information needed to start the server is deduced dynamically. When run as a user program, the information can be given as command line options. For more details see the section *User Mode Bullet Server* below. Functionally the user mode and kernel mode Bullet Servers are identical. They differ primarily in the manner and location of starting them.

It is also possible to run the Bullet Server under UNIX using the program *ubullet*. However the functionality is severely restricted. The performance is also relatively poor. For more details see the section *Unix Bullet Server* below.

Bullet files are identified by capability only. A directory server is used to implement a mapping between ASCII file names and capabilities.

There are three primary operations that can be performed using the Bullet Server: *Create file*, *Read file*, and *Destroy file*. The complete set of operations are described in the *Programming Interface Summary* below.

A file may be created incrementally in the Bullet Server's buffer cache but it remains *uncommitted* until the file is complete. Uncommitted files can be either modified or destroyed. Modification is done using the Bullet Server's editing operations: *b_insert*, *b_modify* and *b_delete*. Along with *std_destroy*, these are the only operations permitted on uncommitted files. For all other purposes they do not exist. Once *committed*, files are immutable; they can only be read or destroyed. The *commit* operation is atomic. When committing a file it is possible to specify that you wish to block until the file has been written to disk, or return immediately and let the file be written to disk later. The latter always occurs within `BS_SWEEP_TIME` milliseconds (see *bullet.h*). This is typically set to one minute.

Because it is possible for the capability for a file to be lost, there is a system of garbage collection to throw away files that have not been *touched* within a certain time period. A file

is *touched* by being accessed or by using the special command *std_touch* described below. More details of garbage collection are given in the *Administration* section below.

It is possible that a client dies while creating a file, leaving an uncommitted file in the buffer cache. To prevent malicious or accidental filling of the buffer cache with files that will never be committed there is a short time out on uncommitted files. If no modification command is received within `BS_CACHE_TIMEOUT` seconds (typically set to ten minutes) the file is destroyed.

The User Mode Bullet Server

The performance of the user mode Bullet Server is marginally slower than that of the kernel mode version but the difference is negligible for most purposes. The primary difference between the user and kernel mode servers is the manner of starting them. The kernel mode version is started when the kernel is booted and determines which disk and how much memory it should use on the basis of the resources that are present. The user mode version must be told which virtual disk to use. This can be done with a path name for the virtual disk on the command line (for example, */super/hosts/foo/vdisk:09*) or by setting the string environment variable `BULLET_VDISK` to be the name of the virtual disk. The user mode server need not run on the same host as the virtual disk server that it uses (as is the case with the kernel mode server), but the best performance is obtained that way.

Options

–c *capability*
: This option tells the server where to publish its capability. The default is */super/cap/ubulletsvr/default*.

–i *numinodes*
: This tells the server to load only the first *numinodes* inodes into the cache. Any files whose inode number is higher than *numinodes* will not be accessible. The effect of this is to reduce the amount of memory taken up by the in-core inode table. This is only intended for use in the coldstart process. It should not be used for normal operation. Note that when this option is used a small warning will be printed before the initialization notice telling the number of inodes that will be cached.

–m *memsize*
: This option tells the server how much memory to allocate for its buffer cache. The default is two megabytes.

–t *nthread*
: This option sets the number of threads accepting requests from clients to be *nthread*. The default is ten (10).

The UNIX Bullet Server

It is possible to run the Bullet Server on a UNIX system which has the Amoeba network driver installed. Note that since Amoeba programs running under UNIX are not multithreaded there can be no sweeper task. This means that files committed without safety will not be written to disk. In addition the performance is poor due to the poor network performance and the fact that there is only one thread to service client requests. However,

sometimes it is still convenient to run the Bullet Server under UNIX if no Amoeba machine is available with sufficient memory for the task. The UNIX Bullet Server will still communicate with the virtual disk system in the usual way and so a machine with a disk and virtual disk server is required unless the virtual disk server also runs under UNIX (see *vdisk*(A)).

As with the user mode server under Amoeba, it is necessary to provide the capability for the virtual disk that the server is to use. This can be done with either the optional command line argument or the string environment variable `BULLET_VDISK`. This can refer to a soap capability or the name of a UNIX file which contains the required capability.

Options

–c *filename*
: This option tells the server where to publish its capability. In this case it is a UNIX file name in which the capability for the server is stored. The default file name is *bullcap*. (Note that this is relative to the current directory!) The routine *host_lookup* is capable of accessing this information under UNIX (see *host_lookup*(L)).

–i *numinodes*
: This tells the server to load only the first *numinodes* inodes into the cache. Any files whose inode number is higher than *numinodes* will not be accessible. The effect of this is to reduce the amount of memory taken up by the in-core inode table. This is only intended for use in the coldstart process. It should not be used for normal operation. Note that when this option is used a small warning will be printed before the initialization notice telling the number of inodes that will be cached.

–m *memsize*
: This option tells the server how much memory to allocate for its buffer cache. The default is two megabytes.

Programming Interface Summary

The programmers' interface consists of commands particular to the Bullet Server (whose names begin with *b_*) and the standard server commands (whose names begin with *std_*). The commands are divided into two categories: administrative and user. A summary of the commands is presented in the following two tables, the first listing the user commands and the second the administrative commands. The interface is described in detail in the library section of the manual (see *bullet*(L) and *std*(L)).

User Functions			
Function Name	Required Rights	Error Conditions	Summary
std_copy	BS_RGT_CREATE in servercap BS_RGT_READ in filecap	RPC error STD_CAPBAD STD_DENIED STD_NOSPACE	Replicate a committed Bullet file
std_destroy	BS_RGT_DESTROY	RPC error STD_CAPBAD STD_DENIED	Destroy a Bullet file
std_info	BS_RGT_READ	RPC error STD_CAPBAD STD_DENIED	Print standard information for a Bullet object
std_restrict		RPC error STD_CAPBAD	Produce a Bullet capability with restricted rights
b_read	BS_RGT_READ	RPC error STD_CAPBAD STD_DENIED STD_ARGBAD STD_NOMEM	Read a file
b_size	BS_RGT_READ	RPC error STD_CAPBAD STD_DENIED	Get the size of a file
b_create	BS_RGT_CREATE BS_RGT_READ in compare cap	RPC error STD_CAPBAD STD_DENIED STD_NOSPACE	Create a file, optionally committing it
b_modify	BS_RGT_READ BS_RGT_MODIFY	RPC error STD_CAPBAD STD_DENIED STD_ARGBAD STD_NOSPACE	Overwrite part of the data of an uncommitted file
b_insert	BS_RGT_READ BS_RGT_MODIFY	RPC error STD_CAPBAD STD_DENIED STD_ARGBAD STD_NOSPACE	Insert data into an uncommitted file
b_delete	BS_RGT_READ BS_RGT_MODIFY	RPC error STD_CAPBAD STD_DENIED STD_ARGBAD STD_NOSPACE	Delete data from an uncommitted file

Administrative Functions			
Function Name	Required Rights	Error Conditions	Summary
std_age	BS_RGT_ALL	RPC error STD_CAPBAD STD_DENIED	Start a garbage collection pass
std_status	BS_RGT_READ	RPC error STD_CAPBAD STD_DENIED	Print Bullet Server statistics
std_touch		RPC error STD_CAPBAD	Reset a Bullet file's time to live
std_ntouch		RPC error STD_CAPBAD	Reset a set of Bullet file's time to live
b_disk_compact	BS_RGT_ALL	RPC error STD_CAPBAD STD_DENIED STD_NOTNOW	Eliminate disk fragmentation
b_fsck	BS_RGT_ALL	RPC error STD_CAPBAD STD_DENIED	Do file system check
b_sync	none	RPC error STD_CAPBAD	flush committed files to disk

Administration

In general there is very little administration required for a Bullet Server once it is running. Installation is non-trivial and is described first.

Installation

At present the Bullet Server runs as a kernel thread. The references to a Bullet Server kernel should be taken to mean a kernel that automatically starts a Bullet Server, either as a kernel thread or as a user-mode process.

To start a Bullet Server you need an Amoeba system with a disk and a large amount of core memory. At least 2 Mbytes is required and 12 Mbytes or more gives good performance. The first step is to label the disk (see *disklabel*(A)). The next step is to create an empty file system on a virtual disk. This is done using the *mkfs*(A) command. It requires an Amoeba kernel running the virtual disk server. You can boot a standard Bullet Server kernel at this point since the Bullet Server will print a warning message and then go into a quiescent state or exit if it finds no Bullet file system on any of the virtual disks. Note that a virtual disk may comprise more than one physical partition and may even span several physical disks (see *vdisk*(A) for more details). If the initial *mkfs* is not satisfactory a subsequent one may be performed on the same virtual disk without problems. If you place the file system on the wrong virtual disk you should run *killfs*(A) on the unwanted Bullet file system since the Bullet Server can only run on one virtual disk and the kernel version takes the first virtual disk with a valid Bullet file system that it finds on the disk system.

Once the *mkfs* is complete, an Amoeba kernel containing a Bullet Server should be

(re)booted.

When it comes up it should print a message of the form:

```
BULLET SERVER INITIALIZATION

Buffer cache size = 0xA0A000 bytes
Largest possible file size = 0x911000 bytes
63743 inodes free, 0 inodes used
317638 disk blocks free, 997 disk blocks used
```

Note that the *Largest possible file size* reflects the space available in the cache for storing files and not necessarily the largest file that it is possible to create, which is almost certainly smaller. Note that even though no files have been created, disk blocks are being used for inodes.

Once it is running and the directory server is running the system administrator should copy the Bullet Server's super capability to the directory */super/cap/bulletsvr/*host_name. Typically this directory entry should have column masks `0xFF:0x0F:0x0F` (see *chm*(U)). Also ensure that the Bullet Server chosen as the default is installed under the name `DEF_BULLETSVR` as defined in *ampolicy.h* (this is typically */profile/cap/bulletsvr/default* but should be installed via the pathway */super/cap/bulletsvr/default*) and should have the same column masks as the other Bullet Server capabilities.

See elsewhere in the *System Administration Guide* for further information about initial installation of Amoeba.

Day to day running

Each time the Bullet Server is started it begins with a file system consistency check. If it finds bad files it throws them away and prints a warning on the console. Except in the case of overlapping files this is always the correct solution. At some stage it will become more friendly and possibly allow interactive maintenance of a damaged file system when there are overlapping files.

The *bstatus*(A) command can be used to examine the current state of the file server. It reports usage of resources and commands. It can be useful for determining usage patterns for tuning the Bullet Server. It also reports the disk block and inode usage.

It is possible for the Bullet Server's cache to become full, primarily due to uncommitted files. Committed files are deleted from the cache on a least-recently-used basis if the server runs short of cache space. Cache compaction may also be resorted to if deleting committed files does not provide enough space. If this also fails to yield sufficient space then the operation will fail. If there are too many occurrences of this then it is probably necessary to increase the main memory capacity of the file server host. Restarting the Bullet Server may also help with the problem but will trash any uncommitted files in the cache.

Cache slots may also be in short supply. It will attempt to throw out the least recently used file which is committed. As a last resort it may even toss out the least recently modified uncommitted file, but it will print a warning on the console if it does this. If this warning is ever seen then it is almost certainly worth reducing the `AVERAGE_FILE_SIZE` (a tunable constant in *bullet.h*) to increase the number of cache slots available.

It is possible that after some time your Bullet Server virtual disk may fill up. There are

several possibilities for dealing with this:

i) Run the *bdkcomp*(A) command. This will eliminate disk fragmentation. It is really a stop-gap solution.

ii) It is possible to add a second physical disk to the machine and (using the *disklabel*(A) program) extend the virtual disk of the Bullet Server to use partitions on the new disk. However it will be necessary to edit the Bullet Server's superblock so that it sees that its virtual disk has grown larger.

    Buying a larger disk and transferring everything to the new disk is also possible.

iii) Get a second Bullet Server machine.

If for some reason it is desired to shutdown a Bullet Server the command *bsync*(A) will cause any uncommitted files that are not yet on disk to be written to disk.

Garbage Collection

To prevent the disk being filled up with files for which no one has the capability, the *std_age* command causes the life-time of a file to be decremented. If the life-time reaches zero then the file is deleted. The *std_touch* command or any access to the file resets the life-time to the maximum value of `MAX_LIFETIME` (which is typically 24 - see *inode.h*). The rate at which *std_age* is called is determined by the *om*(A) server which first does an *std_touch* command on all the objects in the directory tree and then sends the *std_age* command to all the servers registered with it. It is important that the rate at which this is done be set to a reasonable value by the system administrator. Otherwise the disk will fill up with junk if the rate is too slow or the files not registered with the directory server may be garbage collected against the wishes of their owners.

If it is clear that the disk is full of junk then one way to recover the lost disk space is to execute several *std_age*(A) commands in succession (but always fewer than `MAX_LIFETIME`) and then run *om* to do one or two rounds of touching and aging to clear away unreachable files. The touching of files before the last aging is essential. Otherwise all the files may de deleted!

Tuning your Bullet Server

There are some tunable constants for the Bullet Server which can be found in the include file *bullet.h*. They come preset to values which are considered reasonable but it may be necessary to adjust them for your site.

`BS_SWEEP_TIME` is the time in milliseconds between each sweep of the *bullet_sweeper* thread in the Bullet Server. This thread has three functions: The first is to decrement the uncommitted file timeout for any uncommitted files. If the timeout for a file reaches zero the file is deleted. The second function is to write to disk any files which have been committed but not yet written to disk. The third function is to determine if the Bullet Server has been idle for a long period and if so to attempt compaction of the buffer cache. Compacting the buffer cache can be an expensive operation and is thus only attempted in idle periods.

This timeout should not be made too short or the Bullet Server will spend most of the time in the sweeper. The default is 60000 (1 minute).

`BS_LOAD_BASE` is the number of sweep times over which the Bullet Server's load is

averaged to determine idleness. When idle the Bullet Server will perform cache compaction. The default is 2 minutes.

BS_CACHE_TIMEOUT is the uncommitted file timeout. That is, the longest period that an uncommitted file will continue to exist without a modify operation being performed on it. It is a multiple of the BS_SWEEP_TIME – so the timeout will be (BS_CACHE_TIMEOUT * BS_SWEEP_TIME) milliseconds.

If this timeout is made too short then people will not be able to create files. If the timeout is too long then the Bullet Server's cache can be become congested. The default is 10, which in combination with the default sweep time gives a ten minute timeout.

BS_REQBUFSZ is the size of the Bullet Server's request buffer size. This reflects the amount of throughput that can be achieved using the Bullet Server.

This should be made as large as practically possible, but bear in mind that the Bullet Server allocates buffers of this size from the buffer cache. The default is tuned to the Ethernet and limited by the current maximum transaction size to 29768 bytes.

When the Bullet Server starts it tries to allocate the largest buffer cache possible from the available memory. However, because the kernel may need to allocate memory and because it may be desirable to run user processes on the same host as the Bullet Server it is necessary to leave some memory in reserve. BS_MEM_RESERVE bytes are subtracted from the size of the largest available piece of memory before it allocates its buffer cache.

It is important to bear in mind that the directory server is normally started on the same machine as the Bullet Server since it too needs access to a disk. Therefore bear its memory requirements in mind when setting this figure. The default is tuned to the Soap Server and is set at 0x280000 bytes. It can be defined in the *Amakefile* as some other value and will override the value in *bullet.h*.

AVERAGE_FILE_SIZE influences the number of inodes created in the file system by *mkfs*(A) and the number of cache slots used in the Bullet Server. The cache slots take up space in cache memory and reduce the space available for data. The larger the average file size, the fewer the inodes and cache slots. The default setting is 16K bytes. This is based on experience with the Bullet Server. For specialized applications such as data bases where memory usage is important this constant should be tuned to the application.

BS_MIN_CACHE_SLOTS sets a lower limit on the number of cache slots the server can have. It is currently set at 1024. The system can function with less than this but performance will be poor.

BS_MIN_MEMORY sets the minimum amount of cache memory which a Bullet Server insists on having before it will run. The default is 3 Mbytes. Anything less than 8 Mbytes will cause heavy thrashing and should be avoided. Memory is cheap, after all.

See Also

bdkcomp(A), bfsck(A), bsize(U), bstatus(A), bsync(A), disklabel(A), fromb(U), killfs(A), mkfs(A), om(A), printbuf(A), soap(A), std(L), std_age(A), std_copy(U), std_destroy(U), std_info(U), std_touch(U), Tbullet(T), tob(U), vdisk(A).

Name

chbul – change a Soap Server's Bullet Server

Synopsis

```
chbul soap-cap bullet-cap bullet-num
```

Description

Chbul tells the *soap* server whose private capability is *soap-cap*, to change its Bullet Server to to the Bullet Server whose capability is *bullet-cap.* Each Soap Server has two Bullet Servers which it uses. The parameter *bullet-num* can be either 0 or 1 and selects which of the two Bullet Servers is replaced.

This change will eventually be propagated by the Soap Server to its duplicate (if it has one) through gradual updating of all directories to use the new Bullet Server. NB. For a large system this can take a few hours since it uses lazy replication.

Examples

```
chbul /super/cap/soapsvr/soap0 /super/cap/bulletsvr/bullet3 0
```

will change the 0 Bullet Server of *soap* server 0 and its partner to be bullet server *bullet3*.

See Also

soap(A).

Name

chkbuild – inspect the error messages that result from a build of Amoeba

Synopsis

```
chkbuild build.out
```

Description

When a *build*(A) of part or all of Amoeba has been made and all the output from the build has been saved it is important to be able to find any error messages that may have been produced during the build. Typically there is a lot of irrelevant information. *Chkbuild* filters out nearly all the ‘‘noise’’ from the *build* output and shows the error messages. It pipes its output through a pager.

Environment Variables

PAGER – this string environment variable selects the pager to be used. The default is *yap*(U).

Example

```
cd /super/module/amoeba/conf/amoeba
build i80386.ack > build.i80386.ack 2>&1
chkbuild build.i80386.ack
```

See Also

build(A).

Name

deluser – delete the home directory and other information about a user

Synopsis

```
deluser username
```

Description

Deluser deletes the home directory structure of the specified user. It does this in a very simple but safe fashion. It simply removes the link to */super/users/*`username`. and the any links in */super/group/**. *Deluser* prints the name of the user to be deleted, immediately before deleting the home directory.

It is essential that the object manager (see *om*(A)) be in operation to garbage collect of all the resources contained in the deleted directory graph. If there are other links to objects in that directory subgraph then the garbage collection will not remove those objects which are still accessible.

Diagnostics

If the specified *username* does not exist then a warning will be printed.

If the specified *username* is the same as the person running the *deluser* command then a error message will be printed and the command will terminate.

If the object manager is not running or has not been installed then an appropriate warning message will be printed after the directory has been deleted.

Example

```
deluser boris
```

will delete the home directory of the user with the login name *boris*.

See Also

newuser(A), om(A).

Name

di – disk layout of a virtual disk

Synopsis

```
di disk_cap
```

Description

The utility *di* shows the disk layout of the virtual disk specified by *disk_cap*. *Di* outputs the disk partition table and when Amoeba partitions are available, *di* outputs the Amoeba sub-partitions.

The disk server used must be able to access the complete disk (e.g. by using the bootp:nn virtual disk, see *vdisk*(A)). Machine specific information may also be printed out about the boot block information.

Example

```
di /profile/hosts/emt/bootp:00
```

Shows the disk layout of */profile/hosts/emt/bootp:00*.

```
This disk has a SunOS label
Partition start      end        size         owner
00 (a)    0x000004fb 0x00202936 (0x0020243c) Amoeba
                Amoeba partitioning for partition 0:
                vdisk  start      end        size
                 001   0x000004fb 0x00000cca (0x000007d0)
                 002   0x00000ccb 0x00000e5a (0x00000190)
                 003   0x00000e5b 0x00001a12 (0x00000bb8)
                 004   0x00001a13 0x00202936 (0x00200f24)
01 (b)    0x000004fb 0x000004fa (0x00000000) Empty
02 (c)    0x000004fb 0x000004fa (0x00000000) Empty
03 (d)    0x000004fb 0x000004fa (0x00000000) Empty
04 (e)    0x000004fb 0x000004fa (0x00000000) Empty
05 (f)    0x000004fb 0x000004fa (0x00000000) Empty
06 (g)    0x000004fb 0x000004fa (0x00000000) Empty
07 (h)    0x000004fb 0x000004fa (0x00000000) Empty
```

The disk contains 1 partition. This partition is used for Amoeba and is subdivided into 4 vdisks.

See Also

vdisk(A).

Name

disklabel – partition and label disks

Synopsis

```
disklabel host-capability
```

Description

The *disklabel* utility is used to put information on a host's disk(s) which tell the virtual disk server (see *vdisk*(A)) how to use the disk. The label on a disk is used by the virtual disk server to obtain information about the characteristics of the disk and the way it has been partitioned. Partitioning is used to divide the disk into sub-disks, each with their own special use. This is necessary since some programs need an ''entire disk'' but in fact only a small one. Therefore the disk is divided into what look like several small disks. These are known as partitions. The virtual disk server will then take each partition (or perhaps a set of partitions) and present it to clients as a virtual disk. For example, the Bullet Server (see *bullet*(A)) requires a complete virtual disk. Soap (see *soap*(A)) also requires a complete virtual disk, but one of only 1 or 2 Mbytes. Since most systems have only one physical disk it is necessary to partition it into several sub-disks, each of which will be represented as a virtual disk by the virtual disk server.

When a kernel with a virtual disk server is booted, it will create in the kernel's directory, a *bootp* entry for each physical disk, even if the disk is not labeled. The *disklabel* program will search the kernel directory of the host specified on the command line and present all the accessible disks for labeling. It is not necessary to label all of them. Part or all of a disk can be set aside for other purposes.

It is possible to run the *disklabel* utility when the system is in normal operation. It is not necessary to shut the system down or to run the program on any particular processor. However, once *disklabel* has been run, before the virtual disk server can become aware of the new label(s) the machine with the disk(s) must be rebooted.

Note, it is not necessary to run this program on the same machine as the disks or even on a machine of the same architecture. It can be run on any Amoeba host which can access the specified host.

Warning

Labeling a disk is a dangerous operation. If the disk to be labeled already has a label then altering the label in some way will almost certainly mean that the data on the disk can no longer be retrieved. Therefore experimenting with disks containing valuable information is not recommended. Also, as a consequence, confirmation is usually required before proceeding with a destructive operation. It is worth thinking before answering.

Preliminaries

Before labeling a disk it is necessary to know several pieces of information. The first is the type of disk it is. If it is not on the list of disks known to the *disklabel* utility then the disk geometry must be provided. The following information is required: the number of cylinders

on the disk, the number of read/write heads and the number of sectors per track. Other questions will be asked about the geometry but the above three pieces of information are sufficient to answer all the questions. Note that sectors are assumed to be 512 bytes.

The other thing that needs to be done in advance is to determine how many partitions are required and how big they should be. Note that some systems (such as SunOS™) require that partitions begin on cylinder boundaries and this should be taken into consideration when determining partition sizes.

It is important to understand the nature of virtual disks for a good understanding of the disk labeling process and so it is recommended that the *vdisk*(A) manual page be studied. The structure of the disk is as follows. Somewhere near the beginning of the disk is the disk label. The contents of it is usually defined by the operating system for the machine. In the case of Amoeba, it can either be the label of the operating system delivered with the machine (for example, a SunOS disk label on a SPARCstation) or an Amoeba disk label. Typically it contains the disk geometry information and a partition table. The reason for recognizing the label of the original operating system is that it may be possible to share a disk with another operating system. It also simplifies bootstrapping. Amoeba will look at the first block of each disk partition. If the partition has an Amoeba disk label on block 0 then Amoeba will use that disk partition. Otherwise it will not touch it and assume it is for another operating system. Amoeba may use more than one partition from such a disk.

The structure of the data on disks varies according to the possibilities provided by the host operating system. In the case of a 386/DOS system the structure of the disk is (approximately) as in figure 8.1.

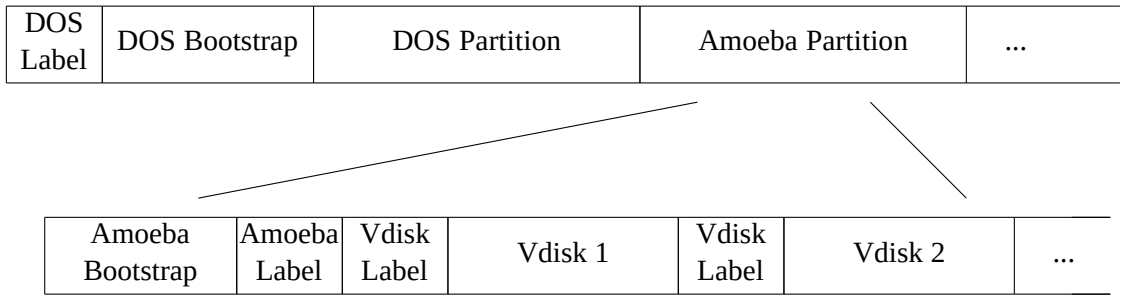


Fig. 8.1. DOS Disk structure

It is not necessary have a partition for DOS. The whole disk can be assigned to Amoeba. Some Amoeba systems may also use an extra partition for bad block mapping. NB. The physical disk must be labeled before Amoeba can use it. For DOS disks this is done with the *fdisk*(A) utility which should be run **before** *disklabel*.

When using Sun hardware, the following disk layout is used. This allows Amoeba to boot from disk using the standard PROMs in the Suns. Note that it is possible to use one Sun partition for Amoeba and the rest for SunOS but the following shows the entire disk given over to Amoeba.

Bad block mapping does not require a separate partition in this case, however a few spare cylinders should be assigned when specifying the disk geometry.

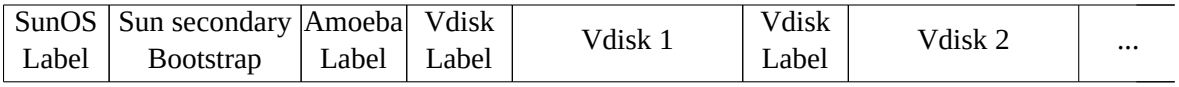


Fig. 8.2. SunOS Disk Structure

At whatever level an Amoeba disk label is found (either in place of the host label or on one or more partitions) it expects partitioning to take place. This means that if there is only one free disk partition on a system, Amoeba can have more than one partition of its own by sub-partitioning that partition. Users do not see partitions but virtual disks. Normally there is a one to one correspondence between partitions and virtual disks but a virtual disk may comprise more than one partition. If so, not all partitions need necessarily be on the same physical disk. It is possible to treat partitions from two or more separate physical devices as a single virtual disk and thus make quite a large disk for a particular server (such as a Bullet Server). This is done using the *concatenation* option (see below).

The *disklabel* utility asks lots of questions and the simplest way to explain its use is by means of an example.

Example

The following is an example of the dialogue that is produced when using the *disklabel* utility. The text printed by the computer is in constant width text and the responses typed at the keyboard are in bold. Comments on what is happening appear in normal (Roman) text. In this case it is labeling the disk on a Sun. The command to do this is:

disklabel /super/hosts/*xxxx*

where *xxxx* is the name of the host with the disk to be labeled.

For a typical Amoeba system four disk partitions are required. One for the Soap Server (about 1 – 2 Mbyte), one for the Boot server (about 1 Mbyte), one for Amoeba kernels to be booted from the disk (about 1 Mbyte) and one for the Bullet Server (as large as possible). Four partitions will be made, namely those needed for a standard system as mentioned above. The first thing that must be established is which host operating system is to be used.

```
Which operating system disk label should be used?
  0. Dos
  1. SunOS

Your selection: 1
```

SunOS is the required host OS in this case. After this the main menu will be displayed. It offers the various possibilities.

```
 0. Exit disklabel program
 1. Display Physical Disks
 2. Display Physical Partitions
 3. Display Virtual Disks
 4. Label Physical Disks
 5. Label Virtual Disks
 6. Merge Virtual Disks
 7. Unmerge Virtual Disks

Your selection:
```

Firstly it is good to know how to stop the program. To quit the program at this menu enter a zero followed by a return. At any time it is also possible to interrupt the program but care should be taken to do it at a point were a consistent state will be left on the disk.

The first step in labeling is to display the information about the disks *disklabel* found attached to the host *xxxx*. To do this select menu item 1.

```
Your selection: 1

  Physical disks available on host ‘‘xxxx’’
  No. Name      Size (in sectors) Label type
   0: bootp:00       640583       SunOS

continue ?
```

What we see is that this host has one disk of 320 MB. It has a SunOS label on it.

If these disks have previously been used for Amoeba then menu items 2 and 3 can also be used to display the current partitioning of the disks and the structure of any previously defined virtual disks.

In response to the ‘‘continue ?’’ pressing any key that transmits a character will cause the message to disappear and the main menu to reappear. The main menu will not be reproduced again in this example. Refer to it above if necessary. Note that in response to yes/no questions you should not press return but only ‘y’ or ‘n’.

Labeling Physical Disks

Now we proceed with labeling the physical disks, namely menu item 4. It should also be pointed out at this point that to label a DOS disk it is necessary to use the program *fdisk*(A) It cannot be done with *disklabel*.

```
Your selection: 4

  Physical disks available on host ‘‘xxxx’’
  No. Name      Size (in sectors) Label type
   0: bootp:00       640583       SunOS
Do you wish to (re)label one of these disks? [y or n]: y
Please enter the number of the disk to be labeled : 0

You selected 0
```

At this point the contents of the disk label will be displayed. In the case of Sun this consists of the disk geometry and the partition table. This information is present on this particular disk since it was previously used for SunOS.

```
This disk already has a label.
The current label is as follows:

      SUNOS Disk Label
Alternates per cyl  : 0
Bytes in Gap1       : 0
Bytes in Gap2       : 0
Interleave Factor   : 0
Number of Cylinders : 1545
Alternate Cylinders : 4
Number of Heads     : 9
Sectors per track   : 46
Label head          : 0
Physical Partition  : 0
Partition Table
Partition  First Cylinder  Number of Sectors
     a             0          55890
     b           135         414000
     c             0         639630
     d             0              0
     e             0              0
     f             0              0
     g          1135         169740
     h             0              0

Changing the label will almost certainly destroy any data on the disk.
Are you sure that you want to change it? [y or n]: y

You can modify this label or make one from scratch.
Do you want to modify this label? [y or n]: y
```

Note that in this case a cylinder contains 9x46=414 sectors. Also note that in the following the current value for a field is displayed between square brackets. If this value is the required one then pressing return will leave it unchanged. Entering a new value next to it will replace the old value. In the case of the geometry it need not change since it is already correct.

In the following the first four and the last two fields can be set to zero since they are not currently used. The total number of cylinders should equal the sum of the *# Cylinders (Used)* and *# Alternate Cylinders*. The alternate cylinders are for bad block mapping and for a disk of this size two alternate cylinders are usually sufficient.

```
Please enter the disk geometry information
Alternates per cyl.   : [0]
Bytes in gap1         : [0]
Bytes in gap2         : [0]
Interleave factor     : [0]
# Cylinders (Used)    : [1545]
# Alternate Cylinders : [4]
# Read/Write Heads    : [9]
# Sectors per Track   : [46]
Label location - head : [0]
Label location - part : [0]
Is all the above information correct? [y or n]: y
```

If you make a label from scratch or if there was no label found on the disk it will give you the following menu:

```
Disk Types:
 1.  CDC Wren IV 94171-307
 2.  CDC Wren IV 94171-344
 3.  CDC Wren V 94181-385
 4.  CDC Wren V 94181-702
 5.  CDC Wren VII 94601-12G
 6.  Fujitsu M2243AS
 7.  Fujitsu M2266SA
 8.  Micropolis 1325
 9.  Other
Enter the type of the disk to be labeled:
```

Additions to this menu should be put in the file *config.c* in the sources.

If the disk present is not one of those mentioned then select ‘‘Other’’. The disk geometry must be provided below. If the disk type is known then it will fill in the geometry automatically. In this example the type is known and so when enquiring about the geometry information the ‘‘known’’ value is printed in brackets. If *return* is typed it will use the known value for each field. Otherwise it can be overridden by typing in the new value and pressing *return*.

Once the geometry is filled in, the disk partition table must be filled in. In this case it is the partition table that SunOS normally uses. It requires that partitions be aligned on cylinder boundaries, as can be seen by the information requested. Note that SunOS also permits partitions to overlap. The Amoeba virtual disk server will reject overlapping partitions, so be careful when specifying this partition table. Once again, if there was already a partition table on the disk, the current values would be printed in brackets after the colon. If cylinder numbers or block counts which do not fit on the disk are given an error message is printed and the information will be requested again. If you realize that you have made a mistake do not worry. At the end it will print a summary and ask if everything is ok. Respond with **n** to be able to re-enter the information.

NB. It is best to make a single partition of the entire disk (or that part which is to be used for Amoeba). The results will then conform to the description in the installation guide. A separate SunOS partition for each Amoeba partition will only lead to confusion.

```
There are 640169 sectors of 512 bytes available (1544 cylinders).
Please enter the partition table information

Partition  a: First Cylinder    : [0]
              Number of Sectors : [55890] 640169

Partition  b: First Cylinder    : [135] 0
              Number of Sectors : [414000] 0

Partition  c: First Cylinder    : [0] 0
              Number of Sectors : [639630] 0

Partition  d: First Cylinder    : [0]
              Number of Sectors : [0]

Partition  e: First Cylinder    : [0]
              Number of Sectors : [0]

Partition  f: First Cylinder    : [0]
              Number of Sectors : [0]

Partition  g: First Cylinder    : [1135] 0
              Number of Sectors : [169740] 0

Partition  h: First Cylinder    : [0]
              Number of Sectors : [0]
Partition Table Summary
Number of Sectors on Disk = 640169
Partition  First Cylinder  Number of Sectors
     a             0        640169
     b             0             0
     c             0             0
     d             0             0
     e             0             0
     f             0             0
     g             0             0
     h             0             0
Is the partition table ok? [y or n]: y
```

At this point it has a complete host operating system disklabel and will (over)write the disk label on the disk if **y** is given in answer to the next question.

```
Last chance.  Do you really want to label this disk? [y or n]: y
Disk label has been written.

Do you wish to label another disk? [y or n]: n
```

The program will now return to the main menu. The next step is to go through all the physical partitions present and determine which can be used for Amoeba. Those that can be used for Amoeba are then labeled with an Amoeba partition label. This label describes the sub-partitioning required. To do this select main menu item 5.

```
Your selection: 5

You must now decide which partitions to allocate to Amoeba.

Disk 'bootp:00' Partition a (640169 blocks):
Can I use this partition for Amoeba? [y or n]: y
You need to give the sub-partitioning of this partition.
Please enter the partition table information
```

At this point four partitions will be made as per the standard installation procedure. Other partitionings can be made for other purposes, of course. Note that now there is no discussion of cylinders. Sector addresses are used. Cylinder addressing is a peculiarity of Sun's.

NB. When partitioning, the block numbers should be given relative to the start of the disk partition. They should not be absolute disk addresses. Therefore, block 0 is the first sector number on the disk.

```
Partition a: First Sector      : 0
             Number of Sectors : 2000

Partition b: First Sector      : 2000
             Number of Sectors : 200

Partition c: First Sector      : 2200
             Number of Sectors : 2000

Partition d: First Sector      : 4200
             Number of Sectors : 635969
Partition Table Summary
Number of Sectors on Disk = 640169
Partition  First Sector  Number of Sectors
    a             0           2000
    b          2000            200
    c          2200           2000
    d          4200         635969
    e             0              0
    f             0              0
    g             0              0
    h             0              0
Is the partition table ok? [y or n]: y
```

At this point if the answer **n** is given then the table will be present for a second attempt. Note that for historical reasons it is necessary to use all the blocks in a partition. This is enforced. The answer **y** should return *disklabel* to the main menu (unless there are more partitions to

label).

Selecting option 3 of the main menu should now display the table of available virtual disks. It should now be possible to exit and reboot the host with the disk. When rebooted it should then have 4 virtual disks available.

If the system has more than one physical disk or several partitions available to Amoeba it is no problem to concatenate several partitions to form a single large disk. Item 6 of the main menu will do that. Note that if a partition forms part of a set of partitions then it is safer to unmerge it before re-subpartitioning it. Use main menu item 7 for this.

See Also

vdisk(A).

Name

doctree – build a configuration tree for the Amoeba documentation

Synopsis

```
doctree [-v] conf-root doc-root template-root
```

Description

Doctree is used to construct the configuration trees for building the Amoeba documentation. Note that *doctree* does not perform the actual build but merely creates the hierarchy with the necessary Amakefiles for building the documentation.

The configuration is constructed by cloning a template configuration tree which contains the directory structure and necessary *Amakefiles*. If the target configuration tree specified does not exist it will be created. If it already exists, then it will be brought up to date. Note that any parts that are no longer in the template tree will not be deleted. If any file in the configuration is different from the corresponding file in the templates then *doctree* will enquire as to whether it should overwrite the file or not. If you respond by typing *q* then no change will be made and the program will terminate. If you answer with *d* then you will be shown the differences and asked again. If you type *y* the change will be made. Any other response will be interpreted as ‘‘do not overwrite but continue’’.

If any of the arguments is not an absolute path name then the output of the *amdir*(U) command will be prepended to it. Otherwise it will be used as is.

The *conf-root* specifies the place where the configuration should be built.

The *doc-root* specifies the place where the source of the documentation is found.

The *template-root* specifies where to get the templates for the *Amakefiles*. The *Amakefiles* will be parameterized with the *doc-root* argument.

Options

–v The verbose option. This causes the program to print details of everything it is doing. This may produce quite a lot of output.

Example

```
doctree /amoeba/conf/doc /amoeba/doc /amoeba/templates
```

will create under the directory */amoeba/conf/doc* a tree structure containing all the Amakefiles to build the various elements of the documentation.

See Also

amake(U), amoebatree(A), build(A), kerneltree(A).

Name

dread – read raw data blocks from a virtual disk

Synopsis

```
dread [-s startblock] [-n num_blocks] [-b block_size] disk_cap
```

Description

Dread reads *num_blocks* blocks from the virtual disk specified by the *disk_cap* capability starting at block number *start_block* and writes it to *stdout*. The block size must be a power of two. The default values for *start_block*, *num_blocks* and *block_size* are respectively 0, 1, and 512.

Error messages are supposedly self explanatory. The exit status is non-zero on error.

Example

```
dread -s 3 /super/hosts/boris/vdisk:01 > data_file
```

will read the first 512 bytes of block 3 from *vdisk:01* on the host called *boris*. The data will be stored in *data_file*.

See Also

dwrite(A).

Name

dwrite – write raw data blocks to a virtual disk

Synopsis

```
dwrite [-s start_block] [-n num_blocks] [-b block_size] disk_cap
```

Description

Dwrite writes *num_blocks* blocks from *stdin* to the virtual disk specified by the *disk_cap* capability starting at block number *start_block*. The block size must be a power of two. The default values for *start_block*, *num_blocks* and *block_size* are respectively 0, 1, and 512.

Error messages are intended to be self-explanatory. The exit status is non-zero on error.

Example

```
dwrite -s 3 -n 10 /super/hosts/boris/vdisk:02 < data_file
```

will write 10 blocks of size 512 bytes (from *data_file*), starting at block 3 to the virtual disk *vdisk:02* on the host *boris*.

See Also

dread(A).

(i80386 ONLY)

Name

fdisk – Amoeba fixed disk utility

Synopsis

```
fdisk [-h n] [-s n] [-c n] [-m master] disk-capability
```

Description

Fdisk is a menu driven program that is used to partition the hard disk of an i80x86 machine. *Fdisk* can be used to modify an existing partition table as well as initialize a new disk. The functions offered by *fdisk* are summarized below.

Fdisk is potentially a harmful program, capable of destroying the contents of a hard disk. To try to reduce the risks of errors, *fdisk* will always ask for confirmation before writing the changes to disk.

Below is a description of the operations that can be performed from the menu:

Display partition table
: Show the contents of the partition table in human readable form.

Create a partition
: A partition table contains four partition table entries which can specify up to four, possibly different, operating systems. *Fdisk* allows you to create a new Amoeba partition, by specifying its size and disk offset. For compatibility reasons the size and offset are specified in cylinders instead of blocks. Note that the created Amoeba partition is uninitialized. Further initialization needs to be done using *disklabel*(A).

  After confirming that this is really intended, the program will ask for the first cylinder and the last cylinder of the disk to be allocated to the new partition.

Change active partition
: One of the four entries in the partition table can be active, i.e., bootable. At boot-time the master boot program will search the table for an active entry, read its first block into memory, and execute it. This allows you to switch among the, possibly different, operating systems on your disk. The *change active partition* option allows any entry in the partition table to be activated. Note: it is usually unwise to deactivate Amoeba, since the Amoeba secondary bootstrap loader provides a mechanism to start other operating systems as well.

Delete a partition
: Apart from creating a new partition table entry, *fdisk* also allows the removal of such an entry. However, before it actually removes the entry it asks for confirmation. Although the result of the removal operation is visible to *fdisk* it does not come into effect until the partition table is written to disk upon exit of the *fdisk* program.

Scan and assign bad sectors
: The Amoeba Winchester hard disk driver is capable of mapping bad blocks onto alternate blocks, provided there exists a mapping table. This option will create such a mapping table and should normally only be used during initial system setup after

formatting the disk and before partitioning it. This operation will cause a small partition to be created to hold the bad block remapping information.

It should be run directly **after** formatting the disk since only blocks marked as *bad* by the formatting will be correctly remapped. *Fdisk* may see other blocks which give ECC errors but it cannot reformat them as bad blocks. If there is a suspicion that new bad blocks have appeared during the lifetime of a disk then this option may well detect them. If bad blocks appear after formatting then before they can be remapped the tracks containing the new bad blocks must be reformatted, or the entire disk if there is no track reformatting available. If a disk is reformatted then this option should be run to remake the mapping table before repartioning.

This option should be run **before** creating other partitions so that the bad blocks can be remapped before they are assigned to a partition.

It is not necessary to create a bad block partition if the disk will do its own bad sector mapping internally.

There are two ways of creating a bad block mapping. One is to let the program scan the disk for bad blocks and generate its own mapping table. If this is desired then when it gives the following prompt answer **y**.

```
Scan disk for bad blocks ?
```

NOTE WELL: This operation can take a long time (over 1 hour for a 1 Gigabyte disk). The other alternative is to manually enter a list of known bad blocks. In this case answer **n** to the previous question and it will then give the prompt

```
Enter bad blocks manually ?
```

Answer with **y** to take this option. The bad block numbers are entered one per line. When the list is complete then enter block 0 to terminate the list. Answering the above question with **n** will return the program to the main menu.

Options

Ordinarily the disk geometry of the disk is obtained from the disk server. However, sometimes it is useful to overrule this information.

−h n — This option overrules the number of heads.

−s n — This option overrules the number of sectors per track.

−c n — This option overrules the number of cylinders on the disk.

−m master — Specify a new master boot block instead of the one built in.

Example

To partition a disk on machine ‘‘bullet’’, the following command should be issued:

```
fdisk /super/hosts/bullet/bootp:00
```

See Also

isaboot(A), vdisk(A).

Name

fifosvr – a first-in first-out file server

Synopsis

```
fifosvr [-f] [-i] [-n numfifos]
```

Description

The *fifo* server is used to implement *named pipes* (also called *fifos*) under Amoeba. Fifos differ from normal pipes in the following two ways:

- fifos have an entry in the file system.
- fifos can be opened independently by separate programs.

A fifo is created by a call to the POSIX primitive *mkfifo*(U). After that, further operations can be performed using the general POSIX file primitives *open*, *read*, *write* and *close*. When a fifo is opened, the *fifo* server returns a new capability that can be used for further operations on the fifo. Amoeba's POSIX emulation library will take care of mapping the POSIX primitives onto operations supported by the *fifo* server.

Because it is possible for the capability of a fifo to be lost, the general system of garbage collection is used for the *fifo* server. See also *std_age(A), bullet*(A) and *om*(A).

Note that fifos will survive crashes of the fifo server.

Options

–i This option causes the *fifo* server to (re)initialize its state. It generates a new server port, and generates a new, clean state file. Any fifo object created by a previous instantiation of the fifo server will become invalid.

–f To prevent accidentally overwriting an existing state file or super capability, the *fifo* server by default refuses to replace them. This option can be used to force replacement of an existing state file or super capability.

–n *numfifos*
When initializing, the *fifo* server reserves a default amount of object numbers for the fifos (typically 500). This option can be used to change that number.

Files

/profile/cap/fifosvr/default – default server capability.
/super/module/fifo/state – default fifo state file.

Programming Interface Summary

The programmers' interface consists of commands particular to the *fifo* server (whose names begin with *fifo_*), general file server commands (whose names begin with *fsq_*), and standard server commands (whose names begin with *std_*). The commands are divided into two categories: administrative and user. A summary of the commands is presented in the following two tables, the first listing the user commands and the second the administrative

commands. Besides the error conditions shown, general RPC errors are also possible.

User Functions			
Function Name	Required Rights	Error Conditions	Summary
fifo_create	BS_RGT_CREATE	STD_CAPBAD STD_DENIED STD_ARGBAD STD_NOSPACE	Create a new fifo
fifo_open	BS_RGT_MODIFY	STD_CAPBAD STD_DENIED STD_ARGBAD STD_NOSPACE FIFO_BADF FIFO_ENXIO	Open a fifo for reading or writing
fifo_share	BS_RGT_MODIFY	STD_CAPBAD STD_DENIED	Increase the usage count for an open fifo
fifo_close	BS_RGT_MODIFY	STD_CAPBAD STD_DENIED	Decrease the usage count for an open fifo
fifo_bufsize	BS_RGT_READ	STD_CAPBAD STD_DENIED	Return the buffer size used to implement fifos
fsq_read	BS_RGT_READ	STD_CAPBAD STD_DENIED STD_ARGBAD FIFO_AGAIN	Read data from an open fifo
fsq_write	BS_RGT_MODIFY	STD_CAPBAD STD_DENIED STD_NOTNOW FIFO_AGAIN FIFO_BADF	Write data to an open fifo
std_destroy	BS_RGT_DESTROY	STD_CAPBAD STD_DENIED	Destroy a fifo
std_info	BS_RGT_READ	STD_CAPBAD STD_DENIED STD_ARGBAD	Print standard information for a fifo object: a | followed by details of reads and writes
std_restrict		STD_CAPBAD	Produce a fifo capability with restricted rights

Administration

In general there is very little administration required for a *fifo* server once it is running. See *om*(A) how to incorporate the *fifo* server in the standard garbage collection process. Once installed, the *fifo* server can be brought under *boot* server control (see *boot*(A)) to keep it up and running.

Administrative Functions			
Function Name	Required Rights	Error Conditions	Summary
std_age	BS_RGT_ALL	STD_CAPBAD STD_DENIED STD_ARGBAD	Start a garbage collection pass
std_status	BS_RGT_READ	STD_CAPBAD STD_DENIED STD_ARGBAD	Print *fifo* server statistics
std_touch		STD_CAPBAD	Reset a fifo's time to live

Installation

The *fifo* server installations itself when started with the **-i** option. This will generate and install a new server capability as well as a new administrative state file, containing the check fields for access protection.

Examples

Start the server for the first time (since it publishes its super capability in */super/cap/fifosvr*, you have to be system administrator to do this):

```
# fifosvr -i &
```

Subsequently it should be started without the **–i** option.

The following creates a fifo, performs some operations, and deletes it:

```
$ mkfifo /tmp/fifo
$ echo foo >/tmp/fifo &
$ echo bar >/tmp/fifo &
$ dir -l /tmp/fifo
/tmp/fifo              Jul 12 14:43  | r = 0; w = 2
$ cat /tmp/fifo
bar
foo
$ del -f /tmp/fifo
```

See Also

boot(A), bullet(A), mkfifo(U), om(A), soap(A), std(L), std_age(A), std_destroy(U), std_info(U), std_touch(U).

Name

flip_ctrl – control status of a FLIP network

Synopsis

```
flip_ctrl [-d <delay>] [-l <loss>] [-o <on>] [-t <trusted>] machine ntw

UNIX ONLY:
flip_ctrl [-d <delay>] [-l <loss>] [-o <on>] [-t <trusted>] ntw
```

Description

Flip_ctrl is used to manage the state of the FLIP network *ntw*. See the description of the options below for details of what can be managed. Under Amoeba the *machine* argument must be specified along with the FLIP network. Under UNIX if the *machine* argument is given then it specifies the Amoeba host to be altered. If the *machine* argument is not present the status of the FLIP network in the UNIX kernel on which *flip_ctrl* is executed is changed.

Note that only Amoeba kernels and UNIX FLIP drivers compiled with the `FLNETCONTROL` option will accept this command.

Options

–d delay Artificially increases the delay of network *ntw*. Not implemented yet.

–l loss Increases artificially the loss rate of network *ntw*.

–o on Switches network *ntw* on or off.

–t trusted Switches network *ntw* to trusted or non-trusted.

Warnings

Only use *flip_ctrl* if you know what you are doing. The machine whose FLIP networks are changed may become unreachable and have to be rebooted.

See Also

kstat(A), flip_dump(A), flip_stat(A).

(UNIX ONLY)

Name

flip_dump – print FLIP tables from UNIX FLIP driver

Synopsis

```
flip_dump
```

Description

Flip_dump prints the current state of the FLIP driver in the UNIX kernel upon which it is run. The information consists of the state of the current RPCs, the contents of the port cache, the known kernel identifiers (kids), network statistics, the routing table and any network driver specific tables such as the fragmentation information for Ethernet. The interpretation of this information is best done with the aid of the source code.

If no FLIP driver has been installed in the kernel it will print

```
flip_dump failed
```

See Also

flip_stat(A), kstat(A).

(UNIX ONLY)

Name

flip_stat – print the FLIP statistics from a UNIX kernel

Synopsis

```
flip_stat
```

Description

Flip_stat prints out the current statistics kept by the FLIP driver in the UNIX host on which the command is run. If no FLIP driver has been installed in the kernel it will print

```
flip_dump failed
```

Otherwise it will print details about the number of RPCs performed and statistics of the low-level network driver(s). To interpret the statistics a knowledge of the network(s) connected to the hosts and an acquaintance with the source code are required.

Example

```
flip_stat
```

might print something like:

```
======== FLIP AMRPC statistics =======
send statistics:
slocate      890 shereis       20 srequest     555 sreply      3139
sack         681 snak           2 salive      1051 senquire       0
sfail          0 sreceived       1
receive statistics:
rlocate    40511 rhereis       80 rrequest    3140 rreply       546
rack        3143 rnak           2 ralive         0 renquire    1051
rfail          0 rreceived     137
timeout     815
======= FLIP interface statistics ===========
slocate        374 shereis          5 snothere         0 sunidata      5431
smultidata       0
rlocate      37529 rhereis         42 rnothere         3 runidata      8087
rmultidata   40511
======== network statistics =======
==== Ethernet: #0 ====
slocate        286 shereis          5 snothere         0 sunidata         0
smultidata     611
rlocate      37571 rhereis         42 rnothere         3 runidata      8087
rmultidata   40531
dropped          6 bcast drop       0
==== interface: #0 ====
```

```
slocate      37571 shereis         42 snothere         3 sunidata      8087
smultidata   40525
rlocate        374 rhereis          5 rnothere         0 runidata         0
rmultidata     797
dropped          0 bcast drop       0
============================
============== ff statistics =======
scredit   :      0 spiggy    :      0 srequest  :      0 smcreq    :      0
smcack    :      0 smcnak    :      0 sabscredit:      0
rcredit   :      8 rpiggy    :      0 rrequest  :      0 rmcreq    :      0
rmcack    :      0 rmcnak    :      0 rabscredit:      0
========= statistics le ========
ip 273938 ie 253 op 65006 oe 2 col 118
========================
```

See Also

flip_dump(A), kstat(A).

Name

ftpd – DARPA Internet File Transfer Protocol server

Synopsis

```
ftpd [-dl] [-t timeout] [-T maxtimeout]
```

Description

Ftpd is the DARPA Internet File Transfer Protocol Daemon. The daemon uses the `TCP` protocol and listens at the port specified in the *ftp* service specification. It is usually started by *ftpsvr*(A) whenever a new ftp client makes a connection.

The various commands accepted by *ftpd* are described in the *Programming Interface* section below.

When the ftp daemon starts it opens a control connection which looks a lot like *Telnet*. When a file transfer is started a special data transfer channel is opened for the duration of the transfer. The ftp daemon will abort an active file transfer only when the ABOR command is preceded by a Telnet ‘‘Interrupt Process’’ (IP) signal and a Telnet ‘‘Synch’’ signal in the command Telnet stream, as described in Internet RFC 959. If a STAT command is received during a data transfer, preceded by a Telnet IP and Synch, transfer status will be returned.

Ftpd interprets file names according to the ‘‘globbing’’ conventions used by *csh*. This allows users to utilize the metacharacters `*?[]{}~`.

Under Amoeba, *ftpd* authenticates users by starting a *login*(A) session for a connecting client. Anonymous login can be implemented by creating a special *ftp* or *anonymous* user without a password.

Options

–d Debugging information is written to the syslog.

–l Each *ftp* session is logged to *stderr*. This should probably be redirected to a file.

–t *timeout*
The inactivity timeout period is set to *timeout* seconds (the default is 15 minutes).

–T *maxtimeout*
A client may also request a different timeout period; the maximum period allowed may be set to *maxtimeout* seconds. The default limit is 2 hours.

Warnings

The anonymous account is inherently dangerous and should be avoided when possible.

The server must have access to the /super capability in order to be able to start a login session. This is the case when ftpd is started by *ftpsvr*(A) under *boot*(A) server control.

Programming Interface Summary

The ftp server currently supports the following ftp requests; case is not distinguished.

Request	Description
ABOR	abort previous command
ACCT	specify account (ignored)
ALLO	allocate storage (vacuously)
APPE	append to a file
CDUP	change to parent of current working directory
CWD	change working directory
DELE	delete a file
HELP	give help information
LIST	give list files in a directory – ‘‘ls –lgA’’
MKD	make a directory
MDTM	show last modification time of file
MODE	specify data transfer – mode
NLST	give name list of files in directory
NOOP	do nothing
PASS	specify password
PASV	prepare for server-to-server transfer
PORT	specify data connection port
PWD	print the current working directory
QUIT	terminate session
REST	restart incomplete transfer
RETR	retrieve a file
RMD	remove a directory
RNFR	specify rename-from file name
RNTO	specify rename-to file name
SITE	non-standard commands (see next section)
SIZE	return size of file
STAT	return status of server
STOR	store a file
STOU	store a file with a unique name
STRU	specify data transfer – structure
SYST	show operating system type of server system
TYPE	specify data transfer – type
USER	specify user name
XCUP	change to parent of current working directory (deprecated)
XCWD	change working directory (deprecated)
XMKD	make a directory (deprecated)
XPWD	print the current working directory (deprecated)
XRMD	remove a directory (deprecated)

The following non-standard or UNIX-specific commands are supported by the SITE request.

Request	Description
UMASK	change umask. – E.g. SITE UMASK 002

IDLE	set idle-timer. – E.g. SITE IDLE 60
CHMOD	change mode of a file. – E.g. SITE CHMOD 755 filename
HELP	give help information. – E.g. SITE HELP

The remaining ftp requests specified in Internet RFC 959 are recognized, but not implemented. MDTM and SIZE are not specified in RFC 959, but will appear in the next updated FTP RFC.

See Also

boot(A), ftp(U), ftpsvr(A), ipsvr(A), telnetd(A).

Name

ftpsvr – a front-end for the FTP daemon

Synopsis

```
ftpsvr [-T <tcp-cap>] [-daemon <ftpd-prog>] [-v]
```

Description

The *ftpsvr* is used to start FTP daemons (see *ftpd*(A)). When a client tries to make an FTP connection, *ftpsvr* starts an FTP daemon for that client.

Ftpsvr is typically started by the *boot*(A) server.

Options

–T *tcp-cap*
: This causes the server to use the TCP/IP server specified by *tcp-cap*. If this option is omitted it uses the server specified by the string environment variable TCP_SERVER. If this is not defined it uses the default TCP server as specified in *ampolicy.h* (typically */profile/cap/ipsvr/tcp*).

–daemon *ftpd-prog*
: This changes the FTP daemon to be started to *ftpd-prog*. The default is */super/admin/bin/ftpd*.

–v Verbose mode. Various debugging information is printed.

Environment Variables

TCP_SERVER specifies the TPC/IP server to use if the **–T** option is not specified.

Files

/super/admin/bin/ftpd is the default FTP daemon.

See Also

ftpd(A), ftp(U).

Name

iboot – inspect or install the boot server's virtual disk

Synopsis

```
iboot [-b bincap] [-c confblk] [-d datablk]
       [-f conffile] [-l] [-s supercap] [-v] vdisk
```

Description

Iboot reads, modifies, or writes the virtual disk *vdisk* used by *boot*(A). Without any options, it prints the configuration file on disk. The options allow you to modify portions of the data, as follows:

Options

–b bincap This changes the capability for the boot server's executable. In other words, you need this to install a new version of the boot server.

–c confblk This assigns the block of the virtual disk where the configuration data starts.

–d datablk This tells at which block the boot server can store its own data.

–f conffile This tells *iboot* on which file the configuration data are stored. On UNIX, this must be a UNIX-file, on Amoeba this must be a bullet-file.

–l This is the option you need to start using a new virtual disk. Without this option, *iboot* verifies that the magic string at block zero of the disk is correct.

–s supercap This sets the super-capability of the boot server to *supercap*.

–v With this option, *iboot* prints the information in block zero of the virtual disk in addition to the configuration file.

After running *iboot* run the command

```
boot_reinit /super/cap/bootsvr/xxx
```

where *xxx* is the name of the boot server. This should indicate success if the *Bootfile* was valid. If not then do not reboot the system under any circumstance. Rather run

```
std_status /super/cap/bootsvr/xxx
```

to find out the problem and then correct the *Bootfile* and repeat the installation process with *iboot*.

Files

The format of the virtual disk is defined in *src/h/server/boot/bsvd.h*. It contains a header in block 0, some space for the boot server, and the configuration file. The format of the configuration file is explained in *boot*(A).

Diagnostics

The error messages of *iboot* are self-explanatory. The exit status is 0 on success, 2 in case of a command line error, and 1 for other problems.

Warning

Use the **–l** option with care. It will overwrite the specified virtual disk without any sanity checks!

Examples

To read the configuration file on the virtual disk */profile/hosts/mark/vdisk:02*, type:

```
iboot /profile/hosts/mark/vdisk:02
```

To install *newboot* as the new boot server executable, type:

```
iboot -b newboot /profile/hosts/mark/vdisk:02
```

The first time you start using a virtual disk as boot-disk, type:

```
makecap boot.cap
iboot -b /bin/boot -f boot.config -s boot.cap /profile/hosts/mark/vdisk:02
```

See Also

boot(A), boot_reinit(A), makecap(U).

Name

ifconfig – set/get the IP address and or netmask

Synopsis

```
ifconfig [-h ipaddr] [-i] [-n netmask] ip-cap
```

Description

The TCP/IP server (see *ipsvr(A))* needs an IP address. This address must be set manually using *ifconfig. Ifconfig* can also set the netmask. If *ifconfig* is called without any option the current IP address and netmask will be reported unless the IP server does not know an IP address for that capability. In that case the error ‘‘hostaddr not set’’ will be reported after a while. The *ip-cap* argument can be either the capability of the host where the IP server is running or the full path of the ‘‘ip’’ capability of the IP server.

Options

–h ipaddr
: This option sets the IP address of the IP server specified by *ip-cap* to *ipaddr*. *Ipaddr* must be in the base-256 numeric format.

–i This option sets the netmask and or IP address only if the netmask or IP address has not already been set.

–n netmask
: This option sets the netmask of the IP server specified by *ip-cap* to *netmask. Netmask* must be in the base-256 numeric format.

Diagnostics

‘‘hostaddr not set’’ will be reported when no IP address is set.

Warnings

It is not possible to set a netmask if no IP address is set.
Some parts of the TCP/IP server block requests until the IP address is set. If an attempt is made to set the IP address but a capability of the TCP server is given then *ifconfig* will block until it is signaled or an IP address is set.

Examples

```
ifconfig /super/hosts/armada1E/ip/ip
```

If the IP address is not set, *ifconfig* will

```
hostaddr not set
```

Now we can set the IP address with

```
ifconfig -h 172.30.201.254 armada1E
```

Ifconfig will repond with:

```
hostaddr= 172.30.201.254 netmask= 255.255.255.0
```

This sets the IP address of the IP server in host armada1E to 172.30.201.254. The IP server will set the netmask to a default value or use an ICMP netmask_request to retrieve a netmask. Both the IP address and the netmask are reported. Now

```
ifconfig armada1E
```

will repond with

```
hostaddr= 172.30.201.254 netmask= 255.255.255.0
```

See Also

ipsvr(A).

Name

installboot – install the hard disk or floppy bootstrap loader on disk

Synopsis

```
installboot.ibm_at [-f] bootstrap-loader disk-capability
installboot.sun bootp disk-capability bootstrap-loader
```

Description

Installboot.ibm_at installs the Amoeba secondary bootstrap loader on hard or floppy disk, and patches the binary according to the ISA bootstrap specifications. The **–f** options specifies whether the bootstrap loader is installed on a floppy or a hard disk. For a floppy disk the bootstrap loader is just written in the first 17 sectors of the disk. For a hard disk the partition table is searched for an active Amoeba partition, and only if such a partition exists is the bootstrap loader written to disk. Besides writing the bootstrap loader on disk it also signs the binary with a magic stamp as required by the ISA primary (ROM or DISK) bootstrap loader.

Installboot.sun is used to install the secondary bootstrap loaded onto a Sun hard disk. It uses the first cylinder of the hard disk to store the bootstrap. For this it needs the *bootp* parameter. It also needs to know the name of the virtual disk where the bootable kernels are kept.

Diagnostics

Installboot.ibm_at complains when the bootstrap loader binary does not have a proper ACK binary header. For hard disk installation the absence of a bootable Amoeba partition is considered fatal.

Files

/super/admin/kernel/isa.flboot	i80386 Floppy bootstrap loader
/super/admin/kernel/isa.hdboot	i80386 Hard disk bootstrap loader
/super/admin/kernel/sun3.hdboot	sun3 Hard disk bootstrap loader
/super/admin/kernel/sun4.hdboot	sun4 Hard disk bootstrap loader

Warning

Installboot is by nature a very dangerous program and can, by improper use, damage the contents of your disk.

Examples

To install the Amoeba bootstrap loader on a i80386's hard disk, type:

```
installboot.ibm_at /super/admin/kernel/isa.hdboot /super/hosts/bullet/bootp:00
```

To do likewise for a Sun 3 with host name *foo*:

```
installboot.sun /super/hosts/foo/bootp:00 /super/hosts/foo/vdisk:01 \
                  /super/admin/kernel/sun3.hdboot
```

See Also

isaboot(A), fdisk(A), mkbootpartn(A).

Name

installk – install an Amoeba kernel

Synopsis

```
installk compiler architecture kernel-file bullet-file
installk -d blk-num compiler architecture kernel-file vdisk-name
```

Description

Installk is used to install Amoeba kernels compiled under either Amoeba or UNIX. There are two places where kernels can be installed to any advantage. One is on a disk boot partition. This is done using the **–d** option. See *prkdir*(A) for details of where on the boot partition to place the kernel. The other is in a bullet file which can then be used by *reboot*(A) or *tftp*(A).

The *kernel-file* specified should contain a kernel as produced by the loader. It should not have been stripped or otherwise modified. It is also necessary to specify the architecture for which the kernel was made and the compiler used to create it. This information is needed to determine whether the kernel needs to be stripped or converted to the form which can be booted on the specified architecture.

Options

–d blk-num
: Write the *kernel-file* to a virtual disk instead of to a bullet file. The *blk-num* argument specifies at which block number on the virtual disk *vdisk-name* at which to start writing the kernel.

Diagnostics

If an unknown compiler or architecture is specified it will print

`installk: unknown compiler:` *compiler*

or

`installk: unknown architecture:` *architecture*

followed by a list of known compilers or architectures.

If the *kernel-file* specified is not a UNIX ordinary file (as defined by *test* *–f*) then it will print

`installk: "`*kernel-file*`" not a file`

If the *kernel-file* specified is not in the expected format for the specified compiler and the kernel needs to be stripped for that architecture then the message

`installk: strip failed for "`*kernel-file*`"`

will be printed. On some systems this may also appear for kernels that are already stripped.

For combinations of compiler and architecture which *installk* does not know about it will print

```
installk: unsupported compiler/architecture combination
```

Warnings

When using the **–d** option, no effort is made to check that the disk written to is valid or that the position written to is sensible with respect to the directory written there. Caveat emptor.

Examples

To install a kernel compiled with the Sun 3 SunOS C compiler on the kernel boot partition of the host *danue*:

```
installk -d 1 sun mc68000 kernel /super/hosts/danue/vdisk:01
```

To install the same kernel as a bullet file in the directory */super/admin/kernel*:

```
installk sun mc68000 kernel sun3.pool
```

See Also

mkkdir(A), prkdir(A), reboot(A).

Name

ipsvr – the TCP/IP server

Synopsis

Built into the kernel. Add the following lines to the source file list of the kernel Amakefile:

```
conf_gw.c[flags="-I$SRC_ROOT/kernel/server/ip/kernel_dep" +
        "-I$SRC_ROOT/kernel/server/ip/amoeba_dep" +
        "-I$SRC_ROOT/kernel/server/ip"],
$K_SVR_TCPIP[flags="-I$SRC_ROOT/kernel/server/ip/kernel_dep" +
        "-I$SRC_ROOT/kernel/server/ip/amoeba_dep" +
        "-I$SRC_ROOT/kernel/server/ip" +
        '-DAM_KERNEL' + '-D_POSIX_SOURCE' ],
```

Description

The TCP/IP server implements the Internet protocols as described in RFC-791 ‘‘Internet Protocol’’, RFC-792 ‘‘Internet Control Message Protocol’’, RFC-793 ‘‘Transmission Control Protocol’’, RFC-768 ‘‘User Datagram Protocol’’, RFC-1256 ‘‘ICMP router discovery messages’’ and others. The rest of this document assumes familiarity with the basic concepts of TCP/IP networking.

The server runs in the kernel so that it can have access to the Ethernet interfaces. When the kernel is running the various capabilities of the the server are published in the directory *ip* under the kernel directory. A `dir -l` of a kernel with a TCP/IP server typically looks like:

```
bootp:00                        @    69564 KB
floppy:00                       @     1440 KB
ip                              %--1234
printbuf                        printbuf server
printer                         printer server
proc                            process/segment server
ps                              %--1234
random                          random number server
sys                             system server
tod                             TOD server
tty:00                          +
```

The contents of the *ip* directory is:

```
eth                             To---s--- 1
ip                              To---s--- 2
tcp                             To---s--- 3
udp                             To---s--- 4
```

These are the capabilities for the Ethernet, IP, TCP and UDP interfaces, respectively.

The default TCP/IP server elements for a user are defined in *ampolicy.h*. Typically they are the names in the following table:

Variable Name	Typical default value
ETH_SVR_NAME	/profile/cap/ipsvr/eth
IP_SVR_NAME	/profile/cap/ipsvr/ip
TCP_SVR_NAME	/profile/cap/ipsvr/tcp
UDP_SVR_NAME	/profile/cap/ipsvr/udp

These defaults can be overridden by either the environment variables below or in some cases by the command line options of the various programs that use the TCP/IP server.

Environment Variables

There are four string environment variables, each referring to one of the above server capabilities of the IP server: `ETH_SERVER`, `IP_SERVER`, `TCP_SERVER` and `UDP_SERVER`. The value of these environment variables is interpreted as a path name that refers to a capability of the TCP/IP server. To use the environment variables to override the default server, issue the following commands:

```
ETH_SERVER=/super/hosts/tcpip_host/ip/eth; export ETH_SERVER
IP_SERVER=/super/hosts/tcpip_host/ip/ip; export IP_SERVER
TCP_SERVER=/super/hosts/tcpip_host/ip/tcp; export TCP_SERVER
UDP_SERVER=/super/hosts/tcpip_host/ip/udp; export UDP_SERVER
```

where *tcpip_host* is the name of the host with the TCP/IP server to be used.

Initialization

In order to work properly, the TCP/IP server needs to have an Internet address, a netmask, and routing tables. These can be configured manually and the routing tables can also be configured automatically.

The Internet address and the netmask must configured manually using *ifconfig*(A). The netmask defaults to 255.255.255.0. The program *setupipsvr*(A) can be useful when configuring TCP/IP servers.

The routing tables can be configured using *add_route*(A). If the IP routers support the Internet Router Discovery Protocol (IRDP) and/or the Routing Information Protocol (RIP), for instance UNIX hosts with the *routed* daemon, then the *irdpd*(A) daemon can be used to automatically configure the routing tables.

Warning

Do not choose Internet addresses at random. The addresses should be obtained from the appropriate authority, especially if the Amoeba network is directly connected to the Internet.

Programming Interface Summary

Access to the *IP* server is provided by the library routines described in *ip*(L).

Administration

The TCP/IP server requires very little administration once it is installed. The main thing is to ensure that the *boot*(A) server maintains the capabilities for the default server in */super/cap/ipsvr*. There are commented-out entries in the *Bootfile* provided with the system that show how to do this. See `IP_cap`, `ETH_cap`, etc. for details. Note that the entries in */super/cap/ipsvr* should have Soap column masks ff:80:80. The *boot* server should also be used to keep related servers running that might be required, for example, *rarp*(A), *tftp*(A) and *irdpd*(A).

It will be necessary to maintain the hosts table *(/etc/hosts)* if the Domain Name System (DNS) is not accessible from the Amoeba network. Note that there is currently no DNS server that runs on Amoeba but the Amoeba programs will use any accessible DNS on the network.

N.B. If there is more than one network connected to the IP server host, the server will only talk to the network with the lowest FLIP network number.

Installation

This section describes the installation and configuration of the TCP/IP related kernels and utilities. The first issue is to decide on which host(s) to run TCP/IP servers. This will be determined by the intended use of the TCP/IP server(s).

The TCP/IP server is typically built into a *pool*, *workstation* or *smallbullet* kernel. It tends to take a lot of space, so building it into a *smallbullet* kernel is only a good idea if the Bullet Server host has sufficient memory to support both the Bullet cache and the IP server.

If it is desired to use an X server with a TCP connection to access other sites or UNIX hosts, then it is a good idea to have a TCP/IP server in the *workstation* kernel where the X server runs. This simplifies addressing the display of the X server from UNIX or another site.

The other main issue is booting hosts using TFTP. (This is typically the case with Sun machines.) At least one TCP/IP server must be able to boot without using TFTP, since the TFTP server under Amoeba needs the TCP/IP server. There are two possible solutions:

1. The first TCP/IP server is in an Amoeba kernel which is booted from disk/floppy. This requires only a small disk with one *vdisk* set up as a boot partition.
2. The kernel with the first TCP/IP server is booted from a TFTP server provided by UNIX.

Once the first TCP/IP server is running, *rarp*(A) and *tftp*(A) can be started by the *boot*(A) server and these can boot the remaining Amoeba processors. (The *boot* server is started on the Bullet Server which is normally booted from disk.) Note that the *rarp* and *tftp* servers typically use more than 1.5 MB and must run on the only machine guaranteed to be up, i.e., the TCP/IP host. Make sure it has enough memory free.

In the following it is assumed that one of the above mechanisms has been provided to boot the kernel containing the first TCP/IP server.

A TCP/IP kernel needs to to be told its Internet address and a netmask using *ifconfig*(A). This can also be done using *setupipsvr*(A) in combination with the *boot* server. See *setupipsvr*(A) for an example of how to set up the *boot* server to do this.

To check if a server knows its own Internet address use the command

```
ifconfig $IP_SERVER
```

It reports something like:

```
hostaddr= 130.37.24.11 netmask= 255.255.255.0
```

or (after a while):

```
hostaddr not set
```

If the Internet address has not (yet) been set then it can be set using *ifconfig* (as follows:

```
ifconfig -h 130.37.24.11 $IP_SERVER
```

with the appropriate address in place of the one shown).

Take a look at the netmask to see if it matches the netmask of the network the machine is on. The netmask can be changed with

```
ifconfig -n <netmask> $IP_SERVER.
```

If the *boot* server has been set up appropriately, the capabilities for the default TCP/IP server will have been installed in */super/cap/ipsvr* each time *tcpip_host* is rebooted. However, the first time it is necessary to install the capabilities manually. Do this using:

```
get /super/hosts/tcpip_host/ip/eth | put /super/cap/ipsvr/eth
get /super/hosts/tcpip_host/ip/ip | put /super/cap/ipsvr/ip
get /super/hosts/tcpip_host/ip/tcp | put /super/cap/ipsvr/tcp
get /super/hosts/tcpip_host/ip/udp | put /super/cap/ipsvr/udp
```

At this point you can use *ping*(A) to check if the server is working.

```
/super/admin/bin/ping tcpip_host
```

will give something like:

```
PING tcpip_host.am.cs.vu.nl (130.37.24.11): 56 data bytes
64 bytes from 130.37.24.11: icmp_seq=0 ttl=60 time=0 ms
64 bytes from 130.37.24.11: icmp_seq=1 ttl=60 time=0 ms
64 bytes from 130.37.24.11: icmp_seq=2 ttl=60 time=0 ms
```

Interrupt this if it seems to be working. It should now also be possible to use *ttn*(U) to talk to other IP hosts on the Amoeba network. Look up the Internet address of a UNIX machine on the same network and try

```
ttn <Internet address of UNIX machine>
```

this will give you something like:

```
connecting to <Internet address of UNIX machine>
connect succeeded !!!!


SunOS UNIX (xxx.cs.vu.nl)

login:
```

CTRL-D can be used to exit from the login prompt.

At this moment you can use TCP/IP to talk to hosts on the same network as your Amoeba IP server but probably not to talk to machines on other networks. Issue the command to display the routing table of the TCP/IP server:

```
pr_routes
```

If nothing is shown it indicates an empty routing table.

The TCP/IP server needs to know some gateways in order to forward packets that are to be sent to hosts on other networks. There are three ways for the TCP/IP server to get a routing table: built into the kernel, added manually using *add_route*(A), or discovered dynamically using the Gateway Advertisement ICMPs. The last method is the most appropriate one at moment. Unfortunately, current gateways do not generate Gateway Advertisement ICMPs and a daemon is needed to generate these ICMPs. Start the program *irdpd*(A) in the background:

```
irdpd -b -I $IP_SERVER -U $UDP_SERVER &
```

After about one minute rerun *pr_routes*(A). It should show some output. For example,

```
1 DEST= 0.0.0.0, NETMASK= 0.0.0.0, GATEWAY= 130.37.24.2, dist= 1 pref= 1
```

Connections to machines on other networks should now be possible.

Until now Internet addresses have been used instead of host names. There two ways to map a hostname to an Internet address or the other way around. The first one is the */etc/hosts* file. You can add the entry

```
127.0.0.1         localhost
```

to the file */super/unixroot/etc/hosts* and try

```
ttn localhost.
```

This should give the same result as

```
ttn 127.0.0.1
```

The other way is the DNS. The TCP/IP programs are compiled with the so-called resolver library. This means that a DNS server can be used to look up Internet addresses. The resolver library uses the file */etc/resolv.conf*. This file should contain one entry:

```
nameserver 127.0.0.1
```

When that Internet address is changed to the Internet address of a DNS server, and an entry

```
domain <your domain>
```

is added then a command like

```
host uunet.uu.net
```

should report:

```
uunet.uu.net has address 137.39.1.2
uunet.uu.net has address 192.48.96.2
uunet.uu.net mail is handled by relay1.UU.NET
uunet.uu.net mail is handled by relay2.UU.NET
```

Likewise

```
ttn <name of UNIX machine>
```

should work.

See Also

add_route(A), boot(A), ftpd(A), ifconfig(A), ip(L), irdpd(A), ping(A), pr_routes(A), rarp(A), setupipsvr(A), telnetd(A).

Name

irdpd – Internet router discovery protocol daemon

Synopsis

```
irdpd [-bds] [-U udp-device] [-I ip-device] [-o priority-offset]
```

Description

Irdpd looks for routers. This should be a simple task, but many routers are hard to find because they do not implement the router discovery protocol. This daemon collects information that routers do send out and makes it available.

At startup *irdpd* sends out several router solicitation broadcasts, as per RFC1256. A good router should respond to this with a router advertisement.

If a router advertisement arrives then no more solicitations are sent. The TCP/IP server has filled its routing table with the info from the advertisement, so it now has at least one router. If the advertisement is sent by a genuine router (the sender is in the table) then the *irdpd* daemon becomes dormant for the time the advertisement is valid. Routers send new advertisements periodically, keeping the daemon silent.

Otherwise *irdpd* will listen for RIP (Router Information Protocol) packets. These packets are sent between routers to exchange routing information. *Irdpd* uses this information to build a routing table.

Every now and then a router advertisement is sent to the local host to give it router information built from the RIP packets.

Lastly, if a router solicitation arrives and there is no router around that sends advertisements, then *irdpd* sends an advertisement to the requester. Note that this is a direct violation of RFC1256, as no host is supposed to send those advertisements. But alas the world is not always perfect, and those advertisements help booting hosts find routers quickly. (Of course, they will lose the router soon if they do not have an *irdpd* daemon themselves.)

Options

–b Broadcast advertisements instead of sending them to the local host only. This may be used to keep hosts alive on a net without advertisements.

–d Debug mode, tell where RIP and advertisements are coming from and where they are sent.

–I *ip-device*
Use the specified *ip-device* to send/receive IP packets.

–o *priority-offset*
Offset used to make the gateway's preferences collected from RIP packets look worse than those found in genuine router advertisements. By default –1024 .

–s Be silent, do not send advertisements to hosts that ask for them.

–U *udp-device*
Use the specified *udp-device* to send/receive UDP packets.

Warnings

Irdpd may help a host that should not be helped, i.e., if it does not have an *irdpd* daemon with RIP collecting trickery. It will make System Administrators pull out their remaining hair trying to find out why a host can access outside networks for a some time after boot, but goes blind afterwards.

Example

Typical usage is as follows, where the host *venus* is running a TCP/IP kernel.

```
HIP=/super/hosts/venus/ip
irdpd -b -I $HIP/ip -U $HIP/udp
```

See Also

RFC1256, ipsvr(A), rarp(A).

Name

isaboot – boot mechanisms for i80386 machines with an ISA bus architecture

Description

Booting an i80386 machine with an ISA (AT) bus architecture consists of multiple stages, depending on the type of the bootstrap mechanism used. How they work is described below.

When an ISA bus machine is turned on it starts executing the ROM-BIOS. ROM-BIOS performs a power-on system test (POST), and when the system functions correctly it will scan the memory between 0xC8000 and 0xE0000 for extra ROM images. These images start with a magic word (0x55AA) followed by their size. If POST finds a valid ROM image it will perform a *far* call and start executing the code within the ROM. Ordinarily, the code in the ROM image is just a small initialization routine which eventually returns, so that POST can continue searching for other ROM images. When all ROM images are processed POST will try to read the primary bootstrap loader from the first sector of floppy disk unit 0, and place it in memory at address 0x7C00. If this read succeeds and the bootstrap loader is valid, POST will jump to it. However, when the read fails or when the loader is invalid, POST will try to read a bootstrap loader from the first sector of hard disk unit 0. If this also fails, POST will display an error message and halt the system. The primary bootstrap loader for the hard disk determines which of the four partitions is active and loads the secondary bootstrap loader associated with it.

The primary bootstrap loader for the floppy disk and the secondary bootstrap loader for the hard disk offer the same functionality (except of course that the former reads from floppy disk, and the latter from hard disk). With this bootstrap loader it is possible to boot one of the kernels from the kernel directory (see *mkkdir*(A) and *prkdir*(A)). To get an overview of the kernels in the kernel directory you should type a question mark at the

```
boot:
```

prompt. To start a kernel you type its name followed by a carriage return. You can start the default kernel, namely the first in the kernel directory, by just typing a carriage return or waiting for about thirty seconds.

Note that it is also possible to give one or more options to the kernel to be booted. These should appear after the name of the kernel to be booted and are of the form

-*option*:*value*

where *option* is the name of the option to be selected and *value* is the value that option should take. For example,

```
boot: smallbullet -noreboot:1
```

will cause the *smallbullet* kernel to be booted with the option *noreboot* set to the value 1. This means that if the kernel halts (perhaps due to an assertion failure) that the machine will not attempt to reboot itself. This can be very useful when tracing kernel bugs.

N.B. There is also a program called *isa.dosboot* which can be found in the directory */super/admin/kernel* which can be installed in the Amoeba bootstrap directory along with the

usual Amoeba kernels. This program when booted will boot the first DOS partition on the hard disk. Therefore if there is also a DOS partition on the disk, by selecting the *isa.dosboot* pseudo program, DOS can be booted instead of Amoeba.

Floppy disk boot

The floppy disk bootstrap loader is installed in the first 17 sectors of the floppy disk. It should be placed there with *installboot.ibm_at* (see *installboot*(A)). The floppy disk bootstrap loader will only boot kernels that are stored on the floppy disk itself. The bootable kernels are specified within the kernel directory which is located at physical sector 18 of the floppy disk.

Hard disk boot

The hard disk bootstrap loader is placed in the first track of the active Amoeba partition using *installboot.ibm_at* (see *installboot*(A)). The hard disk bootstrap program will only boot kernels that are stored on the hard disk. The kernels are stored on a *vdisk*(A) of which the first block contains the kernel directory. The bootstrap loader determines which vdisk is used, but by convention kernels are usually stored on vdisk:01.

Example

Booting the coldstart Amoeba floppy proceeds along the following lines:

```
Amoeba 5.3 Standalone Boot Program

Default kernel: coldstart (just press <ENTER> or wait ~30 seconds)

boot:
```

To get an overview you should type a single question mark (‘?’). This will display a list of available kernels, the values between the parenthesis denote the offset and the size (in blocks) of the kernel.

```
boot: ?
Bootable kernel(s):
    coldstart (1:399)
    workstation (400:499)
    bullet (900:699)
```

You can start any kernel by just typing its name followed by a carriage return. An exception to this mechanism is the default kernel which is booted if no name is given (just press the return key), or when there has been no keyboard activity for approximately thirty seconds.

See Also

fdisk(A), installboot(A), isamkimage(A), isamkprom(A), mkkdir(A), prkdir(A).

Name

isamkimage – convert an 80386 kernel image to an ISA bootable image

Synopsis

```
isamkimage [-v] [-s pad] kernel rom-image
```

Description

Isamkimage converts a kernel executable image for the Intel 80386 ISA-based machines into a bootable image. This program should only be used in conjunction with the RARP/TFTP bootstrap loader PROM.

Options

–v This option causes the program to be more verbose than it usually is.

–s pad This option is used to specify the padding size.

Example

```
isamkimage -v workstation/kernel /super/admin/tftpboot/C01FE745.iX86
```

See Also

isaboot(A), tftp(A).

(i80386/ISA ONLY)

Name

isamkprom – convert an 80386 binary image into an ISA boot ROM image

Synopsis

```
isamkprom [-v] [-s size] binary rom-image
```

Description

Isamkprom converts an executable image into a rommable boot image. It is used to create an Amoeba boot ROM image for the Intel 80386 machines based on the ISA bus architecture.

Options

–v This option causes the program to be more verbose than it usually is.

–s size This option is used to specify the ROM size. The size is specified in number of kilobytes and should be a power of two with a maximum of 64. The default ROM size is 64 Kb.

Diagnostics

The program will not convert the binary image if it does not fit into the specified ROM size.

Warning

Not every binary can be converted into a meaningful ROM image.

File Formats

Isamkprom manipulates the first 6 bytes of the text segment of the kernel which have the following meaning: The first two bytes contain a magic number, the third byte the length of the ROM image (in 512 byte blocks), and the fourth and fifth byte the opcode for a jump instruction. The sixth byte contains a checksum, Apart from the first two and the jump opcode bytes all these fields are generated by *isamkprom*.

Example

```
isamkprom -v -s 16 prom.ne2100 prom.ne2100.16Kb
```

See Also

isaboot(A).

Name

kerneltree – build a configuration tree for Amoeba kernels

Synopsis

```
kerneltree [-v] conf-root src-root template-root arch mach toolset
```

Description

Kerneltree is used to construct the configuration trees for building Amoeba kernels. This program will only function correctly if the *amoebatree*(A) program has been executed. Note that *kerneltree* does not perform the actual build but merely creates the hierarchy with the necessary *toolset* and *Amakefiles* to build the specified kernels. The configuration system has been designed so that versions for different architectures can be easily managed and that cross-compilation is straight-forward.

The configuration is constructed by cloning a template configuration tree which contains the directory structure and necessary *toolset* and *Amakefiles*. If the target configuration tree specified does not exist it will be created. If it already exists, then it will be brought up to date. Note that any parts that are no longer in the template tree will not be deleted. If any file in the configuration is different from the corresponding file in the templates then the program will enquire as to whether it should overwrite the file or not. If you respond by typing *q* then no change will be made and the program will terminate. If you answer with *d* then you will be shown the differences and asked again. If you type *y* the change will be made. Any other response will be interpreted as ‘‘do not overwrite but continue’’.

The *conf-root* specifies the place where the configuration should be built. If it is not an absolute path name then the output of the *amdir*(U) command will be prepended to it. Otherwise it will be used as is.

The *src-root* specifies the source code from which the system is to be built. If it is not an absolute path name then the output of the *amdir*(U) command will be prepended to it. Otherwise it will be used as is.

The *template-root* specifies where to get the templates for the *Amakefiles*. If it is not an absolute path name then the output of the *amdir*(U) command will be prepended to it. Otherwise it will be used as is. The *Amakefiles* will be parameterized with the *conf-root*, *src-root* and the target architecture for which you are compiling.

The *arch* argument specifies the target architecture for which you are compiling.

The *mach* argument specifies the machine for which the Amoeba kernel(s) should be constructed. For a list of known machines, look at the entries in the directory *templates/kernel* in the distribution tree.

The *toolset* argument specifies which compiler, loader, archiver, etc. are to be used to build the system. Under Amoeba this is typically *ack*(U). Under UNIX this will be one of the compilers available with the UNIX system. All the tools will be placed in a directory in the configuration tree and included from there by the *Amakefiles*.

Options

–v The verbose option. This causes the program to print details of everything it is doing. It is very noisy.

Example

```
kerneltree /amoeba/conf/amoeba /amoeba/src /amoeba/templates \
                  i80386 ibm_at ack
```

will create under the directory */amoeba/conf/amoeba/i80386.ack/kernel/ibm_at* a tree structure containing all the Amakefiles to build the various Amoeba kernels for the i80x86 machines.

See Also

amake(U), amoebatree(A), build(A).

Name

killfs – destroy a bullet file system

Synopsis

```
killfs [-r] disk_cap
```

Description

Killfs assumes that the virtual disk specified by *disk_cap* contains a valid Bullet or Group Bullet file system. The superblock of a Bullet file system is on block 0 of the vdisk. Without options it destroys the file system by invalidating the superblock. If the **–r** option is specified it restores the validity of the superblock by undoing the previous *killfs* operation. (NB. It cannot restore a file system damaged in any other way.)

Warnings

This is a catastrophic action in that it will eliminate a file system (although the effect may not be noticed until the file server is rebooted). Therefore it is a bad idea to let just anyone have access to the Bullet Server hosts, since they can easily get the virtual disk capability for the Bullet Server from the kernel directory.

Example

```
killfs /super/hosts/bullet1/vdisk:02
```

will annihilate the Bullet file system on the virtual disk *vdisk:02* of the host *bullet1*.

```
killfs -r /super/hosts/bullet1/vdisk:02
```

will restore the file system.

See Also

bullet(A), gbullet(A), mkfs(A).

Name

kstat – examine the state of a running Amoeba kernel

Synopsis

```
kstat -flags hostname
```

Description

Kstat is used to examine the internal tables and state variables of a running Amoeba kernel. If *hostname* is not an absolute path name it is looked up in the public hosts directory as defined by the variable `HOST_DIR` in the file *ampolicy.h*. (This is typically */profile/hosts*.) If the specified host is down, *kstat* will take a short while to determine this and then report the error ‘‘not found’’. If the path name given was not for an Amoeba host then *kstat* will determine this more quickly but also report the error ‘‘not found’’.

Not all kernels support all flags. If *flags* is **?** then *kstat* will print a list of flags for the internal tables the specified host contains. Otherwise you may give a set of one or more letters selecting the desired kernel tables. These will be printed in the order specified.

Example

```
kstat -? bullet1
```

prints the following list of tables available on the host *bullet1*.

```
C  flip rpc dump
E  Ethernet flow control statistics
F  flip routing table
I  flip interface
N  flip network dump
S  dump sweeper info
b  dump random seed bit info
c  flip rpc statistics
e  dump ethernet statistics
f  flip fragmentation dump
i  flip interface statistics
k  flip rpc kid dump
l  dump Lance statistics
m  dump mpx table
n  flip network statistics
p  flip rpc port dump
s  dump segtab
u  print uptime
v  print version
```

and the command

```
kstat -u bullet1
```

prints the uptime of the Amoeba running on the host *bullet1*.

```
249488 seconds up
```

See Also

printbuf(A).

Name

loadflipdriver – install and initialize the FLIP loadable device driver for SunOS 4

Synopsis

```
loadflipdriver
```

Description

This script is used to load the loadable FLIP driver for SunOS 4 using the SunOS command *modload*. The driver implements the FLIP protocol in the SunOS 4 kernel. It then creates in */dev* the device necessary to access the FLIP driver. It then attempts to initialize the device by executing the *flip_stat*(A) command.

NB. It is necessary to be super-user to execute this command successfully.

This command should be executed each time the UNIX host is rebooted. It is customary to put it in */etc/rc.local* as part of the bootstrap process, once it is clear that it works properly.

Diagnostics

The *modload* command should print the module id when the FLIP driver is successfully loaded.

Files

/dev/flip_ctrl – the device used by the command *flip_ctrl*(A).
/dev/flip_ip – the device used by the wide-area FLIP system.
*/dev/flip_rpc*n – (where n is a number) - the devices used for doing Amoeba RPCs.

Warnings

If an attempt to load the FLIP driver fails it is possible to rerun this command safely. It is not a good idea to try to load the FLIP driver if it has already been successfully loaded.

See Also

amdir(U), flip_ctrl(A), flip_stat(A).

Name

login – login utility

Synopsis

```
login [-bdw] [username]
```

Description

Login lets you log in on Amoeba. When *login* is started it must have unrestricted access to the directory */super/users*. If *username* is not specified it prints a banner which shows the version of Amoeba and then prompts for a user name and a password. If *username* is specified then it only prompts for a password. The user name must be a directory entry in */super/users*. Therefore a user name cannot be empty. Once the user name is entered *login* turns off echoing and asks for a password. An empty password may be valid. Then it creates a rudimentary environment, executes the any commands in the *environment* file */Environ* – which involves validating the password – and finally starts a shell. To its owner process, *login* appears to exit when the shell it started exits.

Options

–b This option causes *login* not to print its banner.

–d This option causes *login* to print debug information about what it encountered in the environment file.

–w This option causes *login* to suppress warning messages.

Diagnostics

Login reports an invalid login when the user does not exist, has no *Environ* file or the password is incorrect. If there is an error while evaluating the *Environ* file then the default environment will be used and warning messages will be printed.

Environment Variables

The following capability environment variables are set by *login*:

`HOME`
: What the user's shell will see as ''/'', i.e., what *login* sees as */super/users/<username>* .

`WORK`
: The initial working directory of the new shell.

`STDIN, STDOUT, STDERR, TTY`
: These point to a tty. They refer to the shell's standard input, output, diagnostics output, and the controlling tty respectively.

The following string environment variables are set by *login*:

`HOME`
: Absolute path name for the home directory. Defaults to ''/home''.

USER

LOGNAME

The login name of the user. Both variables are supported so that software from the different versions of UNIX has a chance of running).

The following variable is used by *login*:

SHELL

This variable is not set by *login* unless it is defined in the *environment* file. If it is set in the *environment* file, *login* will invoke this shell instead of the default */bin/sh* if no *exec* line is defined in */Environ*.

Examples

As a first example, consider someone who only wants to be able to login:

```
passwd 55PLc3q.TynSko
```

Now consider a session server user (see *session*(U)), who wants start his own shell in a directory called */home*. In case a window environment exists, he wants the session server to start *xterm*, which in turn starts the shell. If there is no window environment, *xterm* will not be started. Note that putting *xexec* beyond *exec* is not useful. The *xexec* will be ignored in that case.

```
passwd 55PLc3q.T&nSko
setenv HOME /home
setcap WORK /home
setenv _WORK /home
setenv PATH .:/home/bin:/bin:/profile/util:/profile/module/x11/bin
setenv SHELL /bin/sh
xexec /profile/util/session -a /bin/xterm -reverse -e /bin/sh
exec /profile/util/session -a /bin/sh
```

Files

/Environ – the default environment file.

File Format

Comments in the *Environ* file start with a ‘‘#’’ in the first column and extend to the end of the line.

The commands recognized by *login* are:

passwd	Validate typed password with the crypted version.
setenv	Set the value of an environment variable.
setcap	Set the value of a capability variable.
exec	Execute a shell.
xexec	Execute a shell if and only if running in a window environment.

Passwd takes at most one argument: an encrypted password. If the argument is missing, an empty password will be accepted by *login*. *Setenv* and *setcap* take two arguments: the first

is the name of the variable, the second is the value (a path name to be looked up in the case of *setcap*). The arguments to *exec* form the command to be executed. The initial owner of this process is the original owner of *login*. *Exec* does not return. If the *environment* file does not contain an *exec*, an implicit *exec* of the default shell is performed.

See Also

chpw(U).

Name

makeconf – construct a new Amoeba configuration

Synopsis

```
makeconf [-a|-u] confdir arch toolset mach1 [mach2 ...]
```

Description

Makeconf can be used to generate a complete Amoeba configuration for the architecture and machine types specified using the specified compiler tools. If no options are given it will make both the UNIX and native Amoeba binaries and kernels plus the printable version of the documentation. Note that the X binaries and MMDF binaries for Amoeba are also made. The former consumes a significant amount of disk space. The FLIP driver for UNIX is not made. This must be done separately.

If the *confdir* argument is not an absolute path name then the output of *amdir*(U) will be prepended to it. Otherwise it is used as is. The UNIX configuration will be made in the sub-directory *unix* of the *confdir* and the Amoeba configuration will be made in the sub-directory *amoeba*. The commands *amoebatree*, *doctree*, *kerneltree* and *unixtree* are used to construct the configuration trees. The source code and Amakefile templates are always taken from *`amdir`/src* and *`amdir`/templates* respectively. The command *build*(A) is used to compile everything.

NOTE WELL: *Makeconf* does not update an extant configuration. If a configuration is found for the given architecture and toolset at the place specified by *confdir* it will be removed before the new configuration is made.

Options

–a This option cause only the Amoeba binaries, kernels and documentation to be made. The UNIX binaries are not made.

–u This option cause only the UNIX binaries and documentation to be made. The Amoeba binaries are not made.

Example

```
makeconf conf sparc sunpro sun4m sun4c
```

will make the UNIX and Amoeba configurations in the directories *`amdir`/conf/unix/sparc.sunpro* and *`amdir`/conf/amoeba/sparc.sunpro* respectively and build the documentation in *`amdir`/conf/doc/ref_manual*. The sources and templates used for the build will be taken from *`amdir`/src* and *`amdir`/templates* respectively.

See Also

amdir(U), amoebatree(A), build(A), doctree(A), kerneltree(A).

Name

makesuper – create Soap super files

Synopsis

```
makesuper [-f] [-c cap] bullet0 bullet1 vdisk0 vdisk1 size
```

Description

Makesuper is used to create new super files for Soap, the Amoeba directory server. It also creates an initial directory that can be used as ‘‘root’’ of a new Soap directory graph.

The arguments are the following:

bullet[01] These are the capabilities for the bullet servers that the Soap Servers should use to store their directories. If Soap is going to be used as a duplicated directory service, they should be two different Bullet Servers in order to achieve fault tolerance. If the directory service is not going to be duplicated, argument *bullet1* can be specified as ‘‘–’’.

vdisk[01] These are the capabilities for the virtual disk servers used to store the super files of Soap 0 and Soap 1 respectively. If the directory service is not going to be duplicated, argument *vdisk1* must be specified as ‘‘–’’.

size This argument specifies the size of the super file in blocks. The minimum size allowed is 3; the maximum size depends on the number of blocks of the virtual disk partition. The actual size depends on how many directories are likely to be needed. Each 512 byte block can store the capabilities for 16 directories, so for 3200 directories you will need 3200/16 = 200 blocks.

Under UNIX, the capability for the initial directory is stored by default in the file ‘‘`.capability`’’ in the user’s home directory (taken from environment variable `HOME`). If *makesuper* is executed under Amoeba, the new directory is published as ‘‘`new_root`’’ in the current working directory. Note that *makesuper* refuses to overwrite an already existing capability; the **–c** option (see below) can be used to specify an alternative name.

Makesuper also creates files containing the capability-sets for the private ports of the new Soap Servers. These private ports are used in the duplication protocol and for administrative commands. The capability-sets are stored as files in the current directory, with names ‘‘`super_cs.0`’’ and (in case Soap is duplicated) ‘‘`super_cs.1`’’. They should be published under Amoeba (typically in directory */super/cap/soapsvr*) using *put*(U).

The virtual disk servers and Bullet Servers used by the Soap Servers may in principle be located on any host of the system. However, to achieve good performance it is advisable to run each Soap Server on the host on which both its virtual disk server and its preferred Bullet Server are located.

For bootstrapping purposes, *makesuper* can also be used without the help of a directory server. In this case, files containing the capabilities for the required hosts should be created using *mkhost*(A). The Bullet and virtual disk capabilities can then be specified by a path relative to the corresponding host name (see the *Examples* section below). The host capabilities are looked up with *super_host_lookup*. (See *host_lookup*(L) for details.)

Options

–c cap — This tells *makesuper* to store (or publish) the capability for the initial directory under the name *cap*.

–f — By default, *makesuper* refuses to overwrite a virtual disk that already contains a Soap super file. The **–f** option can be used to override this check.

Warnings

Makesuper makes no effort to protect you from destroying a non-Soap virtual disk partition. The **–f** flag only checks to see if the disk already has a Soap super file on it.

When in doubt, use the command *dread*(A) to fetch the first block of the virtual disk partition. This (binary) block can then be examined using *od*(U).

Examples

The command

```
makesuper h0/bullet h1/bullet h0/vdisk:03 h1/vdisk:03 200
```

creates the super files for a duplicated Soap Server, to be run on hosts *h0* and *h1* respectively. Under UNIX, this could give the following diagnostics:

```
Soap 0 super capset stored in "./super_cs.0"
Soap 1 super capset stored in "./super_cs.1"
Created new capability file "/usr/joe/.capability"
Writing super file #0...
Writing super file #1...
New Soap System ready.
```

To create the super file for a single copy mode Soap Server, storing the capability in ‘‘new_soap’’, specify the following:

```
makesuper -c new_soap host2/bullet - host2/vdisk:03 - 200
```

See Also

dread(A), host_lookup(L), mkhost(A), put(U), soap(A).

Name

makeversion – generates a version number string and compiles it

Synopsis

```
makeversion [filename]
```

Description

Makeversion is a shell script which creates a C source file and the corresponding object file. The source file contains a single array called *version* which is initialized with a string containing information to identify a version of a program. The *filename* argument specifies the name of the output file to be produced. The default is *version.o*.

The intermediate C source file is used to calculate the version string. This file has the same basename as the object file but with suffix *.c*. The contents of this file is a string of the form:

```
char version[] = "vnum by user@host (date)\nin directory pwd";
```

where *vnum* is an integer representing the version number. If the intermediate C file does not exist it is created with version number 1. If it does exist the version number in it is incremented by one and the relevant information for the rest of the string is replaced with the new values.

Environment Variables

`CC` selects the C compiler to be used. The default is */bin/cc.*

Example

```
makeversion joe.o
```

will create the file *joe.c* and compile it to produce *joe.o*. If *joe.c* did not initially exist then the newly created *joe.c* might contain the following:

```
char version[] = "1 by gregor@sharpie (Thu May 31 15:02:54 MET DST 1990)\n
in directory /home/conf/amoeba/sparc.gnu-2/kernel/sun4m/pool";
```

If it did exist then the version number will be greater than 1 and the current date will replace the original date.

Name

mkbootpartn – make a bootable floppy or hard disk partition

Synopsis

```
mkbootpartn [-v] [-o offset] [-f flboot] [-C confdir]
          arch mach toolset vdisk kernel ...
```

Description

Mkbootpartn is used to construct a floppy or virtual disk from which an Amoeba kernel can be booted. In the case of the floppy, a secondary bootstrap (see *installboot*(A)) is installed on the disk. A bootable hard disk should already have a secondary bootstrap. If not, use *installboot*(A) to install one. Thereafter, in both cases, a kernel directory (see *mkkdir*(A)) and one or more kernels which can be booted are installed.

The kernels to be installed on the disk are specified on the command line. The *kernel* arguments, if not absolute path names, specify the names of kernel directories in the configuration tree. For example, if the kernel name *bullet* is given as an argument then the file *`amdir`/conf/amoeba/*`arch.toolset`*/kernel/*`mach`*/bullet/kernel* will be put onto the disk. If *kernel* is an absolute path name then the name of the kernel on the disk can be specified with an optional name as *path-name:kernel-name*. If the *kernel-name* option is omitted the kernels on disk are numbered from 0 as *kernel.*`n`, where n is an integer.

It is also necessary to specify the *vdisk* where the bootable partition is to be made. It can be either a floppy or a virtual disk. It is also necessary to know the *arch*itecture (e.g., i80386), target *mach*ine type (e.g., ibm_at or sun3) and the compiler *toolset* used to make the kernel so that the kernel can be correctly processed. For example, it is necessary to strip kernels made with *sun* compilers for Motorola 680x0 machines. The information is also necessary to determine which type of bootstrap loader should be put onto the disk.

Options

–C *confdir*
: Allows the default configuration directory *`amdir`/conf/amoeba* to be overridden by *confdir*.

–f *flboot*
: This tells *mkbootpartn* to use the file *flboot* as the floppy secondary bootstrap instead of the default in the configuration tree. The standard secondary bootstrap is also normally kept in */super/admin/kernel/isa.flboot* under Amoeba, so if there is no configuration tree available then this path name can be given with the **–f** option.

–o *offset*
: Start writing at disk block number *offset*. The default is 0.

–v Switch off verbose mode. Default is verbose on.

Diagnostics

Other than the usage message, the following are the main messages produced by *mkbootpartn*.

`Amoeba Configuration Root argument expected`

This means that the *confdir* argument for **–C** is missing.

`no configuration directory` *confdir*

The configuration directory *confdir* (either specified with the **–C** option or the default) does not exist.

vdisk `is not a vdisk`

The specified *vdisk* is not a virtual disk object.

`Kernels need more blocks (nnn) than available (mmm)`

The kernels specified are too big to all fit on the virtual disk. The total number of blocks needed is *nnn* while only *mmm* blocks are available on the disk. Note that *nnn* also includes the space needed for the bootstrap loader and the kernel directory.

`Cannot install floppy bootstrap loader`

The *installboot* program failed while trying to install the floppy bootstrap loader.

`Cannot make kernel directory`

The attempt to write the kernel directory to disk failed.

`Cannot store kernel` *name*

The attempt to write the kernel called *name* to disk failed.

`Don't know how to install a` *arch/toolset* `kernel`

The combination of *arch*itecture and *toolset* is unknown to the *mkbootpartn* command. It does not know if any stripping or *a.out* conversion is required.

Files

/super/admin/kernel/isa.hdboot or
$CONFIG/admin/bootblocks/ibm_at/isa.hdboot – i80386 hard disk bootstrap loader for installboot.
/super/admin/kernel/isa.flboot or
$CONFIG/admin/bootblocks/ibm_at/isa.flboot – i80386 floppy disk bootstrap loader for installboot.

Where `$CONFIG` is the default configuration directory or the one specified by the **–C** option, concatenated with the `arch.toolset`.

Warnings

Overwriting an extant bootable partition with kernels that do not function will require that an alternative boot medium be used to restart the machine. Test kernels before installing them as the bootable version on disk.

File Formats

The kernel directory structure and layout of the kernels on disk is described in *mkkdir*(A).

Example

The following makes a bootable floppy for an i80386 ISA bus system containing a *bullet* and a *pool* kernel. It writes it to the floppy drive on host *xxx*.

```
mkbootpartn i80386 ibm_at ack /super/hosts/xxx/floppy:00 \
        /home/conf/bullet/kernel:bullet /home/conf/pool/kernel:pool
```

See Also

amdir(U), dwrite(A), installboot(A), mkkdir(A).

Name

mkfs – make a Bullet file system

Synopsis

`mkfs [-f] [-b` $\log_2$`(blocksize)] [-i num_inodes] diskcap`

Description

Mkfs makes a Bullet file system on the virtual disk specified by *diskcap*. If there is a Bullet file system already present on the partition and the **–f** option was not specified then it will print a warning and stop. Otherwise, it determines the size of the disk by asking the disk server. It then calculates the number of blocks it will need for inodes, but rounds up the number of inodes to fill the last inode block completely. It then prints a short summary of what it is about to do. Immediately after that it begins to modify the disk. Firstly it writes empty inodes to the disk, beginning at block 1 and then lastly it writes the superblock on block 0 of the disk.

Options

–b $\log_2$(blocksize) It is possible to specify the size of the disk block that the Bullet Server will use. To enforce that it is a power of 2, the $\log_2$ of the block size is given. The block size must be greater than or equal to the size of the physical disk block of the underlying device driver. That is typically 512 bytes, which is equivalent to **–b**9. There is an implementation defined maximum which can be discovered by giving an illegal value for the block size (such as 0).

–f This forces an existing Bullet file system that is already on the virtual disk to be (irretrievably) overwritten.

–i num_inodes It is possible to force a particular number of inodes for a file system. The default is the number of physical blocks in the file system divided by the constant `AVERAGE_FILE_SIZE`. (which is a tuning parameter of the Bullet Server defined in the file *bullet.h* and is set to 16K bytes in the original distribution.) Note that the number of inodes specified in this option will be rounded up to fill the last disk block allocated for inodes.

Diagnostics

These should be self explanatory. Errors from *mkfs* itself (rather than the calling program) are of the form:

error message mkfs: b_mkfs failed: *error status*

where the *error status* message reflects the actual error code returned.

Warnings

Mkfs makes no effort to protect you from destroying a non-Bullet file system. It will protect extant Bullet file systems unless the **–f** option is used.

Example

```
mkfs -b 10 /super/hosts/torro/vdisk:01
```

will print

```
Blocksize = 1024 bytes
Sizeof disk = 585329 blocks
Number of inodes = 119488
No. blocks for inodes = 1867
```

and then proceed to write 1867 blocks of empty inodes to the disk, followed by the superblock.

See Also

gb_mkfs(A), killfs(A).

Name

mkhost – make host capability

Synopsis

```
mkhost pathname Ethernet-address
  or
mkhost pathname get-port check-field
```

Description

Mkhost creates a capability for a processor. The capability is stored at the directory server under the given *pathname*, unless *pathname* is a single dash, in which case the capability is written to standard output. When run under UNIX and *pathname* is a single dash, the program is independent of the directory server.

A host's capability is typically generated by giving the Ethernet address for the host. This is used for both the *getport* and the check field of the capability. The Ethernet address must be given as 6 two-digit hexadecimal numbers separated by colons or dashes (consistently). The disadvantage of using the Ethernet address is that it is predictable and anyone can obtain access to a host's directories simply by looking up its Ethernet addresses in the */etc/ethers* file. On systems where it is possible to store secret numbers in NVRAM it is possible to allow hosts to have secret capabilities. The system-specific installation instructions describe how to do this, and when it is possible. In this case the host capability can be specified using the second from described above.

Examples

To store the capability for host *bcd* in the directory server:

```
mkhost /super/hosts/bcd 01:20:4a:2c:1a:f4
```

To write it to a file:

```
mkhost - 01:20:4a:2c:1a:f4 > /usr/amoeba/adm/hosts/bcd
```

Name

mkkdir – make a kernel bootstrap directory for a disk

Synopsis

```
mkkdir
```

Description

Mkkdir makes a special bootstrap directory which is used by the secondary bootstrap loader of Amoeba. This directory contains details of where to find a kernel to boot and how big it is. *Mkkdir* reads a table describing the directory from *stdin* and writes the directory to *stdout*. It contains the start block (which must be greater than zero), the number of blocks allocated for the kernel and the name of the kernel. The format of the table is described below. The kernels must be installed later at the addresses specified in the table. This can be done using *dwrite*(A) (although see *installk*(A) and *mkbootpartn*(A)).

If invalid input is given it will stop processing, a warning will be printed on *stderr* and nothing will be written to *stdout*.

Warnings

If valid but stupid input is given it will not be detected. For example, a directory entry that permits a kernel of only ten blocks is acceptable but will not work if the kernel is actually larger. Similarly, a kernel entry that extends beyond the disk will not be detected. If such a directory is created on a disk then it may no longer be possible to boot Amoeba from disk. In that case it will be necessary to boot from tape, floppy or the network.

File Formats

The input for a kernel directory consists of lines. Each line consists of three fields separated by tabs. The first field is the start block of the kernel on the disk. This must be greater than zero. Block 0 holds the directory information. The second field contains the number of blocks allocated on the disk for storage of the kernel. The third field is a symbolic name by which the kernel can be selected at bootstrap time.

Note that no checking is done for overlapping directory entries or directory entries that do not fit on the disk.

Example

In the following example we create two directory entries on the virtual disk for kernels on the host named *bullet0*. The command *dwrite*(A) will write the kernel directory to block 0 of the specified virtual disk. The table is read from standard input. Therefore we type a CTRL–D at the end of the input.

```
mkkdir | dwrite /super/hosts/bullet0/vdisk:01
1       800     amoeba
801     800     test
^D
```

Each kernel has been allocated 800 blocks (each 512 bytes, thus 400 Kbytes). The first is

found at block 1 and has the symbolic name *amoeba*. The second is found at block 801 and has the symbolic name *test*. It is important that when installing the kernels that they indeed be written to the correct place on the disk.

See Also

dread(A), dwrite(A), installk(A), mkbootpartn(A), prkdir(A), vdisk(A).

Name

multivol – multi-volume (floppy disk) dump utility

Synopsis

```
multivol [-q] [-b n] [-s n] [-n name] disk-size disk-capability
readvol [-l] [-q] [-b n] [-s n] [-n name] disk-capability
```

Description

Multivol reads the standard input and copies it onto multiple (floppy) disk volumes. Each disk volume starts with a volume header followed by disk-size – 1 disk blocks of data. The volume label contains a volume name, current and successor's volume number (if any), and a CRC check sum to detect damaged data.

Multivol will always ask the user for confirmation before it starts writing data onto the disk volume. The confirmation questions are preceded by a user attention signal. This feature may be disabled using the **–q** option.

Readvol reads an ordered set of volumes with the same name, and writes the data contents to standard output. If the input was block aligned, the output is an exact copy of the input of *multivol.*

Readvol enforces an ordered set of input volumes by starting at volume number 0, and reading the successor mentioned in the current volume header. If the present disk volume does not have the expected volume number or volume name, *readvol* will complain and make another request for the expected volume.

Options

–l This option is only available with *readvol.* It causes *readvol* to read only the volume label of a disk and display its contents. This can be used to obtain the name and volume number from a disk which has no sticky label identifying it.

–q Quiet operation mode. Under normal circumstances *multivol* and *readvol* will issue some kind of user attention signal (often an alarm bell) to get attention. This option disables this feature.

–b n Specify the internal buffer size used by *multivol* and *readvol.* The default, 16 KB, will often suffice, but during coldstart it may be necessary to reduce this value. The buffer size is specified as the number of 512 byte blocks.

–s n Specify the start volume number. This option is of dubious value.

–n name Specify the volume name. *Multivol,* as well as *readvol,* use the name ''AMOEBA'' as default label name.

Warnings

The granularity of the volume programs is in disk blocks. If the input stream is not exactly block aligned the *multivol* program will pad it with random bytes, these bytes will, unfortunately, also appear on the output stream as though they were part of the input. Ordinarily this does not pose much of a problem, since the input of *multivol* typically

consists of *tar*(U) file images. However, it might be inconvenient for other uses of this program.

Examples

Multivol is used to create the kernel image floppies for the 80386/AT coldstart process. After a bootable coldstart image is built on a ramdisk kernel, the following command is issued to save it onto two 3.5 inch floppy disks:

```
cd /super/hosts/bullet
dread -n 4878 bootp:00 | multivol -n "RAMDISK-IMAGE" 2880 floppy:00
```

The Amoeba coldstart kernel has a built-in stripped down version of *readvol*, which is capable of reading coldstart images.

To store a large tar image on multiple floppy disks the following command could be used:

```
tar cvBf - . | multivol -s "AMOEBA" 2880 /super/hosts/bullet/floppy:00
```

On a fully operational system, multi-volume disks may be extracted using the following command. In fact, this command is used to unpack the floppy disk distribution set:

```
cd /super
readvol /super/hosts/bullet/floppy:00 | tar xvBf -
```

Name

newsoapgraph – generate a standard soap directory structure

Synopsis

```
newsoapgraph username
```

Description

Newsoapgraph is a command script used to create the standard directory structure. It is only sensible for it to be done immediately after the creation of the first soap directory (see *makesuper*(A)). The initial directory will be the creator's home directory and is known as /. (NB. all users see their home directory as having the name /.) *Newsoapgraph* will create the super directory under it. This is the capability that gives all rights to the entire directory graph and should be protected. Thereafter the directory */super/users* is created and then the initial directory is entered as */super/users/*username. This is the first of many cycles in the directory graph. Thereafter several subdirectories of */super* are generated. Users typically get access to a subset of these via the */profile* directory.

Diagnostics

Diagnostics are those of *mkd*, *chm*, *std_restrict*, *put* and *get*. There should not be any, however.

Environment Variables

The environment variable SPMASK influences the behavior of *put* and *mkd* but *newsoapgraph* explicitly sets the value of SPMASK for the duration of the command to ensure the correct protection for the directories that it creates.

Example

```
newsoapgraph peter
```

will create a new soap directory graph and register the initial directory under the name */super/users/peter*.

See Also

chm(U), get(U), makesuper(A), mkd(U), put(U), std_restrict(U).

Name

newuser – generate the home directory structure for a new user

Synopsis

```
newuser username o|g|go -b bulletsvr -g group1 [-g group2 ...]
        -p pooldir [-u unixroot|-n]
```

Description

Newuser creates the directory structure required for a new user. The new directory structure is added under */super/users/*`username`. *Username* has a maximum length of `MAX_LOGNAME_SZ` as defined in *ampolicy.h* (typically 20 characters). The rights on the user's links to the various publicly accessible directories is specified by the second parameter. Possible values are:

g specifies that the user gets links with just *group* permission.

o specifies that the user gets links with just *other* permission.

og specifies that the user gets links with both *group* and *other* permission.

The remaining arguments can be given in any order and are described in below.

–b *bulletsvr*

Specifies the capability of the bullet server for the new user, relative to */super/cap/bulletsvr*. A restricted version of this capability will be stored in the new user's */profile/cap/bulletsvr* directory under the name *default*. Each user has their own */profile/cap/bulletsvr* directory so altering a user's default Bullet Server is possible at any time.

–g *group*

Specifies the groups to which the new user belongs. At least one must be specified. For each group specified, the new user's root directory will be entered under */super/group/*`group/username` with group permission. Further, a restricted copy of */super/group/*`group` will be entered in the new user's */profile/group* directory.

–p *pooldir*

Specifies the pool directory for the new user. The name must be relative to the directory */super/pools*.

–u *unixroot* and **–n**

Along with the connections to the standard Amoeba directory environment there are optional UNIX emulation directories. If the **–u** option is given, links are also made from
unixroot*/bin* to */super/users/*username*/bin*
unixroot*/etc* to */super/users/*username*/etc*
unixroot*/lib* to */super/users/*username*/lib*
unixroot*/usr* to */super/users/*username*/usr*.

The argument *unixroot* can also be specified as –, in which case the default path */super/unixroot* is used. If the **–u** option is omitted it is equivalent to **–u –**.

If no UNIX emulation is desired then the **–n** option should be given. Do not do this unless except in exceptional circumstances. Without UNIX emulation it is not possible

to run a shell and other basic tools.

After the directory structure of the user has been added to the Soap graph, the file */super/admin/Environ* (see *login*(A)) is copied to the user's root directory and */super/admin/profile* is copied to the user's *home* directory . Finally, *newuser* requests an initial password for the new user.

The directory structure generated for each user is as follows:

/dev

This is partly for UNIX emulation but also to store the capabilities of various servers.

/home

This is the user's home directory.

/profile

This is a subgraph specifying the user's working environment. It can select subsets of pool processors, etc.

/profile/cap

This contains restricted copies of the directory capabilities in */super/cap*, except for */profile/cap/bulletsvr*, described above. However the user is free to build a private *cap* subgraph (which they must maintain themselves) which selects other servers as defaults.

/profile/group

This directory contains capabilities for the groups to which the new user belongs. These capabilities are restricted to group rights and they come from the directory */super/group*.

/profile/module

This directory is a restricted link to */super/module*. This typically contains modules such as X windows, MMDF and compilers.

/profile/pool

This is the directory linked to the *pooldir* parameter. It should contain one subdirectory of processors per architecture, as well as a run server capability corresponding to the pool directory (see *run*(A) and *makepool*(U)).

/profile/util

This is a link to the */super/util* directory, but can be substituted with an alternative set of utilities.

/tmp

This is for temporary files.

Diagnostics

If `username` already exists, *newuser* will complain and refuse to add the new user.

Other diagnostics are those of *mkd*, *chm*, *std_restrict*, *put* and *get*. Under normal circumstances there should not be any, however.

Environment Variables

SPMASK is set for the duration of the command script to provide the correct protection.

Example

```
newuser mikhail g -b bullet0 -p /super/pool -g staff
```

will create a new directory structure under */super/users/mikhail* with *group* access to the public directories and with the pool directory */profile/pool* being a restricted link to */super/pool* and with the UNIX emulation subgraph from */super/unixroot*. *Mikhail* will be a member of the group *staff*.

See Also

deluser(A), login(A), makepool(U), newsoapgraph(A), run(A).

Name

om – perform replication and garbage collection

Synopsis

```
om [-dfrvcC] [-s delay] [-t min_touch] [-p path] [-i file]
           [-b path] ... [path]...
```

Description

Om is an object manager for Amoeba. Its primary function is to do garbage collection of objects which are no longer accessible from the directory graph. On systems with duplicated bullet servers, *om* is also used to replicate files and check the consistency of the replicas.

Garbage collection is necessary since it is possible, for example, to create a file and then throw away or lose its capability. Since the file server does not know that the capability has been lost it cannot release the storage for the inaccessible file. Similarly when a file is deleted, all that normally happens is that its capability is removed from the directory graph. The actual freeing of resources consumed by the file is done later using garbage collection.

Normally *om* runs continuously in the background in server mode but *om* can also be invoked explicitly as a user command to replicate and touch one or more objects (see options, below). A small data file known as the *startup* file (described below) specifies which servers should be replicated and on which servers garbage collection should be performed. *Om* performs replication and touching of all these servers' objects which are reachable from the specified starting directories. When an object is touched its server sets its *time-to-live* field to the maximum, which is typically 24. Once all objects are believed to be touched *om* sends a *std_age* command to the relevant servers which causes the *time-to-live* field for every object managed by the server to be decreased by 1. If *om* is invoked so that it does a cycle of touching and aging once per hour then after one day, all objects which have not been touched at least once in the previous 24 hours will be destroyed.

It is *vital* that aging is not performed on servers where no touching has taken place. When *om* is first started in server mode (i.e., so that it does garbage collection), make sure that it is touching all the objects as expected before enabling *aging* in the *startup* file. If some objects are for any reason inaccessible and *om* is unaware of this, it will age objects without ever touching them. Such objects will be destroyed if it continues for too long. For safety, always use the **–t** option with the **–s** option. This will restrict the damage if anything goes wrong. The number of objects being touched by *om* in server mode can be checked using the *std_status*(U) command. The *bstatus*(A) command should be used to ensure that the Bullet Servers are receiving a reasonable number of *std_touch* commands. See below for details. It is advisable to check the status at least twice per day until it is clear that it is functioning correctly.

Options

–b *path*

Rearrange all capability-sets such that any capabilities with the same port as the capability stored in *path* appear first in the capability-sets. Multiple **–b** options are allowed. In case of conflicts between **–b** options, capabilities corresponding with

options earlier in the argument list are stored earlier in the capability-set.

–c Check the consistency of all capability-set entries that have at least one capability on a server from a replication set. If any capabilities in the entry are not in the replication set or any replication set members are not in the capability-set it prints a warning. The **–d** option will enforce consistency silently. This option cannot be combined with **–C**, **–b**, **–f** or **–d**.

–C Does the same as **–c**, but also tries to compare the contents of all objects in a capability-set. It will print a warning if the replicated objects are not identical. This option cannot be combined with **–b**, **–c**, **–d** or **–f**.

–d This forces all the replicas of an object to be only on the servers in the replication set for that object type. Therefore it removes capabilities from capability-sets where at least one of the capabilities was from one of the servers from a replication set.

–f Forces full replication as specified by the startup file. This causes all missing replicas to be added to the capset even if the *cs_final* field of the capset is set smaller than the number of replication partners.

–i *file*

Use *file* as the startup file. The default startup file is defined in *ampolicy.h* and is typically */super/admin/module/om/startup*.

–p *path*

When *om* runs as a server (the **–s** flag was given) this stores the server capability under the name *path*.

–r For each *path* argument that is a directory, find all directories reachable from that directory and process all entries in all these directories. *Om* first traverses the whole directory graph to find the capability with the most rights for each directory. If the keyword *soap_servers* is present in the startup file, *om* will restrict itself to the directory servers mentioned. Only one attempt to replicate is done for each directory, no matter how many times the directory is encountered in the traversal of a directory graph with circuits.

–s *delay*

Puts *om* in server mode. This has several effects: after each touching/replication pass through the list of specified path names, *std_age* will be called on all the ports specified by the *aging* keyword of the *startup* file. *Om* will also ensure that it starts a new pass *delay* seconds after it started the previous pass. However if the previous pass took more than *delay* seconds then it will start the next pass immediately. This way it is possible to ensure that, barring serious problems, a certain number of passes are done per 24 hours. This is very useful for setting the garbage collection timeouts. Also in server mode, *om* will initiate a server thread which accepts `STD_INFO` and `STD_STATUS` requests. The server publishes the capability that it listens to in the directory specified by `DEF_OMSVRDIR` in *ampolicy.h* (typically */super/cap/omsvr*). The name used is *om1*. It will generate a unique capability under this path name. The `STD_INFO` command returns a string identifying the capability as the ‘‘object manager’’ and the `STD_STATUS` command returns information about how many objects have been processed and what was done to them. This is the same information as printed with the **–v** option, below. This summary message is also printed at the end of each pass of the server.

Note that if the **–s** option is used it is important to also use the **–t** option to protect against *om* possibly failing to touch certain objects but still sending age commands to the server.

–t *min_touch*
Specifies the minimum number of objects that must be touched on each pass, before the *std_age* commands are performed. (This is a safety net, to prevent aging when something went wrong with the touching.) The *touch* keyword in the startup file specifies the servers whose objects should be touched. If this keyword is not present *om* will touch all the capabilities it finds. Note that if **–s** is specified and **–t** is not then the minimum number of objects which must be touched before aging takes place is 10000. If there are fewer than 10000 objects in the system then no aging will be done unless the **–t** option is used to specify a smaller value.

–v Print a line of information as each object is touched or replicated and, at the end (or the end of each pass, in server mode), prints the total number of actions of each type that were performed on the objects.

Startup file

The startup file consists of a sequence of lines. Each line is a sequence of tokens separated by white-space. White-space is a sequence of tabs and spaces. Comments start with a '#' and continue to the end of the line. Outside comments the '\' can be used to strip a character of its special meaning. A '\' immediately before a newline can be used to create continuation lines. Backslashes can also be used to include '#', white-space and backslashes in tokens. Empty lines and lines only containing a comment are skipped.

The first token in all other lines is considered to be a keyword. The following keywords are currently recognized:

`replicas`
Each line with this keyword describes a replication set. All tokens following the keyword should be path names. *Om* assumes that the capability stored under each path name specifies the put-port of a bullet server. These put-ports form a replication set.
Most normal file creation procedures create a bullet file and store the resulting capability in a directory entry. If the bullet server of that file is in one of the replication sets *om* will, when it finds such a file, create copies of the file on all other servers mentioned in the replication set and store the capabilities for those copies in the directory entry.
To be more precise; if *om* finds a directory entry with a capability-set of which at least one capability has same put-port as in a replication set, *om* will check whether all put-ports in the replication set are represented in the capability-set. *Om*, normally used with the **–f** option, will create copies of the file on all missing bullet servers and include these copies in the capability-set for the directory entry.

`aging`
All tokens following the keyword should be path names. After each touching/replication pass in server mode (**–s**), *std_age* will be called on all the ports stored under the path names specified after all `aging` keywords. *Om* assumes that only one capability is stored under each path name.

`soap_servers`

All tokens following the keyword should be path names. When the **−r** option is specified *om* will try to access all directory entries reachable from the paths given in the arguments to *om*. If one or more *soap_servers* keywords are present in the startup file, *om* will only access directories with the same port as the directory servers specified by this keyword. *Om* assumes that only one capability is stored under each path name.

`publish`

The path name under which the server capability is published. See the description under the **−r** option.

`touch`

All tokens following the keyword should be path names. *Om* assumes that only one capability is stored under each path name. This keyword specifies the servers whose objects should be touched. If this keyword is not present *om* will touch all capabilities it finds. It is not necessary to specify the servers mentioned in a replication set after this keyword. *Om* does a *std_info* on all capabilities from the servers which are in any of the replication sets. For bullet servers this has the same effect as a *std_touch.* It is useful to specify all directory servers specified by the startup file, because *std_touch* on a *soap* server has more effects then a *std_info.*

If this keyword is not present *om* will *std_touch* all capabilities found.

Below is an example of a *startup* file. Note that on the touch line we specify all soap objects accessible from / and all objects on the default bullet server. If there is more than one bullet server it should be added to the *touch* entry.

```
# OM startup file for unreplicated file systems.
# Uncomment the "aging" entry when you've found out what number
# of objects should be touched in a run before OM is allowed to
# age the files and directories.  This number can be specified
# using the "-t" option of om.
#aging /super/cap/bulletsvr/default /super/cap/soapsvr/soap0
soap_servers /
publish om1
touch / /super/cap/bulletsvr/default
```

Note also that if there are replicated Bullet Servers then these Bullet Servers need only appear in the *replicas* line. It is not necessary to put them in the *touch* line since they will already be touched automatically by the replication process.

Diagnostics

Om complains if any RPC fails, including *std_touch*, *std_copy*, *std_info*, *name_lookup*, *name_append*. During each pass through a specified set of path names, it will only complain the first time that `RPC_NOTFOUND` is returned for any specific port. Subsequent attempts to send an RPC to the bad port will be suppressed until the end of the pass. Then the list of bad ports will be cleared, in case any of the servers have come back up. If the bad port is in one of the replication sets or one of the touch ports then the current pass is aborted and no aging is done.

Examples

```
om -fr /home
```

causes *om* to replicate and touch everything reachable under the caller's */home* directory.

```
om -fr -s 3600 -t 20000 /super
```

causes *om* to run as a server and every hour to touch and age the entire directory graph. If less than 20000 objects are touched then no aging will be done.

See Also

bstatus(A), capset(L), std_age(A), std_copy(L), std_touch(U).

Name

ping – send ICMP ECHO_REQUEST packets to network hosts

Synopsis

```
ping [-Rdfnqrv] [-c count] [-i wait] [-l preload] [-p pattern]
        [-s packetsize] host
```

Description

Ping uses the ICMP protocol's mandatory ECHO_REQUEST datagram to elicit an ICMP ECHO_RESPONSE from a host or gateway. ECHO_REQUEST datagrams (''pings'') have an IP and ICMP header, followed by a *struct timeval* and then an arbitrary number of ''pad'' bytes used to fill out the packet.

When using *ping* for fault isolation, it should first be run on the local host, to verify that the local network interface is up and running. Then, hosts and gateways further and further away should be ''pinged''. Round-trip times and packet loss statistics are computed. If duplicate packets are received, they are not included in the packet loss calculation, although the round trip time of these packets is used in calculating the minimum/average/maximum round-trip time numbers. When the specified number of packets have been sent (and received) or if the program is terminated with a SIGINT, a brief summary is displayed.

This program is intended for use in network testing, measurement and management. Because of the load it can impose on the network, it is unwise to use *ping* during normal operations or from automated scripts.

Options

–c *count*
: Stop after sending (and receiving) *count* ECHO_RESPONSE packets.

–d This has no effect under Amoeba.

–f Flood ping. Outputs packets as fast as they come back or one hundred times per second, whichever is more. For every ECHO_REQUEST sent, a period ''.'' is printed, while for every ECHO_REPLY received a backspace is printed. This provides a rapid display of how many packets are being dropped. Only the super-user may use this option.
NB. This can be very hard on a network and should be used with caution.

–i *wait*
: Wait *wait* seconds between sending each packet. The default is to wait for one second between each packet. This option is incompatible with the **–f** option.

–l *preload*
: If *preload* is specified, *ping* sends *preload* packets as fast as possible before falling into its normal mode of behavior.

–n Numeric output only. No attempt will be made to lookup symbolic names for host addresses.

–p *pattern*
: You may specify up to 16 ''pad'' bytes to fill out the packet you send. This is useful for

diagnosing data-dependent problems in a network. For example, **−p** *ff* will cause the sent packet to be filled with all ones.

−q Quiet output. Nothing is displayed except the summary lines at startup time and when finished.

−R Record route. Includes the `RECORD_ROUTE` option in the `ECHO_REQUEST` packet and displays the route buffer on returned packets. Note that the IP header is only large enough for nine such routes. Many hosts ignore or discard this option.

−r Bypass the normal routing tables and send directly to a host on an attached network. If the host is not on a directly-attached network, an error is returned. This option can be used to ping a local host through an interface that has no route through it.

−s *packetsize*
Specifies the number of data bytes to be sent. The default is 56, which translates into 64 `ICMP` data bytes when combined with the 8 bytes of `ICMP` header data.

−v Verbose output. `ICMP` packets other than `ECHO_RESPONSE` that are received are listed.

Protocol Information

ICMP Packet Details

An IP header without options is 20 bytes. An `ICMP ECHO_REQUEST` packet contains an additional 8 bytes worth of `ICMP` header followed by an arbitrary amount of data. When a *packetsize* is specified, this indicates the size of this extra piece of data (the default is 56). Thus the amount of data received inside of an IP packet of type `ICMP ECHO_REPLY` will always be 8 bytes more than the requested data space (the `ICMP` header).

If the data space is at least eight bytes large, *ping* uses the first eight bytes of this space to include a timestamp which it uses in the computation of round trip times. If less than eight bytes of pad are specified, no round trip times are given.

Duplicate and Damaged Packets

Ping will report duplicate and damaged packets. Duplicate packets should never occur, and seem to be caused by inappropriate link-level retransmissions. Duplicates may occur in many situations and are rarely (if ever) a good sign, although the presence of low levels of duplicates may not always be cause for alarm.

Damaged packets are obviously serious cause for alarm and often indicate broken hardware somewhere in the *ping* packet's path (in the network or in the hosts).

Trying Different Data Patterns

The (inter)network layer should never treat packets differently depending on the data contained in the data portion. Unfortunately, data-dependent problems have been known to sneak into networks and remain undetected for long periods of time. In many cases the particular pattern that will have problems is something that does not have sufficient ''transitions'', such as all ones or all zeros, or a pattern right at the edge, such as almost all

zeros. It is not necessarily enough to specify a data pattern of all zeros (for example) on the command line because the pattern that is of interest is at the data link level, and the relationship between what you type and what the controllers transmit can be complicated.

This means that if you have a data-dependent problem you will probably have to do a lot of testing to find it. If you are lucky, you may manage to find a file that either cannot be sent across your network or that takes much longer to transfer than other similar length files. You can then examine this file for repeated patterns that you can test using the **–p** option of *ping*.

TTL Details

The TTL value of an IP packet represents the maximum number of IP routers that the packet can go through before being thrown away. In current practice you can expect each router in the Internet to decrement the TTL field by exactly one.

The TCP/IP specification states that the TTL field for TCP packets should be set to 60, but many systems use smaller values (4.3 BSD uses 30, 4.2 used 15).

The maximum possible value of this field is 255, and most UNIX systems set the TTL field of ICMP ECHO_REQUEST packets to 255. This is why you will find you can ‘‘ping’’ some hosts, but not reach them with *ttn*(U) or *ftp*(U).

In normal operation *ping* prints the TTL value from the packet it receives. When a remote system receives a ping packet, it can do one of three things with the TTL field in its response:

- Not change it; this is what Berkeley UNIX systems did before the 4.3 tahoe release. In this case the TTL value in the received packet will be 255 minus the number of routers in the round-trip path.
- Set it to 255; this is what current Berkeley UNIX systems do. In this case the TTL value in the received packet will be 255 minus the number of routers in the path *from* the remote system to the ‘‘pinging’’ host.
- Set it to some other value. Some machines use the same value for ICMP packets that they use for TCP packets, for example either 30 or 60. Others may use completely wild values.

Warnings

Many Hosts and Gateways ignore the RECORD_ROUTE option.

The maximum IP header length is too small for options like RECORD_ROUTE to be completely useful. There is not much that that can be done about this, however.

Flood pinging is not recommended in general, and flood pinging the broadcast address should only be done under very controlled conditions.

See Also

ifconfig(A), ipsvr(A).

Name

pr_routes – print the routing table of an IP server

Synopsis

```
pr_routes [ -I ip-capability ]
```

Description

Pr_routes prints the routing table of an IP server in the following format:

```
<ent-no> DEST= <ip-addr>, NETMASK= <netmask>, GATEWAY= <ipaddr>, dist= %d,
pref= %d [ fixed ]
```

The *dist* field is an estimated or configured distance that can be used in the *ttl* field of an IP header. The *pref* field shows the preference level of the gateway, the higher the better. The word *fixed* is optional. It indicates that the route was added manually or that the route is built-in.

Options

–I ip-capability

This option gives the path name of the capability of the IP server. It can also be given as the name of the host containing the IP server. If this option is omitted the value of the environment variable `IP_SERVER` is used. If this is not set then it uses the default path, as defined in *ampolicy.h* (typically */profile/cap/ipsvr/ip*).

Environment Variables

`IP_SERVER` This environment variable overrides the default path name for the IP server. It is in turn overridden by the **–I** option.

Example

```
pr_routes -I /super/hosts/armada1E/ip/ip
```

and the result can be

```
0 DEST= 0.0.0.0, NETMASK= 0.0.0.0, GATEWAY= 192.31.231.93, dist= 1 pref= 509
1 DEST= 0.0.0.0, NETMASK= 0.0.0.0, GATEWAY= 192.31.231.43, dist= 1 pref= 509
2 DEST= 0.0.0.0, NETMASK= 0.0.0.0, GATEWAY= 192.31.231.42, dist= 1 pref= 509
3 DEST= 4.5.6.7, NETMASK= 255.0.0.0, GATEWAY= 1.2.3.4, dist= 1 pref= 0 fixed
```

The same command could also have been given as:

```
pr_routes -I armada1E
```

See Also

add_route(A), ipsvr(A), irdpd(A).

Name

printbuf – print the console buffer of an Amoeba host

Synopsis

```
printbuf [-s] [-c] [-p] machine
```

Description

Printbuf either prints the contents of the console buffers in the Amoeba host specified by *machine*, or prints output to the console buffer since the last *printbuf* request. This depends on the presence of a printbuf server (see *printbufsvr*(A)) and the **–s** and **–c** command line options.

If the **–s** option is present on the command line, *printbuf* attempts to lookup the *sys* server of the host specified by *machine*. If the **–c** option is present, the printbuf server in *machine* is used. If neither the **–c** nor the **–s** flags are present *printbuf* tries to lookup the printbuf server in *machine* and assumes the **–c** option when successful and the **–s** option otherwise.

If the **–s** option is turned on (either explicitly or implicitly) the entire console buffer is printed once. On the other hand, if the **–c** option is turned on, *printbuf* repeatedly sends *printbuf* requests which return only the additions to the console buffer since the previous request. Thus the illusion of a continuous stream is created. If nothing is printed at all then the **–c** option is probably in effect and there is no new data in the print buffer since the previous *printbuf* request.

If the **–p** option is present *printbuf* will continue even in the presence of errors. If the **–p** option is not present *printbuf* will quit after the first error.

Printbuf attempts to detect series of garbage characters and avoid printing them. In this case it prints *etc...* followed by a new line. Non-printable characters are printed as follows:

backspace	\b
formfeed	\f
null	\0
backslash	\\
all others	\ooo

where *ooo* is a 3-digit octal number.

Diagnostics

`directory lookup of` *hostname* `failed:` *error message*

– means that it could not find the host in any of the searched locations. The actual cause of the error is described by *error message*.

In the following *nnn* and *sss* are integers.

```
returned offset (0xnnn) out of range [0 .. 0xsss)
returned num_bytes (0xnnn) out of range (0 .. 0xsss]
```

– these two messages mean that illegal values were returned by the server answering the printbuf request.

`failed:` *error message*
– means that the request to the host failed for the reason specified by *error message*.

Example

```
printbuf vmesc3
```

See Also
aps(U), kstat(A), printbufsvr(A), syssvr(A).

Name

printbufsvr – optional kernel server to maintain transfer kernel console output

Synopsis

Built into the kernel.
Add

```
$K_SVR_PRINTBUF,
```

to the source list of the kernel *Amakefile*.

Description

The *syssvr* supports the *sys_printbuf* function by sending the complete console buffer in reply to a *printbuf* request. This is useful for occasional examination of the console buffer. In contrast, the printbuf server returns only the additions the print buffer since the previous *printbuf* request and blocks the client when no data is available. This way the *printbuf*(A) utility is able to provide an illusion of direct connection to the output of a kernel's console.

Diagnostics

`STD_NOTNOW` is returned when an attempt is made to start more than one *printbuf* command for a server. In this case

```
printbuf -s
```

will use the *sys* server and can be used for the second *printbuf*.

Programming Interface Summary

Available Functions			
Function Name	Required Rights	Error Conditions	Summary
std_info	NONE	RPC error	Print identifying string.
sys_printbuf	SYS_RGT_READ	RPC error	Report additions to the print buffer

See Also

printbuf(A), syssvr(A).

Name

prkdir – print the contents of a kernel boot directory

Synopsis

```
prkdir
```

Description

Prkdir reads a kernel boot directory from *stdin* and prints a readable version of the contents on *stdout*. The output of *prkdir* is in the format required for input to *mkkdir*(A).

Example

```
dread /super/hosts/bullet0/vdisk:01 | prkdir
```

will read the kernel boot directory on block zero of the specified virtual disk and print it on standard output. This might look something like

```
1       800     amoeba
801     800     test
```

See Also

dread(A), mkkdir(A).

(UNIX ONLY)

Name

profile – display graphically the results of an Amoeba kernel profile

Synopsis

```
profile [-a] [-i interval] [-k kernel] [-m] [-n iterations] [-v] host
```

Description

Profile is used to collect stack traces from an Amoeba kernel and to display a profile of function calls in graphical form, including the amount of time spent in each function.

To use this from UNIX the UNIX kernel must have an Amoeba FLIP driver and the Amoeba kernel must have a profile server (see *profsvr*(A)).

The *profile* command works as follows: It sends a profile request to the profile server in the Amoeba kernel. The server begins filling a buffer with stack traces from the kernel. A stack trace is taken every 10 milliseconds if not otherwise specified by the **–i** option. The buffer contains 30000 bytes. When it is full the profile server returns the buffer to *profile*. More than one buffer will be collected if the **–n** option is specified. Once the data has been collected from the kernel it is analyzed using *nm* and then printed in a graphical format showing the function call structure and the amount of time (in milliseconds) spent in each routine.

Options

–a The kernel was generated using ACK. Use the ACK program *anm* to analyze the kernel's symbol table.

–i *interval*
This sets the time between making stack traces to *interval* milliseconds. The default is 10.

–k *kernel*
This specifies the name of the UNIX file containing the Amoeba kernel that is running on *host*. The symbol table from this file is used to analyze the stack traces. The default is *./kernel*.

–m Merge the instances of a function call into a single total for that function.

–n *iterations*
The number of times the profile server is asked to provide a set of stack traces is set to *iterations*. The default is 1.

–v Verbose mode. Useful for debugging. It is not recommended.

See Also

profsvr(A).

Name

profsvr – kernel profiling server

Synopsis

Built into the kernel.
Add

```
-DPROFILE
```

to the defines and

```
$K_SVR_PROFILE,
```

to the source list of the kernel *Amakefile*.

Description

The profile server runs inside the Amoeba kernel. It is used to obtain stack traces of the kernel which can be analyzed by *profile*(A). The analysis shows the time spent in the various functions in the kernel. This is useful for finding the performance bottlenecks in the kernel.

An example of how to add a profile server to an *Amakefile* can be found in the template directory *templates/kernel/force30/pool*.

The profile server can only deal with one client at a time. If two people attempt to obtain profile information from the same kernel at the same time, one of them will almost certainly get a *not now* error.

Warnings

There is no support for the profile server in Sun 3 kernels.

Programming Interface Summary

Administrative Functions			
Function Name	Required Rights	Error Conditions	Summary
std_info	NONE	RPC error	Print server id
sys_profile	SYS_RGT_CONTROL	RPC error STD_DENIED STD_NOTNOW	Get stack traces

See Also

profile(A).

Name

random – the random number server

Synopsis

Currently built into the kernel

Description

This server provides Amoeba programs with random numbers. No rights are required to use this server. Random numbers are the privilege of all. There is only one command and that returns a random number of the size specified.

Programming Interface Summary

The programming interface consists of the single command particular to the server (whose name begins with *rnd_*) and the standard server commands (whose names begin with *std_*). A summary of the supported commands is presented in the following two tables. For a complete description of the interface routines see *rnd*(L) and *std*(L). Not all the standard commands are supported since they are not all pertinent. *Std_copy* and *std_destroy* are not implemented. *std_age*, *std_restrict* and *std_touch* are implemented but simply return `STD_OK` and do nothing further. They do no error checking.

Standard Functions			
Function Name	Required Rights	Error Conditions	Summary
std_age	NONE	RPC error	Does Nothing
std_info	NONE	RPC error	Returns the string ‘‘Random Number Server’’
std_restrict	NONE	RPC error	Does nothing to the capability
std_touch	NONE	RPC error	Does Nothing

RND Function			
Function Name	Required Rights	Error Conditions	Summary
rnd_getrandom	NONE	RPC error	Returns a random number of the requested size

Administration

There is only one administrative task relating to the random server. That is installing the capability of one of the random servers as the default random server. The place for installing the default server capability is described in *ampolicy.h* by the variable `DEF_RNDSVR` and is typically */profile/cap/randomsvr/default*. However the place to install it is via the path */super/cap/randomsvr/default* which is the public directory. (The */profile* directory may vary from user to user but typically points to the public version.) It is normal practice to allow the boot server (see *boot*(A)) to maintain the default capability.

See Also

boot(A), random(L), std(L), std_info(U).

Name

rarp – the Reverse Address Resolution Protocol daemon

Synopsis

```
rarp [-v] [-E eth-cap] [-I ip-cap] [-chan chan-cap] [-cache]
```

Description

The *rarp* daemon implements the Reverse Address Resolution Protocol as described in RFC 903. This is used mainly by Sun computers to determine their Internet address based on their Ethernet address. The Internet address is then used to get the required kernel to be booted on the machine using the TFTP protocol (see *tftp*(A)).

The *rarp* server requires the *eth* port of an Amoeba TCP/IP server (see *ipsvr*(A)) to access the network. This can be specified on the command line or using an environment variable. In general it is a good idea to run the *rarp* server on the same machine as the TCP/IP server to obtain maximum performance. Note that the TCP/IP server must know its own Internet address before the *rarp* server will function. This can be given to it using the *ifconfig*(A) or *setupipsvr*(A) commands.

Options

–E *eth-capability*
: This tells the *rarp* server the capability of the Ethernet server that it must use to write packets on the Ethernet. This is typically the *eth* capability of a TCP/IP server. It can also be specified by the name of the host where the IP server is running. If this option is not specified it takes the value of the string environment variable ETH_SERVER. If this is not defined then it uses the default ETH_SVR_NAME as defined in the file *ampolicy.h*. This is typically */profile/cap/ip/eth*.

–I *ip-capability*
: If the *rarp* server should respond only to hosts on its own network then it needs to know its own IP address and netmask. It uses this capability to find this out.

–chan *chan-capability*
: This tells the *rarp* server the place to publish the capability for its Ethernet channel. This can be used for examining the status of the connection (see *std_status*(A)).

–cache
: This tells the *rarp* server to cache the Ethernet to Internet address mappings for increased performance. This should only be used when many machines are booted at once.

–v Verbose mode. It prints lots of details about the various requests received.

Environment Variables

ETH_SERVER
: This shell environment variable specifies the name of the *eth* channel to use. It can be overridden by the **–E** command line option.

TCP_SERVER
: This shell environment variable specifies the name of the *tcp* channel to use to look up names in the Domain Name System (DNS).

Warnings

The *rarp* server uses *gethostbyname* to lookup Internet addresses. This function tries the DNS before looking at the */etc/hosts* file. Therefore a TCP server must be configured using the *TCP_SERVER* environment variable or an appropriate capability must be registered as the default TCP server, as defined in *ampolicy.h*. (This is typically placed in */profile/cap/ip/tcp*).

In caching mode the *rarp* server does not make an effort to keep its cache consistent with the */etc/hosts* and */etc/ethers* files or the DNS tables. If caching is enabled and one of these is modified, the *rarp* server should be restarted.

Files

/etc/hosts and */etc/ethers* are used to pair Ethernet addresses with Internet addresses.

Example

To start the rarp server using the TCP/IP server on the host *foo*:

```
rarp -E /super/hosts/foo/ip/eth -chan /super/cap/rarpsvr/foo
```

See Also

ifconfig(A), ipsvr(A), setupipsvr(A), tftp(A).

Name

reboot – reboot a machine with a new kernel binary

Synopsis

```
reboot [options] kernel machine ...
```

Description

If a host is running an Amoeba kernel then *reboot* causes a machine to reboot itself with a new kernel binary. It does this by sending the capability of the new kernel binary (specified by the *kernel* argument) to the system server (see *syssvr*(A)) running in the kernel of the host specified by *machine*. This name is looked up using *host_lookup*(L), so it can be either the name of a machine or the full path name of a capability for a machine.

Options

–a *args* Pass the single string *args* to the kernel which is to be rebooted. The interpretation of these arguments depends on the kernel booted.

Diagnostics

Reboot complains only if the kernel cannot be found or the system server is not present on the specified machine. If the kernel is not valid or defective it will just crash when started.

Warnings

There is no indication of failure of the reboot command, since the kernel has to send the reply before attempting the reboot.

Sending garbage data (like a kernel for a different CPU, MMU or bus architecture) will undoubtedly crash the machine.

Reboot is of no help to reboot a halted machine. Depending on their type, halted machines may be rebooted from a local disk, the network or using a bootstrap Amoeba kernel in ROM.

Examples

To reboot the hosts ‘‘bpl’’, ‘‘blo’’, and ‘‘bne’’:

```
reboot /super/admin/kernel/sun3.pool bpl blo bne
```

To reboot the i80386 host ‘‘pico’’ with an instruction not to automatically reboot after going down:

```
reboot -a "-noreboot:1" /super/admin/kernel/isa.pool pico
```

See Also

host_lookup(L), kstat(A), mkhost(A), printbuf(A), sys_boot(L).

Name

reserve_svr – server for managing processor reservations

Synopsis

```
reserve_svr [-c configdir]
```

Description

Amoeba is often used for running parallel programs on multiple hosts. For testing and measurement purposes it is sometimes necessary to be able to gain exclusive access to some or all of the hosts on a network. Some parallel programs also consume such large amounts of system resources (cpu cycles, memory and network bandwidth) that it is better to run them outside of normal working hours. The reservation server supports these requirements. It allows a user to gain exclusive access to a subset of the available hosts and optionally submit a job for batch execution.

The reservation server keeps a simple database which describes when reservations may and may not be made and what to do on termination of a batch job. It also keeps a list of processor reservations and batch jobs awaiting execution plus any special parameters about exclusive use, and attempts to ensure that all the jobs are scheduled as efficiently as possible. If a job is submitted for which the reservation server is unlikely to be able to satisfy the user's requirements then the job will be rejected. The user interface and capabilities of the reservation system are described in *reserve*(U).

Options

–c *configdir*
: Search for the administration files in the directory *configdir*. The default is */super/admin/module/reserve*.

Files

The objects used by the reservation server are typically stored in the directory */super/admin/module/reserve*.

The file *config.py* contains the basic configuration details for the server, including when reservations cannot be made and what should be done when a reservation terminates (typically send email to the reservee). This information is compiled into a binary form by the reservation server and stored in the file *config.pyc*.

The file *status.py* contains the status of pending reservations in a format understood by the Python interpretter. The binary form is also automatically generated by the reservation server and stored in the file *status.pyc*.

The directory *hosts.orig* contains the processor pool from which hosts may be reserved. It has the same structure as a *run*(A) server pool. The directory *hosts.current* is generated and maintained by the reservation server, based on the contents of the *hosts.orig* pool.

Programming Interface Summary

The server supports the following commands:

User Functions			
Function Name	Required Rights	Error Conditions	Summary
res_reserve	RGT_CREATE in server cap	RPC error STD_CAPBAD STD_DENIED	Make a reservation.
std_destroy	RGT_DELETE in object cap All rights in server cap	RPC error STD_CAPBAD STD_DENIED	If object cap then cancel the reservation. If server cap then halt server.
std_info	RGT_READ in object-cap	RPC error STD_CAPBAD STD_DENIED	Return string containing reservation identifier.
std_restrict	None	RPC error STD_CAPBAD	Make a version of the capability with fewer rights.
std_status	RGT_READ in server cap	RPC error STD_CAPBAD STD_DENIED	Returns as a string, the list of all valid reservation periods, reservable hosts and current reservations.

Administration

Create the pool *hosts.orig*. This should contain all the hosts that may be reserved. It should not contain hosts running system servers. Hosts which are reserved exclusively have their kernel capability changed and this will cause the *boot*(A) server to become confused.

Set values in the file *config.py*. The distribution comes with a sample file. Be sure to adjust the email address for the sender of termination mail messages.

Example

When starting the reservation server from the command line the shell script *reserve_svr* simply starts a python script. However the reservation server will normally be started by the *boot*(A) server. Since running shell scripts from the *boot* server is unreliable the *Bootfile* entry for starting the reservation server should look like the following:

```
Reserver {
   after Session_svr;
   machine SLASH "super/hosts/XXXX";
   bootrate 60000;
   pollrate 30000;
   program {
      SLASH "super/util/python";
      argv {
           "python",
           "/super/module/python/util/reserver.py"
      };
      capv {
           _SESSION = SLASH "dev/session",
           STDIN, STDOUT, STDERR,
           ROOT = SLASH, WORK = SLASH
      };
      environ {
           "PYTHONPATH=/super/module/python/amlib:/super/module/python/lib"
      };
   };
};
```

where XXXX is the host on which the server should run.

See Also

reserve(U).

Name

run – load balancing server

Synopsis

```
run [-D number] [-U number] [-S number]
    [-a dir] [-d] [-i] [-n name] [-p dir]
```

Description

The *run* server is the load balancing server. It attempts load balancing of one or more processor pools by directing each new job to the most appropriate processor, based on available resources and the job's resource requirements.

The *run* server can manage an arbitrary number of pool directories, *pooldir*s for short, each containing subdirectories for one or more processor architectures, dubbed *archdir*s. For example, the default pool directory is */profile/pool* and it might have architecture subdirectories named *sparc*, *i80386* and *mc68000*.

Each *archdir* can contain an arbitrary number of *host* (processor) capabilities. A host may be a member of more than one pool. Pool directories to be maintained by the *run* server can be added at run-time using the program *makepool*(U). This will install a new *run capability* in the *pooldir* under the name *.run*. A request concerning a particular run capability chooses only between the hosts in the corresponding pool directory.

The prime request handled by the *run* server is *run_multi_findhost*. *Run_multi_findhost* takes as argument a run capability and a set of process descriptors. The process descriptors are assumed to be all for the same program but for different architectures. It searches in the pool specified by the run capability for the ‘‘best’’ host on which to run the program. It returns a capability which allows the caller to execute the program on that host. It is normally called indirectly via *exec_multi_findhost* (see *exec_findhost*(L)) which marshals the process descriptors and handles exceptions if the *run* server cannot be reached.

Much of the *run* server's time is devoted to keeping a fairly up-to-date idea of the load on each host without using too much of the system's resources. It tries to minimize time spent polling a host that went down suddenly. The *run* server itself is not replicated, but it is easy to ensure that a copy is kept up and running using the boot server (see *boot*(A)).

The decision about which host to use is a complicated one. One might want to take the following in account: available memory, memory fragmentation (especially since the current version of Amoeba cannot move segments) and whether the text segment is cached in the processor. The most interesting bit of information is unfortunately unavailable: what is the process going to do? Since load balancing is an art in itself — in fact the subject of doctoral dissertations — the *run* server currently uses a simple algorithm, but it is easy to change the algorithm without having to change the whole program structure.

The current algorithm evaluates a fairly simple formula for each host having a suitable architecture, and chooses randomly between the hosts that evaluate within a certain percentage of the highest value found. The formula is based on the amount of memory left after the process is started (this means the process size is used as input to the formula), the expected speed at which the process will run, and the size of the text segment if it is believed to be cached at that host.

$$Speed_h = \frac{KIPS_h}{LoadAvg_h + 1}$$

$$AvailMem_{h,\,pd} = FreeMem_h - MemReq_{h,\,pd} + \texttt{CACHEMEM} * SegCached_{h,\,pd}$$

$$Startup_{h,\,pd} = 1 + \frac{SegCached_{h,\,pd}}{\texttt{CACHETIME}}$$

$$Suitability_{h,\,pd} = \texttt{PREF}(Arch_h) * Speed_h * min\,(\texttt{MEMTRUNC}, AvailMem_{h,\,pd}) * Startup_{h,\,pd}$$

The expected CPU power available to the new process on a host *h* is given by the formula for $Speed_h$. $KIPS_h$ is the computing power available in kilo-instructions per second on host *h* when it is idle (which is estimated by the kernel at boot time), and $LoadAvg_h$ is the estimated number of runnable threads on *h*, measured over the last few seconds. The ''+ 1'' represents a single thread of the new process that is to be started. $SegCached_{h,\,pd}$ is the size of the text segment of process descriptor *pd* if it is thought to be still in the cache on host *h*. Otherwise it is 0. CACHEMEM is a constant between 0 and 1 reflecting the system administrator's expectation that the host *h* is likely to still have the text segment cached.

To make it possible to promote or discourage the use of a particular architecture, the suitability value is additionally multiplied with an architecture dependent factor, PREF–arch. The amount of memory available is taken into account by means of the expression $AvailMem_{h,\,pd}$. To avoid starting many processes on a single host solely for the reason that it has significantly more memory than the others, the Run server ignores available memory exceeding the amount MEMTRUNC. Finally, the expression $Startup_{h,\,pd}$ gives the host a certain preference when its text segment is assumed to be cached at the host, thereby allowing it to start more quickly. The pool-dependent constants PREF-arch, CACHETIME, MEMTRUNC and CACHEMEM in the formula can be modified by the system administrator.

It is possible for the system administrator to disable and enable pools. This is done by setting the ENABLE parameter to 0 or 1, respectively, using *std_params*(A).

Options

The *run* server recognizes the following command line options.

–D number Specifies the number of threads devoted to polling processors that are believed to be down (default 2).

–S number Specifies the number of server threads (default 4).

–U number Specifies the number of threads devoted to polling processors that are thought to be up (default 2).

–a dir Specifies the directory where the *run* server should keep its administration file. The default is DEF_RUN_ADMIN as defined in *ampolicy.h* (typically */super/module/run*).

–d Turns on debugging output. Repeat to get more.

–i Causes the initialization of the *run* server's administration file and publication of a new super capability. This option should only be given the first time a *run*

server is started.

−n name Specifies an alternative name for the *run* server to use when publishing its capability and storing its administration file. This makes it possible to have multiple independent *run* servers. The default name for the *run* server is *default*.

−p dir Specifies an alternative name for the directory in which the *run* server should publish its super capability. The default is `PUB_RUNSVR_DIR` defined in *ampolicy.h* (typically */super/cap/runsvr*).

Diagnostics

During initialization, the *run* server complains about invalid command line syntax, missing (if **−i** is not given) or inconsistent administration file, and a few other unlikely conditions. All these complaints cause the *run* server to exit with a non-zero exit status. After the *run* server has initialized, the only fatal error is its inability to rewrite its administration file. Other conditions only cause a warning to be printed.

Files and Directories

*/super/cap/runsvr/**	Super capabilities for the currently active run servers
/super/module/run/.adm*	Administrative files
*/super/module/run/pools/**	Links to all pool directories
/profile/pool	Default processor pool.
/profile/pool/.run	Default pooldir capability.

Warnings

Host names containing periods should be avoided. They are ignored by both the *run* server and the process startup library.

If the *run* server has crashed, users without the *super* directory capability will not be able to start new processes. The holders of the *super* capability should be able to start processes. The *boot* server should be charged with keeping the *run* server available. Use *std_status*(A) to check if the *run* server is up.

Selecting a host for a program is often a wild guess, since the *run* server can predict neither what a program will do, nor what resources will become available soon. Hence, there should be a second form of load balancing, based on migration of processes to processors where they will run better. However, this is not supported in the current *run* server.

Programming Interface Summary

The programmers' interface consists of commands particular to the *run* server (whose names begin with *run_*) and the standard server commands (whose names begin with *std_*). The commands are divided into two categories: administrative and user. A summary of the commands is presented in the following two tables, the first listing the administrative commands and the second the user commands. The interface for the standard operations is described in detail in *std*(L).

Except where otherwise stated, requests are specific to one *pooldir*. Apart from the error

conditions explicitly mentioned, general RPC errors are also possible.

Administrative Functions			
Function Name	Required Rights	Error Conditions	Summary
std_age	RUN_RGT_ADMIN on super cap	STD_CAPBAD STD_DENIED	Start a garbage collection pass
std_touch	RUN_RGT_ADMIN	STD_CAPBAD STD_DENIED STD_NOSPACE	Rescan the pool directory corresponding to *run* server object, reset its time to live
std_getparams	RUN_RGT_ADMIN	STD_CAPBAD STD_DENIED	Get evaluation parameters of a *run* server object
std_setparams	RUN_RGT_ADMIN	STD_CAPBAD STD_DENIED	Set evaluation parameter of a *run* server object
std_status	RUN_RGT_STATUS on run object or super cap	STD_CAPBAD STD_DENIED	Return *run* server statistics for hosts in a pool

More specific information for some of the administrative operations:

std_status — For pool objects, it returns a printable overview of statistics about the hosts in it. A *std_status* on the super capability gives statistics about all hosts known by the *run* server.

std_setparams — Stores the per-pool parameters for the host evaluation formula.

std_getparams — Retrieves the per-pool parameters for the host evaluation formula. These commands are normally issued by using the program *std_params*(A). Details of the arguments and their use are given in the System Administration section, below.

More specific information for some of the user functions:

run_get_exec_cap — This takes as input the capability for a kernel process server from the caller's processor pool. It returns a) the architecture of that machine, and b) the current capability for creating a process on that machine. The error code `STD_NOTNOW` is returned when the caller's pool is disabled.

run_multi_findhost — Returns the process server capability of the ''best'' host that is suitable to run one of set of process descriptors. The error code `STD_NOTNOW` is returned when no host in the pool specified is able to execute one of the process descriptors provided.

std_info — On the super capability, it returns the string *run server*. On pool objects, it returns a string telling for which architectures there is a running host available. For example, ''`! mc68000 i80386`''.

Note that operations which are *run* server specific (for example *run_multi_findhost)* should not be called directly, but via the higher level *exec* interface (see *exec_findhost*(L)) that also takes care of marshaling and exception handling, in case the *run* server cannot be reached.

User Functions			
Function Name	Required Rights	Error Conditions	Summary
std_destroy	RUN_RGT_DESTROY	STD_CAPBAD STD_DENIED	Destroy a *run* server object
std_info	RUN_RGT_STATUS on run object or super cap	STD_CAPBAD STD_DENIED	Print standard information for a *run* server object
std_restrict	none	STD_CAPBAD STD_DENIED	Produce a *run* server capability with restricted rights
run_create	RUN_RGT_CREATE on super cap	STD_CAPBAD STD_NOSPACE STD_DENIED	Create a new *run* server object for a pool directory
run_get_exec_cap	RUN_RGT_FINDHOST	STD_CAPBAD STD_DENIED STD_ARGBAD STD_NOSPACE STD_NOTNOW	Return exec cap and architecture corresponding to specified process svr
run_multi_findhost	RUN_RGT_FINDHOST	STD_CAPBAD STD_DENIED STD_ARGBAD STD_NOSPACE STD_NOTNOW	Return best host in pool for executing one of a set of process descriptors (for heterogeneous process startup)

Administration

The first time the *run* server is started it should be started with the **–i** option. This will create the necessary administrative files used by the server.

In normal operation, one *run* server should be started for all *pooldir*s together. If *pooldir*s are disjunct (i.e., no host is a member two or more pools) increased performance may be achieved by starting a *run* server for each *pooldir*. In this case, the **–n** option should be used to give the *run* servers a different name.

Creating a new pool can be done by first creating a minimal pool directory containing at least one *archdir*. In order to be useful, the *archdir*s should contain one or more host capabilities of the corresponding architecture. Public hosts can be found in the directory */super/hosts*. The command *makepool*(U) with the pool directory as argument should then be used to make it known to the *run* server, and retrieve and install a capability for it. The *run* server will also make a link to the pool directory created so that the system administrator has an overview of all pools under the server's control. It is then a simple matter to use the *std_params*(A) command to enable and disable various pools.

Before making a new pool directory the default, it is advisable to test it for a while by starting a sub-shell that uses the pool. To do this, set the environment variable RUN_SERVER as follows:

```
$ RUN_SERVER=/profile/newpool/.run /bin/sh
```

When a processor is added to or removed from an *archdir*, a *run* server managing it should be told about this by means of a *std_touch* on the corresponding run capability. If the *run*

server's pool objects are brought under the control of the object manager *om*(A), rescans will typically be performed every hour. It is a good idea to put the *run* server into the touching and aging system of *om* to ensure that temporary pools (such as those used for running Orca programs) are garbage collected and that the resources within the *run* server are thereby released. The default lifetime of a pool is 10 hours.

It is a good idea to experiment a little with the host evaluation formula parameters using *std_params*(A).

For example,

```
std_params -s MEMTRUNC=1024 /super/pool/.run
```

causes the *run* server to treat available memory exceeding 1024 KB as irrelevant to the decision about which host to pick from the public pool. (That is, if there is more than 1024 KB then there is plenty and other factors such as CPU load should take precedence.)

After this, the command

```
std_params -v /super/pool/.run
```

might show something like the following:

```
EQUIVPERC    [0..50 %]          5 (hosts in this range considered equivalent)
MEMTRUNC     [1..64000 Kb]   1024 (ignore free memory exceeding it)
CACHETIME    [1..10000 Kb]    500 (twofold speedup at this text size)
CACHEMEM     [1..200 %]        10 (text segment sharing advantage)
PREF-mc68000 [0..1000 %]      100 (architecture preference multiplier)
PREF-sparc   [0..1000 %]      100 (architecture preference multiplier)
ENABLE       [0..1]             1 (enable/disable pool)
```

Valid parameters are:

Parameter	Min	Max	Default	Description
EQUIVPERC	0	50 %	5 %	hosts whose *value* function is ±EQUIVPERC of each other are considered equivalent
MEMTRUNC	1	64000 KB	2048 KB	hosts with more than MEMTRUNC free memory are considered to have the same amount of memory
CACHETIME	1	10000 KB	500 KB	If the text segment is already cached a 2x speedup is expected for a CACHETIME sized text segment
CACHEMEM	1	200 %	10 %	We expect an average savings of CACHEMEM % of the text size if the text segment is found cached due to it possibly already being in use
PREF-i80386 PREF-mc68000 PREF-vax PREF-sparc	0 0 0 0	1000 % 1000 % 1000 % 1000 %	100 % 100 % 100 % 100 %	For every architecture in a pool this parameter this parameter promotes or discourages its use (e.g., to account for slow rpc or process startup time)
ENABLE	0	1	1	enable/disable the pool

When hosts with highly differing CPU power and/or available memory are present in a pool, it often happens that the fastest host attracts all the work until it is quite overloaded. Although this might seem unfair, most of the time this gives just what one wants (i.e., executing all available jobs as fast as possible). If, however, the performance as predicted by the available CPU power is an exaggeration — for example because the RPC performance of the fast host is worse than that of slower hosts — then one could try to adjust some of the `PREF-arch` parameters.

Another way to make host selection a bit less predicatable is to increase the `EQUIVPERC` parameter for a pool.

It is a good idea to regularly monitor the status of the public pool directory with

```
std_status /super/pool/.run
```

This gives a status report of the following form:

```
HOST     STAT SINCE LREPL LPOLL NPOLL NBEST NCONS   MIPS  NRUN  MBYTE
bullet3  UP    3:10    0s    0s  2285     7    26  3.700 2.988 11.012
laser    UP    6:38    4s    4s  4783     2    26  3.700 1.748  0.864
vmesc1   UP    6:38    0s    0s  4791    10   112  3.000 0.116  2.776
armada06 UP    6:38    4s    4s  4785     9   112  3.000 1.042  2.451
armada05 UP    6:38    2s    1s  4783    13   112  3.000 0.244  2.144
armada04 UP    6:38    1s    1s  4785    10   112  3.000 0.986  1.696
armada03 UP    6:38    0s    0s  4787    11   112  3.000 0.734  2.880
armada02 UP    6:38    2s    2s  4803    38   112  3.000 0.209  2.704
armada01 UP    6:38    5s    5s  4783     4   112  3.000 0.827  2.968
armada00 UP    6:38    4s    4s  4783     4   112  3.000 0.093  2.968
blt      UP    6:38    1s    1s  4783     0    18  0.900 0.000  3.650
hawaii   UP    6:38    4s    4s  4784     4    19 12.500 1.551 10.060
```

The `HOST` column shows the host name; the `STAT` column its status (`UP` or `DOWN`); the `SINCE` column shows the time since the last status change (‘‘hh:mm’’ or various other formats). The columns `LREPL`, `LPOLL` and `NPOLL` are probably only useful for debugging the *run* server; they give host-polling statistics.

The columns `NBEST` and `NCONS` give interesting information about the *run* server’s host selection; the former tells how many times a particular host was selected as being the ‘‘best’’, while the latter tells how many times the host was considered for selection. The columns `MIPS`, `NRUN` and `MBYTE` give the load as returned by *pro_sgetload*.

See Also

ampolicy(H), ax(U), exec_file(L), exec_findhost(L), makepool(U), om(A), process(L), std(L), std_params(A).

Name

sak – the Swiss Army Knife server

Synopsis

sak [-I] [-W] [-w name] [-r] [-p dir] [-c name] [-n #threads]
working-directory

Description

The *Swiss army knife* (*sak*) server provides a mechanism for execution of transactions at a later date. A file containing details of a transaction and the environment required for its execution are submitted to the server along with a *schedule* describing under what condition(s) the transaction should be executed. At present, schedules are restricted to conditions based on time. The *sak* server assumes that all schedules are specified relative to the timezone of the server. Besides the standard server commands (see *std*(L)) it provides two other commands: one to submit jobs and one to list jobs.

Because there is not yet a suitable *exec* server there is also a special thread in the server that accepts a command to execute a program. This thread is normally accessed by submitting a special job using the *submit-job* command which does a transaction with the special thread. See *sak*(L) for details of how to do this.

The *sak* server guarantees 0 or 1 executions of a transaction. If the server notices that a job was not run at the correct time, the client can specify if a job needs to be discarded or executed immediately. Return data from executed transactions are discarded. A return status can be optionally reported.

Options

–I Install the server and start running. This option should be used the first time the server is run and thereafter omitted.

–W Enable the special *exec* server so that the *sak* server can execute programs at a particular time.

–c *name*
Publish the transaction server capability under *name*.

–n *#threads*
Use *#threads* server threads.

–p *dir*
Publish the server capability in the directory *dir*.

–r Republish the server capability.

–w *name*
Publish the exec server capability under *name*.

The *working-directory* command line argument is used as the work directory. Here it looks for the *jobs* and *trans* directories and checks for existing jobs. In this directory there has to be a masterfile (named *sak_masterfile*) so it can restart with the old server capability (unless the **–I** flag is specified).

Programming Interface Summary

The programming interface consists of commands particular to the server (whose names begin with *sak_*) and the standard server commands (whose names begin with *std_*). A summary of the supported commands is presented in the following two tables. For a complete description of the interface routines see *sak*(L) and *std*(L).

Standard Functions			
Function Name	Required Rights	Error Conditions	Summary
std_age	OWNERRIGHTS in server cap	RPC error STD_CAPBAD STD_DENIED STD_COMBAD on job cap	Start garbage collection pass.
std_info		RPC error STD_CAPBAD	Returns name of the sak job.
std_restrict		RPC error STD_CAPBAD	Returns a capability with restricted rights.
std_touch		RPC error STD_CAPBAD	Reset life-time of the job.
std_destroy	OWNERRIGHTS	RPC error STD_DENIED STD_CAPBAD STD_COMBAD on server cap	Cancel the job.

SAK Functions			
Function Name	Required Rights	Error Conditions	Summary
sak_list_job	OWNERRIGHTS	RPC error STD_CAPBAD STD_DENIED STD_COMBAD on server cap STD_NOMEM	Return schedule and options of the job.
sak_submit_job		RPC error SAK_TOOLATE STD_CAPBAD STD_SYSERR STD_ARGBAD STD_NOMEM STD_COMBAD on job cap	Submit new job.

Administration

Administration is trivial. The server can be started in two ways. The first time it will have to be started in install mode, i.e., with the **–I** flag. After that it can be restarted and will continue where the previous server left off. No administration is needed once the server is running.

Installation

If the server is started with the **–I** flag it will try to install a new server in the specified directory. This directory should be empty and inaccessible to normal users. Here it will create two directories named *jobs* and *trans*. All job and transaction files are stored in these directories. Next a masterfile named *sak_masterfile* is created. The masterfile is needed to resume a server after it has died. The default place for the server capability to be installed is */super/cap/saksvr/default*. This can be modified using the **–p** *directory* and **–c** *name* options. Before starting operation, the server will make sure that no old server is running.

Once installed the server can be stopped and restarted at any time. It will read the masterfile, look for jobs in the jobs directory and continue operation. It assumes the server capability is still in place. If by accident the server capability was deleted or corrupted, it can be reinstalled using the **–r** option.

Notes

The *sak* server can only execute transactions. Since there does not exist a transaction to execute a command directly, the *sak* server provides an additional service to execute a command specified by an argument list, string environment list and capability environment list. The service is provided by a special thread within the *sak* server and is completely transparent to the rest of the server. This thread accepts requests on a separate port. Only one type of request is allowed: `SAK_EXEC_FILE`. The buffer should contain a NULL-terminated argument list, a NULL-terminated string environment list and a NULL-terminated capability environment list. This can be used in combination with the normal *sak* server to execute files instead of transactions. For further details see *sak*(L).

The server will only provide the exec service if **–W** is specified on the command line. Since this service uses a separate port, its server capability is installed in */super/cap/saksvr/execdefault* at installation time. This can be modified using the **–p** *directory* and **–w** *name* options.

See Also

at(U), cronsubmit(U), sak(L), std(L).

(UNIX ONLY)

Name

sendcap – send a new user his .capability file

Synopsis

```
sendcap amoeba-login-name [ unix-login-name ]
```

Description

Sendcap runs only under UNIX and requires that the FLIP driver be installed. It extracts from Amoeba the capability of the home directory of the user with *amoeba-login-name*. It converts this to a *.capability* file, uuencodes it and mails it to the UNIX mailbox of the user. If the user's UNIX login name is different from the login name under Amoeba then the *unix-login-name* parameter should be given. It is probably a good idea to modify the message sent to the user to reflect any local conditions or provide introductory information to the new user.

Example

```
sendcap gbh
```

will get the capability for the home directory of the Amoeba user *gbh* and mails it to the UNIX user *gbh*.

See Also

newuser(A).

Name

setupipsvr – tell an IP server its Internet address and netmask.

Synopsis

```
setupipsvr hostname
```

Description

When an IP server starts it needs to be told its Internet address and netmask. *Setupipsvr* provides a convenient way of doing this from the *boot*(A) server. The way it works is as follows:

It does the *ifconfig*(A) necessary to tell the IP server its address information. It gets this information from the file */etc/ipinfo* which must be kept in parallel with the DNS or the file */etc/hosts*.

If that succeeded then it stores a copy of the IP server capability (from */super/hosts/*`hostname`*/ip/ip*) under the name */super/admin/module/ipsvr/*`hostname`.

The *boot*(A) server uses */super/admin/module/ipsvr/*`hostname` as a poll capability. When an IP server starts it always chooses a new capability. Therefore, as long as the poll capability is valid, the *boot*(A) server does not need to reconfigure the IP server. *Setupipsvr* will only be run when the IP server is down or just restarted. Once it has successfully initialized the IP server it will not run again until the IP server is again unavailable.

The argument to *setupipsvr* is the *hostname* of the host where the IP server runs.

Files

/etc/ipinfo – this file contains (hostname, ip-address, netmask) triplets.

Example

Below is a sample bootfile entry for using *setupipsvr*.

Note that since *setupipsvr* is a shell script it must be started with a shell and requires a *session*(U) server to be present.

```
IP_google {
    after Session_svr;
    machine SLASH "super/hosts/google";
    bootrate 17000;
    pollrate 30000;
    program {
          SLASH "bin/sh";
          argv {
              "sh", "-c", "/super/admin/bin/setupipsvr google"
          };
          capv {
              _SESSION = SLASH "dev/session",
              STDIN, STDOUT, STDERR,
              ROOT = SLASH, WORK = SLASH
          };
    };
};
```

See Also

boot(A), ifconfig(A), rarp(A).

Name

showbadblk – Amoeba bad block utility

Synopsis

```
showbadblk disk-capability
```

Description

Showbadblk displays the contents of the bad block table. When no table can be found, or when it is corrupted an appropriate error message is reported.

Example

To display the bad blocks on machine ‘‘bullet’’, the following command should be issued:

```
showbadblk /super/hosts/bullet/bootp:00
```

See Also

fdisk(A), vdisk(A).

Name

soap – the Amoeba directory server

Synopsis

```
soap [-F] [-k max_kbytes] [-n] [-s nthreads] [0|1]
```

Description

Soap† is the Amoeba directory server. It is the naming service for all objects in Amoeba. It provides a mapping from an ASCII string to a set of capabilities. The capability-set contains the capabilities for various copies of the same object.

The server can be run in duplicate so that there are two copies of the program maintaining the data. They communicate with each other via a private port and use a special protocol to maintain data consistency. If one of the servers goes down the other will continue alone and bring its partner up to date when it is restarted. (Therefore, it is also clear that the server can be run without a partner.) When two instances of the server are working together they are said to be in *2 copy mode*. When there is only one instance of the server running (either by design or due to the failure of one server) then the server is in *1 copy mode*. The command line parameter is either 0 or 1. This tells the server which position it holds in a duplicated server. If you are only running a single server then you should start it as server 0. In general server 0 performs certain tasks in preference to server 1, although, in the absence of server 0, server 1 will assume these responsibilities.

It is possible to duplicate the data used by *soap*, either in single or duplicated server mode. *Soap* has two sets of data. On a separate disk partition it keeps details of directories. There is one such partition needed for each partner. Thus, with two partners this data will be replicated. Then for each directory it keeps a bullet file describing the entries in the directories. It has a provision for duplicating the bullet files. If there are two Bullet Servers then they can be used to duplicate the bullet files. The capabilities for the file servers can be given at installation time or added or modified later using the *chbul*(A) command.

Part of the data consistency protocol involves keeping sequence numbers so that the most up to date version of the data is represented by the highest sequence number. This also has implications for the recovery mechanism. If a soap server goes down while in *2 copy mode* it will need to contact the other server (which is likely to have a higher sequence number) during initialization in order to get all directories consistent again. If the other one is unavailable during recovery, the server can only wait and try again later.

If a replicated Soap Server that is trying to make a directory modification is not able to reach its partner right away, it will retry sending the intention for about 30 seconds. If it still fails, it will assume the other side has crashed, and switch over to *1 copy mode*. The server may then safely assume it has the most up-to-date directory state, so if it goes down later on as well, it can recover without waiting for the other side to become available.

Note: the protocol used works as long as there is no network partition between the two replicas. To handle this situation as well, at least three replicas would be required.

———————

† The name *soap* is an acronym derived from *in Search Of A Problem*.

Options

The Soap Server recognizes the following command line options:

−F This forces a switch to one copy mode if the Soap Server cannot reach its partner during startup. The option should *only* be given if the other server is unable to start up properly, e.g. because vital data has been overwritten.

−k max_kbytes
Soap usually cannot keep all the directories in core at the same time, so it implements a cache of directories. The maximum size of this cache can be limited to *max_kbytes* using the **−k** option. The default value is 400.

−n By default, during recovery the Soap Server tries to read in all its directories, to make sure all directory replicas are valid and consistent. This time-consuming procedure is skipped when the **−n** option is given.

−s nthreads
The default number of threads handling directory operations is 6. The **−s** option can be used to force a different number of server threads.

Diagnostics

All diagnostics from *soap* are prefixed with `SOAP x:` where *x* is the server number (either 0 or 1). When *soap* starts up it prints (in this example server 0)

```
SOAP 0: initializing
SOAP 0: Bullet 0 is up
```

If it is using two Bullet Servers then it will also print

```
SOAP 0: Bullet 1 is up
```

and then details about its cache of directories. These are of the form

```
SOAP 0: cache_init: message
```

Where *message* describes some characteristic of the directory cache. Then it prints

```
SOAP 0: using n copy mode
```

where *n* is either 1 or 2. If it says that it is in 2 copy mode then it may be out of date with respect to the other server and have to synchronize with it. In that case it prints

```
SOAP 0: need help from the other side
```

followed by

```
SOAP 0: initialized after help from the other side
```

If there was no recovery to do it will print

```
SOAP 0: our seq_nrs are the same; no recovery necessary
```

Once it is initialized it will print

```
SOAP 0: 6 threads started
```

At this point *soap* is running and ready to accept commands. If the server is duplicated the partner will print the message

```
SOAP 1: other side is back
```

Similarly, if a server detects the absence of its partner (due to crash, reboot or hardware failure of some sort) it will print

```
SOAP 1: other side seems to have crashed
```

If one of the Bullet Servers in use by *soap* goes down and then comes back up then *soap* will print

```
SOAP 0: Bullet n is back up
```

where *n* refers to the number of the Bullet Server as *soap* numbers them.

During the course of starting up *soap* may detect bad data on its virtual disk or in its bullet files. In this case it will print details of what the problem was and how it corrected it.

Environment Variables

SOAP_SUPER must be set in the capability environment and specifies the capability for the disk with the server's directory information.

TOD is a capability environment variable which specifies the time of day server that *soap* should use. If it is not set then *soap* will attempt to use the default time of day server. However, to do this it will have to do a name lookup for the default. This means that there must be another Soap Server running. For reliability, therefore, this variable should always be given in the environment.

RANDOM is a capability environment variable that specifies which random server that *soap* should use. Like TOD, this should always be specified in case there is no other Soap Server running.

Files

Soap keeps the directories in bullet files. It ensures that they are touched and kept consistent. If given two Bullet Servers, *soap* will also ensure that the files are correctly duplicated and regularly touched so that they are not garbage collected. If *soap* is run in two copy mode then each will show a preference for the Bullet Server that corresponds to its copy. This spreads the load evenly over the Bullet Servers.

Warnings

A lot has been done to make the server robust in the face of errors in its data files, but it is still possible that it can get inconsistencies in the data that it cannot recover from. In that case you will have to pick the server you think has the best picture of the world and copy its super file to that of its partner. (Make a backup of the original first of course, by dumping it to a bullet file.)

Programming Interface Summary

Besides the standard server commands *soap* accepts a large number of specialized directory commands. They are summarized in the tables below. For more details of these commands see *std*(L) and *soap*(L).

In the following table all routines have the following errors in common. The table shows only the additional errors particular to each command.

RPC_*	a generic RPC failure
STD_ARGBAD	name too long, etc.
STD_CAPBAD	capability has wrong random check field
STD_DENIED	capability lacks rights the operation requires
STD_NOSPACE	lack of space at the Soap Server
SP_UNAVAIL	equivalent to RPC_NOTFOUND
SP_UNREACH	capability refers to nonexisting directory

Soap Functions

Function Name	Required Rights	Error Conditions	Summary
sp_append	SP_MODRGT	STD_EXISTS	Adds an entry to a directory.
sp_chmod	SP_MODRGT	STD_NOTFOUND	Changes the column masks on a directory entry.
sp_create	READ		Creates a new directory object but does not enter it into another directory.
sp_delete	SP_MODRGT	STD_NOTFOUND	Deletes a directory entry (not a directory).
sp_getmasks	READ	STD_NOTFOUND	Returns the current column masks on a directory entry.
sp_install	SP_MODRGT	SP_CLASH	Atomically update a set of directory entries (all in the same soap server).
sp_lookup	READ	STD_NOTFOUND	Returns the capability-set for a single directory entry.
sp_replace	SP_MODRGT	STD_NOTFOUND	Atomically replaces a single directory entry's capability-set.
sp_setlookup	READ	STD_NOTFOUND	Returns the capability-sets for several specified entries.
sp_list	READ		Returns all the names and masks in a directory.

Standard Functions			
Function Name	Required Rights	Error Conditions	Summary
std_age	SUPERCAP	STD_NOTNOW STD_DENIED STD_CAPBAD	Decrements the time-to-live field of all directories, and removes the unused directories.
std_destroy	SP_MODRGT	SP_NOTEMPTY STD_DENIED STD_CAPBAD	Destroys the specified directory, which must be empty.
std_info	READ	STD_DENIED STD_CAPBAD	Returns string of the form "/dw1234" telling the rights in the directory capability.
std_restrict		STD_CAPBAD	Returns a directory capability with restricted rights.
std_status			Returns an ASCII string of statistics about Soap.
std_touch	READ	STD_DENIED STD_CAPBAD	Resets the time-to-live field of the directory and touches the bullet files storing it.

Administration

Installation

The program *makesuper*(A), which creates the super files and the first directory, is used for this. With *makesuper* one also determines whether the server will initially use multiple bullet files (by specifying multiple Bullet Servers instead of just one) and whether the server will be duplicated or always run in single copy mode. The correct way to use *makesuper* is described in the installation guide.

Maintenance

In general most errors in data files used by *soap* are automatically repaired. When they are not, *sp_fix*(A) can be used to manually delete bad directories (specified by their object number, the same number printed by *soap* in the error messages complaining about the directory). This is necessary when *soap* reports errors in reading directories, and the errors do not go away when *soap* is restarted.

If you modify the Soap Server then the new version can be installed without any down time if the server is duplicated. Simply install the new server in the appropriate place and then kill one of the servers (say server 1). Restart the new version in its place. Once the new version has recovered and started serving, kill the other partner (server 0) and start the new version in its place. (Of course, this will only work if you do not change the protocol that the servers use to communicate with each other.)

It is possible to change the bullet servers used by *soap* with the command *chbul*(A). By giving a null capability to *chbul* it will stop it using a second bullet server at all. After you

have run *chbul* it is important to let the Soap Server have time to migrate its files to the new server. It migrates at most 11 files per minute at present. You can follow the progress of this using the *monitor*(U) command on the private port of server 0 (usually found in */super/cap/soapsvr/soap0*).

Example

Normally *soap* will be started by the boot server (see *boot*(A)). There is an example of how this is done in the standard boot file of the distribution. If however it is necessary to start it from UNIX, the following command will start server 1 on machine *host* if another Soap Server is already running. Of course, you must select the correct disk partition for this server!

```
ax -c -m host -C SOAP_SUPER=/super/hosts/host/vdisk:02 \
              -C TOD=/super/cap/todsvr/default         \
               -C RANDOM=/super/cap/randomsvr/default   \
              /super/admin/bin/soap 1 &
```

File Formats

There are two file formats used by *soap*. One is for its super file and the other for the bullet files that hold the actual directory contents. For maximum flexibility, both formats are written in a byte order independent way.

The super file consists of a super block (block 0 of the disk) followed by an array of capability pairs. The super block structure is defined in *h/server/soap/super.h* and consists of a magic number followed by the following information:

- the private capabilities for the two Soap Servers,
- the public capability that both servers listen to
- the current sequence number (used in 2 copy mode to determine which server has the most up-to-date information)
- the last known copy mode
- the number of blocks of capability data following the super block.
- an intentions list consisting of directory updates that were accepted, but still need to be installed in the super table. At present there is room for 11 intentions.

The rest of the super file is a list of capability pairs. The first pair is the set of capabilities for the actual bullet servers to use. (No directory can have object number 0.) The subsequent pairs are the capabilities for the bullet files containing the directory data.

The format of the bullet files used to store the actual directory data is defined in the file *h/server/soap/soapsvr.h* and is as follows: It consists of a *struct sp_dir* which contains a sequence number, which is used by *starch*(U), the number of rows, the number of columns and the name of the columns plus the check field for the directory object. This is followed by a number of *sp_row* structs (as many as described in the *sp_dir* struct). Each row consists of an character string which is the name of the entry followed by the creation time and a capability-set. Appended to each row is also a set of rights masks, one for each column in the directory.

See Also

chbul(A), chm(U), del(U), dir(U), get(U), makesuper(A), mkd(U), monitor(U), put(U), soap(L), sp_dir(L), sp_mask(L), sp_mkdir(L), std(L), std_age(A), std_info(U), std_restrict(U), std_status(U), std_touch(U), vdisk(A).

Name

sp_fix – delete or get the caps for a SOAP directory

Synopsis

```
sp_fix [ -g | -d ]  soap-disk-partition  dir-number
```

Description

Given the object number for a directory (specified in decimal, not hex), *sp_fix* will either clear its Bullet file capabilities in the *soap* disk partition, effectively deleting it (**–d** flag), or will print the capabilities to *stdout* in the form expected by the *put*(U) command (**–g** flag). This command is only intended to be used as a last ditch measure when *soap* cannot fix a problem with a directory itself.

The safest way to delete the bad directory is to first use

```
sp_fix -g
```

piped into a *put* command, to get the capset for its Bullet files and enter them into the directory graph. At this point they can be examined and manipulated just like any other Bullet files. Then use

```
del -f
```

to destroy the Bullet files. Finally use

```
sp_fix -d
```

to clear the capabilities in the *soap* disk partition.

Examples

```
sp_fix -g /super/hosts/bullet0/vdisk:02 1488 | put /home/tmp/dir.1488
```

will enter the capabilities for the Bullet files of directory 1488 under the path name */home/tmp/dir.1488*.

See Also

capset(L), soap(A).

Name

std_age – cause a server to do garbage collection

Synopsis

```
std_age server_cap
```

Description

Std_age causes the standard server command STD_AGE to be executed on the server specified by *server_cap*. The capability may be required to have special rights and/or object number before the server will execute the command. The idea behind the command is that every virtual object has a maximum life time. Any object not *touched* (see *std_touch*(U)) or otherwise accessed within a certain time period is probably no longer in use and can be destroyed. This effectively deals with the problem of people creating objects but then losing the capability for them.

Normally a special server touches all the objects that it has capabilities for and then sends the STD_AGE command to all the registered servers to cause garbage collection. However this command provides a utility to deal with malicious or incompetent users' overuse of resources.

This command is not applicable to physical objects, such as disks. Servers not capable of supporting this command will reject or ignore it.

Diagnostics

If all is well the command exits with status zero. The only errors are that the look up of the *server_cap* in the directory server failed or that the STD_AGE command failed. In both cases the command prints a message detailing the error and exits with non-zero status.

Warnings

Repetitive invocation of this command can lead to the destruction of objects. It should be used as a tool to repair damaged or over-full systems and not be randomly invoked.

Example

```
std_age /super/hosts/bullet1/bullet
```

causes the STD_AGE command to be run on the Bullet Server on the host *bullet1*. This will result in all its files that have not been touched in the past 24 *std_age* commands being deleted.

See Also

std_touch(U).

Name

std_params – set and/or show server parameters

Synopsis

```
std_params [-s param=value] [-v] server
```

Description

Std_params can be used to print or modify the runtime parameters of a server implementing the *std_getparams/std_setparams* interface. In general, server parameters can be global (in which case *std_params* should probably be applied to the server's super capability) or per object. In order to be as flexible as possible, the *std_params* interface is string-based, and conversions needed (e.g., to integer or boolean) have to be done by the server itself.

Options

–s *param=value* Sets parameter *param* to *value*. Multiple **–s** arguments are allowed, but *std_params* uses a separate RPC for each parameter.

–v Gives a verbose overview of the server's parameters. The information shown is the parameter's name, its type, its current value, and a short description telling what it does.

If no options are given, *std_params* just prints the parameters and their values.

Diagnostics

Error messages — for example, the server's refusal to set a parameter — are self-explanatory.

Example

The command

```
std_params -v /super/pool/.run
```

gives a verbose overview of the parameters for the run object (see *run*(A)) corresponding to the public pool directory. It might print something like the following

```
EQUIVPERC    [0..50 %]        5 (hosts in this range considered equivalent)
MEMTRUNC     [1..64000 Kb] 2048 (ignore free memory exceeding it)
CACHETIME    [1..10000 Kb]  500 (twofold speedup at this text size)
CACHEMEM     [1..200 %]      10 (text segment sharing advantage)
PREF-i80386  [0..1000 %]    100 (architecture preference multiplier)
PREF-mc68000 [0..1000 %]    100 (architecture preference multiplier)
PREF-sparc   [0..1000 %]     50 (architecture preference multiplier)
```

The parameter `MEMTRUNC` for this pool can be decreased to `1024` using the command

```
std_params -s MEMTRUNC=1024 /super/pool/.run
```

After this, the new values can be shown by

```
std_params /super/pool/.run
```

which will print

```
EQUIVPERC        5
MEMTRUNC      1024
CACHETIME      500
CACHEMEM        10
PREF-i80386    100
PREF-mc68000   100
PREF-sparc      50
```

See Also

run(A), std(L).

Name

syssvr – the kernel system server

Synopsis

Built into the kernel.

Description

The *syssvr* is a special server in the kernel that accepts system commands such as *kstat*(A) and *reboot*(A). If no printbuf server is present (see *printbufsvr*(A)) then it also provides a primitive *printbuf* interface.

At initialization time the *syssvr* publishes its capability in the kernel directory under the name *sys*. If it is present in a boot ROM then a *std_info*(U) command on the *sys* capability will return the string ‘‘bootstrap kernel’’. For a normal kernel it will return ‘‘system server’’.

Programming Interface Summary

The actual functions available in the system server vary with the compile-time flags used to compile it. Below is the set of commands accepted when it is fully configured.

Available Functions

Function Name	Required Rights	Error Conditions	Summary
std_info	NONE	RPC error	Print identifying string.
std_restrict	NONE	RPC error	Restrict the rights in a capability.
std_getparams	NONE	RPC error	Currently a noop.
std_setparams	NONE	RPC error	Currently a noop.
sys_boot	SYS_RGT_BOOT	RPC error STD_DENIED	Reboot with new kernel.
sys_printbuf	SYS_RGT_READ	RPC error STD_DENIED	Return console buffer contents.
sys_kstat	SYS_RGT_READ	RPC error STD_ARGBAD STD_DENIED	Return selected kernel statistics.

Note that there are plans to implement *std_setparams* and *std_getparams* so that various network and other parameters can be set interactively.

Administration

The only administration required is the decision as to whether or not a kernel should have a *syssvr*. In almost all cases it is a good idea. The only exception is when a very small kernel is required, such as for a boot ROM.

See Also

printbufsvr(A).

Name

tape – a server interface to tape controllers

Synopsis

Currently built into the kernel

Description

A tape is a record-oriented backup/restore device. The smallest addressable unit is a record. The record size can vary from 1 byte to `MAXINT` bytes, depending on the tape hardware and host operating system (due to the maximum transaction buffer size of Amoeba, a maximum record size of 29696 bytes is used).

The end of a sequence of records can be marked by an *End-Of-File* (EOF) marker. This marker is a special record which can be used to define tape files of records. Most tape controllers are able to search for EOF markers.

The tape server provides a general interface to tape devices. Clients use the tape server to control tape devices. The tape server uses internal tape drivers to access the tape devices.

A tape device consists of one or more tape units. A tape unit is identified by a capability.

The tape server consists of two parts: the tape server itself and the internal tape drivers. The following section deals with the tape server interface, followed by sections concerning tape drivers.

Tape server interface

Each tape operation handled by the tape server requires a tape capability, This capability can be found in the tape unit's host machine directory. The capability is always formed as follows:

`/profile/hosts/`*machine*`/tape:`*number*

The *number* parameter specifies which internal driver and tape unit is concerned.

The tape server recognizes a set of predefined commands. All commands return an error code of the operation, which can be one of the standard Amoeba errors or tape I/O errors.

The standard errors are predefined constants (see *stderr*(L)). `STD_COMBAD` is returned when not using the predefined tape commands, `STD_CAPBAD` is returned when using bad capabilities and `STD_ARGBAD` is returned when passing bad arguments to tape stubs (e.g., a record size which is too big). Upon a successful operation `STD_OK` is returned.

Since tape actions involve Amoeba RPCs, RPC errors might be returned (see *rpc*(L)).

The possible tape I/O error codes are described below.

TAPE_CMD_ABORTED	The last command has been aborted.
TAPE_NOTAPE	The tape unit is off-line.
TAPE_MEDIA_ERR	Tape format error.
TAPE_WRITE_PROT	The tape is write protected.
TS_CMP_ERR	Cannot write data to tape.
TS_DATA_ERR	Invalid record on tape.

TAPE_BOT_ERR	While repositioning, the beginning of tape (BOT) marker was detected.
TAPE_EOF	The end of file (EOF) marker was detected.
TAPE_EOT	The end of tape (EOT) marker was detected.
TAPE_REC_DAT_TRUNC	Tape record longer than expected.
TAPE_POS_LOST	Tape record position lost.

The following summary specifies which operations are valid for the tape server, along with a short description. For the interface specification, see *tape*(L). Along with the tape commands (commands which are prefixed by *tape_*), the tape server recognizes standard commands (commands which are prefixed by *std_*, see *std*(L)).

Currently, rights are not implemented in the tape server. Everybody has full access to the tape server.

The interface is also described in the library sections of the manual. (see *tape*(L) and *std*(L)).

std_info	At a *std_info* call, the tape server request the tape driver an identification of a tape unit. The tape server returns a null terminated, formatted ASCII string.
tape_init	*Tape_init* initializes a tape unit and also loads an inserted tape.
tape_read	*Tape_read* reads a tape record from a tape.
tape_write	*Tape_write* writes a tape record onto a tape.
tape_write_eof	*Tape_write_eof* writes an EOF marker onto the tape.
tape_rewind	*Tape_rewind* rewinds a tape to begin of tape. Users can specify to rewind the tape synchronously or asynchronously. When synchronous, *tape_rewind* finishes once the tape is completely rewound. In the latter case, the command finishes directly; it does not wait until the tape is actually rewound.
tape_status	*Tape_status* requests the status of a tape unit. *Tape_status* returns a null terminated formatted ASCII string which contains the complete status of a unit. The format is tape driver dependent. For a description of the contents, see the tape driver sections.
tape_rskip	*Tape_rskip* skips a number of records on tape. Negative skip counts result in backward skipping.
tape_fskip	*Tape_fskip* skips a number of EOF markers on tape. Negative counts result in backwards skipping. Forward skipping causes the new tape position to be at the beginning of the file, negative skips causes the new tape position to be at the EOF marker.
tape_erase	*Tape_erase* erases a tape. It formats every inch on tape and since tapes can be very long and slow, this command may take some time to finish.
tape_setlvl	*Tape_setlvl* is for debugging tape drivers only. It is used for setting some debug output level in the driver.
tape_why	*Tape_why* converts an Amoeba or tape error code to a null terminated ASCII string (in fact it returns a pointer to an internally kept error table). This routine does not require a capability.

See Also

rpc(L), std(L), stderr(L), tape(L), tape(U).

Name

telnetd – DARPA TELNET protocol daemon

Synopsis

```
telnetd [-debug] [-D (options|report|netdata|ptydata|exercise)]
          [-h] [-l] [port]
```

Description

Telnetd is a server which supports the DARPA standard TELNET virtual terminal protocol. Under Amoeba *telnetd* can only be invoked via the *telnetsvr*(A) server, normally for requests to connect to the TELNET port as indicated by the */etc/services* file. It **cannot** be started from the command line with an alternate TCP port number *port*. To do this it is necessary to modify the *telnetsvr* to pass extra arguments to the *telnetd*. The rest of the description below is purely for information only.

Telnetd implements the various commands and a pseudo-terminal and creates a login process which has the slave side of the pseudo-terminal as *stdin*, *stdout*, and *stderr*. *Telnetd* manipulates the master side of the pseudo-terminal, implementing the TELNET protocol and passing characters between the remote client and the login process. When a TELNET session is started up, *telnetd* sends TELNET options to the client side indicating a willingness to do *remote echo* of characters, to *suppress go ahead*, to do *remote flow control*, and to receive *terminal type information*, *terminal speed information*, and *window size information* from the remote client. If the remote client is willing, the remote terminal type is propagated in the environment of the created login process. The pseudo-terminal in the telnet server operates in cooked mode.

Telnetd is willing to *do*: *echo*, *binary*, *suppress go ahead*, and *timing mark*. *Telnetd* is willing to have the remote client *do*: *linemode*, *binary*, *terminal type*, *terminal speed*, *window size*, *toggle flow control*, *environment*, *X display location*, and *suppress go ahead*.

Options

–debug Causes debugging information to be printed.

–D*flag* This option is used to selectively enable various diagnostics. This allows *telnet* to print out debugging information to the connection, allowing the user to see what *telnetd* is doing. There are several *flags*:

- *options* prints information about the negotiation of TELNET options.
- *report* prints the *options* information, plus some additional information about what processing is going on.
- *netdata* displays the data stream received by *telnetd*.
- *ptydata* displays data written to the pty.
- *exercise* does nothing at present.

–h causes the host information banner not to be sent to the client on startup.

–l always use *linemode* where possible.

Environment Variables

The string environment variable TCPIP_CAP must be set to the output of *c2a* of the capability of the already open TCP/IP channel to used. The already open channel is provided by *telnetsvr*.

Warnings

Some TELNET commands are only partially implemented.

Because of bugs in the original 4.2 BSD *telnet*, *telnetd* performs some dubious protocol exchanges to try to discover if the remote client is, in fact, a 4.2 BSD *telnet*.

The terminal type name received from the remote client is converted to lower case.

Telnetd never sends TELNET *go ahead* commands.

See Also

telnetsvr(A), ttn(U).

Name

telnetsvr – a front-end for the telnet daemon

Synopsis

```
telnetsvr [-T tcp-cap] [-daemon telnetd-prog] [-v]
```

Description

The *telnetsvr* is used to start telnet daemons (see *telnetd*(A)). When a client tries to make a telnet connection, *telnetsvr* starts a telnet daemon for that client.

Telnetsvr is typically started by the *boot*(A) server.

Options

–T *tcp-cap*
: This causes the server to use the TCP/IP server specified by *tcp-cap*. If this option is omitted it uses the server specified by the string environment variable TCP_SERVER. If that is not set it uses the default TCP server as defined in *ampolicy.h* (typically */profile/cap/ipsvr/tcp*).

–daemon *telnetd-prog*
: This changes the telnet daemon to be started to *telnetd-prog*. The default is */super/admin/bin/telnetd*.

–v Verbose mode. Various debugging information is printed.

Environment Variables

TCP_SERVER specifies the default TPC/IP server to use.

Files

/super/admin/bin/telnetd is the default telnet daemon.

See Also

telnetd(A), ttn(U).

Name

tftp – the Trivial File Transfer Protocol server

Synopsis

```
tftp [-U udp-server] [-chan chan-capability] [-cache] [-v]
```

Description

The *tftp* server implements the Trivial File Transfer Protocol as described in RFC 783. This protocol runs on top of the UDP protocol. The primary function of this server is for booting Sun computers.

The server writes TFTP packets on the network using a UDP channel from a TCP/IP server. It is a good idea to run the *tftp* server on the same host as the TCP/IP server that it uses to obtain optimal performance. Note that if the TCP/IP server does not know its own Internet address the *tftp* server will hang until it does. The Internet address can be set using the command *ifconfig*(A).

Options

–U *udp-server*
: This specifies the capability of the UDP server that is used to put the packets onto the network. This is usually the *udp* channel of the Amoeba TCP/IP server. If this option is not specified it defaults to the UDP server defined by the string environment variable UDP_SERVER. If this is not defined it defaults to the string UDP_SVR_NAME defined in the include file *ampolicy.h*. This is typically */profile/cap/ip/udp*.

–chan *chan-capability*
: This specifies where the capability for the UDP channel used by the *tftp* server should be published. This is useful for examining the status of the channel (see *std_status*(A)).

–cache
: This option will cause the *tftp* server to cache one file in memory, enabling it to answer *tftp* requests without having to go to the file server every time. This will increase performance in case many machines are trying to boot the same kernel at the same time.

–v Verbose mode. This will cause the *tftp* server to give a diagnostic for every new *tftp* request.

Environment Variables

UDP_SERVER
: This is a string environment variable that specifies the capability of the UDP channel to use. It can be overridden using the **–U** command line option.

Administration

To boot Sun computers using the *tftp* server it is necessary to provide a special directory containing files whose names are the Internet addresses (in hex) of the machines to be booted. The contents of these files should be the kernels to be booted. This directory is defined by `TFTPBOOT_DIR` in the file *ampolicy.h* and is typically */super/admin/tftpboot*. This directory should contain a link *tftpboot* which is a capability for itself. The best way to do this is

```
cd /super/admin
mkd tftpboot
get tftpboot | put tftpboot/tftpboot
chm ff:0:0 tftpboot
chm ff:0:0 tftpboot/tftpboot
```

This is required for booting Sun 4c series computers. Normally the various kernels are stored in the directory */super/admin/kernel*. This should only be accessible to holders of the *super* capability. One policy is to name kernels according to architecture and type. For example, *sun3.workst*. These kernels should be installed carefully since some must be stripped and have *a.out* headers removed before they can be booted (see *installk*(A)). Copies of the capabilities for kernels are then stored in the *tftpboot* directory under the Internet address for the host that must boot the kernel. An example of the name for the Sun 3 host with Internet address 130.37.20.80 is 82251450. Note that any hexadecimal letters must be in upper case. If the host is a Sun 4c then the link must be called 82251450.SUN4C. or if the host is a Sun 4m then the link must be called 82251450.SUN4M.

NB. In the case of the i80386 architecture the version of the kernel in */super/admin/kernel* is not suitable for booting via TFTP. The kernel must be copied and then modified using *isamkimage*(A). See the *isamkimage*(A) man page for details.

Example

```
tftp -U /super/hosts/foo/ip/udp -chan /super/cap/tftpsvr/foo
```

See Also

ifconfig(A), installk(A), ipsvr(A), rarp(A), std_status(A).

Name

tod – the Time Of Day server

Synopsis

Currently built into the kernel

Description

The time of day server provides an interface to the hardware time of day clock, if it is available. This server will normally only need to appear in one or two kernels on a local-area network. Besides the standard server commands it only provides two other commands: one to set the time and one to read the time. The server should be used to store GMT (Greenwich Mean Time). Commands such as *date*(U) use the timezone database to calculate the local time correctly. The timezone information is kept in the directory */super/module/time/zoneinfo* and is maintained using *zic*(A).

Programming Interface Summary

The programming interface consists of commands particular to the server (whose names begin with *tod_*) and the standard server commands (whose names begin with *std_*). A summary of the supported commands is presented in the following two tables. For a complete description of the interface routines see *tod*(L) and *std*(L). Not all the standard commands are supported since they are not all pertinent. *Std_copy* and *std_destroy* are not implemented. *std_age* and *std_touch* are implemented but simply return `STD_OK` and do nothing further.

Standard Functions			
Function Name	Required Rights	Error Conditions	Summary
std_age	NONE	RPC error	Does Nothing
std_info	NONE	RPC error	Returns the string ‘‘TOD Server’’
std_restrict	NONE	RPC error STD_CAPBAD	Returns a server capability with restricted rights
std_touch	NONE	RPC error	Does Nothing

TOD Functions			
Function Name	Required Rights	Error Conditions	Summary
tod_gettime	NONE	RPC error	Returns the current time of day plus timezone information
tod_settime	RGT_ALL	RPC error STD_CAPBAD STD_DENIED	Modify the server’s idea of the time

Administration

There are three things which currently affect the server directly.

i) Setting the time which is done using *date*(U). To do this you need a capability with full rights.

ii) Installing the correct timezone database.

iii) Installing the capability of one of the time of day servers as the default time of day server. This is necessary since the entire system needs a place where it can reliably find out a value for the time. The place for installing the default server capability is described in *ampolicy.h* by the variable `DEF_TODSVR` and is typically */profile/cap/todsvr/default*. However the place to install it is via the path */super/cap/todsvr/default* which is the public directory. (The */profile* directory may vary from user to user but typically points to the public version.) It is normal practice to allow the boot server (see *boot*(A)) to maintain the default capability.

See Also

boot(A), date(U), std(L), std_info(U), std_restrict(U). tod(L), zdump(A), zic(A).

Name

vdisk – kernel-based virtual disk server

Synopsis

Built into the kernel.

Description

The virtual disk server provides a high performance interface for reading and writing disk blocks. It does **not** implement a file system. File systems can be constructed on top of this interface. The main feature of the server is that it implements *virtual disks*. It can collect one or more disk partitions from one or more physical disks and present them as a single virtual disk to the client. The client is not aware of the underlying hardware. One of the side-effects of this is that it is possible to grow a virtual disk by adding an extra partition if that does not interfere with the semantics of the disk usage. (Some file systems may have details about the total disk size built into information on the disk.)

NB. All the physical disks managed by the server must be physically attached to the machine where the server is running.

The server can manage more than one virtual disk at a time. Their capabilities will appear in the kernel directory on the machine to which the disks are attached under the name *vdisk:nn* where *nn* is a two digit number assigned to the virtual disk.

In addition there is an interface to each physical disk which can be accessed using the standard disk interface routines. This capability provides raw access to the entire disk and is extremely dangerous since it permits writing on any block on the device. The capability for the physical disks appear in the kernel directory of the machine to which they are attached under the name *bootp:nn* where *nn* is a two digit number assigned to the physical disk.

The server does not enforce a fixed size disk block but rather lets the client specify the size of the disk block that is most convenient for the client. The disk block size must be a power of 2 and may range from 512 bytes up to a system defined limit. (The current limit is 16K bytes.) The client specifies the block size as the logarithm (base 2) of the size of the block.

Information about which partitions belong to which virtual disk is written on the disk. Each physical disk begins with a label describing the disk geometry and the partition table. In addition, each partition assigned to Amoeba begins with a special disk label that describes possible sub-partitioning of that partition and the virtual disk to which it belongs. It also contains information about where in the virtual disk it belongs (in case there is more than one physical partition comprising the vdisk).

Note that the server is able to use the disk label of certain other operating systems and thus share a disk by using a subset of the partitions.

When the virtual disk server starts up it reads the disk label. Using the partition table, it examines each partition to see if it is intended for use by Amoeba. (If it is it will have an Amoeba partition label.) It then collects details about which partitions comprise which virtual disk and then proceeds to take requests. The set of commands accepted by the server are very small. Beyond the standard commands it accepts only five others. They are for read, write, size of the disk, geometry of the disk and an ‘‘info’’ request which returns a description of the underlying physical structure of a virtual disk. The interface is described

below.

Programming Interface Summary

The programming interface consists of commands particular to the server (whose names begin with *disk_*) and the standard server commands (whose names begin with *std_*). A summary of the supported commands is presented in the following two tables. For a complete description of the interface routines see *vdisk*(L) and *std*(L). Not all the standard commands are supported since many of them are only relevant to abstract objects and disks are physical objects. Therefore *std_copy* and *std_destroy* are not implemented. *Std_age* and *std_touch* are implemented but simply return `STD_OK` and do nothing further.

Standard Functions			
Function Name	Required Rights	Error Conditions	Summary
std_age	NONE	RPC error	Does Nothing
std_info	RGT_READ	RPC error STD_ARGBAD STD_CAPBAD STD_DENIED	Returns string giving size in K bytes of disk
std_restrict		RPC error STD_CAPBAD	Returns a disk capability with restricted rights
std_touch	NONE	RPC error	Does Nothing

Disk Functions			
Function Name	Required Rights	Error Conditions	Summary
disk_getgeometry	RGT_READ	RPC error STD_CAPBAD STD_DENIED	Return the disk geometry for a virtual disk
disk_info	RGT_READ	RPC error STD_CAPBAD STD_DENIED	Return physical layout of virtual disk
disk_read	RGT_READ	RPC error STD_ARGBAD STD_CAPBAD STD_DENIED	Reads N disk blocks
disk_size	RGT_READ	RPC error STD_ARGBAD STD_CAPBAD STD_DENIED	Return size in blocks of the disk
disk_write	RGT_WRITE	RPC error STD_ARGBAD STD_CAPBAD STD_DENIED	Writes N blocks to disk

Administration

There is practically no day to day administration required for a virtual disk server once it has been installed. The only functions of interest are adding physical disks and replacing disks. The administrative information used by the server to describe virtual disks is kept in partition tables on the disk. This information is put in place by the *disklabel*(A) program. Therefore all administration of virtual disks is done using this tool.

If you need to know the current relationship between virtual disks and the physical disks (for example for repartitioning a disk) the program *di*(A) will print it for you.

See Also

di(A), disklabel(A), std(L), std_info(U), std_restrict(U), vdisk(L).

Name

zdump – timezone dumper

Synopsis

```
zdump [–v] [–c cutoffyear] [zonename] ...
```

Description

(This utility is part of the *localtime* package distributed though USENET. Its purpose is to inspect the timezone definition files created by *zic*(A).)

Zdump prints the current time in each *zonename* named on the command line.

Options

–v For each *zonename* on the command line, print the current time, the time at the lowest possible time value, the time one day after the lowest possible time value, the times both one second before and exactly at each detected time discontinuity, the time at one day less than the highest possible time value, and the time at the highest possible time value, Each line ends with **isdst=1** if the given time is Daylight Saving Time or **isdst=0** otherwise.

–c *cutoffyear*
Cut off the verbose output near the start of the given year.

Warnings

The **–v** option usually takes a long time to run (minutes) and if there are few continuities little is printed most of the time.

If no *zonename* arguments are specified, nothing is printed.

Examples

To show the current time in GMT, MET and EST:

```
zdump GMT MET EST
```

This produces output like:

```
GMT  Mon Jun  1 11:57:02 1992 GMT
MET  Mon Jun  1 13:57:02 1992 MET DST
EST  Mon Jun  1 11:57:02 1992 EST
```

To show the time discontinuities in MET before 1991:

```
zdump –v –c 1991 MET
```

This might produce:

```
MET  Tue Feb 13 11:13:18 1990 MET
MET  Fri Dec 13 20:45:52 1901 GMT = Fri Dec 13 21:45:52 1901 MET isdst=0
MET  Sat Dec 14 20:45:52 1901 GMT = Sat Dec 14 21:45:52 1901 MET isdst=0
MET  Sun Mar 30 00:59:59 1986 GMT = Sun Mar 30 01:59:59 1986 MET isdst=0
MET  Sun Mar 30 01:00:00 1986 GMT = Sun Mar 30 03:00:00 1986 MET DST isdst=1
MET  Sun Sep 28 00:59:59 1986 GMT = Sun Sep 28 02:59:59 1986 MET DST isdst=1
MET  Sun Sep 28 01:00:00 1986 GMT = Sun Sep 28 02:00:00 1986 MET isdst=0
MET  Sun Mar 29 00:59:59 1987 GMT = Sun Mar 29 01:59:59 1987 MET isdst=0
MET  Sun Mar 29 01:00:00 1987 GMT = Sun Mar 29 03:00:00 1987 MET DST isdst=1
MET  Sun Sep 27 00:59:59 1987 GMT = Sun Sep 27 02:59:59 1987 MET DST isdst=1
MET  Sun Sep 27 01:00:00 1987 GMT = Sun Sep 27 02:00:00 1987 MET isdst=0
MET  Sun Mar 27 00:59:59 1988 GMT = Sun Mar 27 01:59:59 1988 MET isdst=0
MET  Sun Mar 27 01:00:00 1988 GMT = Sun Mar 27 03:00:00 1988 MET DST isdst=1
MET  Sun Sep 25 00:59:59 1988 GMT = Sun Sep 25 02:59:59 1988 MET DST isdst=1
MET  Sun Sep 25 01:00:00 1988 GMT = Sun Sep 25 02:00:00 1988 MET isdst=0
MET  Sun Mar 26 00:59:59 1989 GMT = Sun Mar 26 01:59:59 1989 MET isdst=0
MET  Sun Mar 26 01:00:00 1989 GMT = Sun Mar 26 03:00:00 1989 MET DST isdst=1
MET  Sun Sep 24 00:59:59 1989 GMT = Sun Sep 24 02:59:59 1989 MET DST isdst=1
MET  Sun Sep 24 01:00:00 1989 GMT = Sun Sep 24 02:00:00 1989 MET isdst=0
MET  Sun Mar 25 00:59:59 1990 GMT = Sun Mar 25 01:59:59 1990 MET isdst=0
MET  Sun Mar 25 01:00:00 1990 GMT = Sun Mar 25 03:00:00 1990 MET DST isdst=1
MET  Sun Sep 30 00:59:59 1990 GMT = Sun Sep 30 02:59:59 1990 MET DST isdst=1
MET  Sun Sep 30 01:00:00 1990 GMT = Sun Sep 30 02:00:00 1990 MET isdst=0
MET  Mon Jan 18 03:14:07 2038 GMT = Mon Jan 18 04:14:07 2038 MET isdst=0
MET  Tue Jan 19 03:14:07 2038 GMT = Tue Jan 19 04:14:07 2038 MET isdst=0
```

See Also

date(U), ctime(L), zic(A).

Name

zic – timezone compiler

Synopsis

```
zic [-v] [-d directory] [-l localtime] [-p posixrules]
    [-L leapsecondfilename] [-s] [filename] ...
```

Description

(This utility is part of the *localtime* package distributed though USENET. Its purpose is to create the timezone definition files used by ctime(L) and friends. Most of its complexity stems from the diversity in daylight saving time rules employed by different countries.)

Zic reads text from the file(s) named on the command line and creates the time conversion information files specified in this input. If a *filename* is –, the standard input is read.

Options

–d *directory*
Create time conversion information files in the named *directory* rather than in the standard directory named below.

–l *timezone*
Use the given *timezone* as local time. *Zic* will act as if the input contained a link line of the form

Link *timezone* localtime

–p *timezone*
Use the given *timezone*'s rules when handling POSIX-format timezone environment variables. *Zic* will act as if the input contained a link line of the form

Link *timezone* posixrules

–L *leapsecondfilename*
Read leap second information from the file with the given name. If this option is not used, no leap second information appears in output files.

–s Limit time values stored in output files to values that are the same whether they are taken to be signed or unsigned. You can use this option to generate SVVS-compatible files.

–v Complain if a year that appears in a data file is outside the range of years representable by *time*(L) values.

Input lines are made up of fields. Fields are separated from one another by any number of white space characters. Leading and trailing white space on input lines is ignored. An unquoted sharp character (#) in the input introduces a comment which extends to the end of the line the sharp character appears on. White space characters and sharp characters may be enclosed in double quotes (") if they are to be used as part of a field. Any line that is blank (after comment stripping) is ignored. Non-blank lines are expected to be of one of three

types: rule lines, zone lines, and link lines.

A rule line has the form

```
Rule  NAME  FROM  TO  TYPE  IN  ON  AT  SAVE  LETTER/S
```

For example:

```
Rule  USA  1969  1973  –  Apr  lastSun  2:00  1:00  D
```

The fields that make up a rule line are:

NAME Gives the (arbitrary) name of the set of rules this rule is part of.

FROM Gives the first year in which the rule applies. The word **minimum** (or an abbreviation) means the minimum year with a representable time value. The word **maximum** (or an abbreviation) means the maximum year with a representable time value.

TO Gives the final year in which the rule applies. In addition to **minimum** and **maximum** (as above), the word **only** (or an abbreviation) may be used to repeat the value of the **FROM** field.

TYPE Gives the type of year in which the rule applies. If **TYPE** is **–** then the rule applies in all years between **FROM** and **TO** inclusive; if **TYPE** is **uspres**, the rule applies in U.S. Presidential election years; if **TYPE** is **nonpres**, the rule applies in years other than U.S. Presidential election years. If **TYPE** is something else, then *zic* executes the command

yearistype *year type*

to check the type of a year: an exit status of zero is taken to mean that the year is of the given type; an exit status of one is taken to mean that the year is not of the given type.

IN Names the month in which the rule takes effect. Month names may be abbreviated.

ON Gives the day on which the rule takes effect. Recognized forms include:

5	the fifth of the month
lastSun	the last Sunday in the month
lastMon	the last Monday in the month
Sun>=8	first Sunday on or after the eighth
Sun<=25	last Sunday on or before the 25th

Names of days of the week may be abbreviated or spelled out in full. Note that there must be no spaces within the **ON** field.

AT Gives the time of day at which the rule takes effect. Recognized forms include:

2	time in hours
2:00	time in hours and minutes
15:00	24-hour format time (for times after noon)
1:28:14	time in hours, minutes, and seconds

Any of these forms may be followed by the letter **w** if the given time is local ‘‘wall clock’’ time or **s** if the given time is local ‘‘standard’’ time; in the absence of **w** or **s**, wall clock time is assumed.

SAVE Gives the amount of time to be added to local standard time when the rule is in effect. This field has the same format as the **AT** field (although, of course, the **w** and **s** suffixes are not used).

LETTER/S Gives the ‘‘variable part’’ (for example, the ‘‘S’’ or ‘‘D’’ in ‘‘EST’’ or ‘‘EDT’’) of timezone abbreviations to be used when this rule is in effect. If this field is **–**, the variable part is null.

A zone line has the form

```
Zone  NAME                GMTOFF  RULES/SAVE  FORMAT  [UNTIL]
```

For example:

```
Zone  Australia/South−west  9:30  Aus         CST     1987 Mar 15 2:00
```

The fields that make up a zone line are:

NAME The name of the timezone. This is the name used in creating the time conversion information file for the zone.

GMTOFF The amount of time to add to GMT to get standard time in this zone. This field has the same format as the **AT** and **SAVE** fields of rule lines; begin the field with a minus sign if time must be subtracted from GMT.

RULES/SAVE The name of the rule(s) that apply in the timezone or, alternately, an amount of time to add to local standard time. If this field is **–** then standard time always applies in the timezone.

FORMAT The format for timezone abbreviations in this timezone. The pair of characters **%s** is used to show where the ‘‘variable part’’ of the timezone abbreviation goes.

UNTIL The time at which the GMT offset or the rule(s) change for a location. It is specified as a year, a month, a day, and a time of day. If this is specified, the timezone information is generated from the given GMT offset and rule change until the time specified.

The next line must be a ‘‘continuation’’ line; this has the same form as a zone line except that the string ‘‘Zone’’ and the name are omitted, as the continuation line will place information starting at the time specified as the **UNTIL** field in the previous line in the file used by the previous line. Continuation lines may contain an **UNTIL** field, just as zone lines do, indicating that the next line is a further continuation.

A link line has the form

```
Link  LINK-FROM  LINK-TO
```

For example:

```
Link    US/Eastern      EST5EDT
```

The **LINK-FROM** field should appear as the **NAME** field in some zone line; the **LINK-TO** field is used as an alternate name for that zone.

Except for continuation lines, lines may appear in any order in the input.

Lines in the file that describes leap seconds have the following form:

```
Leap    YEAR    MONTH    DAY    HH:MM:SS    CORR    R/S
```

For example:

```
Leap    1974    Dec      31     23:59:60    +       S
```

The **YEAR**, **MONTH**, **DAY**, and **HH:MM:SS** fields tell when the leap second happened. The **CORR** field should be ‘‘+’’ if a second was added or ‘‘-’’ if a second was skipped. The **R/S** field should be (an abbreviation of) ‘‘Stationary’’ if the leap second time given by the other fields should be interpreted as GMT or (an abbreviation of) ‘‘Rolling’’ if the leap second time given by the other fields should be interpreted as local wall clock time.

Warning

For areas with more than two types of local time, you may need to use local standard time in the **AT** field of the earliest transition time’s rule to ensure that the earliest transition time recorded in the compiled file is correct.

File

/profile/module/time/zoneinfo standard directory used for created files

File Format

The timezone information files created begin with bytes reserved for future use, followed by four four-byte values of type **long**, written in a ‘‘standard’’ byte order (the high-order byte of the value is written first). These values are, in order: *tzh_ttisstdcnt* The number of standard/wall indicators stored in the file. *tzh_leapcnt* The number of leap seconds for which data is stored in the file. *tzh_timecnt* The number of ‘‘transition times’’ for which data is stored in the file. *tzh_typecnt* The number of ‘‘local time types’’ for which data is stored in the file (must not be zero). *tzh_charcnt* The number of characters of ‘‘timezone abbreviation strings’’ stored in the file.

The above header is followed by *tzh_timecnt* four-byte values of type **long**, sorted in ascending order. These values are written in ‘‘standard’’ byte order. Each is used as a transition time (as returned by *time*(L)) at which the rules for computing local time change. Next come *tzh_timecnt* one-byte values of type **unsigned char**; each one tells which of the different types of ‘‘local time’’ types described in the file is associated with the same-indexed transition time. These values serve as indices into an array of *ttinfo* structures that appears next in the file; these structures are defined as follows:

```
struct ttinfo {
      long          tt_gmtoff;
      int           tt_isdst;
      unsigned int  tt_abbrind;
};
```

Each structure is written as a four-byte value for *tt_gmtoff* of type **long**, in a standard byte order, followed by a one-byte value for *tt_isdst* and a one-byte value for *tt_abbrind*. In each structure, *tt_gmtoff* gives the number of seconds to be added to GMT, *tt_isdst* tells whether *tm_isdst* should be set by *localtime*(L) and *tt_abbrind* serves as an index into the array of timezone abbreviation characters that follow the *ttinfo* structure(s) in the file.

Then there are *tzh_leapcnt* pairs of four-byte values, written in standard byte order; the first value of each pair gives the time (as returned by *time(L))* at which a leap second occurs; the second gives the *total* number of leap seconds to be applied after the given time. The pairs of values are sorted in ascending order by time.

Finally there are *tzh_ttisstdcnt* standard/wall indicators, each stored as a one-byte value; they tell whether the transition times associated with local time types were specified as standard time or wall clock time, and are used when a timezone file is used in handling POSIX-style timezone environment variables.

Localtime uses the first standard-time *ttinfo* structure in the file (or simply the first *ttinfo* structure in the absence of a standard-time structure) if either *tzh_timecnt* is zero or the time argument is less than the first transition time recorded in the file.

Example

To install the timezone information for Europe and North America, copy the files ‘europe’ and ‘northamerica’ from the zic source directory to an Amoeba directory with *tob*(U), and on Amoeba, in that directory, execute the commands

```
mkd /super/module/time
mkd /super/module/time/zoneinfo
zic -d /super/module/time/zoneinfo europe northamerica
```

The *mkd*(U) commands create the necessary directories (assuming these haven’t been created yet); the **–d** *path* option to *zic* is used to ensure permission to create the timezone files.

Subsequently, to use Middle European Time both as the default local time zone and as the default rule base for POSIX-style timezone environment variables, execute the command

```
zic -d /super/module/time/zoneinfo -l MET -p MET
```

See Also

ctime(L), date(U), del(U), tob(U), zdump(A).

Name

Tsuite – a suite of test programs for Amoeba

Description

The test suite for Amoeba consists of a set of test programs and shell scripts that verify stubs, functions and utilities. Also included is a set of performance measuring programs. The name of a test program or script usually starts with a capital *'T'*.

Test programs and scripts generally test valid and invalid test cases. Valid cases use valid parameters, operations or input data. Invalid cases verify that stubs, functions and utilities return correct error codes for invalid parameters, operations or input data.

Most programs in the test suite have options and arguments to change their default behavior.

Some test programs verify an entire server interface. It is suggested that these tests are run on separate servers to avoid interference with the normal daily operation of Amoeba.

Overview of the test programs available

Tsoap A comprehensive test program for the directory service. See *Tsoap*(T).

Tbullet

Tbullet1

Tbulletc A set of test programs for the Bullet file server. See *Tbullet*(T).

Tmulti_bullet

This is an interactive program checking the integrity of the Bullet Server when doing a specified set of file operations. All file operations are performed in parallel by a specified number of threads, unless sequential mode of operation is requested.
Usage:

```
Tmulti_bullet bulletsvr [alternate-bulletsvr]
```

The alternate bullet server is optional. It is only needed when testing the *std_copy* operation.

Tgetdef A simple program checking the *pro_getdef* request.
Usage:

```
Tgetdef procsvr
```

The argument *procsvr* is the process server of a certain host, e.g., */super/hosts/dodo/proc*.

Tgetload A simple program checking the *pro_getload* request.
Usage:

```
Tgetload procsvr
```

Tgetowner A simple program checking the *pro_getowner* request.
Usage:

```
Tgetowner process-cap
```

For example, the following sequence of commands can be used to inspect the owner of the session server.

```
$ aps -v
PID PPID PGRP STATE   T OWNER          COMMAND
  1    1    1 RUNNING R ax server      session -a
  2    1    2 WAITING R session server -ksh
 53    2    2 RUNNING R session server aps -v
$ Tgetowner /dev/proc/1
Owner of '/dev/proc/1' is E:D:F:5:F:9/0(0)/0:0:0:0:0:0
$ a2c 'E:D:F:5:F:9/0(0)/0:0:0:0:0:0' ownercap
$ std_info ownercap
ax server for versto
```

Tsetowner A simple program checking the *pro_setowner* request.
Usage:

```
Tsetowner process-cap new-owner-cap
```

For example, the following sequence of commands changes the owner of the session server to itself.

```
$ Tsetowner /dev/proc/1 /dev/session
Owner of '/dev/proc/1' set to '/dev/session'.
$ aps -v
PID PPID PGRP STATE   T OWNER          COMMAND
  1    1    1 RUNNING R session server session -a
  2    1    2 WAITING R session server -ksh
 58    2    2 RUNNING R session server aps -v
```

Tdeepen This program calls a few recursive functions. It is mainly used to check the correct handling of the window overflow and underflow interrupts generated by the *sparc* processor.

Tstackovf1

Tstackovf2 These programs check that stack overflow will result in a program crash.

Trnd1

Trnd2 These programs check the correct working of the random number server.
Usage:

```
Trnd1 [randomsvr]
Trnd2 [randomsvr]
```

If the random numver server is not specified, the default one (typically */profile/cap/randomsvr/default*) is used.

Tprio1

Tprio2

Tprio3 These programs check the correct working of preemptive priority scheduling.

Usage:

```
Tprio1 nthreads
Tprio2 [nthreads]
Tprio3
```

Tfloatp

Tctest These programs perform a number of computations, including floating point operations. The programs can be used to check that the floating point status is saved correctly by running multiple instances on the same host in parallel.

Tpipe This program checks the integrity of the *pipe* implementation of Amoeba. See *Tpipe*(T).

Tsignal This program checks the correct working of the POSIX signal emulation under Amoeba. See *Tsignal*(T).

Tam_signals This program checks the correct working of (light-weight) Amoeba signals. See *Tam_signals*(T).

Tmsg_client

Tmsg_server These two programs test the fault-tolerant message-based RPC library (see *msg_rpc*(L) and *msg*(L)). Before starting the tests, make sure that the necessary directories exist as specified by `RRPC_GRP_DIR` and `RRPC_CTX_DIR` in *ampolicy.h*. Then start one or more *Tmsg_server* programs in the background and execute *Tmsg_client*. The client will send the integers `1` to `500` to the replicated service, and wait for a reply. The protocol implemented should guarantee correct behavior even when all but one of the *Tmsg_server* programs crashes.
Usage:

```
Tmsg_client [-v] [-s]
Tmsg_server [-v] [-s]
```

The **–v** option switches the program to verbose mode. The **–s** option increases the waiting time for each RPC with one second (default is immediate response).

Tmsg_grp1

Tmsg_grp2 These two programs test the message-based group management library (see *msg_grp*(L) and *msg*(L)). Before starting the tests, make sure the necessary directories exist as specified by `RRPC_GRP_DIR` and `RRPC_CTX_DIR` in *ampolicy.h*. Then start multiple instances of either program in the background. The program *Tmsg_grp1* will send the integers `0` to `500` to all replicas and exit when it receives its last message back. Program *Tmsg_grp2* sends over additional information to check the correctness of the various marshaling routines.

Tstack This is a simple test of the threads implementation to show that each thread is running on its own stack.

Tfromb.sh This shell script creates ten files with various contents on the Bullet server, waits five minutes, and then reads them back to check that their contents are

still correct.

worm This is a self-replicating program that can be used to test the process and segment server running on each host in the pool.
Usage:

```
worm [-d] [-k]
```

The **–d** option increases the amount of diagnostics produced. If *worm* is called with the **–k** option, all worms currently running on the pool will be told to terminate.

Overview of the performance test programs available

dhrystone

dhrystone2 These programs will give a quick impression of the cpu and C-compiler performance for a particular system. *Dhrystone* performs a predefined number of integer, structure and string operations, and gives a performance rating according to the time used. The program *dhrystone2* is a newer version in which the number of iterations is specified interactively. It is also more verbose.

Tperf_milli

Tperf_null

Tperf_thrdsw

Tperf_thrdsw.preempt These programs test the performance of the system calls *sys_milli*, *sys_null* and *threadswitch* respectively.

Tperf_disk This program tests the performance of the *vdisk* server.
Usage:

```
Tperf_disk [-w] [-d duration] [-b bytes] diskcap
```

By default, *Tperf_disk* only tests disk reads for various block sizes. The **–w** option causes (destructive!) disk writes to be done as well. The **–d** option restricts the test per block size to the specified number of seconds (default is 30 seconds). The **–b** option specifies the number of bytes to be read or written per iteration (default is 64K).

Tperf_tats This program measures the performance of *mutex* operations, *thread* creation, local and remote RPCs for various sizes, and Bullet Server operations.
Usage:

```
Tperf_tats [-i] [-m machine] [multiplier]
```

The **–i** option causes the program to wait for a newline on standard input before starting the tests. The **–m** option specifies the machine on which the child process has to run during the remote RPC tests (otherwise an arbitrary one from the pool is selected). The optional `multiplier` multiplies the number of tests performed by the amount specified.

Tperf_grp This program tests the performance of the group communication in Amoeba. The program must be started by `gax` (see *gax*(U)) as follows:

```
gax -p pool Tperf_grp ncpu nsender size \
        nmess large resil check recover debug
```

The parameters have the following meaning:

`ncpu` The number of group members.

`nsender` The number of members sending data to the group; this should be in the range 1..`ncpu`.

`size` The size of the messages sent. The current group implementation limits this amount to about 8000 bytes.

`nmess` The number of messages sent by each sender.

`large` Determines which variant of the group protocol will be used; see *group*(L).

`resil` The resilience degree of the group communication; this should be in the range 0..`ncpu`.

`check` If non-zero, the data received will be checked for correctness.

`recover` If non-zero, *Tperf_grp* will try to recover from group communication errors.

`debug` A flag that will be passed to the kernel to set the group communication debugging level.

Tpclient

Tpserver

Tpcomp These programs can be used to measure the raw RPC performance of the system.
Usage:

```
Tpserver port
Tpclient [-r] port size count
Tpcomp port1 port2 size count
```

The program *Tpserver* must be started first, with an ASCII port argument that will be used during the test (e.g., `myport`).
The program *Tpclient* will perform `count` RPCs of size `size` with *Tpserver*, and reports the latency and throughput measured. The **−r** flag causes the data to be sent in the reverse direction. If *Tpclient* is started with `count` equal to zero, *Tpserver* will be told to exit.
The program *Tpcomp* acts as a client of the two *Tpservers* specified by the port arguments. It will check the integrity of the random data sent over to both servers, and reports the measured throughput like *Tpclient*.

Trpc_client

Trpc_server

Trpc_comp Like *Tpclient*, *Tpserver* and *Tpcomp*, these programs measure the raw RPC performance. The difference is that these programs use a new RPC interface that allows RPCs greater than 30000 bytes.
Usage:

```
Trpc_server port maxsize rpctime
Trpc_client [-m measure] [-r] port size count
Trpc_comp port1 port2 size count
```

The program *Trpc_server* must be started first, with an ASCII port argument that will be used during the test (e.g., `myport`). Parameter `maxsize` specifies the maximum RPC size that may be performed with this server. Parameter `rpctime` specifies the number of milliseconds the server will wait before sending the reply of an RPC.
The program *Trpc_client* will perform `count` RPCs of size `size` with *Trpc_server*, and reports the latency and throughput measured. The **–m** option will cause a progress report to be written each time `measure` RPCs have been completed. The **–r** flag will cause the data to be sent in the reverse direction. If *Trpc_pclient* is started with `count` equal to zero, *Trpc_pserver* will be told to exit.
The program *Trpc_comp* acts as a client of the two *Trpc_servers* specified by the port arguments. It will check the integrity of the random data sent over to both servers, and reports the measured throughput like *Trpc_client*.

Trrpc_client

Trrpc_server

Trrpc_start These programs test the performance of the replicated RPC package.
Usage:

```
Trrpc_server
Trrpc_client size count
Trrpc_start
```

First, one or more replicas of *Trrpc_server* must be started on background. Next, *Trrpc_client* should be started on background. The parameters specify the size and number of RPCs respectively. Finally, *Trrpc_start* should be executed. It will tell *Trrpc_client* to start the performance test as soon as the *Trrpc_servers* are completely initialized.

See Also

Tam_signals(T), Tbullet(T), Tpipe(T), Tsignal(T), Tsoap(T).

Name

Tbullet – test a Bullet file server

Synopsis

```
Tbullet  [-vgbq] [servername]
Tbullet1 [servername]
Tbulletc [-vgbq] [-n nprocs] [servername]
```

Description

Tbullet and *Tbullet1* test a Bullet file server by calling its stubs with various valid and invalid parameters and checking the results. *Tbulletc* checks the functioning of the Bullet Server when multiple clients are doing simultaneous transactions. The number of clients can be specified as a parameter.

Normally the default Bullet Server is tested with all the test cases and only cases that have failed are reported. Another Bullet Server can be tested by specifying its server capability as the *servername* command-line argument.

Tests with invalid parameters include cases with capabilities for non-existent servers. These cases can take a while because of RPC locate time-outs.

Options

–v	verbose: report the test's progress including successful test cases.
–g	good cases only: execute cases with valid parameters only.
–b	bad cases only: execute cases with invalid parameters only.
–q	quick: do not execute cases with potentially long time-outs.
–n *nprocs*	for Tbulletc, this specifies the number of processes doing file operations in parallel; the default is one process.

Diagnostics

Failed test cases are reported by printing the filename and line number of the source file where the message was generated, the main test routine where the failure was discovered, the function or stub that caused the error, an extra code for easier reference in the source file and a phrase that describes the failure. A header mentions the name of the test and a trailer tells how many failures were reported altogether.

In verbose mode the name of the current test routine and functions and stubs that were executed successfully are also reported. On exit the program returns the total number of failures.

Warnings

Test cases that fail may influence the results of subsequent cases because it may not be possible to recover completely from the failure.

Examples

Test the default Bullet Server with all the test cases except the ones with long time-outs and report both failure and success:

```
Tbullet -vq
```

Test the Bullet Server */profile/cap/bulletsvr/bullet2* with two client processes and with test cases for valid parameters only:

```
Tbulletc -n 2 -g /profile/cap/bulletsvr/bullet2
```

See Also

bullet(A), bullet(L), Tsuite(T).

Name

Tpipe – test POSIX-style pipes

Synopsis

```
Tpipe [-vgb] [-s bsize] [-n bcount]
```

Description

Tpipe tests POSIX-style pipes on Amoeba. A pipe is created between two processes and then blocks of random data are written to one side of the pipe and read from the other. The test also verifies the error codes that should be returned when invalid operations are performed on a pipe, such as writing onto a pipe whose reading side is closed. Normally, only test cases that failed are reported.

Options

–v	verbose: report the test's progress including successful test cases.
–g	good cases only: perform valid operations only.
–b	bad cases only: perform invalid operations only.
–s bsize	transfer blocks of size *bsize* instead of the default size (4K).
–n bcount	transfer *bcount* blocks instead of the default number of blocks (8).

Diagnostics

Diagnostic messages from the process that reads from the pipe are displayed on *stdout* unless the program runs in the background. Messages from a background process and the process writing onto a pipe are redirected to the file *proX.log*, where *X* is the process id of the corresponding process.

Failed test cases are reported by printing the file name and line number of the source file where the message was generated, the main test routine where the failure was discovered, the function that caused the failure, an extra code for easier reference in the source file and a phrase that describes the failure. A header mentions the name of the test and a trailer tells how many failures were reported altogether.

In verbose mode the name of the current test routine and functions that were executed successfully are also reported. On exit the program returns the total number of failures.

Warnings

Test cases that fail may influence the results of subsequent cases because it may not be possible to recover completely from the failure.

Example

Test a pipe with 20 blocks of 100000 bytes and invalid operations and display a trace of the test's progress:

```
Tpipe -v -s 100000 -n 20
```

See Also

posix(L), Tsuite(T).

Name

Tsignal – test POSIX-style signal catching

Synopsis

```
Tsignal [-vgb]
```

Description

Tsignal tests POSIX-style signal catching with the *signal* library function. After the main parent process has installed a handler for a signal with *signal*, a child process is told to generate a signal for the parent. The parent goes to sleep while waiting for the signal to arrive.

The test also verifies the error codes that should be returned when handlers are installed for non-existent signals or signals that cannot be caught such as `SIGKILL`. Normally, only failed test cases are reported.

Options

–v verbose: report the test's progress including successful test cases.

–g good cases only: perform valid operations only.

–b bad cases only: perform invalid operations only.

Diagnostics

Failed test cases are reported by printing the file name and line number of the source file where the message was generated, the main test routine where the failure was discovered, the function that caused the failure, an extra code for easier reference in the source file and a phrase that describes the failure. A header mentions the name of the test and a trailer tells how many failures were reported altogether.

In verbose mode the name of the current test routine and functions that were executed successfully are also reported. On exit the program returns the total number of failures.

Warnings

Test cases that fail may influence the results of subsequent cases because it may not be possible to recover completely from the failure. The program really only tests the *signal* library function. The *kill* library function is not extensively tested.

Do not confuse this test program with the program for testing lightweight signals between threads in Amoeba processes *Tam_signals*.

Example

Test the *signal* function and display a trace of the test's progress:

```
Tsignal -v
```

See Also

posix(L), Tsuite(T).

Name

Tam_signals – test lightweight signals between threads in Amoeba

Synopsis

```
Tam_signals [-vgb]
```

Description

Tam_signals tests lightweight signals between threads in Amoeba processes. Two threads exchange signals and verify the results.

The test also verifies the error code that should be returned when a handler is installed for an illegal signal 0. Normally, only failed test cases are reported.

Options

- **–v** verbose: report the test's progress including successful test cases.
- **–g** good cases only: use valid parameters only.
- **–b** bad case only: use invalid parameter (signal 0) only.

Diagnostics

Failed test cases are reported by printing the file name and line number of the source file where the message was generated, the main test routine where the failure was discovered, the function that caused the failure, an extra code for easier reference in the source file and a phrase that describes the failure. A header mentions the name of the test and a trailer tells how many failures were reported altogether.

In verbose mode the name of the current test routine and functions that were executed successfully are also reported. On exit the program returns the total number of failures.

Warnings

Test cases that fail may influence the results of subsequent cases because it may not be possible to recover completely from the failure. Exceptions are not fully tested because recovery from exceptions is generally not possible.

Do not confuse this test program with the program for testing heavyweight POSIX-style signals *Tsignal*.

Example

Test lightweight signals and display a trace of the test's progress:

```
Tam_signals -v
```

See Also

signals(L), thread(L), Tsuite(T).

Name

Tsoap – test a Soap directory server

Synopsis

```
Tsoap [-vgbq] [pathname]
```

Description

Tsoap tests a Soap directory server by calling its stubs with various valid and invalid parameters and checking the results. Normally the Soap Server of the current working directory is tested with all the test cases. Normally, only failed test cases are reported.

The test creates, modifies and deletes entries with funny names in the current directory. If the current directory is not a suitable place to experiment, the program should be called from another more suitable directory (preferably empty). The test can also be told to use an alternative directory by specifying the directory's *pathname* as command-line argument. An alternative Soap Server can be tested by running the program in a directory on that server.

Tests with invalid parameters include cases with capabilities for non-existent servers. These cases can take a while because they wait for RPC locate time-outs.

Options

- **–v** verbose: report the test's progress including successful test cases.
- **–g** good cases only: execute cases with valid parameters only.
- **–b** bad cases only: execute cases with invalid parameters only.
- **–q** quick: do not execute cases that wait for long time-outs.

Diagnostics

Failed test cases are reported by printing the filename and line number of the source file where the message was generated, the main test routine where the failure was discovered, the function or stub that caused the error, an extra code for easier reference in the source file and a phrase that describes the failure. A header mentions the name of the test and a trailer tells how many failures were reported altogether.

In verbose mode the name of the current test routine and functions and stubs that were executed successfully are also reported. On exit the program returns the total number of failures.

Warnings

Test cases that fail may influence the results of subsequent cases because it may not be possible to recover completely from the failure.

Examples

Test the Soap server of the current working directory with all the test cases except the ones with long time-outs and report both failure and success:

```
Tsoap -vq
```

If another Soap server is ‘‘mounted’’ on */super/testsoap,* it can be tested from a directory *testdir* as follows:

```
Tsoap /super/testsoap/testdir
```

See Also

soap(A), soap(L), Tsuite(T).

Name

worm – self-replicating program

Synopsis

```
worm [-d|-k]
```

Description

This is a self-replicating program. Once started, it attempts to run copies of itself on all available pool processors (of the same architecture). Each copy regularly polls all other copies, and if a copy has disappeared, a new copy is started. This keeps processors and the network busy. The output of each *worm* is normally redirected to the console of its host.

It is difficult to get rid of a collection of worms by rebooting their machines, since as soon as a machine is rebooted a worm on another machine will start a new worm on it. The **–k** option can be used to kill the entire collection.

Options

–d Debugging mode – output is not redirected.

–k Kill mode – kill all other worms that can be found and then exit.

Examples

To start the worms (in the background):

```
worm &
```

To kill all worms:

```
worm –k
```

9 Index

The italic references in the index are to programs or routines described in this manual.

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z