Smx—the Solaris port of MINIX

Paul Ashton (paul@cosc.canterbury.ac.nz)

September 14th, 1996

1 Introduction

Solaris MINIX (smx) is a version of the MINIX operating system that runs as a user process under Solaris 2.x (SunOS 5.x) on SPARC-based Suns. The set of user commands, library functions and system calls is virtually identical to that of standard (PC) MINIX (which we will refer to simply as "MINIX" in this document). Because smx runs as a user process, multiple copies can run simultaneously on one Sun, quite independent of each other and of any other workload present at the time. This flexibility means that smx can be used for laboratory work in OS courses in situations where it would be infeasible to have students modify a native operating system.

This document concentrates on the differences between MINIX and smx (those wanting information on MINIX should see [3]). We begin with an overview, and follow that with installation instructions. We then go on to describe smx from two viewpoints—the user viewpoint and the internal viewpoint. We finish the main body of the report by describing the interaction network monitor that has been added to smx. Three appendices contain: manual pages for the small number programs that run under SunOS, advice for using smx with a class, and an overview of the source code changes made in smx.

Those wanting more information about smx and about interaction network monitoring (including information on the SunOS and Amoeba monitors) should follow the appropriate links from:

`http://www.cosc.canterbury.ac.nz/~paul`

2 Overview

To the user, smx seems much the same as MINIX. Most of the MINIX commands are available. The same library and system calls are available, although compilation is done under SunOS rather than under smx. The reason that smx seems much the same is that it **is** much the same. Nearly all source code in the memory manager, the file system, the internet server and the user commands and the libraries is identical to the MINIX code. Most smx-specific code is concentrated in two areas—the kernel and the bootstrap program, and even then the smx kernel contains a substantial amount of code that is unchanged from the MINIX kernel.

The "hardware" that smx runs on is the virtual machine defined by a SunOS process. The CPU time distributed by the smx scheduler consists of the CPU time given by the SunOS scheduler to the SunOS process running smx. The memory managed by the smx memory manager is part of the address space of the SunOS process running smx. The interrupts are

SunOS signals, with the alarm signal used to simulate the hardware clock. Each smx file system is stored as a SunOS file, with the file formatted as a fixed length MINIX file system. The smx console is the controlling terminal of the SunOS process running smx. One way to think of smx is as a complex threads package for SunOS.

The `minix` program (which runs under SunOS) is used to bootstrap smx, by setting up the smx devices, loading the operating system, and then jumping into smx proper to begin the standard MINIX booting sequence. The standard booting messages appear and the user can then login when prompted for a usercode. Additional terminal sessions can be attached to a running instance of smx using the (SunOS) `mlogin` program. This means that smx is a multi-user system, and that it is easy to experiment with having multiple users.

The only area in which smx suffers from a lack of realism is in its low-level device drivers. Disk accesses are simulated by first calling the SunOS `lseek` system call to move to the appropriate file position, and then making a `read` or `write` system call to transfer the data. The system calls are synchronous, so there is no chance to run another smx process while the transfer is taking place. Similarly terminal input and output is done via `read` and `write` system calls. In the case of input, a SunOS signal is generated whenever input becomes available.

This lack of reality is confined to a very small part of the system. The kernel must still face up to asynchronous interrupts (from the clock, keyboard and ethernet), and the potential for race conditions that results. The kernel code for CPU scheduling and message passing, and all code for the memory manager, file system and internet server is virtually identical to that of MINIX. Consequently, smx provides good support for a wide range of OS laboratories.

The standard set of MINIX manual entries are available from within smx. New entries have been added for smx-specific commands, but otherwise the manual pages have not been changed. Manual pages describing commands specific to the PC should be ignored. Manual entries for the smx programs and scripts that run under SunOS can be found in Appendix A in this document.

2.1 History and acknowledgements

Solaris MINIX has been chiefly developed in the Department of Computer Science at the University of Canterbury, Christchurch, New Zealand. In the (southern hemisphere) summer of 1991/92, Peter Smith (psmith@cs.ubc.ca, then a student at Canterbury) created the first version of SunOS MINIX by porting Macintosh MINIX 1.5.10 to SunOS 4 running on SPARC-based Sun systems. SunOS MINIX was developed further by Paul Ashton and released (as a set of diffs to MINIX 1.5.10) in 1992. Bill Bynum (bynum@cs.wm.edu) contributed a Sun 3 version. Over the (southern hemisphere) summer of 1992/93, Peter Smith developed an initial port of SunOS MINIX to Solaris 2. At that point, the SunOS MINIX distribution contained SPARC/SunOS 4, SPARC/Solaris 2 and Sun 3 versions.

Late in 1995, Paul Ashton took the initial Solaris port of SunOS MINIX, and the (now publicly available) MINIX 1.7.1 release, and from them produced Solaris MINIX. Considerable further development has taken place since then, including integration of the changes in MINIX 1.7.2 to 2.0.

Finally, thanks to Kees Bot (kjb@cs.vu.nl) for considerable amounts of information on MINIX 1.7.x and 2.x, and for his timely updates.

3 Installing smx

Smx is distributed (over the internet and on CD-ROM) as a compressed tar file. This section tells you how to reach the point at which you can run smx. Note that this version of smx has been developed under Solaris 2.5, and has also been tested under Solaris 2.4. Smx may run on earlier versions of Solaris 2, but will not run on Solaris 1.

The steps involved in setting up smx are described in the following sub-sections.

3.1 Installing the SunOS programs

There are a number of SunOS binaries and scripts that you need in your search path before going any further.

1. Check your search path for `install` and `gcc` using the commands:

   ```
   which gcc
   which install
   ```

   If `which` fails to locate `gcc` then you will need to install `gcc` before proceeding. The MINIX CD-ROM contains a `gcc` distribution in a single compressed tar file. `gcc` is available for anonymous FTP from `prep.ai.mit.edu` in the `/pub/gnu` directory.

   Smx installation relies on the SVR4 version of `install`, so if `which` reports that `install` is picked up from `/usr/ucb` you must change your path so that the SVR4 version of `install` (in `/usr/sbin`) is the one used.

2. Create a directory for smx. Change to that directory.

3. Type

   ```
   m=`pwd`
   ```

   At this point the variable `$m` refers to the directory you are now in. If you are using `csh` then you will have to type the following instead:

   ```
   setenv m `pwd`
   ```

4. Untar the smx tar file using the command line:

   ```
   tar xvf TARFILENAME
   ```

   where `TARFILENAME` is the name of the smx tar file.

5. Edit `Makefile` in `$m/src/Solaris` so that `DESTDIR` specifies the directory that you want to install the SunOS binaries and scripts in. Create `DESTDIR` if it doesn't already exist.

6. Inside `$m/src/Solaris`, type:

```
make install
```

7. Ensure that the smx binary directory (`DESTDIR` from step 5) is on your search path.

   For each of the following programs, use `which` to check that the the program picked up from your search path is the program in the smx binary directory: `combine`, `elf2smx`, `make_map_file`, `mcc`, `minix`, `mlogin`, `next_prog_addr`, `relay`.

3.2 Completing the installation process

1. Set the `MX_INCL` and `MX_LIB` environment variables.

   The `mcc` script and some `Makefile`s rely on the correct setting of the `MX_INCL` and `MX_LIB` environment variables. `MX_INCL` should be set to the full pathname of the `$m/include` directory and `MX_LIB` to the full pathname of the directory into which the library files will be installed. This might be the `$m/src/lib` directory itself, so long as you are happy having any library changes installed immediately.

   If you would rather have a separate ("production") library directory, then set `MX_LIB` to a different pathname (`$m/lib` perhaps). If you take this option, it will be up to you to copy new libraries from `$m/src/lib` to `MX_LIB`. This can be done by a `make install` in `$m/src/lib`, or by entering the appropriate copy commands from the shell.

2. Next, the libraries must be created. Change directory to `$m/src/lib`. If `$MX_LIB` is `$m/src/lib` then do a `make all`; otherwise do a `make install`. The libraries will now be present in the `$MX_LIB` directory.

3. The smx load image consists of 5 programs—`kernel`, `mm`, `fs`, `inet` and `init`. All 5 can be generated by doing a `make image` in `$m/src/tools`. This creates an OS boot image in the file `$m/src/tools/image`.

   Note that if networking is disabled (by setting the `ENABLE_NETWORKING` constant to 0 in `$m/include/minix/config.h`), then the smx load image consists of 4 programs—all of the above except for `inet`. In the remainder of the document, where the 5 programs in the smx load image are discussed, remember that there are only 4 if networking is disabled.

4. All of the smx command sources are under `$m/src/commands`. Simple, one source file commands are in the `simple` directory, with more complex ones having their own directories. If you do a `make all` in `$m/src/commands` then all of the program binaries will be created. Smx comes with file systems that contain all of the standard program binaries, so you needn't do a `make all` for the commands as part of the installation process. If you modify a standard program, the best way to load it onto an smx file system is to use the `sunload` command from within smx.

4 Smx from the outside

Now that you have installed smx, here are instructions about how to run it. For details on smx internals, see section 5. A laboratory exercise used to introduce students to smx is available from the `$m/doc` directory.

4.1 Running smx

The major (SunOS) files and programs involved in running smx are:

- The bootstrap program `minix`. This program is linked to run under SunOS.
- The smx configuration file, that specifies (amongst other things) the locations of the image file and of all of the smx filesystems.
- The image file, which is a concatenation of five smx executables: `kernel`, `mm`, `fs`, `inet` and `init`.
- One or more smx filesystems, which are mounted to form a single hierarchy in the standard Unix fashion.

When `minix` is executed, it begins by reading the configuration file. It loads the programs from the image file into smx memory, and opens all of the smx filesystems specified in the configuration file. The `hdx0` file system is taken to be the root filesystem. Any other filesystems are either mounted by the smx `/etc/rc` script from within smx, or are left unmounted, in which case the user can mount them using the smx `mount` command.

To run smx for the first time, change to the `$m/src/tools` directory and run smx using the `minix` command (with no command line arguments). Standard MINIX boot messages appear—ignore the error messages from the networking software as networking is not enabled in the supplied configuration file. When you are prompted for a usercode log into smx as `root`. Most of the standard MINIX commands are available. In addition, `sunread` and `sunwrite` are available to transfer files between SunOS and smx, and `uemacs` (Microemacs) and `tcsh` are also available. The command `shutdown now` shuts down smx in a controlled fashion, ensuring that all file systems are sync'ed before smx terminates.

As no configuration file was specified on the command line above, smx reads configuration information from `./.minix`. The file `$m/src/tools/.minix` specifies the image file to be `./image` (this file was created during installation), and the three disks to be files in `../../disks`. The default `/etc/rc` mounts `/dev/hdx1` on `/usr/bin`, and `/dev/hdx2` on `/usr/man`.

At some point you may want to have a `.minix` file in your home directory that contains full pathnames so that you can run smx without needing to change to `$m/src/tools`.

4.2 Compilation for smx

C programs can be compiled for smx using the `mcc` script as the C compiler and linker. `mcc` provides most of the standard Unix C compiler options, as described in Appendix A.4. `mcc` relies on the environment variables `MX_INCL` and `MX_LIB` to locate the smx include files and libraries.

`mcc` uses `gcc` to compile C code and `ld` to link it. It also uses the smx program `elf2smx` to translate the ELF executable produced by `ld` into the smx `a.out` format. By default, `mcc` deletes the ELF executable once the smx executable has been produced. Having compiled and linked your program using `mcc`, you can transfer the executable to smx using `sunread` make it executable using `chmod`, then run it!

At present, the two MINIX programs (`bc` and `flex`) that rely on `yacc` and `lex` are not available under smx because the SunOS `yacc` and `lex` versions used to compile `bc` and `flex` expect different library functions to those provided in MINIX.

4.3 Multi-user smx

To allow additional login sessions, you must specify a hostname in the smx configuration file. If you select `minix1` as the host name, then the following line should appear in the configuration file:

```
host minix1
```

Once you have booted an instance of smx that uses the configuration file, additional logins are performed using the command

```
mlogin minix1
```

executed on the **same** Sun that the target smx instance is running on. `mlogin` terminates when the smx instance it is connected to is shutdown. It will also exit if the user types the `^Q^U^I^T` sequence of four control characters.

Note that every instance of smx running on a single Sun must have a different host name.

4.4 Multiple instances of smx

Each smx instance needs its own root file system and configuration file (which specifies the location of the root file system, amongst other things), but the `image` file and the `/usr/bin` and `/usr/man` file systems can be shared. To allow sharing of `/usr/bin` and `/usr/man`, edit `/etc/rc` under smx so that these file systems are mounted read-only (add a `-r` to the end of the `mount` command lines). Sharing `/usr/bin` and `usr/man` read-only between all members of a class can save a lot of disk space! If a student runs out of space on their root file system, they can always create another file system and mount that somewhere.

To create a new root file system, just copy an existing one under SunOS. For details on how to create a root file system of a different size, see section 5.5.1.

If you want to create files to support a second instance of smx (perhaps because you want to network together two smx instances by following the instructions in section 4.5 below), then you should:

1. Under smx, change `/etc/rc` so the `/usr/bin` and `/usr/man` are mounted read-only. Do this by adding the `-r` option to the end of the appropriate `mount` command lines. Then exit smx.

2. Create a second root filesystem by copying `$m/disks/root` to (say) `$m/disks/root2`.

3. Create a new minix configuration file by copying the file `$m/src/tools/.minix` to (say) `$m/src/tools/.minix2`.

4. Change the new configuration file so that the value of the `hdx0` option is the name of the new root filesystem.

5. Start up your new smx instance by using the command line:

   ```
   minix new-config-file
   ```

   where `new-config-file` is the name of the new configuration file that you have just created.

4.5 Networked smx

Once you have several smx instances, they can be connected using a simulated ethernet. This involves doing the following:

1. Choose a different internet number for each smx instance. Stick to using one class C network number (that is, vary only the last component of the internet number).
2. Edit the `/etc/rc.net` file on each smx instance, and change the internet number on the first line to be the one selected for that smx instance.
3. Decide on host names for all of your smx instances.
4. Edit the `/etc/hosts` files on all smx instances so that they all contain all of your <internet number, host name> pairs.
5. Shutdown all smx instances.
6. Decide on the name of the "relay file" that will be used to inform each smx instance of the UDP address of the packet relay program.
7. Add the line below to the configuration files of all smx instances.

   ```
   relay_file FILENAME
   ```

   where `FILENAME` is the name you selected in step 6 above.
8. Start `relay` using the command line:

   ```
   relay FILENAME
   ```

   where `FILENAME` is the name you selected in step 6 above.
9. Now you can start your smx instances, and they will be able to communicate via `telnet`, `ftp`, and so on.

Note that the hostnames used in `/etc/hosts` are totally independent of the hostnames used in conjunction with `mlogin`. If you don't want to have network support compiled into smx, set `NETWORKING_ENABLED` to 0 in `$m/include/minix/config.h`, and comment the `make inet` line out of `$m/src/tools/Makefile`.

4.6 Security and smx

There are a number of security issues that arise when using smx. These include:

1. The security provided within smx itself. A lot of effort has been put into making the file protections in smx the same as those in MINIX. The issues here are basically the same as in MINIX.

2. Other (SunOS) users logging in via `mlogin`. As explained further below, the connection between smx and `mlogin` is made through SunOS FIFOs created in a directory under `/tmp`. The permissions on those FIFOs will determine whether other users can use `mlogin` to connect to a running smx. If another user does have access to the FIFOs, then `mlogin` will connect them to your instance of smx. They will then have to go through the standard MINIX login process (so it's a good idea to have set passwords for the `root` and `bin` accounts, which have no passwords by default[1]).

   If another user can login to your running smx, then their actions within smx will be governed by the standard MINIX security system. The fact that they can run `sunread` and `sunwrite` means that they can read and write your SunOS files, because smx is running as one of your processes. Also, they can load their own smx programs that make SunOS system calls directly that may do things like delete SunOS files.

   You could make `sunread` and `sunwrite` executable by `root` only, but the very determined cracker could still use a binary editor to create an executable that made SunOS system calls. Also, given the relative lack of memory protection that exists by default, data structures in the operating system could be corrupted.

   In summary, you don't want hostile users logging into your smx instances because once logged in they can easily be disruptive. If you don't specify the `host` option in the configuration file, then `mlogin` is impossible. If the `host` option is specified, then the permissions on the FIFOs, and the passwords on your smx usercodes are your two lines of defence.

3. The smx ethernet emulation is another potential security problem. The `relay` program your smx instance is communicating through may have been started by another user, which means your ethernet traffic is under the control of another user. If you are using your own `relay`, then there is nothing to stop other smx instances from connecting to it (although if the SunOS file containing the UDP address of the `relay` program is not world-readable a would-be cracker will have to find the port number by trial and error). If a cracker does connect to your `relay` program, then they will be able to direct arbitrary ethernet packets at your smx instances.

5 Smx from the inside

The preceeding sections described how to install smx, and how to use it. In this section we look inside smx and see how MINIX has been changed to produce smx. We first discuss the compilation and linking of smx programs, which is done outside smx. We then go on to discuss the booting process, management of the address space (what goes where, and protection issues), "interrupt" handling, emulation of devices, new smx programs, and debugging of smx.

5.1 Compilation

We will break the discussion of compilation into two parts. The first deals with compilation and linking of the small number of programs that run under SunOS. The second deals with

[1]Note that some of the internet remote login programs will not work for usercodes that do not have a password.

compilation and linking of smx executables, and covers both the operating system and user programs. Finally we give some guidelines on porting software to smx.

5.1.1 Compilation of SunOS executables

The new directory `$m/src/Solaris` contains scripts and programs for use under SunOS. The purposes of the `minix`, `mlogin` and `relay` programs have already been described. The remaining five programs and scripts, `mcc`, `elf2smx`, `combine`, `make_map_file` and `next_prog_addr`, support the creation of smx executables in some way, as described in the next sub-section.

The programs in `$m/src/Solaris` are compiled into standard SunOS 5.x executable programs (in ELF format) by `gcc`. The standard SunOS header files are used (for `stdio.h`, etc). Where an smx header file is to be included, a relative pathname (to `../kernel` or `../../include`) is used. One ramification of this is that if students copy a program from `$m/src/Solaris` to modify it, then `../kernel` and `../../include` must contain the appropriate header files (the easiest way to do this is to setup two symbolic links).

The installation procedure includes doing a `make install` in `$m/src/Solaris` to compile and install the scripts and binaries in the desired directory.

5.1.2 Compilation of smx executables

With the execption of the programs in `$m/src/Solaris`, and a small number of programs that are run to produce source files during creation of smx executables, all other smx programs are compiled using the smx header files (in `$MX_INCL`) and library files (in `$MX_LIB`), and are converted to smx executable format. The `mcc` script is used to compile all `.c` and `.s` files to `.o` files, and to link all smx user programs (except `init`). `mcc` acts as a wrapper for `gcc`, `as` and `ld`, ensuring that in each case appropriate options are supplied. `mcc` accepts most "standard" Unix compilation options, passing them on to the programs it invokes as needed.

Every smx executable program file is created by `elf2smx` from a statically linked ELF executable. Consider the following command line:

```
mcc -o blarg blarg.o
```

When `mcc` links the program, it puts the ELF executable produced by `ld` into `blarg.elf`, then converts `blarg.elf` to `blarg` using `elf2smx`. If the `-N` option has been specified to `mcc` then `blarg.elf` will not be deleted, otherwise it will be deleted. One use of `blarg.elf` is that it contains a symbol table, whereas smx executables do not.

The smx filesystems supplied with smx contain compiled versions of all smx user programs, so the installation process does not involve compiling them. A `make all` in `$m/src/commands` compiles all user programs (except `init`). Compilation of individual programs is also provided for. New versions of standard smx executables can be loaded into smx using `sunload`.

The libraries and C startup files are created during installation by a `make install` in `$m/src/lib`.

Executables of the programs in `$m/src/test` are found in `/usr/test` under smx. They can be compiled with a `make all` in `$m/src/test`, and loaded into smx using the smx script `testload` in `/usr/test`. Note that you will have to edit both `sunload` and `testload` so that each sets `sunosdir` to the name of the appropriate SunOS directory (it is unlikely that your path names will be the same as mine!).

The `mcc` script is used to compile all C and SPARC assembler files for the smx executables loaded from the image file by `minix` (`kernel`, `mm`, `fs`, `inet`, `init`), and linking is performed by `ld`, with each of the five `Makefiles` containing a custom `ld` invocation. The custom `ld` invoction is needed because the `kernel` has its own special entry point, and the other programs use a non-standard C startup routine, which is found in `$MX_LIB`. The start-ups for these five programs are non-standard because they are loaded from the image file, and not `exec`'ed. In the case of the `kernel`, multiple stacks are needed (the layer 1 stack, and 1 stack per layer 2 task). These are all found in the `kernel` data segment, so the MINIX "gap" is 0. For the other four programs, the stack size to use is specified in the Makefile, and this is written to the gap field in the smx executable header by `elf2smx`. Smx allows for growth of data and stack segments in user processes, so the gap field in the smx executable header is ignored for these programs (although the old gap values are still set by most `Makefiles`).

If you want an ELF executable kept for one of the "image" programs, comment out the `rm` command from the appropriate `Makefile`. Each of the five `Makefiles` runs the `combine` script after re-linking the program. If all executables are present, `combine` concatenates them to create a new image file as `$m/src/tools/image`.

5.1.3 Using `ld` map files to assign virtual addresses

As will be dicussed below in 5.3, smx user programs (including `init`) are linked to run at reasonably standard virtual addresses for SunOS processes. At present, `ld` links each smx user program to have its text segment at `0x10000`, and its data segment aligned on an `0x2000` byte boundary. `kernel`, `mm`, `fs` and `inet` are linked to run in distinct address ranges at much higher addresses. The kernel is linked to start executing at an address specified in `$m/src/kernel.map`. `mm` must be linked to execute at the address that immediately follows `kernel`, `fs` must be linked to follow `mm`, and `inet` must be linked to follow `fs`.

The `mm`, `fs` and `inet` `Makefiles` are set up to regenerate the respective `map` files (and re-link the executable according to the new map file), whenever the executable of the preceding program changes. The script `make_map_file` is used to generate a new map file, and it uses `next_prog_addr` to determine where the previous executable ends in virtual memory. If you change `kernel` in a way that causes its size to grow and do a `make` in `$m/src/kernel` only, then the resulting `image` file will contain a `kernel` and an `mm` that overlap in virtual memory. The `minix` program detects such situations and aborts the boot if it discovers one. If you do a `make` in `$m/src/tools`, then the five files necessary for the the image file are created in the right order, and the address assignment will be correct. As well as specifiying the starting addresses of the respective text segments, the `kernel`, `mm`, `fs` and `inet` map files specify that the data segment is aligned on an 8Kb boundary.

The data segment is aligned on an 8Kb boundary by including `A8192` as a data segment attribute in the `$m/src/lib/smx_userprog.map` and `$m/src/kernel/kernel.map` files, and the map files generated by `make_map_file`. I wanted the data segment to begin at the next 8Kb boundary after the end of the text segment. The `A8192` attribute doesn't actually do this—it leaves an 8Kb gap in the address space (perhaps a little more so that the data segment starts on a double-word boundary). This leaves more unused space than if you have the data segment begin at the next 8Kb boundary. If you add in the `R8192` data segment attribute, then the data segment does begin at the next 8Kb boundary. Sadly, the Solaris 2.4 `ld` doesn't support this attribute, so I had to remove it from the three places listed above. If you're running on Solaris 2.5, then you can add the `R8192` back in (in addition to the `A8192`), and save an average of 4Kb per program (on disk and in memory).

5.1.4 Porting software to smx

Although most existing MINIX software can be ported to smx without alteration, there are some areas where changes may be needed. The following list of observations should be used as a guide when porting software to smx:

- In smx executables, read-only data (such as string constants) is stored in the text segment, which is not writable. Changes have been needed to the small number of MINIX programs discovered (so far) that write to a string constant.
- In the SPARC architecture, N byte primitive objects (integers and floating point variables of different sizes, and pointers) must be aligned on N byte boundaries. A few pieces of MINIX software weren't sufficiently careful when doing some non-portable pointer casting and had to be modified to run under smx.
- In smx, dereferencing (following) a null pointer causes a segmentation violation. This doesn't happen in MINIX, and many pieces of MINIX software have had to be fixed to remove null pointer dereferences.
- Any function that has a variable number of arguments might cause problems if it relies on all arguments being on the stack. This is because on the SPARC the first 6 parameters are passed in registers, rather than on the stack. Programs that use `<stdarg.h>` correctly will be OK.
- In smx, `int`'s are 32 bits and `short`s 16 bits. This difference in length has caused some problems for MINIX code in the past.
- Any assembly language code must be rewritten.

5.2 The booting process

Smx booting proceeds in two phases. The `minix` program gets things to the point at which execution inside smx is possible. Execution then switches inside the smx kernel, which executes a somewhat modified version of the standard MINIX bootstrap code.

5.2.1 Booting—the `minix` phase

The first boostrap phase is carried out by the `minix` (SunOS) program. The `minix` program gets most of its configuration information from a file. The debug flag and memory protection can be specified on the command line, overriding any settings in the configuration file.

The main functions of the `minix` program are to:

1. Read the configuration file, and open various smx devices on well-known (to `minix` and `kernel`) SunOS descriptors. Every smx filesystem listed in the config file is opened on its own descriptor. Descriptors for the controlling terminal are opened. If the `host` option appears in the configuration file, a sub-directory is created in `/tmp` containing FIFOs. These FIFOs are opened on well known descriptors. If a network relay is specified then a UDP/IP socket is opened on a well-known descriptor and connected to the `relay`

program. The UDP address of the socket created is registered with the `relay` program so that broadcasts are relayed to this smx instance. If a logfile is specified it is opened for writing on a well known descriptor.

As well, various settings are read from the configuration file, including the name of the image file, the smx memory size, the debug flag and the memory protection setting.

2. A child process is created that will run smx. The parent waits for the child to exit, and then tidies up by restoring the settings of the controlling terminal, and deleting the FIFO directory in `/tmp` (if one was created).

3. The child creates a temporary file that is the same size as smx "physical" memory, and maps it into the child's address space at the virtual address assigned to the beginning of `kernel`'s text segment. This mapped area is the physical memory of smx. The five programs in `image` are then read into this memory.

4. An `smx_bootinfo` structure at the start of `kernel` data segment is filled in. It contains the physical memory size of smx, the debug flag, the protection level, the ethernet address (which is the UDP address of the socket created to send and receive ethernet packets) and details of all the programs loaded (text and data segment sizes, text and data segment virtual addresses, and entry point). `$m/src/kernel/main.o` must be the first object file specified to `ld` when `kernel` is linked so as to ensure that the `smx_bootinfo` structure comes at the start of the `kernel` data segment.

5. Execution then switches to the `kernel` entry point.

5.2.2 Booting—the `kernel` phase

The `kernel` entry point is in a small piece of SPARC assembler code that sets the stack pointer to point into the layer 1 stack, and then calls the C function `main`. The main pieces of smx-specific code added to the standard boot sequence involve setting up the handling of SunOS signals, and setting up of memory protection (both are described further below).

5.3 Management of the address space

As discussed above in 5.1.2, `kernel`, `mm`, `fs` and `inet` are linked to run in contiguous chunks of address space starting well above address 0 (at present, `kernel` is linked to start 32Mb into the address space). During the boot sequence, smx "physical" memory is mapped into the smx SunOS process starting from the address at which `kernel` has been linked to begin at, and `kernel`, `mm`, `fs` and `inet` are loaded into the addresses they were linked to run at.

User processes in smx occupy three areas of physical memory, which contain a text segment, a data segment and a stack segment. All user programs are linked to run at reasonably standard SunOS virtual addresses, which are very near the bottom of the address space. When `kernel` is about to switch execution to a user process, it maps the virtual address ranges where the smx user program has been linked to run to the appropriate areas of the mapped temporary file that contain the three chunks of "physical" memory allocated to that particular smx process. The first smx user process to run wipes out the mappings to the text and data segment of the `minix` program, but it has done its job and execution never returns to it.

In MINIX, one chunk of memory is allocated for the data/gap/stack area, with the space left for growth of the data and stack segments specified in the header of the executable. In smx, separate data and stack segments are supported (for user processes only—the MINIX approach is used for `kernel`, `mm`, `fs` and `inet`), and they can grow as needed. Initially the data segment of a user process is large enough to contain the data and bss, and the stack is large enough to hold the initial stack. The stack of each user process is mapped so that it immediately precedes the `kernel` text segment, which leaves a large gap in virtual memory that both data and stack segments can expand into.

The SunOS `mprotect` system call gives us the ability to go some way towards protecting the various smx address spaces from each other. There are three protection levels, as described below. The protection level to use can be specified in the smx configuration file, and on the `minix` command line. The levels are:

- none. All smx "physical" memory is `rwx` at all times; "virtual" memory mappings for user processes are `rwx`.
- half (the default). `kernel`, `mm`, `fs` and `inet` have their text segments in physical memory set to be `r-x`; the rest of "physical memory" is `rwx` at all times. Virtual memory mappings are `r-x` for text; `rwx` for data and stack.
- full. When execution is in `kernel`, the `kernel` text segment is `r-x` and the data segment is `rwx`. The rest of physical memory is `---`, except for when `kernel` is copying data to or from other address spaces during which the source memory is made readable and the destination writable.

  When execution is in a process outside the kernel, the virtual mapping for the process' text segment is `r-x` and for the data and stack is `rwx` (the mappings in physical memory remain at `---`, except when `mm` or `fs` or `inet` is running when the protection on physical memory is changed because their virtual addresses are the same as their physical addresses). Much of `kernel` is `---`. Some code and data structures (sufficient to turn `kernel` access on and off) are accessible, but not writable. The layer 1 stack is `rwx`, but contains nothing of relevance while execution is outside `kernel`. The kernel cannot, therefore, be corrupted.

Because of the amount of context switching done by smx, full protection is not the default because it is slow. Full protection is useful in tracking down memory corruption problems, however.

One final memory-related point. The click size in smx is 8Kb, as that is the largest page size on current Suns.

5.4 Interrupt handling

In smx, execution is forced into layer 1 by SunOS signals rather than by hardware traps and interrupts. The bulk of the signal handling code is in the files `$m/src/kernel/mpx.c` and `$m/src/kernel/sunsighandle.c`. The latter file includes the standard SunOS `<signal.h>` and contains functions that draw heavily on the information in that header file. To avoid type clashes, many of the standard smx header files cannot be included by `sunsighandle.c`.

The SunOS signal mask used when execution is in layer 1 allows only "error" signals (such as illegal instruction) to be delivered, so what MINIX calls "nested interrupts" cannot occur.

A single signal handler (SunOSsig) is installed to handle all SunOS signals. SunOSsig is responsible for enabling kernel access (if full protection is on), saving the context of the interrupted smx process, calling the appropriate smx handler within layer 1, picking a process to resume, and resuming that process (which enables delivery of all SunOS signals).

There are two SunOS signals that arise asynchronously in ordinary smx operation, and one that arises synchronously. The alarm signal is delivered regularly (currently 20 times a second) to simulate clock ticks. The IO signal is received whenever terminal or ethernet input becomes available. The three smx system calls (send, receive and sendrec) force execution into kernel by sending a USR1 signal to the SunOS process running smx. The ALRM, IO and USR1 signals have their own smx handlers.

The exception handler is called for all other SunOS signals. If a segmentation violation occurs in a user process, and the address being accessed might be on the stack (the address being accessed lies between the data and stack segments) then a SIGSTKFLT is relayed to the memory manager to have the stack extended. Otherwise, if the signal is an "error" signal and arose in an smx user process, exception maps the SunOS signal number to a MINIX signal number and has the memory manager deliver the signal to the offending smx user process. If an "error" signal occurs in layer 1, 2 or 3 code then smx is aborted and various dumps are produced to aid debugging. If the SunOS TERM signal is received then smx immediately terminates. All other SunOS signals are quietly ignored.

In smx, the stackframe_s type (used in MINIX to hold a process context) is defined to be the same size as SunOS ucontext_t structure, which holds a SunOS context. Because of clashes between the smx and SunOS header files, we decided not to include the SunOS <ucontext.h> file directly into smx kernel source files. Instead, the program $m/src/kernel/make_offset.c is used to create the uc_offset.h header file from <ucontext.h>. The uc_offset.h file specifies the size of the SunOS ucontext_t and gwindows structures, as well as the offsets of many fields within the ucontext_t structure. Smx kernel sources include uc_offset.h without introducing conflicts with smx header files.

A gwindows structure is only part of the context of an smx process if, when a signal occurs, it is not possible to flush the contents of the in-use SPARC register windows to the stack. One way of handling such events would be to have a gwindows structure as part of the saved context of every process. Because the gwindows structure is large (currently 2112 bytes, as against 448 for a ucontext_t) and seldom used (only needed while a user process is having its stack extended) an alternative (though less elegant) approach was taken. There is a single save area for a gwindows sturcture. When a SunOS signal arrives, and the saved context contains a gwindows structure, the structure is saved and gwin_proc is set to point to the relevant process. No user processes can run while gwin_proc is non-zero (as we only have one save area). Soon after, mm gets to run. It expands the stack of gwin_proc which then becomes runnable again. gwin_proc is the next user process to run. When execution switches to gwin_proc, the registers in the saved gwindows structure are restored (removing the restriction on scheduling user processes) and gwin_proc is set to zero.

Another issue in this area is delivery of smx signals to signal handlers in smx processes. When an smx signal occurs that is being handled in the smx process the signal is destined for, the smx kernel copies the saved context of the smx process onto the stack of the smx process. A stack frame is then allocated for the signal handler, and the context of the smx process saved in the kernel process table is adjusted so that:

- The program counter is set to the address of the library function __sigreturn in the smx process.

- Register `%o0` is set to the smx signal number, register `%o1` is set to 0, and register `%o2` is set to point to the saved context on the stack. These are the three parameters for the signal handler.
- Register `%o5` is set to the address of the signal handler.

When the kernel next switches to the smx process, execution resumes from the beginning of `__sigreturn`. It saves a copy of the pointer to the context, then calls the handler (whose address is in `%o5`). When the handler returns, `__sigreturn` calls the standard MINIX `_sigreturn` function, passing to it the address of the saved context. `_sigreturn` makes a MINIX system call to MM to have the saved context restored so that the smx process resumes at the point it was at when the signal occurred.

5.5 Devices

All smx devices are somehow emulated using SunOS system calls. Here we will discuss emulation of disks, the ethernet interface, terminals, the hardware clock and the various memory device files. Printing is not provided by smx; the smx printer device driver simply replies with `EINVAL` to any messages sent to it.

5.5.1 Disks

Each smx file system is stored as a single SunOS file. Mappings between smx file systems and SunOS files are recorded in the smx configuration file. Up to eight file systems, `hdx0` to `hdx7`, are currently provided for. In the configuration file, a mapping between a file system and a SunOS file is specified by giving the file system name as the option name, and the SunOS file name as its value. The `hdx0` file system is the root file system, and this option must always be specified. In the standard setup, `hdx1` is mounted as `/usr/bin` and `hdx2` is mounted as `/usr/man`. These file systems are stored in files in `$m/disks`.

The `minix` program opens file system `hdxN` on SunOS file descriptor `DISK_FD + N`. When the disk device driver goes to access a disk with minor device number N, it uses SunOS file descriptor `DISK_FD + N`. The MINIX floppy disk driver has been converted to the smx disk driver, and the winchester driver has been omitted from smx.

It is easy to create new smx file systems. One way to create a new file system is to simply take a copy of an existing file system of the appropriate size. This new file system can then be mounted, and any existing files deleted, under smx. If this is not possible, a new file system can be created by doing the following:

1. Decide on the device name to use. Let's imagine that `hdx3` is selected.
2. Add the following line to the smx configuration file:

   ```
   hdx3 filename
   ```

   where `filename` is the SunOS file name of the file that is to hold the new file system.
3. Create the SunOS file `filename` using `touch(1)` under SunOS.

4. Enter smx. Create the file system using:

   ```
   mkfs -b fs_size /dev/hdx3
   ```

   where `fs_size` is the number of 1Kb blocks in the file system.

5. Mount the new file system:

   ```
   mount /dev/hdx3 mount_point
   ```

   where `mount_point` is the name of the smx directory that you want to mount the new file system on.

Creating a new root file system requires some care. If the new root file system is to be the same size as an existing root file system, then the SunOS `cp` command can be used to create the new root file system. Otherwise, the new root file system should be created as described above, and initialised by doing the following (in all cases use the `-p` option of `cp` to preserve file attributes).

1. Copy all dot files from `/` to the root directory of the new filesystem.

2. Copy the `/bin` and `/etc` directories to the new root. Use the `-r` option of `cp` (as well as `-p`) to copy recursively, and preserve the hard link structure.

3. Create `dev`, `mnt`, `root`, `tmp`, `usr`, `usr/bin`, `usr/man` and `usr/adm` directories in the new root, and make the permissions, owner and group of each new directory the same as the corresponding directory in `/`.

4. Copy (`-r -p`) all sub-directories from `/usr` (except for `/usr/bin`, `/usr/man` and `usr/adm`) into the `usr` on the new root.

5. Change to the `dev` directory in the new root, and use the command:

   ```
   MAKEDEV sun
   ```

   to create the new device files.

The new root filesystem is now ready for use. If you want a login record to be kept on it, create an empty `usr/adm/wtmp`.

It is possible to mount an smx file system read-only (to facilitate this, the `minix` bootstrap tries to open a file containing an smx filesystem read-only if the file cannot be opened read-write).

5.5.2 Ethernet emulation

The ethernet task uses the UDP/IP socket opened by `minix` on `ETHER_FD` (and connected to an instance of the `relay` program) as an ethernet port. The ethernet task returns the UDP/IP address bound to `ETHER_FD` as its ethernet address (consequently `rarpd` cannot be used because ethernet addresses are not constant). It is very handy that ethernet addresses and UDP addresses are both 6 bytes long.

Outgoing packets are written immediately to `ETHER_FD`, and a (MINIX) reply message is sent back to `inet` to that effect. Whenever a packet arrives, a SunOS SIGIO is received by smx. The incoming packet is read into a buffer if there is room, and is discarded if there is no room.

One problem struck in compiling the MINIX networking software for smx was that the `inet` process casts pointers into a packet buffer to be pointers to various sorts of structures. The fact that the ethernet header is 14 bytes was leading to alignment problems for pointers to the various IP headers! For this reason, two bytes of padding have been added to smx ethernet headers to make them 16 bytes. The header expansion involved changes to `$m/include/net/gen/ether.h`, `$m/include/net/gen/eth_hdr.h` and `$m/src/inet/generic/arp.c`. Because the present implementation involves exchanging ethernet packets between smx instances only over a simulated ethernet, this expansion in the ethernet packet header hasn't caused any problems.

It may seem strange that all packets go via the `relay` program, given that it would be possible to send a (non-broadcast) ethernet packet direct to the smx instance it is destined for (the UDP/IP address of the destination port is, after all, in the packet header). Smx can make use of SunOS system calls only, and not SunOS library calls. It is easier to just have the `ETHER_FD` socket `connect`ed by the bootstrap, and just to call `write` in the ethernet driver, than it is to emulate the SunOS library calls needed to individually address each UDP message from the ethernet driver. Also, having all ethernet messages go through one program is somewhat similar to having all messages sent over a shared bus.

5.5.3 Terminals

When `kernel` and `minix` are compiled, the number of consoles supported (`NR_CONS`) is compiled into both programs. When smx is being booted, if the `host` option appears in the configuration file then the directory `/tmp/hostname` is created. In this directory, the following files are created:

- `lock`, which contains two integers in binary form: the number of consoles available via `mlogin` ($NR\_CONS - 1$ because the system console is not implemented through `mlogin`), and a bit map recording which of the $NR\_CONS - 1$ "lines" is currently available.
- For each line I ($1 \leq I \leq NR\_CONS - 1$), a pair of FIFOs called `in.I` and `out.I`. Smx reads input for terminal I from the FIFO `in.I`, and sends output to terminal I to `out.I`.

For terminal line I, `minix` sets up descriptor `TERMINAL_FD + 2`I for reading input from I and `TERMINAL_FD + 2`I `+ 1` for writing output to I. Line 0 is the console, so its descriptors refer to the controlling terminal of the SunOS process that is running smx. For the other lines, the descriptors refer to the FIFOs created in `/tmp/hostname`.

For both the console and additional logins via `mlogin`, the controlling SunOS terminal is set to do no input or output processing, leaving those functions to smx. The SunOS terminals are setup to interrupt (with the IO signal) when new input is available. The terminal input handling done by the smx terminal device driver is quite simple—if input does not fit into the terminal's input buffer then it is discarded.

At the user level, an entry for `xterm` has been added to `/etc/termcap`, and all terminal type entries in `/etc/ttytab` have been set to `xterm`. The termcap entry for `xterm` specifies 24 lines; you can change it if your `xterm` windows are larger. If you use a different X terminal emulator, then you will need to add an entry for it to `/etc/termcap`.

5.5.4 Memory

The MINIX memory device driver is responsible for four special files—`/dev/null`, `/dev/mem` (physical memory), `/dev/kmem` (kernel virtual memory) and `/dev/ram` (the RAM disk). The null device works just as in MINIX, and the RAM disk is not implemented in smx (there is no need for it), so neither is discussed further.

As we saw in 5.3, the "physical" memory of smx is an area mapped into the address space of the SunOS process well above address 0. The `kernel`, `mm`, `fs` and `inet` programs are all linked to run at the addresses they are loaded into in this mapped area, while user processes are mapped to run much lower in the address space.

For both the `mem` and `kmem` device special files, location 0 corresponds to address 0 in the SunOS address space. In the case of `kmem`, the special file extends to the end of the kernel data segment, and in the case of `mem` to the end of smx "physical" memory. At first glance, this approach of having a large gap at the start of the `mem` and `kmem` special files may seem strange. It has been done this way so that programs like `ps` that compute physical addresses then retrieve data from those addresses can operate in the same way as in standard MINIX, that is by doing an `lseek` on the appropriate device to the physical address computed, followed by a `read`.

5.5.5 Clock

The main change to clock handling introduced by smx is that in the `clock_task` main loop `realtime` is updated by finding the current SunOS time using `gettimeofday`, rather than by having `pending_ticks` added to it. This is done because the frequency of SunOS ALRM signals is likely to be much less stable than that of the hardware clock available in native MINIX versions.

5.6 New smx programs

A few smx programs have been added to the standard MINIX ones. Man pages are available under smx.

5.6.1 `sunread` and `sunwrite`

This is actually one program with two hard links to it. A single command line argument is expected, which specifies the pathname of a SunOS file. When invoked as `sunread`, the program copies the SunOS file to stdout. When invoked as `sunwrite`, the program copies stdin to the SunOS file.

5.6.2 sunload

This shell script installs the standard binaries (plus any support files) into `./usr`, `./usr/bin` and `./usr/lib`. If you want to overwrite the current standard binaries, run `sunload` from `/`. If you are creating a new set of binaries, run `sunload` from the appropriate directory.

`sunload` is needed because the smx programs are compiled and linked under SunOS, and the `Makefile`s under SunOS cannot install software into smx file systems. See the manual page under smx for more details.

5.6.3 tcsh and uemacs

These programs have been added to smx because I like using them! Be warned that the support for shell script execution is built into `sh` and not into the memory manager, so shell scripts cannot be executed from `tcsh` by just giving the script name as a command (though of course you can still enter the command line `sh scriptname`).

5.7 Debugging

There is some debugging support provided in smx. This includes:

1. When starting `minix`, debugging output can be enabled via the `-d` command line option, or by specifying the `debug` option in the configuration file with a value of `on`. When debugging is enabled, `minix` prints various pieces of debugging information, and also passes on the `debug` flag to `kernel`, which causes some information to be printed during system startup. At present, no debugging information is printed once booting is completed. Within `kernel` the functions `debug_int`, `debug_str` and `debug_char` are called to produce the debugging information. They produce output only if debugging is enabled.

2. Calls to `printf` can be added to `kernel`, `mm`, `fs` and `inet`. Output appears on the console.

3. If a SunOS error signal is received while execution is in `kernel`, or a kernel panic occurs, then various dumps are printed containing information on the location of the error, and on the state of smx at the time of the error. If the system locks up then control-underscore can be typed at the console to induce a panic.

4. When a SunOS error signal is received while execution is in an smx user process then, in addition to the appropriate smx signal being delivered, details of the signal are printed on the console. I have found this a useful debugging aid. If it becomes annoying, just comment out the `printf` statement in `$m/src/kernel/sunexception.c`.

5. The interaction network monitor (see section 6) records details of some aspects of system operation, with particular emphasis on message passing. By default it records the events that occur as a direct result of each user input. This means that no events are recorded until the user starts entering a usercode during login. To have all events recorded, remove the comments around the definition of `LOG_ALL` in `$m/src/kernel/logging.c`.

   Be warned that log files grow very quickly!

6. The full set of MINIX test programs are provided in `$m/src/test` under SunOS, and `/usr/test` in smx. All of the binaries can be created with a `make all` in `$m/src/test`. Under smx, running `testload` in the `/usr/test` directory loads all of the test files (remember to modify the `testload` script to reflect the SunOS path of the `$m/src/test` directory). To run the tests, execute the command `./run` in `/usr/test`.

   The test programs are best run by the `bin` usercode, as several tests abort if run by `root`. On the other hand, some of the test programs perform additional tests when run by `root`, so running the tests from `bin` and from `root` gives the best coverage.

   The current version of smx passes all of the tests, although one or two race conditions exist in the test suite that sometimes result in spurious errors being reported. None of the test programs has been changed in a material way. The two test scripts have been modified to remove references to commands that are not available under smx (such as `cc`). While the test suite isn't really a debugging tool, it will help to uncover bugs introduced into the OS.

If an smx program falls over, then you will often be left with the address of the instruction where the program crashed, and you will want to find the function that contains that address. The `kernel`, `mm`, `fs` and `inet` text segments occupy unique address ranges (as reported by `minix` with debugging enabled), so it is easy to determine whether a crash occurred in one of them. Once you have determined which program has crashed, you can locate the function that contains the address where the crash occurred by running the SunOS `nm` command on the ELF version of the executable. You can disassemble the ELF executable to find out more about what the program was doing at the point it crashed. By default the ELF executables are deleted after the smx executable has been produced. See 5.1.2 for details on how to stop the ELF version of an smx executable being deleted.

6 The interaction network monitor

Interaction network recording is a technique developed for monitoring distributed systems. Substantial prototypes have been developed for SunOS 4.0 [2] and more recently for Amoeba [1]. The third implementation of the monitor is for smx.

The idea behind the interaction network approach is that it should be possible to record, for each user input, all events that result from the input. The interaction network approach is aimed at providing improved understanding of interactive processing done by complex systems, and in particular is aimed at providing performance information.

The smx `kernel` contains event recording probes and an event recorder. The `minix` bootstrap program opens a SunOS log file if the `logfile` option appears in the configuration file, and the event recorder in `kernel` writes event records relating to (by default) interactive processing to the log file if monitoring is enabled. In the other IN monitors, monitoring can be turned on and off during system operation. This is not yet possible in the smx version, where monitoring can only be enabled or disabled for an entire smx session.

At the end of an smx session during which logging was enabled, you will be left with a file (whose name was specified as the value for the `logfile` option) containing all of the event records recorded during that session. Four programs are available in `$m/src/Solaris/IN` for analysis of the interaction networks present in the session log file. The `logdump` program gives a text dump of all event records in a specified log file. The `insplit` program takes a specified session log file and groups the event records into interaction networks; one per user input. Each interaction network is written to a separate task log file. Each of the task log files produced by `insplit` has a name that consists of 8 hex digits. The `totcl` program takes a specified task log file and from it produces an input file (with a `tcl_in` extension) for `browser`. The `browser` program takes the specified `tcl_in` file and uses it to produce a graphical display of an interaction network.

Doing a `make all` in `$m/src/Solaris/IN` compiles and links `logdump`, `insplit` and `totcl`. The `browser` program is written in TCL/Tk. You will have to change `browser` so that the first line specifies the full path-name of the `wish` program on your system. See `$m/doc/tr-cosc.04.96_ps` for examples of interaction networks recorded for smx 1.7.4 (all of which should be the same as those recorded for smx 2.0).

The IN monitoring package is something of an "optional extra", and so is not yet documented as well as the rest of smx.

Appendix A: Manual pages for SunOS commands

Smx involves use of eight SunOS programs. Three of the eight (`combine`, `make_map_file` and `next_prog_addr`) exist only to support production of the image file in various ways. They are invoked when needed by the relevant `Makefiles`. Because they are not invoked directly by the user, and because they have been adequately explained in section 5.1.2, they will not be discussed here. In this appendix we give information on use of three programs that support execution of smx (`minix`, `mlogin` and `relay`), and two program that support compilation and linking of smx executables (`mcc` and `elf2smx`).

Appendix A.1: `minix`

The syntax for the `minix` command is:

```
minix [-d] [-m (none|half|full)] [config-file]
```

This program opens smx devices, loads the smx kernel, memory manager, file system, internet and init programs into memory and passes control to the smx kernel. A configuration file is used to specify how the smx system is to be set up. If a `config-file` appears on the command line, then that is the name of the configuration file. Otherwise, `minix` looks for the configuration file in `./.minix`, and if that file doesn't exist then it looks for `~/.minix`.

The configuration file is processed by `minix` in a line-oriented fashion. Blank lines are ignored, and lines that start with a `#` are treated as comment lines. All other lines begin with an option name which is followed by the option value. Currently, all options and option values are a single word. Some options (`debug` and `protect`) can be specified on the command line. The `-d` option is equivalent to `debug on` in the configuration file, and `-m value` is equivalent to `protect value`. An option value on the command line takes precedence over a value in the configuration file.

The options that can appear in the configuration file are:

- `hdxI`, where *I* is in the range 0 to 7. The value is the pathname of a SunOS file that contains an smx file system. The specified file system is made available as `/dev/hdxI` under smx. The `hdx0` option must appear in every smx configuration file, as it specifies the root file system. An attempt to access `/dev/hdxI` under smx where there was no `hdxI` option in the configuration file will result in an "Unrecoverable disk error" being reported.

- `image`. The option value is the name of a SunOS file that contains a "boot image", which consists of the smx `kernel`, `mm`, `fs`, `inet` and `init` programs. The `image` option must appear in every configuration file.

- `host`. This option is needed to give your smx system a "host name" so that the `mlogin` program can be used to provide additional logins. If the option value is (say) `paulminix`, then the directory `/tmp/paulminix` is created. A file to control terminal allocation (`/tmp/paulminix/lock`) is created, as well as FIFOs to provide data transfer between `mlogin` instances and smx. The default is for there to be no host name, and no `mlogin` capability.

- `relay_file`. Specifies the name of a file written by a running `relay` program. If this option is specified then networking is enabled. The `relay` process (contacted at the UDP address specified in the file whose name is the `relay` option value) relays packets between smx instances, thereby emulating an ethernet segment.

- `memory`. Specifies the amount of memory to be allocated for smx. The units used are kilobytes. Because the current five "system" programs require nearly 1.2Mb, the memory specified must be at least 1536 (1.5Mb). The current maximum is 32768 (32MB), but this could easily be increased if needed. The default is 3072 (3Mb).

- `protect`. Specifies the level of memory protection between smx processes. Basically, level `none` gives no protection, level `half` prevents writes to text segments, and level `full` gives almost complete protection, by giving each smx process access only to its own address space. The default is `half`, because the `full` option introduces considerable overhead.

- `debug`. Either `on` or `off` can be specified as values. If debugging is enabled, additional messages are printed by `minix` and `kernel` during the booting process. By default, debugging is off.

- `logfile`. The value for this option is the name of a SunOS file to hold event records produced by the interaction network monitor. Its presence enables monitoring. If the `logfile` option is not specified, then no logging is performed.

To be able to run `minix` no matter what your current directory is, put a `.minix` file in your home directory, and make sure that all SunOS pathnames in the file are full pathnames.

If smx locks up, and you are unable to terminate it using `shutdown`, and control-underscore on the console won't terminate smx, then you will have to use `kill` under SunOS to terminate smx. Remember to `kill` the child process so that the parent process can tidy up.

Appendix A.2: `mlogin`

The syntax for the `mlogin` command is:

```
mlogin hostname
```

where `hostname` is the name specified for the `host` option in the configuration file of an instance of smx that is currently running on the Sun on which `mlogin` is invoked. If such an smx instance exists, and a terminal line is available, and the user invoking `mlogin` has write permission on the files in `/tmp/hostname`, then `mlogin` provides a terminal line to the relevant smx instance. To exit from mlogin, hold down the control key and type `quit`.

Appendix A.3: `relay`

The syntax for the `relay` command is:

```
relay relayfile
```

`relay` opens a UDP socket, and writes the name of the host it is running on and the UDP port number of its socket to `relayfile`. Instances of smx use this information in connecting to a running `relay` program via the `relay_file` option in the configuration file.

After writing `relayfile`, the `relay` program enters a packet processing loop. Packets come in three flavours:

- packets whose destination "ethernet" address is the UDP address of the relay program. Such a packet is sent by the `minix` bootstrap program to register with the `relay` program the "ethernet" address of the sending smx instance (which is actually the UDP address of a socket created by the smx instance). The "ethernet" address registered is the source address in the ethernet header.
- broadcast packets. These are forwarded to all registered smx instances except the sender.
- packets addressed to another smx instance. Such packets are forwarded to the appropriate smx instance, as per the destination UDP address in the ethernet destination address of the ethernet packet header.

At the moment, no "de-registration" occurs when an smx instance is shut down. This will leave the `relay` program forwarding broadcasts to smx instances that no longer exist. If this turns out to be a problem in practice then we might have to add some sort of de-registration.

Appendix A.4: `mcc`

`mcc` is a shell script used for compilation of all smx source files (except the few that execute directly under SunOS), and to link all smx user programs (except `init`). `mcc` uses `gcc` to compile C programs, `as` to assemble SPARC assembler files, `ld` to link ELF executables, and `elf2smx` to convert ELF executables to smx executables. In general, `mcc` behaves much as a standard Unix compiler, taking very similar options. There are three options that are special to `mcc`:

- If the `-v` option is specified, then `mcc` simply echoes the commands that it would execute, rather than executing them.
- If the `-S stacksize` option is specified, then `stacksize` is saved and supplied to `elf2smx` (if `-S` is not specified, the stack size supplied to `elf2smx` is 16Kb). Note that the `stacksize` is now ignored by the smx memory manager, as it can extend dynamically the data and stack segments of user processes.

- If the -N option is specified, then the ELF executable (if one is created) is not deleted after elf2smx has produced the smx executable from it.

mcc deals with remaining options in the following way:

- The -c option prevents linking (that is, compile only).
- The -o option specifies the smx executable to create if linking takes place, the object file name if it doesn't.
- Any -l options are collected to supply to ld.
- If -E is specified, it is added to the "compiler options", and linking is suppressed.
- All other options are simply added to the "compiler options" to pass to gcc and as.

All other arguments on the command line should be names of .c, .s and object files. Each of these arguments is considered in turn, and handled in the following way:

- Each .c file is compiled by gcc. Some standard options are supplied by mcc. Certain options are supplied to all "compiles" (C and SPARC assembler). These consist of options to make the $MX_INCL directory the only standard header file directory, and defines to mimic the word size, long size and pointer size defines provided by the ack C compiler. In addition, certain options are supplied to all gcc compiles. These are -funsigned-char (some MINIX library functions assume that the char type is unsigned), -fno-common (elf2smx can't handle program sections produced from common areas) and -O2. Also passed to gcc are the list of "compiler options" present on the mcc command line.

  In the very rare circumstance that a standard SunOS header file is needed, the full pathname must be specified in the source file as the standard SunOS header directories are not searched.

- Each .s file is assembled by as. The standard options common to gcc and as (listed above) are supplied to as, as are the "compiler options" specified on the command line.

Once all compilations and assemblies have occurred, linking is performed unless it is inhibited by one of the command line options. The ld command is supplied with the name of the executable to create, the name of the .o file containing the standard smx C startup, the names of all of the .o files produced by the compilations and assemblies, the names of any other file names (usually .o and .a) from the command line, all -l options from the command line, and the name of the smx standard C library. Again, there are a standard set of ld options, including -dn to have the executable statically linked, an option specifying the name of the entry point in the startup file, an option specifying the map file to use, and an option specifying that $MX_LIB is the only default directory that should be searched when looking for a library.

If the smx executable blarg is being produced, then the ELF executable is produced in blarg.elf by ld. The elf2smx program is used to convert blarg.elf to blarg. The stack size is passed as an option. The file blarg.elf is then deleted unless -N was specified on the mcc command line. Finally all objects produced by compiles and assemblies are deleted unless the -c option was given.

Appendix A.5: `elf2smx`

The syntax for the `elf2smx` command is:

```
elf2smx [-d] [-S stacksize] ELF-executable smx-executable
```

`elf2smx` creates an smx-format executable from an ELF-format executable. The smx executable consists of an smx header, a text segment (containing text and read-only data) and a data segment. The text segment is click-aligned, and a multiple of the click size in length. If the `-d` option is specified then some debugging messages are printed.

If `-S` is specified, then `stacksize` specifies the MINIX "gap" (the amount of space to reserve for heap and stack when a program begins executing). The `stacksize` value is no longer relevant for user processes, as their data and stack segments can be expanded dynamically. The `stacksize` value is still relevant for `kernel`, `mm`, `fs`, `inet` and `init`. The `minix` bootstrap uses the `stacksize` value in allocating stack space for the programs it loads (in the case of `init` only, the stack space can be subsequently expanded).

Appendix B: Class use of smx

This document has mainly concentrated on installing smx for a single user. Where smx is to be used by students, each student could be given a complete smx distribution. This requires quite a lot of disk space, so it will usually be preferable to give them copies of only the parts of the system that they need to change.

For a student to boot their own copy of smx, they will need their own configuration file, and their own root file system (section 5.5.1 describes one way of setting up smx filesystems so that each student has a root file system, and `/usr/bin` and `/usr/man` are provided as shared file systems common to all students). Also, they must have the eight SunOS smx executables (`minix`, `mlogin`, and so on) on their search path. With `MX_INCL` and `MX_LIB` set appropriately, students can compile their own programs for smx, load them into their own smx file systems, and run them.

Where students need to modify smx source code, parts of the smx `src` hierarchy can be made available as needed. In the case of the `kernel` the students could be given the entire `$m/src/kernel` directory. Because changes in the size of the `kernel` executable affect the addresses at which `mm`, `fs` and `inet` are linked to run, students must be able to re-link these programs. Students could have complete copies of these directories, or could have in each case a directory that contains symbolic links to a single collection of `.o` files and the `Makefile`. Each student would also need their own `tools` directory (containing copies of `init` and `Makefile`, or symbolic links to them) where their own `image` file is created.

Nearly all smx commands can be copied as individual files (in the case of commands in `$m/src/commands/simple`) or directories (for other commands). Source files for some commands use relative pathnames to include header files from `kernel` and/or `mm` and/or `fs`. These include the smx user commands `de`, `df`, `fsck`, `fsck1`, `mkfs`, `ps` and `readfs`, and the smx commands `elf2smx`, `minix`, `mlogin` and `next_prog_addr` (this second group also uses relative pathnames to include standard smx header files from `$MX_INCL`).

In summary, if disk space is an issue then smx filesystems, sources, object files and binaries should be shared as much as possible. This can be done by copying only what students need to change, and setting up symbolic links to related directories or individual files. In the past, where students have been asked to modify the `kernel`, I have given them a script to setup symbolic links to all `kernel` files, and when a student wanted to modify a file they removed the symbolic link and replaced it with a copy of the file to be modified.

Smx (in its various forms) has been used in several student projects since 1992. In one course, students used smx without looking at its internals. This gave them the chance to play at being sys admins of there very own Unix-like system. They also used `de` to investigate the structure of smx filesystems. In a more advanced course, the older (MINIX 1.5.10-based) SunOS MINIX has been used in four assignments:

- In 1992, students had to add memory protection to smx. The current smx memory protection is a legacy of this assignment.
- In 1993, students had to carry out a "race condition audit" on the smx `kernel`. The scope of the audit was to describe how smx dealt with race conditions, and to describe any race conditions that still existed.
- In 1994, students were asked to document the smx kernel.
- In 1995, students were asked to use the interaction network monitor to record various "interesting" message passing patterns, and to use displays of these interaction networks as the basis for a report designed to explain various aspects of the operation of MINIX. Technical report 04/96 was a direct result of this assignment.

In 1996, students were asked to modify smx to support separate, expandable data and stack segments.

Finally, the smx lab exercises in `$m/doc` may be useful for introducing students to smx.

Appendix C: Overview of source code changes

The smx directory hierarchy that is installed under SunOS is a slightly modified version of the standard MINIX hierarchy. Some directories have been added, and some new files added to existing directories. Some things are not relevant to smx (such as PC-specific source files and directories) and these have by and large been removed from the hierarchy. Some directories contain MINIX files that while not currently used in smx, may well be added to smx in the future.

For the most part, smx-specific code in existing C source files is conditionally compiled. In a few files, however, changes have been sufficiently extensive that all code not related to smx has been removed (examples include `break.c`, `exec.c` and `forkexit.c` in `$m/src/mm`).

Below is an overview of new files, and files that have been substantially modified. All pathnames are relative to `$m`.

- `disks`. A new directory containing smx file systems.
- `doc`. A new directory for smx documentation.
- `include/sun`. A new directory for smx-specific include files.
- `man/man0`. New manual pages added for smx-specific commands: `sunload`, `sunread`, `sunwrite`.

- `src/Solaris`. A new directory containg programs and scripts that run under SunOS. The `IN` sub-directory contains the interaction network analysis programs.
- `src/commands/scripts/sunload.sh`. An smx script for installing standard smx binaries from SunOS.
- `src/commands/sun4`. A new directory containing smx-specific commands (the `sunread` command).
- `src/commands/tcsh`. A new directory containing sources for tcsh.
- `src/commands/uemacs`. A new directory containing sources for Microemacs.
- `src/etc`. Files modified somewhat for smx.
- `src/kernel`. Substantial changes. Many new source files; most of the remaining files were changed in some way.
- `src/lib/math/add_sun_libc_obs` and `src/lib/sun4/add_sun_libc_obs`. These scripts extract some files from the SunOS C library, and include them in the smx C library.
- `src/lib/smx_userprog.map`. Map file given to `ld` to control virtual address allocation when linking executables that are to become smx user programs.
- `src/lib/sun4`. SPARC assembler sources needed by the library (MINIX message passing functions, `setjmp/longjmp`, making system calls to SunOS), and the standard C startup files for smx user executables and system servers.
- `src/lib/sunsyscall`. SPARC assembler wrappers for all of the POSIX functions. Only `__sigreturn.s` is anything more than a simple wrapper.
- `src/mm`. Small changes to support separate data and stack segments that can be extended dynamically.
- `src/test/testload`. Script to load the binaries from the `src/test` directory into smx.
- `src/tools/.minix`. An smx configuration file.

In addition, all `Makefiles` have been changed to support compilation under SunOS.

References

[1] Paul Ashton. An interaction network monitor for Amoeba. Technical Report TR-COSC10/95, University of Canterbury, Department of Computer Science, October 1995. URL: `http://www.cosc.canterbury.ac.nz/~paul/tr-cosc.10.95.ps.gz`.

[2] Paul Ashton and John Penny. A tool for visualising the execution of interactions on a loosely-coupled distributed system. *Software—Practice and Experience*, 25(10):1117–1140, October 1995.

[3] Andrew S. Tanenbaum and Albert Woodhull. *Operating Systems: Design and Implementation.* Prentice-Hall, Englewood Cliffs, NJ, second edition, 1996.