Makbilan User's Guide

Dror Feitelson

Version of 21.8.94

Contents

1 Introduction

This document describes advanced features of the Makbilan system. It is written for whoever works on the Makbilan system: the system administrator responsible for the Unix host, whoever takes care of hardware configuration changes (adding or deleting PEs), and whoever writes additional run time libraries. In addition, there is a section about advanced programming features that are not fully described in the `ParC Programmer's Manual`.

Related Documents

- `ParC Programmers Manual`.
  This is the manual and tutorial for simple users.
- `MAXI Software Documentation (MSD)`.
  This is the documentation for the software written by us for the Makbilan.
- `Intel 80386 Programmer's Reference manual`. Intel order number 230985.
- `UNIX System V/386`. A whole set of guides and reference manuals.

2 Special Features

2.1 Exception Handling

The MAXI run-time library traps exception conditions and prints an error message when they occur. Alternatively, the programmer may specify a function that is to be invoked when an exception occurs. this is done with a variant of the `signal` system call.

A typical message about an exception condition looks like this:

```
Exception at PE #1
Divide error  at eip=1101
Register values:

     eax = 00000005  ecx = 00080000  edx = 00000000  ebx = 00080168
     esp = 0107df90  ebp = 0107df9c  esi = 00c85f6c  edi = 0107dfc4
  eflags = 00010206  cs  = 00000268  eip = 00001101

Program Terminated with exit code = -1
```

The first two lines are general 32-bit registers. The `eflags` register contains status flags, and the `cs` register points to the code segment. The most important register is `eip`: it is the program counter, and tells you the address of the instruction that caused the exception. To find more about these registers, see the `80386 Programmer's Reference Manual`, chapter 2.

To pinpoint the problem in a human-understandable manner, proceed as follows:

1. Record the value of `eip`, in this case 1101.
2. Recompile the program with the `-g` flag.
3. Disassemble the program into a temporary file, by performing

   `dis p.out >` *tmp_file*

4. Invoke your favorite editor on the temporary file, and search for the address given in the `eip` followed by a colon. In our example, search for the string "`1101:`".

what you will find is something like the following:

```
f_p_a_r1()
        10a4:  eb 64                  jmp    0x64 <110a>
[1][2]  10a6:  8b 45 08               movl   0x8(%ebp),%eax
```

```
        10a9:  0f af 45 10            imull  0x10(%ebp),%eax
        10ad:  03 45 0c               addl   0xc(%ebp),%eax
        10b0:  89 45 08               movl   %eax,0x8(%ebp)
[5]     10b3:  ff 75 24               pushl  0x24(%ebp)
        10b6:  e8 a5 18 00 00         call   0x18a5 <2960>
        10bb:  59                     popl   %ecx
[6]     10fa:  8b 45 08               movl   0x8(%ebp),%eax
        10fd:  99                     cltd
        10fe:  8b 4d 20               movl   0x20(%ebp),%ecx
        1101:  f7 39                  idivl  (%ecx),%eax
        1103:  8b 55 20               movl   0x20(%ebp),%edx
        1106:  89 02                  movl   %eax,(%edx)
[8]     1108:  c9                     leave
        1109:  c3                     ret
        110a:  55                     pushl  %ebp
        110b:  8b ec                  movl   %esp,%ebp
        110d:  eb 97                  jmp    0x97 <10a6>
        110f:  90                     nop
```

The name at the beginning of the code segment, `f_p_a_r1()` in this example, is the name of the function in which the exception occured. The number in square brackets, `[6]` in our example, is the line number in the function. The expression that appeared in this line was translated into a sequence of assembler instructions, and one of them (the one at address 1101) caused the exception.

If the name of the function does not appear familiar, this can have either of two reasons. One possibility is that the exception occured in one of the functions of the run-time library. This is rather unlikely, assuming that the libraries are properly debugged. The other possibility is that the function name is `f_a_p_r`*n*, as in the above example. These functions are generated by the compiler, and represent activities in various parallel constructs. To find what activity caused the problem, use the `+c` flag with `pcc` when compiling the program, and look at the *C* code that is produced.

Exceptions can be caught using the `signal` system call. This system call installs a user-defined handler that is invoked whenever the exception occurs. The handler should correct the error condition that caused the exception. Otherwise the program will enter an infinite loop, where the exception occurs, the handler is called, the instruction is retried, the exception occurs again, and so on.

The synopsis of installing a handler is

`signal(` *err_code*`,` *handler*`);`

The possible error codes are given in table 2-1. They are defined in an include file that may

code	*value*	*meaning*
DIV_ERR	0	divide by zero
DEBUG_EXC	1	debugger trap (breakpoint reached)
NMI	2	nonmaskable interrupt
BREAK_PNT	3	INT 3 instruction (software interrupt)
OVERFLW	4	arithmetic overflow condition
BOUND	5	bounds check failed
INVLD_OP	6	invalid opcode or operand
COPROC_NAVL	7	tried to use nonexsistent coprocessor
DOUBLE_FLT	8	double fault (two things went wrong)
COPROC_ADR_WRAP	9	coprocessor operand address wraparound
INVLD_TSS	10	invalid TSS
SEGMENT_FLT	11	segment not present
STACK_EXP	12	stack overflow, underflow, or missing
GEN_PROT	13	general protection fault
PAGE_FLT	14	page fault
	15	reserved by Intel
COPROC_ERR	16	coprocessor error

Table 2-1: *Error codes for the* `signal` *system call.*

be included in your program by the preprocessor directive

```
#include    "sig_386.h"
```

or

```
#include    "all.h"
```

For more information about the exceptions and how to handle them, see the `80386 Programmer's Reference manual`, chapter 9.

2.2 Random Number Generation

The Makbilan provides a parallel random number generator, with a distinct stream of random numbers on each PE. All the activities that execute on the same PE share the same stream. The lengths of these streams is 10^7, meaning that after 10^7 numbers the streams will start to overlap. It is not possible to set the seeds so that different streams will be generated in different executions.

To use the random number generator, include either of the following preprocessor directives in your program:

```
#include    "rand.h"
```

or

```
#include    "all.h"
```

The generator has to be initialized by the command

```
rand_init();
```

This would normally appear only once in the program, before any activities are spawned. If it is called again, the streams on all the PEs will be reset and start over again.

The random numbers are generated by the function

```
uni()
```

This returns a floating point number that is uniformly distributed in the range $[0, 1]$.

2.3 Monitoring Program Execution

The makbilan provids a resource for monitoring program execution. The monitoring done is a postmortem montoring, ie. during the execution certain events are written to a event log and after termination are recovered and displayed.

Backend The events recorded are:

1. creation of a task.
2. termination of a task.
3. scheduling of a task.
4. a task performs a wait.
5. a task performs sync.
6. a task performs exit.
7. a task performs spawn.
8. a user inserted event.

each event is recorded with:

1. the type of event.
2. the time it occured.

3. its task ID.
4. its parents task ID.
5. the PE it resides on.
6. other relavent info according to the event type(num to spawn..).

The events compiled gives us information about the execution of each task,its relationship to other tasks,information about each PE and an over- all picture of the parallelism of the program.

Frontend The display is made up of 5 main features:

1. the main graph that shows the branching of the program and the state (run,ready,wait..) of each task.
2. a graph that shows the parallelism (number of running tasks/PE #).
3. a popup histogram that shows the load on each PE.
4. a popup tree that shows the heirarchy of tasks.
5. popup user strings that correspond to user inserted data.

The monitoring of programs executed by the makbilan is done in a two-phase process:

Phase 1: compile the program with the mon library and run it with mldgo. The steps are:

1. write your program. Add the command:

   `user_event(`*user_string*`);`

   in your source where you want to trace execution,for instance:

   ```
   pparblock {
       user\_event(``fib(n-1)'');
       fib(n-1);
   }
   :
   {
       user\_event(``fib(n-2)'');
       fib(n-2);
   }
   epar
   ```

2. compile your program with the option

   `pcc +LIB mon` *your_program*

   and any other flags you would like

3. run it with mldgo

   `mldgo` *your_executable* `#`

   run the program with an additional PE. It is used for the monitors clock.

4. you should now have a file called efile in your current directory.

5. if you executed the program with ldgo instead of mldgo you will have a number of files named monitor1 to monitor# in your current directory, type the command

   `moncat #`

   and you will have a filed named efile.

Phase 2: send the output file to any Sun4 machine and run the monitoring program on it.The steps are:

1. login onto the machine.
2. Add the directory `/cc/pp/hardy/bin` to your path.
3. copy your "efile" by performing a `rcp` or `ftp` or any other command that copies files between machines.
4. to run the monitor perform:

   `mkmonitor [-in` *your_efile*`] [-l] [-vr]`

   The options are:

   -in the input file(default is efile).

   -l print internal data in 3 files:

   (a) act_tab - activity table, holds info for each task.
   (b) sp_tab - spawn table, holds info for each spawn.
   (c) event_tab - event table, holds info for each event.

   -vr reverse video.use if display can't be read.

5. If the program has terminated abnormaly do the following:

   (a) On the makbilan run `salvage`. An `efile` will be created.
   (b) Copy it to your current machine.
   (c) Run `mksalvage` on it (use the same flags as `mkmonitor`).

Display Instructions

When the display comes up you receive the main graph.
It consists of:

1. on the far left a timescale.

2. to the right of it the parallelism graph. It shows the number of running tasks every time quantum.

3. to the right of it is the execution graph. It shows how the tasks branch off and the states of the tasks.

4. on the bottom to the right are the 5 command buttons:

   (a) quit - obvious.

   (b) tree - opens a window that shows the heirarchy of tasks at a given time stamp.paragraph

   (c) histo - opens a window that shows the load on all PE's at a given time stamp.

   (d) parallelism - lets you mark a section of the timescale and then shows the parallelism in that section.

   (e) user strings - popups tags on the maingraph that enables the user to popup his inserted strings(inserted with `user_event(`*user_string);).*

5. to the right of the command buttons are the current time stamp and the average parallelism for the program.

6. to the right is the states window.

7. on the far right is the help window that shows how to use the mouse. This window changes according to the command button pushed.

8. the last features are the line and cursor.The line is positioned at the current time stamp.Moving the line can be done with the cursor or arrows.

3 Writing New Libraries

MAXI includes two types of run-time libraries: those that support *ParC*, and those that provide additional services. A program is normally linked to one chosen *ParC* library, and to all the service libraries. This section is mainly about writing new *ParC* libraries. New service libraries are discussed in the last subsection.

New versions of the *ParC* run-time library can be written and tested without interfering with the execution of programs with existing versions of the library. This is achieved by simply telling the compiler which version you want when you compile the program. The best way to create a new version and make it available for use is a two-phase process:

Phase 1: create a development environment in your home directory, develop the code, and test it.

Phase 2: install the new version, making it available to all users. This requires the password for the "makbilan" login.

3.1 Creating a Development Environment

First create a working directory for the new library. The following assumes that you are in this working directory.

The compiler usually works relative to the `/usr/makbilan` subtree. In order to work in your own directory, you need a personal version of the compiler. A special version of the compiler is provided for this purpose. As the compiler front end is a shell script, you can simply copy it to your directory by performing

```
cp /usr/makbilan/src/misc/pcc_dev .
```

Once you have your copy, edit the constant `WORKDIR` to refer to your working directory. You do not need to change anything else. Use `chmod` to make sure that your copy is executable.

New versions of the run-time library are always based on previous versions. Therefore you would probably start by copying the code of an existing library. To do so, it is enough to copy the makefile, as it has a special target that brings you the source files. The command

```
cp /usr/makbilan/src/policy/df/makefile .
```

will copy the makefile of the default library. If you want to base your work on another version, substitute the name of that version for `df` in the path.

To obtain copies of all the required source files, all you need to do is to perform

```
make get
```

Note that the files will be in a read-only mode. Use `chmod` to make those that you need to modify writable.

The `get` command will fail if someone else is actively changing the library at the same time. After you make sure that this is the correct thing to do, you can get the *old* version (before the current changes); otherwise, wait until the new changes are installed. In either case, edit the `GET_FLAG` variable in the makefile and make it the empty string. Then perform `make get` again.

3.2 Developing a New Version

The base version of the library which you are using is described in the `MAXI Software Documentation (MSD)`. Refer to that document for descriptions of the various files. The files you are most likely to change are:

global.h, global.c — definitions of global data structures.

local.h, local.c — definitions of local data structures.

ts_init.c — initialization of all data structures.

ts_spawn.c — the functions that deal with the creation of new tasks.

ts_sync.c — the functions that deal with synchronization.

parc.h — the user interface to the language implementation.

Do not change the file `main.c`. It is supposed to be the same for all versions of the library.

It is perfectly reasonable to add files that implement certain modules that are needed for your version of the library but were not needed before. The names of new files should start with "`ts_`", like those of existing files. The total length of the file name (with the extension) should not exceed 12 characters, because of an SCCS restriction. Add any new files to the `SOURCES` list in the makefile. Also note their dependencies on various `.h` files.

Don't forget to identify your library in the global variable `lib_ident`, which is defined in `global.c`.

Document your code in a style similar to that of existing versions (or better!). If you make changes to an existing module, note this fact in the "Revisions" field in the file header with your name and date. If you create new modules, note your authorship in the header with the date.

To test your code, compile the library by performing `make` with no arguments. This will leave the object files in your working directory, which is where your version of the compiler expects to find them. Using your compiler to compile *ParC* programs will link them with your library: `pcc_dev` does not know about any other versions. Naturally you will test your library with test programs that check the new functionality it was written for. However, don't forget to check other things as well, to ensure that no functionality was lost. It is a good idea to run all the programs in `/usr/makbilan/samples`.

3.3 Installing a New Version

To install a new version of the run-time library, you need to work in the "makbilan" login. Perform the following steps.

1. Decide on a name for the new version. This should be short and descriptive, normally between two and five letters. This is the name that will be used to denote your version in the `+LIB` argument to `pcc`.

2. Create the following new subdirectories using the new name:

   `/usr/makbilan/lib/`*new_name*
   `/usr/makbilan/include/`*new_name*
   `/usr/makbilan/src/policy/`*new_name*
   `/usr/makbilan/src/sccs/policy/`*new_name*

3. Determine exactly which files are unique to your new version. These are probably all the files that you created and edited. Note that this includes the makefile as well! Copy these files to `/usr/makbilan/src/policy/`*new_name*. Edit the SCCS fields in the headers of these files to read

   ```
   File name:    %M%
   Last mod :    %G%
   Version  :    %I%
   ```

4. Update the makefile:

   (a) Update the `TARGET_DIR` and `TARGET_INC` variables to point to the directories you created in step 2.

   (b) Edit the targets of the individual files in the makefile to reflect the fact that the files determined in step 3 should be taken from `/usr/makbilan/src/sccs/policy/`*new_name* and not from any other version. The format for your unique files should be

   `get $(GET_FLAG) $(SCCS)/policy/`*new_name*`/s.`*file_name*

The format for files that are taken from other versions has the version name instead of *new_name*, and does not have the `$(GET_FLAG)`.

(c) Edit the `delta` target in the makefile to allow modifications to your version to be saved in the SCCS system. For each of your unique files, there should be a line like

`delta $(SCCS)/policy/`*new_name*`/s.`*file_name*

Note that the changes relate only to the files that are unique to your new version of the library, not to all the files mentioned in the makefile.

5. Create an initial version under the SCCS version control system. To do so, `cd` to `/usr/makbilan/src/sccs/policy/`*new_name*. For each of your unique files (determined in step 3), perform the following command:

   `admin -i../../../policy/`*new_name*`/`*file_name* `s.`*file_name*

   Don't forget to do this also for the makefile.

6. `cd` back to `/usr/makbilan/src/policy/`*new_name* and remove all of the files there. This is OK because your files were saved under SCCS. If anything goes wrong, you still have them also in your working directory.

   Perform `get ../../sccs/policy/`*new_name*`/s.makefile` to retrieve the makefile.

7. Perform `make install`. This is the final check that everything is OK: it will get your files from SCCS, get the other files from the other versions of the library, compile your library, and then it will install it in the proper public directory. When users subsequently give your library name as an argument to `pcc`, it will know where to find your code.

8. Perform `make rm` to clean everything up.

It is imperative that apart from writing and installing the new version, you also provide appropriate documentation. This includes the documentation in the code, and a short entry in the `/usr/makbilan/src/policy/README` file. In addition, you should write a section about your library for the `MAXI Software Documentation (MSD)`. This should include the main ideas behind your work, the algorithms and data structures used, and a description of any new files that were created. Finally, to let users know about your version, it should be mentioned in the `ParC Programmer's Manual`.

3.4 Upgrading Existing Versions

To edit an existing library (e.g. to add a feature or correct a bug), you should work in that library's directory with the makbilan login. Do the following:

1. Edit the makefile so that `GET_FLAG = -e`.
2. Perform `make get`. This will get you a writable copy of all the unique files for this directory, and prevent any other users from doing the same.
3. Edit the code, and document the changes.
4. Perform `make install` to compile the library with the changes and install it where `pcc` will find it.
5. Perform `make delta` to make a new version of the code under the SCCS version control system.
6. Remove all the files that are left except the makefile, by performing `make rm`.
7. Edit the makefile again so that `GET_FLAG =`, i.e. it is the empty string.

3.5 Writing Service Libraries

Service libraries provide some service that is not tightly linked with *ParC*, and should be provided regardless of the version of the *ParC* library that is being used. Examples include exception handling and random number generation.

The main problem with service libraries is that they might require shared global and local variables. Such variables must be declared in the *ParC* library. To avoid the need to update all the versions of the *ParC* library when you add a service library, two interface files are used. The file `/usr/makbilan/src/service/glob_vars.c` is used to introduce all the global variables in all the service libraries. The file `/usr/makbilan/src/service/loc_vars.c` is used for local variables.

To write a new service library, work in the "makbilan" login. Create a new subdirectory under `/usr/makbilan/src/service`. If your library needs shared global variables, they should be declared in a file named `glob_`*name*`.c`. There should be nothing else in that file (except possible `#include` directives that are necessary for the declaration). The variables should also be declared as `extern` wherever they are used. Shared local variables should be treated likewise, and declared in a file called `loc_`*name*`.c`. The new library files should be stored under SCCS in a directory `/usr/makbilan/src/sccs/service/`*name*. Your makefile should make it possible to extract and restore files, as in other libraries.

To include your new variables in the *ParC* libraries, proceed as follows.

1. You have to change `/usr/makbilan/src/service/makefile`. Perform the following sequence of commands:

```
cd /usr/makbilan/src/service
rm makefile
get -e ../sccs/service/s.makefile
```

2. Edit the makefile so that `GET_FLAG = -e`, and then perform `make glob_vars.c` and `make loc_vars.c`. Of course if you have global variables *or* local variables, but not both, you only need one of these files.

3. If you have shared global variables, edit the file `glob_vars.c` and add the line

   ```
   #include   "glob_name.c"
   ```

4. If you have shared local variables, edit the file `loc_vars.c` and add the line

   ```
   #include   "loc_name.c"
   ```

5. Edit the makefile to reflect the existence of your files. Specifically, do the following:

   (a) Add the dependency of `glob_vars.c` on `glob_`*name*`.c`.

   (b) add a target for `glob_`*name*`.c`, with the following format:

   ```
   glob_name.c:
           get $(SCCS)/name/s.glob_name.c
   ```

   (c) If `glob_`*name*`.c` depends on any `.h` files, add these dependencies and also add targets for these files.

   (d) Do the same for `loc_vars.c` and `loc_`*name*`.c`

6. Use the opportunity to update the file `all.h`. This file includes the interface `.h` files for all the service libraries. If you have a `.h` file that should be included in the user program to use your library, add it to `all.h`. To do so, first get the file by performing `make all.h`.

7. Commit all the updated files to SCCS by performing `make delta`. To commit the makefile itself, perform

   ```
   delta ../sccs/service/s.makefile
   get ../sccs/service/s.makefile
   ```

   Don't forget to change the `GET_FLAG` variable in the makefile back to the empty string before you commit it.

8. Recompile all the *ParC* libraries with the cshell script `make_libs`.

Don't forget to document your work. Add an entry in `/usr/makbilan/src/service/README`, and a full description in the `MAXI Software Documentation (MSD)`. You should also advertise the use of your work in the `ParC Programmer's Manual` and the `Makbilan User's Guide`.

4 Hardware Reconfiguration

4.1 Changing the Number of Processors

In general there is no problem to change the number of PEs, but a few rules must be kept. To understand them, you must know about the numbering of slots in the Multibus-II boxes: the first slot, slot number 0, is in the middle. The rest are numbered circularly to the right. The rules about putting things in slots are:

- The bus controller, CSM/001, must be in slot 0.
- It is suggested to put the Unix host in slot 1.
- It is suggested that the first PE be in slot 3. This PE must have an EPROM for booting.
- all the other PEs can be in any free slot.

The Unix host and the first PE can in principle be in other slots. However, their location must be indicated in a special configuration file used to boot the system: the file `/etc/default/bootserver/config`. This file identifies various "hosts" and says what bootserver to use and what target file to load. Our `config` file says that `/unix` will be loaded on the host in slot 1, and `/usr/lib/rmk/bin/kernel` will be loaded on the host in slot 3 (using the second-stage loader `/usr/lib/rmk/second.rmx`). Thus if you put the Unix host or the first PE in another slot, you must update the `config` file.

The format of the `config` file is described in the `Intel System V/386 MULTIBUS Reference Manual`, section 3 (Intel order number 463328).

5 Unix Host Maintenance

This section would be redundant if we had decent documentation, but we don't. In any case, it serves to highlight the important aspects of system administration on the Unix host. You would probably also want to use the following documents:

- `UNIX System V/386 System Administrator's Guide`
- `UNIX System V/386 System Administrator's Reference Manual`
- `UNIX System V/386 User's Reference Manual`
- `UNIX System V/386 Programmer's Reference Manual`

Note that the manuals that we have are for release 3.0, while the installed system is release 3.2. Expect glitches.

5.1 System Installation

The system is installed from an Intel distribution tape. The sequence of things to do is described in the system logbook, page 4. A printout of the whole sequence from the console is filed in the documentation folder. the main steps to install a new system are:

1. Backup all user files, and make sure the backup is OK.
2. Boot from the tape. This causes the hard disk to be formatted, loosing all the files on it. All the system files are copied in from the tape (this takes 30–45 minutes).
3. Correct the `config` file to reflect the location of the Unix host, and reboot from the disk.
4. Install all the packages from the tape (each one takes about 5 minutes, and there are nine of them).

Alternatively, it is possible to upgrade an existing system. This causes the root file system `/` to be formatted and copied from the disk, but does not effect the `/usr` file system. After installing the upgraded system, you must reinstall the packages as well (although the installation procedure forgets to suggest this).

When the system is up, there are still a number of things to do:

1. Complete system administration:

   (a) Set time and timezone information. This is normally done only once, at setup (the first time you login). The system does not know about JST and JDT, so we cheat.

   (b) Open logins. This may be done at setup time or later, using `sysadm adduser`. It should be possible to create logins that do not require a password, e.g. for demo, but it is not clear how.

2. Restore any files that have been destroyed by the installation. See the section on backup below. If the users have new user ID numbers in the new system, use `chown` to restore correct file ownership.

3. Change system defaults:

   (a) Delete the line "`is:2:initdefault:`" from the files `/etc/inittab` and `/etc/conf/cf.d/init.base`. This causes the system to ask you what mode to enter when it is booted, rather than going automatically into multiuser mode. If using more than one parallel machine where some file systems are connected by NFS change the line to "`is:3:initdefault:`".

   (b) Make the system less strict about passwords by changing the parameters in `/etc/default/passwd`.

5.2 Booting the Unix System

In normal situations, there is no reason to reboot the system. However, glitches do happen. If the system is completely stuck, and you cannot login from another terminal and save it by killing some process, you need to reboot.

To reboot, do the following:

1. Press the `reset` button on the CSM/001 bus controller.

2. A sequence of `x`'s will begin to appear on the console. Interrupt them by typing `U` (shift `u`).

3. Get a menu, and choose option 2.

4. Type `t49` to reset the SCSI bus.

5. Type `b` to reboot the system.

If you typed the `U` too late, the system will enter its self diagnostics. You can let it do so, which takes some 10 minutes; it will then go to the boot phase automatically. Alternatively, you can interrupt the diagnostics with a `^C` (control-C). This will put you in the operator's interface. type `t49` and `b` to proceed immediately to the boot phase.

During the booting, you have to do the following if the line "`is:x:initdefault:`" is absent from `/etc/inittab`. :

4. When the system asks what run level to enter, you actually have only two options:

   `s` : enter single-user mode. Do this only to correct a horrible error in the system that prevents it from booting normally and letting you login as root.

   `2` : enter multiuser mode. This is the normal thing to do.

   `3` : enter NFS mode if there are file systems connected by NFS.

5. If the system says that there may be a dump memory image in the swap device, and asks if you want to save it, answer `n`.

5.3 System Administration

Most things you might want to do are handled by an interactive menu-driven system administration program, called `sysadm`. any subcommand can be chosen directly by giving it as an argument to `sysadm` when it is invoked. A full description is given in the `UNIX System V/386 System Administrator's Reference Manual`. The menu tree includes the following options:

1. diskmgmt: disk management menu

   1: checkfsys: check a removable medium file system for errors

   2: cpdisk: make exact copies of a removable medium

   3: erase: erase data from removable medium

   4: format: format new removable diskettes

   5: harddisk: hard disk management menu

      1) addbadblocks: enter bad sector information

      2) display: display hard disk partitioning

   6: makefsys: create a new file system on a removable medium

   7: mountfsys: mount a removable medium file system

   8: umountfsys: unmount a removable medium file system

2. filemgmt: file management menu

   1: backup: backup files from built-in disk to removable media

   2: bupsched: backup reminder scheduling menu

      1) schedcheck: schedule backup reminder checks

2) chedmsg: schedule backup reminder message

3: diskuse: display how much of the built-in disks are being used

4: fileage: list files older than a particular date

5: filesize: list the largest files in a particular directory

6: restore: restore files from "backup" & "store" media to built-in disk

7: store: store files and directories of files onto removable media

3. machinemgmt: machine management menu

1: whoson: print list of users currently logged onto the system

4. packagemgmt: package management menu

1: rfsmgmt: remote file sharing management menu

1) confgmgmt: RFS domain configuration management menu
2) resrcmgmt: RFS resource management menu
3) setuprfs: set up your machine to run RFS
4) startstop: start / stop RFS menu

2: uucpmgmt: basic networking utilities menu

1) devicemgmt: manage devices (list, add, delete)
2) pollmgmt: manage poll entries (list, add, delete)
3) portmgmt: manage I/O ports (list, modify)
4) systemmgmt: manage remote systems entries (list, add, delete, call)

5. softwaremgmt: software management menu

1: displaypkg: list packages already installed

2: installpkg: install new software package onto built-in disk

3: removepkg: remove previously installed package from built-in disk

6. syssetup: system setup menu

1: admpasswd: assign or change administrative passwords

2: datetime: set the date, time, time zone and daylight savings time

3: nodename: set the node name of this machine

4: setup: set up your machine the very first time

5: syspasswd: assign system passwords

7. ttymgmt: tty management menu

1: lineset: show tty line settings and hunt sequences

2: mklineset: create new tty line settings and hunt sequences

    3: modtty: show and optionally modify characteristics of tty lines

8. usermgmt: user management menu

    1: addgroup: add a group to the system

    2: adduser: add a user to the system

    3: delgroup: delete a group from the system

    4: deluser: delete a user from the system

    5: lsgroup: list groups in the system

    6: lsuser: list users in the system

    7: modadduser: modify defaults used by adduser

    8: modgroup: menu of commands to modify group attributes

        1) chgname: change name of a group on the system

    9: moduser: menu of commands to modify a user's login

        1) chgloginid: change a user's login ID

        2) chgpasswd: change a user's password

        3) chgshell: change a user's login shell

5.4 Backups

There are three different ways to store files on a tape and restore them later.

sysadm

As noted in the previous section, `sysadm` has a facility for backup and restore. Regrettably, this doesn't seem to work.

cpio

This command creates an archive of many files. To store, `cd` to the root of the subtree in the file system, put a tape in the drive, and perform

```
find .  -depth -print | cpio -o > /dev/rmt/tape
```

To get only selected subtrees rather than the whole tree, give their paths instead of "`.`". For example

```
find subdir sub/subsubdir -depth -print | cpio -o > /dev/rmt/tape
```

will copy all the files in these two subdirectories. To store the root file system without other mounted file systems such as `/usr`, `cd` to `/` and perform

```
find .  -depth -mount -print | cpio -o > /dev/rmt/tape
```

To get a list of what there is on the tape, put it in the drive and perform

```
cpio -it < /dev/rmt/tape
```

To restore from a tape, `cd` the directory from which the files were stored, and perform

```
cpio -idm "*" < /dev/rmt/tape
```

To restore only certain subdirectories (or single files) give patterns that define them, as in

```
cpio -idm "sub/*" "d1/d2/file" < /dev/rmt/tape
```

Note that the patterns should not start with a `/`.

A full description of `cpio` is given in the `UNIX System V/386 User's Reference Manual`.

tar

This is the conventional Unix tape archive utility.